# Efficient Evaluation of Parameterized Pattern Queries

Cédric du Mouza
CEDRIC, CNAM
France
dumouza@cnam.fr

Philippe Rigaux
LAMSADE, U. Paris-Dauphine
France
rigaux@lamsade.dauphine.fr

Michel Scholl
CEDRIC, CNAM
France
scholl@cnam.fr

## ABSTRACT

Many applications rely on sequence databases and use extensively pattern-matching queries to retrieve data of interest. This paper extends the traditional pattern-matching expressions to *parameterized* patterns, featuring variables. Parameterized patterns are more expressive and allow to define concisely regular expressions that would be very complex to describe without variables. They can also be used to express additional constraints on patterns' variables.

We show that they can be evaluated without additional cost with respect to traditional techniques (e.g., the Knuth-Morris-Pratt algorithm). We describe an algorithm that enjoys low memory and CPU time requirements, and provide experimental results which illustrate the gain of the optimized solution.

## Categories and Subject Descriptors

I.5.2 [**Computing Methodologies**]: Pattern Recognition—*Design Methodology*; H.2.3 [**Database Management**]: Languages

## General Terms

Performance, Languages

## Keywords

Parameterized patterns, Query evaluation

## 1. INTRODUCTION

The detection of patterns in sequence databases is a common problem in many applications such as detection of common behaviors, analysis of stock market prices, search for sequences of DNA, 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, 21, 20, 23]. Many languages proposed in these works extend in some way (querying, aggregation, data mining) the functionalities of SQL with some variant of regular expressions, and many query evaluation techniques build 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 subsequences where `a` is preceded and followed by the same value. This extension provides a much more expressive and flexible querying framework because 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 promote 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.

A potential problem associated to 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 achieve this search [4, 12, 6]. Our main contribution is to present an extension of the KMP algorithm [12] to parameterized pattern matching which preserves its linear time complexity and enjoys low space requirements. More specifically, it satisfies the two following properties:

1. each symbol of the input sequence is checked only once;
2. the memory space requirement of a query $q$ is proportional to the number of variables in $q$.

We implemented our algorithm and made evaluations that confirm our expected results. The experimental evaluation shows that our algorithm saves of large amount of computations and therefore decreases the query evaluation time.

## Related work

Several studies on sequence databases aim at extending SQL with pattern-matching operators. In [21], the authors present a language called SEQUIN based on SQL in order to query sequences. In [18] 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 of [20, 19] allows to express sequences of predicates, and covers repetitive patterns, arbitrary aggregates and disjunctive patterns. The paper describes an extension of the KMP algorithms to evaluate


*CIKM'05,* October 31–November 5, 2005, Bremen, Germany.

matching queries over such sequences of predicates. Some other papers [11, 10, 14] present algorithms for querying and mining similar subsequences, as well as event detection from time series data (i.e., sequences of real numbers). [14] for instance describes the necessary restrictions of the SQL language when dealing with streams, focusing on the aggregation problem. In [16] the authors describe a fast mining algorithm for retrieving spatio-temporal periodic patterns for objects moving on a partitioned map. It supports the "undefined" symbol inside a pattern. All these approaches are significantly different from ours. In particular there is nothing similar to the concept of parameterized pattern, featuring variables, proposed in our data model.

Other representative papers about string-search are [2, 5, 22]. Some works also deal with the problem of searching approximate patterns [24, 13]. None of these algorithms handle variables in patterns. To the best of our knowledge, the only work that considers patterns with parameters is [3], but the goal is to match two parameterized strings together by finding a renaming of the parameters. In [8] 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. The present paper can be seen as a further improvement of these techniques.

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 optimized solution. We provide experimental results in Section 5. Some conclusions are drawn in Section 6.

## 2. MOTIVATING EXAMPLES

We provide in this section an illustration of our motivation with some representative applications. Their common feature is to store 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.

### Protein databases

Applications that deal with DNA or proteins rely on a database that stores millions of sequences [17]. Consider for instance a protein sequence database. Basically a protein is composed of between 100 and 200 amino acids. There are 20 distinct amino acids that are commonly denoted by a label 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 label to denotes the amino acid named Asparagine. Beyond classical pattern matching on such sequences, our language allows to perform advanced parameterized search. For instance the parameterized pattern `Q.@x.L.`$\varepsilon$`.Q.@x.L` matches the sequences where the substring `Q.@x.L` is found twice, `@x` being bound to the *same* value in both occurrences ($\varepsilon$ denotes any subsequence).

