Efficient On-line Evaluation of Mobility Pattern Queries

Cédric du Mouza\(^1\), Philippe Rigaux\(^2\) and Michel Scholl\(^1\)

March 17, 2005

Abstract

We propose an efficient algorithm for objects tracking based on mobility patterns. We adopt an unconventional data model that relies on a discrete space representation instead of the traditional Euclidian one. The trajectories of moving objects are then represented as finite sequences of discrete moves from one location to another, and mobility patterns are expressions used to query the database.

The paper focuses on the on-line evaluation of queries, and describes an algorithm which extends traditional pattern-matching techniques to handle pattern expressions featuring variables. We show that our technique enjoys low memory and CPU time requirements, and provide experimental results which illustrate the gain of the optimized solution.

1 Introduction

Since several years, the research devoted to geometric data management in databases has strongly focused on spatio-temporal data modeling and its related aspects: query languages [24, 11, 10, 13, 23], indexing [16, 4, 21], uncertainty [9, 5], novel query types (e.g., nearest neighbors) [15, 30, 27], etc. The emergence of some recent technologies, namely networks of sensors, wireless devices and location-tracking systems, has also triggered the study of extended data models for mobile and/or continuously evolving data [31, 1].

By far, the most important class of spatio-temporal applications are those that refer to the management of moving objects. Indeed most of the significant research achievements [2, 26, 20, 19, 12, 25, 28] deal with some of the novel problems raised by continuously moving points. Earlier proposals were built on the foundations resulting from the rich outcomes of research in spatial data management in the 90’s. Such proposals used a classical geometric modeling of trajectories in the Euclidian space, extending the data structures and operations defined in the 2D space to handle time-dependent locations seen as polylines in a 3D space. Typically, in such frameworks, the location of a moving object is represented as a (linear) function of time and the database stores the function parameters. However, as the particular nature of moving points and their applications were more faithfully taken into account, new representation paradigms were proposed. For instance several studies on network-constrained moving objects are based on a graph representation and exploit graphs properties that match more closely and accurately the requirements of such applications.

In this paper we explore a novel way for expressing queries in moving object databases. A partition in 2D zones serves as a reference to describe the moving objects trajectories. This representation has applications in traffic control systems, post-acquisition analysis, clustering and classification of trajectories. Some other

\(^{1}\)Lab. CEDRIC, CNAM, Paris, France, \{dumouza,scholl\}@cnam.fr

\(^{2}\)LAMSADE Univ. Paris-Dauphine, Paris, France, rigaux@lamsade.dauphine.fr
studies [14, 29] adopt this discrete framework which significantly reduces the complexity of operations with respect to a continuous space representation. Unlike the afore mentioned papers we do not restrict the discrete space to be a regular grid. Any partition whose granularity and topology match the user’s needs can be considered.

Our model supports queries based on sequences of events, i.e., moves from one zone to another, called mobility patterns. The data model is presented in [6], and its multi-scale aspects are investigated in [7]. In the following we focus on the evaluation of continuous queries based on mobility patterns, i.e., queries that watch a population of moving objects and whose result is continuously updated by events that give the new locations of objects. We introduce variables in queries which allow to capture more general mobility patterns. Our main contribution is an algorithm which minimizes both storage requirements and query cost. More specifically, it satisfies the two following properties:

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

The first property ensures that we never have to fetch data from the past regarding the trajectory of an object $o$, since the information brought by the last event is always sufficient to know whether $o$ belongs or not to the query result. The second property corresponds to minimal memory storage. It follows that query results can be updated efficiently, and thus the model scales easily to large settings.

We implemented our algorithm in a prototype and made evaluations that confirm our forecast results. It should be underlined that, unlike the traditional complexity analysis which measures the cost of a query evaluation with respect to the number of I/Os, we consider the number of main memory operations necessary to maintain a query result through a stream of events. In this respect, the experimental evaluation shows that our algorithm saves of large amount of computations and therefore decreases the continuous query evaluation time.

