PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

REMATCH: A NOVEL REGEX ENGINE FOR FINDING ALL MATCHES

NICOLÁS ANDRE VAN SINT JAN CAMPOS

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Advisor:
CRISTIAN RIVEROS

Santiago de Chile, October 2023

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA

REMATCH: A NOVEL REGEX ENGINE FOR FINDING ALL MATCHES

NICOLÁS ANDRE VAN SINT JAN CAMPOS

Members of the Committee:
CRISTIAN RIVEROS
DOMAGOJ VRGOC
GONZALO NAVARRO
JUAN REUTTER

Thesis submitted to the Office of Research and Graduate Studies
in partial fulfillment of the requirements for the degree of
Master of Science in Engineering

Santiago de Chile, October 2023

To my parents.

ACKNOWLEDGEMENTS

Write in a sober style your acknowledgements to those persons that contributed to the development and preparation of your thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The abstract must contain between 100 and 300 words. The abstract must be written in English and Spanish. In the case of doctoral theses, the layout of the abstract page is different, so please check the template provided by the OGRS.

Keywords: thesis template, document writing, **(Write here the keywords relevant and strictly related to the topic of the thesis)**.

RESUMEN

El resumen debe contener entre 100 y 300 palabras. El resumen debe ser escrito en inglés y español. En el caso de tesis de doctorado, el formato de la página del resumen es distinta, por favor verifique la plantilla entregada por la Dirección de Postrgrado.

Palabras Claves: plantilla de tesis, escritura de documentos, **(Colocar aquí las palabras claves relevantes y estríctamente relacionadas al tema de la tesis)**.

1. INTRODUCTION

Regular expressions, or RegEx, are one of the most used technologies for managing text data. The development of RegEx engines started in the early 70s (Thompson, 1968; Johnson, Porter, Ackley, & Ross, 1968), and they are now a common part of many complex information systems such as compilers, databases, or search engines. Moreover, modern RegEx engines are highly-optimized systems that are crucial for finding patterns in diverse areas like biology (Navarro & Raffinot, 2005), literature (Dorosz & Szczerbinska, 2009), or medicine (Flores, Figueroa, & Pezoa, 2021).

Given a regular expression and a document, the task of a RegEx engine is to find all occurrences, or *matches*, of the pattern in the document. For this, RegEx engines deploy the so-called *leftmost-longest* paradigm (IEEE & Group, 2018), meaning that they find the match which is the leftmost one, and from there they find the longest possible match. The process is then repeated starting from the rightmost position of the previous match[1]. For example, if we want to evaluate the RegEx `aa` over the document $a_0a_1a_2a_3$ (here the subindices are for referencing positions; the document consists of the letter a repeated four times), a typical RegEx engine will output the matches a_0a_1 and a_2a_3. In particular, RegEx engines will not output a_1a_2 since the first leftmost-longest match ends with a_1.

The leftmost-longest semantics is standard for RegEx engines, as it captures the majority of meaningful matches, although not all of them. However, in some scenarios adopting an "all-match semantics" is a valuable and desirable feature for the users. For instance, in DNA analysis we will often need to match patterns (called motifs) onto a DNA sequence, and these can overlap. The question of finding overlapping matches with RegEx is also recurrent in user discussions (*How to find overlapping matches with a regexp?*, 2013; *How can I match overlapping strings with regex?*, 2014; Srivastava, 2017). For information extraction, the all-match semantics leaves freedom to the user to extract all positions, called

[1]Although RegEx engines follow different matching rules, the leftmost-longest rule is at the core of most modern engines. For a detailed discussion see (Friedl, 2006).

spans, where there is relevant information in a document. Therefore the all-match semantics is a desirable feature for RegEx engines that, to the best of our knowledge, no engine supports natively.

To overcome the problem of finding all-matches, RegEx engines offer look-around operators, namely, operators that allows checking if a subexpression can be matched forward or backward from the current position, without advancing from the current position. For instance, by using look-around, we can modify the expression `aa` to `(?=(aa))` and find the missing match a_1a_2 over the above document. Despite this example, look-around operators cannot discover all matches for every RegEx expression. For instance, given the look-around definition, one cannot extract two matches that start at the same position (for concrete examples see Section 2 and Section **??**).

In terms of implementation, RegEx engines are usually divided into three categories: DFA-based, NFA-based, and recursive NFA-based (Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...), 2007). DFA is generally the fastest evaluation strategy, followed by (plain) NFA. In contrast, recursive NFA-based engines use backtracking, which is susceptible to well-documented performance issues, like regular expression denial of service attacks (ReDos) (Friedl, 2006), where the engine can exhibit exponential time performance (Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...), 2007). From the positive side, recursive NFA-based engines have the advantage of keeping track of the evaluation, which allows implementing operators like look-around and back-references. In summary, until now, the only way of finding all matches (in some cases) is by using look-around operators implemented by recursive NFA-based engines, which suffer from unfortunate performance issues.

To overcome these issues, this thesis presents REmatch, a RegEx engine supporting the all-match semantics, and its accompanying regular expression language REQL. Contrary to the status quo of RegEx evaluation, REmatch is based on a new evaluation strategy, inspired by the theory of enumeration algorithms (Segoufin, 2013), that allows finding all

the matches, and avoids the exponential behavior of recursive NFA evaluation. Moreover, REmatch performance is comparable to popular RegEx engines, while at the same time finding all the matches, thus obtaining the best of both worlds. Specific contributions of the thesis are as follows:

(i) We introduce the REQL query language, which extends classical RegEx with variables and the all-match semantics.