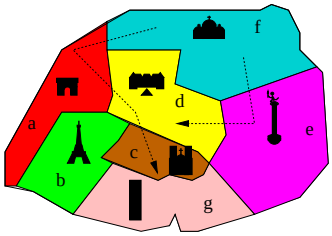



**Figure 1: A reference map**

Although an equivalent pattern can be expressed with a regular expression that enumerates all the possible bindings, its size is likely to discourage any user.

Another feature of our language is its ability to express additional constraints on variables. For instance 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. Along with a fast algorithm for pattern detection, these functionalities are unmatched by existing sequence query languages.

### User behavior on web sites

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 know where are the best places for advertisements. Assume that each page is uniquely referred by an *url*. The database can thus store the sequences of page urls – or *histories* – successively crawled by a user [9, 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 is found (and thus when a value is bound to `@x`).

The pattern `A.@x.`$\varepsilon$`.B.@x` matches all the sequences where a page `@x` is accessed successively from two distinct pages, respectively `A` and `B`. Note that we can combine our pattern expressions in powerful constructs. If we do not wish to mention explicitely `A` and `B` in the previous example, we can relax the pattern as `@y.@x.`$\varepsilon$`.@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.

### Moving objects

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 label from an alphabet $\Sigma$. This partition is the reference map $\mathcal{M}$ supporting queries. Figure 1 shows a simplified map of Paris, divided in *arrondissements*, with $\Sigma = \{a, b, c, d, e, f, g\}$.

Assume that each object is equipped with a location-aware device that periodically sends its position. 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?

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. Variables can also be seen as references to some symbols of the sequences, on which additional constraints can be expressed. Finally we show in the following that, in spite of this enhanced expressiveness, we can still rely on efficient pattern matching operations to evaluate our parameterized patterns.

# 3. THE MODEL

In our model, an object $o$ (e.g., a protein, a web user, a moving object) is represented by an identifier together with a sequence of symbols. All symbols belong to a discrete and finite domain $\Sigma$.

DEFINITION 1 (REPRESENTATION OF AN OBJECT.). *An object $o \in \mathcal{O}$ is a pair $(oid, seq)$ where oid denotes the identifier of the object and $seq =< z_1.z_2.\cdots.z_n >$ is a word in $\Sigma^*$.*

EXAMPLE 1. *Figure 1 shows two moving objects on the partitioned map, $o_1$ and $o_2$.*
*These objects are then represented as follows:*

- $(o_1, \texttt{f.a.d.c})$
- $(o_2, \texttt{f.e.d})$

Let $\mathcal{V}$ be a set of variables such that $\Sigma \cap \mathcal{V} = \emptyset$. In the following, letters `a`, `b`, `c`, ... denote symbols from $\Sigma$, and `@x`, `@y`, `@z`, ... variables.

DEFINITION 2 (PATTERN). *A* pattern *is a word $t_1.t_2 \ldots t_n$ in $(\Sigma \cup \mathcal{V})^*$.*

In their simplest form, patterns are words in $\Sigma^*$ such as, for instance, $Q_1 =$ `a.d.c` and $Q_2 =$ `b.c.e.f`. The interpretation of a pattern $P$ *without variable* is natural: a sequence $T$ *matches* a pattern $P$ if $P$ is a subsequence of $T$. For instance, since $o_1.seq =$ `f.a.d.c` $= \texttt{f}.Q_1$ then $o_1$ belongs to the result of query $Q_1$ whereas neither $Q_1$ nor $Q_2$ are subsequences of $o_2.seq$. Variables are useful to capture more general sequences where symbols are not explicitly assigned to specific symbols. $Q_3$ in the moving objects application for instance refers to *another zone* and $Q_4$ to *one departure zone*. The patterns for these queries are as follows.

- $Q_3 =$ `f.@x.d`
- $Q_4 =$ `@x.a.@x.@y`

The interpretation of patterns (with variables) is an extension of the subsequence matching semantics previously given: a sequence $T$ matches a pattern $P$ if one can substitute each variable in $P$ by a symbol from $\Sigma$, such that the resulting pattern is a subsequence of $T$. More formally:

DEFINITION 3 (SUBSTITUTION AND VALUATION). *A* substitution *$\nu$ 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 $\Sigma$. $\nu$ is a* valuation *if $t_i \in \Sigma$, for all $i \in [1, n]$*

$\nu(P)$ denotes the pattern obtained from $P$ by replacing, for each $x_i/t_i \in \nu$, 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 $bound(\nu)$. Sometimes, if $x$ is bound to $t$, for brevity $t$ will be referred to as $\nu(x)$.

