Trajectory Sequential Patterns with Regular Expression Constraints Including Spatial Queries

Juan P. Gardella[1] and Leticia I. Gómez[2] and Alejandro A. Vaisman[3]

[1] Universidad Nacional de La Plata
jpgardel@senasa.gov.ar
[2] Instituto Tecnólogico de Buenos Aires
lgomez@itba.edu.ar
[3] Universidad de Buenos Aires
avaisman@dc.uba.ar

Abstract. Moving object (MO) data representation and computing have received a fair share of attention over recent years from the database community. Replacing raw trajectory data (i.e., MO positions at different time instants) by sequences of application-dependent *stops* occurred at so-called places of interest (PoIs) leads to the notion of semantic trajectories. Different techniques exist for sequential pattern analysis of trajectories defined in this way. One of them, RE-SPaM, expresses sequential patterns by means of regular expressions built not only over item identifiers, but also over constraints defined on the (temporal and non-temporal) attributes of the items to be analyzed. This analysis could be greatly enriched if spatial and non-spatial data associated with the MO are taken into account. In this paper we show how we can take advantage of the extensibility properties of RE-SPaM to augment its expressive power by allowing to include spatial queries in the constraints. For this, we make use of Piet, a framework allowing to integrate OLAP, GIS and MO data, and its associated query language denoted Piet-QL, providing a link between moving object data and their geographic environment.

1 Introduction

Typical queries in Geographic Information Systems (GIS) [10] ask for geometric objects that satisfy some condition, or involve the aggregation of geographic measures (i.e. area, length). Although it is usual in GIS practice to store non-spatial data in thematic layers (also called *themes*), when aggregation is involved, non-spatial GIS data can be stored in a data warehouse to achieve better performance. Then, OLAP (On Line Analytical Processing) [9] tools and algorithms can be used for exploiting the data warehouse. In OLAP, data are perceived as a *data cube* where each cell contains a measure or set of measures representing facts and the contextual information which conforms *dimensions*. A recently proposed paradigm, denoted SOLAP [11] states the basic requirements for efficiently providing integration between GIS and OLAP.

Moving object data (MOD) applications [7] have been steadily gaining attention from the GIS community. The behavior of moving objects is traceable

by means of electronic devices (e.g., GPS, RFID), producing trajectory data, which can be analyzed in order to obtain interesting patterns. Since locations of a moving object are reported as a time-ordered sequence, sequential pattern algorithms appear as natural tools for querying and mining trajectory databases. Moving object positions are captured at a given time interval, with a certain granularity. Thus, the trajectory of a moving object is given by samples composed of a finite number of tuples of the form $< Oid, x, y, t >$, stating that at a certain point in time, namely t, the object Oid was located at coordinates (x, y). A very active research area in this setting is the discovery of sequential patterns in trajectories [3, 8]. Many recent proposals perform this kind of analysis not over the original MOD, but over a database built based on the ideas introduced by Spaccapietra *et al.* [12], where it is assumed that objects move over a map that contains disjoint geometries, and there is also semantic information associated with them in the form of attributes. These geometries are denoted *Places of Interest* (PoIs). When a moving object spends a fair amount of time within a PoI, the PoI is considered a *stop* of the trajectory, and all (x, y) points in the PoI are replaced by a data object representing a stop. This allows considering each object's trajectory as a *sequence of stops* instead of a *sequence of points*. Thus, sequential pattern analysis can be applied to this approximation of trajectories, also called "semantic trajectories", given that they can provide more information than the one provided by the x, y, t points alone.

1.1 Problem statement and contributions

Several authors have emphasized the need of discovering patterns in trajectories at different temporal and/or spatial granularities. However, to the best of our knowledge, the problem of finding sequential patterns that accounts for the characteristics of the *geographic environment* in which the objects move has not been addressed so far. Here we study this problem and provide a solution, building over previous work in the fields of SOLAP and sequential pattern mining.

