Parameterized Pattern Queries

Cédric du Mouza [†,⋆], Philippe Rigaux [‡,⋆] and Michel Scholl [†,⋆]

[†] *Lab. CEDRIC, CNAM, Paris, France, {dumouza,scholl}@cnam.fr*

[‡] *LAMSADE, Univ. Paris-Dauphine, Paris, France, Philippe.Rigaux@dauphine.fr*

[⋆] *WISDOM Project, http://wisdom.lip6.fr*

Abstract

We introduce *parameterized pattern queries* as a new paradigm to extend traditional pattern expressions over sequence databases. A parameterized pattern is essentially a string made of constant symbols or variables where variables can be matched against any symbol of the input string.

Parameterized patterns allow a concise and expressive definition of regular expressions that would be very complex to describe without variables. They can also be used to express additional constraints, to "relax" pattern expressions by allowing more freedom, and finally to cluster patterns in order to minimize the number of symbols' comparisons.

Key words: Representation and Manipulation of Data, Pattern matching, Optimization

1 Introduction

The detection of patterns in sequence databases is a requirement of many applications such as analysis of stock market prices (where a common problem for example consists in detecting two consecutive peaks in price patterns in order to make a buying decision), search for DNA sequences, stream mining, etc. Motivated by these applications, several models have been recently proposed to express pattern-based queries and to retrieve efficiently sequences that match them [1–5]. Many languages proposed in these articles extend in some way (querying, aggregation, data mining) the functionalities of SQL with some variant of regular expressions and query evaluation techniques built on well-known pattern matching algorithms.

In this paper, we propose an extension of traditional patterns with *variables* which can be bound to any value of the underlying discrete domain during query evaluation. For instance, if `a` is a value and `x` is a variable, the parameterized pattern `@x.a.@x` denotes all the substrings where `a` is preceded and followed by the *same* value. This extension provides a much more expressive and flexible querying framework. Indeed, the presence of variables offers many opportunities to match a pattern with a sequence by simply changing the variables bindings. Moreover, the introduction of variables promotes patterns to first-class query objects, since variables can be used in other parts of a query for expressing constraints (e.g., `@x != c`), joins (`@x = @y`, where `x` and `y` appear in different patterns), output of variable values, etc. The constraints may be simple comparisons between variables and constants, or more complex application-dependent predicates. Actually, as shown in Section 4, the specific constraint language is independent from our extension because taking account of constraints is a part of our pre-processing phase.

A potential problem associated with any extension of pattern-matching queries is the cost of the query evaluation algorithms. In the traditional setting (pattern-matching on strings) several well-known algorithms have been proposed to efficiently perform this search, either exactly [6–8] or approximately. Our main contribution is an extension of the KMP algorithm [7] to parameterized pattern matching which preserves its linear time complexity and enjoys low space requirements. More specifically, we show that the space requirements are independent from the alphabet or input sequence sizes, and that the time complexity matches that of related algorithms in the worst case, and is lower on the average.

Next, we analyze the potential of parameterized queries to relax the matching constraints. Pattern `c.a.b` for instance can be relaxed to `@x.a.b`, or `c.a.@y`, or `@x.a.@y`, etc., by replacing one constant symbol by a variable. Obviously, a parameterized pattern matches a superset of the sequences matched by the same pattern where one variable is bound to a constant symbol. We provide a formal description of query relaxation. We also show how this feature can be exploited when many patterns have to be evaluated together against an input sequence.

Relaxation allows the clustering of query patterns, and leads to a common evaluation of the parts shared by similar patterns. This constitutes a useful feature in publish/subscribe applications where a crucial property of the notification system is its ability to group subscriptions and to filter out events in order to minimize useless or redundant comparisons [9,10].

We implemented our algorithms and carried out several evaluations. The experimental evaluation shows that our algorithm saves a large amount of computations and therefore decreases the query evaluation time compare to a naïve approach. The filtering approach is also implemented and discussed.

In the following we describe first (Section 2) some motivating applications with sample queries. Section 3 introduces the data model and Section 4 is devoted to query evaluation algorithms, including our solution. In Section 5 we consider the evaluation of multiple patterns over a large input sequence and describe how parameterized patterns can be used to filter out many redundant or useless comparisons. We provide experimental results in Section 6. Related work is covered by Section 7. Some conclusions are drawn in Section 8. This paper extends shorter versions presented in [11] and [12]. More specifically, we improve the representation of the edges and propose a quadratic algorithm for building the table of edges, instead of the $O(n^3)$ naive algorithm presented in [11]. The present paper also features the proofs of the results from both [11] and [12], as well as a detailed complexity analysis. Finally it provides additional motivating examples and experiments to validate our approach.

2 Informal presentation

We provide in this section an illustration of our approach with some representative applications. Their common feature is to represent as *sequences* the evolution of values over a discrete domain for some information of interest, and to perform querying and analysis tasks on these sequences.

2.1 Motivating examples

Applications that deal with DNA or proteins rely on a database that stores millions of sequences [13]. Consider for instance a protein sequence database. Basically a protein is often composed of between 100 and 200 amino acids. There are 20 distinct amino acids that are commonly denoted by a symbol with one letter. Here is the example of lysozyme, composed of 130 amino acids:

K V F E R C E L A R T L K R L G M D G Y R G I S L A N W M C L A K W
E S G Y N T R A T N Y N A G D R S T D Y G I F Q I N S R Y W C N D G K
T P G A V N A C H L S C S A L L Q D N I A D A V A C A K R V V R D P Q
G I R A W V A W R N R C Q N R D V R Q Y V Q G C G V

where for instance N is the standard symbol to denote the amino acid named Asparagine. Beyond classical pattern matching on such sequences, our language allows advanced parameterized search. For instance the parameterized pattern `@x.Q.L.@x` matches the sequences where the substring `Q.L` is found, preceded and followed by the same letter. Variables provide a concise way to represent patterns that otherwise would be expressed with a regular expression that enumerates all the possible bindings, and whose size is likely to discourage any user.

Next, consider another application that aims at analyzing the behavior of web users on a site in order either to improve the ergonomy of the site, or to find the best places for advertisements. Assume that each page is uniquely referred to by an *url*. The database can thus store the sequences of page urls – or *histories* – successively crawled by a user [14,15]. A simple query of interest in such a context is for instance to search for users that came back to page `A` after visiting another page. This can be expressed by the pattern `A.@x.A`. The value of `@x` can be output if required when a match has been found (and thus when a value is bound to `@x`).

Finally, let us take a spatio-temporal application that will serve as a support to our examples in the rest of the paper. As in [16], we consider a partition of a 2D embedding space such that each zone is uniquely identified with a symbol from an alphabet Σ. This partition is the reference map $\mathcal{M}$ supporting queries.

Assume that each object is equipped with a location-aware device that periodically sends its position and that $\Sigma = \{a, b, c, d, e, f, g\}$. Since each object moving in the partitioned area crosses a sequence of distinct zones (we assume at least one event per zone), our patterns can be used to query such sequences. Here is a sample of such queries (they will be referred to by $Q_i, i = 1, \cdots, 4$ in the following)

- Q_1: which objects went through zone `a`, then crossed zone `d` and moved to zone `c`?
- Q_2: which objects went through zone `b`, then crossed `c` and `e` and then moved to `f`?
- Q_3: which objects went from `f` to `d` crossing another zone?
- Q_4: which objects left one zone to reach `a`, then came back to their departure zone before going to another zone?

The corresponding patterns for these queries are respectively Q_1 =`a.b.c`, Q_2 =`b.c.e.f`, Q_3 =`f.@x.d` with an instantiation of `@x` different from `f` and `d`, and Q_4 =`@x.a.@x.@y`, with an instantiation of `@x` different from those of `@y`.

2.2 Features of the model

The main feature of our language is the introduction of variables in pattern expressions. Variables bring flexibility in many aspects: expression of constraints, query relaxation, comparison and filtering of patterns.

Variables can be seen as references to some symbols of the input sequence. This creates a bridge between the pattern expression and the classical query predicates. A first potential use is the expression of additional constraints. Consider for instance the DNA application. The pattern `Y.A.@x` can be combined with the constraint `Polar(@x)` where `Polar` is a predicate which checks whether the instantiation of `@x` belongs to the `Polar` class of amino acids. Second, variables can simply be introduced in the query result. Referring to the previous example, this would allow to know exactly which amino acid `@x` has been met when a match is found with `Y.A.@x`.

Next, by replacing constant symbols with variables, the user can relax the constraints on sequences expressed by a pattern, in order to capture more sequences. Returning to the web log analysis application, we saw that `A.@x.B.@x` matches all the sequences where a page `@x` is accessed successively from two distinct pages, respectively `A` and `B`. If we do not wish to mention explicitely `A` and `B` in the previous example, we can relax the pattern as `@y.@x.@z.@x`, along with the constraints `@x != @y` and `@y != @z`. This matches all the sequences where page `@x` is accessed successively from two distinct pages, whatever their urls.

Finally adding or removing variables provides a useful mean to compare patterns or to filter out patterns evaluation. As mentioned above, variables can be seen as a way to express concisely a class of variable-free patterns. Consider the spatio-temporal application, and assume that several patterns are registered in order to detect the trajectories that go from zone `g` to zone `d`. Some possible expressions are `g.c.d`, `g.b.d`, `g.e.d`. A relaxation common to these three patterns is `g.@x.d` which represents the class of variable-free patterns for trajectories that go from `g` to `d` with one intermediate step. The specification of such "summaries" brings the following benefits:

- From a classification point of view, one can define a comparison strategy based on the addition or removal of variables. More specifically, we exhibit a partial order on patterns based on the number of variables necessary to transform one into another.
- The second benefit is less common, although important as well. Basically, if a trajectory does *not* match `g.@x.d`, the common relaxation of our three patterns, then it will not match *any* of them. This property allows to avoid many useless computations by creating clusters of pattern-based queries and by filtering out the evaluation of these queries.

In summary our approach aims at providing a flexible, powerful and efficient pattern-based query language. The flexibility is brought by the presence of variables. The larger the number of variables in a pattern, the larger the number of matches which can be found. Of course, the additional cost, if any, of these extended functionalities has to be evaluated. We show in the following that, in spite of an apparent added complexity, we can still rely on efficient pattern matching operations.

3 The model

Let Σ be a finite set of symbols and $\mathcal{V}$ be a set of variables such that $\Sigma \cap \mathcal{V} = \emptyset$. In the following, letters `a`, `b`, `c`, ... denote symbols from Σ, and `@x`, `@y`, `@z`, ... variables. A *sequence* is a word in Σ^*. A *pattern* is a word $t_1.t_2 \ldots t_m$ in $(\Sigma \cup \mathcal{V})^*$. In their simplest form, patterns are words in Σ^* such as, for instance, $Q_1 =$ `a.d.c` and $Q_2 =$ `b.c.e.f`. For clarity, we use the string concatenation symbol "." to separate symbols from Σ or variables in a pattern.

The interpretation of a pattern P *without variable* is natural: a sequence T *matches* a pattern P if P is a substring of T. The interpretation of patterns (with variables) is an extension of this trivial semantics: a sequence T matches a pattern P if one can substitute each variable in P by a symbol from Σ, such that the resulting pattern is a substring of T. More formally:

Definition 1 (Substitution and valuation) *A* substitution σ *is a finite set of the form* $\{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$ *where* $x_i \in \mathcal{V}, i = 1, \ldots, n$, *and each* t_i *is either a variable in* $\mathcal{V}$ *or a symbol in* Σ. σ *is a* valuation *if* $t_i \in \Sigma$, *for all* $i \in [1, n]$.