If $P =$ `a.b.@x.@y.b.@z.b` and $\nu =$ {`@x/c, @z/@x`}, then $\nu(P) =$ `a.b.c.@y.b.@x.b`. In the following $var(P)$ denotes the set of variables in $P$.

Note that if $\nu$ is a valuation and $var(P) \subseteq bound(\nu)$, then $\nu(P)$ is a word in $\Sigma^*$. Hence the definition:

DEFINITION 4 (INTERPRETATION OF A PATTERN). *A sequence $T$ matches a pattern $P$ iff there exists a valuation $\nu$ such that $\nu(P)$ is a subsequence of $T$.*

Let us now turn our attention to queries. A query is built from patterns and predicates over the patterns' variables. We commonly wish to allow some "holes" into our patterns. For instance we could try to retrieve all the mobile objects that went through zone `a`, then crossed zone `d` and moved to zone `c`, and later went from `f` to `d` crossing another zone. In this example we want to know the objects that satisfied $Q_1$, then $Q_2$, without any restriction about the different visited zones during these two subsequences. Consequently we adopt the following definition for a query.

DEFINITION 5 (QUERY). *A query $q$ is a pair $(\mathcal{P}, \mathcal{C})$ such that*

1. *$\mathcal{P}$ is of the form $P_1.\varepsilon.P_2.\varepsilon.\cdots.\varepsilon.P_k$, where $k > 0$, $P_1, P_2, \ldots, P_k$ are patterns and $\varepsilon$ is the "undetermined" symbol which matches any (possibly empty) subsequence.*
2. *$\mathcal{C} = \{c_1, c_2, \cdots, c_m\}, m \geq 0$, is a (possibly empty) set of predicates over $var(\mathcal{P})$.*

Since the successive patterns of $\mathcal{P}$ must be matched in order, the semantics of a query is straightly deduced from that of a pattern: a sequence satisfies a query iff there exists a valuation of $var(\mathcal{P})$ which satisfies *successively* the different patterns of the query and the predicates in $\mathcal{C}$ . In the following we focus on the evaluation of patterns. Note that the predicates are application-dependent i.e., we might have spatial predicates in a mobile application, or specialized condition on proteins, or url comparisons.

# 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. All the symbols used throughout the paper are listed in Table 1.

| Symbol | Meaning |
|---|---|
| $P$ | A pattern |
| $m$ | The length of a pattern |
| $l$ | A position within a pattern |
| $e, e_1, e_2, \cdots$ | Edges |
| $\nu, \sigma$ | Resp.: a valuation, a substitution |
| $t_1, t_2, \cdots$ | Symbols or variables from $\Sigma \cup \mathcal{V}$ |
| $a, b, c, \cdots$ | Symbols from $\Sigma$ |
| $@x, @y, @z, \cdots$ | Symbols from $\mathcal{V}$ |

**Table 1: Table of symbols used in the paper**

## 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:

1. a *matching attempt* between a pattern $P$ and a sequence $T$ at a position $i$;
2. 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 $\Sigma$, and these bindings define a valuation $\nu$, called the *runtime valuation* which is 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 label $T[i+j]$. This binding remains in effect until the end of the matching attempt.
2. else, if `@x` $\in$ *bound*($\nu$) the comparison is successful if and only if $T[j]$ is equal to $\nu$(`@x`).

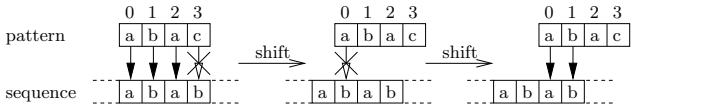
Consider the matching attempt for $P =$ `a.@x.b.@x` and $T =$ `a.c.b`. The comparisons are successful for $j = 0, 1, 2$. When $P[1] =$ `@x` is compared to $T[1] =$ `c`, variable `@x` is bound to label `c`. The valuation $\nu$ is, at this point, {`@x/c`}. The following comparison $P[4] = T[4]$ can then be successful only if the next label read in the sequence's representation is `c`, the current instantiation of `@x`. It follows that we have to maintain, for each object, the current substitution, i.e. a list of the current bindings of the query variables.

If all the comparisons are successful, then so is the matching attempt, else there is a failure. In both cases the SHIFT operation is performed.

### *The* SHIFT *operation*

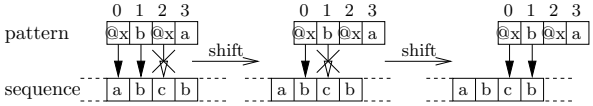
A COMPARE operation is performed each time a new label is read. Whenever a failure occurs (say, at position $l$, with $0 \leq l \leq m-1$), SHIFT shifts the pattern by one position and a comparison with the $l-1$ last labels of the sequence has to be done. If the matching is successful, one reads the next label of the sequence, else a new shift is necessary. Figure 2 shows an example.

