REmatch: a novel regex engine for finding all matches

Cristian Riveros
PUC & IMFD

Nicolás Van Sint Jan
PUC & IMFD

Domagoj Vrgoč
PUC & IMFD

Abstract

In this paper, we present the REmatch system for information extraction. REmatch is based on a recently proposed enumeration algorithm for evaluating regular expressions with capture variables supporting the all-match semantics. It tells a story of what it takes to make a theoretically optimal algorithm work in practice. As we show here, a naive implementation of the original algorithm would have a hard time dealing with realistic workloads. We thus develop a new algorithm and a series of optimizations that make REmatch as fast or faster than many popular RegEx engines while at the same time being able to return all the outputs: a task that most other engines tend to struggle with.

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 [22, 37], 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 [28], literature [10], or medicine [13].

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 [21], 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 [17, 18, 35]. For information extraction, the all-match semantics leaves freedom to the user to extract all positions, called 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 7).

In terms of implementation, RegEx engines are usually divided into three categories: DFA-based, NFA-based, and recursive NFA-based [8]. 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) [14], where the engine can exhibit exponential time performance [8]. 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.

Contribution. To overcome these issues, this paper 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 [34], 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 paper are as follows:

(1) We introduce the REQL query language, which extends classical RegEx with variables and the all-match semantics.
(2) 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 [12] and incorporates new optimization techniques, allowing REmatch to compete with modern RegEx engines.
(3) 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.

Outline. In Section 2 we introduce REQL. We then explain each module of the REmatch architecture (see Figure 5). Section 3 presents the rewriting module, Section 4 the filtering module, and Section 5 the output module. Section 6 explains the evaluation algorithm of REmatch. Section 7 puts all components together and displays the experimental comparison with other engines. We conclude in Section 8 by discussing possible future work.

[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 [14].

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 [11] 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.

Documents and spans. We follow the theoretical framework of documents and spans introduced in [11]. 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)[2]. 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)[3]. An example of a document is given in Figure 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, $d_1([0, 4\rangle) =$ `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 $\text{span}(d)$.

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:

```
e  :=  a | . | [w] | [^w] | !x{e} |
       ee | e|e | e* | e+ | e? | e{n,m}
```

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 $0 \leq$ `n` $\leq$ `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[4].

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}`

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 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.*

$d_1 :=$
```
t h a t h a t h a t
0 1 2 3 4 5 6 7 8 9
```

Figure 1: A sample document for illustration purposes.

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*[5] if every subquery `e` satisfies the following four conditions: (1) if `e = !x{e1}`, then `x` $\notin$ var(`e1`), (2) if `e = e1 e2`, 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 [11], 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.

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: 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 $[x \rightarrow s]$ to define the singleton mapping that only maps x to the span s (e.g., $\mu_1 = [x \rightarrow [0, 4\rangle]$), 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 [26]. This is done in Table 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

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

[3] In [11], the first position is 1. We use 0 to be compliant with programming languages and RegEx engines which use 0 as the start position.

[4] 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.

[5] In [11], expressions satisfying these conditions are called *functional*.

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

Table 1: The inductive semantics of REQL queries.

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, $[\![\mathtt{e}]\!]_d$, 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 = ␣!word{[Aa]\w+}[␣.]`

where ␣ *denotes a single white space, and* `\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*[6]*. If we evaluate* `e1` *over document* d_2 *in Figure 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.*

[6]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.

$d_2 :=$ `The ant is an amazing architect.`

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Figure 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, 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 := ␣!w1{[Aa]\w+}␣!w2{[Aa]\w+}[␣.]`

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 we will get mappings:*

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

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:*

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

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.*

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. [11]. 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 $[\![$`!x{e}`$]\!]_d$ in Table 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 = `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 [`w1` $\mapsto [0, 3\rangle$] will be produced, and, for example [`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.

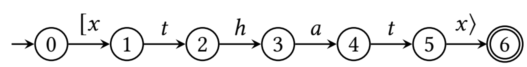
Logical VA. A *logical variable-set automata* (logical VA) is a finite state automaton extended with captures variables. Formally, a logical VA $\mathcal{A}$ is a tuple (Q, δ, q_0, q_f), where Q is a finite set of states, q_0 and q_f are the initial and the final state, and δ is a transition relation 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 $\mathcal{A}$ over d, denoted by $[\![\mathcal{A}]\!]_d$ is defined as the set of all μ^ρ where ρ is an accepting run of $\mathcal{A}$ 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 [16], we can covert every REQL query into a logical VA, giving us a logical plan for the query.

Proposition 3.2. *For every REQL query* `e`*, one can build in linear time a logical VA* $\mathcal{A}$ *such that* $[\![\texttt{e}]\!]_d = [\![\mathcal{A}]\!]_d$ *for every document* d*.*

Note that logical VA is an extension of *variable-set automata* (VA) from [11]. 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.

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:

$$\to (0) \xrightarrow{t} (1) \xrightarrow{h} (2) \xrightarrow{a} (3) \xrightarrow{t} (4) \xrightarrow{[x^{-4}} (5) \xrightarrow{x\rangle} (\!(6)\!)$$

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 [1].

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 $\mathcal{A}$ 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 $\mathcal{A}$ over d iff

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

namely, if we can evaluate $\mathcal{A}$ 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 $\mathcal{A}$ 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 [8] 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 [9]. The main goal of split-correctness is to find a REQL query $\mathsf{e}_{\mathcal{A}}$ with a single variable such that $[\![\mathsf{e}_{\mathcal{A}}]\!]_d$ is a valid segmentation of $\mathcal{A}$ 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 $\mathcal{A}$ and d, finds a valid segmentation for $\mathcal{A}$ over d. Second, we look for a "cheap" segmentation algorithm, i.e., a process that runs 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.

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

```
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⟩
```

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 $\mathcal{A} = (Q, \delta, q_0, q_f)$ 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 $q \in S$, $a \in \mathrm{set}(C)$, and v_i is a variable marker (e.g., $[x$ or $x\rangle$) for every $i \leq m$. We also define $\delta^*(S, \epsilon)$ such that $p \in \delta^*(S, \epsilon)$ 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 $\mathcal{A}$ over d. The algorithm simulates $\mathcal{A}$ over d by keeping a set of states $S \subseteq Q$ of active runs, namely, each $q \in S$ represents a run of $\mathcal{A}$ 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 $\mathrm{next}_\delta(S, a)$ returns a triple $(S', \mathtt{output}, \mathtt{ends})$ where $S' \subseteq Q$ and `output`, `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 $q_f \in \delta^*(S', \epsilon)$, 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 $\mathrm{next}_\delta(S, a)$, 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, $[\![\mathcal{A}]\!](d[i, j\rangle) \neq \emptyset$. 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 $[\![\mathcal{A}]\!](d[i, j\rangle) \neq \emptyset$. Indeed, these invariant is enough to prove that the algorithm is correct.

Theorem 4.2. *Given a logical VA $\mathcal{A}$ 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 $\mathcal{A}$ 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 7 for further discussion).

5 Output module

The output of a REQL query `e` over a document d can be prohibitively large, namely, of size $O(|d|^{2|e|})$, since for the all-match semantics we could even ask for all substrings being matched to all the variables. Since such queries are, at least in principle, expressible in REQL, we need to be able to handle them in REmatch. For this, we deploy the notion of enumeration algorithms and output delay, which measure the efficiency of an algorithm with respect to both its input and its output.

More formally, we use the framework of *enumeration algorithms*, which received a lot of attention in the database community [2, 5, 12, 19, 20, 24, 33, 34, 39]. In enumeration algorithms, we are required to produce the (finite) output set $O = \{o_1, \ldots, o_k\}$, in any order, and without repetitions. Such algorithms operate in two phases:

(1) The pre-processing phase builds a data structure which allows enumerating the results;
(2) The enumeration phase retrieves the outputs from the data structure.

In the case of REQL queries, the desired result is the set of all the output mappings. We will say that an enumeration algorithm works with *output-linear delay*, if the time to print out the i-th output o_i, measured as the time needed from printing the $(i-1)$-st output o_{i-1} to finishing the output o_i, is proportional to the length of o_i, and independent of the size of the document, the query, or the size of the output set O. The algorithm is also required to terminate immediately after outputting the final output. If these conditions are met, the time needed to enumerate O is $O(|O|)$, hence output-linear.

Next we describe the REmatch module for managing the system memory and the data structure supporting output-linear delay.

