Ranking Approximate Answers to Semantic Web 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 a user querying semistructured data such as RDF without knowing its structure. In these circumstances, it is helpful if the querying system can perform an approximate matching of the user's query to the data and can rank the answers in terms of how closely they match the original query. Our approximate matching framework allows us to incorporate standard notions of approximation such as edit distance as well as certain RDFS inference rules, thereby capturing semantic as well as syntactic approximations. The query language we adopt comprises conjunctions of regular path queries, thus including extensions proposed for SPARQL to allow for querying paths using regular expressions. We provide an incremental query evaluation algorithm which runs in polynomial time and returns answers to the user in ranked order.

1 Introduction

The volume of semistructured data available to users on the web continues to grow significantly. In particular, there has recently been a substantial increase in the amount of RDF data being generated and made available, often in the form of *linked data* [1]. Given the volume and heterogeneity of this data, users are frequently unaware of the structure of the data. This has given rise to linked data browsers such as Tabulator [1]. However, users may also wish to query such large repositories of linked data and so there is a need to assist them by providing querying systems which do not require that users' queries necessarily match exactly the data structures being queried.

In this paper we consider general semistructured data modelled as a graph structure, with RDF linked data being a particular application of this model. Our data model is that of a directed graph $G = (V, E)$, where each node in V is labelled with a constant and each edge e is labelled with a symbol $l(e)$ drawn from a finite alphabet Σ. Such a model does not allow for the representation of RDF blank nodes, e.g., but these are specifically discouraged for linked data [2].

We are interested in developing efficient algorithms which allow for the approximate matching of users' queries on such data, with the answers to queries being returned to users in ranked order. In order to achieve this, we restrict our query language to that of conjunctive regular path queries [3]. A conjunctive regular path query Q consisting of n query triples (or conjuncts) is the form

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

in which 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 Q, and each R_i, $1 \leq i \leq n$, is a regular expression over Σ.

The idea of using regular expressions to specify paths through graphs has been around for a long time (e.g. [4]), having been taken up by the semistructured data community (e.g. [3]), and more recently by the RDF community (e.g. [5, 6]) in proposed extensions to SPARQL [7].

The exact answer to a conjunctive regular path query Q on a graph G is specified as follows. We first find, for each $1 \leq i \leq n$, a relation r_i over the scheme (X_i, Y_i) such that tuple $t \in r_i$ iff there is a path from $t[X_i]$ to $t[Y_i]$ in G that satisfies R_i, that is, 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 Z_1 to Z_m.

Example 1. As an example, consider an RDF graph G involving information about a transport network (Fig. 1). Some nodes represent cities (denoted by URIs) and others represent names of cities. Edges state that we can travel from one city to another city by train, bus, or by airplane.

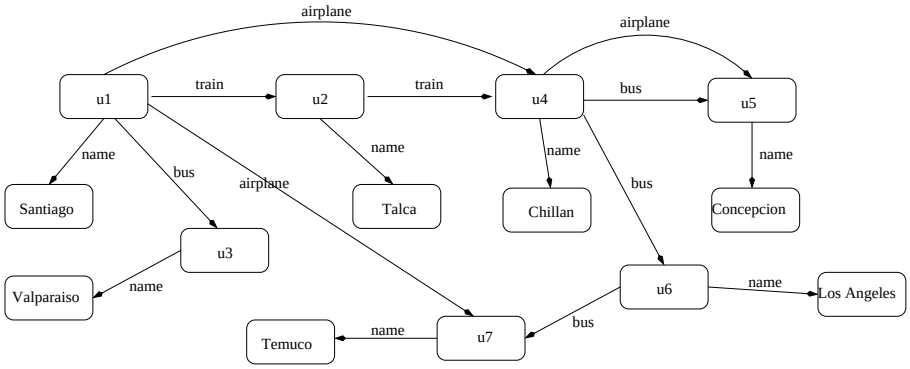

Fig. 1. An RDF graph of a basic transport network.

Suppose that we want to find the cities from which we can travel to city u5 using only airplanes as well as to city u6 using only trains or busses. This can be expressed by the following conjunctive regular path query[3] Q:

$$?X \leftarrow (?X, (airplane)^+, u5), (?X, (train|bus)^+, u6)$$

[3] In the concrete syntax of queries, we precede variable names with '?' as in SPARQL.

When Q is evaluated on G, the set of bindings for $?X$ generated by the first conjunct is $\{u1, u4\}$, while that for the second conjunct is $\{u1, u2, u4\}$. Hence the answer is $\{u1, u4\}$. □

As stated above, we also want to allow for approximate matching of queries on graphs. Such approximate (or similarity) matching can take many forms and has been the subject of much study [8–13]. In general, we are interested in using *weighted regular transducers* to model the kinds of approximations allowed since single conjunct queries with a transducer applied can be evaluated *incrementally* in polynomial time [9]. Incremental evaluation allows for answers to be returned incrementally to a user in ranked order, without the user having to wait for entire query evaluation to end. In this paper, we extend these results to show that *multiple* conjunct queries can also be evaluated incrementally in polynomial time, using ideas from [14].

