Distributed Multi-Source Regular Path Queries

Maryam Shoaran and Alex Thomo

University of Victoria, Canada
{maryam,thomo}@cs.uvic.ca

Abstract. Regular path queries are the building block of almost any mechanism for querying semistructured data. Despite the fact that the main applications of such data are distributed, there are only few works dealing with distributed evaluation of regular path queries. In this paper we present a message-efficient and truly distributed algorithm for computing the answer to regular path queries in a multi-source semistructured database setting. Our algorithm is general as it works for the larger class of weighted regular path queries on weighted (as well) semistructured databases.

1 Introduction

Semistructured data is the foundation for a multitude of applications in many important areas such as information integration, Web and communication networks, biological data management, etc. The data in these applications is conceptualized as edge-labeled graphs, and there is an inherent need to navigate these graphs by means of a recursive query language. As pointed out by seminal works in the field (cf. [8, 13, 5–7]), regular path queries (RPQ's) are the "winner" when it comes expressing navigational recursion over semistructured data. These queries are in essence regular expressions over the database edge symbols, and in general, one is interested in finding query-matching database paths, which spell words in the (regular) query language.

Taking an example from spatial network databases (such as [19]), suppose that the user wants to find database paths consisting mainly of highway segments and tolerating up to k provincial roads or city streets. Clearly, such paths can easily be captured by the regular path query

$$Q = \mathit{highway}^* \parallel (\mathit{road} + \mathit{street} + \epsilon)^k,$$

where $\parallel$ is the shuffle operator.

In this paper, we consider generalized RPQ's with preference weights as introduced in [11]. For example, the user can write

$$Q = (\mathit{highway} : 1)^* \parallel (\mathit{road} : 2 + \mathit{street} : 3 + \epsilon)^k,$$

to express that she ideally prefers highways, then roads, which she prefers less, and finally she can tolerate streets, but with an even lesser preference.

Moreover, inherent database edge weights (or importance) can be naturally incorporated to scale up or down query preferences. Thus, in our spatial example, the edge importance could simply be the edge-length, and so, traversing a 100 kms highway would be less preferable than traversing a 49 kms provincial road, even though in general provincial roads are less preferable than highways.

Based on query-matching paths, there are two ways of defining the answer to an RPQ. The first is the single-source variant [1, 3], where the answer is defined to be the set of objects reachable from a given source by following some query-matching path. The second is the multi-source variant [13, 5–7, 11], where the answer is defined to be the set of *pairs* of objects that are connected by some query-matching path.

For generalized RPQ's, in the single-source variant, the answer is the set of (b, w) pairs, where w is the weight of the cheapest query-matching path connecting the database source object with object b.

On the other hand, in the multi-source variant, the answer is the set of (a, b, w) triples, where w is the weight of the cheapest query-matching path connecting database objects a and b.

In this paper, we focus on the second variant of generalized RPQ's. As the main applications based on semistructured data are distributed, we look at RPQ's from a distributed strategy angle.

Computing the answer to a generalized RPQ in the multi-source variant amounts to computing the "all-pairs shortest paths" in the subgraph of database paths spelling words in the query language. However, for each user query, there would be a new subgraph on which to compute all-pairs shortest paths, and such a subgraph cannot be known in advance, but rather only after the query evaluation finishes. This is "too late" for applying algorithms, which need global knowledge of the whole graph. With such algorithms, the user cannot see partial answers while waiting for the query to finish, and there is extra computation and communication overhead incurring after the subgraph [relevant to the query] is determined. Thus, the Flloyd-Warshall algorithm and its distributed variants are not approriate to our database setting.

Regarding work on distributed shortest path computation, we remark here Haldar's algorithm in [12], which computes all-pairs shortest paths with the best known number of messages. Our algorithm is in part inspired by this work. We consider our algorithm a generalization of Haldar's algorithm. However, Haldar's algorithm makes two simplifying (global knowledge) assumptions, which are: (1) each node knows the graph size and (2) each node knows the identities of all other nodes. Clearly, as we explained above they are not true in our setting, and our generalization is essential. Also, due to the fact that the subgraph [relevant to a query] is computed on the fly, we have challenging subtleties that need to be carefully addressed.

Our algorithm works under the assumption that the nodes of the relevant graph are computed on demand and they have local [neighbor] knowledge only. The central idea of our algorithm is to overlap computations starting from different database objects. We achieve this overlap in a careful way in order to guarantee the expansion of the best path first, in a similar spirit with the Dijkstra's methodology. However, at the same time we allow multiple expansions at different processes, which is what makes the algorithm truly distributed.

