Distributed Multi-Source Regular Path Queries

by

Maryam Shoaran
B.Sc. University of Shahid Beheshti 2005

A Thesis Submitted in Partial Fullfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science, Faculty of Engineering

© Maryam Shoaran, 2007
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.

Distributed Multi-Source Regular Path Queries

by

Maryam Shoaran
B.Sc. University of Shahid Beheshti 2005

Supervisory Committee

Dr. Alex Thomo (Department of Computer Science)
Supervisor

Dr. William Wadge (Department of Computer Science)
Departmental Member

Dr. Venkatesh Srinivasan (Department of Computer Science)
Departmental Member

Dr. Ahmed E. Hassan (Department of Electrical Engineering)
Outside Member

Supervisory Committee

Dr. Alex Thomo (Department of Computer Science)
Supervisor

Dr. William Wadge (Department of Computer Science)
Departmental Member

Dr. Venkatesh Srinivasan (Department of Computer Science)
Departmental Member

Dr. Ahmed E. Hassan (Department of Electrical Engineering)
Outside Member

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 thesis 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 has several desirable properties. First, it is general as it works for the larger class of weighted regular path queries on weighted semistructured databases. Second, it performs a progressive evaluation, that is, partial answers can be represented to the user as soon as they are computed, while she is waiting for new answers to arrive. Third, the proposed algorithm is symmetric among processes, i.e., they all run the same algorithm. And finally, it does not need a separate termination detection algorithm as it can detect the global termination simply by using an spanning tree.

Table of Contents

Supervisory committee ii

Abstract iii

Table of Contents iv

Acknowledgements vi

1. Introduction 1
   - 1.1 Graph Databases and Applications 1
     - 1.1.1 Spatial Databases 2
     - 1.1.2 Citation Data 4
     - 1.1.3 Web Site Data 6
   - 1.2 Generalized Regular Path Queries (RPQ's) 9
     - 1.2.1 Two Kinds of Weights 9
     - 1.2.2 Single Source vs. Multiple Sources 11
   - 1.3 Distributed Strategy 12
   - 1.4 Organization 15

2. Formalization 16
   - 2.1 Databases 16
   - 2.2 Classical RPQ's 16
   - 2.3 Weighted RPQ's 18
   - 2.4 Outline of Query Evaluation 19

3. Distributed Algorithm 21
   - 3.1 Informal Description 21
   - 3.2 Formal Algorithm 26
   - 3.3 Detailed Example 31
   - 3.4 Reflections 37

4. Termination 40
4.1 Impossibility of Deadlock 40
4.2 Termination Detection 43
5. Complexity 44
5.1 Message Complexity 44
5.2 Discussion 45
6. Soundness and Completeness 46
6.1 Some Observations and a Fact 46
6.2 Soundness 48
6.3 Completeness 48
7. Conclusions and Future Work 50

Acknowledgements

I would like to express my deepest gratitude, first and foremost, to my supervisor, Dr. Alex Thomo, for his leadership, understanding, and patience throughout my graduate career. I appreciate his expertise in the area, which provided insights that guided and challenged my thinking, also his kind support and assistance in writing this thesis.

I would also like to acknowledge Dr. W. Wadge and Dr. V. Srinivasan, as well as Dr. A. E. Hassan from Electrical Engineering Department for taking time out from their busy schedules to serve as my committee members. I am extremely thankful of Dr. V. Srinivasan for reading the initial version of the paper, on which this thesis is based, and providing useful comments.

Special thanks go towards my family for the support they provided me through my entire life. Without their love, patience, and encouragement I would never have been able to complete my educational pursuits.

Above all, I will thank Allah for giving me the knowledge, vision, and ability to proceed this challenge. It is only through his grace that achievements can truly be accomplished.

Chapter 1

Introduction

1.1 Graph Databases and Applications

Semistructured data is the foundation for a multitude of applications in many important areas such as information integration, Web and communication networks, spatial networks, biological data management, etc (cf. [1]). 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, 17, 5, 6, 7]), regular path queries (RPQ's) are the "winner" when it comes to expressing navigational recursion over semistructured data[1]. Regular path queries are in essence regular expressions over the database edge symbols. In general, one is interested in finding query-matching database paths, which spell words in the (regular) query language.

In the following we show three examples of useful graph databases and some typical queries on them.

[1]Some form of recursion is possible to be expressed using some extensions introduced in the SQL–99 standard [9]. However, it is only implemented in one major system, IBM's DB2.

1.1.1 Spatial Databases

Taking an example from spatial network databases such as TIGERLine [25] (see Figure 1.1 for a visualization), 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 a regular path query

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

where $\parallel$ is the shuffle operator (see e.g. [14]).

In this and other examples, there might be weights associated with the database edges. Also, the user might assign preference weights to the query elements. It is quite natural to extend regular path queries to take into consideration such weights during evaluation. We informally explain this in the next section, while in Section 2.3 we formally define the notion of weighted query answers.

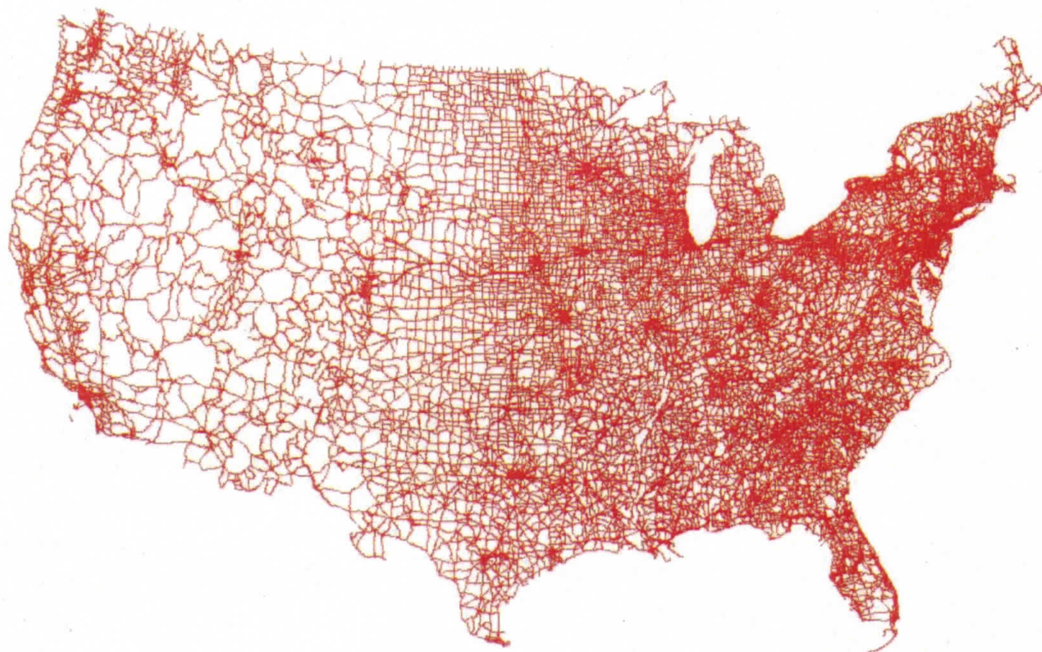

Figure 1.1: TIGERLine spatial database

1.1.2 Citation Data

As another example, consider the citation graph derived from The National Library of Medicine's (NLM) MEDLINE/PubMed biomedical literature database,