A weighted regular transducer is basically a finite state automaton in which the transitions are labelled with triples rather than single symbols. A transition from state s to state t labelled with triple (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 would be interpreted as specifying that symbol a in a query can match the label b of a graph edge with cost i.

Weighted regular transducers provide for greater generality than we need for most of our examples. They also make the agorithms more difficult to explain. Hence for most of the paper we restrict ourselves to using approximate regular expression matching [15], which can easily be specified using weighted regular transducers [9]. The edit operations which we allow in approximate matching are insertions, deletions and substitutions of symbols, along with insertions of inverted symbols (corresponding to edge reversals) and transpositions of adjacent symbols, each with an assumed cost of 1.

Example 2. Consider again the application in Example 1 and assume that a user who has little knowledge of the structure of the data formulates the following query to retrieve all the cities reachable from Santiago by non-stop flights:

$$?X \leftarrow (Santiago, airplane, ?X)$$

The query does not return any answers because it does not match the structure of the graph. However, allowing edit operations such as insertions of symbols and inverted symbols (indicated by using '$-$' as a superscript to the symbol and corresponding to matching an edge in the reverse direction), each at an assumed cost of 1, the regular expression $airplane$ can be successively relaxed to the regular expression $((name)^- \cdot airplane \cdot name)$, which captures as answers the city names of Temuco and Chillan. Since the cost of each edit operation is set to 1, these answers can be regarded as having a distance of 2 from the original query (two insertions).

Suppose now that, after acquiring some knowledge of the structure of the data, the user formulates the following query to find cities reached from Santiago by train:

$$?X \leftarrow (Santiago, ((name)^- \cdot (train)^+ \cdot name), ?X)$$

If the user is prepared to consider replacing a train journey by a bus journey, but at a cost of 1 per replacement, this can be specified as the only edit operation allowed on the user's query. The same effect can be achieved by a transducer with a single state with two self-edges to/from it: one labelled with $(train, 0, train)$, i.e. *train* can be matched by *train* with cost 0, and one labelled with $(train, 1, bus)$, i.e. *train* can be matched by *bus* with cost 1. Now, the results of the query will be returned to the user ranked by the number of bus journeys: Talca and Chillan will have rank 0, Valparaiso, Concepcion and Los Angeles will have rank 1, and Temuco will have rank 2. □

We do not expect that a user would necessarily want or be able to design a weighted regular transducer in order to capture their requirements in terms of approximate matching. Instead, we would expect that a query interface would provide a user with readily understandable options from which to select their requirements. Our concern in this paper is in showing that weighted regular transducers and conjunctive regular path queries provide a good framework within which to investigate approximate querying of graph-structured data.

In the next section, we consider the special case of computing approximate answers for queries consisting only of a single conjunct. We show that approximate answers can be computed in time which is polynomial in the size of the query and the input graph. We also show that these answers can be computed incrementally and returned to the user in ranked order. Section 3 generalises to the case of multi-conjunct queries and shows that incremental computation can still be achieved in polynomial time as long as the queries are acyclic and have a fixed number of head variables. Related work is covered in Sect. 4 which also shows how our approach generalises previous work in a number of areas. Conclusions and future work are described in Sect. 5.

2 Single-Conjunct Queries

Recall that our data model is that of a directed graph $G = (V, E)$, where each node in V is labelled with a constant and each edge e is labelled with a symbol $l(e)$ drawn from a finite alphabet Σ. In queries, we will allow edges to be traversed both from their source to their target node and in reverse, from their target to their source node. So we now introduce the notion of the *inverse* of an edge label l, denoted by l^-, which is used to specify a reverse traversal of an edge. Let $\Sigma^- = \{l^- \mid l \in \Sigma\}$. If $l \in \Sigma \cup \Sigma^-$, we use l^- to mean the *inverse* of l, that is, if l is a for some $a \in \Sigma$, then l^- is a^-, while if l is a^- for some $a \in \Sigma$, then l^- is a.

A *single-conjunct query* consists of an expression of the form:

$$Z_1, Z_2 \leftarrow (X, R, Y)$$

where X and Y are constants or variables, R is a regular expression, and each of Z_1 and Z_2 is one of X or Y. 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.

2.1 Exact Semantics

Because of the presence of inverse operators in our queries, we need to introduce the notion of a *semipath* in G in order to define the semantics of a query [3]. A *semipath* p in $G = (V, E)$ from $x \in V$ to $y \in V$ is a sequence of the form $(v_1, l_1, v_2, l_2, v_3, \ldots, v_n, l_n, v_{n+1})$, where $n \geq 0$, $v_1 = x$, $v_{n+1} = y$ and for each v_i, l_i, v_{i+1} either $v_i \xrightarrow{l_i} v_{i+1} \in E$ or $v_{i+1} \xrightarrow{l_i^-} v_i \in E$. The semipath p *conforms* to regular expression R if $l_1 \cdots l_n \in L(R)$, the language denoted by R.