The data structure. In general, the pre-processing phase builds a data structure encoding all the mappings that are to be enumerated. We next explain which operations this structure needs to support in order to encode outputs of a REQL query. For this, consider again the document d_1 from Figure 1 and the REQL query:

```
e4 := !x{th}.*!y{hat}
```

that extracts the substring `th` in the variable x followed by the substring `hat` in the variable y. One output here is μ_1, with $\mu_1(x) = [0, 2\rangle$, and $\mu_1(y) = [4, 7\rangle$. Notice that for each output mapping, we need to define when a variable is opened, and when it is closed, in order to define the span it captures. In REmatch we will represent a mapping as an *output sequence* of pairs (S, i), where S is a set of variable markers (e.g., $S = \{x\rangle, [y\}$), denoting when a span associated with the variable x starts, or finishes, respectively. Therefore, the mapping μ_1 is represented by the output sequence[7]:

$$([x, 0), (x\rangle, 2), ([y, 4), (y\rangle, 7).$$

In essence, REmatch will create a data structure encoding this information for each mapping. Since many mappings will share information (for instance, μ_2, with $\mu_2(x) = [0, 2\rangle$, and $\mu_2(y) = [7, 9\rangle$ has the x-part identical to μ_1), we can exploit this fact to create a succinct representation of all the outputs.

The data structure we will use to represent the set of all output sequences in $[\![\texttt{e}]\!]_d$, for a REQL query `e` and a document d is called *enumerable compact set*, or ECS for short, and was first introduced in [3, 27]. The ECS data structure can be thought as a directed acyclic graph (DAG) with three types of nodes `n`:

(i) a (unique) terminal node, denoted `emptyNode` (or $\perp$ for short), which has no children and tells us that we reached the end of an output;
(ii) content nodes, which store a pair (S, i) from an output sequence and have a single child `n'`; and
(iii) union nodes, which have two children $\texttt{n}_1$ and $\texttt{n}_2$.

[7]Strictly speaking, we should write $(\{[x\}, 0), (\{x\rangle\}, 2), \ldots$. For the sake of simplification, we omit the set brackets whenever it is possible.

More importantly, every node n defines a set of mappings $[\![\mathsf{n}]\!]$, represented as a set of output sequences. Specifically, the emptyNode represents the set $[\![\mathsf{emptyNode}]\!] = \{\epsilon\}$, the output node n with the child n′ represents the set $[\![\mathsf{n}']\!] \cdot \{(S, i)\}$, and the union node with children n_1 and n_2 the set $[\![\mathsf{n}_1]\!] \cup [\![\mathsf{n}_2]\!]$.

Example 5.1. *Consider again the query* e4*, and document d_1 from Figure 1. The list of all mappings in $[\![\mathsf{e4}]\!]_d$, as represented by their output sequences is:*

$$\mu_1: \quad ([x,0\rangle, (x\rangle, 2), ([y,4), (y\rangle, 7)$$
$$\mu_2: \quad ([x,0), (x\rangle, 2), ([y,7), (y\rangle, 10)$$
$$\mu_3: \quad ([x,3), (x\rangle, 5), ([y,7), (y\rangle, 10)$$

The ECS for this set of output sequences is given in Figure 3. Here the rightmost union node represents the output sequences of μ_1, μ_2, and μ_3. As we can see, following paths from this node to the $\perp$ node represent the three output sequences. The shared output for the variable y in μ_2 and μ_3 is represented only once.

Node manager (Optimization). The key step when evaluating a REQL query over a document is building the ECS which encodes all the outputs. Naively building the ECS would require allocating the memory for each node being added, which is highly inefficient. Therefore, REmatch includes a *node manager* module, denoted NM, which allocates memory in bulk, and acts as a garbage collector for the ECS. The NM module essentially creates a memory pool for storing the nodes in the ECS. Each time NM fills the pool, it allocates the memory for another pool double the original size. This strategy allows memory allocation to occur infrequently and in big chunks, thus preventing fragmentation and multiple pointer dereferencing. In addition, the NM module acts as a lazy garbage collector by keeping a pointer count for each node in the memory pool. Once the pointer counter hits zero, it moves the node to a pool of nodes that can be deleted. This will happen when a node is not required anymore by the evaluation algorithm, detecting that it will not be reached from any newly added node, thus allowing to delete the node and other nodes whose pointer count hits zero by this removal. As stated previously, the memory locations of deleted nodes are not liberated but are instead overwritten by newly created nodes, in which case these locations are removed from the pool of nodes to be deleted. We use the command NM.discard(n) when we wish to signal that a node is to be discarded.

The node manager NM is also in charge of implementing the following three operations over nodes:

- NM.emptyNode, which creates the terminal node;
- NM.extend(n, (S, i)), which creates a content node which contains (S, i), and links this newly created node to n; and
- NM.union($\mathsf{n}_1, \mathsf{n}_2$), which takes nodes n_1 and n_2, and creates a new union node, representing the union of both outputs.

NM implements these three operations, taking constant time for each operation. Moreover, NM module supports enumerating outputs from any given node n, which we denote by NM.enumerate(n). More importantly, this enumeration takes output-linear delay, and one can do it at any point without further preprocessing. This last fact is the guarantee for REmatch to retrieve all outputs $[\![\mathsf{e}]\!]_d$ with output-linear delay. We refer to [27] for the implementation details of such operations and enumeration procedure.

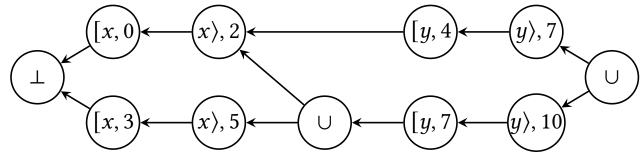


Figure 3: The ECS representing three outputs.

Early output (Optimization). Note again that NM allows enumerating the outputs from a node at any point without further preprocessing. This fact is crucial for the optimization that we call *early output enumeration.* Specifically, when evaluating a logical VA over a document, we can often detect whether we reach a final state before the entire document is read completely (for details, see Section 6). In essence, this means that we can provide certain outputs to the user at this point before continuing to construct the ECS in its entirety. The benefits are twofold: (i) the outputs are delivered to the user as soon as possible, similarly as pipelined evaluation is done in databases; and (ii) once we enumerate these outputs, we can delete the unused nodes, thus saving storage space. This optimization is highly effective in decreasing the memory usage and decreasing the time to deliver the first output (see Section 7).

6 Evaluation module

To evaluate REQL, we compile logical VA into so-called *extended VA*, a "physical" automata model closer to the evaluation algorithm than to the query language. We use this model for the REQL evaluation algorithm used in REmatch, which makes one pass over the document and produces the resulting mappings with output-linear delay. This evaluation algorithm refines and improves the theoretical algorithm presented in [12] for evaluating extended VA by taking care of the memory usage, simplifying the algorithm, and applying various optimizations presented in the previous sections.

Extended VA. An *extended variable-set automaton* (eVA) [11] is a finite-state automaton extended with capture variables in a way analogous to logical VA. The difference is that reading and outputting variable markers can be done in a single transition[8] Formally, an eVA $\mathcal{E}$ is a tuple (Q, q_0, F, δ), where Q is a finite set of *states*; $q_0 \in Q$ is the initial state; $F \subseteq Q$ is the set of final states; and δ is a *transition relation* consisting of transitions of the form (q, a, S, q') where $q, q' \in Q$, $a \in \Sigma \cup \{\blacksquare\}$, and S is a set of variable markers (e.g., $S = \{[x, y\rangle\}$). The meaning behind the transition (q, a, S, q') is that when the automaton is in the state q and reads the i-th letter $a_i = a$, then it switches to state q' and outputs (S, i), where S is the set of variables that are opened and closed *before* reading the i-th letter. In particular, when $S = \emptyset$, the automaton changes the state and nothing is output.

The fact that a transition can open or close multiple variables at the same time allows us to handle nested variables (e.g. `!x{!y{a}}`) in a single automaton transition. Furthermore, the $\blacksquare$ symbol is used as a special input to denote the end of a document (i.e., an End Of File, or EOF). This will be useful in the run of an eVA for outputting spans $[i, j\rangle$ over a document of size n, where i and j range from 0 to n (i.e., we need $n + 1$ possible positions).

[8]The name eVA is also used by [12] for an automaton model outputting sets of markers. However, the eVA of [12] require letter and variable transitions to be separate, while we process them simultaneously. We use the same name for consistency.

