Finding Top-k Approximate Answers to Path Queries

Carlos A. Hurtado[1], Alexandra Poulovassilis[2], and Peter T. Wood[2]

[1] Faculty of Engineering and Sciences
Universidad Adolfo Ibáñez, Chile
carlos.hurtado@uai.cl

[2] School of Computer Science and Information Systems
Birkbeck, University of London, UK
{ap,ptw}@dcs.bbk.ac.uk

Abstract. We consider the problem of finding and ranking paths in semistructured data without necessarily knowing its full structure. The query language we adopt comprises conjunctions of regular path queries, allowing path variables to appear in the bodies and the heads of rules, so that paths can be returned to the user. We propose an approximate query matching semantics which adapts standard notions of approximation from string matching to graph matching. Query results are returned to the user ranked in order of increasing "distance" to the user's original query. We show that the top-k approximate answers can be returned in polynomial time in the size of the database graph and the query.

1 Introduction

The volume and heterogeneity of semistructured data being made increasingly available, e.g. in the form of RDF linked data [4], necessitates support for users in formulating queries over this data. In particular, users need to be assisted by query processing systems that do not require users' queries to match precisely the structure of the data [9, 12, 15, 18]. Moreover, in many application areas, users need to be able to find *paths* through large volumes of semistructured data, e.g. in bioinformatics [19, 20] and community and social network analysis [2, 21].

In this paper, we consider the problem of a user posing path queries on semistructured data without necessarily knowing the full structure of the data. We consider a general data model comprising a directed graph $G = (V, E)$. Each node in V is labelled with a constant and each edge is labelled with a pair (l, c), where l is a label drawn from a finite alphabet Σ and $c \in \mathbb{N}$ is a cost associated with traversing the edge[3].

There has been much work on using regular expressions to specify paths through data (e.g. [1, 6, 7, 13]). In recent work [16], we considered approximate

[3] This general graph model encompasses RDF data, for example (with all edge costs being 1 in this case), except that it does not allow for the representation of RDF's "blank" nodes. However, blank nodes are discouraged for RDF linked data [14].

matching of *conjunctive regular path queries* [6], of the form

$$(Z_1, \ldots, Z_m) \leftarrow (X_1, R_1, Y_1), \ldots, (X_n, R_n, Y_n)$$

where each X_i and Y_i, $1 \leq i \leq n$, is a variable or constant, each Z_i, $1 \leq i \leq m$, is a variable appearing in the body of the query, and each R_i, $1 \leq i \leq n$, is a regular expression over Σ. However, in that work we did not allow path variables within queries, and were not able to return paths as results to users' queries.

Here, we consider *extended regular path* (ERP) queries of the form:

$$X_{i_1}, P_{i_1}, Y_{i_1}, \ldots, X_{i_r}, P_{i_r}, Y_{i_r} \leftarrow (X_1, (R_1 : P_1), Y_1), \ldots, (X_n, (R_n : P_n), Y_n)$$

where each X_i and Y_i, $1 \leq i \leq n$, is a variable or constant; each R_i, $1 \leq i \leq n$, is a regular expression over Σ; and each P_i, $1 \leq i \leq n$, is a path variable. Variables take values which are node labels while path variables take values which are paths.

The answer to an ERP query Q on a graph G is specified as follows: we find, for each conjunct $(X_i, (R_i : P_i), Y_i)$, $1 \leq i \leq n$, a relation r_i over the scheme (X_i, P_i, Y_i, C_i) such that tuple $t \in r_i$ iff there is a path $t[P_i]$ of cost $t[C_i]$ from $t[X_i]$ to $t[Y_i]$ in G that satisfies R_i i.e. whose concatenation of edge labels is in $L(R_i)$; we then form the natural join of relations $r_1, \ldots, r_n$ and project over the variables and path variables appearing in the head of the query. In returning instantiations of the path variables P_i within query answers, we do not include the start and end nodes of the path since these are instantiated by the variables X_i, Y_i (see Examples 1 and 2 below).

We generally want to return the k lowest-cost paths satisfying Q. For each conjunct $(X_i, (R_i : P_i), Y_i)$, $1 \leq i \leq n$, the cost, $cost(p)$, of a path p which satisfies R_i is simply the sum of the costs of each of the edges of p. For the query as a whole, we assume that a monotonically increasing function $qcost(P_1, \ldots, P_n)$ is specified. For simplicity, in our examples below we assume that a cost of 1 is associated with edges, and that $qcost(P_1, \ldots, P_n) = cost(P_1) + \cdots + cost(P_n)$.