Given a single-conjunct query Q and graph G, let θ be a matching from variables and constants of Q to nodes of G that maps each constant to itself. We say that the tuple $\theta(Z_1, Z_2)$ *satisfies* Q on G if there is a semipath from $\theta(X)$ to $\theta(Y)$ which conforms to R. The *exact answer* of Q on G, denoted $Q(G)$, is the set of tuples which satisfy Q on G.

2.2 Approximate Semantics

We also allow for approximate answers to queries using approximate regular expression matching. As stated in the Introduction, we can in fact use a weighted regular transducer to specify the approximations, but it is easier to explain using only approximate regular expression matching. The edit operations that we allow for approximate matching are insertions, deletions, inversions, substitutions, and transpositions of adjacent symbols. The user can specify which, if any, of these edit operations should be undertaken by the system when answering a particular query. A different cost can be associated with applying each operation.

For strings, the edit distance from string w to string v is the minimum cost of any sequence of edit operations which transforms w to v. In our case, we treat the labels on edges as atomic and it is sequences of such labels which are transformed using edit operations. The edit distance from a semipath w to a semipath v is the minimum cost of any sequence of edit operations which transforms the sequence of edge labels of w to the sequence of edge labels of v; for the sake of simplicity, in this paper we assume that all edit operations have a cost of 1. The distance of a semipath w to a regular expression R, denoted $dist(w, R)$, is the minimum edit distance from w to any semipath that conforms to R.

Given a matching θ from variables and constants of a query Q to nodes in a graph G, we have that the tuple $\theta(Z_1, Z_2)$ has distance $dist(p, R)$ to Q, where p is a semipath from $\theta(X)$ to $\theta(Y)$ which has the minimum edit distance to R of any semipath from $\theta(X)$ to $\theta(Y)$ in G. Note that if p conforms to R, then $\theta(Z_1, Z_2)$ has distance zero to Q.

Given a graph $G = (V, E)$ and single-conjunct query Q, the *approximate top-k answer* of Q on G is a list containing the k tuples $\theta(Z_1, Z_2)$ with minimum

distance to Q, ranked in order of increasing distance to Q. The *approximate answer* of Q on G contains all the tuples at any distance to the query (ranked by distance). Notice that for $k = |E|^2$, the approximate answer is equal to the approximate top-k answer.

2.3 Approximate Automaton

In this section we will introduce the notion of the approximate automaton of a regular expression R: the *approximate automaton of R at distance d*, where d is an integer, accepts all strings at distance at most d from R.

For any regular expression R we can construct an NFA 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. Then, the approximate automaton at distance d is obtained following a standard construction used in approximate string matching [15].

We make d copies of M_R, each copy denoted M_R^j, $0 \leq j \leq d$, whose states are those of M_R with superscript j, representing distance j from the original automaton M_R. The only initial state in the approximate automaton M is s_0^0, in other words the initial state of M_R at distance 0. The final state of each M_R^j remains a final state in the approximate automaton.

Subautomaton M_R^j is connected to subautomaton M_R^{j+1} by $O(|R|)$ transitions for deletions, $O(|\Sigma| \cdot |R|)$ transitions for insertions, $O(|\Sigma| \cdot |R|)$ transitions for substitutions, $O(|R|)$ transitions for inversions, and $O(|R|^2)$ transitions for transpositions.

Figure 2 shows the skeleton of the construction for insertions and deletions. Transitions for deletions are labelled with ϵ, and transitions for insertions are labelled with Σ, Σ^- (because we may insert a label or an inverse label). Figure 3 shows the additional states and transitions needed for transpositions, inversions, and substitutions.

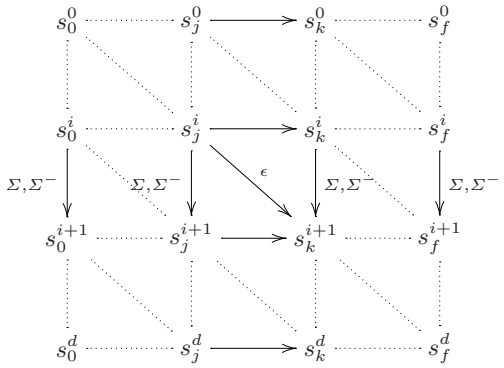

Fig. 2. Approximate automaton at level d (insertions and deletions only).

