Approximation and Relaxation of Semantic Web Path Queries

Alexandra Poulovassilis[a], Petra Selmer[a], Peter T. Wood[a]

[a] *Knowledge Lab, Department of Computer Science and Information Systems, Birkbeck, University of London, UK*

Abstract

Given the heterogeneity of complex graph data on the web, such as RDF linked data, it is likely that a user wishing to query such data will lack full knowledge of the structure of the data and of its irregularities. Hence, providing flexible querying capabilities that assist users in formulating their information seeking requirements is highly desirable. In this paper we undertake a detailed theoretical investigation of query approximation, query relaxation, and their combination, for this purpose. The query language we adopt comprises conjunctions of regular path queries, thus encompassing recent extensions to SPARQL to allow for querying paths in graphs using regular expressions (SPARQL 1.1). To this language we add standard notions of query approximation based on edit distance, as well as query relaxation based on RDFS inference rules. We show how both of these notions can be integrated into a single theoretical framework and we provide incremental evaluation algorithms that run in polynomial time in the size of the query and the data, returning answers in ranked order of their 'distance' from the original query. We also combine for the first time these two disparate notions into a single 'flex' operation that simultaneously applies both approximation and relaxation to a query conjunct, providing even greater flexibility for users, but still retaining polynomial time evaluation complexity and the ability to return query answers in ranked order.

Keywords: Graph query languages, Query approximation, Query relaxation

1. Introduction

The volume of graph-structured data on the web continues to grow, most recently in the form of RDF Linked Data [1]. At the time of writing, there were 295 publicly-accessible large datasets, spanning a variety of domains, such as the life sciences, geographical and government domains[1]. The prevalence

[1]http://lod-cloud.net

of graph DBMSs such as Sparksee[2], Neo4j[3] and OrientDB[4], has also greatly increased over the past few years, and they have been used in areas as diverse as social network analysis[5], recommendation services[6] and bioinformatics[7]. The increasing awareness that keyword-based searches alone do not provide the user with enough semantics or contextual structure to return useful answers from information on the web is exemplified by the recent inception of the Google Knowledge Graph [2]. This uses semantic information garnered from Google's knowledge base to enhance the search results returned by their search engine.

The data relating to such graph-modelled application domains may be complex, heterogeneous and evolving in terms of its structure and content, making it difficult for users to formulate queries that precisely match their information seeking requirements. In this paper we consider the application of *query approximation* and *query relaxation* techniques to the evaluation of *conjunctive regular path queries* (*CRPQ*s) over graph data, with the aim of assisting users in formulating queries over complex, irregular graph-structured data and in retrieving results that are of relevance to them. We define the italicised terms in Section 2 and begin here with an overview of our running example, to motivate our work.

Motivating Example: Suppose two flight insurance companies agree to share some of their data. Figure 1 shows part of the merged data graph that might arise. For simplicity here, we assume that common concepts have the same name in both the datasets and that the subscripts on the node and edge labels indicate the dataset from which each was derived (dataset 1 or dataset 2). F_1 and F_2 are classes, representing the Flight class in each of the datasets. Similarly, P_1 and P_2 represent the Person class; E_1 represents the Employee class — which is present only in dataset 1; and N_1 and N_2 represent the class National Insurance Number (which is used in the administration of the UK National Social Security system). Of the various edge labels in Figure 1, explained in full in the caption, fn denotes $flightNumber$, ppn denotes $passengerPassportNumber$, and pn denotes $passportNumber$.

Figure 2 shows additional classes, properties, and relationships between them, that serve to semantically integrate the two datasets. Again, the various node and edge labels are explained in full in the caption to Figure 2.

A user familiar with dataset 1 may pose the following CRPQ, Q_1, in an attempt to find the passport numbers of passengers on flight number 'FL56' (as instantiations of the variable Y):

$$Y \leftarrow (\text{'FL56'}, fn_1, Y), (Y, pn_1^-.type, P_1)$$

This query however returns no answers because of errors in the first query *conjunct*, ('FL56', fn_1, Y). The edge label ought to be fn_1^- instead of fn_1,

[2]http://sparsity-technologies.com/
[3]http://neo4j.com/
[4]http://orientdb.com/
[5]http://neo4j.com/use-cases/social-network, https://snap.stanford.edu/data/
[6]http://neo4j.com/use-cases/real-time-recommendation-engine/
[7]http://bio4j.com/

indicating an incoming, rather than an outgoing, relationship from the node denoting the flight number to the adjacent node. Moreover, the edge label ppn_1 is missing.

Assuming the availability of *query approximation*, rather than trying to correct their original query the user may pose instead the following query, Q_2, which allows the first conjunct to be automatically approximated by including the operator APPROX:

$$Y \leftarrow APPROX(\text{`FL56'}, fn_1, Y), (Y, pn_1^-.type, P_1)$$

The system will now incrementally apply edit operations to fn_1, comprising insertions, deletions, inversions etc. of edge labels, returning to the user answers at increasing 'cost' from their original query, for as long as the user wishes. Specifically, the sequence of edit operations that replaces fn_1 by fn_1^- and inserts ppn_1 after fn_1^- will return the relevant result '1234' (at a cost of 2α, say, if α is the cost of each of the edit operations made).

In an attempt to retrieve additional relevant answers, and assuming also the the availability of *query relaxation*, the user may now pose the following query, Q_3, which also allows the second conjunct to be automatically relaxed by including the operator RELAX:

$$Y \leftarrow APPROX(\text{`FL56'}, fn_1, Y), RELAX(Y, pn_1^-.type, P_1)$$

Using the information in Figure 2, the system will automatically relax property pn_1 to its superproperty pn and class P_1 to its superclass P (there is a 'subproperty' edge between pn_1 and pn, and a 'subclass' edge between P_1 and P), at a cost of 2β, say, assuming a cost β for each relaxation operation. An additional relevant result '6789' will now be returned to the user at an overall cost of $2(\alpha + \beta)$.