$\sigma(P)$ denotes the pattern obtained from P by replacing, for each $x_i/t_i \in \sigma$, each occurrence of x_i in P by t_i. Each element x_i/t_i is called a *binding* for x_i and the set of variables $\{x_1, x_2, \ldots, x_n\}$ is denoted by *bound*(σ). Sometimes, if x is bound to t, for brevity t will be referred to as $\sigma(x)$. If, for instance, $P =$ `a.b.@x.@y.b.@z.b` and $\sigma = \{$`@x/c, @z/@x`$\}$, then $\sigma(P) =$ `a.b.c.@y.b.@x.b`. In the following *var*(P) and *symb*(P) denote respectively the set of variables and constant symbols in P.

Note that if σ is a valuation and *var*$(P) \subseteq$ *bound*(σ), then $\sigma(P)$ is a word in Σ^* (for the sake of clarity we shall denote a valuation by ν). Hence the definition:

Definition 2 (Interpretation of a pattern) *A sequence s matches a pattern P (denoted $s \in \mathcal{L}(P)$) iff there exists a valuation ν such that $\nu(P)$ is a substring of s.*

A query q is simply a pair $(P, \mathcal{C})$ such that P is a pattern, and $\mathcal{C} = \{c_1, c_2, \cdots, c_m\}$, $m \geq 0$, is a (possibly empty) set of predicates over *var*(P). The semantics of a query is straightforward from that of a pattern: a sequence satisfies a query iff there exists a valuation of *var*(P) which satisfies the pattern of the query and the predicates in $\mathcal{C}$. In the following we focus on the evaluation of patterns. The evaluation of queries checks the progressive bindings of variables against the predicates. Note that the predicates are application-dependent i.e., we might have spatial predicates in a mobile application, or specialized conditions on proteins, or url comparisons.

Symbol	**Meaning**
P	A pattern
m	The length of a pattern
l	A position within a pattern ($0 \leq l \leq m-1$)
n	the length of the sequence
$e, e_1, e_2, \cdots$	Edges
ν, σ	Resp.: a valuation, a substitution
$t_1, t_2, \cdots$	Symbols or variables from $\Sigma \cup \mathcal{V}$
$a, b, c, \cdots$	Symbols from Σ
$@x, @y, @z, \cdots$	Symbols from $\mathcal{V}$

Table 1
Table of symbols used in the paper

4 Query evaluation

We present now two algorithms for an evaluation of parameterized patterns. The first one follows a naïve approach which repeatedly checks the new read symbols and backtracks on the sequence whenever a mismatch occurs. The second one is our optimized technique. The notations used throughout the paper are listed in Table 1.

4.1 The naïve approach

The first algorithm is a simple extension of well-known pattern-matching techniques to patterns with variables and relies on the following operations: a *matching attempt* between a pattern P and a sequence T at a position i, and a *shift* of P whenever a mismatch occurs.

The Compare *operation*

A matching attempt compares, one by one, from left to right, the symbols $P[0], P[1], \ldots, P[m-1]$ of the pattern to the symbols $T[i], T[i+1], \ldots, T[i+m-1]$ of the sequence. During the matching attempt, the variables in $var(P)$ are progressively bound to symbols in Σ, and these bindings define a valuation ν, called the *runtime valuation*, initially empty. If $P[j]$ is a variable `@x`, the following binding rules apply:

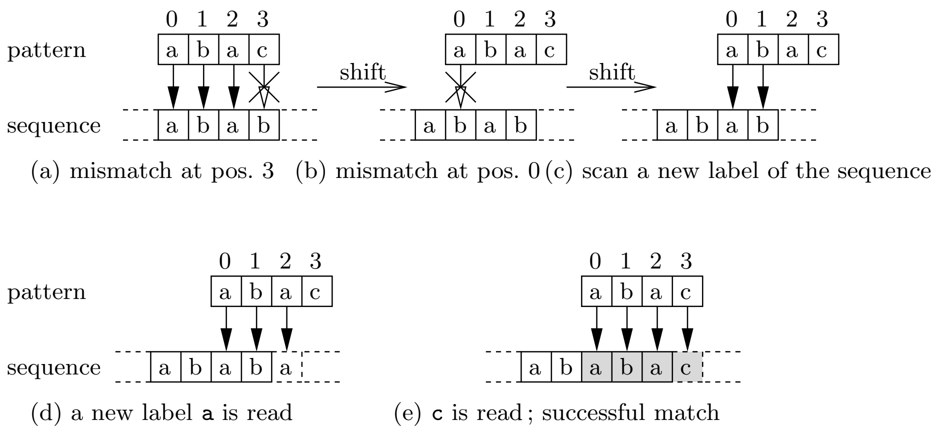
(1) if `@x` $\notin bound(\nu)$, the comparison is always successful and $\nu := \nu \cup$

{`@x/T[i+j]`} i.e., `@x` is bound to symbol $T[i+j]$. This binding remains in effect until the end of the matching attempt;

(2) else, if `@x` $\in$ *bound*(ν) the comparison is successful if and only if $T[j]$ is equal to ν(`@x`).

Consider for instance the matching attempt for P = `a.@x.b.@x` and T = `a.c.b`.... The comparisons are successful for $j = 0, 1, 2$. When $P[1]$ = `@x` is compared to $T[1]$ = `c`, variable `@x` is bound to symbol `c`. The valuation ν is, at this point, {`@x/c`}. The following comparison $P[3] = T[3]$ can then be successful only if the next symbol read in the sequence's representation is `c`, the current instantiation of `@x`. If all the comparisons are successful, then so is the matching attempt, else there is a mismatch. In both cases the Shift operation is performed.

The Shift *operation*



(a) mismatch at pos. 3 (b) mismatch at pos. 0 (c) scan a new label of the sequence

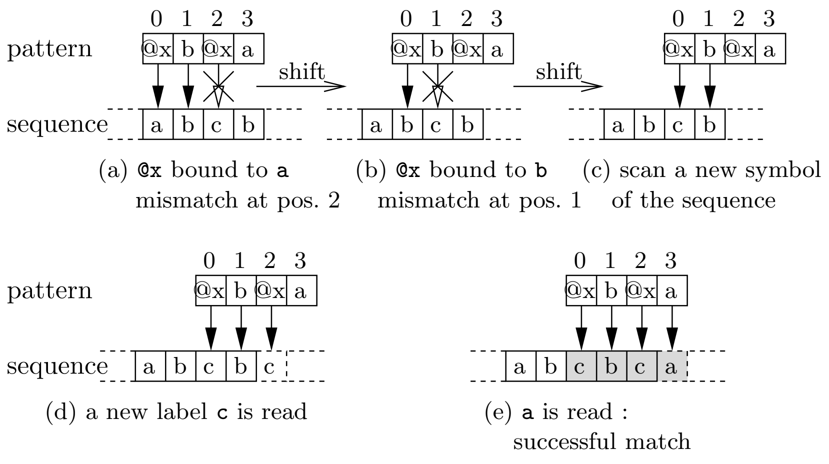
(d) a new label `a` is read (e) `c` is read; successful match

Fig. 1. Matching attempt for a pattern without variable

A Compare operation is performed each time a new symbol is read. Whenever a mismatch occurs (say, at position l, with $0 \leq l \leq m-1$), Shift shifts the pattern by one position and a comparison with the $l-1$ last symbols of the sequence has to be done. If the matching is successful, one reads the next symbol of the sequence, else a new shift is necessary. Figure 1 shows an example.

When the pattern contains variables the algorithm is quite similar except for the binding of the variables. Whenever a mismatch occurs, the current substitution is deleted: all the bindings are discarded. The pattern is shifted one symbol to the right. Figure 2 illustrates this algorithm.

This technique is simple but costly since the algorithm to test the whole sequence runs in $O(m \times n)$. Each symbol of the sequence is potentially checked several times against the pattern.



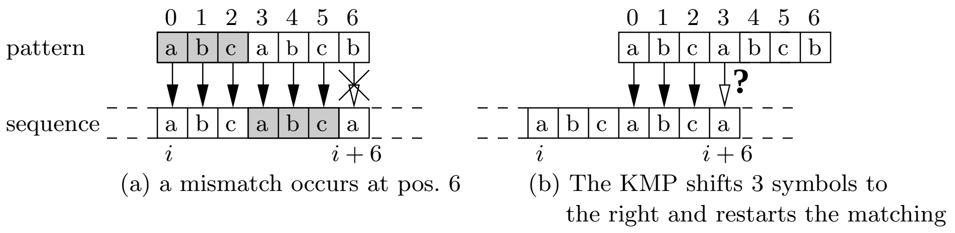
pattern
sequence
shift
0 1 2 3
@x b @x a
a b c b
(a) @x bound to a mismatch at pos. 2
(b) @x bound to b mismatch at pos. 1
(c) scan a new symbol of the sequence
(d) a new label c is read
(e) a is read : successful match

Fig. 2. Matching attempt for a pattern with variables

4.2 Optimized evaluation

The KMP algorithm

The Knuth, Morris and Pratt (KMP) [7] algorithm relies on the observation that, in the case of a mismatch, several symbols can be skipped. Moreover the pattern contains all the information needed for determining the number of symbols to be skipped. This is illustrated in Figure 3 with the pattern $P =$`a.b.c.a.b.c.b` and the substring $T =$ `a.b.c.a.b.c.a`.



pattern
sequence
0 1 2 3 4 5 6
a b c a b c b
a b c a b c a
?
i
i + 6
(a) a mismatch occurs at pos. 6
(b) The KMP shifts 3 symbols to the right and restarts the matching

Fig. 3. Example of a shift determined by the KMP algorithm

A mismatch occurs at position 6 of the pattern. We successfully superposed $P[0].\cdots.P[5]$ on the lastly read six symbols of T. A shift of one or two symbols to the right *always* leads to a mismatch. Indeed after a shift of one symbol, $P[0] =$`a` is compared to $T[i+1] =$`b`. Similarly a shift of two symbols attempts to superpose $P[0] =$`a` on $T[i+2] =$`c`.

Nonetheless, a shift of three symbols to the right is possible since $P[0].\cdots.P[2] = T[i+3].\cdots.T[i+5]$ (Figure 3(b)). It turns out that this shift is allowed because $P[0].\cdots.P[2] = P[3].\cdots.P[5]$. Therefore it can be determined by examining the pattern, at compile-time, independently from any specific sequence.

More generally, for each substring $s_l = P[0].\cdots.P[l-1]$, $l < m$ of P, we need to know the longest prefix e_l of s_l which is *also* a suffix of s_l. Such a string e_l

is called an *edge*. If the mismatch occurs at position l in the pattern, then the shift is of length $l - 1 - |e_l|$. Figure 4 illustrates this.

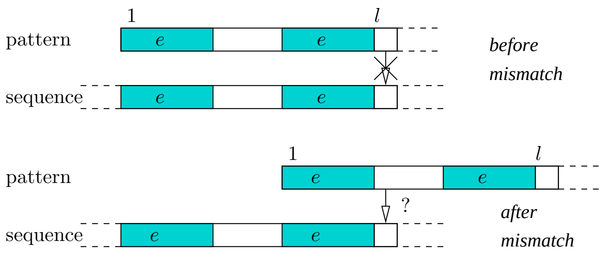


Fig. 4. Using an edge for determining the appropriate shift.

Note that taking the *longest* suffix means that the shift is minimal, and guarantees that the algorithm does not miss any solution. The edges are precomputed and stored in a table called the *table of edges*.