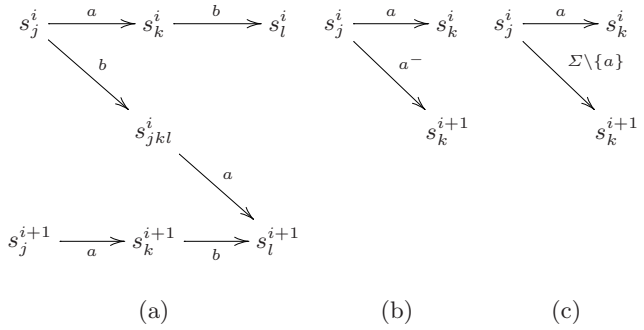

Fig. 3. Transitions for (a) transpositions, (b) inversions and (c) substitutions.

The following lemma shows two basic properties of the approximate automaton.

Lemma 1. *Let $G = (V, E)$ be a database, R be a regular expression, and M be the approximate automaton of R at distance d. (i) M has size $O(d \cdot (|R| + |\Sigma| \cdot |R| + |R|^2))$. (ii) If $d = |R| + |E|$, then every semipath in G conforms to M.*

Proof. (i) This follows directly from the definition of the approximate automaton. (ii) Any semipath w in G can be accepted by the approximate automaton through transitions labelled with ϵ that delete all the symbols that appear in R followed by transitions that add all the labels of w. □

2.4 Evaluation of Single-Conjunct Queries

Consider the single-conjunct query Q:

$$Z_1, Z_2 \leftarrow (X, R, Y)$$

In order to evaluate Q on graph G, we start by constructing the approximate automaton M of R at distance $d = |R|+|E|$. We next 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 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 in G.

We associate a cost with each transition in the product automaton as follows. Any transition between a pair of states such that both derive from states of the approximate automaton M with the same superscript has cost 0, while all other transitions have cost 1.

Assume for the moment that X is a node n of G. We then perform a uniform cost traversal of H starting from vertex (s_0^0, n). We keep a list of visited nodes, so that no node is visited twice. The size of the search tree is in $O(|V|)$, where V is the set of vertices of H. Whenever we reach a vertex (s_f^i, m) in H we output

m. The distance of m to the query Q is given by the total cost of the path from (s_0^0, n) to the vertex (s_f^i, m) in the traversal tree.

When both Z_1 and Z_2 are variables and assuming that Z_1 equals X, Q can be evaluated by answering the query $n, Z_2 \leftarrow (n, R, Y)$ for each node n in G.

Lemma 2. *Let $G = (V, E)$ be a graph and Q be a single-conjunct query using regular expression R. Assuming $|E| > |R|$, the approximate answer of Q on G can be computed in time $O(|V| \cdot |E|^2 \cdot |R|^2)$.*

Proof. Assuming $|E| > |R|$, the size of the approximation automaton M of R at distance $|R| + |E|$ is $O(|E| \cdot |R|^2)$. The size of $H = M \times G$ is $O(|E|^2 \cdot |R|^2)$ (we can discard disconnected nodes from H). So computing the approximate answer takes $O(|V| \cdot |E|^2 \cdot |R|^2)$, that is $|V|$ traversals of H, each one at cost equal to the size of H. □

Example 3. Consider the graph depicted in Fig. 1 and a slight modification of the query from Example 2:

$$?X \leftarrow (Santiago, (airplane)^+, ?X)$$

Figure 4 shows the approximate automaton of R at distance $d = 2$, where we use a for *airplane* and n for *name* and, for simplicity, we consider only insertions of n or n^- and inversion of a.

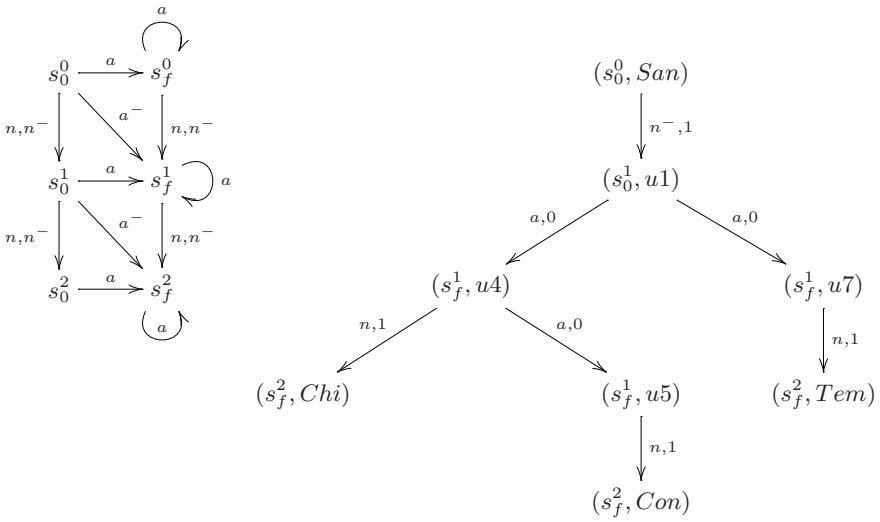

Fig. 4. The approximate automaton ($d = 2$) for $R = a^+$ and a traversal tree.