Section 2 introduces the data model. Section 3 is devoted to query evaluation algorithms, including our optimized solution. We provide experimental results in Section 4. Some conclusions are drawn in Section 5.

2 The model

This section briefly outlines the main components of the data model, namely data representation, mobility patterns, and query language. A longer presentation can be found in [8].

Data representation

We consider a partition of the 2D embedding space such that each zone is uniquely identified with a label from an alphabet $\Sigma$. This partition is the reference map $M$ supporting queries. Figure 1 shows a simplified map of Paris, divided in arrondissements, with $\Sigma =$ \{a, b, c, d, e, f, g\}.

Defining which partitions are relevant is beyond the scope of this paper. For our concerns, it suffices to note that each object moving in the partitioned area successively crosses a sequence of distinct zones: our patterns are built from such sequences. Trajectories and their classification are very sensitive to the partition of space which depends on the thematic focus. An alternate thematically “neutral” partition consists in building patterns on a grid-like partition in equal-sized cells, as discussed in [18].

Next, we consider a set $O$ of objects moving over $M$ and assume that each object is equipped with a location-aware device that periodically sends its position. In our model, a moving object is represented by
an identifier together with a trajectory which is itself a sequence of labels representing the successive zones crossed by the object.

**Definition 1 (Representation of an object.).** A moving object $o \in O$ is a pair $(oid, traj)$ where $oid$ denotes the identifier of the object and $traj = \langle z_1, z_2, \ldots, z_n \rangle$ is a word in $\Sigma^*$ and $z_i \neq z_{i+1}, i \in [1, n - 1]$.

Clearly, the description of a trajectory can be obtained from a GPS device which periodically provides the object’s location. Each new event appends a new label to the trajectory’s representation. It is important to note that the continuous aspect of trajectories is ignored because it is not relevant for our purposes. Since we associate each location with a zone in a map, we just have to assume that the GPS device provides at least one location for each zone crossed by an object. Note also that we eliminate repeated successive labels from a pattern because such a repetition does not provide any useful information as long as we cannot guarantee that the events related to an object are separated by constant time intervals.

**Example 1.** Figure 1 shows two moving objects, $o_1$ and $o_2$. Each dot during their motion corresponds to an event. These objects are described by the pairs:

- $(o_1, f . a . d . c)$
- $(o_2, f . e . d)$

**Patterns**

Let $\mathcal{V}$ be a set of variables such that $\Sigma \cap \mathcal{V} = \emptyset$. In the following, letters $a, b, c, \ldots$ denote labels from $\Sigma$, and $@x, @y, @z, \ldots$ variables. Our goal is to express queries such as:

- $Q_1$: which objects went through zone $a$, then crossed zone $d$ and currently stand in zone $c$?
- $Q_2$: which objects went through zone $b$, then crossed $c$ and $e$ and currently stand in $f$?
- $Q_3$: 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 where they currently stand?

These queries are examples of mobility patterns (patterns in short), defined as follows.

**Definition 2 (Pattern).** A pattern is a word $t_1.t_2 \ldots t_n$ in $(\Sigma \cup \mathcal{V})^*$ such that $t_i \neq t_{i+1}, i < n$. 

![Figure 1: A reference map](image)
In their simplest form, patterns are words (without symbol repetition) in \( \Sigma^* \) such as, for instance, \( Q_1 = \text{a.d.c} \) and \( Q_2 = \text{b.c.e.f} \). The interpretation of a pattern \( P \) without variable is natural: a trajectory \( T \) matches a pattern \( P \) if \( P \) is a suffix of \( T \). The suffix represents here the most recent part of the continuous stream of GPS events. Of course a match between a pattern and a trajectory may be true for the current location of an object, observed at instant \( t \), and invalidated at the next observation instant.