The KMP algorithm is decomposed into two steps. First an *offline* scan of the pattern to detect the edges for each substring in the pattern; second an *online* use of the table of edges to apply the appropriate shift each time a mismatch occurs. Using the table of edges often avoids the need to check an input symbol several times when performing a matching attempt.

Extended KMP algorithm

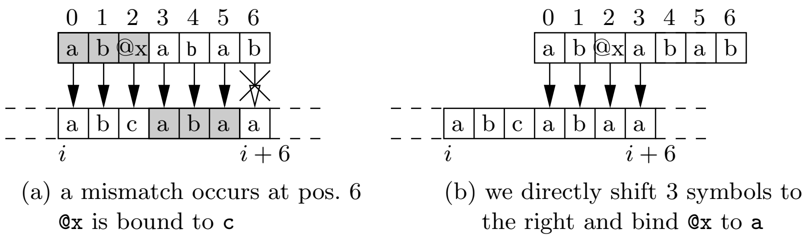
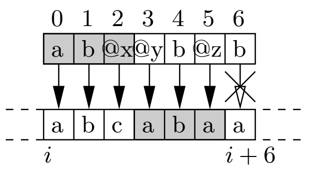
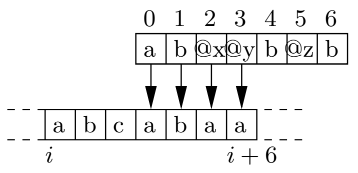

0 1 2 3 4 5 6
a b @x a b a b
a b c a b a a
i
i + 6
(a) a mismatch occurs at pos. 6 @x is bound to c
0 1 2 3 4 5 6
a b @x a b a b
a b c a b a a
i
i + 6
(b) we directly shift 3 symbols to the right and bind @x to a

Fig. 5. A shift for a pattern containing variables

Consider the pattern $P =$ `a.b.@x.a.b.a.b` and the example of Figure 5. A mismatch occurs at position 6. The shift is determined by two conditions. First, if we consider the pattern prefix `a.b.@x.a.b.a` and assume that `@x` is bound to `a`, one may envisage the shift that superposes the prefix $P[0].P[1].P[2]$ with the suffix $P[3].P[4].P[5]$. *After* the shift, $P[2] =$ `@x` is bound to `a`. So $P[0].P[1].P[2]$ may be superposed with the suffix $P[3].P[4].P[5]$ This first condition is necessary but not sufficient: assume we have a variable `@current` that stores the current symbol of the sequence, then the mismatch occurred because the value of `@current` (*i.e.*, $\nu($`@current`$)$) does not match with $P[6] =$ `b`. One must check that it matches the symbol $P[3]$ *after* the shift (Figure 5.b). Since $P[3] =$ `a` $= \nu($`@current`$)$, the necessary and sufficient conditions are fulfilled and the shift is possible.



(a) mismatch at pos. 6, `@x` bound to c, `@y` to a, and `@z` to a

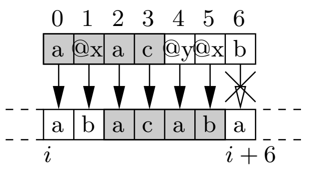


(b) shift 3 pos. to the right, `@x` bound to a, `@y` to a and `@z` not bound

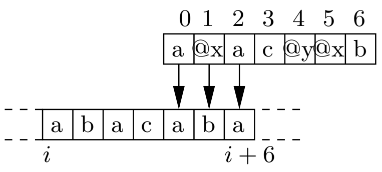
Fig. 6. A shift involving a substitution

Next, consider a more complex case where the bindings *after* the shift depend on the bindings *before* the mismatch (Figure 6). Once again a mismatch occurs at position 6, `@x` being bound to `c`, `@y` to `a` and `@z` to `a`. The current symbol that leads to the mismatch is ν(`@current`) = `a`.

The first condition applies to the edges of the sub-pattern $P[0].\cdots.P[5]$ = `a.b.@x.@y.b.@z`. The prefix `a.b.@x` can be superposed with the suffix `@y.b.@z` if and only if `@y` is bound to `a`. In that case the shift binds `@x` to the former binding of `@z`, `a`. There is no condition on the value of the current symbol of the sequence: since $P[3]$ is the first occurrence of a variable, `@y`, it is always possible to bind `@y` to the value of `@current` after the shift. The analysis of the pattern at compile-time gives all the needed information to evaluate the conditions and the substitutions that must be performed at runtime.



(a) mismatch at pos. 6, `@x` bound to `b`, `y` to `a` since `@x` is bound to `b`, edge of length 4 is not allowed.



(b) since `@y` was bound to `a`, and the read symbol is a, applying an edge of length 2 is possible ; then `@x` is now bound to `b`

Fig. 7. A shift that depends on the runtime valuation and on the value of `@current`

Here is a last example which develops the full mechanism of conditions/ substitutions. In Figure 7.a, a mismatch occurs at position 6. The sub-pattern $P[0].\cdots.P[5]$ is `a.@x.a.c.@y.@x`. An edge of length 4 can be envisaged, since the prefix `a.@x.a.c` can be superposed on the suffix `a.c.@y.@x`, the necessary condition being that $\nu(P[5]) = P[3]$, i.e., `@x` must be bound to `c` prior to the shift. This is not the case in Figure 7. Next we can consider an edge of length 2 which superposes the prefix `a.@x` on the suffix `@y.@x`. This is only possible if `@y` is bound to `a`, which, on the example of Figure 7, is true. It remains to check the condition on `@current`: for a shift of length 2, one must have ν(`@current`) = $P[2]$. This is verified and the shift can be performed.

As shown by the previous examples, the choice of an edge at run-time depends on conditions on the variables bindings and on the value of the current symbol of the sequence. Moreover a shift determines a substitution of variables values which depends, partially or totally, on the runtime valuation. The notion of *edge* covers these conditions and substitutions.

Definition 3 (Edge of a pattern) *Let P be a pattern of length m. An edge of P is a triple $(length, \nu_{min}, \sigma_{shift})$, where ν_{min} is a valuation and σ_{shift} a substitution, which satisfies the following properties:*

- $\nu_{min}(\sigma_{shift}(P[0].\cdots.P[length-1])) = \nu_{min}(P[m-length].\cdots.P[m-2].\texttt{@current})$,
- *there does not exist an edge $e' = (length, \nu'_{min}, \sigma'_{shift})$ with $\nu'_{min} \subseteq \nu_{min}$.*

The valuation ν_{min} expresses the necessary and sufficient conditions for applying the shift: given the runtime substitution ν, the edge e is applicable iff $\nu_{min} \subseteq \nu$ (we sometimes say that ν is *compatible* with ν_{min}). Finally σ_{shift} is the substitution used to bind the edge's variables after the shift. Both ν_{min} and σ_{shift} are computed at compile time.

Assume that during the superposition of a pattern P on a sequence T a mismatch occurs at position l of P. If $(length, \nu_{min}, \sigma_{shift})$ is an edge of $P[0].\cdots.P[l-1]$ and ν_{min} is compatible with the runtime valuation ν, then we can shift P of $l - length - 1$ symbols to the right and restart the matching process at position $length + 1$ for P (see Figure 4). The new runtime valuation is $\nu \circ \sigma_{shift}$.

To illustrate this, let the subpattern be `@x.b.@y.c.@z.@x.a.d`. There exists an edge $e(4, \nu_{min}, \sigma_{shift})$ of length 4 with $\nu_{min} = \{\texttt{@x/b}, \texttt{@current/c}\}$ and $\sigma_{shift} = \{\texttt{@x/@z}, \texttt{@y/a}\}$

This is interpreted as follows. If a mismatch occurs with the last symbol of P, a shift of size 4 can be performed if the following conditions hold: (i) `@x` is bound to `b` and (ii) `@current` is bound to `c`.

Then the prefix `@x.b.@y.c` can be superposed with `@z.@x.a.@current`. Since `@x` replaces `@z`, it takes the value assigned to `@z` by the runtime valuation, while `@y` takes always the value `a`. One easily verifies that: $\nu_{min}(\sigma_{shift}(\texttt{@x.b.@y.c})) = \nu_{min}(\texttt{@z.@x.a.@current}) = \texttt{@z.b.a.c}$.

Computing the edges

A brute-force technique for determining the edges of P consists in considering, for each subpattern $P[0 \ldots m-1]$, all the possible edges of length $i < m$. This algorithm runs in $O(m^3)$ (number of comparisons). A better algorithm uses the edges already determined at a given position $l-1$ to deduce the edges at position l. The algorithm relies on the following property (see Figure 8):

Lemma 1 *Let $e = (i+1, \nu_{min}, \sigma_{shift})$ be an edge for $P[0 \ldots l]$. Then there exists an edge $e' = (i, \nu'_{min}, \sigma'_{shift})$ for $P[0 \ldots l-1]$ with*

$$\begin{cases} \nu'_{min} = \nu_{min|_{var(P[l-i \ldots l-1])}} \cup \{\texttt{@current}/P[i-1]\} \\ \sigma'_{shift} = \sigma_{shift|_{var(P[0 \ldots i-1])}} \end{cases}$$

where $\sigma_{|_{var(q)}}$ (resp. $\nu_{min|_{var(q)}}$) denotes the restriction of σ (resp. ν_{min}) to the variables in q.

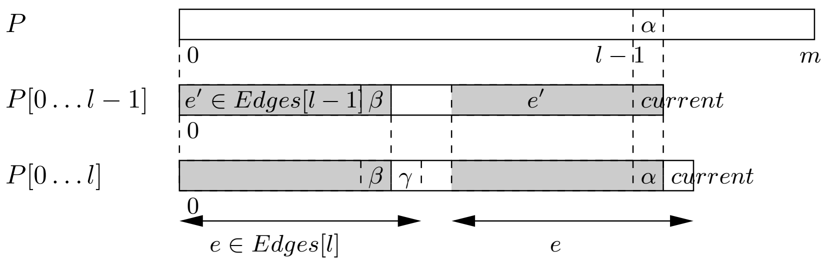


Fig. 8. Computation of *Edges*$[l-1]$ from *Edges*$[l]$

Proof: Let $e = (i+1, \nu_{min}, \sigma_{shift})$ be an edge of $P[0 \ldots l]$. From the definition of an edge, $\nu_{min}(\sigma_{shift}(P[0 \ldots i])) = \nu_{min}(P[l-i \ldots l]) = \omega.\alpha$, with $\omega.\alpha \in (\Sigma \cup \mathcal{V})^* \times (\Sigma \cup \mathcal{V})$. Therefore we have $\nu_{min}(\sigma_{shift|_{var(P[0 \ldots i-1])}}(P[0 \ldots i-1])) = \nu_{min|_{var(P[l-i \ldots l-1])}}(P[l-i \ldots l-1]) = \omega$

In other words, let $\nu'_{min} = \nu_{min|_{var(P[l-i \ldots l-1])}} \cup \{\texttt{@current}/P[i-1]\}$ and let $\sigma'_{shift} = \sigma_{shift|_{var(P[0 \ldots i-1])}}$ then $(i-1, \nu'_{min}, \sigma'_{shift})$ is an edge e' for the pattern $P[0 \ldots l-1]$. □

