Enhanced Regular Path Queries on Semistructured Databases

Dan Stefanescu[1] and Alex Thomo[2]

[1] Suffolk University, Boston, USA, dan@mcs.suffolk.edu
[2] University of Victoria, Victoria, Canada, thomo@cs.uvic.ca

Abstract. Regular path queries are the basic navigational component of virtually all the mechanisms for querying semistructured data commonly found in information integration applications, Web and communication networks, biological data management etc. We start by proposing weight-enhanced regular path queries with semantics that allow user-assigned preference (query) weights to be naturally combined with quantitative database link-information for driving the navigation.
Motivated by the fact that the main applications of semistructured data involve distributed data sources, we focus next on the distributed evaluation of the weight-enhanced path queries. We present a distributed algorithm for evaluating our proposed queries in a *multi-source* setting. Our algorithm is general in the sense that it does not assume a known topology of the network and it can work using asynchronous communication. This algorithm can also be used to solve multi-source shortest path problems for which the full graph is not known in advance. To the best of our knowledge our algorithm is the first to address this problem in such a setting.

1 Introduction

Nowadays, modeling and/or representing the data as labeled graphs has become very common in many areas such as communication and traffic networks, Web information systems, data integration, biological data management, cartography, etc (see e.g. [8, 7, 13]). As it has been recognized by seminal works (see [1, 12]), the basic querying mechanism for such data are (regular) path queries, which are formulated using regular expressions, that provide to the user a recursive way of navigating (partially unknown) graph data.

As an example from airline networks, imagine a user who wants to find all the pairs of cities that can be reached from each other by taking Air Canada. Notably, this can easily be captured by using a path query such as $aircanada^*$, which has to be evaluated starting from each city Air Canada flies from.

In order to evaluate our example path query, a query processor would try to find all the paths consisting of *aircanada*-labeled edges. However, there are two problems associated with path queries like the above. First, a selective query (such as $aircanada^*$) may return too few answers. For example, two cities such as Vancouver (*YVR*) and San Diego (*SAN*) might not have an all-*aircanada* labeled

route[3] and thus, the pair (*YVR, SAN*) will not be in the returned answer for the query. On the other hand, partnerships between airlines are a well known fact, and clearly the system should be able to return the pair (*YVR, SAN*) even in the absence of an Air Canada route from Vancouver to San Diego. In the particular example there is a jointly operated route with Air Canada flying from Vancouver to Los Angeles and continuing with United Airlines from Los Angeles to San Diego. It might seem that this is a simple matter of specifying the disjunctive path query *aircanada* + *united* instead. However, this second query does not differentiate between purely Air Canada routes (that the user might prefer) and the joint partnership routes. Hence, the system should allow the user to specify alternative choices, which can be weighted by her preferences. Furthermore, the system should also be able to present a ranking of the returned answers with respect to the expressed user preferences.

In this paper, we address this problem by proposing weight-enhanced path queries (WEPQ's). Intuitively, we allow the user to specify weighted edge alternatives in path queries. For example, instead of $aircanada^*$, the user could specify the weighted path query

$$(aircanada{:}0 + united{:}1 + usairways{:}3)^* + (aircanada{:}0)^* \cdot (alaska{:}0) \cdot (aircanada{:}0)^*$$

to express her preferences, which in plain language are: "I would like to take Air Canada routes (paths with weight 0), or United Airlines routes but with lower preference (weight 1), or US Airways but with even lower preference (weight 3). Furthermore, I can accept at no cost a single Alaska segment connecting two Air Canada routes."

The second problem with simple path queries is that their semantics do not take into consideration other properties of the traversed links other than the mere link label. To illustrate with the above example, a query such as $aircanada^*$, would return all the pairs of cities connected through an aircanada route without any indication of the length of the path used to compute the answers. This is just an example of what other information might be present at the database links, and clearly, there are other examples of useful information such as frequency of flights for a particular link, time of the day, special service offered, etc. Notably, it is easy to map such information into (virtual) link weights as we navigate the database.

After introducing an "edge importance aware" generalization of graph databases, we formally define the notion of weighted answers to WEPQ's on such databases. The computed weights of the answers enable their ranking according to the user preference for following certain database paths. Moreover, we propose query semantics in which the weight of an automaton transition is further scaled up or down by the importance of the traversed database edge. Thus, in our spatial example, the edge importance could simply be the edge-length, and so, traversing a 1000 *miles united*-edge would be less preferable than traversing a

[3] A route may contain intermediate cities.