Contributions: Regular expressions have been used as a powerful mechanism for querying graph-structured data for over 20 years (e.g. [3]) and have been adopted by the semistructured data community (e.g. [4]) and more recently by the RDF community (e.g. [5, 6]) in SPARQL 1.1 [7]. Building on techniques from [8, 9], the work in [10] showed that approximate matching of CRPQs, using edit operations on edge labels, can be undertaken in polynomial time. The edit operations considered in that work were insertions, deletions, substitutions, transpositions and inversions of edge labels (corresponding to reverse traversal of edges). The work in [11] considered relaxation of conjunctive queries (but not CRPQs) over RDF data, and formalised relaxation using RDFS entailment. In [12], we extended relaxation to the more general case of CRPQs and allowed either approximation or relaxation to be applied to each CRPQ conjunct. The present paper makes several contributions. Firstly, we extend the work of [12] by considering the full range of edit operations on CRPQs and by giving full details of the algorithms and full proofs of the theoretical results. Secondly, in Section 4 we propose a new query operator, FLEX, that applies both approximation and relaxation to a query conjunct, thus providing greater flexibility to users in formulating their queries by not requiring them to select whether approximation or relaxation should be applied to a given query conjunct; as we will see in Section 4, there are answers returned by CRPQs using FLEX semantics which cannot be returned by any CRPQ using APPROX/RELAX semantics. Thirdly, to prepare for this integrated treatment of query approximation and relaxation, we introduce in Section 3.1 the notion of the 'triple form' of a sequence of edge labels, which is a departure from the direct application of edit operations to strings in [12] and which gives for the first time a uniform framework for handling both query approximation and query relaxation for CRPQs.

Outline of the paper: The rest of the paper is structured as follows. In Section 2 we give the necessary background, introducing our graph-based data model, which comprises a data graph and an ontology graph, and our query language, which supports conjunctive regular path queries (CRPQs). In Section 3 we give a formal definition of CRPQs, discuss exact matching of single-conjunct CRPQs, and give formal presentations of approximate matching and relaxation of such queries. We show how the approximate answer to a single-conjunct CRPQ can be computed in time that is polynomial in the size of the query and the data graph, with answers being returned in ranked order of their 'distance' from the original query. We also show how the relaxed answer to a single-conjunct CRPQ can be computed in time that is polynomial in the size of the query, the data graph and the ontology graph, again with answers being returned in ranked order. Finally, we discuss the evaluation of multi-conjunct CRPQs, each of whose conjuncts may be approximated or relaxed, and the complexity of query answering.

In Section 4 we discuss the application of both approximation and relaxation to an individual query conjunct using the new query operator, FLEX. We show how the evaluation of single-conjunct CRPQs that have FLEX applied to them can be undertaken using a combination of the techniques used for approximation and relaxation of single-conjunct queries. The evaluation is again accomplished in time that is polynomial in the size of the query, the data graph and the ontology graph, with answers being returned in ranked order. We also discuss the characteristics of multi-conjunct CRPQs in which conjuncts can have FLEX applied to them, considering query evaluation, complexity, and expressiveness.

In Section 5 we review related work in flexible query processing for semi-structured and graph-structured data. Section 6 summarises the contributions of the paper, and gives our concluding remarks and directions for further work.

2. Preliminaries

In this paper we consider a general graph-structured data model comprising a directed graph $G = (V_G, E_G, \Sigma)$ and a separate ontology $K = (V_K, E_K)$. The set V_G contains nodes each representing either an entity instance or an entity class, which we term entity nodes and class nodes, respectively. The set $E_G \subseteq V_G \times (\Sigma \cup \texttt{type}) \times V_G$ represents relationships between the members of V_G. If $e = (x, l, y) \in E_G$, then l is called the *label* of edge e. Node x is the

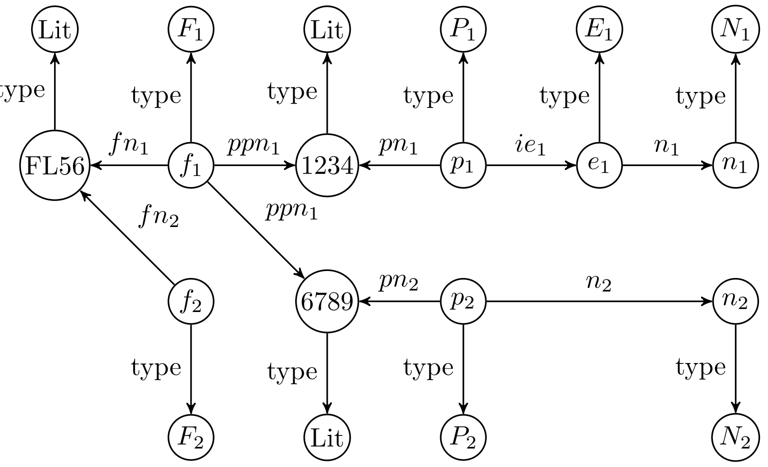

Figure 1: Example data graph G, where: F denotes *Flight*, P denotes *Person*, E denotes *Employee*, N denotes *NationalInsuranceNumber*, fn denotes *flightNumber*, ppn denotes *passengerPassportNumber*, pn denotes *passportNumber*, ie denotes *isEmployee* and n denotes *hasNationalInsuranceNumber*.

source of e, while y is its *target*. We assume that the *alphabet* Σ is finite and that $\texttt{type} \notin \Sigma$.

The set V_K contains nodes, each representing either an entity class or a property; in other words, V_K is the disjoint union of two subsets V_{Class} and V_{Prop}, the subset of class nodes and the subset of property nodes, respectively. The edges in E_K capture subclass relationships between class nodes, subproperty relationships between property nodes, and domain and range relationships between property nodes and class nodes. Hence, $E_K \subseteq V_K \times \{\texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}\} \times V_K$. We assume that $\Sigma \cap \{\texttt{type}, \texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}\} = V_{Prop} \cap \{\texttt{type}, \texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}\} = \emptyset$, and that of V_G representing classes are contained in V_{Class}.

Our graph model comprises a fragment of the RDFS vocabulary: `rdf:type`, `rdfs:subClassOf`, `rdfs:subPropertyOf`, `rdfs:domain`, `rdfs:range`, which we abbreviate in this paper by the symbols $\texttt{type}, \texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}$, respectively. The model does not allow for the representation of RDF's 'blank' nodes (which are indeed discouraged by some authors for linked data [13]) and we leave their consideration as future work.