(ii) We present REmatch, a RegEx system whose architecture allows evaluating REQL using output-linear delay. For this, we develop a new evaluation method which extends the theoretical algorithm of (Florenzano, Riveros, Ugarte, Vansummeren, & Vrgoc, 2020) and incorporates new optimization techniques, allowing REmatch to compete with modern RegEx engines.

(iii) We develop a set of experiments to evaluate the effect of different optimizations on REmatch performance, and compare it to existing RegEx engines. Although REmatch uses a more general semantics, we show that its performance stacks well compared to other engines.

In Chapter 2 we introduce REQL. We then explain each module of the REmatch architecture (see Figure **??**). Chapter 3 presents the rewriting module, Chapter 4 the filtering module, and Chapter **??** the output module. Chapter **??** explains the evaluation algorithm of REmatch. Chapter **??** puts all components together and displays the experimental comparison with other engines. We conclude in Chapter **??** by discussing possible future work.

2. REQL: A REGEX QUERY LANGUAGE FOR IE

This section introduces REQL, a RegEx Query Language for information extraction, that we implement in REmatch. The language is an extension of the classical RegEx syntax (e.g. POSIX Basic Regular Expressions) familiar to most users. On the other hand, the semantics is inspired by the document spanner framework (Fagin, Kimelfeld, Reiss, & Vansummeren, 2015) that captures all appearances of a pattern in the document.

In the following, we present the formal syntax and semantics of REQL, and provide several examples of REQL queries.

2.1. Documents and spans

We follow the theoretical framework of documents and spans introduced in (Fagin et al., 2015). For us, a document d is simply a string over some finite alphabet (e.g. the ASCII charset, UTF-8, or a similar encoding scheme)[1]. We write $d = a_0 a_1 \dots a_{n-1}$ to denote a document of length $|d| = n$ where a_i is the i-symbol (note that the first symbol starts from 0)[2]. An example of a document is given in Figure 2.1. A *span* of a document d (also called a *match*) is a pair $s = [i, j\rangle$ of natural numbers i and j with $0 \leq i \leq j \leq |d|$. In that case, s is associated with the continuous region of the document d whose content is the substring of d from position i to position $j-1$. We denote this substring by $d(s)$ or $d(i, j)$. For instance, $d_1([0,4\rangle) = \texttt{that}$, since this is the content of the string d in positions 0 through 3. Notice that if $i = j$, then $d(s) = d(i, j) = \varepsilon$, the empty string. Given two spans $s_1 = [i_1, j_1\rangle$ and $s_2 = [i_2, j_2\rangle$ such that $j_1 = i_2$, we define their *concatenation* as $s_1 \cdot s_2 = [i_1, j_2\rangle$. The set of all spans of d is denoted by $\text{span}(d)$.

[1] Note that a multi-line document is simply a single string.

[2] In (Fagin et al., 2015), the first position is 1. We use 0 to be compliant with programming languages and RegEx engines which use 0 as the start position.

Figure 2.1. A sample document for illustration purposes.

2.2. Syntax

Syntactically, REQL is similar to standard regular expressions, apart from a special construct `!x{e}`, which states that a substring matching `e` should be stored into the variable name `x`. Formally, the syntax of REQL queries can be defined as follows:

$$\begin{aligned} \texttt{e} \;:=\; & \texttt{a} \mid \texttt{.} \mid \texttt{[w]} \mid \texttt{[\^{}w]} \mid \texttt{!x\{e\}} \mid \\ & \texttt{ee} \mid \texttt{e|e} \mid \texttt{e*} \mid \texttt{e+} \mid \texttt{e?} \mid \texttt{e\{n,m\}} \end{aligned}$$

Here, `a` is a character (e.g., ASCII charset or UTF-8), the dot symbol is a wildcard for any character, and `[w]` or `[^w]` are a char class or the negation of a char class, respectively, where `w` declares a set of characters. We use the standard notation of ranges of ASCII characters found in POSIX for declaring char classes (e.g. `[a-z]`, `[A-Z0-9apt]`, etc) and write set(`w`) to denote the set of characters represented by `w` (e.g. set(`a-z`) $= \{a, b, ..., z\}$). Moreover, `x` is a variable name where the character `!` is used to differentiate a variable name from a letter or string of the alphabet. This, along with the use of { and } for delimiting the captured subregex is the only special notation where we differ from POSIX. Finally, `n` and `m` are numbers such that $0 \leq \texttt{n} \leq \texttt{m}$. In the REmatch system, REQL also allow the usual regex abbreviations for character classes (e.g. `\d` for a digit, or `\w` for a word, etc), however, we do not include them in the formal definition in order to keep the presentation concise[3].

Example 2.1. *To give a preliminary example of how REQL works, assume that we would like to extract all the occurrences of the word "that" from a text document. This can be done in REQL as follows:*

$$e0 := !x\{that\}$$

[3]We remark that the start-of-file symbol (`^`) and end-of-file symbol (`$`) are currently not supported in REmatch. However, adding them is a straightforward exercise.