300 *miles usairlines*-edge, even though initially in the query, flying with United Airlines was more preferable than flying with US Airways.[4]

Analogous to the airline example, we can also describe our semantics in a Web scenario. Here, the database edges are hyperlinks, and the values assigned to each edge could be based on the inverse of the page-rank of the linked page, using an algorithm similar to the one used to rank pages in a search engine.

In almost every scenario where the data is represented in a graph based fashion, prime examples of which are information integration, Web based information systems, airline reservation systems etc, the data is distributed among many different sources. Clearly, it is imperative to not only show how a query can be evaluated in a centralized way, but also to devise truly distributed algorithms for this purpose. With "truly" in this context we want to say that the data shipping paradigm, commonly used by the today's XQuery processors, is not to be considered as a viable distributed solution. Instead, a query shipping paradigm should be used, where the queries are appropriately decomposed, and where each source works towards accomplishing specific tasks related to the local data only. Hence, we turn our attention to devising a distributed algorithm for the evaluation of our proposed WEPQ's. Our assumption is that the database graph is partitioned into several autonomous processors which do not share memory. Communication between these processors is achieved exclusively through message passing.

In our setting, we tackle the more challenging problem of evaluating a query starting from multiple database nodes. Clearly, multi-source path queries impose a much higher load on the system, since we need to find all the possible paths spelling words in the query language, as opposed to finding paths starting from some root only. Moreover, the database paths that one has to follow starting from different nodes might have a great amount of overlap and a naive processing would do extensive redundant work. To see this, imagine we want to evaluate *aircanada*:0 + *united*:1, starting from Vancouver and Toronto. Such routes will have many intersections such as for example in Los Angeles. An intelligent query processing would not follow more than once sub-paths starting from Los Angeles.

We present a distributed algorithm that completely avoids traversing database paths multiple times. Through an iterative weight-correcting process, our algorithm computes and ranks the query answers. A nice feature of our algorithm is that any snapshot of the query answer at any point in time will be a partial answer to the user query. The practical consequence of this is that the user would very soon see some partial answers to her query, and along the time that she is willing to wait, new answers will show up, while the existing answers will possibly be improved.

Related Work. Similar queries have been also introduced in [2] in the context of Web data. However, our query semantics are more general, and furthe-

[4] We assume that the user is not interested in optimizing the distance traveled. Thus, our problem is not about multi-feature ranking. Rather we assume that the user is interested in optimizing her preferences, which she can tolerate to be weighted by the cost of satisfying them.

more we explore in detail different aspects of handling, generating, and evaluating such queries.

Interestingly, our expression syntax is similar with the one used in [4, 5]. Syntactically, the difference is that in [4, 5], the expressions were on symbol-weight-symbol triples as opposed to symbol-weight pairs that we use. Semantically, the expressions of [4, 5] were used for capturing knowledge or constraints about databases.

We also want to mention here the work of [3] and [9] on how to produce ranked XPath answers in XML information retrieval (IR) systems. In [3] and [9], the focus is on ranking the node answers according to the text contents of the selected (by XPath) XML nodes. Namely, the node texts are ranked using IR relevance criteria. We want to stress here that our approach is very different from [3] and [9]. We rank the node answers according to the "quality" (based on user preferences) of the paths used for reaching the nodes. Thus, we present a graph *structural* approach to ranking the query answers, which is inherently different from IR text approaches of [3] and [9].

Surprisingly, to the best of our knowledge only very few works have dealt with a distributed evaluation of path queries. The most important works in this direction are [1], [11], and [10]. In [1], the architecture is similar with ours but the queries are simple path queries without weights, and furthermore the evaluation is considered starting from some root node only. The single root assumption is also made in [10], focusing in a generalization of path queries that is different from the one in the present paper. Similarly with [1] and [10] , [11] also studies the single root scenario for simple (unweighed) path queries. However, in [11], the main objective is to minimize the number of communication messages and for achieving this [11] suggests an approach, which distributes the load unevenly among processors.

Our distributed query evaluation algorithm is inherently different from the algorithms in [1, 11, 10] because here we solve the more difficult problem of evaluating the query starting from multiple database objects (or all objects).[5] Moreover, our queries are semantically different from those considered in [1, 11, 10].