Next we define how an eVA produces outputs. A *run* ρ of $\mathcal{E}$ over a document $d = a_0 \cdots a_{n-1}$ is a sequence of the form:

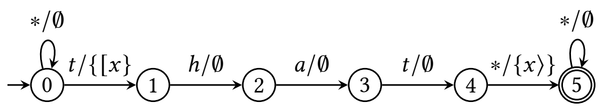
$$\rho \;=\; q_0 \xrightarrow{b_0/S_0} q_1 \xrightarrow{b_1/S_1} q_2 \xrightarrow{b_2/S_2} \;\ldots\; q_n \xrightarrow{b_n/S_n} q_{n+1} \qquad (1)$$

where $b_0 b_1 \ldots b_n = a_0 \ldots a_{n-1}\blacksquare$, S_j is a set of variable markers, and $(q_j, b_j, S_j, q_{j+1}) \in \delta$, for $0 \leq j \leq n$. Also, we say that ρ is *accepting* if $q_{n+1} \in F$. A run ρ like (1) naturally defines an output sequence as $\text{out}(\rho) = \text{out}(S_0, 0) \cdot \ldots \cdot \text{out}(S_n, n)$ such that $\text{out}(S_i, i) = (S_i, i)$ if $S_i \neq \emptyset$, and ϵ, otherwise. Finally, the semantics of $\mathcal{E}$ over d, denoted by $[\![\mathcal{E}]\!]^{\text{seq}}_d$, is defined as the set of all output sequences $\text{out}(\rho)$ where ρ is an accepting run of $\mathcal{E}$ over d.

eVA has several differences compared to logical VA: (1) an eVA starts from the beginning of the document, (2) produces output sequences instead of mappings, and (3) the transitions are fired by symbols (i.e., not by char classes). These relaxations plus read-captures transitions will considerably simplify the evaluation algorithm. Indeed, we can show that if we start from a logical VA $\mathcal{A}$, then we can construct an equivalent eVA $\mathcal{E}$ in linear time.

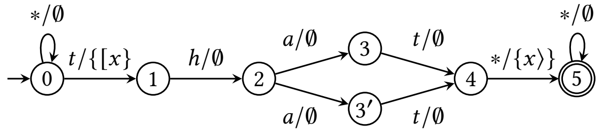
Proposition 6.1. *For every logical VA $\mathcal{A}$ compiled from a REQL query* `e`*, one can build, in linear time, an* eVA *$\mathcal{E}$ such that $\mathcal{E}$ evaluated over a document d produces precisely the output sequences representing the mappings in $[\![\mathcal{A}]\!]_d = [\![\texttt{e}]\!]_d$.*

Example 6.2. *To illustrate Proposition 6.1, recall query* `e0` *and its equivalent logical VA $\mathcal{A}_0$ from Example 3.1. The following extended VA $\mathcal{E}_0$ is equivalent to $\mathcal{A}_0$:*



Here we draw an edge $q \xrightarrow{a/S} q'$ to denote a transition from q to q' that output S when reading a letter a, and use $*$ to denote any letter. Intuitively, we have moved transitions with variable markers "forward" if we compare $\mathcal{E}_0$ with $\mathcal{A}_0$. For instance, $\mathcal{A}_0$ had transitions $0 \xrightarrow{[x} 1 \xrightarrow{t} 2$, but instead $\mathcal{E}_0$ has a direct transition $0 \xrightarrow{t/[x} 1$. Since an extended VA needs to consume the entire document, we add self-loops in the initial and the final state; for instance $4 \xrightarrow{t/x\rangle} 5$. Also note that the transition $0 \xrightarrow{*/\emptyset} 0$ closes the variable `x` *before the character (EOF or any other character) is consumed.*

Determinization. There is a critical issue with using eVA as our guide for evaluating REQL: several runs can produce the same output sequence. To illustrate this issue in its simplest form, assume the following extended VA $\mathcal{E}_0'$, which is a mild modification of $\mathcal{E}_0$:



One can check that $\mathcal{E}_0'$ is equivalent to $\mathcal{E}_0$, namely, $[\![\mathcal{E}_0']\!]^{\text{seq}}(d) = [\![\mathcal{E}_0]\!]^{\text{seq}}(d)$ for every document d. However, for every output sequence of $[\![\mathcal{E}_0']\!]^{\text{seq}}(d)$, two different runs are witnessing it: one crossing the state 3 and another crossing the state 3′. Then, if we guide an evaluation algorithm of eVA with runs, for $\mathcal{E}_0'$ we will enumerate each output sequence twice, although the user expects to extract each output without duplicates. Similar behavior can happen for eVAs coming from REQL queries with quantifiers or alternations.

To remove duplicate runs from eVA, we use a subclass called *deterministic* eVA. We say that an eVA $\mathcal{E}$ is *deterministic* if its transition relation δ satisfies that for every two transitions $(q, a_1, S_1, q_1') \in \delta$ and $(q, a_2, S_2, q_2') \in \delta$, if $(a_1, S_1) = (a_2, S_2)$, then $q_1' = q_2'$. In other words, given a state q, the next state is determined by the pair (a, S). The reader can check that $\mathcal{E}_0$ is deterministic but $\mathcal{E}_0'$ is not.

A deterministic eVA ensures that for every document and output sequence, there is at most one accepting run. This correspondence is crucial for our evaluation algorithm, given that we can simulate runs and construct the output, without worrying about duplicates. Fortunately, we can always "determinize" a non-deterministic eVA $\mathcal{E} = (Q, q_0, F, \delta)$ by using a subset construction, similar to the standard determinization procedure for NFAs. More precisely, we define $\mathcal{E}^{\text{det}} = (Q^{\text{det}}, q_0^{\text{det}}, F^{\text{det}}, \delta^{\text{det}})$ such that: $Q^{\text{det}} = 2^Q$, $q_0^{\text{det}} = \{q_0\}$, $F^{\text{det}} = \{X \mid X \cap F \neq \emptyset\}$, and:

$$\delta^{\text{det}} = \{(X, a, S, X') \mid \forall q' \in X'.\, \exists q \in X.\, (q, a, S, q') \in \delta\}.$$

One can check that $\mathcal{E}$ and $\mathcal{E}^{\text{det}}$ are equivalent (i.e., they define the same output for every document), and that $\mathcal{E}^{\text{det}}$ is deterministic.

An inconvenience of $\mathcal{E}^{\text{det}}$ is that its size is exponential in $|\mathcal{E}|$. Fortunately, for the evaluation algorithm it is not necessary to construct entire $\mathcal{E}^{\text{det}}$. Instead, we can start from the initial set $\{q_0\}$ and traverse only the transitions and states needed by the next letter. Each time that we need a new state or transitions of $\mathcal{E}^{\text{det}}$, we use $\mathcal{E}$ to build them and cache it in main memory for future access. This is the purpose of the *determinization module* of REmatch, called `DET`. Specifically, `DET` has a method `next` such that, given a state $X \in Q^{\text{det}}$ and a letter a, `DET.next`(X, a) computes a list ℓ with all pairs (S, X') such that $(X, a, S, X') \in \delta^{\text{det}}$. Instead, if ℓ was already computed before (and cached), `DET.next`(X, a) outputs ℓ immediately. Then by using `DET` we only need to compute each state and transition once. Moreover, the number of states accessed by `DET` depends on $\mathcal{E}$ and the input document d. In practice, this size is small and at most three or four times the size of $\mathcal{E}$.

Algorithm's variables. In Algorithm 2 we present the main algorithm for REmatch. This algorithm evaluates an eVA $\mathcal{E}$ over the input document d, enumerating all output sequences $[\![\mathcal{E}]\!]^{\text{seq}}(d)$. Two main components used by the algorithm are the node manager `NM`, introduced in Section 5, and the determinization module `DET`, introduced above. In addition, we use *states-sets* constructed by `DET`, *set-lists* that store states-sets, and *nodes* created and operated by `NM`. We introduced nodes `n` in Section 5.

The states-sets, denoted by X in the algorithm, are built and cached by the determinization module for representing a set $X \subseteq Q$. Each states-set X has two variables: X.`n` and X.`phase`. The former can store a node that represents the current outputs of runs that reached X. Instead, the latter is an integer that encodes the current phase number: if X.`phase` $= i$ then the i-th iteration was the last one that reached X. In practice, `phase` will help to know whether it is the first time we reached X during some iteration. Further, each states-set has a method X.`isFinal` that outputs `TRUE` if, and only if, X is a final set (i.e., $X \cap F \neq \emptyset$). In the implementation, this is a flag that the determinization module sets when creating X.