To the best of our knowledge, only very few works present a distributed evaluation of regular path queries. In [17], a distributed algorithm is presented, which works based on local knowledge only. However, it has a message complexity which is quadratically worse than our complexity in this paper.

Besides [17], other works that have delt with distributed RPQ's are [3, 18, 16, 14]. All four consider the single-source variant of RPQ's.

Finally, regarding the usefulness of weighted RPQ's, we refer the reader to [9, 10, 17, 11], which study such queries in a multitude of important applications.

The rest of the paper is organized as follows. In Section 2 we give the definitions we are based on. In Section 3, we present our distributed algorithm. In Section 4, we discuss the message complexity.

2 Databases and Weighted RPQ's

We consider a database to be an edge-labeled graph with positive real values assigned to the edges. Intuitively, the nodes of the database graph represent objects and the edges represent relationships (and their importance) between the objects.

Formally, let Δ be an alphabet. Elements of Δ will be denoted $R, S, \ldots$. As usual, Δ^* denotes the set of all finite words over Δ. We also assume that we have a universe of objects, and objects will be denoted $a, b, c, \ldots$. A *database* DB is then a weighted graph (V, E), where V is a finite set of objects and $E \subseteq V \times \Delta \times \mathbb{R}^+ \times V$ is a set of directed edges labeled with symbols from Δ and weighted with numbers from $\mathbb{R}^+$.

Before talking about weighted preference path queries, it will help to first review the classical path queries.

A *regular path query* (RPQ) is a regular language over Δ. Computationally, an RPQ is a finite state automaton (FSA) $\mathcal{A} = (P, \Delta, \tau, p_0, F)$, where P is the set of states, Δ is the alphabet, $\tau \subseteq P \times \Delta \times P$ is the transition relation, p_0 is the initial state, and F is the set of final states. For the ease of notation, we will blur the distinction between RPQ's and FSA's that represent them.

Let $\mathcal{A}$ be a query FSA and $DB = (V, E)$ a database. Then, the *answer* to $\mathcal{A}$ on DB is defined as

$$Ans(\mathcal{A}, DB) = \{(a, b) \in V : a \xrightarrow{w} b \text{ in } DB \text{ and } w \text{ is accepted by } \mathcal{A}\},$$

where $\longrightarrow$ denotes a path in the database.

Now, let $\mathbb{N} = \{1, 2, \ldots\}$. A *weighted finite state automaton* (WFSA) $\mathcal{A}$ is a quintuple $(P, \Delta, \tau, p_0, F)$, where P, p_0, and F are similarly defined as for a classical FSA, while the transition relation τ is now a subset of $P \times \Delta \times \mathbb{N} \times P$. Query WFSA's are given by means of weighted regular expressions (WRE's). The reader is referred to [2] for efficient algorithms translating WRE's into WFSA's.

Given a weighted database $DB = (V, E)$, and a query WFSA $\mathcal{A} = (P, \Delta, \tau, p_0, F)$, the preferentially *scaled weighted answer* (SWAns) of $\mathcal{A}$ on DB is

$$SWAns(\mathcal{A}, DB) = \{(a, b, r) \in V \times V \times \mathbb{R}^+ :$$
$$r = inf \left\{ \sum_{i=1}^{n} k_i r_i : (p_{i-1}, R_i, k_i, p_i) \in \tau, (c_{i-1}, R_i, r_i, c_i) \in E \right\} \Bigg\},$$

where $p_n \in F$, $c_0 = a$, and $c_n = b$.

As an example, consider the database DB and query automaton $\mathcal{A}$ in Fig. 1. There are three paths going from object a to object c. The shortest path consisting of a single edge T of weight 1, is not the cheapest path according to the query. Rather, the cheapest path is the one spelling RS. The other path, spelling RT, does not match any query automaton path, so it is not considered at all. Hence, we have that $(a, c, 3)$ is the answer with respect to a and c.

Similarly, we find the other query answers and finally have $SWAns(\mathcal{A}, DB)$ $= \{(a, b, 1), (a, c, 3), (a, d, 6), (a, a, 7), (b, c, 5), (b, d, 8), (b, a, 9)\}$.