`http://www.nlm.nih.gov/bsd/licensee/2006_baseline_doc.html.`

This database is rapidly expanding and as of the 2006 edition, the baseline database contains more than 15 million citations. A fragment of the MEDLINE/Pub–Med citation graph is shown in Figure 1.2. The nodes of the graph are chemical terms. There is an edge between two terms A and B if there is some article in which A and B co-occur. The edges and little circles on the edges are color coded depending on the number of articles:

gray the terms co-occur in 1 article

yellow co-occurrence in 2-5 articles

pink co-occurrence in 6-10 articles

green co-occurrence in >10 articles.

The circles on the edges are in essence the exact co-occurrence weights of the connected chemical terms. They should not be confused with the nodes of the graph.

In the lower panel of the same figure, a magnified region of the graph is shown for clarity.

Figure 1.2: MEDLINE/PubMed citation graph

Several interesting regular path queries can be expressed for this domain. For example the user can ask for all the pairs of chemical terms connected by *pink* edges with at most one intervening *yellow* edge. For this, the user would give the regular path query:

$$Q = pink^* \cdot (yellow + \epsilon) \cdot pink^*.$$

1.1.3 Web Site Data

Finally, we want to show a third example from the Web site domain. Here, the nodes of the graph are websites and the edges are the HREF links (see Figure 1.3 for a fragment of such a graph). In this example, the nodes are in fact labeled as follows:

green	site in the *org* domain
blue	site in the *com* domain
red	site in the *gov* domain
orange	site in the *uk* domain.
dark green	site in the *net* domain.

Observe that we can easily trasform such a graph into an edge-labeled graph for being consistent with the other examples. It can be done by introducing for each node a self loop labeled with the same color as the node, and labeling the internode edges with ϵ.

An interesting regular path query in this domain is for example

$$Q = red^{+} \cdot green^{+},$$

which would give all the pairs (a, b) of web sites, where a is a govermental site, b is a noncommercial-nonprofit organization site, and b can be reached from a by hopping only through other *gov* or *org* sites. In other words, this query asks for all the (a, b) pairs, where b is transitively "trusted" or "recommended" by a.

Figure 1.3: A Web site interconnection graph

1.2 Generalized Regular Path Queries (RPQ's)

1.2.1 Two Kinds of Weights

From the examples in the previous section, we can see that quite often there can be weights associated with the edges (or nodes) of the database. For instance,

1. In the spatial data example, the weights of the edges are the kilometric costs to traverse the edges.

2. In the MEDLINE/PubMed data example, the weights are the exact co-occurrence numbers.

3. In the Web sites example, the weights can be derived by considering the reputation score of the Web sites.

On the other hand, the user can assign preference weights to the query elements. In this thesis, we consider generalized RPQ's with preference weights as introduced in [12]. 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.

Regarding the MEDLINE/PubMed biomedical citation example, the user can write

$$Q = (\mathit{pink} : 1)^* \cdot (\mathit{yellow} : 2 + \epsilon) \cdot (\mathit{pink} : 1)^*$$

to express that she prefers *pink* edges representing co-occurrence in 6–10 articles over *yellow* edges, which indicate co-occurrence in 2–5 articles.

Finally, regarding the Web graph example, the user can write

$$Q = (red : 1)^* \cdot (green : 2)^*,$$

to express that she prefers *red* edges, i.e., govermental sites, over *green* edges, which represent nonprofit organizations.

Now, the database edge weights can be naturally combined with query preferences when there exists a meaningful "multiplication operator" which combines the two kinds of weights. Of course, it is completely the choice of the user to specify query preferences, or to indicate her interest in considering the importance of traversed database edges.

In the spatial case, the combination of weights can be done by simply multiplying the edge weight by the query element weight for each "database edge label – query element" match. Thus, for example, 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][2].

In the MEDLINE/PubMed biomedical citation case, in order to emphasize that the more co-occurrences an edge represents the more important it is, we can consider the inverse of the co-occurrence weights. Thus, for example, a *pink* edge representing

[2]This is because $49 \cdot 2 < 100 \cdot 1$.

10 co-occurrences would be more important than an another *pink* edge representing 6 co-occurrences. Notably, combining these (inverse) edge weights with query preference weights can again be done using simple multiplication [for each "database edge label – query element" match].

Finally, the Web sites case is very similar to the MEDLINE/PubMed case, and we again could consider here the inverse of the site reputation values as the database weights, and still use simple multiplication to combine them with query preference weights.

1.2.2 Single Source vs. Multiple Sources

Based on query-matching paths, there are two ways of defining the answer to an RPQ. The first is the single-source variant [1, 2], 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 [17, 5, 6, 7, 12], where the answer is defined to be the set of *pairs* of objects that are connected by some query-matching paths.

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 the 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.

While both variants have their applications, we believe that the second variant is more general and useful. If we visualize the edges of the database graph as binary relations, and RPQ's as restricted (possibly with recursion) datalog queries, then the first variant imposes an equality constraint in front of the query, while the second one does not. Moreover, even if a user asks for the evaluation of an RPQ starting from a source that she desires, other users might want the evaluation to start from different sources. Thus, a proactive (or speculative) database system might decide to pursue a multi-source evaluation in the first place in order to cache answers that could possibly be required later.

In this thesis, we focus on the second variant of generalized RPQ's.

1.3 Distributed Strategy

As the main applications based on semistructured data are distributed, we look at RPQ's from a distributed strategy angle. Notably, when it comes devising a distributed algorithm, one is concerned about the number of messages exchanged among the participating machines. This is because sending a message is much more expensive than performing main memory computations.

Now, returning to the single-source versus multi-source discussion, one would

think that the second variant needs more messages than the first. However, contrary to such intuition, we show that this is not (necessarily) true in a distributed setting. Namely, we present a distributed algorithm with a message-number complexity which is asymptotically equal to the lower bound of the message-number complexity for evaluating single-source RPQ's.