Intuitively, the query captures the positions of a substring `that` *into the variable x. This query also illustrates a key feature of our semantics (defined below): there can be overlapping matches. To make this more clear, consider the document d_1 in Figure 2.1. The query above will result in precisely three matches for the variable* `x`*, corresponding to the three occurrences of the substring* `that` *in the document we are processing. The first match will be in positions $[0,4\rangle$, the second in $[3,7\rangle$, and the last match in $[6,10\rangle$. We notice that the middle match $[3,7\rangle$ will not be captured by most regular expression tools, unless some sort of a look-around operator is used.*

The reader could notice that the above syntax is so general that one can define the capture of the same variable multiple times. For instance, a query like `!x{a!x{b}}` defines the capture of `x` twice. For this reason, REQL has some simple syntactic restrictions to use variables correctly. Let var(`e`) be the set of all variables names used in `e`. We say that a REQL query is *well-designed*[4] if every subquery `e` satisfies the following four conditions: (1) if `e` $=$ `!x{e1}`, then `x` $\notin$ var(`e1`), (2) if `e` $=$ `e1e2`, then var(`e1`) $\cap$ var(`e2`) $= \emptyset$; (3) if `e` $=$ `e1|e2`, then var(`e1`) $=$ var(`e2`); and (4) if `e` is equal to `e1*`, `e1+`, `e1?` or `e1{n,m}`, then var(`e1`) $= \emptyset$. One can easily check that queries `!x{a!x{b}}`, `!x{a}!x{b}`, `a|!x{b}`, or `(!x{a}b)*` are not well-designed. Instead, queries like `!x{a}!y{b}`, `!x{a}|!x{b}`, or `!x{a}(b)*` do satisfy all conditions and then are well-designed. Note that, as shown in (Fagin et al., 2015), the well-designed condition does not diminish the query language's expressive power. Then from now on, we will consider all the queries we evaluate to be well-designed.

2.3. Semantics

We define the matches extracted by REQL in terms of mappings. Formally, a *mapping* for a document d is a (partial) function μ from variables to spans of d. Intuitively, a mapping represents a single match that a REQL query makes on a document d. For instance, in our previous example, the query `e0` will produce three mappings as its output:

[4] In (Fagin et al., 2015), expressions satisfying these conditions are called *functional*.

$$
\begin{aligned}
[\![\texttt{e}]\!]_d &= \{\mu \mid (s,\mu) \in [\![\texttt{e}]\!]_d\} \\
[\![\texttt{a}]\!]_d &= \{(s,\emptyset) \mid s \in \text{span}(d) \text{ and } d(s) = \texttt{a}\} \\
[\![\texttt{.}]\!]_d &= \{([i,i+1\rangle,\emptyset) \mid 0 \leq i < |d|\} \\
[\![\texttt{[w]}]\!]_d &= \{(s,\emptyset) \mid s \in \text{span}(d) \text{ and } d(s) \in \text{set}(\texttt{w})\} \\
[\![\texttt{[\^{}w]}]\!]_d &= \{(s,\emptyset) \mid s \in \text{span}(d) \text{ and } d(s) \notin \text{set}(\texttt{w})\} \\
[\![\texttt{!x\{e\}}]\!]_d &= \{(s,\mu) \mid \exists (s,\mu') \in [\![\texttt{e}]\!]_d : \ d(s) \neq \varepsilon, \\
&\qquad \texttt{x} \notin \text{dom}(\mu') \text{ and } \mu = \mu' \cup [x \to s] \ \} \\
[\![\texttt{e1 e2}]\!]_d &= \{(s,\mu) \mid \exists (s_1,\mu_1) \in [\![\texttt{e1}]\!]_d . \exists (s_2,\mu_2) \in [\![\texttt{e2}]\!]_d : \\
&\qquad s = s_1 \cdot s_2 \text{ and } \mu = \mu_1 \cup \mu_2\} \\
[\![\texttt{e1|e2}]\!]_d &= [\![\texttt{e1}]\!]_d \cup [\![\texttt{e2}]\!]_d \\
[\![\texttt{e*}]\!]_d &= [\![\varepsilon]\!]_d \cup [\![\texttt{e}]\!]_d \cup [\![\texttt{e e}]\!]_d \cup [\![\texttt{e e e}]\!]_d \cup \cdots \\
[\![\texttt{e+}]\!]_d &= [\![\texttt{e (e*)}]\!]_d \\
[\![\texttt{e?}]\!]_d &= [\![\texttt{e}]\!]_d \cup \{([i,i\rangle,\emptyset) \mid 0 \leq i \leq |d|\} \\
[\![\texttt{e\{n,m\}}]\!]_d &= [\![\texttt{e} \overbrace{\cdots}^{\texttt{n}\text{-times}} \texttt{e (e?)} \overbrace{\cdots}^{(\texttt{m}-\texttt{n})\text{-times}} \texttt{(e?)}]\!]_d
\end{aligned}
$$

Table 2.1. The inductive semantics of REQL queries.

μ_1, with $\mu_1(x) = [0,4\rangle$, μ_2, with $\mu_2(x) = [3,7\rangle$, and μ_3, with $\mu_3(x) = [6,10\rangle$. We write $\text{dom}(\mu)$ to denote the domain of μ and $\mu_1 \cup \mu_2$ for the disjoint union of mappings whenever $\text{dom}(\mu_1) \cap \text{dom}(\mu_2) = \emptyset$. We also use the notation $[x \to s]$ to define the singleton mapping that only maps x to the span s (e.g., $\mu_1 = [x \to [0,4\rangle]$), and use $\emptyset$ for the trivial empty mapping (where the domain is the empty set).

With the formalism of mappings, we can give a concise declarative semantics for REQL, similarly as in (Maturana, Riveros, & Vrgoc, 2018). This is done in Table 2.1. The semantics is defined by structural induction on `e` and has two layers. The first layer, $[\![\texttt{e}]\!]_d$, defines the set of all pairs (s,μ) with $s \in \text{span}(d)$ and μ a mapping such that (1) `e` successfully matches the substring $d(s)$ and (2) μ results as a consequence of this successful match. For example, the REQL query a matches all substrings of input document d equal to a, but results in only the empty mapping. On the other hand, `!x{e}` matches all substrings that are matched by `e`, but assigns to x the *non-empty* span s that delimits the substring being matched, while preserving the previous variable assignments. Similarly,

in the case of concatenation `e1 e2` we join the mapping defined on the left with the one defined on the right. Notice that these mappings will not share any variables, given that the expression is assumed to be well-formed. The second layer, $[\![\texttt{e}]\!]_d$, then simply gives us the mappings that `e` defines when matching the entire document. Note that when `e` is an ordinary regular expression (i.e., no variables), then the empty mapping is output if the entire document matches `e`, and no mapping is output otherwise.