This lemma leads to an optimized algorithm EdgesConstruction that iteratively constructs the table of edges, deducing *Edges*$[l]$ from the edges in *Edges*$[l-1]$. Each step of the algorithm is of the form:

```
EdgesConstruction (step $l$)
$Edges[l] := (0, \emptyset, \emptyset)$
for each $(i, \nu_{min}, \sigma_{shift}) \in Edges[l-1]$
    // Initialize $\nu'_{min}$ and $\sigma'_{shift}$
    $\nu'_{min} := \nu_{min}; \sigma'_{shift} := \sigma_{shift}$
    Remove from $\nu'_{min}$ and $\sigma'_{shift}$ the conditions and substitutions on @current.
```

```
        // Set the new constraints linked to @current
        if $P[i] \in \Sigma$ then $\nu'_{min} := \nu_{min} \cup \{\texttt{@current/P[i]}\}$
        else $\sigma'_{shift} := \sigma_{shift} \cup \{P[i]/\texttt{@current}\}$
        endif
        if $P[i-1] \in \Sigma$ then
            if $P[l-1] \in \Sigma$ and $P[l-1] = P[i-1]$ then $(i+1, \nu'_{min}, \sigma'_{shift}) \in Edges[l]$
            else if $P[l-1] \in \mathcal{V}$ then
                // if there is no substitution $P[l-1]/P[i-1]$, add it to $\nu'_{min}$
                if $P[l-1]/P[i-1] \in \nu_{min}$ then $(i+1, \nu'_{min}, \sigma'_{shift}) \in Edges[l]$
                else $(i+1, \nu'_{min} \cup \{P[l-1]/P[i-1]\}, \sigma'_{shift}) \in Edges[l]$
                endif
            endif
        else // $P[i-1]$ is a variable, modify the substitution if needed
            if $P[i-1]/P[l-1] \in \sigma_{shift}$ then $(i+1, \nu'_{min}, \sigma'_{shift}) \in Edges[l]$
            else $(i+1, \nu'_{min}, \sigma'_{shift} \cup \{P[i-1]/P[l-1]\}) \in Edges[l]$
            endif
        endif
    endfor
```

Consider the pattern `a.@x.b.a.@x.@y` and assume that we already computed the edges at position 4: $Edges[4] = \{(0, \emptyset, \emptyset), (1, \{\texttt{@current/a}\}, \emptyset), (2, \emptyset, \{\texttt{@x/@x}\})\}$. For each of these edges of resp. length 0, 1 and 2, we compute the possible edges of length $1, 2$ and 3 and add the default edge $(0, \emptyset, \emptyset)$.

- $(0, \emptyset, \emptyset) \in Edges[4]$ *and* $P[5] \in \mathcal{V} \;\Rightarrow (1, \{\texttt{@current/a}\}, \emptyset) \in Edges[5]$
- $(1, \{\texttt{@current/a}\}, \emptyset) \in Edges[4]$ *and* $(P[0], P[4]) \in \Sigma \times \mathcal{V}$ *and* $P[1] = \texttt{@x}$ $\Rightarrow (2, \{\texttt{@x/a}\}, \{\texttt{@x/@current}\}) \in Edges[5]$
- $(2, \emptyset, \{\texttt{@x/@x}\}) \in Edges[4]$ *and* $(P[1], P[4]) \in \Sigma^2$ *and* $P[2] = \texttt{b}$ $\Rightarrow (3, \{\texttt{@current/b}\}, \{\texttt{@x/@x}\}) \in Edges[5]$

Finally we add the default edge $(0, \emptyset, \emptyset)$ to $Edges[5]$.

Proposition 1 EdgesConstruction *is in the worst case quadratic in the size of the pattern.*

Proof: First note that in the worst case, we have l edges at the position l of a pattern P, with lengths ranging from 0 to $l-1$. Algorithm EdgesConstruction computes an edge of $Edges[l]$ with length $i+1$ from an edge of $Edges[l-1]$ with length i if a matching between $P[l]$ and $P[i]$ is successful. So the set $Edges[l]$ is computed from the set $Edges[l-1]$ by carrying out one comparison for each edge of $Edges[l-1]$ (we also add the default edge to $Edges[l]$). Consequently in the worst case l comparisons are necessary, which leads to an amount of $\sum_{l=1}^{m-1} l = \frac{(m-1)(m-2)}{2}$ comparisons for the whole table of edges. □

Finally, given a query $q(P, \mathcal{C})$, the computation of edges must take into account the constraints $\mathcal{C}$. Once the table of edges is built as described above, any edge whose variables binding is incompatible with one of the query constraints must be removed. We do not elaborate further on the constraint language, since constraint evaluation is clearly indenpendent of the core of our algorithm.

Representation of edges

The basic idea for the edge representation consists in building two structures. The first one is a set of arrays of edges, one for each position in the pattern, that stores all the existing edges at that position. The second one describes the association between the variables valuations and the edges.

More formally, let $L_l < l$ be the number of edges at position l in the pattern P. The edges are stored in an array $Edges[l] = [e^l_0, e^l_1, \cdots, e^l_{L_l}]$, sorted on their length in ascending order. Let P_l denote the prefix of P until position l: $P_l = P[0]. \cdots .P[l]$. In addition we maintain, for each l and for each variable v in $P_l \cup \{\texttt{@current}\}$, an array of bits f^l_v. Given a symbol α in P_l and $k \leq L_l$, the entry $f^l_v[\alpha, k]$ is initialized as follows.

$$f^l_v[\alpha, k] = \begin{cases} 1 \; if \;\; \nexists(@x_j/\beta) \in e^l_k.\nu_{min} \; with \; \beta \neq \alpha, \\ 0 \; otherwise \end{cases}$$

Recall an edge is applicable if its valuation is compatible with the run time valuation. This happens if all variable bindings in the edge valuation are in the run time valuation. We set entry $f^l_v[\alpha, k]$ to 1 if the binding of v to k is compatible with the edge e^l_k. Since a variable in ν_{min} may only be bound to a symbol of P, all the $f^l_v[\alpha, k]$ for $\alpha \in \Sigma - symb(P)$ are identical. Consequently for each variable v, we need to maintain at most $|symb(P)| + 1$ entries, one for each $\alpha \in symb(P)$, and one that represents the compatibility for all the symbols in $\Sigma - symb(P)$.

Here are the edges of the pattern $P = \texttt{a.@x.b.a.@x.@y.a}$, to be used when a comparison fails with $P[6] = \texttt{a}$.

- $e^6_0 = (0, \emptyset, \emptyset)$ the default edge that corresponds to a shift of the whole pattern;
- $e^6_1 = (2, \{\texttt{@y/a}\}, \{\texttt{@x/@current}\})$, because if ν is compatible with $\{\texttt{@y/a}\}$, then the end of the sequence is of the form `a.@current`; it matches the beginning of P, `a.@x`, `@x` being bound to `@current`.
- $e^6_2 = (3, \{\texttt{@x/a}, \texttt{@current/b}\}, \{\texttt{@x/@y}\})$, because if ν is compatible with $\{\texttt{@x/a}, \texttt{@current/b}\}$, the end of the sequence is of the form $\texttt{a.}\nu(\texttt{@y})\texttt{.b}$. It matches the first three symbols of P, `a.@x.b`, the new binding of `@x` being

the former binding of `@y`.

There is no edge of length 4, because when a mismatch occurs, `@current` is not `a`. One cannot find edges of length 5 or 6. The array of bits for these edges are:

$$f^6_{\texttt{@x}}[\alpha] = \begin{cases} [1 \;\; 1 \;\; 1] \; for \; \alpha = \texttt{a} \\ [1 \;\; 1 \;\; 0] \; otherwise \end{cases} \qquad f^6_{\texttt{@y}}[\alpha] = \begin{cases} [1 \;\; 1 \;\; 1] \; for \; \alpha = \texttt{a} \\ [1 \;\; 0 \;\; 1] \; otherwise \end{cases}$$

$$f^6_{\texttt{@current}}(\alpha) = \begin{cases} [1 \;\; 1 \;\; 1] \; for \; \alpha = \texttt{b} \\ [1 \;\; 1 \;\; 0] \; otherwise \end{cases}$$

The matching process relies on this array of bits to efficiently determine which edge must be applied when a mismatch occurs. Basically, when a mismatch is detected at position l, we consider all the variables v of P_l and perform a logical AND on the bit arrays $f^l_v[\nu(v)]$. The bits set to 1 indicate which edges can be applied for the run time valuation. We choose of course the longest edge. Since the edges are sorted in ascending order on their length, the rightmost bit set to 1 gives the shift that must be performed.

With the previous example, assume that a mismatch occurs at position 5, with the following runtime valuation: $\nu = \{\texttt{@x/a}, \texttt{@y/c}, \texttt{@current/b}\}$. In order to choose the edge, the following AND is performed:

$$E = f^6_{\texttt{@x}}(\texttt{a}) \wedge f^6_{\texttt{@y}}(\texttt{c}) \wedge f^6_{\texttt{@current}}(\texttt{b}) = [1 \;\; 1 \;\; 1] \wedge [1 \;\; 0 \;\; 1] \wedge [1 \;\; 1 \;\; 1] = [1 \;\; 0 \;\; 1]$$

The right-most 1 in E is at the third position; edge e^6_2 must be chosen.

4.4 The MATCH *algorithm*

The evaluation of a pattern P maintains a buffer (or "sliding window") B containing the last $|P|$ symbols of the input sequence T. When a match is found, the content of B constitutes the result output to the user. Note that during a matching attempt at position i, the run-time valuation ν of a variable with position k is the symbol $T[i + k]$, which can be found directly in B.

The following MATCH algorithm is invoked when a new symbol is read and assigned to the special variable `@current`. It takes as an input the current position l in P and returns the new position l' in P. If a match is found, the substring that matches P is output.

```
MATCH($l$)
Input: $l$, the current position in $P$
Output: $l'$, the next position in $P$
begin
    $mismatch := false$; $match := false$;
    if $(l < m)$ then
        if $(P[l] \in \mathcal{V}$ and $P[l] \notin bound(\nu))$ then // $P[l]$ is a variable not yet bound
            $l' := l + 1$
        else if $(P[l] = \nu(\texttt{@current})$ or $\nu(P[l]) = \nu(\texttt{@current}))$ then
            // $P[l]$ is equal to (or already bound to) $\nu(\texttt{@current})$
            $l' := l + 1$
        else
            $mismatch := true$ // there is a mismatch
        endif
    endif
    // If the whole pattern has been matched, the attempt is successful
    if $(l' = m)$ then
        $match := true$; output the content of buffer $B$;
    endif
    // Mismatch or match : in both cases perform the shift
    if $(mismatch$ or $match)$ do
        // Get the right edge
        $E := f^{l}_{\texttt{@current}}(\nu(\texttt{@current}))$
        for each $(\texttt{@x}_i/\alpha \in \nu)$ do $E := E \wedge f^{l}_{@x_i}(\alpha)$
        $e :=$ the edge that corresponds to the rightmost 1 in $E$
        // Right shift of the pattern
        $l' := e.length + 1$
    endif
    return $l'$
end
```

Complexity analysis

Let us first examine the required storage. The main structure is the set of bit arrays used to determine the shift. Given a position l, let $V \leq l$ be the number of variables and $l-V$ the number of constant symbols in $P[0].\cdots.P[l-1]$. For each variable we represent its compatibility with at most l edges as an array of size $(l - V + 1) \times \lceil \frac{l}{w} \rceil$ where w denotes the size of a machine word. The size of the arrays for a position l can be estimated to be $V \times (l - V + 1)\lceil \frac{l}{w} \rceil$, that is, at worse $(\frac{l}{2})^2 \lceil \frac{l}{w} \rceil$. However for a large number of applications, m is smaller than the word size w. Then $\lceil \frac{l}{w} \rceil = 1$ and the size of the arrays for a position is effectively at worse $(\frac{l}{2})^2$. Therefore the space required for storing this set of arrays over all the positions in the pattern of size m is $\sum_{l=0}^{m} (\frac{l}{2})^2 = \frac{m.(m+1)(2m+1)}{24}$, thus in $O(m^3)$. Note that this is independent both from the alphabet and from the input sequence.