Example 1. This example is movitated by the L4All system [8, 23], which allows lifelong learners to create and maintain a chronological record of their learning, work and personal episodes — their *timelines* — with the aim of fostering collaborative formulation of future goals and aspirations; see Figure 1 for an example timeline. Episodes have a start and end date associated with them (for simplicity, these are not shown in Figure 1). Episodes are ordered within the timeline according to their start date — as indicated by edges labelled `next`. There are many different categories of episode, for example `University` and `Work` in our example. Associated with each category of episode are several properties — for simplicity, we have shown just two of these, `subj[ect]` and `pos[ition]`.

A key aim of the L4All system is to allow learners to share their timelines with others and to identify possibilities for their own future learning and professional development from what others have done. For example, suppose that Mary has studied English at university, and she wants to find out what possible future career choices there are for her by seeing what others who studied English have gone on to do. The following ERP query Q_1 can be formulated (in

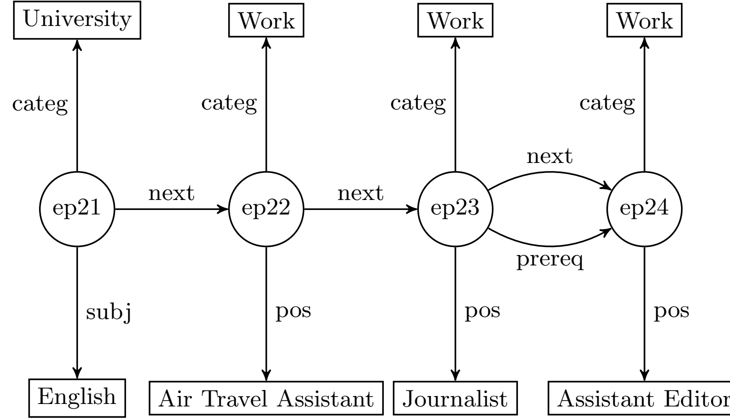

Fig. 1. A fragment of timeline data.

the concrete query syntax, variable names are preceded with '?'; in this query, the ?Ei instantiate to episodes and the ?Pi to paths):

```
?E1,?P3,?E2,?E2,?P4,?Cat,?E2,pos,?Pos <-
        (?E1,categ:?P1,University),(?E1,subj:?P2,English),
        (?E1,next+:?P3,?E2),(?E2,categ:?P4,?Cat),(?E2,pos:?P5,?Pos)
```

When Q_1 is evaluated on a database of timeline data that includes the subgraph shown in Figure 1, the results returned include

```
ep21,[next],ep22,ep22,[categ],Work,ep22,[pos],AirTravelAsssistant;
ep21,[next,ep22,next],ep23,ep23,[categ],Work,ep23,[pos],Journalist;
ep21,[next,ep22,next,ep23,next],ep24,ep24,[categ],Work,ep24,[pos],
     AssistantEditor;
```

with costs of 5, 6 and 7, respectively. □

As stated earlier, in general we want to allow for approximate matching of such queries. Grahne and Thomo [12] explored approximate matching of single-conjunct regular path queries using a *weighted regular transducer*. This is a finite state automaton in which transitions are labelled with triples. A transition from state s to state t labelled (a, i, b) means that if the transducer is in state s, it can move to state t on input a with cost i while outputting b. In our context, such a transition specifies that symbol a in a query can match the label b of a graph edge with cost i. Grahne and Thomo showed that such queries can be evaluated incrementally in polynomial time, with results being returned to the user in ranked order of similarity to the original query. In [16], we showed that multiple-conjunct regular path queries can also be evaluated incrementally in polynomial time, using ideas from [17] on evaluation of top-k query results. In this paper, we extend this approach by considering extended regular path queries, which allow path variables within queries and are able to return paths within the result set. We focus on approximate regular expression matching [24], which

can easily be specified using weighted regular transducers [12]. As in [16], the edit operations we allow in approximate matching of ERP queries are: insertions, deletions and substitutions of symbols; inversions of symbols (corresponding to edge reversals); and transpositions of adjacent symbols, each with an assumed edit cost of 1. In practice, we envisage the user being able to select the set of edit operations they wish to be applied to their query and the relative cost associated with each one, reflecting their own preferences and the application semantics.

In the next section, we present a motivating example for approximate matching of ERP queries. In Section 3, we consider first the case of computing approximate answers for ERP queries containing only one conjunct. We show that the top-k approximate answers can be returned in polynomial time in the size of the database graph and the query, and we also discuss how the top-k answers can be computed incrementally. In Section 4 we extend this complexity result to the case of general ERP queries, provided the query conjuncts are acyclic. We note that these results are a non-trivial extension of our earlier work in [16], since paths are now able to be returned to the user. In general, there may be an infinite number of paths between two given nodes of the database graph (due to the possible presence of cycles in the graph and also the fact that we allow edges to be traversed in the reverse direction in queries). This makes finding the top-k paths considerably more complicated than finding the top-k start and end nodes of *some* path, as in [16]. We discuss related work in Section 5 and give our conclusions and directions for future work in Section 6.