Gómez *et al.* recently introduced RE-SPaM, a language that can express sequential patterns by means of regular expressions over constraints defined in terms of the (temporal and non-temporal) attributes of the items to be analyzed, where an item is a tuple composed of an object identifier, a time instant, and a place of interest [5][4]. These patterns can be used during the sequential pattern mining process to prune sequences that, although satisfying minimum support requirements, are not of interest to the user [2].

In order to retrieve spatial data, we need a query language that can return spatial objects. For this, we use Piet-QL [6][5], a language that supports the Piet data model [4], a proposal aimed at integrating GIS and OLAP in a single framework. Piet-QL is an SQL-like query language that can express complex and powerful spatial and OLAP queries, and supports the operators included in the Open Geospatial Consortium specification[6] for SQL. In addition, it incorporates

[4] http://piet.exp.dc.uba.ar/mo-patterns
[5] http://piet.exp.dc.uba.ar/pietql
[6] http://www.opengeospatial.org

the necessary syntax to integrate OLAP operations through the MDX standard[7]. The language resulting from the integration between RE-SPaM and Piet-QL is denoted RE-SPaM^{++}. This language delivers capabilities not present in other proposals, whose usefulness we show in this paper.

Throughout the paper we assume we have trajectory data, originally in the form (O_{id}, x, y, t), that are transformed in sequences of PoIs. Objects move in a geographic space represented in a real-world map of Belgium, consisting in five layers, containing geographic information on rivers, regions, provinces, districts, and cities. The rivers are represented as polylines, cities as points, and the other layers as polygons. There is also a data warehouse with information about stores and sales for different regions in Belgium.

1.2 Related work

Some recent proposals address the problem of mining patterns in MODs, based on the notion of "places of interest". Giannotti *et.al.* introduced *t-patterns*, for mining sequential patterns on regions of interest [3]. A t-pattern is of the form Railway Station $\overset{1h10min}{>}$ Castle Square $\overset{2h15min}{>}$ Museum. Karli and Saygin [8] propose to obtain patterns over so-called "important places" (a region where a traced object spends a fair amount of time) at different time granularities. Note that these patterns are basically defined by extension. On the contrary, regular expressions allow defining more complex patterns, intensionally. The idea of using regular expressions for trajectory analysis was first proposed by Mouza and Rigaux [1]. They present a language based on regular expressions for querying mobility patterns where each zone is represented by its label (a constant) or by a variable (@x). In this language, each occurrence of a variable in the pattern is instantiated with the same value. The language, however, has some limitations. For instance, it cannot deal with time constraints or categories, neither supports variables or data describing the geographic environment. Gómez *et al.* extend this language using regular expressions to express sequential patterns over places of interest [5], resulting in the RE-SPaM language, discussed in Section 2.

The remainder of the paper is organized as follows. Section 2 gives an overview of the languages involved in our proposal. Section 3 introduces the RE-SPaM^{++} query language and provides examples. Section 4 describes how the sequential pattern mining algorithm is modified to allow an efficient implementation of our ideas. We conclude in Section 6.

2 Preliminaries

Piet-QL. From the kinds of queries Piet-QL supports, we are interested in the ones that return spatial objects. We introduce Piet-QL through an example. Consider the query "Districts in Belgium with at least one sale in 2007".

[7] http://msdn2.microsoft.com/en-us/library/ms145506.aspx

```
SELECT GIS bel_dist.name
FROM bel_dist
WHERE bel_dist IN(
   SELECT CUBE
   filter([Store].[Store District].Members,
   [Measures].[Unit Sales]>0)
   FROM [Sales]
   slice [Time].[2007])
```

Here, bel_dist represents a layer containing the districts in Belgium. The keyword GIS tells that the query returns spatial objects. The query contains a sub-query of OLAP type (indicated by the `CUBE` keyword), that is, a query that returns a data cube. This sub-query is expressed in a language which is a slight variation of MDX, operates over a data cube, called `Sales`, takes a cube slice corresponding to sales in 2007, and filters out the stores with no sales. The hierarchy of the Store dimension in the cube is of the form storeId -> store city -> store district -> store province, meaning that store sales aggregate over cities, districts and province, in that order. The expression containing the path `[Store].[Store District].Members` in the sub-query returns the districts with at least one unit sold. System metadata allows matching the identifiers of the geometric objects with the identifiers of the level members in the OLAP dimension (in the example above, the members of the level *Store District*). Finally, we obtain the districts in the layer *bel_dist* with at least one unit sold. A detailed description of the language can be found in [6].