The rest of the paper is organized as follows. In Section 2, we introduce weight-enhanced path queries (WEPQ's) and their semantics. Section 3 provides a short review of the evaluation of classical path queries and the intuition behind the algorithm for evaluating WEPQ's. In Section 4, we present our distributed algorithm and discuss its message complexity.

2 Databases and Weight-Enhanced Path Queries

We consider a database to be an edge-labeled graph with 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.

[5] Even for simple shortest path problems, the algorithms for multi-source variants are inherently different from the algorithms for single-source variants. Usually, the former are dynamic programming approaches, while the latter are greedy approaches.

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. Words will be denoted by $u, w, \ldots$. 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 introducing weighted preference path queries, it will help to first review the classical path queries.

A *path query* (PQ) is a regular language over Δ. For the ease of notation, we will blur the distinction between regular languages and regular expressions that represent them. Let Q be a PQ and $DB = (V, E)$ a database. Then, the *answer* to Q on DB is defined as

$$Ans(Q, DB) = \{(a, b) \in V : a \xrightarrow{w} b \text{ in } DB \text{ for some } w \in Q\},$$

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

Now, let $\mathbb{N} = \{0, 1, 2, \ldots\}$. A *weight enhanced path query* (WEPQ) is a regular language over $\Delta \times \mathbb{N}$. The "words" of such a language are in fact pairs in $\Delta^* \times \mathbb{N}$. Observe that, given a WEPQ Q, and a word w on Δ, w can be weighted in Q by different real numbers. As an example, it could be that there exist $r_1, r_2 \in \mathbb{R}$ (there can many more such numbers) such that both (w, r_1), and (w, r_2) are in Q. In order to capture the best preference that query Q gives to the word w, we define the *Q-preference* of w, denoted with $pref_{\mathcal{Q}}(w)$, to be the smallest of the values associated with with w in Q.

Next, we define the *weighted answer (WAns)* to a WEPQ Q on DB starting from some object o as

$$WAns(Q, DB) = \{(a, b, k) \in V \times V \times \mathbb{N} : k = inf\,\{pref_{\mathcal{Q}}(w) : a \xrightarrow{w} b \text{ in } DB\}\}$$

Naturally, WEPQ's can be represented by "weighted automata." Formally, a *weighted automaton* $\mathcal{A} = (P, \Delta, \tau, p_0, F)$ consists of a finite set of states P, an input alphabet Δ, an initial state p_0, a set of final states F, and a transition relation $\tau \subseteq P \times \Delta \cup \{\epsilon\} \times \mathbb{N} \times P$.

Given a weighted automaton $\mathcal{A} = (P, \Delta, \tau, p_0, F)$, we say that a word $w \in \Delta^*$ is *accepted through a k-weighted transition path* if there exists a sequence $(p_0, w_1, k_1, p_1), \ldots, (p_{n-1}, w_n, k_n, p_n)$ of state transitions of τ (where $\{w_1, \ldots, w_n\} \subseteq \Delta \cup \{\epsilon\}$), such that $p_n \in F$, $w = w_1 \ldots w_n$, and $k = k_1 + \cdots + k_n$. When referring to a transition path as the above, we could also say that it is a "$(k_1, \ldots, k_n)$-weighted transition path" in order to concisely specify that such a path consists of n transitions with respective weights $k_1, \ldots, k_n$. We denote the set of all accepted words of $\mathcal{A}$ (regardless of transition path followed) by $\mathcal{A}(L)$.

Now, we can equivalently define $pref_{\mathcal{Q}}(w)$ using weighted automata. Thus, let $\mathcal{A}_{\mathcal{Q}}$ be an arbitrary weighted automaton for Q. For w on Δ, the *Q-preference* of w is

$$pref_{\mathcal{Q}}(w) = \begin{cases} inf\,\{k : w \text{ is accepted though a } k\text{-weighted transition path in } \mathcal{A}_{\mathcal{Q}}\} \\ \infty, \text{ if } w \notin \mathcal{A}_{\mathcal{Q}}(L). \end{cases}$$

In our definition of $WAns(Q, DB)$, we do not use the real values that could possibly be assigned to the edges of the database graph. As mentioned in the Introduction, such values can be used to scale up or down the transitions during the query evaluation.