Example 1. Figures 1 and 2 illustrate fragments of a data graph G and an ontology K, respectively, which we first presented in Section 1. These may have resulted from the integration of two heterogeneous datasets, from two flight insurance companies, under a common ontology. In Figure 2, `d` and `r` denote, respectively, `dom` and `range`. The subscripts on node and edge labels indicate the dataset from which each was derived (dataset 1 or dataset 2) We see from Figure 1 that there is some overlap in the datasets, through the literals 'FL56' and '6789'. The first of these literals is a flight number while the second is a

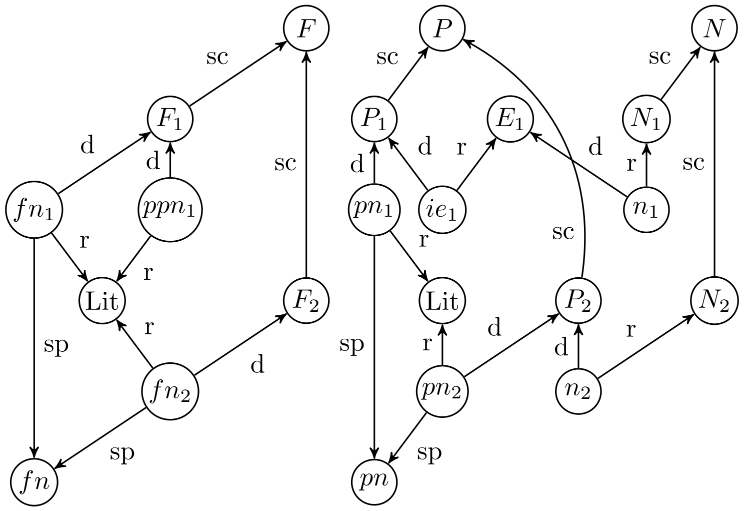

Figure 2: Example ontology K, where: d denotes `dom`, r denotes `range`, F denotes *Flight*, P denotes *Person*, E denotes *Employee*, N denotes *NationalInsuranceNumber*, *fn* denotes *flightNumber*, *ppn* denotes *passengerPassportNumber*, *pn* denotes *passportNumber*, *ie* denotes *isEmployee* and *n* denotes *hasNationalInsuranceNumber*.

passport number. □

Our query language is that of *conjunctive regular path queries* (CRPQs) [4]. A CRPQ over a graph G is of the form:

$$(Z_1, \ldots, Z_m) \leftarrow (X_1, R_1, Y_1), \ldots, (X_n, R_n, Y_n) \tag{1}$$

where each X_i and Y_i, $1 \le i \le n$, is a variable or constant, each Z_i, $1 \le i \le m$, is a variable appearing in the body of the query, and each R_i, $1 \le i \le n$, is a regular expression over the alphabet of edge labels.

Given a CRPQ Q and graph G, let θ be a mapping from $\{X_1, \ldots, X_n, Y_1, \ldots, Y_n\}$ to V_G such that (i) each constant is mapped to itself, and (ii) for each conjunct (X_i, R_i, Y_i), $1 \le i \le n$, there is a path from $\theta(X_i)$ to $\theta(Y_i)$ in G whose sequence of edge labels is in the language denoted by R_i. Let Θ be the set of such mappings. Then the (exact) answer of Q on G is $\{\theta(Z_1, \ldots, Z_m) \mid \theta \in \Theta\}$[8].

The work in [10] built on techniques from [8, 9] and [15] to show that approximate matching of CRPQs can be undertaken in polynomial time, subject to certain assumptions which we discuss in Section 3.7. The query edit operations considered were insertions, deletions and substitutions of edge labels, inversions of edge labels (corresponding to reverse traversals of graph edges), and transpositions of adjacent labels, each with an assumed edit cost. Query

[8]Proposed approaches for evaluating CRPQs include automaton-based methods, translation into Datalog or into recursive SQL, search-based processing, and reachability indexing — see [14] for an overview and citations of representative works.

results were returned incrementally to the user in order of their increasing edit distance from the original query.

Example 2. Referring again to Figures 1 and 2, a user familiar with the first dataset may pose the query Q_2 that was listed in the Motivating Example earlier, in an attempt to find passport numbers relating to flight number 'FL56', making use of the query approximation techniques proposed in [10]:

$$Y \leftarrow APPROX(\text{'FL56'}, fn_1, Y), (Y, pn_1^-.type, P_1)$$

The Motivating Example describes how the result '1234' can be returned by this query. □

The work in [12] proposed combining approximation and ontology-based relaxation for CRPQs. Either approximation or relaxation could be applied to each conjunct of a CRPQ, although edge inversions and transpositions were not considered within the set of edit operations.

Example 3. Continuing with the query in the previous example, the user may pose the query Q_3 that was listed in the Motivating Example earlier, which also allows the second conjunct to be relaxed:

$$Y \leftarrow APPROX(\text{'FL56'}, fn_1, Y), RELAX(Y, pn_1^-.type, P_1)$$

The Motivating Example describes how the additional result '6789' can be returned by this query. □

3. Approximation and Relaxation of CRPQs

In this section we first discuss exact matching of single-conjunct CRPQs, followed in Section 3.2 by approximate matching as performed by the APPROX operator. In Sections 3.4 and 3.5 we discuss relaxation of single-conjunct CRPQs based on information from an ontology, as well as how answers for a conjunct to which the RELAX operator has been applied can be computed. We also describe how answers can be returned to the user incrementally in Sections 3.3 and 3.6. Finally, in Section 3.7, we discuss the evaluation of multi-conjunct CRPQs, each of whose conjuncts may have APPROX or RELAX applied to them.

3.1. Single-Conjunct Queries

Let $G = (V_G, E_G, \Sigma)$ be a graph as defined in Section 2. We will allow each edge $e = (x, l, y) \in E_G$ to be traversed both from its source x to its target y and from its target y to its source x. In order to specify the traversal from target to source, it is useful to define the *inverse* of an edge label l, denoted by l^-. Let $\Sigma^- = \{l^- \mid l \in \Sigma\}$. If $l \in \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$, we use l^- to mean the *inverse* of l, that is, if l is a for some $a \in \Sigma \cup \{\texttt{type}\}$, then l^- is a^-, while if l is a^- for some $a \in \Sigma \cup \{\texttt{type}\}$, then l^- is a.

A *single-conjunct regular path query* Q over a graph $G = (V_G, E_G, \Sigma)$ has the form:

$$vars \leftarrow (X, R, Y) \tag{2}$$

where X and Y are constants or variables, R is a regular expression over Σ $\cup\Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}$, and *vars* is the subset of $\{X, Y\}$ that are variables. If X or Y is a constant, then that constant must appear in V_G if Q is to return a non-empty answer on G (see Definition 1 below).