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 Floyd-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 [13], 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 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 knowledge only about their neighbors. 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 to the Dijkstra's methodology. However, at the same time we allow multiple expansions at different processes, which makes the algorithm truly distributed.

Another interesting feature of our algorithm is that it does not need a separate termination detection thread. Termination detection is one of the widely studied problems in distributed computing. This is a challenging problem and the general solutions to it are achieved by running a separate (from the main processing) termination detection algorithm (cf. [16]). On the contrary, our algorithm is such that it can itself naturally detect the termination without the need of some extra overhead.

To the best of our knowledge, only very few works present a distributed evaluation of regular path queries. In [23], a distributed algorithm is presented, which works based on local knowledge only. However, it has a message complexity which is

quadratically worse than the complexity of our algorithm proposed in this thesis.

Besides [23], other works that have delt with distributed RPQ's are [2, 24, 22, 18]. All four consider the single-source variant of RPQ's.

Finally, regarding the usefulness of weighted RPQ's, we refer the reader to [10, 11, 23, 12], which study such queries in a multitude of important applications.

1.4 Organization

The rest of the thesis is organized as follows. Chapter 2 is dedicated to the underlying definitions of regular path queries and their extensions. In Chapter 3, we present our distributed algorithm. Next, in Chapter 4 and 5, we discuss its termination and complexity, respectively. Then, in Chapter 6, we show the soundness and completeness of our algorithm. Finally, Chapter 7 concludes the thesis.

Chapter 2

Formalization

2.1 Databases

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 by $R, S, \ldots$. 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 by $a, b, c, \ldots$. 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 $\mathbb{R}^+$.

As an example consider the database shown in Figure 2.1.

2.2 Classical RPQ's

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

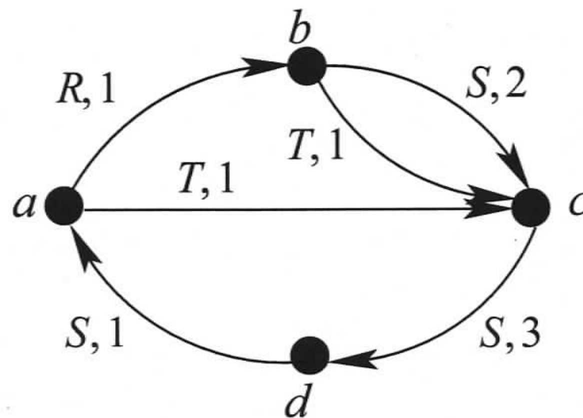

Figure 2.1: A graph database DB

RPQ is a Finite State Automaton (FSA)

$$\mathrm{A} = (P, \Delta, \tau, p_0, F),$$

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.

As an example, consider the database DB and query automaton $\mathcal{A}$ in Figure 2.1

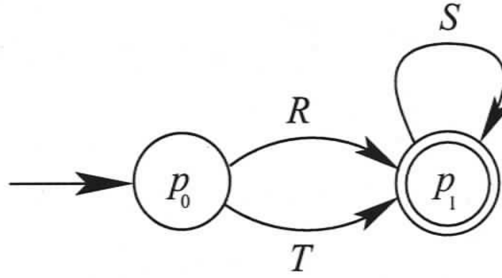

Figure 2.2: A classical query automaton $\mathcal{A}$

and Figure 2.2, respectively. For this query, we have that

$$Ans(\mathcal{A}, DB) = \{(a,b),(a,c),(a,d),(a,a),(b,c),(b,d),(b,a)\}.$$

2.3 Weighted RPQ's

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 [3] 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

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

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

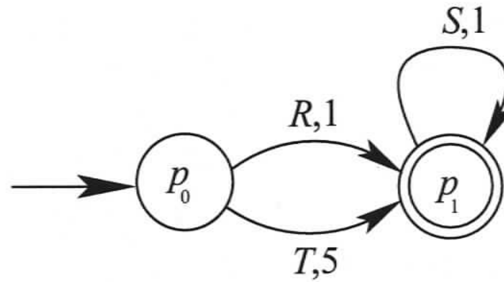

Figure 2.3: A query automaton $\mathcal{A}$

As an example, consider the database DB and query automaton $\mathcal{A}$ in Figure 2.1 and Figure 2.3, respectively. 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 objects a and c.

Similarly, we find the other query answers and finally have

$$SWAns(\mathcal{A}, DB) \;=\; \{(a,b,1),(a,c,3),(a,d,6),(a,a,7),(b,c,5),(b,d,8),(b,a,9)\}.$$

2.4 Outline of Query Evaluation

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 object-state pairs from the database and the query automaton. For this, let $\mathcal{A}$ be a query FSA. Starting from an object a of a database DB, we first

create the pair (a, p_0), where p_0 is the initial state in $\mathcal{A}$. Then, we create all the pairs (b, p) such that there exists an edge from a to b in DB and a transition from p_0 to p in $\mathcal{A}$, and furthermore the labels of the edge and the transition 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 constructed. 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 (b, p), where p is a final state, from (a, p_0), where p_0 is the intial state. Namely, if (b, p) is reachable from (a, p_0), then (a, b) is a tuple in the query answer.

Now, having 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 (a, p_0) to all (b, p), 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.

Chapter 3

Distributed Algorithm

3.1 Informal Description

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 still 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 below).

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 Chapter 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 label symbol) 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 task requests 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[1]. 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:

[1]Indeed, *provisional* tuples in this table are organized in a priority queue.

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 status of 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 back to P_a.

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 process 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

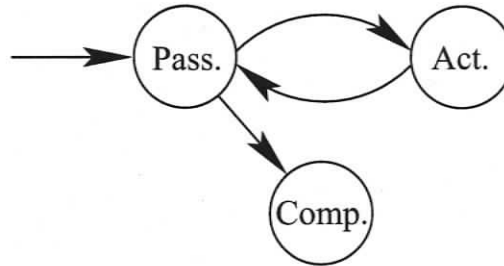

Figure 3.1: Task Status Diagram.

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 table (message) 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) change the status of the p_x–task to *passive*, (2) update (relax) the *provisional* tuples in $P_a.p_x.T$ as appropriate if there are tuples with the same keys in $P_b.p_y.T^*$, and (3) add to the 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 $P_a.p_x.T$.