When the pattern contains variables the algorithm is quite similar except for the binding of the variables. Whenever a failure occurs, the current substitution is deleted: all the bindings are discarded. The pattern is shifted one symbol to the right. Figure 3 illustrates this algorithm.



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

**Figure 2: Matching attempt for a pattern without variable**



(a) `@x` bound to `a` failure at pos. 2 (b) `@x` bound to `b` failure at pos. 1 (c) scan a new symbol of the sequence

**Figure 3: Matching attempt for a pattern with variables**

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

## 4.2 Optimized evaluation

We propose an optimization relying on an extension of the string-matching algorithm from Knuth, Morris and Pratt (KMP) [12, 6]. We first briefly sketch the KMP algorithm before describing its extension.

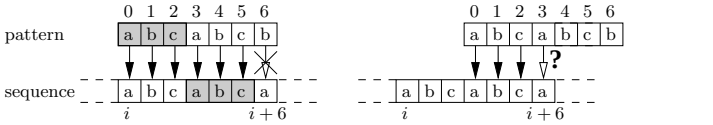
### *The KMP algorithm*

The KMP algorithm relies on the observation that, in the case of a failure, 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 4 with the pattern $P =$`a.b.c.a.b.c.b` and the subsequence $T =$ `a.b.c.a.b.c.a`.

A failure 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 failure. 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 4(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 $s_l = P[0].\cdots.P[l-1]$,



(a) a failure occurs at pos. 6 (b) The KMP shifts 3 symbols to the right and restarts the matching

**Figure 4: Example of a shift determined by the KMP algorithm**

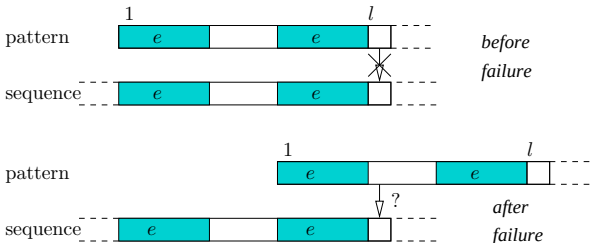
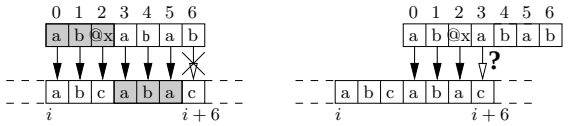



**Figure 5: Using an edge for determining the appropriate shift**



(a) a failure occurs at pos. 6 `@x` is bound to `c`

(b) we directly shift 3 symbols to the right and bind `@x` to `a`

**Figure 6: A shift for a pattern containing variables**

$l < m$ 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 failure occurs at position $l$ in the pattern, then the shift is of length $l - 1 - |e_l|$. Figure 5 illustrates this.

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 can be decomposed into two steps:

- an *offline* scan of the pattern to detect, for each substring in the pattern, the corresponding edge;
- an *online* use of the table of edges to apply the appropriate shift each time a failure occurs.

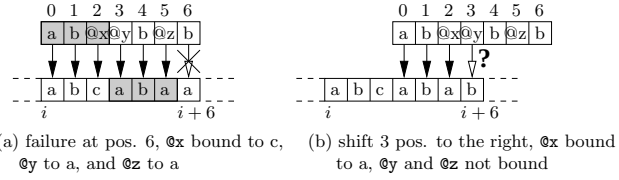
Using the table of edges when performing a matching attempt, avoids to check several times an input symbol, and the number of comparisons is therefore linear in the size of the sequence. In the following we extend this algorithm for our patterns with variables and describe an efficient evaluation process.

## *Extended KMP algorithm*

Consider the pattern $P$ = `a.b.@x.a.b.a.b` with a single variable `@x` and the example of figure 6.

When the failure occurs at position 6, `@x` is bound to `c`. If we consider the string `a.b.c.a.b.a`, the longest prefix which is also a suffix is `a.b`. However this shift removes the binding of `@x` and we can bind this variable to another symbol. Actually it is now possible to match the first three symbols of $P[0].\cdots.P[5]$ with the last three, providing that `@x` is bound to `a` *after* the shift.