In order to take into consideration the edge values, we define the *scaled Q-preference* of a word w (of length n) through a scaling vector $\mathbf{scale} = (r_1, \ldots, r_n) \in \mathbb{R}^n$, as

$$pref_{\mathcal{Q}}(w, \mathbf{scale}) = \begin{cases} inf\,\{r : w \text{ is accepted though a } (k_1, \ldots, k_n)\text{-weighted} \\ \qquad \text{transition path in } \mathcal{A}_{\mathcal{Q}}, \text{ and } r = k_1 r_1 + \cdots + k_n r_n\} \\ \infty, \text{ if } w \notin \mathcal{A}_{\mathcal{Q}}(L). \end{cases}$$

For a path π in DB, we define $\mathbf{scale}_\pi$ to be the scaling vector obtained in the natural way from the values of each edge along π. We are ready now to define the path *scaled weighted answer (SWAns)* of a WEPQ Q on DB as

$$\begin{aligned} SWAns(Q, DB) = \{(a, b, r) \in V \times V \times \mathbb{R} : \\ k = inf\,\{pref_{\mathcal{Q}}(w, \mathbf{scale}_\pi) : \\ \pi \text{ is a path } a \xrightarrow{w} b \text{ in } DB\}\}. \end{aligned}$$

Next, we show how to efficiently transform a weighted automaton $\mathcal{A}$, into one with ϵ-free transitions, in such a way that the essential features of $\mathcal{A}$ are preserved. The ϵ-freeness is essential in properly computing the answer of a WEPQ.

From the automaton $\mathcal{A}$ we will construct another "weight equivalent" automaton $\mathcal{B}$. We will use $\epsilon\text{-}closure(p)$, similarly to [6], to denote the set of all states q, such that there is path π, from p to q in $\mathcal{A}$, spelling ϵ.

Obviously, we will keep all the non-ϵ transitions of $\mathcal{A}$ in the automaton $\mathcal{B}$, that we are constructing.

Now, we will insert an R-transition ($R \neq \epsilon$) in $\mathcal{B}$ from a state p to a state q whenever there is in $\mathcal{A}$ a path π, spelling ϵ, from p to an intermediate state r and there is an R-transition, from that state r to the state q. We take special care here for computing the weight of these transitions. Formally, if $\mathcal{A} = (P, \Delta, \tau_{\mathcal{A}}, p_0, F)$, then $\mathcal{B} = (P, \Delta, \tau_{\mathcal{B}}, p_0, G)$, where $G = F \cup \{p_0 : \text{ if } \epsilon\text{-}closure_{\mathcal{A}}(p_0) \cap F \neq \emptyset\}$ and

$$\begin{aligned} \tau_{\mathcal{B}} = \{(p, R, n, q) : (p, R, n, q) \in \tau_{\mathcal{A}}\} \cup \\ \{(p, S, m + n, q) : \exists r \in \epsilon\text{-}closure_{\mathcal{A}}(p), \\ \text{such that } (r, S, n, q) \in \tau\}, \end{aligned}$$

where the weight m is the weight of the cheapest path from p to r in $\mathcal{A}$ spelling ϵ, and (r, S, n, q) is the cheapest S-transition from r to q.

It is easy to verify about the above constructed automaton $\mathcal{B}$ that

Theorem 1. *For a given word w, there is k-weighted transition path spelling w in $\mathcal{B}$ if and only if there is such a path in $\mathcal{A}$.*

Also observe that in the above construction, although there can be exponentially many ϵ-paths from state p to state r, we insert only one transition from p to q for each symbol labeling a transition from r to q. Hence, we have that the size of $\mathcal{B}$ is polynomial on the size of $\mathcal{A}$. Moreover, note that there can be many transitions from r to s labeled with the same symbol but having different weight. However, we only consider the cheapest of them.

3 Path Queries, Reachability, and Shortest Paths

Before presenting the distributed (weighted) query evaluation, we will review the classical method of query evaluation. In essence, the evaluation proceeds by creating state-object pairs from the query automaton and the database. For this, let $\mathcal{A}$ be an NFA that accepts a query Q. Starting from some 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 exist 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 the existing ones, until we are not anymore able to do so. At that point, we produce as the answer to the query the set of object pairs (a, b), such that b has been associated with some final state of the query automaton $\mathcal{A}$.