Figure 4 also shows the traversal tree starting from (s_0^0, San), where San, Chi, Con and Tem are abbreviations for *Santiago*, *Chillan*, *Concepcion* and *Temuco*, respectively. Here we assume that all operations have cost 1.

As we saw previously, there are no answers at rank 0. The answers at rank 1 (indicated by being associated with state s_f^1) are $u4$, $u5$ and $u7$, while those at rank 2 are Chi, Con and Tem. □

2.5 Incremental Evaluation

We may compute edges of the graph $H = M \times G$ above incrementally, avoiding the precomputation and materialization of the entire graph H. Recall that a vertex 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 weight.

The on-demand computation of edges of H is performed by calling a function *Succ* with a node (s_i^j, n) of H. The function returns a set of transitions $\stackrel{e,w}{\rightarrow} (s_k^l, m)$, such that there is an edge in H from (s_i^j, n) to (s_k^l, m) with label e and weight w. As an example, in Fig. 4 $Succ(s_f^1, u4)$ returns the following transitions: $\stackrel{a,0}{\rightarrow} (s_f^1, u5)$ (without relaxation—cost 0) and $\stackrel{n,1}{\rightarrow} (s_f^2, Chi)$ (insertion of label n at cost 1).

A procedure for computing *Succ* is shown overleaf. Here, M_R is the original NFA constructed to recognise $L(R)$. The function `nextStates`(M_R, s, c) returns the set of states in M_R that can be reached from state s on reading input c. We assume that this function can also take as an argument a state s^i in the approximate automaton at distance i from M_R. If `nextStates`(M_R, s, c) returns $\{s_1, s_2, \ldots, s_k\}$, then `nextStates`$(M_R, s^i, c)$ returns $\{s_1^i, s_2^i, \ldots, s_k^i\}$.

Incremental evaluation proceeds by performing the initialisations shown below, where d is used to record distances of nodes, $\texttt{visited}_R$ stores triples (v, n, s) representing the fact that node n was visited in state s starting from node v, and Q_R is a priority queue ordered by distance. The procedure *getNext* (see overleaf) returns the next tuple for a conjunct in ranked order.

Incremental evaluation for a single-conjunct query

```
begin
    construct NFA M_R for R, with initial state s_0 and final state s_f
    visited_R ← ∅
    d ← 0
    foreach node n in G do enqueue(Q_R, (n, n, s_0, d))
    while (v, n, d) ← getNext(X, R, Y) ≠ null do output (v, n, d)
end
```

In particular, when called with conjunct (X, R, Y), the procedure *getNext* begins by dequeuing a tuple (v, n, s, d) from Q_R, where v and n are nodes in G, s is a state in the NFA M_R, and d is the distance associated with visiting n in state s having started from v. If s is the final state then the triple (v, n, d) is returned. Otherwise we continue to traverse G from n.

Example 4. If we apply our incremental evaluation algorithm to Example 3, we only need to enqueue $(San, San, s_0, 0)$ initially because X is the constant San. This tuple is dequeued by *getNext* and $Succ(s_0, San)$ is called. Since the only

Procedure `Succ`(s^i, n)

Input: state s^i and graph node n
Output: set of transitions which are successors of (s^i, n)
$W \leftarrow \emptyset$
for $(n, a, m) \in G$ **do**
  for $p^i \in$ `nextStates`(M_R, s^i, a) **do**
    add $\stackrel{a,0}{\rightarrow} (p^i, m)$ to W ; `/* normal traversal */`
for $(m, a, n) \in G$ **do**
  for $p^i \in$ `nextStates`(M_R, s^i, a^-) **do**
    add $\stackrel{a^-,0}{\rightarrow} (p^i, m)$ to W ; `/* reverse traversal */`
for $(n, a, m) \in G$ *such that* `nextStates`$(M_R, s^i, a) = \emptyset$ **do**
  add $\stackrel{a,1}{\rightarrow} (s^{i+1}, m)$ to W ; `/* insertion of` a `*/`
for $(m, a, n) \in G$ *such that* `nextStates`$(M_R, s^i, a^-) = \emptyset$ **do**
  add $\stackrel{a^-,1}{\rightarrow} (s^{i+1}, m)$ to W ; `/* insertion of` a^- `*/`
for $p^i \in$ `nextStates`(M_R, s^i, b) *for each* $b \in \Sigma$ **do**
  add $\stackrel{\epsilon,1}{\rightarrow} (p^{i+1}, n)$ to W ; `/* deletion of` b `*/`
for $(n, a, m) \in G$ *and* $(m, b, u) \in G$ **do**
  for $p^i \in$ `nextStates`(M_R, s^i, a) *and* $q^i \in$ `nextStates`(M_R, p^i, b) **do**
    add $\stackrel{ba,1}{\rightarrow} (q^{i+1}, u)$ to W ; `/* swap of` a `and` b `*/`
return W