Figure 3.1 illustrates different possible statuses that a task can have during the execution of the algorithm. At the moment of creation, each task has *passive* status. If

a *passive*–status task doesn't have any *provisional* tuple in its table, the corresponding process changes its status to *completed*. Otherwise, the process can start expansion of the *provisional* tuples in the task table. Starting expansion of a tuple, task status is changed to *active* which, as mentioned in Expansion thread, prevents expansion of other *provisional* tuples until receiving reply to the last request message. When an *active*–status task receives a reply message for recent expansion it starts Update thread, at the end of which the status of task is changed to *passive* making the task ready for another expansion. So, *passive* and *active* statuses can be changed to each other several times during the execution of the algorithm, but the status of a *completed* task will not change any more.

It is also worth noting here that the change in status of a tuple, which can only be from *provisional* to *optimal*, happens when corresponding task has *active* status.

3.2 Formal Algorithm

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 $\langle p_0, \textit{passive}, \{\ldots\}\rangle$ for itself. The table $\{\ldots\}$ (referred to as $P_a.p_0.T$) is initialized as follows:

   (a) insert tuple $[(a, p_0), (a, p_0), 0, \textit{opt}]$, and

   (b) For each edge-transition match,

      (a, R, r, b) in DB and

      (p_0, R, k, p) in $\mathcal{A}$,

      insert tuple $[(b, p), (b, p), k \cdot r, \textit{prov}]$

      (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 and 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 task 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 $\langle p_y, [p_x, (c, p_z), w_{ab}] \rangle$ from P_a:

if there is not yet a p_y–task

then create a task $\langle p_y$, *passive*, $\{\ldots\}\rangle$ and initialize its table similarly as in the first phase.

That is,

(a) insert tuple $[(b, p_y), (b, p_y), 0, opt]$, 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 $[(d, p_u), (d, p_u), k \cdot r, prov]$

(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 request 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 have 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 the 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 $[(d, p_u), (_, _), v, s]^2$ in $P_b.p_y.T^*$, which has a smaller weight (v) than a same-key tuple $[(d, p_u), (_, _), _, prov]$ 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).

(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

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

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

In Chapter 6, 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.

3.3 Detailed Example

We illustrate Algorithm 1 by a detailed example. This example is also useful to illustrate some additional points, which we discuss in the next section.

Consider the database and query automaton in Figure 3.2 and Figure 3.3, respectively.

A possible sequence of actions of Algorithm 1 on this example is given in Table 3.4. In the first column labeled "T" we number the hypothetical time points in which we observe the system. An explanation of the actions at each time point follows.