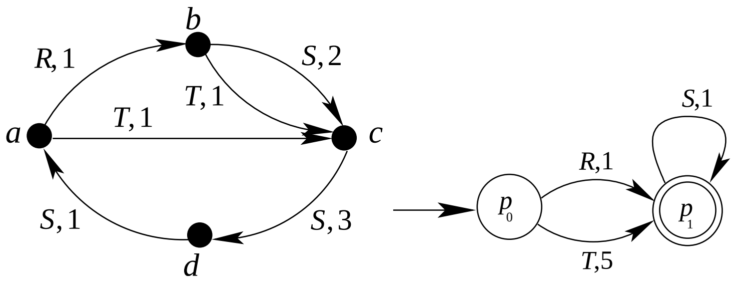

Fig. 1. A database DB and a query automaton $\mathcal{A}$

In order to help understanding of our distributed algorithm, we will first review the well-known method for the evaluation of classical RPQ's (cf. [1]). The evaluation proceeds by creating state-object pairs from the query automaton and the database. For this, let $\mathcal{A}$ be a query FSA. Starting from an object a of a database DB, we first create the pair (p_0, a), where p_0 is the initial state in $\mathcal{A}$. Then, we create all the pairs (p, b) such that there exists a transition from p_0 to p in $\mathcal{A}$, and an edge from a to b in DB, and furthermore the labels of the transition and the edge match. In the same way, we continue to create new pairs from existing ones, until we are not anymore able to do so. In essence, what is happening is a lazy construction of a Cartesian product graph of the query automaton with the database graph. Of course, only a small (hopefully) part of the Cartesian product is really contructed. This ultimately depends on the selectivity of the query.

After obtaining the above Cartesian product graph, producing query answers becomes a question of computing reachability of nodes (p, b), where p is a final state, from (p_0, a), where p_0 is the intial state. Namely, if (p, b) is reachable from (p_0, a), then (a, b) is a tuple in the query answer.

Now, when having instead a weighted query automaton and database, one can build a weighted Cartesian product graph. It is not difficult to see that, in order to compute weighted answers, we have to find, in the Cartesian product graph, the cheapest paths from all (p_0, a) to all (p, b), where p is a final state in the query automaton $\mathcal{A}$.

As we mentioned in the Introduction, in general there is a different Cartesian product graph for each query. Thus, a useful distributed algorithm must not rely on having global knowledge about the topology of this graph, since it will only be known after the completion of the query evaluation.

3 Distributed Algorithm

The key feature of our algorithm is the overlapping of computations starting from different database objects. We assume that each database object has only local knowledge about the database graph, that is, it only knows the identities of its neighbors and the labels and weights of its outgoing edges. Further, we assume that each object a, is being serviced by a dedicated process for that object P_a. Our algorithm can be easily modified for the case when subgraphs of the database (as opposed to single objects) are being serviced by the processes. In such a case, many of the basic computation messages are sent and received locally by the processes from and to themselves.

First, the query automaton is sent to each process. Such a service is commonly achieved by distributively creating a minimum spanning tree (MST) of the processes before any query starts to be evaluated (cf. [4] for a message optimal MST algorithm).

We can note here that such an MST can be used by the processes to transmit their id's and get so to know each other. However, we do not require this coordination step. Even if such a step is undertaken, the real challenge [which remains] is that the relevant subgraph of the [query–database] Cartesian product cannot be known in advance for a new query. In other words, a shortest path algorithm has to work with a target graph not known beforehand.

Continuing the description of our algorithm, each process starts by creating an initial task for itself. The tasks are "keyed" (uniquely identified) by the automaton states, with the initial tasks being keyed by the initial state p_0. Each task has three components:

1. an automaton state,
2. a status flag that can switch between *active*, *passive*, and *completed* values, and
3. a table (or set) of tuples representing knowledge about "objects reached so far" along with additional information (to be precisely described soon).

A typical task will be written as $\langle p_x, \mathit{status}, \{\dots\}\rangle$. We will refer to the table $\{\dots\}$ as $P_a.p_x.T$ or $p_x.T$ when P_a is clear from the context. The tuples in this table have four components, and will be written as $[(c, p_z), (b, p_y), \mathit{weight}, \mathit{status}]$, where

1. (c, p_z) states that the algorithm, starting from object a and state p_x, has reached (possibly through multiple hops) object c and state p_z,
2. (b, p_y) states that the best path (known so far) to reach (c, p_z) is by passing via object b and state p_y, where b and p_y are neighbors of a and p_x in the database and query automaton respectively,
3. *weight* is the weight of this best path (determined as in Section 2), and
4. *status* is a flag switching from *prov* to *opt* values telling whether *weight* is provisional and would possibly be improved or optimal and permanently stay as is.