In the following, we provide several examples from English text analysis to grasp the power of REQL for information extraction and to see its differences concerning classical RegEx. The reader can test these examples and other REQL queries in our REmatch beta demo available on `www.rematch.cl`.

Example 2.2. *A typical task in language analysis is detecting words with particular roots, or more precisely lexemes, which are basic units of meaning. For example, one could be interested in words in the English language that start with the prefix 'a'. To extract all such words from a text, we can simply use the following REQL expression:*

$$e1 = \text{␣}!word\{\,[Aa]\backslash w+\}\,[\text{␣}.]$$

where ␣ denotes a single white space, and $\backslash w$ denotes the char class of words characters, as commonly used is Perl-compatible regular expressions. Note that in `[␣.]` the `.` denotes the dot symbol and not a wildcard. This is consistent with the classic RegEx syntax, since a wildcard symbol is useless for defining a char class.

In `e1` we are looking for a word staring with the letter 'a'. To assure we will capture the entire word, we preceded it by a space, and we require that after reading it we see either a space or a dot symbol[5]*. If we evaluate `e1` over document d_2 in Figure 2.2, we will get four mappings assigning the variable `word` to the spans $[4,7\rangle$, $[11,13\rangle$, $[14,21\rangle$, and $[22,31\rangle$, representing the words "ant", "an", "amazing", and "architect", respectively.*

[5]This example is for illustration purposes. The actual expression would allow arbitrary spacing and sentence punctuation, and allow matching the first word in the sentence.

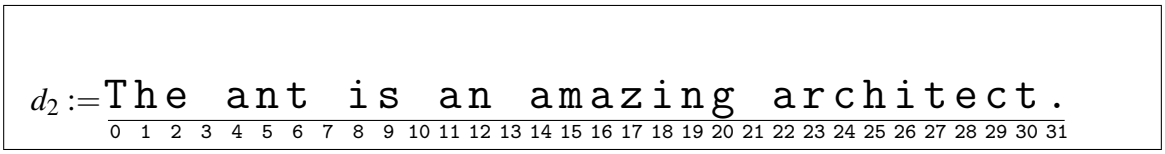


Figure 2.2. Document containing a sentence from the book "What is a man?" by Mark Twain.

In classical RegEx, round parentheses denote a capture group for extracting a substring. That is, `(R)` will extract what is matched by the RegEx `R`. We could therefore try to express `e1` from Example 2.2 by the expression: `␣([Aa]\w+)[␣.]` which replaces REQL's capture variables with a capture group. However, when evaluated over the document d_2 in Figure 2.2, one fewer output will be produced; namely: the span $[14,21\rangle$ corresponding to the word "amazing" will be missing. This is due to leftmost-longest match semantics deployed by classic RegEx engines, which will consume the white space following the word "an", therefore preventing the expression from matching "amazing". A typical workaround for this problem is the use of *look-ahead operators*, which allow to check whether a string is present starting from some position. A RegEx expression equivalent to `e1` would then be `␣([Aa]\w+)(?=[␣.])` which upon matching a word will look-ahead for a space or a dot, without advancing with the current match. In general, using look-ahead operators is somewhat cumbersome, and, as we show below, is not sufficient to capture all the matches in some cases. In contrast, REQL supports the all-match semantics by default: it returns a match for *every* span in the document where the specified pattern occurs.

Example 2.3. *Suppose that the user wants to process the English text into k-grams (i.e., k consecutive words in a text) that satisfy some particular pattern. Specifically, suppose this user wants to extract all* 2*-grams where each word begins with the letter 'a'. We can extract them by running the following REQL query:*

$$e2 := \text{␣}!w1\{[Aa]\backslash w+\}\text{␣}!w2\{[Aa]\backslash w+\}[\text{␣}.]$$

Note that `e2` *is the extension of* `e1` *where now we use two variable names, called* `w1` *and* `w2`*, for obtaining the substrings of the first and second words, respectively. For instance,*

if we run `e2` *over* d_2 *in Figure 2.2 we will get mappings:*

$$[\texttt{w1} \mapsto [11,13\rangle, \texttt{w2} \mapsto [14,21\rangle] \quad [\texttt{w1} \mapsto [14,21\rangle, \texttt{w2} \mapsto [22,31\rangle]$$

representing the 2*-grams "an amazing" and "amazing architect".*

Note that the previous query cannot be obtained by any RegEx engine without "look-arounds", given that 2-grams can overlap.