1. All processes create a task $\langle p_0,\ passive,\ \{\ldots\}\rangle$ for themselves and initialize their tables.

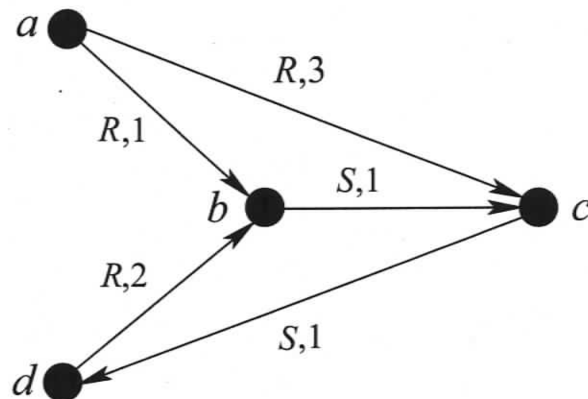

Figure 3.2: A database

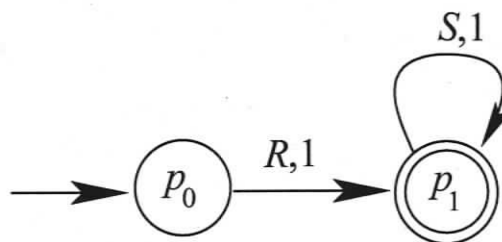

Figure 3.3: A query automaton

T	P_a	P_b	P_c	P_d
1	$\langle p_0, \mathit{passive},$ $\{[(a,p_0),(a,p_0),0,o],$ $[(b,p_1),(b,p_1),1,p],$ $[(c,p_1),(c,p_1),3,p]\}\rangle$	$\langle p_0, \mathit{passive},$ $\{[(b,p_0),(b,p_0),0,o]\}\rangle$	$\langle p_0, \mathit{passive},$ $\{[(c,p_0),(c,p_0),0,o]\}\rangle$	$\langle p_0, \mathit{passive},$ $\{[(d,p_0),(d,p_0),0,o],$ $[(b,p_1),(b,p_1),2,p]\}\rangle$
2	$\langle p_0, \mathit{active},$ $\{[(a,p_0),(a,p_0),0,o],$ $\mathbf{[(b,p_1),(b,p_1),1,p]},$ $[(c,p_1),(c,p_1),3,p]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(b,p_0),(b,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{passive},$ $\{[(b,p_1),(b,p_1),0,o],$ $[(c,p_1),(c,p_1),1,p]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(c,p_0),(c,p_0),0,o]\}\rangle$	$\langle p_0, \mathit{active},$ $\{[(d,p_0),(d,p_0),0,o],$ $\mathbf{[(b,p_1),(b,p_1),2,p]}\}\rangle$
3	$\langle p_0, \mathit{passive},$ $\{[(a,p_0),(a,p_0),0,o],$ $[(b,p_1),(b,p_1),1,o],$ $[(c,p_1),(b,p_1),2,p]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(b,p_0),(b,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{passive},$ $\{[(b,p_1),(b,p_1),0,o],$ $[(c,p_1),(c,p_1),1,p]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(c,p_0),(c,p_0),0,o]\}\rangle$	$\langle p_0, \mathit{passive},$ $\{[(d,p_0),(d,p_0),0,o],$ $[(b,p_1),(b,p_1),2,o],$ $[(c,p_1),(b,p_1),3,p]\}\rangle$
4	$\langle p_0, \mathit{active},$ $\{[(a,p_0),(a,p_0),0,o],$ $[(b,p_1),(b,p_1),1,o],$ $\mathbf{[(c,p_1),(b,p_1),2,p]}\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(b,p_0),(b,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{active},$ $\{[(b,p_1),(b,p_1),0,o],$ $\mathbf{[(c,p_1),(c,p_1),1,p]}\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(c,p_0),(c,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{passive},$ $\{[(c,p_1),(c,p_1),0,o],$ $[(d,p_1),(d,p_1),1,p]\}\rangle$	$\langle p_0, \mathit{active},$ $\{[(d,p_0),(d,p_0),0,o],$ $[(b,p_1),(b,p_1),2,o],$ $\mathbf{[(c,p_1),(b,p_1),3,p]}\}\rangle$
5	$\langle p_0, \mathit{passive},$ $\{[(a,p_0),(a,p_0),0,o],$ $[(b,p_1),(b,p_1),1,o],$ $[(c,p_1),(b,p_1),2,o],$ $[(d,p_1),(b,p_1),3,p]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(b,p_0),(b,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{passive},$ $\{[(b,p_1),(b,p_1),0,o],$ $[(c,p_1),(c,p_1),1,o],$ $[(d,p_1),(c,p_1),2,p]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(c,p_0),(c,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{passive},$ $\{[(c,p_1),(c,p_1),0,o],$ $[(d,p_1),(d,p_1),1,p]\}\rangle$	$\langle p_0, \mathit{passive},$ $\{[(d,p_0),(d,p_0),0,o],$ $[(b,p_1),(b,p_1),2,o],$ $[(c,p_1),(b,p_1),3,o],$ $[(d,p_1),(b,p_1),4,p]\}\rangle$
6	$\langle p_0, \mathit{active},$ $\{[(a,p_0),(a,p_0),0,o],$ $[(b,p_1),(b,p_1),1,o],$ $[(c,p_1),(b,p_1),2,o],$ $\mathbf{[(d,p_1),(b,p_1),3,p]}\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(b,p_0),(b,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{active},$ $\{[(b,p_1),(b,p_1),0,o],$ $[(c,p_1),(c,p_1),1,o],$ $\mathbf{[(d,p_1),(c,p_1),2,p]}\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(c,p_0),(c,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{active},$ $\{[(c,p_1),(c,p_1),0,o],$ $\mathbf{[(d,p_1),(d,p_1),1,p]}\}\rangle$	$\langle p_0, \mathit{active},$ $\{[(d,p_0),(d,p_0),0,o],$ $[(b,p_1),(b,p_1),2,o],$ $[(c,p_1),(b,p_1),3,o],$ $\mathbf{[(d,p_1),(b,p_1),4,p]}\}\rangle$ $\langle p_1, \mathit{completed},$ $\{[(d,p_1),(d,p_1),0,o]\}\rangle$
7	$\langle p_0, \mathit{passive},$ $\{[(a,p_0),(a,p_0),0,o],$ $[(b,p_1),(b,p_1),1,o],$ $[(c,p_1),(b,p_1),2,o],$ $[(d,p_1),(b,p_1),3,o]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(b,p_0),(b,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{passive},$ $\{[(b,p_1),(b,p_1),0,o],$ $[(c,p_1),(c,p_1),1,o],$ $[(d,p_1),(c,p_1),2,o]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(c,p_0),(c,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{passive},$ $\{[(c,p_1),(c,p_1),0,o],$ $[(d,p_1),(d,p_1),1,o]\}\rangle$	$\langle p_0, \mathit{passive},$ $\{[(d,p_0),(d,p_0),0,o],$ $[(b,p_1),(b,p_1),2,o],$ $[(c,p_1),(b,p_1),3,o],$ $[(d,p_1),(b,p_1),4,o]\}\rangle$ $\langle p_1, \mathit{completed},$ $\{[(d,p_1),(d,p_1),0,o]\}\rangle$
8	$\langle p_0, \mathit{completed},$ $\{[(a,p_0),(a,p_0),0,o],$ $[(b,p_1),(b,p_1),1,o],$ $[(c,p_1),(b,p_1),2,o],$ $[(d,p_1),(b,p_1),3,o]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(b,p_0),(b,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{completed},$ $\{[(b,p_1),(b,p_1),0,o],$ $[(c,p_1),(c,p_1),1,o],$ $[(d,p_1),(c,p_1),2,o]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(c,p_0),(c,p_0),0,o]\}\rangle$ $\langle p_1, \mathit{completed},$ $\{[(c,p_1),(c,p_1),0,o],$ $[(d,p_1),(d,p_1),1,o]\}\rangle$	$\langle p_0, \mathit{completed},$ $\{[(d,p_0),(d,p_0),0,o],$ $[(b,p_1),(b,p_1),2,o],$ $[(c,p_1),(b,p_1),3,o],$ $[(d,p_1),(b,p_1),4,o]\}\rangle$ $\langle p_1, \mathit{completed},$ $\{[(d,p_1),(d,p_1),0,o]\}\rangle$

Figure 3.4: A possible execution of Algorithm 1. Due to space constraints, we have abbreviated *prov* by p, and *opt* by o. We show in **bold** the tuples under expansion.

2. (a) P_a and P_d do have *provisional* tuples in the tables of their p_0–tasks, thus, they make their p_0–tasks *active* and expand their cheapest *provisional* tuples.

   For expansion, they send a request message to P_b for the creation of a p_1–task.

   On the other hand, processes P_b and P_c do not have *provisional* tuples in their p_0–tasks. Hence, they make their p_0–tasks *completed.* That is, there are no $(b, _, _)$ and $(c, _, _)$ query answers to be expected.

   (b) P_b receives the request messages from P_a and P_d and creates the p_1–task. Also, P_b initializes this task as described in the algorithm. Of course, P_b creates only one such task to serve both P_a and P_d, and thus, we see here an effective computation overlap.

   Then, P_b establishes the appropriate communication channels between its p_1–task and the p_0–tasks in P_a and P_d.

   P_b is not only asked to create the p_1–task, but also to provide information about the (b, p_1)–keyed tuple. Since the status of this tuple in the p_1–task of P_b is *optimal,* P_b sends its $p_1.T$ knowledge to $P_a.p_0$ and $P_d.p_0$ adding along the way the *weights* of the related channels.

3. Upon receiving the reply message from P_b, processes P_a and P_d update the

tables of their p_0–tasks. Note that the statuses of the (b, p_1)–keyed tuples in $P_a.p_0.T$ and $P_d.p_0.T$ become *optimal*.

P_a relaxes the (c, p_1)–keyed tuple in $p_0.T$ and changes its via to (b, p_1). P_d adds to $P_d.p_0.T$ the rest of the $P_b.p_1.T^*$ tuples setting their via component to (b, p_1). Then, P_a and P_d change the status of their p_0–tasks to *passive* becoming thus ready for the next expansion.

4. (a) P_a and P_d make the status of their p_0–tasks *active*, and expand the tuples $[(c, p_1), (b, p_1), 2, prov]$ and $[(c, p_1), (b, p_1), 3, prov]$, respectively by sending request messages to process P_b.

   (b) P_b has already a p_1–task, and thus, it just establishes communication channels with P_a and P_d specialized for (c, p_1).

   As the status of the (c, p_1)–keyed tuple in $P_b.p_1.T$ is *provisional*, P_b cannot yet reply back to P_a or P_d. Instead, P_b makes the status of p_1–task *active* and starts its processing. That is, P_b selects the cheapest *provisional* tuple, i.e., the tuple $[(c, p_1), (c, p_1), 1, prov]$, and sends a request message to P_c to create task p_1.