**Example 2.** Consider again the two objects of Figure 1:

- since \( o_1 : \text{traj} = \text{f.a.d.c} = \text{f} \) then \( o_1 \) belongs to the result of query \( Q_1 \).
- since neither \( Q_1 \) nor \( Q_2 \) are suffices of \( o_2 : \text{traj} \), \( o_2 \) does not belong to their result.

Variables are useful to capture more general sequences where moves are not explicitly assigned to specific labels. \( Q_3 \) for instance refers to another zone and \( Q_4 \) to one departure zone. The patterns for these queries are as follows.

- \( Q_3 = \text{f.x.d} \)
- \( Q_4 = \text{a.x.e.y} \)

The interpretation of patterns (with variables) is an extension of the suffix matching semantics previously given: a trajectory \( T \) matches a pattern \( P \) if one can substitute each variable in \( P \) by a symbol from \( \Sigma \), such that the resulting pattern is a suffix 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 label 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 \( \text{bound}(\nu) \). Sometimes if \( x \) is bound to \( t \), for brevity \( t \) is referred to as \( \nu(x) \). If, for instance, \( P = \text{a.b} \cdot \text{x.e.y.b} \cdot \text{z.b} \) and \( \nu = \{x/c, z/x\} \), then \( \nu(P) = \text{a.b.c.e.y.b} \cdot \text{z.b} \).

Let \( \text{var}(P) \) denote the set of variables in \( P \).

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

**Definition 4 (Interpretation of a pattern).** A trajectory \( T \) matches a pattern \( P \) iff there exists a valuation \( \nu \) such that \( \nu(P) \) is a suffix of \( T \).

Since the trajectory representation evolves as new events are received, the match between a query pattern and a trajectory must be tested periodically –almost continuously– for each object. Our goal is to perform this matching test with minimal space and time consumption.

**Query language**

Patterns can easily be introduced in a SQL-based query language, extended with a \texttt{matches} boolean operator, as illustrated by the following examples.

- Give all the objects that traveled from \( \text{a} \) to \( \text{f} \), then traveled from \( \text{f} \) to \( \text{c} \).
SELECT *
FROM Mob
WHERE traj matches('a.f.c')

The \textit{matches} function checks whether a suffix of the spatio-temporal attribute \textit{traj} matches the mobility pattern \textit{a.f.c}.

- Give all the objects that went from \textit{a} to \textit{b}, crossing another zone in between?

SELECT *
FROM Mob
WHERE traj matches('a.@x.b')
AND @x != 'a' AND @x != 'b'

It is possible to express additional constraints on the instantiation (binding) of a variable, using equalities or inequalities. The user requires in this example the object to leave \textit{a}, crossing a zone distinct from \textit{a} and \textit{b}, prior to reaching \textit{b}.

- Q3. Give all the objects that went from \textit{f} to \textit{d} by crossing another zone, and came back to \textit{f} through the same zone.

SELECT *
FROM Mob
WHERE traj matches('f.@x.d.@x.f')
AND @x != 'f' AND @x != 'd'

Recall that we consider here the \textit{continuous} evaluation of such queries. In other words their result must be maintained during a given (and possibly unbounded) period of time. An object can be added or removed from the result set during the query lifetime, depending on its most recent moves and we must be able to notify all the users of these changes. This might be challenging when the population of moving objects is large and many queries must be maintained.

3 Query evaluation

We present now two algorithms for a continuous evaluation of mobility patterns. The first one follows a naïve approach which repeatedly checks the incoming events and backtracks on the trajectory whenever a mismatch occurs. The second one is our optimized technique. All the symbols used throughout the paper are listed in Table 1.