A *regular expression* R over $\Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}$, 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 sequence, a is any label in $\Sigma \cup \{\mathtt{type}\}$, "_" denotes the disjunction of all constants in $\Sigma \cup \{\mathtt{type}\}$, and the operators have their usual meaning.

A *weighted* non-deterministic finite state automaton (NFA), M_R, of size $O(R)$ can be constructed to recognise the language denoted by R, $L(R)$, using Thompson's construction (which makes use of ϵ-transitions) [16]. Each transition of M_R is labelled with a label from $\Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}$, and has a weight, or cost, which is zero. We use weighted automata in order to model the costs associated with approximation and relaxation. The full definition of M_R is given below as Definition 8.

Definition 1. A *semipath* [4] p in $G = (V_G, E_G, \Sigma)$ from $x \in V_G$ to $y \in V_G$ is a sequence $(v_1, l_1, v_2, l_2, v_3, \ldots, v_n, l_n, v_{n+1})$, where $n \geq 0$, $v_1 = x$, $v_{n+1} = y$ and for each v_i, l_i, v_{i+1} either $(v_i, l_i, v_{i+1}) \in E_G$ or $(v_{i+1}, l_i^-, v_i) \in E_G$. A semipath p *conforms* to a regular expression R if $l_1 \cdots l_n \in L(R)$.

Given a single-conjunct regular path query Q and graph G, let θ be a mapping from $\{X, Y\}$ to V_G that maps each constant to itself. We term a mapping such as θ a *(Q, G)-matching.* A tuple $\theta(vars)$ *satisfies* Q on G if there is a semipath in G from $\theta(X)$ to $\theta(Y)$ which conforms to R. The *exact answer* of Q on G is the set of tuples which satisfy Q on G. $\square$

The following result on the complexity of exact query answering follows from Lemma 1 in [17].

Proposition 1. *Given single-conjunct regular path query Q and graph G, the exact answer of Q on G can be found in time which is polynomial in the size of Q and G.*

We now define the *triple form* of both a semipath in a graph and a sequence of labels in $L(R)$. The definition uses the notion of a *triple pattern*, which is a triple each of whose components may be either a constant or a variable. We use triple forms as a uniform syntax to which we can apply approximation and relaxation.

Definition 2. Let p be a semipath $(v_1, l_1, v_2, l_2, v_3, \ldots, v_n, l_n, v_{n+1})$, $n \geq 1$, in G. A *triple form* of p is a sequence of triple patterns

$$(v_1, l_1, W_1), (W_1, l_2, W_2), \ldots, (W_{n-1}, l_n, v_{n+1})$$

where $W_1, \ldots, W_{n-1}$ are distinct variables. If p is of length zero, then p is of the form (v, ϵ, v) and the only triple form of p is (v, ϵ, v). $\square$

Definition 3. Given a query Q with single conjunct (X, R, Y), let $q = l_1 l_2 \cdots l_n$, $n \geq 1$, be a sequence of labels in $L(R)$. A *triple form* of (Q, q) is a sequence of triple patterns

$$(X, l_1, W_1), (W_1, l_2, W_2), \ldots, (W_{n-1}, l_n, Y)$$

where $W_1, \ldots, W_{n-1}$ are distinct variables not appearing in Q. If $q = \epsilon$, then the triple form of (Q, q) is (X, ϵ, Y). $\square$

Definition 4. Let T be a sequence of triple patterns

$$(W_0, l_1, W_1), (W_1, l_2, W_2), \ldots, (W_{n-1}, l_n, W_n)$$

such that $n \geq 1$, $W_1, \ldots, W_{n-1}$ are variables, and W_0 and W_n are variables or constants. Any triple pattern (W_{i-1}, l_i, W_i) in which $l_i \in \Sigma^- \cup \{\texttt{type}^-\}$ is said to be *inverted*; otherwise the triple pattern is *non-inverted*. The *normalised form* of an inverted triple pattern (W_{i-1}, l_i, W_i) is (W_i, l_i^-, W_{i-1}), while the normalised form of a non-inverted triple pattern is the triple pattern itself. The *normalised form* of T comprises the normalised form of each triple pattern in T. $\square$

Example 4. Assume we have G as shown in Figure 1, query Q comprising the single conjunct $(\text{‘FL56’}, fn_1^- \cdot ppn_1, X)$, where ‘FL56’ is a constant and X is a variable, and $q = fn_1^- \cdot ppn_1$. The triple form of (Q, q) is

$$(\text{‘FL56’}, fn_1^-, W_1), (W_1, ppn_1, X)$$

for the normalised triple form is:

$$(W_1, fn_1, \text{‘FL56’}), (W_1, ppn_1, X)$$

$\square$

3.2. Approximate Matching of Single-Conjunct Queries

Approximate matching of a single-conjunct query Q against a graph G is achieved by applying edit operations to the sequence of labels in $L(R)$, where R is the regular expression used in Q. Let q be a sequence of labels in $L(R)$ and l be an arbitrary label in $\Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$.

An *edit* operation on q is one of the following:

(i) the *insertion* of label l into q,

(ii) the *deletion* of label l from q,

(iii) the *substitution* of some label other than l by l in q,

Each edit operation has a cost, which is a positive integer; the costs may be different for different edit operations. We assume throughout that the cost of substitution is less than the combined cost of insertion and deletion (otherwise the substitution operation would be redundant, as it would be no more costly to achieve such an edit through an insertion and a deletion operation).

In [10], we defined two additional edit operations: (i) the *inversion* of a label in q and (ii) the *transposition* of a pair of adjacent labels in q. We note that both these operations can be subsumed semantically by the edit operations given above; inversion is subsumed by substitution in which some label $l \in \Sigma$ is replaced by l^-, and transposition is achieved by either applying substitution to the pair of labels to be transposed or by a combination of insertion and deletion of labels.

Definition 5. The *application of an edit operation to a sequence of triple patterns* T of the form

$$(X, l_1, W_1), (W_1, l_2, W_2), \ldots, (W_{n-1}, l_n, Y),$$

where $n \geq 1$, X and Y are variables or constants, and $W_1, \ldots, W_{n-1}$ are distinct new variables, is defined as follows:

The result of a substitution on T is

$$(X, l'_1, W_1), (W_1, l'_2, W_2), \ldots, (W_{n-1}, l'_n, Y)$$

such that there must be some $1 \leq j \leq n$ such that $l_j \neq l'_j$ (l'_j has been substituted for l_j) and $l_i = l'_i$, for each $i \neq j$, $1 \leq i \leq n$.