2 Motivation

We begin with an example illustrating approximate matching of ERP queries.

Example 2. Consider again Example 1. We notice that the timeline in Figure 1 has an edge labelled `prereq` from `ep23` to `ep24`. Such annotations can be created by a timeline's owner, indicating that this person believes that undertaking an earlier episode was a prerequisite to them being able to proceed to or achieve a later episode. Knowing of the existence of such metadata, Mary might instead pose the following query Q_2 (where `next+` in Q_1 has been replaced by `prereq+`) in order to find out what she might do after her English degree:

```
?E1,?P3,?E2,?E2,?P4,?Cat,?E2,pos,?Pos <-
    (?E1,categ:?P1,University),(?E1,subj:?P2,English),
    (?E1,prereq+:?P3,?E2),(?E2,categ:?P4,?Cat),(?E2,pos:?P5,?Pos)
```

However, this will return no results relating to the example timeline of Figure 1 even though it is evident that this timeline does in fact contain information that would be relevant to Mary's enquiry.

In practice, users may or may not create `prereq` metadata relating to their timelines. For example, in relation to the timeline of Figure 1, it is possible that undertaking an English degree was in fact a prerequisite for this person becoming a Journalist, i.e. that there should also be an edge labelled `prereq` from

ep21 to ep23. So it is desirable to provide users with flexible ways of querying such timeline data. For example, by allowing replacement of the symbol prereq by the symbol next, the regular expression prereq+ in query Q_2 can be approximated by next.prereq* and prereq.next.prereq*, both at edit distance 1 from prereq+. This allows ep21,[next],ep22, ep22,[categ],Work, ep22,[pos],AirTravelAsssistant to be returned to Mary.

Mary may judge this result to be not relevant to her English degree and may seek further results from the system. The regular expressions next.prereq* and prereq.next.prereq* can both be approximated by next.next.prereq* (as well as by other expressions), now at edit distance 2 to Q_2. This allows ep21,[next,ep22,next],ep23,ep23,[categ],Work,ep23,[pos],Journalist and ep21,[next,ep22,next,ep23,prereq],ep24,ep24,[categ],Work,ep24 [pos],AssistantEditor to be returned. Mary may judge both of these as being relevant, and she can then request the system to return the whole of this user's timeline for her to explore further. □

In applications such as this we expect that a visual query interface would provide the user with a set of options from which to select their query approximation requirements, and the costs associated with the selected edit operations. Our main concern in this paper is in investigating the evaluation and approximate matching of ERP queries over graph-structured data.

3 Simple ERP Queries

We recall that our data model is that of a directed graph $G = (V, E)$. Each node in V is labelled with a constant and each edge in E is labelled with a pair (l, c), where l is a drawn from a finite alphabet Σ and $c \in \mathbb{N}$ is a cost.

In processing queries, we allow edges of G to be traversed both from their source to their target node and also in reverse, from their target to their source node. The *inverse* of an edge label l, denoted by l^-, is used to specify a reverse traversal of an edge. Let $\Sigma^- = \{a^- \mid a \in \Sigma\}$. If $l \in \Sigma \cup \Sigma^-$, then l^- denotes the *inverse* of l. In particular, for some $a \in \Sigma$, if l is a then l^- is a^-, while if l is a^- then l^- is a.

A *simple ERP query* is an expression of the form:

$$X, P, Y \leftarrow (X, (R\!:\!P), Y)$$

where X and Y are constants or variables, R is a regular expression over Σ, and P is a path variable. A *regular expression* R over Σ is defined as follows:

$$R := \epsilon \mid a \mid a^- \mid _ \mid (R1 \cdot R2) \mid (R1|R2) \mid R^* \mid R^+$$

where ϵ is the empty string, a is any symbol in Σ, a^- is the inverse of a, "_" denotes the disjunction of all constants in Σ, and the operators have their usual meaning.