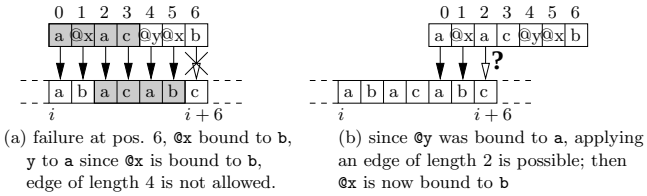
Next, consider a more complex case where the bindings *after* the shift depend on the bindings *before* the failure (Figure 7). A failure occurs at position 6, `@x` being bound to `c`, `@y` to `a` and `@z` to `a`. The best shift superposes the last three symbols of the sequence on the first three of the pattern, with a new binding of `@x` to `a` whereas `@y` and `@z` are no longer instantiated. Note that in that case the new binding of `@x` is the former binding of `@z`. Here again, an analysis of the pattern at compile-time gives all the needed information to perform the substitution of values at runtime.



(a) failure 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` and `@z` not bound

**Figure 7: A shift involving a substitution**



(a) failure 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`, applying an edge of length 2 is possible; then `@x` is now bound to `b`

**Figure 8: A shift that depends on the runtime valuation**

Finally a last example (Figure 8) shows that the edge sometimes depends on the binding of variables *before* the shift. In Figure 8.a, a failure occurs at position 6. If we consider an edge of length 4, the suffix `a.c.@y.@x` may be superposed on the prefix `a.@x.a.c` if `@x` is bound to `c` prior to the shift. This is not the case in Figure 8 since `@x` is bound to `b`. If we consider an edge of length 2, the suffix `@y.@x` must be superposed on the prefix `a.@x`. This is only possible if `@y` is bound to `a`. Hence the applicability of an edge might depend on the current runtime valuation.

## *The table of edges*

As shown by the previous examples, the computation of edges is strongly related to the variable bindings. Moreover a shift might determine a substitution of variables values which depends, partially or totally, on the runtime valuation. We now define the notion of *edge* for patterns with variables.

Definition 6 (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 $\nu_{min}$ is a valuation and $\sigma_{shift}$ a substitution, which satisfies the following properties:*

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

An edge $e = (length, \nu_{min}, \sigma_{shift})$ describes a shift of size $m - length - 1$. The valuation $\nu_{min}$ expresses a necessary and sufficient condition for applying the shift: given the runtime substitution $\nu$, the edge $e$ is applicable iff $\nu_{min} \subseteq \nu$ (we sometimes say that $\nu$ is *compatible* with $\nu_{min}$). Finally $\sigma_{shift}$ is the substitution used to bind the edge's variables after the shift. Both $\nu_{min}$ and $\sigma_{shift}$ are computed at compile time.

Assume that the superposition of a pattern $P$ on a sequence $T$ fails at position $l$ of $P$. If $(length, \nu_{min}, \sigma_{shift})$ is an edge of $P[0].\cdots.P[l-1]$ and $\nu_{min}$ is a subset of the runtime valuation $\nu$, then we can shift $P$ of $l - length - 1$ symbols to the right and restart the matching process at position $length + 1$ for $P$. The new runtime valuation is $\nu \circ \sigma_{shift}$. We illustrate these concepts with the following example.

EXAMPLE 2. *Consider the subpattern* `@x.b.@y.c.@z.@x.a`. *There exists an edge* $e(3, \nu_{min}, \sigma_{shift})$ *of length 3 with:*

- $\nu_{min} = \{$`@x/b`$\}$
- $\sigma_{shift} = \{$`@x/@z`, `@y/a`$\}$

*This is interpreted as follows. During a shift of size 3, the subpattern* `@x.b.@y` *must be superposed on* `@z.@x.a`. *Hence* `@x` *replaces* `@z`, `b` *replaces* `@x` *and* `@y` *replaces* `a`. *This superposition is possible iff the runtime valuation of* `@x` *is* `b`, *therefore the minimal valuation is* $\nu_{min} = \{$`@x/b`$\}$.

*Next, since* `@x` *replaces* `@z`, *it takes the value assigned to* `@z` *by the runtime valuation. Variable* `@y` *takes always the value* `a`. *Therefore the substitution is* $\sigma_{shift} = \{$`@x/@z`, `@y/a`$\}$. *One easily verifies that:*

$$\nu_{min}(\sigma_{shift}(\texttt{@x.b.@y})) = \nu_{min}(\texttt{@z.@x.a}) = \texttt{@z.b.a}$$

*Finally, let the last read labels of a sequence be* `c.b.a` *when the failure occurs. The current runtime valuation is* $\nu = \{$`@x/b`, `@z/c`$\}$. *Since* $\nu_{min} \subseteq \nu$, *the edge is applicable and the shift of size 3 can be performed. The valuation after the shift obtained from the substitution:* $\texttt{@x} = \nu(\sigma_{shift}(\texttt{@x})) = \nu(\texttt{@z}) = \texttt{c}$, *and* $\texttt{@y} = \nu(\sigma_{shift}(\texttt{@y})) = \texttt{a}$.