We now show that the time complexity is linear in the sequence size n. We estimate the average time complexity by taking into account the *filtering rate* of a pattern which gives the probability to match $P[0.\cdots.l-1]$ after reading l symbols. We assume a uniform distribution of symbols from Σ in a sequence (our model does not depend on this assumption, but a more precise distribution law is application-dependent). Each constant symbol can be found with probability $\frac{1}{|\Sigma|}$. As far as variables are concerned, the first occurrence of a variable has no impact on the filtering rate because any symbol will match. All the other occurrences must be bound to the same value in Σ. In summary we estimate the *selectivity* of a symbol at position l in a pattern $P[0\dots m]$ by:

$$\tau_P(l) = \begin{cases} \frac{1}{|\Sigma|} \; if \;\; \alpha \in \Sigma \cup var(P[0\dots l-1]) \\ 1 \; otherwise \end{cases}$$

The filtering rate is then estimated by the following formula:

$$\tau(P) = \prod_{k=0}^{m} \tau_P(k)$$

When Match receives a new symbol, the probability to be at position l on the pattern is $\tau(P[0\dots l-1])$ and the mismatch probability at that position is $1-\tau_P(i)$. In case of mismatch, we must perform as many AND operations as there are distinct variables in $P[0].\cdots.P[l-1]$. The cost of AND depends on the size w of a computer word and can be estimated as $\left\lceil \frac{l}{w} \right\rceil$ (recall that there are at most l edges at position l).

The average cost for processing a symbol involves at least one comparison, and the possible computation of a shift. It can be estimated to be:

$$1 + \sum_{l=1}^{m} \tau(P[0\dots l-1]) \times (1-\tau_P(l)) \times k_l \times \left\lceil \frac{l}{w} \right\rceil + \tau(P[0\dots m]) \times k_m \times \left\lceil \frac{m}{w} \right\rceil$$

where k_l denotes the number of distinct variables in $P[0].\cdots.P[l]$. The last term corresponds to a successful matching.

The worst case corresponds to a pattern which only consists of distinct variables. In that case $\tau(P[0\dots l-1]) = 1$ and $(1-\tau(P[l])) = 0$ for $0 < l \leq m$. The worst-case cost for processing a symbol is $k \times \left\lceil \frac{m}{w} \right\rceil$, and $n \times k \times \left\lceil \frac{m}{w} \right\rceil$ for a sequence of length n. If the pattern is small with respect to the size of a word, the time complexity is $O(n \times k)$.

5 Multiple patterns evaluation

With this algorithm at hand, we can turn our attention to the evaluation of multiple patterns over a large input sequence.

5.1 Patterns containment

Patterns are equivalent up to a renaming of variables, e.g., `b.@x.a` $\equiv$ `b.@y.a`. Note also that adding at the end of a pattern P a variable which does not already appear in P yields an equivalent pattern, i.e., `b.@x.a` $\equiv$ `b.@x.a.@y`. Indeed, since `@y` can be bound to any symbol, a sequence that matches `b.@x.a` matches `b.@x.a.@y` as well, and conversely [1] . We use as equivalence class representatives *normalized patterns.* Let the *marking* of a pattern P be the pattern where each variable v of $var(P)$ is replaced by `@x` marked by a subscript over $\mathbb{N}$, representing the rank of the first occurrence of v in P. A pattern P is *normalized* iff the following conditions hold: (i) $P = marking(P)$, and (ii) if P is of the form P'.`@x`, then `@x` $\in var(P')$. It is straightforward that for any pattern P, there exists a unique equivalent normalized pattern. For instance the normalized pattern of `@y.c.@x.@y.@z` is $\texttt{@x}_1.c.\texttt{@x}_2.\texttt{@x}_1.\texttt{@x}_3$.

The containment relation holds on equivalent classes of patterns. A pattern P_1 contains a pattern P_2 if any sequence S matching P_2 at position l matches also P_1 at l. This means first that $|P_1| \leq |P_2|$, and second that there exist two valuations ν_1 and ν_2 such that both $\nu_1(P_1)$ and $\nu_2(P_2)$ are substrings of S starting at l. Therefore:

Definition 4 *A pattern P_1* contains *a pattern P_2, denoted $P_2 \trianglelefteq P_1$, iff for each valuation ν_2, there exists a valuation ν_1 such that $\nu_1(P_1)$ is a prefix of $\nu_2(P_2)$.*

Syntactically, containment means tighter matching constraints. There exists several ways of doing so: by suffixing a pattern with some constant symbols, by replacing a variable with a constant symbol, and finally by binding the variables.

It follows that the containment of P_2 in P_1 can be decided by a comparison of the patterns' symbols from left to right. At each position one checks that the current symbol from P_1 is a "relaxation" of the corresponding symbol of P_2, according to the following rules: (i) a constant symbol can be relaxed to itself or to a variable; (ii) a variable can be relaxed to another variable whose subscript is greater or equal.

[1] We consider unbounded sequences, thus the length of a pattern is never an issue.

Consider again $P_1 = \texttt{a.@x}_1\texttt{.b}$ and $P_2 = \texttt{a.c.b}$ Then, `a` can be relaxed to `a`, `c` to $\texttt{@x}_1$ and `b` to `b`: $P_2 \trianglelefteq P_1$. The second condition ensures that no problem comes from the multiple occurrences of a same variable. Take for instance $P_1 = \texttt{a.@x}_1\texttt{.b.@x}_1$ and $P_2 = \texttt{a.@x}_1\texttt{.b.@x}_2$. The variable $\texttt{@x}_2$ in P_2 cannot be relaxed to the second occurrence of $\texttt{@x}_1$ in P_1. However the second occurrence of $\texttt{@x}_1$ in P_1 can be relaxed to $\texttt{@x}_2$. Indeed, in that case, $P_1 \trianglelefteq P_2$.

It is easily verified that $\trianglelefteq$ is a partial order on $(\Sigma \cup \mathcal{V})^*$. Furthermore, it can be shown that two patterns have a unique minimal common ancestor or least upper bound (*lub*) with respect to relation $\trianglelefteq$.

Proposition 2 *The set of patterns over $(\Sigma \cup \mathcal{V})$ ordered by $\trianglelefteq$ is an upper semi-lattice.*

Proof: Let P_1 and P_2 be two patterns and $\mathcal{U}$ be the set of (normalized) patterns P containing P_1 and P_2 : $P_1 \trianglelefteq P$ and $P_2 \trianglelefteq P$. $\mathcal{U}$ is non empty, because it contains at least the pattern $P_{max} = \texttt{@x}_1$. Assume that we have two distinct lubs in $\mathcal{U}$, P_{lub_1} and P_{lub_2}, and let $l = min(|P_{lub_1}|, |P_{lub_2}|)$. Then, for each $i \in [0, l-1]$,

- either $P_{lub_1}[i] \in \Sigma$, and there exists $k \in \{1,2\}$ such that $P_k[i] = P_{lub_1}[i]$. Therefore, either $P_{lub_2}[i] = P_k[i] = P_{lub_1}[i]$ or $P_{lub_2}[i] \in \mathcal{V}$. If $P_{lub_2}[i] \in \mathcal{V}$ we can build a new pattern P'_{lub_2} by replacing $P_{lub_2}[i]$ by $P_k[i]$ in P_{lub_2}. This pattern belongs to $\mathcal{U}$ and satisfies $P'_{lub_2} \trianglelefteq P_{lub_2}$, so P_{lub_2} is not a lub. A similar reasoning holds for P_{lub_1}.
- or $P_{lub_1}[i] \in \mathcal{V}$, and $P_{lub_2}[i]$ also belongs to $\mathcal{V}$. If there is another occurrence of $P_{lub_1}[i]$ in P_{lub_1}, it must be the same for $P_{lub_2}[i]$ otherwise $P_{lub_2} \trianglelefteq P_{lub_1}$.

It follows that P_{lub_1} and P_{lub_2} are equal. □

The following algorithm computes the *lub* of two patterns. Function NewVar returns a variable name that has not yet been used. Subst(d_1, d_2, r) is a set of substitutions. A triple (d_1, d_2, r) associates a pair of symbols (d_1, d_2) from, respectively, P_1 and P_2, with a symbol r in the lub.

```
LUB($P_1[0 \dots m_1], P_2[0 \dots m_2]$)
Input: two patterns $P_1$ and $P_2$
Output: the lub of $P_1$ and $P_2$
begin
    SUBST := $\emptyset$; $m := min(m_1, m_2)$ // $m$ is the maximal length of the lub
    for ($k := 0$ to $m$) do
        if ($\exists x, P_1[k], P_2[k], x) \in$ SUBST) then $lub[k] := x$
            else if ($(P_1[k], P_2[k]) \in \Sigma^2$ and $P_1[k] = P_2[k]$) then $x := P_1[k]$
                else $x :=$ NEWVAR()
        SUBST := SUBST $\cup \{(P_1[k], P_2[k], x)\}$
        endif
        $lub[k] := x$
    **endfor**
    **return** NORMALIZE($lub$)
**end**
```

The final normalization is necessary to remove the useless variables which may appear in the suffix.

5.2 *Filtering*

From the previously defined interpretation of the containment relation, If P_1 and P_2 are two patterns, then the set of sequences matched by LUB(P_1, P_2) is a superset of the sequences respectively matched by P_1 and P_2. Then, we can construct hierarchies of patterns ordered by containment, and use the root of each subtree as a filter of the patterns contained in the subtree.

In principle it would be possible to maintain the transitive reduction of the containment graph but its construction turns out to be too costly[2]. Moreover this is not necessary since it is sufficient to maintain one and only one path from the root to any node. This guarantees that any pattern can be reached, if needed.

Our goal is thus to construct a spanning tree of the containment graph such that the average evaluation cost is minimized. Intuitively, the minimization implies choosing the most filtering paths. Let us look at Figure 9, assuming an alphabet with only four symbols `a`, `b`, `c` and `d`. The pattern set is $\{$`@x`$_1$, `a.@x`$_1$`.b`, `@x`$_1$`.c`, `a.b.c.d`$\}$. The transitive reduction of the graph is shown at the top of the figure, and there exists two possible spanning trees, shown in the bottom part.

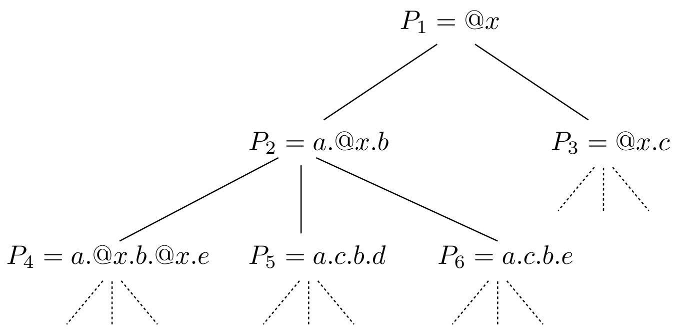


Fig. 9. Issues in patterns tree construction

[2] The computation of the transitive reduction runs in $O(n)$ for a graph of size n, for each node insertion [17]. Therefore one might, in the worst case, need to update all the nodes of the graph when a new pattern is inserted.