Due to the presence of inverse operators in queries, we use the notion of a *semipath* in G in order to define the semantics of queries [6]. A *semipath* p in $G =$

(V, E) from $x \in V$ to $y \in V$ is a sequence of the form $v_1, l_1, v_2, l_2, \ldots, v_n, l_n, v_{n+1}$, where $n \geq 0$, $v_1 = x$, $v_{n+1} = y$ and for each v_i, l_i, v_{i+1} either $v_i \overset{(l_i,c_i)}{\rightarrow} v_{i+1} \in E$ or $v_{i+1} \overset{(l_i^-,c_i)}{\rightarrow} v_i \in E$, for some cost c_i. A semipath p *conforms* to a regular expression R if $l_1 \cdots l_n \in L(R)$, the language denoted by R.

Given a simple ERP query Q and a graph G, a *matching* θ is a function that maps any node variable in Q to a node in G, any node constant to itself, and the triple (X, P, Y) to a semipath from $\theta(X)$ to $\theta(Y)$ in G. We write that $\theta(X, P, Y)$ is an *approximate answer* of Q on G. An *exact answer* is an approximate answer $\theta(X, P, Y)$ that conforms to R.

For approximate query matching, sequences of edge labels can be transformed using the edit operations selected by the user. The edit distance from a semipath p_1 to a semipath p_2 is the minimum cost of any sequence of edit operations which transforms the sequence of edge labels of p_1 to the sequence of edge labels of p_2 (for simplicity, in this paper we assume that all edit operations have a cost of 1). The edit distance of a semipath p to a regular expression R, denoted $\mathit{edist}(p, R)$, is the minimum edit distance from p to any semipath in G that conforms to R.

Given an approximate answer $\theta(X, P, Y)$ of Q on G defined as above, its *distance* to Q is

$$f(\theta(X, P, Y) = \alpha \cdot \mathit{editd}(\theta(X, P, Y), R) + \beta \cdot \mathit{cost}(\theta(X, P, Y))$$

where $\mathit{cost}(\theta(X, P, Y))$ is the sum of the edge costs of the semipath $\theta(X, P, Y)$ and the coefficients α and β are weightings of the edit distance and the path cost contributions to the overall distance, set according to the requirements of the user or the application. The approximate answers are returned in increasing order of their distance to Q. For example, if the user sets $\alpha = 5$ and $\beta = 1$, then exact answers of cost ≤ 5 are returned before any answers at edit distance > 0.

Given a graph $G = (V, E)$ and a simple ERP query Q, the *approximate top-k answer* of Q on G is a list containing the k approximate answers of Q on G with minimum distance to Q, ranked in order of increasing distance to Q.

The evaluation of a simple ERP query begins by constructing the *approximate automaton* M of the regular expression R at some edit distance h, where M accepts all strings at edit distance at most h from R. The construction of M proceeds as in the case of single-conjunct regular path queries, as we described in [16], and we recall this here:

We first construct an NFA (nondeterministic finite automaton) M_R to recognise $L(R)$, using Thompson's construction. This ensures that M_R has a single initial state, denoted s_0, a single final state, denoted s_f, and $O(|R|)$ states. The approximate automaton M at edit distance h is obtained following a standard construction used in approximate string matching [24]: h copies of M_R are made, each copy denoted M_R^j, whose states are those of M_R with superscript j, representing distance j from the original automaton M_R. The only initial state in M is s_0^0, and the final state of each M_R^j remains a final state in M. Subautomaton M_R^j is connected to subautomaton M_R^{j+1} by $O(|R|)$ transitions for deletions, $O(|E| \cdot |R|)$ transitions for insertions (assuming only symbols which are labels of

edges in G will be inserted), $O(|E| \cdot |R|)$ transitions for substitutions similarly, $O(|R|)$ transitions for inversions, and $O(|R|)$ transitions for transpositions.

Given a semipath p in G with no edge being traversed more than once, and the approximate automaton M for regular expression R at edit distance $|R|+|E|$, we note that p conforms to M. This is because p can be accepted by M through $|R|$ transitions that delete all the symbols that appear in R followed by at most $|E|$ transitions that add all the edge labels of p.

However, because of the possibility of cycles in the semipaths resulting from queries, in order to find the top k approximate answers we may need an approximate automaton at distance $|R| + 2k|E|$. For example, consider the case of a graph G with only two nodes, n and m, connected by a single edge labelled a, along with regular expression $R = b$. The top ranking answer is at distance 1, corresponding to substituting b with a. In general, the ith answer is at distance $2i - 1$, corresponding to the substitution of b with a and $i - 1$ insertions of a^- followed by a. (We note that even if the edges of G were not able to be traversed in reverse order, and edge inversions or insertions of inverse edge labels into R were not allowed, the maximum required distance for the approximate automaton would still be $|R| + k|E|$.)