The matching attempt is performed by the following MATCH algorithm. MATCH is invoked when a new label $s$ is read from a sequence $T$. It takes as inputs $s$, the runtime valuation $\nu$ and the current position $l$ in $P$. MATCH attempts the matching between $T$ and the suffix of pattern $P$ starting at $l$. It returns the new runtime valuation $\nu'$ and the new position $l'$ in $P$.

```
MATCH(s, l, ν)
Input: s (sequence label), l (position in P), ν (runtime valuation)
Output: ν' (new valuation for P), l' (next position in P)
begin
    if (P[l] ∈ V and P[l] ∉ bound(ν)) then
        // P[l] is a variable not yet bound
        ν' := ν ∪ {P[l]/s}
        l' := l + 1
    else if (P[l] = s or ({P[l]/t_l} ∈ ν and t_l = s)) then
        // P[l] is equal to (or already bound to) s
        ν' := ν
        l' := l + 1
    else // Mismatch between P[l] and s: use edges
        if (l = 0) then
            return (∅, 0) // No applicable edge: shift the whole pattern
        else
            (ν', l') := EDGESHIFT (ν, l)
            return MATCH (s, l', ν')
        end
    endif
    // if the last symbol of P was successfully matched,
    // the sequence id is added to the resultset
    if (l' = m + 1) then
        addSequenceToResultSet()
        // shift the pattern to detect a possible occurrence later
        (ν', l') := EDGESHIFT (ν', l')
    endif
    return (ν', l')
end
```

If $l'$ is the size of $P$, then the pattern has been fully recognized and the sequence is added to the query result. Otherwise, MATCH returns a new position $l' < m$ in $P$. Upon reception of a new symbol from $T$, a matching attempt will resume between the suffix of pattern $P$ starting at position $l'$ and sequence $T$.

MATCH calls the procedure EDGESHIFT which takes the longest edge $e$ such that the $\nu_{min}$ valuation is a subset of $\nu$. Once $e$ has been found, the shift is performed as follows:

- $P$ is shifted of $l - length - 1$ symbols to the right and the current position in $P$ becomes $e.length + 1$;
- the new runtime valuation $\nu'$ is $\nu \circ \sigma_{shift}$.

EDGESHIFT takes as inputs, the runtime valuation and the current position in pattern $P$ and returns the new runtime valuation and a new position in $P$.

```
EDGESHIFT (ν, l)
begin
    // Take the edges associated with the current position
    // in P, stored in decreasing length order
    ν' = ∅
    for each e in edges[l] do
        if (e.ν_min ⊆ ν) then
            // Edge found, possibly the default edge whose length is 0
            // Right shift of the pattern
            l' := e.length
            // Set the new current substitution
            for each {x_j/t} in σ_shift do
                if (t ∈ Σ) then ν' := ν' ∪ {x_j/t}
                else ν' := ν' ∪ {x_j/ν(t)}
            endfor
            return (ν', l')
        endif
    endfor
end
```

Consider for instance the pattern $P =$ `a.@x.b.a.@x.@y.c.d`. Whenever a failure occurs at position $l = 6$, we need to consider the following edges for the sub-pattern $P[0].\cdots.P[5] =$ `a.@x.b.a.@x.@y`:

- $(0, \emptyset, \emptyset)$ the default edge that corresponds to a shift of the whole pattern;
- $(1, \{$`@y/a`$\}, \emptyset)$, because if $\nu$ is compatible with $\{$`@y/a`$\}$ (i.e., the latter is a subset of the former), then the sequence suffix is of the form `a.@x.b.a.@x.a`, and a shift of one symbol is possible;
- $(2, \{$`@x/a`$\}, \{$`@x/@y`$\})$, because if $\nu$ is compatible with $\{$`@x/a`$\}$, the sequence suffix is of the form **a.a.b.a.a.**$\nu$(**@y**). We can replace `a.@y` by `a.@x`, the new binding of `@x` being the old binding of `@y`.
- $(3, \{$`@y/b`$\}, \{$`@x/@x`$\})$ is an edge, for similar reasons.

We cannot find any edge whose length is either 4 or 5.