It is worth mentioning here that the state-object pairs induce an (implicit) edge labeled graph with these pairs as its nodes. Regarding the edges, let (q, b) be obtained by another pair, say (p, a), through a database edge and automaton transition both labeled by R. Then, we consider an R-labeled edge from (p, a) to (q, b) in the induced graph.

Now, when having a weighted query automaton, we can modify the classical matching algorithm to build instead a weighted state-object graph. This can be achieved by assigning to the edges of this graph the corresponding automaton transition weights scaled by the corresponding database edge-values.

It is not difficult to see that, in order to find the weighted answers to the query, we have to find, in the state-object graph, the shortest paths from the "sources" (p_0, a) to all the nodes (p, b), where p is a final state in the query automaton $\mathcal{A}$.

However, the challenge is that when the database is very large and distributed, we cannot afford to construct the above graph, and then use some centralized shortest path algorithm on it.

In the next section, we present a distributed algorithm which computes the multi-source shortest paths in the state-object graph, while constructing it on the fly and achieving all the possible overlap in the computations starting from each source.

4 Distributed Evaluation of Path Queries

Our algorithm has two interwoven components: the computation of query answers and its termination detection. In this paper, we focus on the computation

of query answers. The termination detection will be discussed in a companion paper.

The central idea of our algorithm is to overlap computations starting from different objects. We assume that each database object, say a, is being serviced by a dedicated process for that object P_a.

Each process starts by creating an initial task for itself. The tasks are "keyed" by the automaton states, with the initial tasks being keyed by the initial state, say p_0. Each task $\langle p, \ldots \rangle$ corresponding to some object a (serviced by P_a), is eventually selected for "expansion," which is the creation and sending of new tasks to other processes whenever there is an automaton transition originating at state p that matches a database edge originating at the object a. Let (p, R, q, k) be such a transition matching a database edge (a, R, b, t). Then the process P_a will send the task $\langle q, \ldots \rangle$ to the process P_b servicing the object b. The process P_b, upon receival of the task $\langle q, \ldots \rangle$, will establish a virtual communication channel with the process P_a for the originating task $\langle p, \ldots \rangle$. This channel is weighted by kt. In a sense, the completion of the task $\langle p, \ldots \rangle$ in P_a depends on the completion of the task $\langle q, \ldots \rangle$ in P_b.

Notably, overlapping of computations happens when a process receives the same task multiple times from different neighboring processes. In such a case the receiving process does not accept the "new" task, but instead it creates only a virtual communication channel with the sending process as explained above. The overlapping of computations is reminiscent of view-based optimization of queries.

Whenever a process receives (the first time) a task keyed by a final state, it sends as an answer, through the backward communication channels, the *id* of the object that it services. The receiving processes will back-propagate such answers, through their *appropriate* backward communication channels, satisfying along the way the p_0 keyed tasks. The meaning of "appropriate" will become clear in the detailed algorithm that follows. The back-propagated answers will be weighted by the cost of traversing the communication channel paths. Recall that the cost of a communication channel is the product of the corresponding automaton transition and database edge weights. At the joint points, i.e. at the tasks receiving the same answer from different paths, we "relax" the weight of the answer by setting it equal to the smallest received weight.

Now, we formally present our algorithm, which, as we mentioned earlier, emphasizes the distributed computation of query answers ignoring (for the moment) the termination detection of the computation.

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 weighted automaton $\mathcal{A} = (P, \Delta, \tau, p_0, F)$ for a WEPQ Q. This automaton is sent first to all the processes.[6]

[6] This does no harm to the true distribution of computation. Instead of transferring whole parts of the data (as in the XQuery paradigm), we simply send (once only)

Output: The weighted answers to the query Q evaluated on the database DB.

Method: Each process P_a creates a task $\langle p_0, \{\}, \texttt{unexpanded}\rangle$ for itself. If p_0 is also a final state, insert $(a, 0)$ in the pair-set of the above p_0-task, which would become $\langle p_0, \{(a, 0)\}, \texttt{unexpanded}\rangle$.