3.1 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 \textit{matching attempt} between a pattern \textit{P} and a trajectory \textit{T} at a position \textit{i};
2. a \textit{shift} of \textit{P} whenever a mismatch occurs.
Symbol | Meaning
--- | ---
P | A pattern
m | The length of a pattern.
l | A position within a pattern (0 ≤ l ≤ m − 1).
e, e₁, e₂, ⋯ | Edges.
ν, σ | Resp.: a valuation, a substitution.
t₁, t₂, ⋯ | Symbols or variables from Σ ∪ V
a, b, c, ⋯ | Symbols from Σ
@x, @y, @z, ⋯ | Symbols from V

Table 1: Table of symbols used in the paper

The **COMPARE** operation

A matching attempt compares, one by one, from left to right, the symbols P[0], P[1], ⋯, P[m − 1] of the pattern to the symbols T[i], T[i + 1], ⋯, T[i + m − 1] of the trajectory. 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* which is initially empty. If P[j] is a variable @x, the following binding rules apply:

1. if @x ∉ bound(ν), the comparison is always successful and ν := ν ∪ {@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 ∈ 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 label c. The valuation ν is, at this point,{@x/c}. The following comparison P[4] = T[4] can then be successful only if the next label appended to the trajectory’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 event (a new label) is received. Whenever a failure occurs (say, at position l, with 0 ≤ l ≤ m − 1), **SHIFT** shifts the pattern by one position and a comparison with the l − 1 last labels of the trajectory has to be done. If the matching is successful, one waits for the next event, 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.

This technique is simple but costly since the algorithm runs in $O(m \times |T|)$. Each label of the trajectory is potentially checked several times against the pattern. Note however that we are not compelled to have a disk access in order to read the past trajectory when a failure occurs at position i. Indeed the suffix x of the trajectory, starting at position i, can be rebuilt by replacing the variables in the pattern by their current
Figure 2: Matching attempt with a pattern without variable binding. For instance in figure 3(a) we know, when the failure occurs, that the first two symbols of the pattern, $\mathcal{A} x . b$, match the first two symbols of the trajectory. Since $\nu = \{ \mathcal{A} x / a \}$ at this point, the trajectory suffix is $a . b$.

Figure 3: Matching attempt with a pattern with variables

3.2 Optimized evaluation

We propose an optimization relying on an extension of the string-matching algorithm from Knuth, Morris and Pratt (KMP) [17, 3]. KMP provides a convenient basis for dealing with the more general problem of database pattern queries featuring variables [22]. 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 trajectory \( T = \ldots a.b.c.a.b.c.a \).

![Diagram](image.png)

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

A failure occurs at position 6 of the pattern. We successfully superposed \( P[0] \ldots P[5] \) on the last 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 \( P[0] \ldots P[2] = T[i+3] \ldots T[i+5] \) (Figure 4(b)). It turns out that this shift is allowed because \( P[0] \ldots P[2] = P[3] \ldots P[5] \). Therefore it can be determined by examining the pattern, at compile-time, independently from any specific trajectory.

More generally, for each substring \( s_l = P[0] \ldots P[l] \) 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 \( l - |e_l| \). Figure 5 illustrates this.

![Diagram](image.png)

Figure 5: 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 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 trajectory. In [17] the authors propose an algorithm which builds the table of edges in time \( O(m) \). 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.

![Figure 6: A shift for a pattern containing variables](image)

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 $P[0]..P[5]$ with the last three, providing that $@x$ is bound to $a$ after the shift.

![Figure 7: A shift involving a substitution](image)

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 trajectory 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 run-time.

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 $a.c.@x.@y$ 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 $a.c.@x$. This is only possible if $@y$ is bound to $a$. Hence the applicability of an edge might depend on the current run-time 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 run-time valuation. We now define the notion of edge for patterns with variables.

**Definition 5 (Edge of a pattern).** Let $P$ be a pattern of length $m$. An edge of $P$ is a triple $(\text{length}, \nu_{\text{min}}, \sigma_{\text{shift}})$, where $\nu_{\text{min}}$ is a valuation and $\sigma_{\text{shift}}$ a substitution, which satisfies the following properties:
The valuation after the shift obtained from the substitution:

\[ e \]

time.

expresses a necessary and sufficient condition for applying the shift: given the run-time substitution \( P \) trajectory label substitution used to bind the edge's variables after the shift. Both \( \theta_y \) is always the value \( l \) position length; \( \theta_z \).