Algorithm 2 Evaluation of an extended variable-set automaton $\mathcal{E} = (Q, \delta, q_0, F)$ over the document $b_0 \dots b_n$ where $b_0 \dots b_{n-1}$ is the original document and $b_n = \blacksquare$ is an EOF symbol. DET and NM are the determinization module and node manager, respectively.

```
1:  procedure Evaluate(ℰ, b_0 ... b_n)
2:      DET.initialize(ℰ)
3:      InitializeLists
4:      for i = 0 to n do
5:          for all X ∈ setslist do
6:              ℓ ← DET.next(X, b_i)
7:              if ℓ ≠ empty then
8:                  UpdateSets(X, ℓ, i)
9:              else
10:                 NM.garbage(X.n)
11:         setslist.swap(setslist′)
12:         setslist′.clear
13:     Enumerate

14: procedure InitializeLists
15:     setslist.clear
16:     setslist′.clear
17:     X_0 ← DET.initialStateSet
18:     X_0.phase ← −1
19:     X_0.n ← NM.emptyNode
20:     setslist.add(X_0)
21:
22: procedure Enumerate
23:     for all X ∈ setslist do
24:         if X.isFinal then
25:             NM.enumerate(X.n)

26: procedure UpdateSets(X, ℓ, i)
27:     for all (S, X′) ∈ ℓ do
28:         n′ ← X.n
29:         if S ≠ ∅ then
30:             n′ ← NM.extend(n′, (S, i))
31:         if X′.phase < i then
32:             X′.phase ← i
33:             setslist′.add(X′)
34:             X′.n ← n′
35:         else
36:             X′.n ← NM.union(X′.n, n′)
```

We will also use set-lists, denoted by setslist in the algorithm, which are linked-list of states-sets. For this data structure, we assume a method setslist.clear to empty the list, setslist.add(X) to add X at the end, and setslist.swap(setslist′) to swap the content between two lists. To iterate over each element X in the list, we conveniently write "**for all** $X \in$ setslist". We assume any straightforward implementation of set-list that takes constant time for each call to these methods. Finally, during the algorithm, we use two set-lists variables, setslist, and setslist′. We assume that setslist, setslist′, NM, and DET can be globally accessed by all methods.

Main algorithm. The main method of Algorithm 2 is Evaluate, which receives as input an eVA $\mathcal{E} = (Q, \delta, q_0, F)$ and the document $b_0 \dots b_n$. Recall that $b_0 \dots b_{n-1}$ is the original document and $b_n = \blacksquare$ is the EOF symbol. The algorithm starts by initializing the determinization module DET with $\mathcal{E}$ at line 2. The `initialize` method is for storing $\mathcal{E}$ inside DET and using it later during the determinization. Then we do the initialization of setslist and setslist′ by calling InitializeLists (line 3). This method clears both lists (lines 15-16) and adds the state-set X_0 to setslist (line 20). This state-set represents $\{q_0\}$, namely, the determinization initial set. For this, DET provides a method DET.initialStateSet that outputs $X_0 = \{q_0\}$ (line 17). Then we initiate X_0.phase with -1 (line 18) and fill X_0.n with the empty output node (line 19). Intuitively, no iterations have accessed X_0 yet, and the $\mathcal{E}^{\text{det}}$-run at X_0 only has the empty output.

The evaluation algorithm processes the document $b_0 \dots b_n$ symbol by symbol, from $i = 0$ to n (line 4). During the i-th iteration, the setslist keeps all state-sets X that can be reached by runs reading $b_0 \dots b_{i-1}$, and X.n a node storing all outputs of these runs. Instead, setslist′ will contain the new state-sets after reading the new letter b_i. Intuitively, to build setslist′ from setslist we fire all state-sets X in setslist one-by-one (line 5) by calling the method DET.next(X, b_i) with symbol b_i (line 6). As explained previously, this method gives a list ℓ of pairs (S, X'), where there is a transition $(X, b_i, S, X') \in \delta^{\text{det}}$. Then, if $(S, X') \in \ell$, we must extend the output sequences in X.n with (S, i), and store them in X'. We do this updating by calling the method UpdateSets(X, ℓ, i) (line 8), discussed below. If the list ℓ is empty (e.g., DET detects that by reading b_i from X, there is no way to continue), then we call the node manager NM and mark X.n for the garbage collection (line 10). Notice that, by using the determinization manager, we can detect the right place to reuse memory in the algorithm. Finally, when we end firing all state-sets in setslist, we swap the two lists and clean setslist′ to start the iteration again (lines 11-12).

Update method. The workhorse of the evaluation algorithm is UpdateSets(X, ℓ, i), which is in charge of updating each state-set X' by extending the outputs in X.n with (S, i), for each $(S, X') \in \ell$. For this purpose, we iterate over each $(S, X') \in \ell$, create a copy n′ of X.n, and extend all output sequences in n′ with (S, i) if $S \neq \emptyset$ (lines 28-30). Next, we need to update X'.n depending on whether it is the first time or not that we reach X'. For this, we use variable X'.phase: if X'.phase $< i$, then we update X'.phase to i, add X' to setslist′, and set X'.n equal to n′ (lines 31-34). Otherwise, it is not the first time that we reach X', and we need to union the output sequences in X'.n with the ones in n′ (lines 35-36).

For the correctness of UpdateSets, we cannot reach any state-set X in two consecutive iterations (i.e., both in the $(i-1)$- and i-th iteration for some i). Otherwise, we could use X.n by reading and updating it simultaneously, possibly erasing its content. For this reason, during the rewriting module we duplicate the logical VA, alternating between even and odd positions. This construction forces that, at this stage, we cannot reach any state-set X in two consecutive iterations as needed.

UpdateSets passes and updates the outputs from the $(i-1)$- to the i-th layer, which is the heaviest part of the algorithm. Therefore, it is crucial to perform UpdateSets as efficiently as possible. For this, we use linked-lists and phase variables, which allows us to check in a single instruction whether X' is already in setslist′ or not. These tricks are standard in the implementation of RegEx engines [8]. The novelty is then to incorporate these optimizations in a new evaluation algorithm for computing all matches.

Example 6.3. *Figure 4 displays a graphic of how Algorithm 2 would run with eVA $\mathcal{A}_0$ from Example 6.2 and document d_0. Below document d_0, we draw as columns the setslists that are computed after reading the corresponding symbol. For this example, each state-set is a single state, and then each number in a setslist corresponds to a state in $\mathcal{A}_0$. Below each state-set X, we draw in grey the node X.n from the ECS that the algorithm construct.*

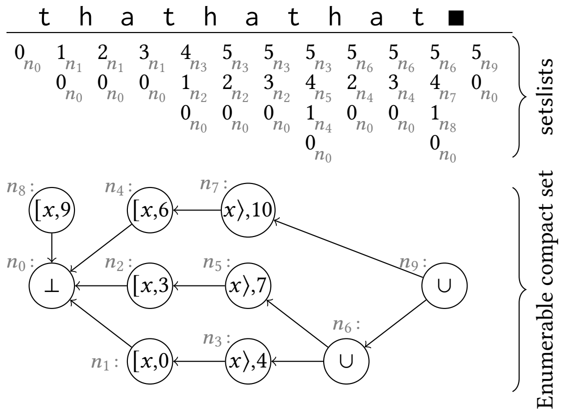


Figure 4: An example of how Algorithm 2 works.

Next index (Optimization). A significant step for the evaluation algorithm is the call to the DET.next function (line 6). Given a state-set X and a symbol b_i, the first time that the algorithm calls DET.next(X, b_i) the DET module must compute a list with all pairs (S, X') such that $(X, a, S, X') \in \delta^{\text{det}}$, and save it in its cache. Then, for later calls to DET.next(X, b_i), the DET module must quickly find this list in the cache. The evaluation uses the DET's cache multiple times, making it one of the heaviest parts of the computation. To decrease the load of the algorithm, we add an index to each state set X, which quickly allows finding the next state-set given a b_i. Currently, REmatch only supports ASCII documents; therefore, we implement this index as an array with 128 entries. This array on each state-set allows us to quickly find the next state-set, considerably improving the performance of the evaluation algorithm. Here, for the next index, we are sacrificing space versus time. Of course, a more compact next index (e.g., for non-ASCII documents) could save space during the evaluation. We leave this for future work.

Enumeration. When we get to the end of the document, setslist contains all state-set X that can be reached by runs of $\mathcal{E}^{\text{det}}$ when reading $b_0 \dots b_n$. In particular, X.n has the node representing all outputs of these runs. Then we call Enumerate (line 13), which iterates over all $X \in$ setslist that are final, and enumerates the output sequences in X.n by calling the enumerate method of the node manager (lines 23-25). As described in Section 5, we can perform an early output enumeration whenever we reach a final state at the i-th iteration. For ease of presentation, we code Algorithm 2 with the enumeration procedure after reading the whole document.

7 Experimental evaluation

In this section we provide an experimental evaluation showing the viability of REmatch in practice. For this we designed our experiments around several real-world text corpora, and:

(1) Set internal baselines by showing how various optimizations described throughout this paper affect the performance of REmatch (Subsection 7.2);
(2) We compare the performance of REmatch against established RegEx engines (Subsection 7.3).

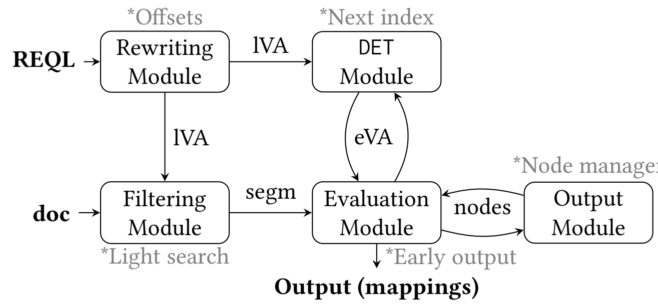


Figure 5: Architecture of REmatch. Optimizations are marked with * and in grey. lVA stands for Logical VA, eVA for Extended VA, and segm for segments.