   (c) Upon receiving the request message from P_b, process P_c creates and initializes a p_1–task. Also, P_c establishes a communication channel with P_b, which is specialized for (c, p_1). Since the status of the (c, p_1)–keyed tuple

is *optimal*, P_c replies back to P_b with the message $\langle P_c.p_1.T^* \rangle$.

[The rest of the steps are briefly described below.]

5. (a) Upon receiving the reply message from P_c, P_b updates its $p_1.T$ table as appropriate.

   (b) Now, P_b has an *optimal* status for the (c, p_1)–keyed tuple in $p_1.T$, and thus, replies back to P_a and P_d with the message $\langle P_b.p_1.T^* \rangle$.

   (c) Upon receiving the reply message from P_b, P_a and P_d update their $p_0.T$ tables as appropriate.

6. (a) P_a and P_d expand the tuples $[(d, p_1), (b, p_1), 3, prov]$ and $[(d, p_1), (b, p_1), 4, prov]$, respectively.

   (b) In effect, P_b expands $[(d, p_1), (c, p_1), 2, prov]$, and then P_c expands $[(d, p_1), (d, p_1), 1, prov]$. P_c requests from P_d to create a p_1–task and provide information about (d, p_1).

   (c) P_d creates and initializes the p_1–task and replies back to P_c with the message $\langle P_d.p_1.T^* \rangle$.

7. (a) Upon receiving the reply message from P_d, P_c updates its $p_1.T$ table as appropriate. Then, P_c replies back to P_b with the message $\langle P_c.p_1.T^* \rangle$.

(b) Upon receiving the reply message from P_c, P_b updates its $p_1.T$ table as appropriate. Then, P_b replies back to P_a and P_d.

(c) Upon receiving the reply message from P_b, P_a and P_d update their $p_0.T$ tables as appropriate.

8. Finally, as there are no more provisional tuples in any of the tasks, they attain a *completed* status.

3.4 Reflections

Observe that we can terminate as soon as the p_0–tasks become *completed* in each of the processes. There is no need to continue until the completion of the rest of the tasks. Their completion would not bring any new query answers, thus we can safely abort the processes.

Note that, we can incrementally report the query answers as soon as their corresponding tuples become *optimal* in the table of a p_0–task in some process. For example, $(a, b, 1)$ and $(d, b, 2)$ can be reported at time point 3, $(a, c, 2)$ and $(d, c, 3)$ can be reported at time point 5, and so on.

At time point 4, when P_a and P_d expand the (c, p_1)–keyed tuples requesting P_b to provide information about such a tuple in $P_b.p_1.T$, it happens that this tuple is the cheapest *provisional* tuple in $P_b.p_1.T$. Another instance of such a situation is at time point 6, in which again the requested information is about a tuple that is the

cheapest *provisional* tuple in $P_b.p_1.T$. These are not coincidental and by the following theorem, we show that this is indeed a property of the algorithm which guarantees the soundness (see Chapter 6). Of course, in some situations the request might be for an *optimal* tuple, and in such a case there is no need for further expansion in order to reply back. Note, that the following theorem is about the case when the request is for a *provisional* tuple in the receiver process.

Theorem 2 *If a process, through a task request message, is asked to provide information about a* provisional *tuple, then this tuple is the cheapest one among such tuples in the requested task.*

Proof. Suppose process P_a asks process P_b to expand a tuple in its p_y–task. Let the expanded tuple in P_a be $[(c, p_z), (b, p_y), w_{ac}, prov]$. This expansion will ask from P_b to provide information about the (c, p_z)–keyed tuple in its p_y–task. Let this tuple be $[(c, p_z), (_, _), w_{bc}, prov]$. We want to show that this tuple is the cheapest among the *provisional* tuples in $P_b.p_y.T$.

Since (b, p_y) is the via component of the (c, p_z)–keyed tuple in $P_a.p_x.T$, we conclude that this tuple has got its weight, during an update phase, from the tuple $[(c, p_z), (_, _), w_{bc}, prov]$ in $P_b.p_y.T$ after adding the weight of the corresponding communication channel.

Along with the $[(c, p_z), (_, _), w_{bc}, prov]$ tuple, P_a got from P_b all the other tuples in $P_b.p_y.T$, on whose weights the same channel weight w_{ab} was added. Now, since

$[(c, p_z), (b, p_y), w_{ac}, prov]$ is the cheapest *provisional* tuple in $P_a.p_x.T$, and its weight w_{ac} is in fact equal to $w_{ab} + w_{bc}$, we have that $[(c, p_z), (_, _), w_{bc}, prov]$ is the cheapest tuple in $P_b.p_y.T$. ■

Chapter 4

Termination

4.1 Impossibility of Deadlock

In the following theorem we show that the algorithm terminates and it does not enter an infinite loop. That is, eventually there will be no more *provisional* tuples in the tables of the p_0 tasks, which is the condition for termination of the algorithm at each process.

Theorem 3 *Algorithm 1 (positively) terminates.*

Proof. Suppose there is a deadlock. Without loss of generality and for better clarity, assume there are only three processes involved in a deadlock[1].

Such deadlock can assumedly be created in the following scenario.

1. Process P_a expands tuple $[(d, p_u), (b, p_y), w_{ad}, prov]$ in its p_x–task. Thus, it sends a corresponding message to P_b requesting a p_y–task and asking information about the (d, p_u)–keyed tuple in this p_y–task.

2. Process P_b already has a p_y–task, but cannot reply back at the moment since there is some tuple $[(e, p_v),\ (c, p_z),\ w_{be}, prov]$ in the p_y–task, whose w_{be} weight

[1]Similar reasoning can be done for the cases of two or more than three processes.

is smaller than the weight of the (d, p_u)–keyed tuple. Thus, P_b sends a message to P_c requesting a p_z–task and asking information about the (e, p_v)–keyed tuple in this p_z–task.

3. Process P_c already has a p_z–task, but cannot reply back at the moment since there is some tuple $[(f, p_w), (a, p_x), w_{cf}, prov]$ in the p_z–task, whose w_{cf} weight is smaller than the weight of the (e, p_v)–keyed tuple. Thus, P_c sends a message to P_a requesting a p_x–task and asking information about the (f, p_w)–keyed tuple in this p_x–task.

4. Process P_a has an (f, p_w)–keyed tuple in the table of its p_x–task, and this tuple has a *provisional* status. Note that an (f, p_w)–keyed tuple certainly exists in the p_x–task of P_a. This is because otherwise, the via node-state pair of the (f, p_w)–keyed tuple in $P_c.p_z$ would not be (a, p_x).