RE-SPaM. The RE-SPaM data model is basically composed of category schemas, category occurrences, category instances, and the table of items (ToI). Our tourist application includes four category schemas, namely *hotels*, *restaurants*, *airports* and *tourist attractions*. Each category schema is composed of a set of attributes that describe it. An element in a category is denoted a *category occurrence*, and the set of all occurrences in all categories in an application is denoted a *category instance*. A set of category instances for our running example is shown in Figure 1 (for example, the category *hotels* has two occurrences). A value of the attribute *geom* represents the geometric extension of the corresponding category occurrence. For example, in the first tuple, *pol*1 can be *Point(10 20)*. Adding a time interval to a category occurrence, produces an **item**. The time interval of an item is described by its initial and final instants, and denoted [ts, tf]. A pair $(O_{id}, item)$ is a tuple in the ToI. For the same O_{id}, the time-ordered sequence of items represent the "semantic" trajectory of the object. Figure 2 shows a *normalized* instance of the ToI corresponding to the category instances of Figure 1. There are two moving objects, O_1 and O_2, and the table contains only the O_{id}, the category occurrence identifier, and the temporal attributes. All other attributes are stored elsewhere.

Over this model, a pattern language based on regular expressions is built. The atoms in RE-SPaM are constraints expressed as formulas over attributes

Category	Instance
hotels	$[(ID, H1), (categoryName, hotel), (geom, pol1), (star, 3)]$ $[(ID, H2), (categoryName, hotel), (geom, pol2), (star, 5)]$
restaurants	$[(ID, R1), (categoryName, restaurant), (geom, pol3), (typeOfFood, French), (price, cheap)]$ $[(ID, R2), (categoryName, restaurant), (geom, pol4), (typeOfFood, French), (price, expensive)]$ $[(ID, R3), (categoryName, restaurant), (geom, pol5), (typeOfFood, Italian), (price, cheap)]$
airports	$[(ID, A1), (categoryName, airport), (geom, pol6), (type, International)]$ $[(ID, A2), (categoryName, airport), (geom, pol7), (type, Local)]$ $[(ID, A3), (categoryName, airport), (geom, pol8), (type, International)]$
attractions	$[(ID, C1), (categoryName, touristattraction), (geom, pol9), (name, CathedralofO.L.), (price, free)]$ $[(ID, C2), (categoryName, touristattraction), (geom, pol10), (name, CastleofG.theD.), (price, free)]$

Fig. 1. A set of instances

OID	Items
O_1	([(ts,04/08/2008 14:05), (tf, 04/08/2008 14:33), (ID,R2)]) ([(ts,04/08/2008 17:30), (tf,04/08/2008 18:48), (ID,R3)]) ([(ts,08/08/2008 06:22), (tf,08/08/2008 07:05), (ID,R1)]) ([(ts,08/08/2008 17:10), (tf,08/08/2008 18:17), (ID,R1)])
O_2	([(ts,19/08/2008 09:00), (tf,19/08/2008 10:20), (ID,R1)]) ([(ts,19/08/2008 17:00), (tf,19/08/2008 18:12), (ID,R2)])

Fig. 2. An instance of the Normalized ToI

of the complex items defined above. Constraints consist in conjunctions of expressions, enclosed between squared brackets. The regular expression language is built in the usual way, supporting the standard operators ('()','*','+','?','.','|'). The language also supports variables (strings preceded by '@').

As an example, a pattern expressing trajectories of tourists who visit hotel H1 and then a place characterized as 'cheap' or that serves French food, reads:

`[ID='H1'].([price='cheap']|[typeOfFood='French'])`

The second constraint does not mention IDs, only categorical attributes. The disjunction is evaluated as follows: 'cheap' places are restaurants R1 and R3 (Figure 1). Places that serve French food are R1 and R2. During the mining process, the items which satisfy these conditions are computed, without the need of explicit enumeration of all the possibilities.