We begin by describing the datasets and experiments used in the remainder of this section. All the data, queries and the implementation of REmatch, can be found at [1].

7.1 Experimental setup

The implementation. REmatch was implemented in C++, and includes all the components presented in the paper. An overview of the REmatch architecture can be found in Figure 5. Each module was described in a section of the paper. The Rewriting module (Section 3) takes a REQL expression and converts it to a logical VA (lVA), which is used both by the DET module (Section 6), and by the Filtering module (Section 4). The latter processes the input document by splitting it into segments containing outputs, which are then passed to the Evaluation module (Section 6). The Evaluation module runs Algorithm 2 by communicating with the DET module to obtain the next state of the eVA, and with the Output module (Section 5), which creates the nodes of the data structure encoding the output mappings. Specific optimizations were highlighted in each section.

The datasets. We use the following three real-world text corpora:

(1) Literature. This is a combined corpus of collected works by English literature greats: Mark Twain, William Shakespeare, and Charles Dickens. We used texts provided by the Project Gutenberg[9], and concatenated them into a single document of size 50.7 MB and around 50 million characters.
(2) DNA. This dataset consists of DNA sequences. In particular, we used the list of proteomes of the zebrafish organism, as provided by the BLAST initiative [6]. The combined size of this dataset is 38.5 MB and 38.5 million characters.
(3) SPARQL. Our final dataset consists of public query logs of the (now defunct) British Museum SPARQL endpoint, as collected by the Linked SPARQL Queries Dataset team [36]. We merged all the logs into a single document weighing 71.1 MB, and consisting of roughly 76 million characters.

The queries. Our queries for each dataset are designed as follows:

(1) Literature. Here we take a list of common English language morphemes[10], as provided by [25]. We then specify queries that look for 2-grams [25], that is, two consecutive words

[9] https://www.gutenberg.org/

[10] A morpheme is the smallest meaningful constituent of a linguistic expression, https://en.wikipedia.org/wiki/Morpheme.

each containing a morpheme from our list (e.g. the first word ends in -ing, and the second one in -er).

(2) DNA. Motif detection is a key task in DNA analysis[11]. In this query set we select 100 DNA motifs from the Prosite database [15] that commonly occur in our dataset. Our queries take any such pair of motifs, and look for their occurrences in the proteomic sequence separated by at most 20 characters.

(3) SPARQL. Our logs have distinct SPARQL queries listed in each line. For our expressions we fix two sets of up to three SPARQL keywords [40] (e.g., `WHERE` or `OPTIONAL`), and extract two consecutive queries where the first one contains the keywords from the first set, and the second one from the second set.

For each dataset roughly 10,000 queries were generated. We then sample 150 queries from each set and use these for our experiments. The queries were designed such they cover real world-scenarios where overlapping matches occur naturally. For instance, in the DNA dataset, a starting motif might be paired with multiple occurrences of an end motif, which is a natural use case for the all-match semantics, and something that is difficult to capture using modern RegEx engines, as shown in Table 3.

How we ran the queries. All the experiments were run on a Apple M1 Pro 10 cores/10 threads machine with clock speed 2064 – 3220 MHz and 16GB RAM. The operating system used was MacOS 13.1. Queries were run in succession, and each query was executed 5 times. We collected the average runtime and memory consumption of each query, and consider these in our measurements.

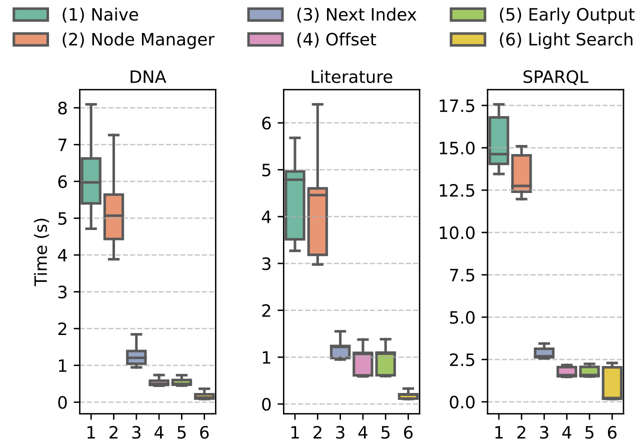
7.2 What do our optimizations do?

Throughout the paper we described a series of optimizations which define the REmatch system architecture, as illustrated in Figure 5. Of course, a natural question to ask is what is the effect of each one of these optimizations, and whether implementing the basic algorithm for computing all matches of a REQL expression would be competitive enough already? In this subsection, we test that hypothesis, and show that a vainilla implementation of Algorithm 2 runs several orders of magnitude slower that the full REmatch stack. Additionally, we test how each single optimization affects the performance of REmatch.

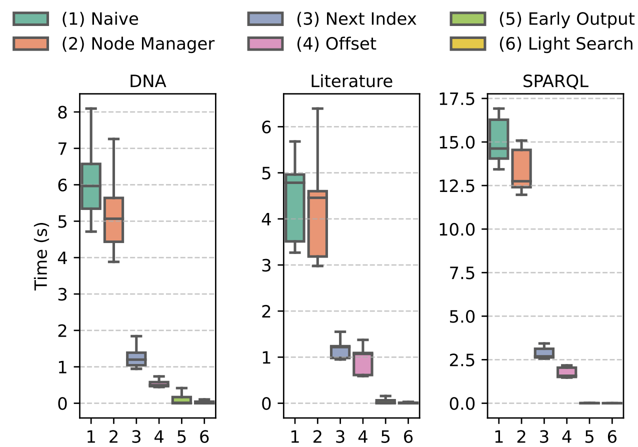
For this, we run the experiments described in subsection 7.1, and test the following versions of REmatch:

- Naive, which is just the implementation of Algorithm 2.
- Node Manager, which adds the `NM` module to discard unusable nodes as soon as possible (Section 5).
- Next Index, which uses a bit array for ASCII characters in a transition for quick access (Section 6).
- Offset, which postpones storing the (potential) output information as much as possible (Section 3).
- Early Output, which outputs results as soon as they are available (Section 5).
- Light Search, which finds a valid segmentation of the document that are guaranteed to produce the output and runs the full algorithm over these segments (Section 4).

[11] https://en.wikipedia.org/wiki/Sequence_motif



(a) Runtime performance



(b) Time to produce the first output

Figure 6: Performance gains of REmatch optimizations

Here each new version includes all the optimizations previously listed. For example, the Offset version of the algorithm includes both the Node Manager and the Next index optimizations. As the optimizations are designed to help the algorithm in various aspects, we test their impact along three dimensions: (i) time; (ii) memory consumption; and (iii) time needed to return the first output. Boxplots for time and first output time are presented in Figure 6, while Table 2 shows the average memory consumption.

Discussion. As we can observe (Figure 6 (a)), among all three datasets, adding the Node Manager module makes the runtime of our experiments drop noticeably, however the real savings come from the Next index optimization, which allows us check whether a character triggers an automaton transition instantaneously. The Offset and Light Search version reduces the runtime in a meaningful manner as well. Concerning memory consumption (Table 2), Node Manager drastically reduces the memory consumption compared to the Naive version. Due to storing more information in each transition, Next index increases memory consumption, and

	DNA	Literature	SPARQL
NAIVE	823.3	379.9	1229.69
NODE MANAGER	6.08	2.1	8.75
NEXT INDEX	18.32	2.33	10.1
OFFSET	18.66	2.32	7.78
EARLY OUTPUT	16.09	2.21	3.64
LIGHT SEARCH	23.36	2.09	3.87

Table 2: Average memory usage of REmatch versions (in MB).

OFFSET and EARLY OUTPUT tend to decrease it again due to discarding nodes that are not needed any more by the algorithm sooner. Considering time to first output (Figure 6 (b)), we can see that NEXT INDEX drops the runtime by an order of magnitude, and another order of magnitude is gained by EARLY OUTPUT, whose main objective is precisely this: deliver the outputs to the user as soon as possible. Overall, our experiments show that the proposed optimizations, which are closely linked to the REmatch architecture (Figure 5), indeed improve the performance of base algorithm, denoted NAIVE, by orders of magnitude.

7.3 Comparison with other engines

The setup. Here we do a thorough analysis of how REmatch compares to classic RegEx processing libraries. For this, we tried to be thorough, and include both engines that can approximate the all-match semantics using look-around operators, and the ones that cannot. In our comparison the following engines are used:

- PCRE [30] version: 8.45;
- PCRE2 [31] version 10.40 (using JPCRE2 C++ wrapper [23]);
- Boost.Regex [7] version 1.81.0;
- Oniguruma [29] version 6.9.7.1;
- RE2 [32] version 2021–11–01; and
- TRE [38] version 0.8.0.