Initially, when a p_x-task is created, process P_a tries to find all the outgoing edges from a, which match (w.r.t. the symbol label) outgoing transitions from p_x. Let (a, R, r, b) be such an edge which matches transition (p_x, R, k, p_y). Then, P_a inserts tuple $[(b, p_y), (b, p_y), k \cdot r, prov]$ in table $P_a.p_x.T$. If there are multiple $(a, \_, \_, b)$ - $(p_x, \_, \_, p_y)$ edge-transition matches, then only the match with the cheapest weight product is considered.

Each process P_a starts by creating and initializing a *passive* p_0-task, which is possibly selected next for processing. We say "possibly" because a process might receive new tasks from neighboring processes.

When a task is selected for processing, its *provisional*-status tuples (or *provisional* tuples in short) will be "expanded" in a best-first order with respect to their weights. If there are no more *provisional* tuples in the table of the p_0-task, then the task attains a *completed* status, and the process reports its *local termination*.

All (working) processes run in parallel exactly the same algorithm, which consists of four concurrent threads. These threads are as follows:

Expansion: A process P_a selects a *passive* task, say p_x–task, which still has provisional tuples in its table.
Then, P_a makes the p_x–task *active*, and selects for expansion the cheapest *provisional* tuple in its table $P_a.p_x.T$.
The *active* status for the p_x–task prevents the expansion of other *provisional* tuples in $P_a.p_x.T$.
Next, P_a sends a request message to its neighbor P_b asking it to: (1) create a task p_y, and (2) send its "knowledge" regarding the $[(c, p_z), \_, \_, \_]$ tuple.

Task Creation: When a process P_b receives a request message from P_a (w.r.t p_x) for the creation of a task, say p_y, it creates a p_y-keyed task (if such does not exist) and properly initializes it. Next, P_b establishes a virtual communication channel between its p_y-task and the p_x-task of P_a. This communication channel is specialized for the relevant tuple (keyed by (c, p_z)), whose expansion caused the request message. The weight of the channel will be equal to the cost of going from (a, p_x) to (b, p_y), which is in fact the weight of the (b, p_y)–keyed tuple in $P_a.p_x.T$.
Notably, overlapping of computations happens when process P_b receives another request message for the same task from a different neighboring process. In such a case, the receiving process P_b only establishes a communication channel with the sending process.

Reply: After creating the communication channel, process P_b will send table $P_b.p_y.T$ backward to task $P_a.p_x$. This backward message will be sent only when the (c, p_z)-keyed tuple in $P_b.p_y.T$ attains an *optimal* status. The weight of the communication channel is added to the weights of the tuples as they are bundled together to be sent. We refer to this modified (message) table as $P_b.p_y.T^*$.

Update: When a process P_a receives from some process P_b a backward reply message, which is related to a tuple $[(c, p_z), \_, \_, prov]$ of task $P_a.p_x$, and contains the table $P_b.p_y.T^*$, it will: (1) update (relax) the *provisional* tuples in $P_a.p_x.T$ as appropriate (if there are tuples with the same keys in $P_b.p_y.T^*$), (2) add to table $P_a.p_x.T$ all tuples of $P_b.p_y.T^*$, which do not have any "peer" (tuple with the same key) in $P_a.p_x.T$, and (3) change the status of the p_x-task to *passive.*

Formally our algorithm is as follows.

Algorithm 1

Input:

1. A database DB. For simplicity we assume that each database object, say a, is being serviced by a dedicated process for that object P_a.
2. A query WFSA $\mathcal{A} = (P, \Delta, \tau, p_0, F)$.

Output: The answers to query $\mathcal{A}$ evaluated on database DB.

Method:

1. **Initialization**: Each process P_a creates a task $\langle p_0, passive, \{\ldots\}\rangle$ for itself. The table $\{\ldots\}$ (referred to as $P_a.p_0.T$) is initialized as follows:
   (a) insert tuple $[(a, p_0), (a, p_0), 0, opt]$, and
   (b) For each edge-transition match,
   (a, R, r, b) in DB and
   (p_0, R, k, p) in $\mathcal{A}$,
   insert tuple $[(b, p), (b, p), k \cdot r, prov]$
   (if there are multiple $(a, \_, \_, b) - (p_0, \_, \_, p)$ edge-transition matches, then the cheapest weight product is considered.)