Procedure `getNext`(X, R, Y)

Input: node in query evaluation tree corresponding to conjunct (X, R, Y)
Output: triple (v, n, d), where v and n are instantiations of X and Y
while *nonempty*(Q_R) **do**
  $(v, n, s, d) \leftarrow$ *dequeue*(Q_R)
  add (v, n, s) to `visited`$_R$
  if $s = s_f^i$ *for some* i **then return** (v, n, d)
  foreach $\stackrel{c,w}{\rightarrow} (s', m) \in Succ(s, n)$ *s.t.* $(v, m, s') \notin$ `visited`$_R$ **do**
    enqueue$(Q_R, (v, m, s', d + w))$
return *null*

edge adjacent to San in G is one labelled $name$ with target San and source $u1$, and there is no transition labelled with n^- from s_0 in M_R, $Succ$ adds $\stackrel{n^-,1}{\rightarrow}$ $(s_0^1, u1)$ to W and returns. This results in $(San, u1, s_0^1, 1)$ being enqueued.

The while loop now repeats and $(San, u1, s_0^1, 1)$ is dequeued. $Succ(s_0^1, u1)$ is called and because there is a transition labelled $airplane$ from s_0^1 to s_f^1 in M_R and edges in G from $u1$ to $u4$ and $u7$ labelled $airplane$, $\stackrel{a,0}{\rightarrow}$ $(s_f^1, u4)$ and $\stackrel{a,0}{\rightarrow}$ $(s_f^1, u7)$ are added to W. Although the $Succ$ procedure shown would also consider traversals to $u2$ and $u3$ by inserting bus and $train$ respectively, recall that we are assuming for this example that only $name$ and $(name)^-$ can be added. After $Succ$ returns, $(San, u4, s_f^1, 1)$ and $(San, u7, s_f^1, 1)$ are enqueued. These tuples are at the front of the queue and when dequeued result in $(San, u4, 1)$ and $(San, u7, 1)$ being returned because s_f^1 is a final state. The algorithm continues in this way until the entire tree shown in Fig. 4 has been traversed. □

3 Multi-Conjunct Queries

Assume that we have a conjunctive query Q consisting of n conjuncts:

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

in which each X_i and Y_i, $1 \leq i \leq n$, is a variable or constant, and each Z_i is a variable appearing in the body of Q. Given a matching θ from variables and constants of Q to nodes in a graph G, we have that the tuple $\theta(Z_1, \ldots, Z_m)$ has distance $dist(p_1, R_1) + \cdots + dist(p_n, R_n)$ to Q, where each p_i is the semipath from $\theta(X_i)$ to $\theta(Y_i)$ which has the minimum edit distance to R_i of any semipath from $\theta(X_i)$ to $\theta(Y_i)$ in G. The *approximate top-k answer* of Q on G is the list of k tuples of the form $\theta(Z_1, \ldots, Z_m)$ with minimum distance to Q, ranked in order of increasing distance to Q.

We also require that the conjuncts of a query Q are *acyclic*, which is needed to ensure polynomial-time evaluation[4] and means that there is a join tree T whose nodes are the conjuncts of Q such that, for every variable in Q, the subgraph of T induced by the conjuncts containing the variable is connected.

Our query evaluation algorithm is based on one of the rank-join algorithms developed by Ilyas et al. [14], specifically their hash ripple join. That algorithm takes as input m relations, each of whose sets of tuples has already been ranked according to some citerion, a join condition, a monotone combining ranking function and the number k of ranked results to be output. Ranking in their case is by decreasing value of some score, whereas in our case it is by increasing distance from Q.

In our case, given a graph $G = (V, E)$, for each conjunct (X_i, R_i, Y_i) we can use the algorithm described in Sect. 2.5 to compute a relation r_i over scheme (X_i, Y_i, D_i) where, for tuple $t \in r_i$, $t[D_i]$ represents the minimum distance from

[4] It is well known that the problem of deciding if the natural join of n relations is nonempty is NP-complete.

R_i of any path from $t[X_i]$ to $t[Y_i]$ in G. Joining the results then uses the hash ripple join of [14].

Lemma 3. *Given a graph G and a multi-conjunct query Q with a fixed number of head variables m, the approximate answer of Q on G can be computed in time polynomial in the size of Q and G.*

Proof. Each conjunct in Q can be evaluated in polynomial time, as shown in Lemma 2. The result follows from the acyclicity condition and the fixed number of head variables [16]. □

Because Q is acyclic, we can construct a query evaluation tree E for Q, which consists of nodes denoting join operators and nodes representing conjuncts of Q. We initialise all the algorithm structures by calling the procedure *open*, shown below, with the root of E. The part of procedure *open* dealing with nodes denoting joins is adapted from [14]: hash tables are built for each child node (operands LN and RN) and *open* is called recursively for each; the $firstTuple$ property of each is set to *true* (this is used in the $getNext$ procedure—see below). The part of *open* dealing with nodes denoting conjuncts (X, R, Y) performs the same initialisations as in Sect. 2.5. For notational simplicity, we assume that each regular expression in a conjunctive query Q is distinct. Hence, for conjunct (X, R, Y) we can use M_R to denote its NFA, Q_R its priority queue, etc.