Now consider a matching attempt and a failure occurring at position 5, with $\nu = \{$`@x/a`, `@y/c`$\}$. The valuation $\nu$ is not compatible with $\nu_{min}$ of the edge $(3, \{$`@y/b`$\}, \{$`@x/@x`$\})$. However it is compatible with that of the edge $(2, \{$`@x/a`$\}, \{$`@x/@y`$\})$. Consequently, using this edge, one shifts two symbols to the right and initializes a matching attempt at the third position of the pattern with the new runtime valuation.

# 5. EXPERIMENTS

We have implemented and compared two algorithms in Java on a Pentium PIV processor (3GHz) with 1GB of memory. The first algorithm, NAÏVE, is the naive one described in Section 4.1, and the other one is the extended KMP algorithm which uses the table of edges. A simulator generates synthetic trajectories and parameterized patterns, and the evaluation of queries is performed and analyzed over this synthetic dataset.

## 5.1 Data generation

The chosen territory for vehicle trajectories is a map of France. We consider several subdivisions of France whose finest is a partition into 21 administrative regions. Trajectories of a given vehicle are simulated as follows. The departure region of a vehicle is chosen at random. To simulate the receipt of GPS positions, time is modeled as a sequence of time instants. At each time instant, with probability $p_i$, each vehicle enters region $i$ or stays (with probability $1 - \sum p_i$) in its current region. If it leaves a region for a contiguous region $i$, then the label corresponding to region $i$ is added to its trajectory, and this event triggers a next step in the matching algorithms. The probability of entering (leaving) a region depends on the region importance (e.g. big cities are regions with high traffic). The set of the whole trajectories (*i.e.*, sequences of crossed zones) is stored in the database.

Mobility patterns (queries) are generated as follows. Sequences of regions with fixed size are drawn at random. The regions are chosen contiguous on the map. With a given probability $p_v$ a region symbol in the pattern is replaced by a variable $v$ drawn with repetitions from a fixed set of variables. For example if $p_v = 0.25$, one symbol out of 4 in the pattern on the average is replaced by a variable.

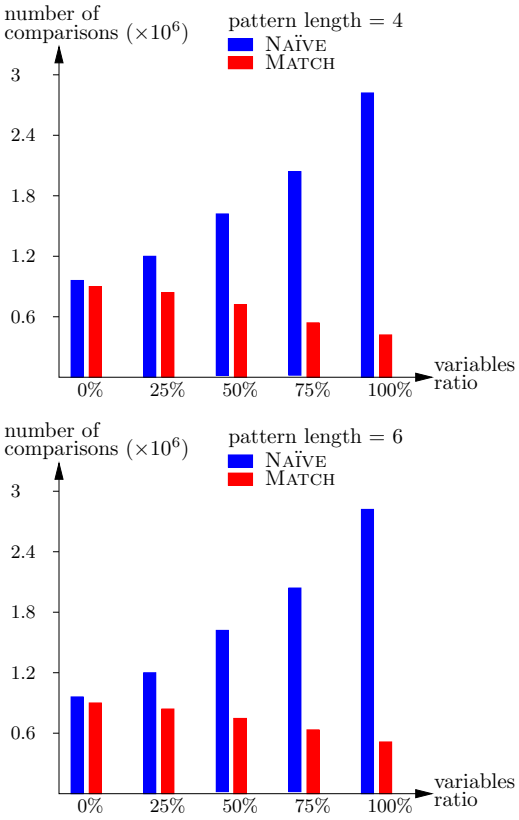



**Figure 9: Evolution of the number of comparisons for different pattern lengths and ratios of variables**

## 5.2 Experiment

The evaluation of the two algorithms is based on the total number of comparisons between a symbol of the pattern and a symbol of the trajectory. In the following, the number of vehicles is 100,000, and the average performance was taken over a set of 500 queries. We observe the cost and resource consumption of both algorithms, assuming that the vehicles moved during 20 time units, and evaluate their performance with respect to the length of the patterns and the average ratio of variables in each pattern.

Figure 9 illustrates the impact of the ratio of variables on the total number of comparisons, when the number of administrative regions (the size of the labels vocabulary) is equal to 21. Shorter (resp. longer) patterns capture too many (resp. too few) vehicles and are therefore not meaningful. The two graphs in Figure 9 display respectively the results for pattern lengths 4 and 6.

As expected, our extended KPM algorithm always outperforms the nave one. An interesting feature is that the ratio of variables has an opposite influence on the performance of the algorithms. Whereas the number of comparisons decreases in MATCH when the ratio of variables grows, it increases for NAÏVE. Moreover, the larger the ratio of variables and the smaller the pattern length, the higher the saving with MATCH. For instance, with 25% of variables and a 4-symbols pattern, the saving is 13%. When the pattern is only composed of variables (ratio = 1) the saving reaches 85%.