   If at point (b) there is no edge-transition match, then make the status of the p_0-task *completed.*

2. Concurrently execute all the four following threads at each process in parallel until termination is detected. [For clarity, we describe the threads at two processes, P_a and P_b.]

3. **Expansion**: [At process P_a]
   (a) Select a *passive* p_x-task for processing. Make the status of the task *active.*
   (b) Select the cheapest *provisional*-status tuple, say $[(c, p_z), (b, p_y), w, prov]$ from table $P_a.p_x.T$.

(c) Request P_b, with respect to state p_y, to provide information about (c, p_z).
For this, send a message $\langle p_y, [p_x, (c, p_z), w_{ab}]\rangle$ to P_b, where w_{ab} is the cost of going from (a, p_x) to (b, p_y), which is equal to the weight of the (b, p_y)–keyed tuple in $P_a.p_x.T$.

(d) Sleep, with regard to p_x-task, until the reply message for (c, p_z) comes from P_b.

4. **Task Creation**: [At process P_b]
Upon receiving a message $\langle p_y, [p_x, (c, p_z), w]\rangle$ from P_a:
if there is not yet a p_y-task
then create a task $\langle p_y$, *passive*, $\{\dots\}\rangle$ and initialize its table similarly as in the first phase.
That is,
(a) insert tuple $[(b, p_y), (b, p_y), 0, \mathit{opt}]$, and
(b) For each edge-transition match,
(b, R, r, d) in DB and
(p_y, R, k, p_u) in $\mathcal{A}$,
insert tuple $[(d, p_u), (d, p_u), k \cdot r, \mathit{prov}]$
(if there are multiple $(b, \_, \_, d)$–$(p_y, \_, \_, p_u)$ edge-transition matches, then the cheapest weight product is considered.)
Also, establish a virtual communication channel with P_a. This channel relates the p_y-task of P_b with the p_x-task of P_a. Further, it is indexed by (c, p_z) and is weighted by w_{ab} (the weight included in the received message).
else [P_b has already a p_y-task.] Do not create a new task, but only establish a communication channel with P_a as described above.

5. **Reply**: [At process P_b]
When in the p_y-task, the tuple $[(c, p_z), (\_, \_), \_, \_]$ is or becomes optimally weighted, *reply back* to all the neighbor processes, which had sent a task requesting message $\langle p_y, [\_, (c, p_z), \_]\rangle$ to P_b.
For example, P_b sends to such a neighbor, say P_a, through the corresponding communication channel, the message $\langle P_b.p_y.T^*\rangle$, which is table $P_b.p_y.T$ after adding the channel weight to the weight of each tuple.

6. **Update**: [At process P_a]
Upon receiving a reply message $\langle P_b.p_y.T^*\rangle$ from a neighbor P_b w.r.t. the expansion of a (c, p_z)-keyed tuple in table $P_a.p_x.T$ do:
(a) Change the status of (c, p_z)-keyed tuple to *optimal*.
(b) For each tuple $[(d, p_u), (\_, \_), v, s]$[1] in $P_b.p_y.T^*$, which has a smaller weight (v) than a same-key tuple $[(d, p_u), (\_, \_), \_, \mathit{prov}]$ in $P_a.p_x.T$, replace the latter by $[(d, p_u), (b, p_y), v, s]$.
(c) Add to $P_a.p_x.T$ all the rest of the $P_b.p_y.T^*$ tuples, i.e., those which do not have corresponding same-key tuples in $P_a.p_x.T$.
Also, change the via component of these tuples to be (b, p_y).

[1] s is the status which can be *prov* or *opt*.

(d) **if** the p_x-task does not have anymore *provisional* tuples,
then make its status *completed*.
If $p_x = p_0$, then report that all query answers from P_a have been computed.
else make the status of the p_x-task *passive*.

Finally upon termination, which happens when all the tasks in every process have attained *completed* status, set

$$eval(\mathcal{A}, DB) = \{(a, b, r) : [(b, p_y), (\_, \_), r, opt)] \in P_a.p_0.T \text{ and } p_y \in F\}.$$

In the full paper[2], we show the soundness and completeness of our algorithm. Based on them, the following theorem can be stated.