   On the other hand, process P_a has the p_x–task in *active* status waiting for a reply to the expansion of the (d, p_u)–keyed tuple. This prevents P_a to expand any other tuple including the (f, p_w)–keyed tuple. Hence, it cannot reply back to P_c and the deadlock assumedly occurs.

Now, we show that such a situation cannot happen during the execution of our algorithm.

Since P_a expands the tuple $[(d, p_u), (b, p_y), w_{ad}, prov]$ in the table of the p_x–task, we have that w_{ad} is the smallest weight among the *provisional* tuples of the p_x–task. In particular, $w_{af} \geq w_{ad}$, where w_{af} is the weight of the (f, p_w)–keyed tuple in $P_a.p_x.T$.

Process P_a has to get information about the (d, p_u)–keyed tuple through its neighbor process P_b, which is the via process for that tuple.

By the **Update** thread, we have that

$$w_{ad} = w[(a, p_x), (b, p_y)] + w_{bd},$$

where $w[(a, p_x), (b, p_y)]$ is the cheapest weight product of a matching automaton transition from p_x to p_y with a database edge from a to b, and w_{bd} is the weight of the (d, p_u)–keyed tuple in $P_b.p_y.T$. Hence,

$$w_{af} \geq w_{ad} = w[(a, p_x), (b, p_y)] + w_{bd}.$$

As P_b selects the tuple keyed by (e, p_v) to expand, we have $w_{bd} \geq w_{be}$. Therefore, it can be concluded that

$$\begin{aligned} w_{af} &\geq w[(a, p_x), (b, p_y)] + w_{be} \\ &= w[(a, p_x), (b, p_y)] + w[(b, p_y), (c, p_z)] + w_{ce}. \end{aligned}$$

According to the deadlock scenario outlined in the beginning of this proof, P_c tries to expand tuple $[(f, p_w), (a, p_x), w_{cf}, prov]$ of the p_z–task when it is asked for the information of the (e, p_v)–keyed tuple. So, $w_{ce} \geq w_{cf}$, and hence,

$$w_{af} \geq w[(a, p_x), (b, p_y)] + w[(b, p_y), (c, p_z)] + w_{cf}$$

$$= w[(a,p_x),(b,p_y)] + w[(b,p_y),(c,p_z)]$$

$$+ w[(c,p_z),(a,p_x)] + w_{af}.$$

However, recall from Chapter 2 that the edge weights are positive numbers, and thus the above cannot happen, reaching so a contradiction. ■

4.2 Termination Detection

As mentioned earlier, the algorithm should terminate when each process has a *completed* p_0–task. However, there is the question of how to detect the global termination of our algorithm. For this, we use the same minimum spanning tree (MST) of processes, which was used for initially sending the query (see Chapter 3). The termination detection procedure using this MST is as follows. A leaf process reports "p_0–task completed" to its parent when such a condition is true [for the leaf]. A non-leaf process reports "p_0–task completed" to its parent only when this condition is true for itself and all its children. Clearly, when the root of the MST has a completed p_0–task, and receives similar completion messages from all its children, the algorithm declares global termination.

Chapter 5

Complexity

5.1 Message Complexity

In this thesis, 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 Chapter 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 4 *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.

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 $n \cdot s \cdot (n \cdot s) \cdot (1 + 1) = 2 \cdot n^2 \cdot s^2$. ■

5.2 Discussion

We remark that the above upper bound coincides with the message lower bound of Ramarao and Venkatesan in [20] for the distributed computation of *single-source* shortest paths. Hence, we conclude that our algorithm is optimal with respect to the number of messages. However, the messages in [20] have a constant size, while our messages have $O(n \cdot s)$ size. On the other hand, our problem is more difficult than the classical *single-source* shortest paths problem of [20].

Chapter 6

Soundness and Completeness

In this Chapter we show the soundness and completeness of our algorithm. For the former, we show that each reported query answer is optimally weighted. For the later, we show that all the query answers are indeed reported.

6.1 Some Observations and a Fact

Let $[(c, p_z), (_, _), _, prov]$ be a provisional tuple in $P_a.p_x.T$. Eventually, this tuple will get expanded. Until that moment its via and weight components might possibly change. Suppose that at the moment of expansion, the tuple is $[(c, p_z), (b, p_y), w_{ac}, prov]$. The existence of this tuple tells P_a that a path π_{ac} of object-state pairs, starting from (a, p_x), passing via (b, p_y), and ending in (c, p_z), has been traversed and its cost is w_{ac}.

After the expansion, the tuple $[(c, p_z), (b, p_y), w_{ac}, prov]$ changes status to optimal, and also $P_a.p_x.T$ gets [through the back-replies] new provisional tuples corresponding to all the possible one-step expansions of path π_{ac}.

This is because process P_c,

1. receives or has earlier received [through the chain of forward messages] a request

for (c, p_z),

2. possibly creates a p_z–task [if it does not exist], and

3. replies back with an optimal (c, p_z)–keyed tuple, along with provisional tuples corresponding to its neighbors.

What is important is that, the provisional tuples of P_c represent all the possibilities for further single step expansion of path π_{ac}, and these tuples will find their way [through back-replies] to table $P_a.p_x.T$. Of course they are appropriately weighted along the way by the cost of the communication channels.

Thus, $P_a.p_x.T$, through its provisional tuples, possesses full information about the expansion frontier of the object-state paths starting at (a, p_x). Namely, each provisional tuple corresponds to one such path. Furthermore, by Theorem 2, expanding the cheapest provisional tuple triggers the expansion of the cheapest path. In other words, we can state the following fact:

Fact 1: *Paths are expanded in order of their costs.*

If the costs are positive non-zero numbers, then the above fact guarantees both the soundness and completeness of our algorithm.

6.2 Soundness

In our algorithm, in essence we do rounds of path expansions. From Fact 1, the first path to reach (c, p_z) is the cheapeast of such paths, starting at (a, p_x) and reaching to (c, p_z). Hence, if p_z is a final state, the weight associated with object c in the query answer is the cheapest possible, i.e., optimal.

6.3 Completeness

Suppose that there exists a query answer (a, c, w_{ac}), but the algorithm misses it. In other words, there exists a path π_{ac} from (a, p_x) to (c, p_z) with cost w_{ac} and p_z being a final state, but the algorithm does not expand it.

Let this path π_{ac} be the cheapest such path, and the object-state pairs in it be: $(a, p_x) = (b_1, p_1)$, (b_2, p_2), ..., $(b_n, p_n) = (c, p_z)$.