Example 3. Consider the sub-pattern \( \theta_x \cdot \cdot \theta_y \cdot \cdot \theta_z \cdot \cdot \theta_x \cdot \cdot \theta_x \cdot \cdot \). There exists an edge \( e(3, \nu_{\min}, \sigma_{shift}) \) of length 3 with:

- \( \nu_{\min} = \{ \theta_x/b \} \)
- \( \sigma_{shift} = \{ \theta_x/\theta_z, \theta_y/a \} \)

This is interpreted as follows. During a shift of size 3, the sub-pattern \( \theta_x \cdot \cdot \theta_y \cdot \cdot \theta_z \cdot \cdot \theta_x \cdot \cdot \theta_x \cdot \cdot \) must be superposed on \( \theta_z \cdot \cdot \theta_x \cdot \cdot \). Hence \( \theta_x \) replaces \( \theta_z \), \( \theta_x \) replaces \( \theta_x \) and \( \theta_y \) replaces \( \theta_x \). This superposition is possible if the run-time valuation of \( \theta_x \) is \( \theta_x \), therefore the minimal valuation is \( \nu_{\min} = \{ \theta_x/b \} \).

Next, since \( \theta_x \) replaces \( \theta_z \), it takes the value assigned to \( \theta_z \) by the run-time valuation. Variable \( \theta_y \) takes always the value \( a \). Therefore the substitution is \( \sigma_{shift} = \{ \theta_x/\theta_z, \theta_y/a \} \). One easily verifies that:

\[ \nu_{\min}(\sigma_{shift}(\theta_x, \cdot, \theta_y)) = \nu_{\min}(\theta_z, \theta_x, a) = \theta_z, \theta_x, a \]

Finally, let the suffix of a trajectory be \( c \cdot b \cdot a \) when the failure occurs. The current run-time valuation is \( \nu = \{ \theta_x/b, \theta_z/c \} \). 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: \( \theta_x = \nu(\sigma_{shift}(\theta_x)) = \nu(\theta_z) = c \), and \( \theta_y = \nu(\sigma_{shift}(\theta_y)) = a \).

The matching attempt is performed by the following MATCH algorithm. MATCH is invoked when a new trajectory label \( s \) is appended to a trajectory \( T \). It takes as inputs \( s \), the run-time 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 run-time valuation \( \nu' \) and the new position \( l' \) in \( P \).

Figure 8: A shift that depends on the run-time valuation

- \( \nu_{\min}(\sigma_{shift}(P[0], \ldots, P[length - 1])) = \nu_{\min}(P[m - length], \ldots, P[m - 1]) \)
- 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 run-time 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 during the superposition of a pattern \( P \) on a trajectory \( T \) a failure occurs at position \( l \) of \( P \). If \( (length, \nu_{\min}, \sigma_{shift}) \) is an edge of \( P[0], \ldots, P[l - 1] \) and \( \nu_{\min} \) is a subset of the run-time 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 \( P \) (see Figure 5). The new run-time valuation is \( \nu \circ \sigma_{shift} \). We illustrate these concepts with the following example.
MATCH($s, l, \nu$)

**Input:** $s$ (trajectory label), $l$ (current position in $P$), $\nu$ (run-time valuation)

**Output:** $\nu'$ (new valuation for $P$), $l'$ (next position in $P$)

**begin**

if $(P[l] \in V \text{ and } P[l] \notin \text{bound}(\nu))$ then // $P[l]$ is a variable not yet bound

\[ \nu' := \nu \cup \{P[l]/s\} \]

\[ l' := l + 1 \]

else if $(P[l] = s \text{ or } (\{P[l]/t_i\} \in \nu \text{ and } t_i = s))$ then // $P[l]$ is equal to (or already bound to) $s$

\[ \nu' := \nu \]

\[ l' := l + 1 \]

else // Mismatch between $P[l]$ and $s$: use edges

if ($l = 0$) then

return $(\emptyset, 0)$ // No applicable edge: shift the whole pattern

else

$(\nu', l') := \text{EDGESHIFT}(\nu, l)$

return MATCH$(s, l', \nu')$

end

endif

// if the last symbol of $P$ was successfully matched, the trajectory id is added to the resultset

if ($l' = m + 1$) then

addTrajectoryToResultSet()