Functions are supported in RE-SPaM in the forms functionName(attr, ...) = 'constant', and functionName(attr, ...) = @variable, and can be defined ad-hoc. Syntactically, the first parameter may be an *attribute of a category occurrence* (for example, typeOfFood in our running example), or a *temporal attribute* (ts, tf, or their subparts). All other parameters must be literals, and the function also returns a literal. For example, a function $compares(price, c)$, compares the attribute *price* with a literal, and returns 'equal', 'less', or 'greater than'; the first parameter is an attribute of the category occurrences of *restaurants* and *tourist attractions*, and the second one is a constant. The function can be invoked as $compares(price,$'100'$)$. Also rollup functions à la OLAP can be defined to return ranges of time for a temporal attribute of an item (e.g., 'Early Morning', 'Morning',..). The query "Trajectories that visit two places (the second one offering cheap prices), at the same part of the day (e.g., both of them during the morning) on October 10th, 2008" uses this function, reading:

[rollup(ts_time,'range','Time')=@z $\wedge$ ts_date='10/10/2008'].
[rollup(ts_time, 'range','Time')= @z $\wedge$ ts_date='10/10/2008' $\wedge$
price='cheap']

Note that RE-SPaM could be used as a query language over the trajectory database, or to prune the patterns obtained during the mining process.

3 The RE-SPaM^{++} Language

Since moving objects evolve in a geographic environment, we would like to allow geometric conditions to be included in the patterns. We present RE-SPaM^{++}, a language that integrates Piet-QL and RE-SPaM allowing to add SOLAP conditions to constraints in the regular expressions of RE-SPaM. Syntactically, this extension is very simple: we only add a `WITH` statement to a Piet-QL `SELECT` clause. This statement generates a sort of materialized view that is used in a RE-SPaM expression. Thus, the language allows not only single statements but also programs comprising sequences of Piet-QL and RE-SPaM statements.

The kinds of functions discussed in Section 2 are not enough to support geometric conditions in regular expression-based constraints. A Piet-QL query returns a cursor over tuples (i.e., a set of literals), not a literal. Thus, we need to define new kinds of functions. The syntax for these functions consists in a first parameter which corresponds to an *attribute of a category occurrence* (for example, *geom*) or a *temporal attribute* (ts, tf, or their sub-parts). The second parameter must be of the form `a.b`, where the semantics is that `b` is the name of an attribute, and `a` is the name of a table associated to some *WITH clause*. The function returns a literal.

For example, if a Piet-QL query returns the geometries of regions crossed by rivers, in a structure named `r.geom` (using the WITH clause), we can then use this result to define a function that checks whether the value of the attribute `geom` (e.g., the geometry of the PoI in our running example) is contained by any of the geometries in the cursor defined by `r.geom`. The function returns 'true' or 'false', and it is invoked as *containedBy*($geom, r.geom$). We now give some examples that illustrate the use of RE-SPaM^{++}.

Q1. Trajectories that stop at a place which belongs to a region that contains a river, and whose next stop is an airport or a tourist attraction.

```
WITH TABLE regRiver(the_geom) AS
SELECT GIS DISTINCT(bel_regn.the_geom)
FROM bel_regn, bel_river
WHERE contains(bel_regn.the_geom,bel_river.the_geom);
```

[*containedBy*(geom, regRiver.the_geom)='true'].
([categoryName='Airport']|[categoryName='Tourist Attraction'])

The Piet-QL part returns a set of geometric objects (polygons) representing regions containing rivers, in the cursor `regRiver(the_geom)`. In the RE-SPaM

part of the query, the first constraint checks if the PoI is contained in one of the regions in the set. In other words, when an item in the Table of Items is being evaluated (e.g., during the mining process or just using RE-SPaM^{++} as a query language), the corresponding PoI geometry (represented by the attribute `geom`) is compared against each of the geometric elements in the cursor.