Let (b_k, p_k), where $k \leq n$, be the last object-state pair in π_{ac}, which the algorithm has reached. In other words, all of (b_1, p_1), (b_2, p_2), ..., (b_k, p_k) have been reached, but (b_{k+1}, p_{k+1}) has not been reached.

By the way the algorithm works, we know that when (b_k, p_k) is reached, $P_a.p_x.T$ does receive information [through back-replies] about (b_{k+1}, p_{k+1}) in the form of a provisional (b_{k+1}, p_{k+1})–keyed tuple.

Surely, the (b_{k+1}, p_{k+1})–keyed tuple will be eventually expanded, and some path will be extended by (b_{k+1}, p_{k+1}). But, which path? The via of the (b_{k+1}, p_{k+1})–keyed

tuple might have changed along the way, and it may not be (b_2, p_2) when the turn of the (b_{k+1}, p_{k+1})–keyed tuple comes to be expanded.

Since we assumed that π_{ac} is the cheapest path starting from $(a, p_x) = (b_1, p_1)$ and ending in $(b_n, p_n) = (c, p_z)$, we have that (b_1, p_1), (b_2, p_2), ..., (b_k, p_k), (b_{k+1}, p_{k+1}) is also the cheapest path starting from $(a, p_x) = (b_1, p_1)$ and ending in (b_{k+1}, p_{k+1}).

Now, by Fact 1, we have that in our algorithm, the first path to reach (b_{k+1}, p_{k+1}) is the cheapest one. Since the cheapest path is (b_1, p_1), (b_2, p_2), ..., (b_k, p_k), (b_{k+1}, p_{k+1}), we have that the via of the (b_{k+1}, p_{k+1})–keyed tuple in $P_a.p_x.T$, at the moment of expansion will be (b_2, p_2).

Similar reasoning applies also to the intermediate processes, and finally, the pair (b_{k+1}, p_{k+1}) will indeed be the new addition to the path (b_1, p_1), (b_2, p_2), ..., (b_k, p_k).

Chapter 7

Conclusions and Future Work

We have presented a fully distributed algorithm for evaluating generalized regular path queries on database graphs. We have explained that this is more difficult than simply computing shortest paths on the database. This is due to the fact that the object-state graph is only known after the query is evaluated, but not beforehand.

Our proposed algorithm is such that it works by computing simultaneously both query answers and cheapest paths reaching them without need for global knowledge of database graph. We showed that the algorithm is message–optimal, regarding number of messages, and it evaluates queries progressively. That is, the user can see partial answers quickly, as soon as they are computed, while she is waiting for new answers to arrive. Some other remarkable properties of our proposed algorithm are symmetric execution and easy termination detection. All the processes servicing different database objects run the same algorithm and each process is capable to detect its local termination. Also, the global termination can be detected simply by using an spanning tree and there is no need to have a separate termination detection algorithm.

For future work, we are investigating the possibility of making our algorithm

fault-tolerant, which is certainly a desirable property to have, especially in grid environments. A fault-tolerant algorithm needs a fault-tolerant termination detector which forms another future challenge. In fact, we have some encouraging preliminary results in this direction, which remain to be further researched.

Bibliography

[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] Abiteboul S., V. Vianu. Regular Path Queries with Constraints. *J. of Computing and System Sciences* 58 (3), pp. 428–452, 1999.

[3] Allauzen C., M. Mohri. A Unified Construction of the Glushkov, Follow, and Antimirov Automata. *Proc. of MFCS'06.*

[4] Awerbuch B. Optimal distributed algorithms for minimum-weight spanning tree, counting, leader election and related problems. *Proc. of STOC'87.*

[5] Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. Answering Regular Path Queries Using Views. *Proc. of ICDE'00.*

[6] Calvanese D., G. Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on Regular Path Queries. *SIGMOD Record* 32 (4), pp. 83–92, 2003.

[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.*

[8] Consens M. P, A. O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. *Proc of PODS'90.*

[9] Finkelstein S. J., N. Mattos, I. S. Mumick, and H. Pirahesh. Expressing recursive queries in SQL. *ISO WG3 report X3H2–96–075, March, 1996.*

[10] Flesca S., F. Furfaro, and S. Greco. Weighted Path Queries on Web Data. *Proc. of WebDB'01.*

[11] Grahne G., and A. Thomo. Regular Path Queries Under Approximate Semantics. *Ann. Math. Artif. Intell.* 46 (1–2), pp. 165–190, 2006.

[12] Grahne G., A. Thomo, and W. Wadge. Preferentially Annotated Regular Path Queries. *Proc. of ICDT'07.*

[13] Haldar S. An "All Pairs Shortest Paths" Distributed Algorithm Using $2n^2$ Messages. *J. of Algorithms,* 24 (1), pp. 20–36, 1997.

[14] Hopcroft J. E., and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[15] Lai H. T., and F. L. Wu. An (N-1)-Resilient Algorithm for Distributed Termination Detection. *IEEE Trans. Parallel Distrib. Syst.,* 6 (1), pp. 63–78, 1995.

[16] Matocha J., and T. Camp. A Taxonomy of Distributed Termination Detection Algorithms. *J. of Systems and Software,* 43 (3), pp. 207–221, 1998.

[17] Mendelzon A. O., and P. T. Wood, Finding Regular Simple Paths in Graph Databases. *SIAM J. Comp.* 24 (6), pp. 1235–1258, 1995.

[18] Miao Z., D. Stefanescu, A. Thomo. Grid-Aware Evaluation of Regular Path Queries on Spatial Networks. *Proc. of AINA'07.*

[19] Planet–Lab: http://www.planet-lab.org.

[20] Ramarao S. V. K., S. Venkatesan. The Lower Bounds on Distributed Shortest Paths. *Inf. Process. Lett.*, 48 (3), pp. 145–149, 1993.

[21] Shoaran M., A. Thomo. Distributed multi-source regular path queries. *submitted.*

[22] Stefanescu D., A. Thomo, and L. Thomo. Distributed Evaluation of Generalized Path Queries *Proc. of SAC'05.*

[23] Stefanescu D., A. Thomo. Enhanced Regular Path Queries on Semistructured Databases. *Proc. of QLQP'06.*

[24] Suciu D., Distributed Query Evaluation on Semistructured Data. *ACM Trans. on Database Systems,* 27 (1), pp. 1–62, 2002.

[25] TIGER: Topologically Integrated Geographic Encoding and Referencing system, US Census Bureau, http://www.census.gov/geo/www/tiger.

[26] Vardi M. Y. A Call to Regularity. *Proc. PCK50 - Principles of Computing & Knowledge, Paris C. Kanellakis Memorial Workshop '03,* pp. 11.