// shift the pattern to detect a new possible occurrence later

$(\nu', l') := \text{EDGESHIFT}(\nu', l')$

end

return $(\nu', l')$

end

If $l'$ is the size of $P$, then the pattern has been fully recognized and the trajectory 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 x of pattern $P$ starting at position $l$ and trajectory $T$.

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

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

EDGESHIFT $(\nu, l)$

**begin**

// Take the edges associated with the current position in $P$, stored in decreasing length order

$\nu' = \emptyset$

for each $e$ in edges$[l]$ do

if ($e.\nu_{min} \subseteq \nu$) then

// An edge is found, possibly the default edge whose length is 0

// Right shift of the pattern

\[ l' := e.\text{length} \]

// Set the new current substitution

for each $\{x_j/l\}$ in $\sigma_{\text{shift}}$ do

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

- \((0, \emptyset, \emptyset)\) 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 trajectory suffix \( x \) is of the form \( a \cdot @x \cdot b \cdot a \cdot @x \cdot a \), and a shift of one symbol is possible;
- \((2, \{@x / a\}, \{@x / @y\})\), because if \( \nu \) is compatible with \( \{@x / a\} \), the trajectory suffix \( x \) is of the form \( a \cdot a \cdot b \cdot a \cdot a \cdot \nu / (@y) \). We can replace \( a \cdot @y \) by \( a \cdot @x \), 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 run-time valuation.

### Building the table of edges

A brute-force technique for building the table of edges of \( P \) consists in considering, for each subpattern \( P[0 \ldots m - 1] \), all the possible edges of length \( i \leq m - 1 \). 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 9):

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

\[
\begin{align*}
\nu'_{\min} &= \nu_{\min}|_{\text{var}(P[l \ldots l - 1])} \\
\sigma'_{\text{shift}} &= \sigma_{\text{shift}}|_{\text{var}(P[0 \ldots l - 1])}
\end{align*}
\]

where \( \sigma|_{\text{var}(q)} \) (resp. \( \nu_{\min}|_{\text{var}(q)} \)) denotes the restriction of \( \sigma \) (resp. \( \nu_{\min} \)) to the variables in \( q \).

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

\[
\nu_{\min}(\sigma_{\text{shift}}|_{\text{var}(P[0 \ldots l - 1])}(P[0 \ldots i - 1])) = \nu_{\min}|_{\text{var}(P[l \ldots i - 1])}(P[l \ldots i \ldots l - 1]) = \omega
\]

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

This lemma leads to an optimized algorithm \textsc{EdgesConstruction} that iteratively constructs the table of edges, deducing \( \text{Edges}[l] \) from the edges in \( \text{Edges}[l - 1] \). Each step of the algorithm is of the form:
Example 4. Consider the pattern \texttt{a}@x.b.a.@x.@y and assume that we already computed the edges at position 4:

\[
\text{Edges}[4] = \{(0, \emptyset, \emptyset), (1, \{@x/a\}, \emptyset), (2, \emptyset, \{@x/@x\})\}
\]

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 \((0, \emptyset, \emptyset)\).