For engines that support look-around operators (PCRE, PCRE2, Boost and Oniguruma) we rewrite the experiments from Subsection 7.1 so that they retrieve all the matches. For RE2 and TRE, which do not support look-around operators, we rewrite the queries using capture groups such they resemble as closely as possible the original experiments, although they do not perform the same task.

The results are presented in Figure 7 and Table 3. Here we compare only in terms of time. The memory consumption was relatively stable along all the engines, with REmatch generally using slightly more memory, which is justified by the extra bookkeeping needed to retrieve all the matches. REmatch had some spikes in memory usage for the DNA dataset, and PCRE2 in the SPARQL dataset, but the document size still dominated memory usage significantly.

In this discussion, it is essential to recall that REmatch is incomparable with standard RegEx engines, given that it uses a different query language for always finding all matches (even DNA queries with look-around provide less number of outputs). Then it is not our purpose here to prove that REmatch is "faster" than other RegEx engines. Quite the opposite, we want to test that, although REmatch is running a more heavy processing task, the performance is comparable to the standard and highly optimized RegEx engines.

Discussion. As we can see, REmatch shows good performance as compared to other engines. On the Literature dataset, REmatch is bested only by RE2, which does not look for all the outputs. On the DNA dataset REmatch is a close third, bested only by RE2 and PCRE2 by a tiny margin. When it comes to SPARQL the story is similar. We remark that in this dataset TRE throws an error on every query. Comparing the number of outputs, in Table 3 we can observe that REmatch generally does more work compared to other engines. This is particularly evident when comparing to engines not using look-around operators (RE2 and TRE). Even for the engines with look-around supported, we sometimes cannot capture all the outputs (for instance when two nested matches start at the same position), as witnessed in Table 3. Overall, the experiments illustrate that the task of encountering *all* the outputs, including overlapping ones, can be done with minimal overhead when comparing with classical RegEx matching.

	DNA	Literature	SPARQL
REmatch	**16,187.4**	**706.6**	**29,424.2**
RE2	10,556.9	704.9	12,287.8
PCRE	13,130.4	705.1	**29,424.2**
PCRE2	13,130.4	705.1	**29,424.2**
Boost	13,130.4	642.6	**29,424.2**
Oniguruma	13,130.4	705.5	**29,424.2**
TRE	10,556.9	704.2	N/A

Table 3: Average number of outputs (highest in bold).

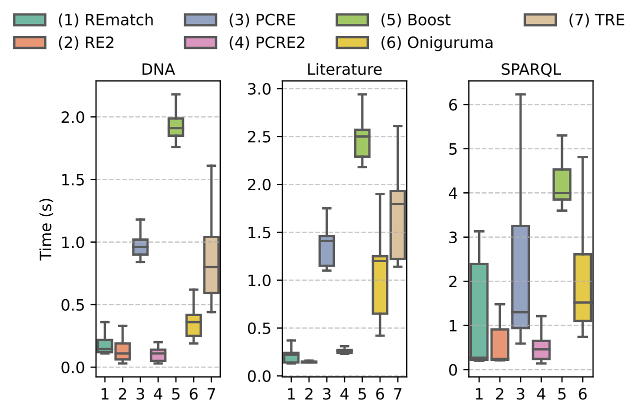


Figure 7: Runtime metrics of our experiments

8 Conclusions

This paper presents REmatch, a novel RegEx engine allowing to find all matches. The reader can test the all-match semantics at our beta demo available on www.rematch.cl. Such a semantics is relevant in areas like literature or DNA analysis, where overlapping matches do matter. We experimentally prove that although REmatch does a more demanding job, it does so with almost no additional cost when compared to classical RegEx engines, making it a good candidate for use cases where all matches are needed. Finally, this paper presents the architecture, algorithms, and optimizations for capturing all matches. However, we see plenty of room for new optimizations like state reductions, filtering, or indices. We leave this for future work.

References

[1] [n.d.]. REmatch Website. https://github.com/REmatchChile/REmatch-paper. Accessed on 2023-02-01.

[2] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. 2021. Constant-Delay Enumeration for Nondeterministic Document Spanners. *ACM Trans. Database Syst.* 46, 1 (2021), 2:1–2:30.

[3] Antoine Amarilli, Louis Jachiet, Martin Muñoz, and Cristian Riveros. 2022. Efficient Enumeration for Annotated Grammars. In *PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022.* 291–300. https://doi.org/10.1145/3517804.3526232

[4] Marcelo Arenas, Francisco Maturana, Cristian Riveros, and Domagoj Vrgoč. 2016. A Framework for Annotating CSV-like Data. *Proc. VLDB Endow.* 9, 11 (July 2016), 876–887. https://doi.org/10.14778/2983200.2983204

[5] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. 2020. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News* 7, 1 (2020), 4–33.

[6] BLAST: Basic Local Alignment Search Tool 2022. https://blast.ncbi.nlm.nih.gov/doc/blast-help/downloadblastdata.html.

[7] Boost Regex Library 2022. https://github.com/boostorg/regex.

[8] Russ Cox. 2007. https://swtch.com/~rsc/regexp/regexp1.html.

[9] Johannes Doleschal, Benny Kimelfeld, Wim Martens, Yoav Nahshon, and Frank Neven. 2019. Split-Correctness in Information Extraction. In *PODS 2019*. ACM, 149–163.

[10] Krzysztof Dorosz and Anna Szczerbinska. 2009. Enhancing Regular Expressions for Polish Text Processing. *Comput. Sci.* 10 (2009), 19–36. https://doi.org/10.7494/csci.2009.10.3.19

[11] Ronald Fagin, Benny Kimelfeld, Frederick Reiss, and Stijn Vansummeren. 2015. Document Spanners: A Formal Approach to Information Extraction. *J. ACM* 62, 2 (2015), 12:1–12:51.

[12] Fernando Florenzano, Cristian Riveros, Martín Ugarte, Stijn Vansummeren, and Domagoj Vrgoc. 2020. Efficient Enumeration Algorithms for Regular Document Spanners. *ACM TODS* 45, 1 (2020), 3:1–3:42.

[13] Christopher A. Flores, Rosa L. Figueroa, and Jorge E. Pezoa. 2021. Active Learning for Biomedical Text Classification Based on Automatically Generated Regular Expressions. *IEEE Access* 9 (2021), 38767–38777. https://doi.org/10.1109/ACCESS.2021.3064000

[14] Jeffrey EF Friedl. 2006. *Mastering regular expressions.* " O'Reilly Media, Inc.".

[15] The Swiss-Prot group. 2022. The PROSITE database. https://ftp.expasy.org/databases/prosite/prosite.dat. World Wide Web Consortium.

[16] John E. Hopcroft and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley.

[17] How can I match overlapping strings with regex? 2014. https://stackoverflow.com/questions/20833295/how-can-i-match-overlapping-strings-with-regex/33903830.

[18] How to find overlapping matches with a regexp? 2013. https://stackoverflow.com/questions/11430863/how-to-find-overlapping-matches-with-a-regexp.

[19] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *International Conference on Management of Data (SIGMOD)*. ACM, 1259–1274.

[20] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. 2020. General dynamic Yannakakis: conjunctive queries with theta joins under updates. *VLDB J.* 29, 2-3 (2020), 619–653.

[21] IEEE and The Open Group. 2018. https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html.

[22] Walter L. Johnson, James H. Porter, Stephanie I. Ackley, and Douglas T. Ross. 1968. Automatic Generation of Efficient Lexical Processors Using Finite State Techniques. *Commun. ACM* 11, 12 (dec 1968), 805–813. https://doi.org/10.1145/364175.364185

[23] JPCRE2 C++ wrapper for PCRE2 library 2022. https://github.com/jpcre2/jpcre2.

[24] Katja Losemann and Wim Martens. 2014. MSO queries on trees: enumerating answers under updates. In *Joint Meeting of the Conference on Computer Science Logic (CSL) and the ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 67:1–67:10.

[25] Andrea D. Sims Martin Haspelmath. 2010. *Understanding Morphology*. Hodder Education.

[26] Francisco Maturana, Cristian Riveros, and Domagoj Vrgoc. 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.* ACM, 125–136.

[27] Martin Muñoz and Cristian Riveros. 2022. Streaming Enumeration on Nested Documents. In *25th International Conference on Database Theory, ICDT 2022, March 29 to April 1, 2022, Edinburgh, UK (Virtual Conference)*. 19:1–19:18. https://doi.org/10.4230/LIPIcs.ICDT.2022.19

[28] Gonzalo Navarro and Mathieu Raffinot. 2005. New Techniques for Regular Expression Searching. *Algorithmica* 41, 2 (2005), 89–116. https://doi.org/10.1007/s00453-004-1120-3

[29] Oniguruma – a modern and flexible regular expressions library 2022. https://github.com/kkos/oniguruma.