Procedure `open(`N`)`

Input: node N in the query evaluation tree
allocate a priority queue Q_N associated with node N
if N *is* $LN \bowtie RN$ **then**
  build hash tables for LN and RN
  $open(LN)$; $LN.firstTuple \leftarrow true$
  $open(RN)$; $RN.firstTuple \leftarrow true$
  $T_{LN \bowtie RN} \leftarrow 0$; `/* T is the threshold */`
else `/* N is a conjunct (X,R,Y) */`
  construct NFA M_R for R, with initial state s_i and final state s_f
  $\texttt{visited}_R \leftarrow \emptyset$
  $d \leftarrow 0$
  foreach *node* n *in* G **do** $enqueue(Q_R, (n, n, s_i, d))$

Incremental evaluation of query Q proceeds by calling the procedure $getNext$ with the root of the join tree E. There are two versions of $getNext$, one for when the evaluation tree node represents a join, shown overleaf and adapted from [14], and one for when it represents a conjunct, which was discussed previously in Sect. 2.5.

The $getNext$ procedure for a join begins by choosing from which operand of the join to retrieve tuples. We do not consider here the various heuristics that might be used to decide this. For a tuple t, we denote the distance value of t

Procedure `getNext`$(LN \bowtie RN)$

Input: node N (corresponding to a join) in the query evaluation tree
while $empty(Q_{LN \bowtie RN})$ **or** $head(Q_{LN \bowtie RN})[D] > T_{LN \bowtie RN}$ **do**
  determine next input I // either LN or RN
  $t \leftarrow getNext(I)$
  if $I.firstTuple$ **then**
    $I_{top} \leftarrow t[D]$
    $I.firstTuple \leftarrow false$
  $I_{bottom} \leftarrow t[D]$
  $T_{LN \bowtie RN} \leftarrow \min(LN_{top} + RN_{bottom}, LN_{bottom} + RN_{top})$
  insert t in hash table for I
  probe other hash table with t
  foreach *valid join combination* u *of* t *with* s, *say* **do**
    $u[D] \leftarrow t[D] + s[D]$
    $enqueue(Q_{LN \bowtie RN}, u)$
return $dequeue(Q_{L \bowtie R})$

by $t[D]$. For input I (either LN or RN), I_{top} represents the distance value of the first tuple retrieved from I (i.e., the smallest distance in I), while I_{bottom} represents the distance of the most recently retrieved tuple from I. Although I_{top} will always be 0 when I is a conjunct, this will not necessary be the case when I is a join. The threshold value T represents the smaller of the two distances given by $LN_{top} + RN_{bottom}$ and $LN_{bottom} + RN_{top}$. These two values give the possible distances arising from joining the first tuple of LN with the most recent tuple from RN, or vice versa, either of which remains possible until the end of the operation. The smaller of these two distances gives the smallest possible distance for a join tuple that has yet to be computed; in other words, no tuple that might result from the join of tuples yet to be retrieved with tuples already retrieved or yet to be retrieved can have a distance less than T. It is therefore safe to output a join tuple whose distance is equal to T.

4 Related Work and Generalisations

Jagadish et al. [11] develop a framework for similarity-based queries, but it turns out to be too powerful to permit efficient query evaluation. For this reason, Grahne and Thomo [9] restrict themselves to using weighted regular transducers for performing transformations to regular path queries in order for them to match semi-structured data approximately. Their regular path queries consist only of a single conjunct and do not include inversions.

Similarity-based querying is also the focus of iSPARQL [13], where resources are compared using similarity measures. One similarity measure used is that of edit distance between strings. However, similarity is measured with respect to the resources themselves rather than the paths connecting resources; thus it is complementary to our approach.

SPARQLeR extends SPARQL with regular path expressions designed for querying semantic associations [5]. Only exact answers are considered. nSPARQL adds *nested* regular expressions to SPARQL and shows that these are necessary in order to answer queries using the semantics of the RDFS vocabulary by directly traversing the RDF graph [6]. It would be interesting to see whether our approach of augmenting ordinary regular expressions with a regular transducer could similarly achieve this.

Kanza and Sagiv consider querying semistructured data using flexible matchings which allow paths whose edge labels simply contain those appearing in the query to be matched [12]. 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 [8, 10]. Dolog et al. [8] use conditional rewriting rules on query patterns to perform both query refinement by including user preferences as well as query relaxation. Our edit operations and, more generally, regular transducers also effectively introduce a form of query rewriting, so it would be interesting to compare the relative expressive power of the two approaches. Our previous work in [10] was concerned with relaxing queries on RDF by considering query generalisation with respect to an RDFS vocabulary (see that paper for references to other work adopting a similar approach). The substitution edit operation of our work here subsumes the property relaxation semantics used in [10].