Example 2.4. *We end by showing another capacity of REQL for extracting contextual information, another feature not supported by RegEx. Suppose that, in addition to the* 2*-grams, the user wants to extract the sentence where the match happens. This additional information could be useful for understanding the context where these* 2*-grams are used. For this, we can modify our query* `e2` *as follows:*

$$\texttt{e3} := \backslash\texttt{.!sent\{ [\^{}.]*␣!w1\{[Aa]\backslash w+\}␣!w2\{[Aa]\backslash w+\}(␣[\^{}.]*)?\backslash.\ \}}$$

Here, the new variable `sent` *will store the information containing the sentence where the* 2*-gram occurs. The reader can check that if we evaluate* `e3` *over* d_2*, then we will obtain the mappings of Example 2.3 where each mapping will have in addition the variable* `sent` *maps to* $[0,31\rangle$*, which represents the whole sentence. Interestingly, this semantics context of a match cannot be extracted by RegEx, even if we use look-ahead operators. The main issue for look-ahead operators is that due to the leftmost-longest semantics, no two matches starting at the same position can be returned, which is an issue in our case.*

2.4. Comparison with RegEx and Document Spanners

As previously explained, we base REQL on RegEx syntax and the semantics of the Document Spanners framework. The purpose of reusing RegEx syntax is that users feel familiar with the query language and operators. However, as the previous examples show, the all-match semantics differs from the leftmost-longest semantics from classical RegEx.

Therefore, we introduce REQL as a new query language rather than presenting it as an extension of RegEx.

The class of regular expressions with extraction variables was first introduced by Fagin et al. (Fagin et al., 2015). Their framework is called Document Spanners, and it formalizes the process of extracting relations from text documents. We base REQL semantics on Document Spanners, although there are several differences. First, while Document Spanners are a theoretical tool for information extraction, REQL is a user-oriented query language with a programming syntax based on RegEx. Second, the semantics proposed in Document Spanners is anchored at the beginning and end of the document (like regular expressions in theory), where REQL is unanchored; namely, the query is evaluated anywhere in the document (similar to RegEx engines). Third, REQL semantics disallows capturing ε substrings (see $[\![\texttt{!x\{e\}}]\!]_d$ in Table 2.1), where Document Spanners allow this. We decided to remove ε-substring capturing since it is not very helpful for users, and its removal simplifies several optimization procedures in REmatch.

Comparing REmatch with classical RegEx engines, they both use classical operators and shortcuts, and match a substring from any position in the document, as opposed to the theoretical approaches to regular expressions. The main difference comes from capture variables and the all-match semantics. As we have seen, the combination of the two prevent simulating REQL's capture variables in RegEx using capture groups. Similarly, all-match semantics lies outside of the scope of RegEx, since matches starting at the same position cannot be simulated even using the look-around operators. As a minimal example for this, consider the document $d = \texttt{aaa}$, and the REQL expression `e = !{a*}`, which will produce six matches in this case, each one corresponding to a non-empty substring of d. On the other hand, using look-ahead will not help us to capture this in RegEx. The most obvious way would be to use an expression of the form `(?=(a*))`, however, at position 0, only the match $[\texttt{w1} \mapsto [0,3\rangle]$ will be produced, and, for example $[\texttt{w1} \mapsto [0,1\rangle]$ will be omitted, since they both start at the same position.

3. REWRITING MODULE

The first step for the evaluation of a REQL query is the compilation and rewriting into a logical plan, called a *logical VA*. This plan is essentially an automaton with variables that is equally expressive as a REQL query. Furthermore, logical VA is suitable for rewriting. Specifically, we perform an *offset* transformation over the logical VA that keeps the semantics of the query but improves the performance of the evaluation algorithm. Next we explain these two components.

3.1. Logical VA

A *logical variable-set automata* (logical VA) is a finite state automaton extended with captures variables. Formally, a logical VA $\mathscr{A}$ is a tuple (Q, δ, q_0, q_f), where Q is a finite set of states, q_0 and q_f are the initial and the final state, and δ is a transition relation consisting of letter transitions (q, C, q'), and variable transitions $(q, [x, q')$ or $(q, x\rangle, q')$, where $q, q' \in Q$, C is a char class (e.g, a letter `a`, `[w]` or `[^w]`) and x is a variable. The $[x$ and $x\rangle$ are special symbols to denote the opening or closing of a variable x. In the following, we refer to $[x$ and $x\rangle$ collectively as *variable markers*.

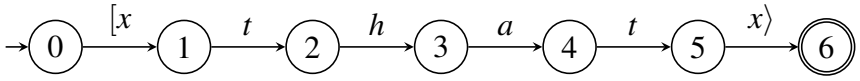
A configuration of a logical VA over a document d is a tuple (q, i) where $q \in Q$ is the current state and $i \in [0, |d|]$ is the current position in d. A run ρ over $d = a_0 a_1 \cdots a_{n-1}$ is a sequence:

$$\rho \;=\; (q_0, i_0) \xrightarrow{o_1} (q_1, i_1) \xrightarrow{o_2} \cdots \xrightarrow{o_m} (q_m, i_m)$$

where $(q_j, o_{j+1}, q_{j+1}) \in \delta$ and $i_0, \ldots, i_m$ is an increasing sequence, and $i_{j+1} = i_j + 1$ if o_{j+1} is a char class such that $a_{i_j} \in \text{set}(o_{j+1})$ (i.e. the automata moves one position in the document only when reading a letter) and $i_{j+1} = i_j$ otherwise. Furthermore, ρ must satisfy that variables are opened and closed in a correct manner, namely, each x is closed at most once and only if it is opened previously. We say that ρ is *accepting* if $q_m = q_f$ in which case we define the mapping μ^ρ that maps x to $[i_j, i_k\rangle$ if, and only if, $o_{i_j} = [x$ and $o_{i_k} = x\rangle$ in ρ. Notice that we do not require that $i_0 = 0$, nor $i_m = n$; namely, an accepting run

can start or end at any position in the document d, as long as it consumes a contiguous substring of d. Finally, the semantics of $\mathscr{A}$ over d, denoted by $[\![\mathscr{A}]\!]_d$ is defined as the set of all μ^ρ where ρ is an accepting run of $\mathscr{A}$ over d.