1. Repeat 2, 3, and 4 at each process P_a in *parallel*, until termination is detected.
2. For each unprocessed yet task $\langle p, \{\ldots\}, \texttt{unexpanded}\rangle$
   (a) **if** there is a t-weighted R-edge, from object a to some object b in DB, and there is a k-weighted R-transition from state p to some state q in automaton $\mathcal{A}$
   then (P_a will expand the task to P_b)
   P_a creates a message $\langle a, p, q, u\rangle$, where $u = kt$, and sends it to process P_b.
   else P_a is "stuck" with respect to this task, and sends an empty message $\langle\rangle$ through each communication channel (see step 3) $\langle (a, p), (\_, \_), x\rangle$, to the processes at the other end of the channels.
   (b) Task $\langle p, \{\}, \texttt{unexpanded}\rangle$ will change to `expanded`.
3. Upon receival of a message $\langle c, r, p, v\rangle$ (due to expansion of an r-keyed task of P_c)
   (a) **if** P_a does not have yet a task $\langle p, \{\ldots\}, \_\rangle$
   then P_a creates a corresponding task $\langle p, \{\}, \texttt{unexpanded}\rangle$ for itself, and establishes a virtual communication channel $\langle (a, p), (c, r), v\rangle$ between this task and the r-keyed task of P_c.
   if p is a final state
   then P_a inserts $(a, 0)$ in the pair set of $\langle p, \{\}, \texttt{unexpanded}\rangle$, which becomes $\langle p, \{(a, 0)\}, \texttt{unexpanded}\rangle$.
   else (P_a has already a task $\langle p, \{\ldots\}, \_\rangle$)
   If there is not already a communication channel $\langle (a, p), (c, r), v\rangle$, create one as above.
   (b) Next, for each object-weight pair (d, y) in the p-task pair-set, P_a sends a message $\langle d, y + v\rangle$ through the communication channel $\langle (a, p), (c, r), v\rangle$.
4. Upon receival of a message $\langle e, z\rangle$ through some channel, say $\langle (\_, \_), (a, p), y\rangle$:
   (a) P_a checks whether there exists a pair $(e, \_)$ in the p-task pair-set. If there exists such a pair, say (e, w), update (relax) it with $(e, \texttt{min}\{z, w\})$; otherwise insert (e, z).
   (b) **if** in step (a) a change, which is an update or insertion, did happen (say we got (e, z) in the above pair-set)
   then P_a propagates the change upwards by sending through each channel $\langle (a, p), (\_, \_), v\rangle$ the message $\langle e, z + v\rangle$.
   else (when in step (a) there was no change) P_a does nothing. It has already good object weights for the p-keyed task, which have been (or will be soon) propagated upwards.

---

the query automaton, which is polynomial in the size of the regular expression given by the (human) user.

Finally upon termination, which happens when there are no more messages sent but not yet received, set

$$eval(\mathcal{A}, DB) = \{(a, b, r) : (b, r) \text{ is in the pair-set of the } p_0\text{-task of process } P_a\}.$$

It is easy to verify that the following theorem is true.

Theorem 2. *Upon termination of the above algorithm, we have that*

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

Now, we illustrate the Algorithm 1 by the following example. Consider the database and the query automaton in Figure 1, *left* and *right* respectively. Due to space constraints, we have abbreviated *unexpanded* by u, and *expanded* by e. A possible sequence of actions for Algorithm 1 is given in Table 1. In the first

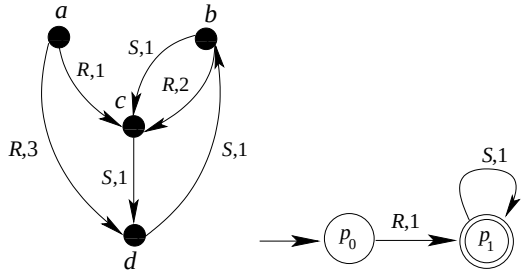

Fig. 1. A database and a query automaton

column labeled "T" we number the hypothetical time (break)points in which we observe the system. An explanation for the (possible) actions at each time point follows.

1. All the processes create a task $\langle p_0, \{\}, u\rangle$ for themselves.
2. P_a expands the tasks $\langle p_0, \{\}, u\rangle$ and sends the task $\langle p_1, \{\}, u\rangle$ to both P_c and P_d. P_c and P_d observe that p_1 is a final state and insert $(c, 0)$ and $(d, 0)$ respectively in their p_1-task pair-set. Next, P_c and P_d send $\langle c, 1\rangle$ and $\langle d, 3\rangle$ respectively to P_a through the appropriate virtual channels.
3. P_b expands the tasks $\langle p_0, \{\}, u\rangle$ and sends a p_1-task to P_c. Since P_c has already received such a task before (from P_a), it does not create a new task, but only establishes a virtual channel with P_b for the originating p_0-task. Also, P_c sends $\langle c, 2\rangle$ to P_b.
4. P_c expands the tasks $\langle p_0, \{\}, u\rangle$. It gets stuck.
5. P_c expands the tasks $\langle p_1, \{\}, u\rangle$ and sends a p_1-task to P_d. Since P_d has already received such a task before, it does not create a new task, but only establishes a virtual channel with P_c for the originating p_1-task. Also, P_d sends $\langle d, 1\rangle$ to P_c. P_c in turn sends $\langle d, 2\rangle$ to P_a, and $\langle d, 3\rangle$ to P_b. P_a will update(relax) the weight for d from 3 to 2.
6. P_d expands the task $\langle p_0, \{\}, u\rangle$. It gets stuck.