The result of an insertion into T is

$$(X, l'_1, W_1), (W_1, l'_2, W_2), \ldots, (W_n, l'_{n+1}, Y)$$

such that there is a $1 \leq j \leq n+1$ for which $l_i = l'_i$, for each $1 \leq i \leq j-1$, and $l_i = l'_{i+1}$, for each $j+1 \leq i \leq n$ (l'_j is the inserted label).

If $n > 1$, the result of a deletion from T is

$$(X, l'_1, W_1), (W_1, l'_2, W_2), \ldots, (W_{n-2}, l'_{n-1}, Y)$$

such that there is a $1 \leq j \leq n$ for which $l_i = l'_i$, for each $1 \leq i \leq j-1$, and $l_{i+1} = l'_i$, for each $j \leq i \leq n-1$ (l_j is the deleted label). If $n = 1$, the result of a deletion from T is (X, ϵ, Y) (where l_1 is the deleted label).

If T is of the form (X, ϵ, Y), then only the insertion operation applies, the result of which is (X, l', Y), where l' is the inserted label. □

Definition 6. Given graph G, semipath p in G, query Q with single conjunct (X, R, Y), (Q, G)-matching θ, sequence of labels $q \in L(R)$, triple form T_q for $(\theta(Q), q)$, and triple form T_p for p:

- we write $T_q \preceq_A T_p$, if T_q can be transformed to T_p (up to variable renaming) by a sequence of edit operations. The *cost* of the sequence of edit operations on T_q is the sum of the costs of each operation.
- the *approximation distance* from p to $(\theta(Q), q)$ is the minimum cost of any sequence of edit operations which yields T_p from T_q. The cost of the empty sequence of edit operations (so T_q is already a triple form of p) is zero. If T_q cannot be transformed to T_p, then the approximation distance is infinity.
- the *approximation distance* from p to $\theta(Q)$ is the minimum approximation distance from p to $(\theta(Q), q)$ for any sequence of labels $q \in L(R)$.
- the *approximation distance* of $\theta(Q)$, denoted $adist(\theta, Q)$, is the minimum approximation distance to $\theta(Q)$ from any semipath p in G.
- the *approximate answer* of Q on G, denoted $Q_A(G)$, is a list of pairs $(\theta(vars), adist(\theta, Q))$, where θ is a (Q, G)-matching, ranked in order of non-decreasing approximation distance.
- the *approximate top-k answer* of Q on G is a list containing the first k tuples in $Q_A(G)$. □

We now describe how $Q_A(G)$ can be computed in time polynomial in the size of Q and G. A similar process was described in [10], but that paper included only sketch proofs of the theoretical results. Broadly, the steps are as follows: (1) construct a *query automaton* M_Q, (2) construct an *approximate automaton* A_Q, (3) construct the *product automaton* H of A_Q and G, and (4) perform shortest path traversals of H in order to find the approximate answer of Q on G. Each of the terms introduced above are defined next.

Definition 7. A *weighted non-deterministic finite state automaton* (NFA) M is a tuple $(S, A, \delta, S_0, S_f, \xi)$, where: S is a set of states; A is an alphabet of labels; $\delta \subseteq S \times A \times \mathbb{N} \times S$ is the transition relation; $S_0 \subseteq S$ is the set of start states; $S_f \subseteq S$ is the set of final states; and ξ is a final weight function mapping each state in S_f to a non-negative integer [18]. Given a transition $(s, a, c, t) \in \delta$, we sometimes say that the transition is *from s to t* and call a the label and c the cost of the transition.

We call a sequence of transitions from an initial to a final state of M a *run*. Given a sequence of labels p, a *run for p* is a sequence of transitions of the form $(s_1, a_1, w_1, s_2), \ldots, (s_{n-1}, a_{n-1}, w_{n-1}, s_n)$, where s_1 is an initial state, s_n is a final state, and $p = a_1 \cdots a_{n-1}$. The *cost* of the run is $w_1 + \cdots + w_{n-1} + \xi(s_n)$. We sometimes say that the run is *from s_1 to s_n*. □

Definition 8. Let R be a regular expression defined over alphabet $\Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$. A weighted NFA M_R recognising $L(R)$ can be constructed in the same way as a normal NFA recognising $L(R)$, except that each transition and each final state has a zero weight associated with it. Formally, $M_R =$

$(S, \Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}, \delta, \{s_0\}, \{s_f\}, \xi)$, where there is only one initial state s_0 and one final state s_f, and ξ maps s_f to zero.

Let Q be a single-conjunct query with conjunct (X, R, Y). The *query automaton* M_Q for Q is the same as M_R but with annotations on the initial and final states. In particular, if X (or, respectively, Y) in Q is a constant c, then s_0 (s_f) is annotated with c; otherwise s_0 (s_f) is annotated with the wildcard symbol $*$ which matches any constant. $\square$

Definition 9. Let Q be a single-conjunct query with conjunct (X, R, Y), and $M_R = (S, \Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}, \delta, \{s_0\}, \{s_f\}, \xi)$ be the weighted NFA for R. We construct the *approximate automaton* A_Q *for* Q by first constructing an *approximate automaton* A_R *from* M_R. The approximate automaton is constructed in a number of steps:

- First the automaton A_R^1 with *deletions* is constructed from M_R. Automaton A_R^1 is the same as M_R except that the set of transitions δ' includes all those in δ along with the set $\{(s, \epsilon, c_d, t) \mid (s, a, 0, t) \in \delta \wedge s \neq t\}$, where c_d is the cost of deletion.

- Next an automaton A_R^2 without ϵ-transitions is constructed from A_R^1, using the method of [18]. Briefly, the method first computes the *ϵ-closure* of A_R^1, which is the set of pairs of states connected by a sequence of ϵ-transitions along with the minimum summed weight for each such pair. Then $A_R^2 = (S, \Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}, \delta'', \{s_0\}, S, \xi')$, where the transitions in δ'' comprise the non-epsilon transitions in δ' along with each transition (s, b, w, u) such that pair (s, t) with weight w is in the ϵ-closure and transition $(t, b, 0, u) \in \delta'$ $(b \neq \epsilon)$. Because every original (non-looping) transition in δ' has an associated ϵ-transition, all states will be final. The final state function ξ' is defined as follows. For final state s_f, $\xi'(s_f) = \xi(s_f)$. For each state $s \neq s_f$, $\xi'(s)$ is the minimum weight of the pairs (s, s_f) in the ϵ-closure.