Theorem 1 *Upon termination of the above algorithm, we have that*

$$eval(\mathcal{A}, DB) = SWAns(\mathcal{A}, DB).$$

It is worth mentioning here that any snapshot of $eval(\mathcal{A}, DB)$ at any time during the execution of the above algorithm is a partial answer to the query. Hence, an answer can be immediately reported as soon as the corresponding tuple attains an optimal status. Upon termination, all the answers would have been reported. Also, in the full paper we show that

Theorem 2 *Algorithm 1 (positively) terminates.*

4 Complexity

In this paper, we restrict our discussion to message complexity only. We show that the upper bound of the message complexity for Algorithm 1 is quadratic in the number of database objects. In fact, we can further qualify this as the number of database objects involved in the Cartesian product explained in Section 2. This number ultimately depends on the query selectivity, and in practice one hopes that the (lazy) Cartesian product size is much smaller than the size of the database (cf. [1]).

Theorem 3 *The maximum number of messages required for a query evaluation is $2 \cdot n^2 \cdot s^2$, where n is the number of objects in DB, and s is the number of states in $\mathcal{A}$.*

Proof. We base our claim on the following facts:

1. The number of tasks in each process is bounded by s.
2. Between two tasks in different processes, there can be up to $n \cdot s$ communication channels, which are indexed by an object-state pair.
3. Only one forward message is needed to cause the creation of a communication channel.

[2] http://www.cs.uvic.ca/~thomo/all_to_all.pdf

4. Each communication channel is traversed only once, which happens when the tuple keyed by the object-state of the channel becomes optimally weighted.

Since we have n processes, the upper bound for the total number of messages is $n \cdot s \cdot (n \cdot s) \cdot (1+1) = 2 \cdot n^2 \cdot s^2$. $\square$

References

1. Abiteboul S., Buneman P., and Suciu D. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufmann, San Francisco CA (1999)
2. Allauzen C., M. Mohri. A Unified Construction of the Glushkov, Follow, and Antimirov Automata. *Proc. of MFCS'06*, LNCS 4162, Springer (2006) 110–121.
3. Abiteboul S., V. Vianu. Regular Path Queries with Constraints. *J. of Computing and System Sciences* 58(3) (1999) 428–452
4. Awerbuch B. Optimal Distributed Algorithms for Minimum-Weight Spanning tree, Counting, Leader Election and Related Problems. *Proc. of STOC'87*, ACM (1987) 230–240
5. Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. Answering Regular Path Queries Using Views. *Proc. of ICDE'00*, IEEE (2000) 389–398
6. Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on Regular Path Queries. *SIGMOD Record* 32(4) (2003) 83–92
7. Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. View-based Query Processing: On the Relationship between Rewriting, Answering and Losslessness. *Proc. of ICDT '05*, LNCS 3363, Springer (2005) 321–336
8. Consens M. P, A. O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. *Proc of PODS'90*, ACM (1990) 404–416
9. Flesca S., F. Furfaro, and S. Greco. Weighted Path Queries on Semistructured Databases. *Inf. Comput.* 204(5) (2006) 679–696
10. Grahne G., and A. Thomo. Regular Path Queries Under Approximate Semantics. *Ann. Math. Artif. Intell.* 46(1–2) (2006) 165–190
11. Grahne G., A. Thomo, and W. Wadge. Preferentially Annotated Regular Path Queries. *Proc. of ICDT'07*, LNCS 4353, Springer (2007) 314–328
12. Haldar S. An "All Pairs Shortest Paths" Distributed Algorithm Using $2n^2$ Messages. *J. of Algorithms,* 24(1) (1997) 20–36
13. Mendelzon A. O., and P. T. Wood, Finding Regular Simple Paths in Graph Databases. *SIAM J. Comp.* 24(6) (1995) 1235–1258
14. Miao Z., D. Stefanescu, A. Thomo. Grid-Aware Evaluation of Regular Path Queries on Spatial Networks. *Proc. of AINA'07*, IEEE (2007) 158–165
15. Planet–Lab: www.planet-lab.org
16. Stefanescu D., A. Thomo, and L. Thomo. Distributed Evaluation of Generalized Path Queries *Proc. of SAC'05*, ACM (2005) 610–616
17. Stefanescu D., A. Thomo. Enhanced Regular Path Queries on Semistructured Databases. *Proc. of QLQP'06*, LNCS 4254, Springer (2006) 700–711
18. Suciu D., Distributed Query Evaluation on Semistructured Data. *ACM Trans. on Database Systems,* 27(1) (2002) 1–62
19. TIGER: Topologically Integrated Geographic Encoding and Referencing system, US Census Bureau http://www.census.gov/geo/www/tiger
20. Vardi M. Y. A Call to Regularity. *Proc. PCK50 - Principles of Computing & Knowledge, Paris C. Kanellakis Memorial Workshop '03*, ACM (2003) 11