Let M be the approximate automaton at distance $|R| + 2k|E|$. We form the product automaton $H = M \times G$, viewing each node in G as both an initial and a final state. H contains as nodes all the pairs (x, y) such that x is a node of M and y is a node of G, and edges $((x_1, y_1), (x_2, y_2))$ labelled (l, d) such that there is an edge (x_1, x_2) labelled l in M and there is an edge (y_1, y_2) labelled (l, c) in G[4]. The cost d of the edge in H is derived as follows. Transitions in M are either between pairs of states with the same superscript (no edit operation) or from a state with superscript i to one with superscript $i + 1$ (an edit operation). If x_1 and x_2 have the same superscript, $d = \beta \cdot c$; otherwise $d = \alpha + \beta \cdot c$ (since, for simplicity we are assuming all edit operations have cost 1).

Proposition 1. *Let $Q : X, P, Y \leftarrow (X, (R\!:\!P), Y)$ be a simple ERP query, and $G = (V, E)$ be a graph. Then, $x, l_1, v_1, \ldots, v_{n-1}, l_n, y$ is in the approximate top-k answer of Q on G if and only if (s_0^0, x), $l_1, (s_{j_1}^{i_1}, v_1)$, $\ldots, (s_{j_{n-1}}^{i_{n-1}}, v_{n-1}), l_n, (s_f^i, y)$ is in the k shortest paths that connect nodes of the form (s_0^0, X) with nodes of the form (s_f^i, Y) in H.*

Proof. From the construction of $H = M \times G$, it follows that (i) $\theta(X, P, Y) = x, l_1, v_1, \ldots, v_{n-1}, l_n, y$ is an approximate top-k answer of Q on G if and only if $p = (s_0^0, x), l_1, (s_{j_1}^{i_1}, v_1), \ldots, (s_{j_{n-1}}^{i_{n-1}}, v_{n-1}), l_n, (s_f^i, y)$ is a path in H; and (ii) the cost of p (i.e. the sum of its edge costs) is equal to $f(\theta(X, P, Y))$. The proposition follows directly from (i) and (ii). □

Proposition 2. *Let $G = (V, E)$ be a graph and $Q : X, P, Y \leftarrow (X, (R\!:\!P), Y)$ be a simple ERP query. If $|E| > |R|$ and $|E| > |V|$, then the approximate top-k answer of Q on G can be computed in time $O(k|R||V|^2|E|^3(|E| + \log(k|E|)))$.*

---

[4] Of course y_1 may equal y_2 in which case the transition in M is labelled with $(\epsilon, 0)$.

Proof. The automaton M_R has $O(|R|)$ states and $O(|R|^2)$ transitions. Assuming $|E| > |R|$, the approximate automaton M of R at distance $|R| + 2k|E|$ has $O(k|R||E|)$ states and $O(k|R||E|^2)$ transitions. Therefore $H = M \times G$ has $O(k|R||V||E|)$ nodes and $O(k|R||E|^3)$ edges.

Eppstein's algorithm [10] can be used to find the k shortest paths connecting two nodes in a graph. It has complexity $O(u + v \log v + k)$, where u and v are the number of edges and nodes, respectively, in the graph.

In order to compute the approximate top-k answers we use Proposition 1 and proceed in two main steps.

In the first step, we compute the k shortest paths that connect each pair of nodes of the form (s_0^0, X),(s_f^i, Y) in H. For each pair of such nodes we apply Eppstein's algorithm. A single execution of Eppstein's algorithm takes $O(k|R||E|^3 + k|R||V||E| \log(k|R||V||E|))$. Assuming $|E| > |V|$ and $|E| > |R|$, this can be simplified to $O(k|R||E|^2(|E| + \log(k|E|))$.

There are $O(|V|)$ nodes of the form (s_0^0, X), and $O(|V||E|)$ nodes of the form (s_f^i, Y) in H. Therefore, the number of calls to Eppstein's algorithm is in $O(|V|^2|E|)$. Hence, the first step of the computation is in $O(k|R||V|^2|E|^3(|E| + \log(k|E|))$.

In the second step, we select the k shortest paths among the partial lists of paths returned in the first step. There are $O(|V|^2|E|)$ lists of paths. The second step takes time in $O(|V|^2|E| + k \log(|V|^2|E|))$, which is dominated by the complexity of the first step. Therefore the the approximate top-k answer of Q on G can be computed in time $O(k|R||V|^2|E|^3(|E| + \log(k|E|))$. □

Eppstein's algorithm needs to operate on the graph $H = M \times G$. An optimization here is to compute edges of the graph H incrementally, avoiding the precomputation and materialization of the entire graph H. Recall that a node in H is a pair (s_i^j, n), where s_i^j is a state of automaton M and n is a node of G. Each edge of H is labelled with a symbol and a cost. The on-demand computation of edges of H is performed by calling a function *Succ* (shown overleaf) with a node (s_i^j, n) of H. The function returns a set of transitions $\stackrel{e,d}{\rightarrow} (s_k^l, m)$, such that there is an edge in H from (s_i^j, n) to (s_k^l, m) with label (e, d).