- \((0, \emptyset, \emptyset) \in \text{Edges}[4]\) and \((P[0], P[5]) \in \Sigma \times \mathcal{V}
  \Rightarrow (1, \{@y/a\}, \emptyset) \in \text{Edges}[5]\)
- \((1, \{@x/a\}, \emptyset) \in \text{Edges}[4]\) and \((P[1], P[5]) \in \mathcal{V}^2
  \Rightarrow (2, \{@x/a, \@x\}, \emptyset) \in \text{Edges}[5]\)
- \((2, \emptyset, \{@x/@x\}) \in \text{Edges}[4]\) and \((P[2], P[5]) \in \Sigma \times \mathcal{V}
  \Rightarrow (3, \{@y/b\}, \{@x/@x\}) \in \text{Edges}[5]\)

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

Proposition 1. The algorithm \textsc{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 in \( \text{Edges}[l-1] \) of a pattern \( P \), with lengths ranging from 0 to \( l - 2 \). Algorithm \text{EDGES CONSTRUCTION} computes an edge of \( \text{Edges}[l] \) with length \( i + 1 \) from an edge of \( \text{Edges}[l-1] \) with length \( i \) if a matching between \( P[l] \) and \( P[i] \) is successful. So the set \( \text{Edges}[l] \) is computed from the set \( \text{Edges}[l-1] \) by carrying out one comparison for each edge of \( \text{Edges}[l-1] \) (we also add the default edge to \( \text{Edges}[l] \)). 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.\( \square \)

4 Experiments

In order to validate our approach, we have implemented and compared two algorithms in Java on a Pentium PIV processor (3000MHz) with 1024Ko of memory. The first algorithm, \text{NAÏVE}, is the naive one described in section 3.1, which shifts the pattern only one symbol to the right whenever a failure occurs. The other one is the extended KMP algorithm which uses the table of edges. We compare their performance on a simulated mobile objects tracking application. Vehicles move through the administrative regions of the French territory. A simulator generates synthetic trajectories and mobility patterns, and the continuous evaluation of queries is performed and analyzed over this synthetic dataset.

4.1 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).

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.

![Graphs showing the evolution of the number of comparisons for different pattern lengths and ratios of variables](image-url)

Figure 10: Evolution of the number of comparisons for different pattern lengths and ratios of variables
4.2 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 system supports a sample of 500 continuous queries. We observe the cost and resource consumption of both algorithms during 20 time units, and evaluate their performance with respect to the following parameters:

- the length of the patterns;
- the average ratio of variables in each pattern.

Figure 10 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. We successively consider the pattern lengths 4, 6, 8 and 10. Shorter (resp. longer) patterns capture too many (resp. too few) vehicles and are therefore not meaningful. The two graphs in Figure 10 present respectively the results for the pattern lengths 4 and 6.

As expected, our extended KPM algorithm always outperforms the naïve 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. Actually the difference between the two algorithms when the ratio of variables is low corresponds to the gain of the “pure” KMP algorithm on strings.

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.

Figure 11 shows the average number of edges per query with respect to the ratio of variables.

We also studied the influence of the size of vocabulary $\Sigma$ (region labels). Clearly, in the context of the standard KMP, a small vocabulary augments the probability of finding edges in a pattern. In order to evaluate how this impacts our extended algorithm, we aggregated some of the 21 initial regions to get a smaller number of larger regions. The results obtained for a partition into 12 or 6 regions show that the size of the partition has a low impact on the performance. For instance with a partition of the space into 6 regions, the number of edges increase was less than 10% and the number of computations decrease was less than 10%.

Finally, Figure 12 shows that, as expected, the number of comparisons grows linearly wrt the duration of continuous queries. The number of regions is set to 21 and the ratios of variables are respectively 25% and
Figure 11: Average number of edges for a pattern of length 6

Figure 12: Evolution of the number of comparisons with time for a 6-length pattern and different ratios of variables

50%. 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.

5 Conclusion

This paper proposes an extension of the standard pattern matching KMP algorithm [17] to mobility patterns, i.e., parameterized sequences of locations. This extended algorithm is suited to continuous query answering in settings where a number of queries track large datasets of moving objects. As shown 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 trajectory.
Potential for other optimizations remains to be explored. In particular, we aim at taking into account richer spatial relationships among the regions of the map to improve the selectivity of the continuous query evaluation. By considering the adjacency of regions 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. Finally we plan to study the transposition of our technique to some application domains dealing with large sequences (e.g., DNA sequences).

References