Example 3.1. *Consider the REQL query* `e0` *of Example 2.1. The following is a logica VA representing* `e0`*:*



In this figure, the states are $\{0,\ldots,5\}$*, where* 0 *and* 5 *are the initial and final state, respectively. The edges between states are transitions, where the first and last edges are variable transitions, i.e., they open and close x with the variable markers* $[x$ *and* $x\rangle$*, respectively, and the middle edges are letter transitions.*

Example 3.1 shows how to compile a REQL query into a logical VA. Using a Thomson-liked construction (Hopcroft & Ullman, 1979), we can covert every REQL query into a logical VA, giving us a logical plan for the query.

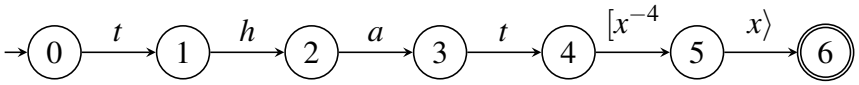
PROPOSITION 3.1. *For every REQL query* `e`*, one can build in linear time a logical VA* $\mathscr{A}$ *such that* $[\![\texttt{e}]\!]_d = [\![\mathscr{A}]\!]_d$ *for every document d.*

Note that logical VA is an extension of *variable-set automata* (VA) from (Fagin et al., 2015). The main difference between the two models is that logical VA uses char classes in its letter transitions whereas VA uses individual letters. Moreover, a logical VA can start a run at any position, whereas VA starts from the beginning of the document. Although both models are equally expressive, we use logical VAs as logical plans for compiling REQL formulas in practice.

3.2. Offsets (Optimization)

In some cases, opening a variable can be postponed in order not to store the information about runs that will not result in an output. To illustrate this, consider again the

expression `e0` and its logical VA of Example 3.1. Intuitively, our algorithm needs to store the position information for the opening of a variable x every time a `t` would be read. If the document we are reading has the text `thasty`, this run would then be extended for two more steps, although it will eventually be abandoned, and not result in any outputs. In cases such as these, we can actually postpone (offset) opening of the variable x by transforming the logical VA as follows: (i) first read the word `that`; (ii) now open a variable marker $[x$, but remember that it was actually opened four symbols before (i.e. it has an offset 4); (iii) proceed with the current run. When reconstructing the output, we will start reading four symbols before the position that is stored for $[x$. The transformation of the logical VA from Example 3.1 would look as follows:



The notation $[x^{-4}$ is used in order to signal that the variable x was actually opened four positions before it was recorded in the run.

Offsets are implemented in the rewriting module of REmatch after constructing the first logical VA from a REQL query. When the query contains quantifiers or alternations special care must be taken for offsetting the variables. More details are provided at (*REmatch Website*, n.d.).

4. FILTERING MODULE

In this section, we present the module in charge of filtering the input document and reducing the load of the main algorithm. The plan is to run a light process that scans the input and quickly finds sections of the document where there is at least one output. Formally, let $\mathscr{A}$ be the logical VA constructed from a REQL query, and let d be a document. For a mapping μ and a number ℓ, let $\mu_{+\ell}$ be a mapping shifted by ℓ, namely, for every variable x, $\mu_{+\ell}(x) = [i+\ell, j+\ell\rangle$ where $\mu(x) = [i, j\rangle$. A *segmentation* of d is a sequence $[i_1, j_1\rangle, \ldots, [i_k, j_k\rangle$ of spans of d such that $j_h < i_{h+1}$ for every $h < k$. We say that the segmentation is *valid* for $\mathscr{A}$ over d iff

$$[\![\mathscr{A}]\!]_d = \cup_{h=1}^{k} \{\mu_{+i_h} \mid \mu \in [\![\mathscr{A}]\!](d[i_h, j_h\rangle)\}$$

namely, if we can evaluate $\mathscr{A}$ over d by considering the segments $d[i_h, j_h\rangle$ of d and shifting the results. The task of the filtering module is to search for a good segmentation that is valid for $\mathscr{A}$ over d, and which can be computed quickly. Of course, the segmentation $[0, |d|\rangle$ is always valid, and we wish to refine it into smaller segments whenever possible.

Filtering the document into disjoint segments is not new when evaluating regular expressions. For example, RE2 (Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...), 2007) runs a deterministic automaton back and forth to find the starting and ending positions for the leftmost-longest match. Unfortunately, such approaches are unsuitable for our setting. Our approach follows the *split-correctness* framework proposed in (Doleschal, Kimelfeld, Martens, Nahshon, & Neven, 2019). The main goal of split-correctness is to find a REQL query $\mathsf{e}_{\mathscr{A}}$ with a single variable such that $[\![\mathsf{e}_{\mathscr{A}}]\!]_d$ is a valid segmentation of $\mathscr{A}$ over d. The filtering module in REmatch is inspired by split-correctness, but we improve the approach in two ways. First, REmatch does not restrict the filtering module on using single-variable expressions for finding a segmentation. Instead, for filtering, we consider any segmentation algorithm (i.e., not necessarily based on a regex) that, given $\mathscr{A}$ and d, finds a valid segmentation for $\mathscr{A}$ over d. Second, we look for a "cheap" segmentation algorithm, i.e., a process that runs

Algorithm 1 [LIGHT SEARCH] The segmentation algorithm for a logical VA $\mathscr{A} = (Q, \delta, q_0, q_f)$ over a document $d = a_0 \dots a_{n-1}$.

```
1:  procedure FILTERING(𝒜, a_0 ... a_{n-1})
2:      S ← ∅
3:      i ← 0, j ← 0
4:      for ℓ = 0 to n do
5:          (S, output, ends) ← next_δ(S, a_ℓ)
6:          if output then
7:              j ← ℓ + 1
8:          else if ends then
9:              if i < j then
10:                 Enumerate [i, j⟩
11:             i ← ℓ + 1
12:     if i < j then
13:         Enumerate [i, j⟩
```

faster than the evaluation algorithm. Indeed, using regex for filtering does not payoff if computing the segmentation takes longer than evaluating the target query itself.