In order to decide which solution minimizes the evaluation cost, we assign to each out-edge of a node N a *weight* equal to the filtering rate of N. The problem reduces now to find the minimum spanning tree, *i.e.*, the connected subgraph containing all the nodes such that the sum of the edge weights is minimal [18]. From our application point of view, this means that for each pattern P one keeps the path to P which is the most filtering one in the graph of the containment relation. In the case of Figure 9, the bottom left choice is the best one. Intuitively, only $\frac{1}{16}$ of the matching attempts will have to evaluate the pattern `a.c.b.d`, instead of $\frac{1}{4}$ if the solution of the right part were adopted.

5.3 The clustering algorithm

Since subscriptions (patterns) may be added or removed dynamically, one must maintain incrementally a tree T of patterns. However, maintaining incrementally the optimal solution results in a very significant cost (in the worst case, the whole tree must be reconstructed). We propose an algorithm which delivers a non-optimal solution but runs quite efficiently. Our experiments show that it still provides an effective reduction of the overall evaluation cost with respect to the trivial solution that evaluates separately each of the submitted patterns.

The insertion of a new pattern P is performed in two steps:

- **Candidate parent selection**.
  A node N in T is a *candidate parent* for P if the following conditions hold: (i) $P \lhd N$, and (ii) for each child N' of N, $P \not\lhd N'$, i.e., P is strictly contained in N but does not contain any child of N.
  The algorithm performs a depth-first search to seek for a candidate parent. Starting from the root, one chooses at each level the most selective child which contains P. When such a child no longer exists the candidate parent is found. Note that this is an heuristic which avoids to follow an unbounded number of paths in the tree, but does not guarantee that the "best" candidate parent (i.e., the most selective one) is found.
- **Lub selection**.
  Once the candidate parent N is found, the second step inserts P as a child or grandchild of N as follows. First, for each child N' of N, one computes $lub(P, N')$ and keeps only those lubs which are strictly contained in N. Now:

(1) if at least one such lub $L = lub(P, N')$ has been found, the most selective one is chosen, and a new subtree $L(P, N')$ is inserted under N;
(2) else P is inserted as a child of N.

Let us take as an example the left tree of Figure 10. The pattern $P =$ `b.a.c.a`

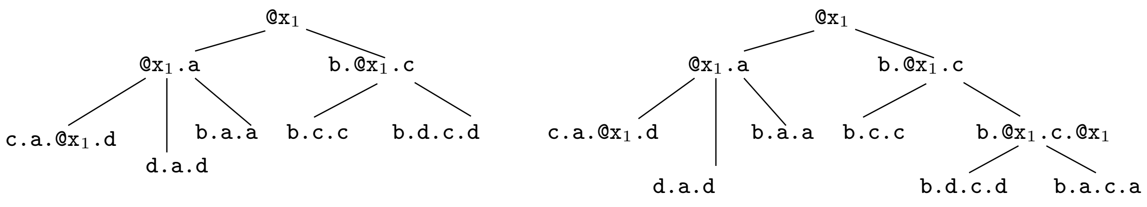


Fig. 10. Insertion of the pattern b.c.a

must be inserted. First one checks P against the root node. Since $P \lhd \mathtt{@x}_1$, we consider the two children of the root. P is strictly contained in both $\mathtt{@x}_1\mathtt{.a}$ and $\mathtt{b.@x}_1\mathtt{.c}$ which constitute therefore two possible paths. Since $\tau(\mathtt{@x}_1\mathtt{.a}) = \frac{1}{|\Sigma|}$ and $\tau(\mathtt{b.@x}_1\mathtt{.c}) = \frac{1}{|\Sigma|^2}$, the second is chosen. None of its children contain P, therefore $\mathtt{b.@x}_1\mathtt{.c}$ is the candidate parent.

Next one determines the lubs of P with each child of $\mathtt{b.@x}_1\mathtt{.c}$, and keeps those which are strictly contained in $\mathtt{b.@x}_1\mathtt{.c}$. For instance $lub(\mathtt{b.a.c.a}, \mathtt{b.c.c}) = \mathtt{b.@x}_1\mathtt{.c}$, so it is not kept. However $lub(\mathtt{b.a.c.a},\ \mathtt{b.d.c.d}) = \mathtt{b.@x}_1\mathtt{.c.@x}_1$ is a candidate lub. One finally obtains the tree of the right part of Figure 10.

It may happen that several equivalent choices are possible. Assume for instance that the pattern $P' = \mathtt{b.a.d}$ is inserted. The candidate parent is $\mathtt{@x}_1\mathtt{.a}$, and one must choose between $L_1 = lub(P', \mathtt{d.a.d}) = \mathtt{@x}_1\mathtt{.a.d}$ and $L_2 = lub(P', \mathtt{b.a.a}) = \mathtt{b.a}$. Both have the same filtering rate. The tie-breaking procedure compares the prefixes of length $l < min(|L_1|, |L_2|)$, starting with $l = 1$. As soon as one of the prefixes is found to be more selective than the other one, the corresponding lub is chosen. The rationale is that a mismatch will occur more quickly, and that less comparisons are necessary. In our example, since $\mathtt{@x}_1$ is a less filtering prefix than b we create the lub $lub(P', \mathtt{b.a.a}) = \mathtt{b.a.@x}_1$.

The insertion algorithm follows a single path from the root to a node in the pattern tree. At each level, a comparison must be carried out with each child. Its complexity is determined by the following properties.

Proposition 3 *The depth of a pattern tree T is bound by $l+1$ where l denotes the size of the longest pattern in T. Each internal node N of the tree has at most $|\Sigma| \times |var(N)|$ children.*

Proof: Assume that the size of longest pattern is l, and that there exists a node N of length $l' \leq l$ whose depth is $l+2$. Since N is contained in its parent N', N' relaxes some constraints and/or is shorter. There cannot be more than $l'+1$ relaxation operations to reach the root of the tree from l. Therefore its depth is at most $l' \leq l+1$.

Next, consider the pattern equivalent to the parent whose size is equal to the longer child. It is obtained by adding free variables until we reach the appropriate length. We note then that two children can not replace a given free variable of their parent by the same constant symbol, since in that case a lub would have been created for these children. So for the first free variable $\texttt{@x}_1$ of the lub we can have only $|\Sigma|$ different patterns that instantiate $\texttt{@x}_1$, each of them with a different instantiation for $\texttt{@x}_1$, Consider now the patterns that do not instantiate $\texttt{@x}_1$. We can iterate the same reasoning for the second free variable $\texttt{@x}_2$ and so on until the last free variable of the non-normalized parent. □

5.4 Multi-pattern evaluation

The evaluation mechanism associates one automaton with each pattern in the pattern tree. These automata may be *active* or *inactive*, according to the following rules:

(1) Initially the automaton $\mathcal{A}_{root}$ associated with the root is active; others automata are inactive.
(2) When a matching is found for an automaton $\mathcal{A}_P$, all the inactive automata associated with the sons of P become active; $\mathcal{A}_P$ itself becomes inactive.
(3) When a matching fails for some automaton $\mathcal{A}_P$, its parent becomes active and $\mathcal{A}_P$ becomes inactive.

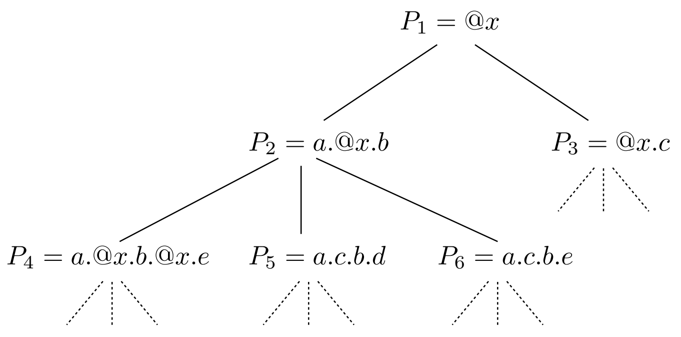


Fig. 11. Evaluation over a pattern tree

Consider the tree of Fig. 11 and the sequence `a.c.b.c.a.d.b.`$\cdots$. Each pattern P_i is associated with an automaton $\mathcal{A}_{P_i}$. Initially the automaton $\mathcal{A}_{P_1}$ is the only one active. Figure 12 illustrates the successive activation status.

The first symbol of the sequence matches P_1. Hence $\mathcal{A}_{P_2}$ and $\mathcal{A}_{P_3}$ become active in turn (Figure 12(b)). The next symbol is `c`: a matching is found with P_3. $\mathcal{A}_{P_3}$ becomes inactive, while the automata of its sons become active (Figure 12(c)). Let us now focus on the subtree rooted at P_2. When the third symbol, `b`, is read from the sequence, a matching is found. $\mathcal{A}_{P_2}$ becomes inactive, $\mathcal{A}_{P_4}$, $\mathcal{A}_{P_5}$ and $\mathcal{A}_{P_6}$ become active (Figure 12(d)).

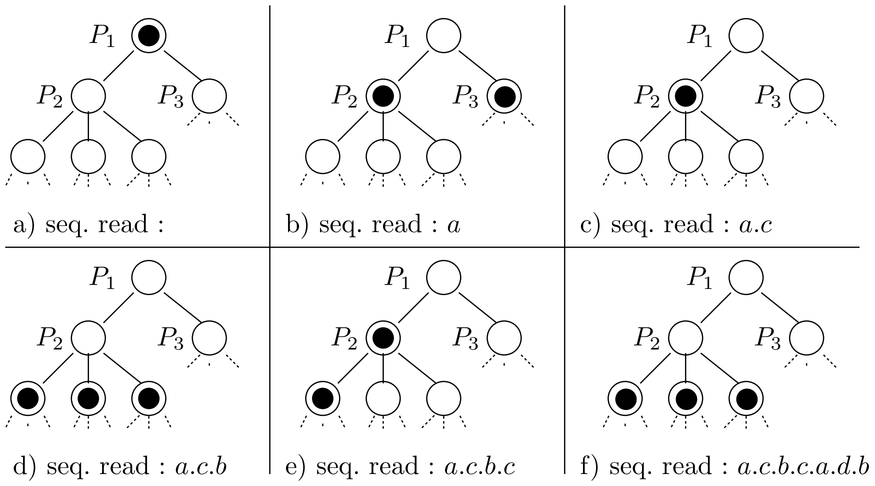


Fig. 12. Active patterns during the evalution of $a.c.b.c.a.d.b$

When the fourth symbol, c, is received, the matching attempt fails for P_5 and P_6 (Figure 12(e)). The evaluation proceeds then as follows:

- P_4 is matched by the sequence; notifications are sent and the automaton $\mathcal{A}_{P_4}$ remains active;
- $\mathcal{A}_{P_5}$ and $\mathcal{A}_{P_6}$ become inactive but $\mathcal{A}_{P_2}$, their common parent, becomes active.

Finally after receiving `a.d.b`, $\mathcal{A}_{P_2}$ reaches a successful state and activates again $\mathcal{A}_{P_5}$ and $\mathcal{A}_{P_6}$ (Figure 12(f)). In that case a mismatch occurs at once because the second symbol is `d` whereas both $\mathcal{A}_{P_5}$ and $\mathcal{A}_{P_6}$ expect a `c`. Actually, when several sibling patterns fail, their common prefix is represented by their parent. Making active the parent is a way to "factorize" the query evaluation for this prefix, but this does not guarantee that any of its children will succeed.