Q2. Trajectories that stop at a place with cheap prices, which is very close to a district located in a region crossed by a river, and then at the Castle of Gerard the Devil (G. the D.), finishing there.

```
WITH TABLE district(the_geom) AS
SELECT GIS DISTINCT(bel_dist.the_geom)
FROM bel_dist, bel_regn, bel_river
WHERE intersects(bel_regn.the_geom, bel_river.the_geom) and
contains(bel_regn.the_geom, bel_dist.the_geom);
```

```
[price='cheap'  ∧  short_distance(geom,
district.the_geom)='true'].[name='Castle of G. the D.'])
```

This example also shows how the Piet-QL part of the query is used to link the trajectories of the moving objects to the geographic space where they evolve. Here, the Piet-QL query returns districts (i.e., polygons) in a map. At evaluation time, each geometry of the PoI where a trajectory stops is compared with the geometry of each district in the cursor, to check if the PoI being visited is close to it (we are not interested in how this 'closeness' is computed, we just give this query as an example to illustrate the power of the language).

Q3. Trajectories that visit a place with cheap prices and then stop at an airport (finishing there), such that *both stops* are either located in regions crossed by a river (although not necessarily the same region), or not crossed by rivers.

```
WITH TABLE reCrRi(the_geom) AS
SELECT GIS DISTINCT(bel_regn.the_geom)
FROM bel_regn, bel_river
WHERE intersects(bel_regn.the_geom, bel_river.the_geom);
```

```
[price='cheap'  ∧  containedBy(geom,reCrRi.the_geom)= @x].
[categoryName='Airport' ∧ containedBy(geom,reCrRi.the_geom)=@x ]
```

In this case, the variable $@x$ is of boolean type. At evaluation time, the variable is bound to 'true' or 'false', and the two constraints are evaluated with this value. In this example, the two constraints in the RE-SPaM^{++} expression include the geometric function *containedBy*.

4 The RE-SPaM^{++}Algorithm

We explain now how RE-SPaM^{++} is used within a sequential pattern mining algorithm. The algorithm for finding frequent patterns is a variation of the one de-

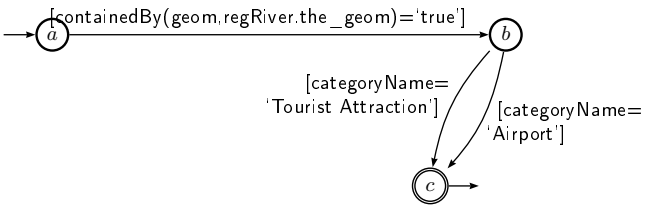

Fig. 3. Automaton for Q1

scribed in detail in [5]. The input to the algorithm is an RE-SPaM^{++} program[8], and a value for the *minimum support* required for the discovered sequences. In a nutshell, the support of a sequential pattern is the number of sequences in the ToI that satisfy such pattern, out of the total number of sequences in the database. During evaluation, the discovered patterns are restricted to the ones satisfying the RE-SPaM^{++} expression. In what follows we focus on the changes introduced on the original algorithm in order to support the new features of RE-SPaM^{++}, although to make the explanation self-contained we need to briefly go over the RE-SPaM algorithm. For the sake of clarity, we proceed by means of an example, using query Q1 from Section 3.

As a first step, a deterministic finite automaton (DFA) that accepts the language generated by the regular expression (RE) is built. The DFA for the RE-SPaM^{++} part of Q1 is shown in Figure 3. The labels of the edges of the automaton are *constraints* that must be satisfied by the sequence that is being evaluated at each step of the algorithm that we describe next. The evaluation of the RE proceeds in incremental phases, building, at each step, candidate sets C_i of sequences of length i. In short, in the first step it builds the set C_1 with candidate sequences (i.e. sequences of POIs) of length one. The automaton is used to prune candidate sequences that do not satisfy expressions not involving temporal attributes. At the final phase of the step, the ToI is queried to detect which sequences must be discarded because they do not have the *minimum support*. During a step k, the self-join of C_{k-1} is used to produce C_k, and then the automaton and the ToI are, again, used for pruning. Temporal attributes and variables in the constraints can be evaluated at different moments. We can either: (a) postpone their evaluation to the final phase of the algorithm; or, (b) evaluate them as soon as possible to reduce the size of the intermediate candidate sequences C_k. This is the approach we follow.