4.1. Light search (Optimization)

In REmatch, the filtering module finds a segmentation by simulating the logical VA over the document, but only storing the starting and ending position where there is at least one output. For this we need the following extension of the transition relation. Let $\mathscr{A} = (Q, \delta, q_0, q_f)$ be a logical VA. We extend δ to a function δ^* that given a set $S \subseteq Q$ and a letter a, outputs all states that can be reached from S by using zero or more variable transitions and then a letter transition that satisfies a, namely, $p \in \delta^*(S, a)$ if there exists a sequence of transitions in δ of the form $q \xrightarrow{v_1} q_1 \xrightarrow{v_2} \cdots \xrightarrow{v_m} q_m \xrightarrow{C} p$ such that $q \in S$, $a \in \text{set}(C)$, and v_i is a variable marker (e.g., $[x$ or $x\rangle$) for every $i \leq m$. We also define $\delta^*(S, \varepsilon)$ such that $p \in \delta^*(S, \varepsilon)$ if, and only if, p can be reached from a state in S, by only using variable transitions.

Algorithm 1, also called LIGHT SEARCH, presents the filtering procedure for finding a segmentation of $\mathscr{A}$ over d. The algorithm simulates $\mathscr{A}$ over d by keeping a set of states $S \subseteq Q$ of active runs, namely, each $q \in S$ represents a run of $\mathscr{A}$ over a prefix of d that could

produce an output. The workhorse of Algorithm 1 is the function next_δ (see line 5). Given a set $S \subseteq Q$ and a letter a, the function $\text{next}_\delta(S,a)$ returns a triple $(S', \texttt{output}, \texttt{ends})$ where $S' \subseteq Q$ and $\texttt{output}, \texttt{ends}$ are boolean values. The first component S' is equal to $\delta^*(S,a) \cup \delta^*(\{q_0\},a)$. Intuitively, $\delta^*(S,a)$ are all states that one can reach from S by using some variable transitions and reading a. On the other hand, $\delta^*(\{q_0\},a)$ are the new states that one can reach by starting from q_0 and by reading a. Recall that a match can be made from any position in the document, so we start a fresh run by using $\delta^*(\{q_0\},a)$. The second component `output` is true iff $q_f \in \delta^*(S',\varepsilon)$, namely, `output` is true when there is a run that reaches the final state. Finally, `ends` is true iff $\delta^*(S,a) = \emptyset$, which tells whether the runs in S ends with the new letter. When implementing Algorithm 1 in REmatch, we cache the output $\text{next}_\delta(S,a)$, in order to compute it at most once for every pair (S,a).

We have all the ingredients to explain how Algorithm 1 works. The algorithm keeps a set of active states S, and two pointers i and j. As we already mentioned, S contains all active states when reading the document. Instead, the pair (i,j) stores the last span $[i,j\rangle$ (called active span) where there is an output, namely, $[\![\mathscr{A}]\!](d[i,j\rangle) \neq \emptyset$. We assume here that, if $j \leq i$, then the algorithm has not found a segment yet (i.e., from the previous segment that was output). In lines 2-3, we start by setting $S = \emptyset$ (i.e., no active runs) and $(i,j) = (0,0)$ (i.e., no active span). Then we iterate sequentially over each letter a_ℓ. For each letter, we compute $\text{next}_\delta(S,a_\ell)$ returning as output the triple $(S', \texttt{output}, \texttt{ends})$ where S' is the new set of active states (line 5). If `output` is true, an active state can reach q_f and the segment $[i,\ell+1\rangle$ contains an output. Then by setting $j = \ell+1$ we update the new active span (lines 6-7). Instead, if there is no output and `ends` is true, then all active states of the previous iteration end with the new letter a_ℓ, and we can start a new active span by setting $i = \ell$ (line 11). However, if $i < j$, then we cannot extend more the active span represented by (i,j), we can safely return the active span $[i,j\rangle$ and continue (lines 9-10). Finally, after the document ends (lines 12-13) we check if there is an active span that was not output (i.e., $i < j$), and return it if this is the case.

Example 4.1. *In the following figure, we display the execution of Algorithm 1 for the logical VA of query* `e0` *(see Example 3.1) over the document* `thathatsthat`*. For each letter, we show below the value of variables* ℓ*, S,* `output`*,* `ends`*, i, and j after finishing each iteration.*

	`t`	`h`	`a`	`t`	`h`	`a`	`t`	`s`	`t`	`h`	`a`	`t`
$\ell =$	0	1	2	3	4	5	6	7	8	9	10	11
$S = \emptyset$	{2}	{3}	{4}	{2,5}	{3}	{4}	{2,5}	$\emptyset$	{2}	{3}	{4}	{2,5}
`output` =	F	F	F	T	F	F	T	F	F	F	F	T
`ends` =	T	F	F	F	F	F	F	T	T	F	F	F
$i = 0$	0	0	0	0	0	0	0	7	8	8	8	8
$j = 0$	0	0	0	4	4	4	7	7	7	7	7	12

By following the run of the algorithm, we can check that it outputs the segmentation $[0,7\rangle$ *and* $[8,12\rangle$ *corresponding to the substrings* `thathat` *and* `that`*, respectively.*

Algorithm 1 maintains the invariant that, after reading a_ℓ, i is a position before any of the current active runs started and, if $i < j$, then j is the latest position such that $[\![\mathscr{A}]\!](d[i,j\rangle) \neq \emptyset$. Indeed, these invariant is enough to prove that the algorithm is correct.