6 Experiments

In order to validate our approach, we have implemented and compared two algorithms in Java on a Pentium 4 processor (3GHz) with 1,024MB of memory. The first algorithm, Naïve, is the naïve one described in Section 4.1, which shifts the pattern only one symbol to the right whenever a mismatch occurs. The other one is the extended KMP algorithm which uses the table of edges. We evaluate their respective performance with respect to the following parameters:

- the length of the patterns;
- the number of variables in each pattern;
- the size of the alphabet.

		number of variables within the pattern					
		0	1	2	3	4	5
$\lvert\Sigma\rvert = 2$	Naïve	2.01	2.19	2.38	2.78	3.08	3.64
	Match	1.00	1.12	1.30	1.49	1.65	1.85
$\lvert\Sigma\rvert = 4$	Naïve	1.33	1.44	1.59	1.84	2.20	2.39
	Match	1.00	1.09	1.18	1.37	1.53	1.79
$\lvert\Sigma\rvert = 10$	Naïve	1.11	1.22	1.32	1.51	1.82	2,00
	Match	1.00	1.10	1.17	1.36	1.51	1.68
$\lvert\Sigma\rvert = 30$	Naïve	1.04	1.14	1.22	1.44	1.67	1.78
	Match	1.00	1.09	1.18	1.31	1.52	1.73

Fig. 13. Number of operations versus number of variables ($\times 10^6$), for various alphabet sizes; pattern length: 10

Our experimental setting includes a random text of 1 000 000 symbols, a pattern size with a length between 5 and 20, and four alphabet sizes, ranging from 2 to 30. The average performance was taken over a set of 500 queries to avoid results biased by a peculiar pattern aspect (all the variables at the beginning or at the end for instance). The evaluation of the two algorithms is based on the total number of operations (comparisons and AND operations). Note that the length of the patterns chosen in the experiments is lower than a machine word size (nowadays, mostly 32 or 64 bits). Nonetheless a performance degradation is expected for long patterns.

6.1 Evaluation of the extended KMP algorithm

We successively study the impact of the alphabet size and that of the pattern length. Figure 13 illustrates the impact of the number of variables in the pattern on the total number of operations, for different alphabet sizes. As expected, our extended KMP algorithm always outperforms the naïve one. The gain decreases with the size of the alphabet. For 5 variables for instance, the gain is 50% with an alphabet size of $|\Sigma| = 2$, while it reaches 30% for $|\Sigma| = 4$, 10% for $|\Sigma| = 10$, and only 3% with an alphabet of 30 symbols. This result was expected since our algorithm relies on KMP that performs better for small size alphabets [8]. Indeed, with large alphabets, there is a high probability to have a mismatch on the first symbol of the pattern, thus both Naïve and Match realize a single comparison, then shift by one. Small size alphabets, on the opposite, allow large partial matchings before a mismatch. In

		variables rate within the pattern					
		0%	10%	20%	30%	40%	50%
$m = 5$	NAÏVE	1.97	2.16	2.34	2.58	2.82	3.16
	MATCH	1.00	1.14	1.29	1.48	1.66	1.88
$m = 10$	NAÏVE	2.01	2.19	2.38	2.78	3.08	3.64
	MATCH	1.00	1.12	1.30	1.49	1.65	1.85
$m = 15$	NAÏVE	2.00	2.16	2.35	2.76	3.06	3.59
	MATCH	1.00	1.11	1.27	1.49	1.62	1.86
$m = 20$	NAÏVE	2.00	2.21	2.44	2.72	3.11	3.68
	MATCH	1.00	1.13	1.33	1.43	1.65	1.88

Fig. 14. Number of operations versus percentage of variables in the pattern for different lengths of pattern ($\times 10^6$), alphabet size=2

that case our algorithm determines the best shift and avoids all the redundant comparisons performed by NAÏVE.

Figure 13 illustrates also the impact of variables on the processing cost, both for NAÏVE and MATCH. Without variable, our algorithm is standard KMP (a single operation per input symbol). With an alphabet size of 2, the cost is increased by 30% if the number of variables is 2, and by 85% for 5 variables. Indeed, without any variable the algorithm finds the good shift, without performing any operation on the variables valuations. The presence of variables necessitates several AND operations to detect the correct shift when a mismatch occurs. Regarding the behavior of NAÏVE in the presence of variables, we note that longer partial matchings are possible on the average, which means that longer sub-patterns must be re-processed after a mismatch. This explains that, on the average, the performance is affected as well.

It turns out that the number of variables has a low impact on the relative gain of MATCH with respect to NAÏVE. For instance with 0 variable (*i.e.*, our algorithm reduces to the standard KMP) and $|\Sigma| = 2$, the gain is 50%, and so is approximately the gain with $|\Sigma| = 30$. This is explained by two opposite effects of variables. First more variables lead to more edges, and therefore extend the possibilities of having an edge compatible with the current valuation. The counterpart is that the probability of having longer edges (so smaller shifts) increases. Moreover, remind that when a mismatch occurs, we perform an AND operation for each variable located before the mismatch position.

Figure 14 shows the number of operations as a function of the number of variables, for various pattern sizes. The algorithm performance is not sensitive

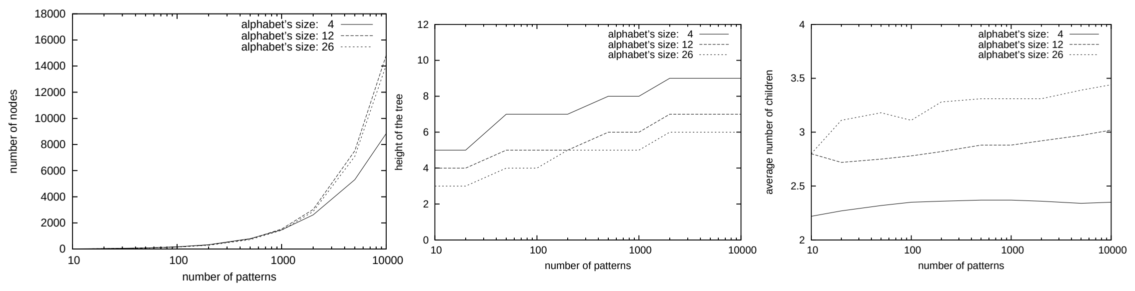


Fig. 15. Tree properties with respect to the number of patterns

to the pattern length for a given rate of variables. We observe nonetheless an exception for small patterns ($m = 5$ or less), where Naïve performs better than with large patterns. The explanation is that with small patterns a shift occurs after a few symbols, even if there is no mismatch, since we quickly reach the end of the pattern. With larger patterns we may read more symbols before being compelled to shift. Our algorithm avoids this and achieves a constant cost whatever the size of the pattern is. The global cost is only impacted by the ratio of variables. Figure 14 shows that for 50% of variables, the cost is approximately 1.85×10^6 for a pattern length of 5, 10, 15 or 20.

6.2 Filtering

The filtering approach has been implemented. We evaluated its impact over a set of patterns. The evaluation cost is measured with respect to the following parameters: (1) the number of submitted patterns, (2) the size of the alphabet, (3) the length of the patterns.

Figure 15 summarizes some properties of the pattern tree with respect to several alphabet sizes, for a number of patterns that varies from 1 to 10,000. The top-left graph shows that the size of the alphabet does not affect the total number of nodes when the number of patterns is less than 1,000 because of the low probability to draw duplicate patterns. The impact of the alphabet size on the number of nodes becomes significant for 1,000 patterns and more, and can be directly related to the probability of adding an already existing pattern, which is of course higher for small alphabets.

As expected, with a larger alphabet, the internal nodes have more children. The size of the alphabet has an opposite effect on the height of the tree. The reason is essentially that internal nodes contain more variables when the alphabet's size is large, and therefore capture more patterns. The probability of having two "close" patterns tends to be low, and prevents the generation of precise lubs. When a new pattern is introduced, if the size of the alphabet is large, there is a high probability to insert it as a child of an existing node, rather than creating a new internal node. These differences grow with respect

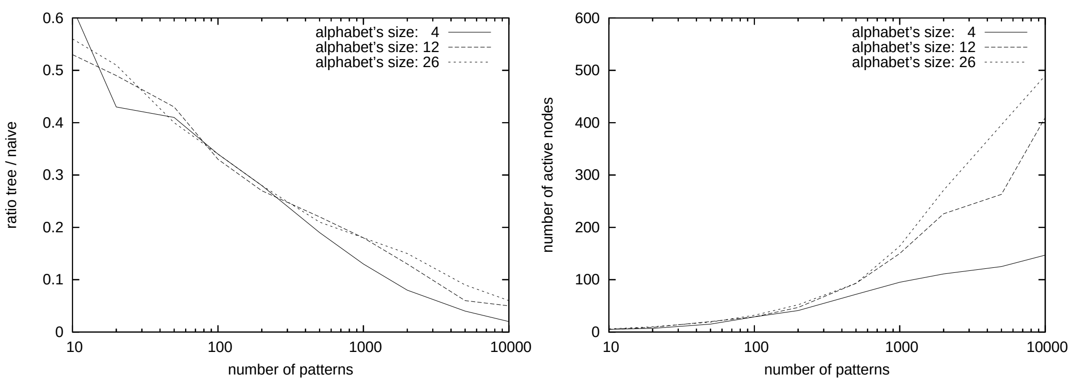


Fig. 16. Evaluation cost with respect to the number of patterns

to the number of patterns.

Figure 16 illustrates the benefit of an evaluation based on our structure compared to the independent processing of each pattern. The higher the number of patterns, the more important the gain. The ratio between the number of comparisons for the two solutions is 0.55 for 10 patterns, reaches 0.35 for 100 patterns, 0.15 for 1.000 and 0.05 for 10,000. This ratio hardly depends on the size of the alphabet, even if a small alphabet gives slightly better results over a large number of patterns, for previously presented reasons.

On the other hand, the number of simultaneously active automata is strongly related to the properties of the tree, and is therefore influenced by the alphabet size. Two of these properties, namely the filtering rate of an internal node and its number of children, have a divergent impact. As shown by the cost model, the filtering rate of internal nodes tends to be higher for large alphabets, while the number of children grows (on the average) for each node. The latter factor explains the relative importance of active automata for large alphabets, since each time a matching has been found for an internal node, all its children must be activated.

Finally, we performed additional tests with real-life data, *i.e.*, DNA sequences of different plants [19]. For these data we evaluate the gain of the MATCH algorithm over NAÏVE when searching for subsequences with various lengths, from short subsequences to very large ones, and with various variables ratios. The results for our tests with a substring of size 117591 symbols of the glycine DNA sequence are reported in Figure 17. These results on real-life dataset confirm the evaluation on synthetic data. The gain of MATCH over NAÏVE is 1.3 (Figure 17 with length=10) as for synthetic dataset (Figure 13 with $|\Sigma| = 4$).

		ratio of variables within the pattern					
		0%	10%	20%	30%	40%	50%
length = 10	Naïve	156	169	187	216	259	281
	Match	118	128	139	161	180	210
length = 20	Naïve	159	173	187	219	261	286
	Match	118	131	141	162	183	214
length = 50	Naïve	161	174	189	219	265	286
	Match	118	133	145	165	183	215
length = 100	Naïve	161	174	190	219	265	287
	Match	118	134	145	165	185	215

Fig. 17. Number of operations versus percentage of variables ($\times 10^3$), for various lengths of pattern with a DNA glycine sequence with 117591 symbols

7 Related work

The problem addressed in this paper is related to several research areas, including sequences databases, exact and approximate string matching and context-sensitive string matching.