Since the algorithm proceeds in incremental steps, functions must be repeatedly evaluated. Besides the need of evaluating if a portion of a candidate sequence satisfies some constraint labeling the edges of the automaton, we need to take care efficiently of the expressions introduced in RE-SPaM^{++} programs, namely the cursors explained in the previous section. To accomplish an efficient implementation we borrow ideas from dynamic programming techniques, where,

[8] We denote *RE-SPaM^{++} program* the Piet-QL and the regular expression parts altogether, and RE-SPaM^{++} the part corresponding to the regular expression.

once a function is evaluated with some parameters *its result is stored in a cache* avoiding recalculation. Two types of caching can be exploited, which we denote *macro and micro cache,* respectively. The former stores the result of a function of type fn('*value*', *tableName.attribute*) (that is, geometric functions returning sets). The latter stores the result of a function of the form fn('*value*', '*literal*', ...). We explain these ideas by means of an example.

Consider query Q1 from Section 3, asking for trajectories that pass through a place which belongs to regions that contain a river and finish at an airport or a tourist attraction. The Piet-QL part of the query returns regions in the cursor denoted `regRiver.the_geom`. Belgium is composed of three regions: Vlaams Gewest, Brussel-Hoofdstad, and the Wallonne. Brussel-Hoofdstad is *crossed by* the Kanaal van Charleroi river, but this river is not *contained* by the region. Vlaams Gewest *contains* the Ieperlee river and the Wallonne region *contains* the Ourthe Occle river. Thus, both regions are in the cursor after the Piet-QL query is evaluated. Now, we move on to the RE:

```
[containedBy(geom, regRiver.the_geom)='true'].
([categoryName='Airport']|[categoryName='Tourist Attraction'])
```

The algorithm starts by building C_1 with all the category occurrences. Then, it uses the automaton for pruning: if a candidate sequence does not satisfy any path in the automaton, it is pruned. We have three paths of length 1 in the automaton (see Figure 3). The expression `[categoryName = 'Airport']` is satisfied by the three airport occurrences of Table 1 (A1 though A3). The expression `[categoryName='Tourist Attraction']` is satisfied by the castle and the cathedral (i.e., occurrences C1 and C2). The third expression, `[`*containedBy*`(geom, regRiver.the_geom)]` must be evaluated for the category occurrences of hotels and restaurants, to check which PoIs are in regions crossed by rivers (returned by the Piet-QL part of the RE-SPaM^{++} program). Given that the expression contains a function, the cache is used for this evaluation. The engine first looks up in the *macro-cache* (implemented as a hash table) for the value associated with the key *containedBy*('*pol*1', *regRiver.the_geom*). No value is retrieved in this case, and the function starts browsing the cursor. The first tuple in the cursor is $\langle$'*Vlaams Gewest*'$\rangle$. Instead of evaluating the function, the *micro-cache* is now queried for the value associated with the key *containedBy*('*pol*1', '*Vlaams Gewest*'). We know that '*pol*1' (the geometry of Hotel H1) is contained in the Vlaams Gewest region. Thus, the association between *containedBy*('*pol*1', '*Vlaams Gewest*') and 'true' is stored in the micro-cache. Since the value returned by the function is 'true', there is no need to continue browsing the cursor. Moreover, *the macro-cache is also updated*, associating the key *containedBy*('*pol*1', *regRiver.the_geom*) with the value 'true' (see below how this is used in the next step). Next, we evaluate evaluation *containedBy*('*pol*2', *regRiver.the_geom*). Again, nothing is retrieved from the macro cache, therefore we must browse the cursor. The engine looks up in the micro-cache for a value associated with *containedBy*('*pol*2', '*Vlaams Gewest*'). Since nothing is retrieved, the function is evaluated, returning the value 'false', and this association is stored in the micro-cache. Finally, the candidate sequence