Theorem 4.1. *Given a logical VA* $\mathscr{A}$ *and a document d, Algorithm 1 outputs a sequence of spans* $[i_1,j_1\rangle,\ldots,[i_k,j_k\rangle$ *that is a valid segmentation for* $\mathscr{A}$ *over d.*

Note that the load of computing Algorithm 1 is low: we need one pass over the document and for each letter we need to perform a small number of simple operations (i.e., apply the function next_δ and check at most two if-statements). Given that we can cache the output $\text{next}_\delta(S,a)$ for each new pair (S,a), the cost per new letter is low when the caching of the function next_δ stabilizes. This is probably the main reason why filtering runs faster than performing the main evaluation algorithm (see Section **??** for further discussion).

5. CONCLUSIONS

Nothing to say. Be happy.

REFERENCES

Cox, R. (2007). https://swtch.com/~rsc/regexp/regexp1.html.

Doleschal, J., Kimelfeld, B., Martens, W., Nahshon, Y., & Neven, F. (2019). Split-correctness in information extraction. In *Pods 2019* (pp. 149–163). ACM.

Dorosz, K., & Szczerbinska, A. (2009). Enhancing regular expressions for polish text processing. *Comput. Sci.*, *10*, 19–36. Retrieved from https://doi.org/10.7494/csci.2009.10.3.19 doi: 10.7494/csci.2009.10.3.19

Fagin, R., Kimelfeld, B., Reiss, F., & Vansummeren, S. (2015). Document spanners: A formal approach to information extraction. *J. ACM*, *62*(2), 12:1–12:51.

Florenzano, F., Riveros, C., Ugarte, M., Vansummeren, S., & Vrgoc, D. (2020). Efficient enumeration algorithms for regular document spanners. *ACM TODS*, *45*(1), 3:1–3:42.

Flores, C. A., Figueroa, R. L., & Pezoa, J. E. (2021). Active learning for biomedical text classification based on automatically generated regular expressions. *IEEE Access*, *9*, 38767–38777. Retrieved from https://doi.org/10.1109/ACCESS.2021.3064000 doi: 10.1109/ACCESS.2021.3064000

Friedl, J. E. (2006). *Mastering regular expressions*. " O'Reilly Media, Inc.".

Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to automata theory, languages and computation*. Addison-Wesley.

How can i match overlapping strings with regex? (2014). Retrieved 2023-05-04, from https://stackoverflow.com/questions/20833295/how-can-i-match-overlapping-strings-with-regex/33903830

How to find overlapping matches with a regexp? (2013). Retrieved 2023-05-04, from https://stackoverflow.com/questions/11430863/how-to-find-overlapping-matches-with-a-regexp

IEEE, & Group, T. O. (2018). *Open group standard vol. 1: Base definitions, issue 7, chapter 9*. https://pubs.opengroup.org/onlinepubs/9699919799/

basedefs/V1_chap09.html.

Johnson, W. L., Porter, J. H., Ackley, S. I., & Ross, D. T. (1968, dec). Automatic generation of efficient lexical processors using finite state techniques. *Commun. ACM*, *11*(12), 805–813. Retrieved from https://doi.org/10.1145/364175.364185 doi: 10.1145/364175.364185

Maturana, F., Riveros, C., & Vrgoc, D. (2018). Document spanners for extracting incomplete information: Expressiveness and complexity. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI symposium on principles of database systems, houston, tx, usa, june 10-15, 2018* (pp. 125–136). ACM.

Navarro, G., & Raffinot, M. (2005). New techniques for regular expression searching. *Algorithmica*, *41*(2), 89–116. Retrieved from https://doi.org/10.1007/s00453-004-1120-3 doi: 10.1007/s00453-004-1120-3

Rematch website. (n.d.). https://github.com/REmatchChile/REmatch-paper. (Accessed on 2023-02-01)

Segoufin, L. (2013). Enumerating with constant delay the answers to a query. In *Joint 2013 EDBT/ICDT conferences, ICDT '13 proceedings, genoa, italy, march 18-22, 2013* (pp. 10–20).

Srivastava, A. (2017). *Java 9 regular expressions*. Packt Publishing. (https://www.oreilly.com/library/view/java-9-regular/9781787288706/)

Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Communications of the ACM*, *11*(6), 419–422.

APPENDIX

A. FIRST APPENDIX

We can write equations here too:

$$\int_0^\infty e^{-x^2} dx \tag{A.1}$$

And more...

B. AN INTERESTING SHORT STORY

Let us enjoy reading this story of Hunting With The Lion.

It was a dry summer. The animals in the forest were beginning to find it difficult to get food.

A bear, a wolf and a jackal thought it would be better to join hands with a lion and do the hunting. They approached lion and he too agreed. The four of them went off hunting.

The hunting party came across a buffalo. The fox and wolf chased the buffalo. The bear intercepted the buffalo. The lion killed him.

The fox made shares out of the buffalo. When they were about to take their shares the lion roared and said, ”Well friends, the first share is mine for my leadership. The second share is mine for, it is I who killed. The third share is also mine for I need it for my cubs. Anyone who needs a share can take the fourth. But before that you will have to win me.”

All the three left the place without a single word.

MORAL : **If you are might, you are right**.