- Thirdly, an automaton A_R^3 with *substitutions* is constructed from A_R^2. Automaton A_R^3 is the same as A_R^2 except that the transitions of A_R^3 comprise those of A_R^2 along with transitions of the form $(s, b, w + c_s, t)$, where c_s is the cost of *substitution*, for each transition $(s, a, w, t) \in \delta''$ and label $b \in \Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}$ $(b \neq a)$.

- Finally, the approximate automaton A_R is constructed from A_R^3 by including *insertions*. Automaton A_R is the same as A_R^3 except that the transitions of A_R comprise those of A_R^3 along with transitions of the form (s, a, c_i, s), where c_i is the cost of insertion, for each state $s \in S$ and label $a \in \Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}$.

The *approximate automaton* A_Q *for* Q is formed from A_R by annotating the initial and final states in A_R with the annotations from the initial and final states, respectively, in M_Q. $\square$

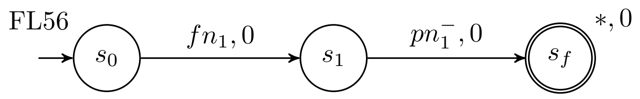

Figure 3: Query automaton M_Q for conjunct ('FL56', $fn_1 \cdot pn_1^-, X$).

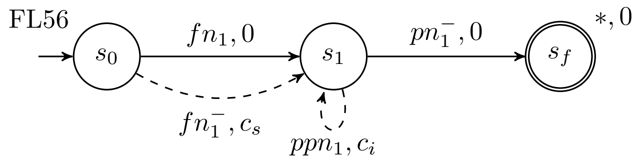

Figure 4: Fragment of approximate automaton A_Q for conjunct ('FL56', $fn_1 \cdot pn_1^-, X$).

Example 5. Consider the graph G shown in Figure 1, and the following approximated query Q:

$$X \leftarrow APPROX(\text{'FL56'}, fn_1 \cdot pn_1^-, X)$$

The query automaton M_Q for Q is shown in Figure 3. We see that M_Q is comprised of two transitions, each labelled with a cost of zero; the initial state, s_0, is annotated with the constant 'FL56'; and the final state, s_f, is annotated with the wildcard symbol $*$, and has a weight of zero.

A fragment of the approximate automaton A_Q for Q is shown in Figure 4. Two transitions — represented by the dashed lines — appear in A_Q but not in M_Q. The transition (s_0, fn_1^-, c_s, s_1) indicates that the label fn_1 has been substituted by fn_1^-, and the transition (s_1, ppn_1, c_i, s_1) indicates that the label ppn_1 has been inserted before pn_1^-; c_s and c_i denote the cost of these edit operations, respectively. □

We next show that using automaton A_R is sufficient to find all sequences of labels generated by edit operations at an approximation distance k from a given query. To do so, we make use of the concept of a *trace* of edit operations, introduced in [19]. A trace is essentially a sequence of edit operations in which order is unimportant and redundancy is not present. More specifically, edit operations are applied only to labels in the original sequence, and redundant operations (such as the insertion of some previously-deleted label) are not performed. Therefore, without loss of generality, we can assume that in our edit sequences, all deletions come first, followed by all substitutions which are then followed by all insertions.

Lemma 1. *Let Q be a single-conjunct CRPQ with conjunct (X, R, Y). Let A_R be the automaton constructed for R as described in Definition 9 above, q be a sequence of labels in $L(R)$, and p be a sequence of labels in $(\Sigma \cup \Sigma^- \cup \{\texttt{type} \cup \texttt{type}^-\})^*$ corresponding to a semipath in G. The approximation distance from p to q is equal to the minimum cost of a run for p in A_R.*

Proof. Given graph G and (Q, G)-matching θ, let T_q be the triple form of $(\theta(Q), q)$, and T_p be a triple form for p. The approximation distance from p to q is defined as the minimum cost of any sequence of edit operations which yields T_p from T_q. The proof proceeds by induction on the number of edit operations used in any such minimum-cost edit sequence.

Basis: If no edit operations are used, then, by definition, the cost of the edit sequence is 0, and T_q is already a triple form of p. Thus, there is a run of cost 0 in A_R. Clearly, this run is of minimum cost.

Induction: For the inductive step, assume that there is an $n \geq 0$ such that, for all $m \leq n$, if m edit operations are used in a minimum-cost edit sequence of cost c which yields T_p from T_q, then the minimum cost of a run for p in A_R is c.

Now consider a sequence of labels p which requires $n + 1$ edit operations in a minimum-cost edit sequence to produce T_p from T_q. Let this edit sequence be given by $S = P_0 \preceq_A P_1 \preceq_A \cdots \preceq_A P_n \preceq_A P_{n+1}$, where $P_0 = T_q$ and $P_{n+1} = T_p$. Let the cost of a deletion, substitution and insertion be denoted by c_d, c_s and c_i, respectively. The cost of the sequence S is $k = n_d c_d + n_s c_s + n_i c_i$, where n_d, n_s and n_i are the number of deletions, substitutions and insertions, respectively, used in S for some $n_d \geq 0$, $n_s \geq 0$ and $n_i \geq 0$, and $n_d + n_s + n_i = n + 1$.

Using a result from [19], we can assume that in edit sequence S all deletions appear first, followed by all substitutions, and then all insertions. Let op_E denote the edit operation applied to P_n to yield $P_{n+1} = T_p$. The proof proceeds by considering the possible alternatives for op_E:

(1) *op_E is a deletion*: There are two cases to consider, depending on whether or not the deleted triple pattern is the last in P_n.

(i) We first consider the case in which the deleted triple pattern is not the last in P_n. So assume that op_E deletes the triple pattern (W_{m-1}, b, W_m) in P_n to produce $P_{n+1} = T_p$, transforming the pair of triple patterns $t'_f = (W_{m-1}, b, W_m), (W_m, g, W_{m+1})$ in P_n into $t_f = (W_{m-1}, g, W_m)$ in P_{n+1}. Let the subsequence of sequence S up to P_n be denoted by S'. By definition, sequence S' uses n edit operations and has cost $k - c_d$. By the inductive hypothesis, there is a minimum cost run r of cost $k - c_d$ in A_R for the sequence corresponding to triple form P_n. Suppose that in run r the subsequence t'_f in P_n is matched by the transitions (s_1, b, d_1, s_2) and (s_2, g, d_2, s_3) in A_R. We will show that there is a minimum cost run of cost k in A_R which matches T_p.