7. P_d expands the tasks $\langle p_1, \{(d,0)\}, u\rangle$ and sends a p_1-task to P_b. P_b observes that p_1 is a final state inserts $(b,0)$ in its p_1-task pair-set. Also, P_b sends $\langle b, 1\rangle$ to P_d through the appropriate virtual channel. P_d propagates this new answer by sending $\langle b, 4\rangle$ to P_a, and $\langle b, 2\rangle$ to P_c.
8. Further computation occurs leading, upon termination, to this final snapshot.

T	P_a	P_b	P_c	P_d
1	$\langle p_0, \{\}, u\rangle$	$\langle p_0, \{\}, u\rangle$	$\langle p_0, \{\}, u\rangle$	$\langle p_0, \{\}, u\rangle$
2	$\langle p_0, \{(c,1),(d,3)\}, e\rangle$	$\langle p_0, \{\}, u\rangle$	$\langle p_0, \{\}, u\rangle$ $\langle p_1, \{(c,0)\}, u\rangle$	$\langle p_0, \{\}, u\rangle$ $\langle p_1, \{(d,0)\}, u\rangle$
3	$\langle p_0, \{(c,1),(d,3)\}, e\rangle$	$\langle p_0, \{(c,2)\}, e\rangle$	$\langle p_0, \{\}, u\rangle$ $\langle p_1, \{(c,0)\}, u\rangle$	$\langle p_0, \{\}, u\rangle$ $\langle p_1, \{(d,0)\}, u\rangle$
4	$\langle p_0, \{(c,1),(d,3)\}, e\rangle$	$\langle p_0, \{(c,2)\}, e\rangle$	$\langle p_0, \{\}, e\rangle$ $\langle p_1, \{(c,0)\}, u\rangle$	$\langle p_0, \{\}, u\rangle$ $\langle p_1, \{(d,0)\}, u\rangle$
5	$\langle p_0, \{(c,1),(d,2)\}, e\rangle$	$\langle p_0, \{(c,2),(d,3)\}, e\rangle$	$\langle p_0, \{\}, e\rangle$ $\langle p_1, \{(c,0),(d,1)\}, e\rangle$	$\langle p_0, \{\}, u\rangle$ $\langle p_1, \{(d,0)\}, u\rangle$
6	$\langle p_0, \{(c,1),(d,2)\}, e\rangle$	$\langle p_0, \{(c,2),(d,3)\}, e\rangle$	$\langle p_0, \{\}, e\rangle$ $\langle p_1, \{(c,0),(d,1)\}, e\rangle$	$\langle p_0, \{\}, e\rangle$ $\langle p_1, \{(d,0)\}, u\rangle$
7	$\langle p_0, \{(b,4),(c,1),(d,2)\}, e\rangle$	$\langle p_0, \{(c,2),(d,3)\}, e\rangle$ $\langle p_1, \{(b,0)\}, u\rangle$	$\langle p_0, \{\}, e\rangle$ $\langle p_1, \{(b,2),(c,0),(d,1)\}, e\rangle$	$\langle p_0, \{\}, e\rangle$ $\langle p_1, \{(d,0),(b,1)\}, e\rangle$
8	$\langle p_0, \{(b,3),(c,1),(d,2)\}, e\rangle$	$\langle p_0, \{(b,4),(c,2),(d,3)\}, e\rangle$ $\langle p_1, \{(b,0),(c,1),(d,2)\}, e\rangle$	$\langle p_0, \{\}, e\rangle$ $\langle p_1, \{(b,2),(c,0),(d,1)\}, e\rangle$	$\langle p_0, \{\}, e\rangle$ $\langle p_1, \{(b,1),(c,2),(d,0)\}, e\rangle$

Table 1. A possible sequence of snapshots for Algorithm 1

It is worth mentioning 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. The answer would be partial because: (a) there could still be objects to be discovered during the navigation, and (b) the weights of the already discovered objects could be further improved, i.e. lowered, should we wait further for the algorithm to continue. However, depending on the application a "quicker" partial answer could be more desirable than the complete answer.