To illustrate, consider the conjunct `(?E1,prereq+:?P3,?E2)` of query Q_2 in Example 2. Suppose that $\alpha = 5$ and $\beta = 1$. Calling $Succ(s_0^0, ep21)$ returns transitions $\stackrel{next,\alpha+\beta}{\rightarrow} (s_0^1, ep22)$ (insertion of *next*), and $\stackrel{\epsilon,\alpha}{\rightarrow} (s_f^1, ep21)$ (deletion of *prereq*). Continuing with the first of these, calling $Succ(s_0^1, ep22)$ returns transitions $\stackrel{next,\alpha+\beta}{\rightarrow} (s_0^2, ep23)$ (insertion of *next*), $\stackrel{next-,\alpha+\beta}{\rightarrow} (s_0^2, ep21)$ (insertion of *next*−), and $\stackrel{\epsilon,\alpha}{\rightarrow} (s_f^2, ep22)$ (deletion of *prereq*). The third of these results in an answer `ep21,[next],ep22` at distance 11. Continuing with the first of these, calling $Succ(s_0^2, ep23)$ returns transitions $\stackrel{prereq,\beta}{\rightarrow} (s_f^2, ep24)$ (normal traversal), $\stackrel{\epsilon,\alpha}{\rightarrow} (s_f^3, ep23)$ (deletion of *prereq*), and several higher-cost ones. This results in answers that include `ep21,[next,ep22,next,ep23,prereq],ep24` at distance 13, and `ep21,[next,ep22,next],ep23` at distance 17.

Procedure $\texttt{Succ}(s^i, n)$

$W \leftarrow \emptyset$
for $(n, (a, c), m) \in G$ **do**
  for $p^i \in \texttt{nextStates}(M_R, s^i, a)$ **do**
    add $\overset{a,\beta c}{\rightarrow} (p^i, m)$ to W ; `/* normal traversal */`
for $(m, (a, c), n) \in G$ **do**
  for $p^i \in \texttt{nextStates}(M_R, s^i, a^-)$ **do**
    add $\overset{a^-,\beta c}{\rightarrow} (p^i, m)$ to W ; `/* reverse traversal */`
for $(n, (a, c), m) \in G$ *such that* $\texttt{nextStates}(M_R, s^i, a) = \emptyset$ **do**
  add $\overset{a,\alpha+\beta c}{\rightarrow} (s^{i+1}, m)$ to W ; `/* insertion of` a `*/`
for $(m, (a, c), n) \in G$ *such that* $\texttt{nextStates}(M_R, s^i, a^-) = \emptyset$ **do**
  add $\overset{a^-,\alpha+\beta c}{\rightarrow} (s^{i+1}, m)$ to W ; `/* insertion of` a^- `*/`
for $p^i \in \texttt{nextStates}(M_R, s^i, b)$ *for each* $b \in \Sigma$ **do**
  add $\overset{\epsilon,\alpha}{\rightarrow} (p^{i+1}, n)$ to W ; `/* deletion of` b `*/`
for $(n, (a, c_1), m) \in G$ *and* $(m, (b, c_2), u) \in G$ **do**
  for $p^i \in \texttt{nextStates}(M_R, s^i, b)$ *and* $q^i \in \texttt{nextStates}(M_R, p^i, a)$ **do**
    add $\overset{ba,\alpha+\beta(c_1+c_2)}{\rightarrow} (q^{i+1}, u)$ to W ; `/* swap of` a `and` b `*/`
return W

4 General ERP Queries

We now extend the notion of a simple ERP query to one of a general ERP query in which multiple conjuncts may appear. A *general ERP query* Q is an expression of the form

$$X_{i_1}, P_{i_1}, Y_{i_1}, \ldots, X_{i_r}, P_{i_r}, Y_{i_r} \leftarrow (X_1, (R_1 : P_1), Y_1), \ldots, (X_n, (R_n : P_n), Y_n).$$

where the conjuncts on the righthand side are required to be *acyclic* [11].