[30] PCRE – Perl Compatible Regular Expressions 2022. https://www.pcre.org/.

[31] PCRE2 – Perl-Compatible Regular Expressions 2022. https://github.com/PCRE2Project/pcre2.

[32] RE2 regular expression library 2022. https://github.com/google/re2.

[33] Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. 2022. Enumeration for FO Queries over Nowhere Dense Graphs. *J. ACM* 69, 3 (2022), 22:1–22:37. https://doi.org/10.1145/3517035

[34] Luc Segoufin. 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.* 10–20.

[35] Anubhava Srivastava. 2017. *Java 9 Regular Expressions.* Packt Publishing. https://www.oreilly.com/library/view/java-9-regular/9781787288706/.

[36] The LSQ team. 2015. The Linked SPARQL Queries Dataset. http://aksw.github.io/LSQ/.

[37] Ken Thompson. 1968. Programming techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422.

[38] TRE – a lightweight, robust, and efficient POSIX compliant regexp matching library 2021. https://github.com/laurikari/tre.

[39] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *Proc. VLDB Endow.* 13, 9 (2020), 1582–1597.

[40] W3C Sparql 2013. SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/. World Wide Web Consortium.

A Compilation of REQL queries into logical VA

Let e be a REQL. We'll show that it there exists an ε-logical VA (that is, a logical VA that allows transitions that do not consume letters) $\mathcal{A}$ such that $[\![\mathtt{e}]\!]_d = [\![\mathcal{A}]\!]_d$ over any document d. We proceed by induction. For each of the base cases shown in Table 1 we can construct a LogicalVA $\mathcal{A}$ with two states, one initial and one accepting, and with a single transition labeled as the char class equivalent to e from the initial state to the accepting state. Is easy to see that this LogicalVA $\mathcal{A}$ satisfies $[\![\mathtt{e}]\!]_d = [\![\mathcal{A}]\!]_d$ over any document d.

The inductive case goes as follows: without any loss of generality suppose that $\mathtt{e}_1$ and $\mathtt{e}_2$ are two REQLs such that there exists $\mathcal{A}_1 = (Q_1, \delta_1, q_0^1, q_f^1)$ and $\mathcal{A}_2 = (Q_2, \delta_2, q_0^2, q_f^2)$ both LogicalVAs that satisfy $[\![\mathtt{e}_1]\!]_d = [\![\mathcal{A}_1]\!]_d$ and $[\![\mathtt{e}_2]\!]_d = [\![\mathcal{A}_2]\!]_d$ over any document d. Then,

- For $\mathtt{e} = \mathtt{e}_1\mathtt{e}_2$ the LogicalVA $\mathcal{A} = (Q_1 \cup Q_2, \delta_1 \cup \delta_2 \cup \{(q_f^1, \varepsilon, q_0^2)\}, q_0^1, q_f^2)$ is such that over any document d, $[\![\mathtt{e}]\!]_d = [\![\mathcal{A}]\!]_d$.
- For $\mathtt{e} = \mathtt{e}_1 | \mathtt{e}_2$ the LogicalVA $\mathcal{A} = (Q_1 \cup Q_2 \cup \{q_0, q_f\}, \delta_1 \cup \delta_2 \cup \{(q_0, \varepsilon, q_0^1), (q_0, \varepsilon, q_0^2), (q_f^1, \varepsilon, q_f), (q_f^2, \varepsilon, q_f)\}, q_0, q_f^2)$ is such that over any document d, $[\![\mathtt{e}]\!]_d = [\![\mathcal{A}]\!]_d$.
- For $\mathtt{e} = \mathtt{!x\{e_1\}}$ the LogicalVA $\mathcal{A}_{\mathtt{e}} = (Q' \cup \{q_0, q_f\}, \delta' \cup \{(q_0, [x, q_0'), (q_f', x\rangle, q_f)\}, q_0, q_f)$ is such that over any document d, $[\![\mathtt{e}]\!]_d = [\![\mathcal{A}_{\mathtt{e}}]\!]_d$.
- For $\mathtt{e} = \mathtt{e}_1\mathtt{*}$ the LogicalVA $\mathcal{A} = (Q_1, \delta_1 \cup \{(q_0^1, \varepsilon, q_f^1), (q_f^1, \varepsilon, q_0^1)\}, q_0^1, q_f^1)$ is such that over any document d, $[\![\mathtt{e}]\!]_d = [\![\mathcal{A}]\!]_d$.
- For $\mathtt{e} = \mathtt{e}_1\mathtt{?}$ the LogicalVA $\mathcal{A} = (Q_1, \delta_1 \cup \{(q_0^1, \varepsilon, q_f^1)\}, q_0^1, q_f^1)$ is such that over any document d, $[\![\mathtt{e}]\!]_d = [\![\mathcal{A}]\!]_d$.
- For $\mathtt{e} = \mathtt{e}_1\mathtt{+}$ the LogicalVA $\mathcal{A} = (Q_1, \delta_1 \cup \{(q_f^1, \varepsilon, q_0^1)\}, q_0^1, q_f^1)$ is such that over any document d, $[\![\mathtt{e}]\!]_d = [\![\mathcal{A}]\!]_d$.
- For $\mathtt{e} = \mathtt{e}_1\mathtt{\{n,m\}}$ we can see that over any document d, $[\![\mathtt{e}]\!]_d = [\![(\mathtt{e}_1)^n(\mathtt{e}_1\mathtt{?})^{m-n}]\!]_d$ which can be easily shown to have an equivalent LogicalVA by applying induction with the previous rules.

An ε-logical VA can be easily transformed into a logical VA by removing all the ε transitions using depth-first search. Thus, we can conclude that $[\![\mathtt{e}]\!]_d = [\![\mathcal{A}]\!]_d$ over any document d for any REQL e.

B Offset optimization

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

```
!x{sparql[^\n]*}\n
```

which looks for the keyword `sparql`, and then proceeds to capture the text until the first end of line symbol is reached. In a log file, such as the ones studied in [4], this roughly corresponds to capturing a SPARQL query. Notice that here Algorithm 2 would store the position information for the opening of a variable `x` every time an `s` would be read. If the document we are reading has the text `sparx`, this run would then be extended for three more steps, although it will eventually be abandoned, and not result in any outputs. Given the node management required when changing the state of the ECS structure, this can bring a significant overhead to evaluation. In cases such as these, we can actually postpone (i.e. offset) the opening of the variable `x`. by proceeding as follows: (i) first read the word `sparql`; (ii) now open a variable `x`, but remember that it was actually opened six symbols before (i.e. it has an offset 6); (iii) proceed until the end of the expression. Then, when reconstructing the output in case of a successful run, we will simply start reading the output six symbols before the position that is actually stored in `ds` from Algorithm 2. Intuitively, one could write the expression above as:

$$\mathtt{sparql!x\{^{-6}[\hat{}\backslash n]*\}\ \backslash n}$$

In order to signal that when opened, the variable `x` actually stores characters starting six positions before the guarded position.

Offset logical VA. To formalize this concept we extend the definition of a logical VA as follows: a *logical variable-set automata* (offset logical VA) $\mathcal{A}_{\mathtt{offset}}$ is a tuple $(Q, \delta, \tau, q_0, q_f)$ that has the same structure as a logical VA, with the addition of an *offset marker* function τ that takes a variable marker as input and returns an integer.

A configuration, a run and an accepting run of $\mathcal{A}_{\mathtt{offset}}$ are defined exactly as if $\mathcal{A}_{\mathtt{offset}}$ were a logical VA. The only change is in the definition of its semantics. Suppose that

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

is an accepting run of $\mathcal{A}_{\mathtt{offset}}$. The mapping μ^ρ then is such that it maps x to $[i_j, i_k\rangle$ if, and only if, $o_{i_j+\tau([x)} = [x$ and $o_{i_k+\tau(x\rangle)} = x\rangle$. In other words, the mapping μ^ρ acts in the same way as if $\mathcal{A}_{\mathtt{offset}}$ were a logical VA, but taking into account the offsets given by τ for each of the variable markers present in the transitions of $\mathcal{A}_{\mathtt{offset}}$. Therefore, the semantics $[\![\mathcal{A}_{\mathtt{offset}}]\!]_d$ of $\mathcal{A}_{\mathtt{offset}}$ over a document d are defined as the set of all μ^ρ where ρ is an accepting run of $\mathcal{A}_{\mathtt{offset}}$ over d.

For the reasons that were explained previously. It is desirable to be able to construct an offset logical VA $\mathcal{A}_{\mathtt{offset}}$ from an initial logical VA $\mathcal{A}$ such that: (i) the semantics are preserved, i.e. $[\![\mathcal{A}_{\mathtt{offset}}]\!]_d = [\![\mathcal{A}]\!]_d$ and (ii) the offset function τ of $\mathcal{A}_{\mathtt{offset}}$ is optimal in a way to maximize the offsets for the variable markers, so that the decision of changing the state of the ECS is postponed while reading the document. In REmatch, this is done by the following algorithm, which we call `offset`.