As suggested by Example 2 where a user was prepared to consider travelling by bus rather than by train, our approach in this paper has some overlap with work which allows users to specify preferences in queries (e.g. [17]).

We can also apply our work to keyword search in relational databases. Given nodes v_1 to v_n in G and assuming that only insertion edit operations are allowed in approximate matchings, the query

$$X \leftarrow (X, \epsilon, v_1), \ldots, (X, \epsilon, v_n)$$

returns as its first result a *median* of G, that is, a node with minimum total distance to v_1 to v_n. A median is effectively what the BANKS algorithm finds [18], where v_1 to v_n represent keywords in a relational database.

5 Conclusions

In this paper, we have developed a framework within which we can model various aspects of approximate answers to queries on semistructured resources including RDFS. We have presented incremental algorithms which allow answers to be returned to a user in ranked order in polynomial time.

There are several improvements and generalisations which we are investigating. One obvious one is a mechanism to allow users and/or interface designers to specify their requirements in terms of approximation. Another is the ability to have the paths themselves returned to users, rather than simply nodes.

We are currently working on an implementation of the algorithms presented above. This will allow us to determine both the utility and the practical efficiency of the various operations for approximate matching.

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. Berners-Lee, T., Chen, Y., Chilton, L., Connolly, D., Dhanaraj, R., Hollenbach, J., Lerer, A., Sheets, D.: Tabulator: Exploring and analyzing linked data on the semantic web. In: Proc. 3rd Int. Semantic Web User Interaction Workshop. (2006)
2. Heath, T., Hausenblas, M., Bizer, C., Cyganiak, R.: How to publish linked data on the web (tutorial). In: Proc. 7th Int. Semantic Web Conf. (2008)
3. Calvanese, D., Giacomo, G.D., Lenzerini, M., Vardi, M.Y.: Containment of conjunctive regular path queries with inverse. In: Proc. Seventh Int. Conf. on Principles of Knowledge Representation and Reasoning. (2000) 176–185
4. Cruz, I.F., Mendelzon, A.O., Wood, P.T.: A graphical query language supporting recursion. In: Proc. ACM SIGMOD Conf. (1987) 323–330
5. Kochut, K., Janik, M.: SPARQLeR: Extended SPARQL for semantic association discovery. In: Proc. 4th European Semantic Web Conference. (2007) 145–159
6. Pérez, J., Arenas, M., Gutierrez, C.: nSPARQL: A navigational language for RDF. In: Proc. 7th Int. Semantic Web Conf. (2008) 66–81
7. Prud'hommeaux, E., Seaborne, A., eds.: SPARQL Query Language for RDF. W3C Candidate Recommendation (6 April 2006)
8. Dolog, P., Stuckenschmidt, H., Wache, H.: Robust query processing for personalized information access on the semantic web. In: Proc. 7th Int. Conf. on Flexible Query Answering Systems. (2006) 343–355
9. Grahne, G., Thomo, A.: Approximate reasoning in semi-structured databases. In: Proc. 8th Int. Workshop on Knowledge Representation meets Databases. (2001)
10. Hurtado, C.A., Poulovassilis, A., Wood, P.T.: Query relaxation in RDF. Journal on Data Semantics **X** (2008) 31–61
11. Jagadish, H.V., Mendelzon, A.O., Milo, T.: Similarity-based queries. In: Proc. Fourteenth ACM Symp. on Principles of Databases Systems. (1995) 36–45
12. Kanza, Y., Sagiv, Y.: Flexible queries over semistructured data. In: Proc. Twentieth ACM Symp. on Principles of Databases Systems. (2001) 40–51
13. Kiefer, C., Bernstein, A., Stocker, M.: The fundamentals of iSPARQL: A virtual triple approach for similarity-based semantic web tasks. In: Proc. 6th Int. Semantic Web Conf. (2007) 295–309
14. Ilyas, I.F., Aref, W.G., Elmagarmid, A.K.: Supporting top-k join queries in relational databases. The VLDB Journal **13** (2004) 207–221
15. Wu, S., Manber, U.: Fast text searching allowing errors. Commun. ACM **35**(10) (October 1992) 83–91
16. Gottlob, G., Leone, N., Scarcello, F.: The complexity of acyclic conjunctive queries. J. ACM **43**(3) (May 2001) 431–498
17. Siberski, W., Pan, J.Z., Thaden, U.: Querying the semantic web with preferences. In: Proc. 5th Int. Semantic Web Conf. (2006) 612–624
18. Bhalotia, G., Hulgeri, A., Nakhe, C., Chakrabarti, S., Sudarshan, S.: Keyword searching and browsing in databases using BANKS. In: Proc. 18th Int. Conf. on Data Engineering. (2002) 431–440