Suppose that, prior to the removal of ϵ-transitions, t'_f corresponds to the sequence of transitions in A_R given by $s' = \Gamma, (s_4, b, 0, s_2), \Delta, (s_5, g, 0, s_3)$, where Γ and Δ represent sequences of ϵ-transitions. Sequence Γ represents the presence of x_1 ϵ-transitions which include y_1 ϵ-transitions each of cost c_d, representing the deletion of labels in T_q prior to b in some triple form prior to P_n in S, where $x_1 \geq 0$, $y_1 \leq x_1$, and $s_1 = s_4$ if $x_1 = 0$. Sequence Δ represents the presence of x_2 ϵ-transitions which include y_2 ϵ-transitions each of cost c_d, representing the deletion of labels from T_q between b and g in some triple form prior to P_n in S, where $x_2 \geq 0$, $y_2 \leq x_2$, and $s_2 = s_5$ if $x_2 = 0$.

After the removal of ϵ-transitions, s' is transformed to $t' = (s_1, b, d_1, s_2), (s_2, g, d_2, s_3)$ in run r, where $d_1 = y_1 c_d$ and $d_2 = y_2 c_d$, as shown in Figure 5. Thus, the cost of run r is $k - c_d = d_1 + d_2 + e$, where e is the cost of the remaining

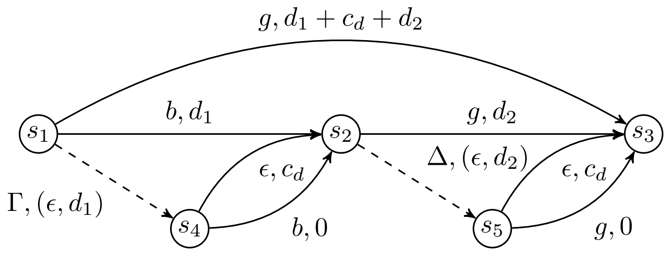

Figure 5: Automaton for the deletion of the label b in T_{p_k} (b is not the last label).

transitions in r. By construction, the transition $(s_4, \epsilon, c_d, s_2)$ representing the deletion of b is present in A_R, as shown in Figure 5.

There is a sequence of ϵ-transitions from s_1 to s_4 of cost d_1. Along with the ϵ-transition from s_4 to s_2, this means there is a sequence of ϵ-transitions from s_1 to s_2 of cost $d_1 + c_d$.

There is a sequence of ϵ-transitions from s_2 to s_5 of cost d_2. Therefore there is a path of ϵ-transitions from s_1 via s_4 and s_2 to s_5 of cost $d_1 + c_d + d_2$. As there is a transition of cost 0 labelled with g from s_5 to s_3, a transition labelled with g from s_1 to s_3 of cost $d_1 + c_d + d_2$ would have been added to A_R during the removal of ϵ-transitions (see Figure 5). Therefore, there is a minimum cost run of cost k in A_R which matches T_p.

(ii) We now consider the case where the deleted triple pattern *is* the last in P_n. Assume that op_E deletes the triple pattern (W_m, g, W_{m+1}) in P_n, transforming triple form $t'_f = (W_{m-1}, b, W_m), (W_m, g, W_{m+1})$ in P_n into $t_f = (W_{m-1}, b, W_m)$ in $P_{n+1} = T_p$. As in case (i), we know by the inductive hypothesis that there is a minimum cost run r of cost $k - c_d$ in A_R for the sequence corresponding to triple form P_n. Suppose that in run r the subsequence t'_f in P_n is matched by the transitions (s_1, b, d_1, s_2) and (s_2, g, d_2, s_3) in A_R. We will show that there is a minimum cost run of cost k in A_R which matches T_p.

Suppose that prior to the removal of ϵ-transitions, t'_f is matched in A_R by the partial sequence of transitions given by s', as defined previously. After the removal of ϵ-transitions, s' is transformed to t', also as given previously. Using the same reasoning as in the case for when the deleted label is not the last in P_n, we note that $d_2 = y_2 c_d$, representing the deletion of y_2 ϵ-transitions, succeeding b and preceding g in T_q, in some sequence prior to P_n.

If there were m ϵ-transitions succeeding g in T_q, these would all have been deleted prior to P_n in sequence S'. As g is the last label, then, from the inductive hypothesis, we have that $\xi(s_3) = d_3$, where $d_3 = mc_d$. Thus, the cost of run r is $k - c_d = d_1 + d_2 + d_3 + e$ where e is the cost of the remaining transitions in r.

By construction, the transition $(s_5, \epsilon, c_d, s_3)$ representing the deletion of g is present in A_R. There is a sequence of ϵ-transitions from s_2 to s_5 of cost d_2, and, along with the ϵ-transition from s_5 to s_3 of cost c_d, this means there is a sequence of ϵ-transitions from s_2 to s_3 via s_5 of cost $d_2 + c_d$. Thus $\xi(s_2)$ set to $d_2 + c_d + d_3$. Thus, there is a minimum cost run of cost k which matches T_p.

(2) *op_E is a substitution*: Suppose op_E replaces some triple $t' = (W_{m-1}, a, W_m)$ in P_n by $t = (W_{m-1}, b, W_m)$ in $P_{n+1} = T_p$. Hence the subsequence of sequence S up to P_n uses n edit operations and has cost $k - c_s$. By the inductive hypothesis, there is a minimum cost run r of cost $k - c_s$ in A_R for the sequence corresponding to triple form P_n.

Suppose that in run r triple t' is matched by the transition $f' = (s_1, a, d, s_2)$. Then, by construction, A_R has a transition $f = (s_1, b, d + c_s, s_2)$ matching t. Substituting f' in r by f yields a run r' in A_R with cost k which matches T_p.

(3) *op_E is an insertion*: Suppose op_E inserts a triple pattern $t = (W_{m-1}, a, W_m)$ after another triple pattern $t' = (W_{m-2}, h, W_{m-1})$ in P_n to produce $P_{n+1} = T_p$. Hence the subsequence of sequence S up to P_n uses n edit operations and has cost $k - c_i$. By the inductive hypothesis, there is a minimum cost run r of cost $k - c_i$ in A_R for the sequence corresponding to triple form P_n.