IDs
A1
A2
A3
C1
C2
H1
H2
R1
R2
R3

IDs
A1
A2
A3
C1
C2
H1
R1
R2
R3

Fig. 4. Computing C_1: before (left) and after (right) pruning with automaton

IDs	
A1	A1
A1	A2
...	...
A1	R3
...	...
A2	R3
...	...
H1	A1
H1	A2
...	...
R3	R3

IDs	
A2	A1
A2	A2
...	...
C1	A1
...	...
H1	A1
H1	A2
...	...
R2	A3
R2	C1
...	...

Fig. 5. C_2 before (left: 81 tuples), and after (right: 40 tuples) pruning with automaton

H2 is discarded since it can not satisfy any path of length one in the automaton. Thus, all sequences containing H2 are pruned. Analogously, the system finds out if 'pol3', 'pol4' and 'pol5' are contained in some tuple of the cursor regRiver.the_geom, by taking advantage of the macro and micro caches. Figure 4 shows the initial and final states of C_1. In the final phase of the first step (the step with $k = 1$), C_1 is analyzed against the ToI to check for minimum support.

For the second step, suppose that all candidate sequences of C_1 are maintained (i.e., they will be checked for minimum support). Thus, the system populates the set C_2 in the second step by joining C_1 with itself. This self-join is the one typically used in sequential pattern mining [13], adapted to the case of itemsets of length 1. Two tuples t_1 and t_2 match in the join $C_{k-1} \bowtie C_{k-1}$, if the last $k-2$ of the items in t_1 coincide with the first $k-2$ items in t_2. Then, a new tuple is formed as the union of the items in t_1 and t_2. Similarly to the first step, using the automaton the engine determines the candidate sequences of length two that are satisfied by some path of length two and discards the rest. To optimize the evaluation of functions we use the macro-cache. When deciding if the candidate sequence of length 2 $\{H1A2\}$ is accepted by some path of length two in the automaton, the function $containedBy(\text{`}pol1\text{'}, regRiver.the\_geom)$ must be evaluated. But now, this key and its associated value *can be obtained from the macro-cache*, since its value was calculated in the previous step, and this cache was updated accordingly. The process continues until a step k such that no sequences of length k exist in the ToI. In Figure 5 we show the state of C_2 before and after pruning.

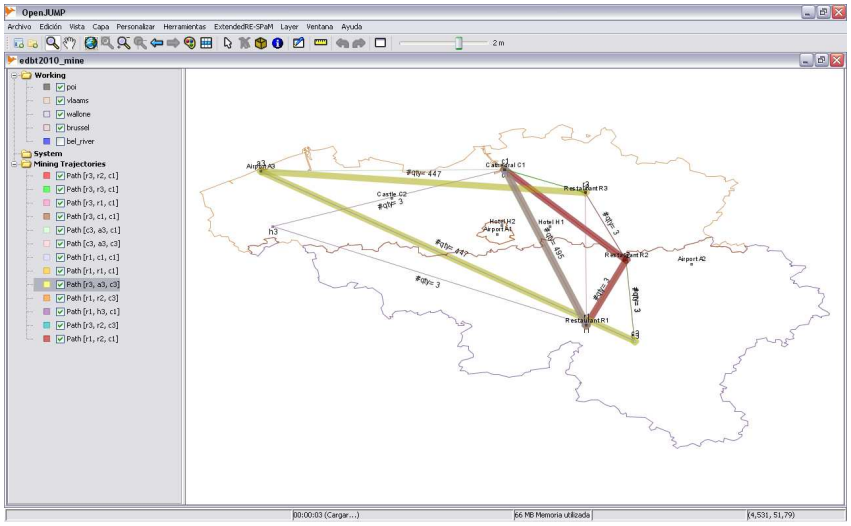

Fig. 6. Implementation tool: output window

5 Implementation and Experimental Results

We implemented RE-SPaM^{++} and incorporated this language into the Piet framework [4][9]. Figure 6 depicts the graphic output of the system. Patterns of length two (i.e., involving three PoIs) are shown. The thickness of the lines reflects the relative support of the discovered pattern, and the edges are labeled with the number of associated trajectories. For example, pattern [r3,a3,c3] has 447 occurrences, and the lines are much thicker than the ones corresponding to pattern [r1,h3,c1], which has only 3.