The approximate semantics of a general ERP query Q is a straightforward extension to the semantics of simple ERP queries. Given a matching θ, a *pre-answer* to Q is a tuple $\theta((X_1, P_1, Y_1), \ldots, (X_n, P_n, Y_n))$. The pre-answers of Q on G are now ranked according to a function

$$g(\theta((X_1, P_1, Y_1), \ldots, (X_n, P_n, Y_n)) = f_1(\theta(X_1, P_1, Y_1)) + \ldots + f_n(\theta(X_n, P_n, Y_n))$$

where each $f_i(P_i) = \alpha \cdot \mathit{editd}(\theta(X_i, P_i, Y_i), R_i) + \beta \cdot \mathit{cost}(\theta(X_i, P_i, Y_i))$.

The rank position of a pre-answer $\theta((X_1, P_1, Y_1), \ldots, (X_n, P_n, Y_n))$ defines the rank position of its corresponding answer $\theta(X_{i_1}, P_{i_1}, Y_{i_1}, \ldots, X_{i_r}, P_{i_r}, Y_{i_r})$.

A naive way to compute an ERP query is firstly to compute approximate answers of each query atom (which are simple ERP queries so we can use the method explained in Section 3), and then rank-join them according to function g. However, it may be that the top k tuples for one query atom do not join with

those of another. The following proposition shows that we can, in fact, still find the top k answers to an ERP query in polynomial time.

Proposition 3. *Let $G = (V, E)$ be a graph and Q be a general ERP query such that Q is acyclic and the number of head variables in Q is fixed. The top-k approximate answers of Q on G can be computed in time polynomial in the size of G and Q.*

Proof. (sketch) For each conjunct $(X_i, (R_i : P_i), Y_i)$ we consider all possible mappings θ which map X_i and Y_i to nodes in G. Then for each instantiated conjunct $(\theta(X_i), (R_i : P_i), \theta(Y_i))$, we use the algorithm of Section 3 to compute the top-k approximate answers of the simple ERP query associated with this conjunct, along with the cost f_i of each answer. This gives us a relation r_i over the scheme (X_i, P_i, Y_i, C_i), as described in the Introduction. Relation r_i is polynomial in size and can be computed in polynomial time, as shown in the first step in the proof of Proposition 2.

Now we sort each r_i and then apply a rank-join algorithm (according to the cost function g) to $r_1, \ldots, r_n$ in order to obtain the top-k approximate answers of Q on G. Because Q is acyclic and has a fixed number of head variables, the number of intermediate results is polynomial in the size of Q and G, and the answer can be computed in time polynomial in the size of Q and G [11]. □

As noted in Section 3, we would like to compute the top-k approximate answers incrementally. We conjecture that this can be done by modifying the join algorithm we presented in [16], taking into account the evaluation approach sketched in the above proposition, and this is an area of future work.

5 Related Work

As mentioned in the Introduction, Grahne and Thomo [12] use weighted regular transducers for approximate matching of regular path queries, but only for single conjuncts. In more recent work [13], they consider regular path queries in which users can specify preferences by annotating the symbols in queries with weights. However, the setting we consider in the present paper differs from the above in that our graph edges have costs (as well as symbols) and we are are interested in returning the top-k paths to the user. In this sense, our work is related to the considerable amount of work on finding the k shortest paths (see for example [10]), except that our edges are also labelled with symbols over which regular expressions are formed. Such a combination of labels has been considered previously [3], but only for finding *a* shortest path (and with exact matching of symbols) rather than k shortest paths and approximate matching.

The approximate matching aspect of our work is related to a large volume of other previous work in flexible querying, query relaxation and cooperative query answering. For example, Kanza and Sagiv consider querying semistructured data using flexible matchings which allow paths whose edge labels contain those appearing in the query to be matched [18]; such semantics can be captured by our

approach by allowing transpositions and insertions as the only edit operations. In cooperative query answering, overconstrained queries are automatically relaxed, usually by means of query generalisation or containment [5, 9, 15, 22]. Certain of these semantic rewritings can be obtained by the syntactic approximations we consider here. Computing answers incrementally and returning approximate answers to conjunctive regular path queries in ranked order was investigated in our recent work [16]. However, the costs of paths were based only on edit distance rather than a combination of edit distance and path cost. Also, the actual top-k paths were not returned to users, a feature which makes queries both more useful in a number of applications as well as considerably more complicated to evaluate.

6 Conclusions

We have investigated flexible querying of graph-structured data where the top k paths satisfying a query are returned to the user in ranked order. Although we motivated the requirement for flexible path queries by a lifelong learning application, many other areas such as bioinformatics, social network analysis, transportation etc., could benefit from the provision of this functionality.