The NAÏVE algorithm blindly shifts the pattern one position ahead when a failure occurs, without trying to determine whether the shift can possibly be successful or not. When the pattern is highly selective (i.e., has low chances to match with a trajectory), a failure is likely to occur on the first or second symbol, and the behavior of NAÏVE remains close to that of MATCH because in that case the edges information does not bring much added value.

The difference evolves with the number of variables because variables make the pattern more generic, and therefore more compliant to match, at least partially, with trajectories. This is where the support of the table of edges brings much, and where the behaviors of the two algorithms diverge. In the case of NAÏVE, the number of comparisons is proportional to the size of the partial matching. Indeed, when a failure occurs at position $l$, all the possible shifts between 0 and $l$ must be successively investigated by NAÏVE, and each shift repeatedly compares the same trajectory symbols with different parts of the pattern.

On the other hand, MATCH takes advantage of the table of edges to limit the number of comparisons. When a failure occurs at position $l$, the density of variables in the pattern favors the existence of one or several edges, and thus makes it possible to perform the appropriate shift without any additional comparison.

Finally, Figure 10 shows that, as expected, the number of comparisons grows linearly wrt the duration of the observations (thus, the length of the sequences of moves). The number of regions is set to 21 and the ratios of variables are 25%. At each time instant, the system receives on the average the same number of events for the vehicles, with the same probability to encounter a failure during the matching attempt. This justifies the limitation of the duration to 20 time units for our experiment.

# 6. CONCLUSION

This paper proposes an extension of the standard pattern matching KMP algorithm [12] to parameterized patterns. This extended algorithm is suited to query answering in settings where the datasets of sequences is large. As shown

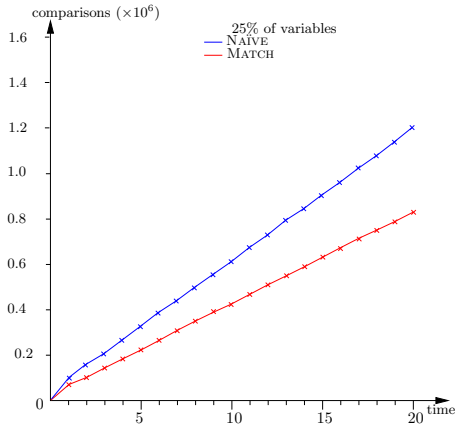



**Figure 10: Number of comparisons with time for a 6-length pattern and different ratios of variables**

by our evaluation, our technique provides a significant improvement over the naïve approach which merely shifts one position at-a-time. Indeed, the extended KMP algorithm avoids the burden of repeated comparisons of the same part of a sequence.

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.

# 7. REFERENCES


[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 3–14, 1995.

[2] A. Apostolico and R. Giancarlo. The boyer-moore-galil string searching strategies revisited. *SIAM J. Comput.*, 15(1):98–105, 1986.

[3] B. S. Baker. Parameterized Pattern Matching by Boyer-Moore Type Algorithms. In *Proc. of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 541–550, 1995.

[4] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[5] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. *Inf. Process. Lett.*, 71(3-4):107–113, 1999.

[6] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[7] C. du Mouza and P. Rigaux. Multi-scale Classification of Moving Objects Trajectories. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, 2004.

[8] C. du Mouza and P. Rigaux. Mobility Patterns. *GeoInformatica*, 2005. To appear.

[9] C. I. Ezeife and M. Chen. Mining Web Sequential Patterns Incrementally with Revised PLWAP Tree. In *WAIM*, pages 539–548, 2004.

[10] D. Q. Goldin and 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.

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

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

[13] G. M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1):63–78, 1988.

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

[15] H.-F. Li, S.-Y. Lee, and M.-K. Shan. On mining webclick streams for path traversal patterns. In *WWW (Alternate Track Papers & Posters)*, pages 404–405, 2004.

[16] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'04)*, pages 236–245, 2004.

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

[18] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, pages 84–95. IEEE Computer Society, 1998.

[19] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and Optimizing Sequence Queries in Database Systems. *ACM Trans. on Database Systems*, 29(2), 2004.

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

[21] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A Model for Sequence Databases. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 232–239, 1995.

[22] I. Simon. String matching algorithms and automata. In *Results and Trends in Theoretical Computer Science*, pages 386–395, 1994.

[23] A. P. Sistla, T. Hu, and V. Chowdhry. Similarity based retrieval from sequence databases using automata as queries. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 237–244, 2002.

[24] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1):132–137, 1985.