Preliminary experiments were aimed at assessing the impact of introducing geometric functions in regular expressions. Due to space limitation we do not give a detailed report of the experimental results, but a general comment on them. Experiments showed that the time overhead introduced by the parsing of the Piet-QL part of a RE-SPaM query is negligible. With respect to execution time, we know that it depends on several parameters, like the size of the intermediate C_k sets, minimum support, and the kinds of expressions used in the query, among other factors. Since GSP finds all frequent sequences in a database, and RE-SPaM^{++} restricts these sequences to the ones matching the regular expressions, adding a spatial predicate to a constraint in general results in a lower number of sequences. Thus, two opposite effects appear: on the one hand the use of functions requires less memory space due to the smaller size of intermediate C_k; on the other hand, function evaluation impacts over execution time. Our hypothesis was that the former reduction compensates the cost of evaluating the function. This hypothesis was confirmed by our experiments. Moreover, the

[9] A demo is available at http://piet.exp.dc.uba.ar/extendedrespam/installation.pdf.

overall performance of the algorithm does not decrease substantially when using the cached functions and execution times appear compatible with user needs. For low values of support, the number of frequent sequences obtained increases, therefore the execution times also increase because of the higher number of iterations (and the size of intermediate C_k sets).

6 Conclusion

We have presented a language that allows to include spatial OLAP queries in regular expressions that are used during sequential pattern mining for pruning sequences that are of no interest to the user. We believe this is a relevant feature when dealing with sequential patterns in trajectory databases, and, to the best of our knowledge, no proposal of this kind has been introduced so far in the field. Our experimental results show that this language enhancement is not achieved at the expense of algorithm execution times.

References

1. C. du Mouza and P. Rigaux. Mobility patterns. In *Proceedings of the STDBM'04*, pages 1 – 8, Toronto, Canada, 2004.
2. M. N. Garofalakis, R. Rastogi, and K. Shim. Mining sequential patterns with regular expression constraints. In *IEEE Transactions on Knowledge and Data Engineering*, 2002.
3. F. Giannotti, M. Nanni, D. Pedreschi, and F. Pinelli. Trajectory pattern mining. In *KDD'07*, pages 667–680, 2007.
4. L. Gómez, S. Haesevoets, B. Kuijpers, and A. A. Vaisman. Spatial aggregation: Data model and implementation. *Information Systems*, 34:551–576, 2009.
5. L. I. Gómez and A. A. Vaisman. Efficient constraint evaluation in categorical sequential pattern mining for trajectory databases. In *EDBT*, pages 541–552, 2009.
6. L. I. Gómez, A. A. Vaisman, and S. Zich. Piet-ql: a query language for gis-olap integration. In *GIS*, page 27, 2008.
7. R. H. Güting and M. Schneider. *Moving Objects Databases.* Morgan Kaufman, 2005.
8. S. Karli and Y. Saygin. Mining periodic patterns in spatio-temporal sequences at different time granularities. *Intelligent Data Analysis*, 13(2):301–335, 2009.
9. R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling, 2nd. Ed.* J.Wiley and Sons, Inc, 2002.
10. P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases.* Morgan Kaufmann, 2002.
11. S. Rivest, Y. Bédard, and P. Marchand. Modeling multidimensional spatio-temporal data warehouses in a context of evolving specifications. *Geomatica, 55 (4)*, 2001.
12. S. Spaccapietra, C. Parent, M. L. Damiani, J. A. Fernandes de Macedo, F. Porto, and C. Vangenot. A conceptual view on trajectories. *Data Knowl. Eng.*, 65 (1):126–146, 2008.
13. R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the Fifth Int'l Conference on Extending Database Technology (EDBT)*, 1996.