Offset algorithm. In the implementation of REmatch, the `offset` optimization is done in the early stages of preprocessing the query. After the first parse of the expression, REmatch will build a eVA as described in [12], namely each transition either opening or closing a single variable (referred to as a capture transition), or processing a symbol (called a reading transition). We also assume no ε transitions to be present in the automaton.

First, for each variable opening we build a list of all the transitions that open this variable, and we do the same for each variable closing. The variable opening/closing is then offset in bulk, namely, either all of the transitions of the form $[x$ (or $x\rangle$) are moved at once (in order to preserve consistency), or none is. Let `captureList` be a list of all capture transitions opening or closing some variable (for instance, they are all of the form $[x\rangle$). For a capture transition $p \xrightarrow{[x} q$ we wish to see if there is also a transition $q \xrightarrow{a} r$, in order to interchange the transition reading the letter a, and the one opening the variable x. That is, we wish to achieve the following transformation in the states of our eVA:

$$p \xrightarrow{[x} q \xrightarrow{a} r \qquad \Rightarrow \qquad p \xrightarrow{a} q \xrightarrow{[x} r$$

We will say that we can offset $[x$ if the following conditions hold for every transition $p \xrightarrow{[x} q$ appearing in `captureList` associated with $[x$:

(1) q is not a final state;
(2) There are no transition of the form $q \xrightarrow{[y} r$, or of the form $p' \xrightarrow{v} q$, with v a variable marker, in the automaton;
(3) There is at least one transition of the form $q \xrightarrow{a} r$;
(4) For all transitions $q \xrightarrow{a} r$, q can not be reachable from r; and
(5) For any other transition $p' \xrightarrow{[x} q'$, if we take any $q \xrightarrow{a} r$, then q' is not reachable from r.

The first condition prevents offsetting a variable that leads to an accepting run. The second condition prevents manipulating a state which has multiple capture transitions associated with it. The third condition assures we actually have a transition to offset. Fourth transition makes sure that in case of moving the $[x$ variable marker forward, we will not create any loops involving this variable marker. Finally, the last condition ensures that moving one $[x$ variable marker will not result in an inconsistent run involving another such transition. This could, for instance, happen if we had both $p \xrightarrow{[x} q \xrightarrow{a} r$ and $p' \xrightarrow{[x} q' \xrightarrow{a} q$ in our automaton, since offsetting the first $[x$ transition would result in an automaton that has a run opening x twice.

Given each such list `captureList` (for example for $[x\rangle$), we manipulate each set of transitions $p \xrightarrow{[x} q \xrightarrow{a} r$, by switching the $[x$ and a symbols (in reality, an auxiliary state is created for this). The new `captureList` is then populated by adding the transitions $q \xrightarrow{[x} r$ to it, and the process is repeated as long as the newly created list satisfies the five conditions specified above.

Finally, we process the variable offsets by finding the reverse topological order of all the capture transitions. Then we offset the variables in this order so that we can move them as far forward as possible. To illustrate why this is important, consider the automaton consisting of a single run as follows:

$$q_0 \xrightarrow{[x} q_1 \xrightarrow{a} q_2 \xrightarrow{x\rangle} q_3 \xrightarrow{b} q4$$

If we were to first offset the $[x$ variable marker, this would result in:

$$q_0 \xrightarrow{a} q_1 \xrightarrow{[x^{-1}} q_2 \xrightarrow{x\rangle} q_3 \xrightarrow{b} q4$$

and $x\rangle$ could not be offset due to condition (2) above. On the other hand, processing the variables in reverse topological order results in:

$$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{[x^{-2}} q_3 \xrightarrow{x\rangle^{-1}} q4$$

.

C Proof of Theorem 4.2

Proof. Let us name the sequence as $\sigma = [i_1, j_1\rangle, \ldots, [i_k, j_k\rangle$. It not hard to see that σ is a segmentation. First notice that i and j can only change their values to ℓ and $\ell + 1$ in the execution, so as $0 \leq \ell \leq n$, then every span in σ must be contained in the document. Now suppose that $k > 1$ (otherwise σ is trivially a segmentation). Pick any $1 \leq h < k$. Then Algorithm 1 adds the span $[i_h, j_h\rangle$ in line 10 after an **else if** statement that ensures that $j_h \leq \ell$ in that iteration. Rightly afterwards on line 11 i starts holding the value $\ell + 1$, then $j_h < i$ right at the end of that iteration. So whenever the algorithm outputs $[i_{h+1}, j_{h+1}\rangle$ it is certain that $j_h < i_{h+1}$ because i cannot decrease in value.

Let us prove now that σ is a valid segmentation. Let $\mu \in [\![\mathcal{A}]\!]_d$ be a mapping of $\mathcal{A}$ over d. We must show that there exists $1 \leq h \leq k$ such that there is a mapping $\mu' \in [\![\mathcal{A}]\!](d[i_h, j_h\rangle)$ that satisfies $\mu = \mu'_{+h}$. As we have the mapping μ then there is an accepting run

$$\rho \;=\; (q_0, \iota_0) \xrightarrow{o_1} (q_1, \iota_1) \xrightarrow{o_2} \cdots \xrightarrow{o_m} (q_m, \iota_m)$$

over d that satisfies $q_m = q_f$. The run ρ defines the mapping μ, therefore all the spans that μ holds are contained inside the span $[\iota_0, \iota_m\rangle$. We proceed to show that the span $[\iota_0, \iota_m\rangle$ must be contained inside a span $[i_h, j_h\rangle$ of σ. In Algorithm 1, consider the iteration inside the main loop when $\ell = \iota_0$. We can be sure that in this iteration the function $\texttt{next}_\delta(S, a_\ell)$ does not return `ends` as true, because we know that ρ is accepting and it doesn't end at ι_0. Therefore the variable i does not change its value and it must satisfy that $i \leq \iota_0$ at the end of this iteration. The same argument can be made for the iterations that follow, namely when $\ell \in \{\iota_1, \ldots, \iota_m\}$. The variable i must satisfy $i \leq \iota_0$ at the end of each of these iterations. Now consider the iteration when $\ell = \iota_m$. It is clear that $\texttt{next}_\delta(S, a_\ell)$ will return `outputs` as true because ρ is accepting. Hence, j will change its value to $\iota_m + 1$ in this iteration, Thus $j < \iota_m$. Given that the mapping μ cannot contain a zero-length span (otherwise $\mathcal{A}$ would not constitute a valid logical VA), then $i \leq \iota_0 < \iota_m \leq j$. In future iterations is easy to see that:

(i) the value of j can only increase,
(ii) the value of i can cannot change before outputting a span, and
(iii) the algorithm will eventualy output a span $[i_h, j_h\rangle$ that satisfies $i_h \leq \iota_0 < \iota_m \leq j_h$.

Therefore, $[\iota_0, \iota_m\rangle$ is contained inside a span $[i_h, j_h\rangle$ of σ. Now if we take the document $d[i_h, j_h\rangle$ it must hold that there is a mapping $\mu' [\![\mathcal{A}]\!](d[i_h, j_h\rangle)$ defined by the accepting run

$$\rho' \;=\; (q_0, \iota_0 - h) \xrightarrow{o_1} (q_1, \iota_1 - h) \xrightarrow{o_2} \cdots \xrightarrow{o_m} (q_m, \iota_m - h)$$

of $\mathcal{A}$ over $d[i_h, j_h\rangle$. Furthermore, it satisfies that $\mu = \mu'_{+h}$.

The converse is straightforward. Let $\mu' \in [\![\mathcal{A}]\!](d[i_h, j_h\rangle)$ be a mapping of $\mathcal{A}$ over $d[i_h, j_h\rangle$ for some $1 \leq h \leq k$. Therefore, there exists an accepting run

$$\rho' \;=\; (q_0, \iota_0) \xrightarrow{o_1} (q_1, \iota_1) \xrightarrow{o_2} \cdots \xrightarrow{o_m} (q_f, \iota_m)$$

of $\mathcal{A}$ over $d[i_h, j_h\rangle$. This means that it must exist an accepting run

$$\rho \;=\; (q_0, \iota_0 + h) \xrightarrow{o_1} (q_1, \iota_1 + h) \xrightarrow{o_2} \cdots \xrightarrow{o_m} (q_f, \iota_m + h)$$

of $\mathcal{A}$ over d. It is straightforward that the mapping μ^ρ is the same mapping μ' with an offset, i.e. $\mu^\rho = \mu'_{+h}$. Thus, $\mu'_{+h} \in [\![\mathcal{A}]\!]_d$. □