In this paper, we have focussed on the complexity of evaluating such queries and returning paths to the user in ranked order. However, providing a mechanism to allow users and/or application designers to specify their requirements in terms of approximation and ranking is crucial for such a system to be usable. We are currently working on this as part of an implementation of the query evaluation techniques described here. Such an implementation will allow us to determine both the utility and the practical efficiency of our approximate matching techniques, as well as to investigate suitable values for the coefficients α and β for specific application domains.

Acknowledgements This work was supported by the Royal Society under their International Joint Projects Grant Programme. In addition, Carlos Hurtado was partially funded by Fondecyt project number 1080672.

References

1. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The LOREL query language for semistructured data. *Int. J. Digit. Libr.*, 1(1):68–88, April 1997.
2. B. Aleman-Meza, M. Nagarajan, C. Ramakrishnan, L. Ding, P. Kolari, A. P. Sheth, I. B. Arpinar, A. Joshi, and T. Finin. Semantic analytics on social networks: experiences in addressing the problem of conflict of interest detection. In *Proc. 15th Int. Conf. on the World Wide Web*, pages 407–416, 2006.
3. C. L. Barrett, K. R. Bisset, M. Holzer, G. Konjevod, M. V. Marathe, and D. Wagner. Engineering label-constrained shortest-path algorithms. In *Proc. 4th Int. Conf. on Algorithmic Aspects in Information and Management*, pages 27–37, 2008.

4. T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, and D. Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *Proc. 3rd Int. Semantic Web User Interaction Workshop*, 2006.
5. H. Bulskov, R. Knappe, and T. Andreasen. On querying ontologies and databases. In *Proc. 6th Int. Conf. on Flexible Query Answering Systems*, pages 191–202, 2004.
6. D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Containment of conjunctive regular path queries with inverse. In *Proc. Seventh Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 176–185, 2000.
7. I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *Proc. ACM SIGMOD Conf.*, pages 323–330, 1987.
8. S. de Freitas, I. Harrison, G. Magoulas, A. Mee, F. Mohamad, M. Oliver, G. Papamarkos, and A. Poulovassilis. The development of a system for supporting the lifelong learner. *British Journal of Educational Technology*, 37(6):867–880, 2006.
9. P. Dolog, H. Stuckenschmidt, and H. Wache. Robust query processing for personalized information access on the semantic web. In *Proc. 7th Int. Conf. on Flexible Query Answering Systems*, pages 343–355, 2006.
10. D. Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1998.
11. G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 43(3):431–498, May 2001.
12. G. Grahne and A. Thomo. Approximate reasoning in semi-structured databases. In *Proc. 8th Int. Workshop on Knowledge Representation meets Databases*, 2001.
13. G. Grahne, A. Thomo, and W. W. Wadge. Preferentially annotated regular path queries. In *Proc. 11th Int. Conf. on Database Theory*, pages 314–328, 2007.
14. T. Heath, M. Hausenblas, C. Bizer, and R. Cyganiak. How to publish linked data on the web (tutorial). In *Proc. 7th Int. Semantic Web Conf.*, 2008.
15. C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Query relaxation in RDF. *Journal on Data Semantics*, X:31–61, 2008.
16. C. A. Hurtado, A. Poulovassilis, and P. T. Wood. Ranking approximate answers to semantic web queries. In *Proc. 6th European Semantic Web Conference*, pages 263–277, 2009.
17. I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13:207–221, 2004.
18. Y. Kanza and Y. Sagiv. Flexible queries over semistructured data. In *Proc. Twentieth ACM Symp. on Principles of Databases Systems*, pages 40–51, 2001.
19. Z. Lacroix, H. Murthy, F. Naumann, and L. Raschid. Links and paths through life sciences data sources. In *Proc. 1st Int. Workshop on Data Integration in the Life Sciences*, pages 203–211, 2004.
20. W.-J. Lee, L. Raschid, P. Srinivasan, N. Shah, D. L. Rubin, and N. F. Noy. Using annotations from controlled vocabularies to find meaningful associations. In *Proc. 4th Int. Workshop on Data Integration in the Life Sciences*, pages 247–263, 2007.
21. J. Lehmann, J. Schüppel, and S. Auer. Discovering unknown connections—the DBpedia relationship finder. In *Proc. 1st SABRE Conference on Social Semantic Web*, pages 99–110, 2007.
22. H. Stuckenschmidt and F. van Harmelen. Approximating terminological queries. In *Proc. 5th Int. Conf. on Flexible Query Answering Systems*, pages 329–343, 2002.
23. N. van Labeke, A. Poulovassilis, and G. D. Magoulas. Using similarity metrics for matching lifelong learners. In *Proc. 9th Int. Conf. on Intelligent Tutoring Systems*, pages 142–151, 2008.
24. S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10):83–91, Oct. 1992.