Complexity. Here we are interested in the number of messages since to send a message is an order of magnitude more expensive than to perform a main memory operation. We show the following.

Theorem 3. *The number of messages in Algorithm 1 is bounded by* $(E \cdot |\tau|)^2$*, where* E *is the number of edges in* DB*, and* $|\tau|$ *is the cardinality of the transition relation of* $\mathcal{A}$*.*

Proof Sketch. Each physical database edge can "accommodate" in the worst case $|\tau|$ virtual communication channels. To set up a communication channel one "forward" message is needed. Now the question is how many times a communication channel is traversed "backward," by the update $\langle d, y\rangle$ messages. It is not difficult to see that a new wave of possible update messages can be propagated backwards for each forward message $\langle c, r, p, v\rangle$ in step 3 of the algorithm. As a wave of updates could have in the worst case up to $E \cdot |\tau|$ backward messages, we get the claimed upper bound. □

We can make a tradeoff between the time the nodes wait before initiating an update-wave, and the query response time. If each process waits a certain time

before back-propagating query answers (as opposed to immediately sending all the answers that the process knows), then it is possible that better weighted answers will arrive to the processes, and many back-propagation messages will be cancelled. Not sending right away might be good for "throughtput" when we have a big number of executing queries. Such a strategy might significantly reduce the stress to the system making it possible to execute faster a set of queries.

As mentioned before, (due to space constraints) we do not present here the termination detection algorithm. It will be found in a future companion paper.

References

1. S. Abiteboul, V. Vianu. Regular Path Queries with Constraints. *Journal of Computing and System Sciences* 58(3) 1999, pp. 428-452.
2. Flesca, S., Furfaro, F., and Greco, S. Weighted path queries on web data. *In Proceedings of the 4th International Workshop on the Web and Databases (WebDB '01).* Informal Proceedings, pp. 7–12.
3. Grabs T., and H.-J. Schek ETH Zürich at INEX: Flexible Information Retrieval from XML with PowerDB-XML *Proc. INEX Workshop 2002: pp. 141–148*
4. Grahne, G., and Thomo, A. Approximate reasoning in semistructured data. *In Proceedings of the 8th International Workshop on Knowledge Representation meets Databases (KRDB '01).* Online Proceedings, http://ceur-ws.org/Vol-45/03-thomo.ps
5. Grahne, G., and Thomo, A. Query answering and containment for regular path queries under distortions. *In Proceedings of the 3rd International Symposium on Foundations of Information and Knowledge Systems (FoIKS '04).* Lecture Notes in Computer Science 2942, Springer, 2004, pp. 98–115.
6. Hopcroft, J., E., and Ullman, J., D. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.
7. Mendelzon, A., O., Mihaila, G., A., and Milo, T. Querying the World Wide Web. *International Journal on Digital Libraries,* 1 (1), 1997, pp. 57–67.
8. Mendelzon A. O., and P. T. Wood, Finding Regular Simple Paths in Graph Databases. *SIAM J. Comp.* 24 (6) : 1235–1258, 1995.
9. Meyer, H., I. Bruder, A. Heuer, and G. Weber The Xircus Search Engine. *INEX Workshop 2002, pp. 119–124*
10. Stefanescu D. C., Thomo, A, and Thomo, L. Distributed evaluation of generalized path queries *Proc. Proceedings of the 2005 ACM Symposium on Applied Computing* 2005 pp. 610–616.
11. Suciu D., Distributed query evaluation on semistructured data. *ACM Transactions on Database Systems,* 27 (1), 2002, pp. 1–62.
12. Vardi. M. Y. A Call to Regularity. *Proc. PCK50 - Principles of Computing & Knowledge, Paris C. Kanellakis Memorial Workshop '03*, pp. 11.
13. Lacroix, Zoe and Raschid, Louiqa and Naumann, Felix and Vidal, Maria Esther. Exploring Life Sciences Data Sources Technical report 2003.