Most studies on sequence databases aim at extending SQL with pattern-matching operators. In [2], the authors present a language called SEQUIN based on SQL in order to query sequences. In [20] sequences are considered as sorted relations where each tuple is assigned to a number that represents its position in the sequence. A shift operator using this number is defined in order to join tuples of the same sequence. The SQL-TS language presented in [21,22] shares several ideas with the present proposal. It applies to sequences of tuples an extension of the KMP algorithm to detect trends in the evolution of some attributes values. The focus is rather put on time series and on queries over successive versions of a constantly changing value. Some other papers for querying and mining similar sequences, as well as for detecting events from time series data (i.e., sequences of real numbers) are [23–27]. In [16] the authors describe a mining algorithm for retrieving spatio-temporal periodic patterns for objects moving on a partitioned map. It supports the “undefined” symbol inside a pattern but the data model, as well as the techniques used, are essentially different from ours.

The problem examined in this paper can be related to approximate string matching, which accepts a matching up to a predefined number of errors.

There has been an important amount of work in this area, see for example [28–31]. [30] presents the *agrep* algorithm, an extension of [32] to find a matching allowing k errors. An error is either an insertion, a deletion or a substitution. The space complexity is $O(k \times |\Sigma|)$, and the worst-case time complexity in $O(k \times n \times \lceil \frac{m}{w} \rceil)$. Our problem is related to theirs if we restrict errors to substitutions. However, even in that case, a major difference between approximate string matching and parameterized patterns is that errors are allowed at any position. This requires some computations for each input symbol, whereas we can take advantage of the a-priori knowledge on the variables positions, variables repetitions and query constraints to limit these computations to the cases where a mismatch occurs. Instead, using an approximate string matching algorithm would require many post-matching tests to remove a posteriori some solutions. Our algorithm constitutes a more specialized and efficient solution to the parameterized matching problem.

Introducing variable parts in patterns has been investigated in [33–36]. In [34] the goal is to match two parameterized strings together by finding a renaming of the parameters. The statement of the "context sensitive string matching problem" [36] is basically similar to ours, the major difference being that any word in Σ^* can be substituted for a variable. The evaluation problem is shown to be NP-complete in [33], even for approximate solutions [36], and the containment is shown to be undecidable in [35]. Our restriction (a variable matches a single symbol) allows for an efficient linear-time evaluation algorithm and for an easy decision procedure for containment. In [37] we investigated regular expressions with variables, and described a class of expressions with limited space requirements. The model was applied to mobile objects tracking, where trajectories are seen as sequences over a partitioned map. This paper can be seen as a further improvement of these techniques.

Several papers deal with the problem of multi-patterns matching [38–41]. The authors of [39] propose an incremental algorithm which builds a multi-pattern tree. The relaxation does not rely on variables, but on a "wildcard" character. We believe that our refinement relation is more precise. Moreover their evaluation process is quite different from ours. The multi-pattern evaluation presented in [40] relies on two deterministic automata, one built on the prefix of the set of patterns and the other on the reverse patterns. They use the second automaton at a given position until a mismatch occurs, and use the first one to determine the "shift" on the sequence. The technique is designed for simple patterns without variables.

8 Conclusion

This paper introduced parameterized pattern queries as an extension of classical pattern queries over databases, and proposed an extension of the standard pattern matching KMP algorithm [7] to parameterized patterns. This extended algorithm is suited to query answering in settings where the dataset of sequences is large. As shown by our evaluation, our technique provides an improvement over the naïve approach which merely shifts one position at-a-time by avoiding the burden of repeated comparisons of the same part of a sequence. Moreover it follows from our experiments that parameterized patterns offer an opportunity for aggregation and filtering in a multi-pattern evaluation context, with quite effective gains.

Potential for other optimizations remains to be explored. In particular, we aim at taking into account richer relationships among the different symbols of the alphabet to improve the selectivity of the query evaluation. By considering the adjacency of regions in the tracking application for instance, we can detect unsatisfiable patterns, eliminate some inconsistent edges, or remove from consideration objects that do move in a region "covered" by a given pattern. Further, we believe that the ideas of the present work could be extended by applying our variable-relaxation mechanism to the nodes of tree-structured documents (e.g., HTML or XML) [41,42]. We plan to investigate this new class of pattern-matching applications.

References

[1] R. Agrawal, R. Srikant, Mining Sequential Patterns., in: Proc. IEEE Intl. Conf. on Data Engineering (ICDE), 1995, pp. 3–14.

[2] P. Seshadri, M. Livny, R. Ramakrishnan, SEQ: A Model for Sequence Databases, in: Proc. IEEE Intl. Conf. on Data Engineering (ICDE), 1995, pp. 232–239.

[3] G. Mecca, A. J. Bonner, Finite Query Languages for Sequence Databases, in: Proc. Intl. Workshop on Database Programming Languages, 1995.

[4] R. Sadri, C. Zaniolo, A. M. Zarkesh, J. Adibi, Optimization of Sequence Queries in Database Systems, in: Proc. ACM Symp. on Principles of Database Systems (PODS), 2001.

[5] A. P. Sistla, T. Hu, V. Chowdhry, Similarity Based Retrieval from Sequence Databases using Automata as Queries, in: Proc. Intl. Conf. on Information and Knowledge Management (CIKM), 2002, pp. 237–244.

[6] R. S. Boyer, J. S. Moore, A Fast String Searching Algorithm., Commun. ACM 20 (10) (1977) 762–772.

[7] D. Knuth, J. Morris, V. Pratt, Fast Pattern Matching in Strings, SIAM J. Computing 6 (2) (1977) 323–350.

[8] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.

[9] J. Pereira, F. Fabret, F. Llirbat, D. Shasha, Efficient Matching for Web-based Publish/subscribe Systems, in: Proc. Intl. Conf. on Cooperative Information Systems (CoopIS), 2000, pp. 162–173.

[10] S. Madden, M. A. Shah, J. M. Hellerstein, V. Raman, Continuously Adaptive Continuous Queries over Streams, in: Proc. ACM SIGMOD Symp. on the Management of Data (SIGMOD), 2002.
URL `citeseer.ist.psu.edu/madden02continuously.html`

[11] C. du Mouza, P. Rigaux, M. Scholl, Efficient evaluation of parameterized pattern queries., in: Proc. Intl. Conf. on Information and Knowledge Management (CIKM), 2005, pp. 728–735.

[12] C. du Mouza, P. Rigaux, M. Scholl, On-line Aggregation and Filtering of Pattern-Based Queries, in: Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM), 2006.

[13] D. W. Mount, Bioinformatics Sequence and Genome Analysis, Second Edition, CSHL Press, 2004.

[14] C. I. Ezeife, M. Chen, Mining Web Sequential Patterns Incrementally with Revised PLWAP Tree, in: Proc. Intl. Conf. on Web-Age Information Management (WAIM), 2004, pp. 539–548.

[15] H.-F. Li, S.-Y. Lee, M.-K. Shan, On Mining Webclick Streams for Path Traversal Patterns., in: WWW (Alternate Track Papers & Posters), 2004, pp. 404–405.

[16] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, D. W. Cheung, Mining, Indexing, and Querying Historical Spatiotemporal Data, in: Proc. Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD), 2004, pp. 236–245.

[17] A.V.Aho, M.R.Garey, J.D.Ullman, The Transitive Reduction of a Directed Graph, SIAM Society for Industrial and Applied Mathematics 1 (1972) 131–137.

[18] M. J. Atallah, S. Fox (Eds.), Algorithms and Theory of Computation Handbook, CRC Press, Inc., 1998.

[19] Carnegie Institution of Washington - Department of Plant Biology, http://cellwall.stanford.edu/.

[20] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, M. Krishnaprasad, SRQL: Sorted Relational Query Language, in: Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM), IEEE Computer Society, 1998, pp. 84–95.

[21] R. Sadri, C. Zaniolo, A. M. Zarkesh, J. Adibi, A Sequential Pattern Query Language for Supporting Instant Data Mining for e-Services, in: Proc. Intl. Conf. on Very Large Data Bases (VLDB), 2001.

[22] R. Sadri, C. Zaniolo, A. Zarkesh, J. Adibi, Expressing and Optimizing Sequence Queries in Database Systems, ACM Trans. on Database Systems 29 (2).

[23] S.-W. Kim, J. Yoon, S. Park, T.-H. Kim, Shape-based Retrieval of Similar Subsequences in Time-Series Databases, in: Proc. ACM Symposium on Applied Computing, 2002, pp. 438–445.

[24] D. Q. Goldin, P. C. Kanellakis, On Similarity Queries for Time-Series Data: Constraint Specification and Implementation, in: Proc. Intl. Conf. on Principles and Practice of Constraint Programming (CP'95), 1995.

[25] V. Guralnik, J. Srivastava, Event Detection from Time Series Data, in: Proc. Intl. Conf. on Knowledge Discovery and Data Mining (SIGKDD), 1999, pp. 33–42.

[26] L. Chen, R. T. Ng, On The Marriage of Lp-norms and Edit Distance, in: Proc. Intl. Conf. on Very Large Data Bases (VLDB), 2004, pp. 792–803.

[27] Y.-N. Law, H. Wang, C. Zaniolo, Query Languages and Data Models for Database Sequences and Data Streams, in: Proc. Intl. Conf. on Very Large Data Bases (VLDB), 2004, pp. 492–503.

[28] E. Ukkonen, Finding Approximate Patterns in Strings., J. Algorithms 6 (1) (1985) 132–137.

[29] G. M. Landau, U. Vishkin, Fast String Matching with k Differences., J. Comput. Syst. Sci. 37 (1) (1988) 63–78.

[30] S. Wu, U. Manber, Fast Text Searching Allowing Errors., Commun. ACM 35 (10) (1992) 83–91.

[31] G. Navarro, A Guided Tour to Approximate String Matching, ACM Computing Surveys 33 (1) (1997) 31–88.

[32] R. A. Baeza-Yates, G. H. Gonnet, A new approach to text searching, ACM Press, Cambridge, MA, USA, 1989, pp. 168–175.

[33] D. Angluin, Finding Patterns Common to a Set of Strings., Journal of Computer and System Sciences 21 (1) (1980) 46–62.

[34] B. S. Baker, Parameterized Pattern Matching by Boyer-Moore Type Algorithms, in: Proc. of ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 541–550.

[35] T. Jiang, A. Salomaa, K. Salomaa, S. Yu, Decision Problems for Patterns, Journal of Computer and System Sciences 50 (1995) 53–63.

[36] V. T. Chakaravarthy, R. Krishnamurthy, The Problem of Context Sensitive String Matching., in: Combinatorial Pattern Matching (CPM), 2002, pp. 64–75.

[37] C. du Mouza, P. Rigaux, Mobility Patterns, GeoInformaticaTo appear.

[38] C. Forgy, Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem., Artif. Intell. 19 (1) (1982) 17–37.

[39] J. Cai, R. Paige, R. E. Tarjan, More Efficient Bottom-Up Multi-Pattern Matching in Trees., Theoretical Computer Science 106 (1) (1992) 21–60.

[40] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, W. Rytter, Fast Practical Multi-Pattern Matching., Inf. Process. Lett. 71 (3-4) (1999) 107–113.

[41] C. Y. Chan, W. Fan, P. Felber, M. N. Garofalakis, R. Rastogi, Tree Pattern Aggregation for Scalable XML Data Dissemination., in: Proc. Intl. Conf. on Very Large Data Bases (VLDB), 2002, pp. 826–837.

[42] D. Shasha, J. T.-L. Wang, H. Shan, K. Zhang, ATreeGrep: Approximate Searching in Unordered Trees., in: Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM), 2002, pp. 89–98.