Assume that in run r triple t' is matched by the transition $f' = (s_1, h, d, s_2)$, where $d \geq 0$. By construction there is a transition (s_2, a, c_i, s_2) in A_R, and hence a sequence $(s_1, h, d, s_2), (s_2, a, c_i, s_2)$ matching t' and t. Using this sequence instead of (s_1, h, d, s_2) in r yields a run of cost k in A_R. The case for inserting a triple pattern *before* another triple pattern is analogous. □

Definition 10. Let $A_Q = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, \{s_0\}, S, \xi)$ be an approximate automaton and $G = (V_G, E_G, \Sigma)$ a graph. We can view G as an automaton with set of states V_G, alphabet Σ, set of initial states V_G, and set of final states V_G. There is a transition from state s to state t labelled a in the automaton if and only if there is an edge $(s, a, t) \in E_G$. We can then form the *product automaton*, H, of A_Q and G. Formally, H is the weighted automaton $(T, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \sigma, I, T, \xi)$, where $I \subseteq T$ is a set of initial states and all states in T are final. The set of states T is given by $\{(s, n) \mid s \in S \wedge n \in V_G\}$. The set of transitions σ consists of transitions of the form

- $((s, n), a, c, (s', n'))$ if $(s, a, c, s') \in \delta$ and $(n, a, n') \in E_G$,
- $((s, n), a^-, c, (s', n'))$ if $(s, a^-, c, s') \in \delta$ and $(n', a, n) \in E_G$.

The set of initial states I is given by $\{(s_0, n) \mid n) \in V_G\}$. We overload the use of ξ as the final weight function, carrying over the weights from final states in A_Q to those in H. The annotations on initial and final states in H are also carried over from the corresponding initial and final states in A_Q. □

H can be viewed either as an automaton or as a graph, whichever is appropriate in a given context. When H is viewed as an automaton, we will use the terms *states*, *transitions* and *runs*; when viewed as a graph, we will use the terms *nodes*, *edges* and *paths*.

We can now define how $Q_A(G)$, the approximate answer of Q on G, can be computed:

(i) We construct the weighted NFA M_R from R, using Thompson's construction [16], and then the query automaton M_Q from M_R.

(ii) We construct the approximate automaton A_Q from M_Q.

(iii) We form the product automaton, H, of A_Q with G.

(iv) Let the conjunct of Q be (X, R, Y). Assume first that X is a constant u. Assume also that $u \in V_G$, for otherwise $Q_A(G)$ is empty. We perform a shortest path traversal of H starting from the initial state (s_0, u), incrementing the total cost of the path by the cost of the transition. Whenever we reach a final state (s_f, v) in H we output v, provided v *matches* the annotation on (s_f, v), along with the cost of the path. Recall that if Y is a constant the annotation on s_f will be that constant, and if Y is a variable the annotation will be the wildcard symbol $*$. Node v matches the annotation if and only if the annotation is v or $*$. Now assume X is a variable. In this case, we perform a shortest path traversal of H, outputting nodes as above, starting from state (s_0, u) for each node $u \in V_G$.

The above construction of an approximate automaton differs from that given in [10] where the NFA for approximate matching of regular expression R was constructed using a number of copies of the NFA for recognising R, each corresponding to matching at a different distance. Hence, in that NFA, distance was represented implicitly by the 'copy number' of states, rather than explicitly using a weight as above.

The next two lemmas show firstly the correctness of the traversal of the product automaton, H; and secondly that the approximation distance from a semipath in a graph G to the matchings for a single-conjunct query Q is equal to the minimum cost of a corresponding run in H.

Lemma 2. *There is a run in H from (s_0, v_0) to (s_f, v_n) of cost k if and only if there is a semipath from v_0 to v_n in G and a run of cost k from s_0 to s_f in A_Q, for some initial state s_0 and some final state s_f in A_Q.*

Proof. There are two types of transition in H, as given in the definition of its construction. The first type of transition, $((s, n), a, c, (s', n'))$, is in H if and only if there is an edge labelled a from n to n' in G and a transition labelled a from s to s' with a cost of c in A_Q. The second type of transition, $((s, n), a^-, c, (s', n'))$, is added to H if and only if there is an edge labelled a from n' to n in G and a transition labelled a^- from s to s' with a cost of c in A_Q.

Given the above, it is straightforward to show (by induction, for example) that there is a sequence of transitions of the form $((s_0, v_0), a_1, c_1, (s_1, v_1)), \ldots,$ $((s_{j-1}, v_{j-1}),\ a_{j-1}, c_{j-1}, (s_f, v_j))$ in H of cost $k = c_1 + \cdots + c_{j-1} + \xi[s_f]$ if and only if there is a semipath from v_0 to v_j in G and a sequence of transitions of the form $(s_0, a_1, c_1, s_1), \ldots,$ $(s_{j-1}, a_{j-1}, c_{j-1}, s_f)$ of cost k in A_Q. □

Lemma 3. *Let θ be a matching from a query Q with single conjunct (X, R, Y) to a graph $G = (V_G, E_G, \Sigma)$, where $\theta(X) = v_0$ and $\theta(Y) = v_n$ for some $v_0, v_n \in V_G$. Let p be a semipath from v_0 to v_n in G, and H be the product automaton of A_Q and G. The approximation distance from p to $\theta(Q)$ is k if and only if k is the minimum cost of a run for the sequence of labels comprising p from (s_0, v_0) to (s_f, v_n) in H, for some initial state s_0 and some final state s_f in A_Q.*

PROOF. ($\Rightarrow$) We know, by definition, that if the approximation distance from p to $\theta(Q)$ is k, then k is the *minimum* approximation distance from p to $(\theta(Q), q)$ for any sequence of labels $q \in L(R)$. For any such $q \in L(R)$, we also know, by definition, that if the approximation distance from p to $(\theta(Q), q)$ is k, then k is the minimum cost of any sequence of edit operations which yields triple form T_p from triple form T_q.

From Lemma 1, we know that if p has approximation distance k from q, the minimum cost of a run from s_0 to s_f for p in A_R, and hence A_Q, is k. From Lemma 2, we know that there is a run of cost k for p from (s_0, v_0) to (s_f, v_n) in H. There can be no run of cost less than k for p in H since this would contradict the fact that p is of approximation distance k from $\theta(Q)$.

($\Leftarrow$) Suppose that the minimum cost of a run for the sequence of labels comprising p from (s_0, v_0) to (s_f, v_n) in H, for some initial state s_0 and some final state s_f in A_Q, is k. This means that, by Lemma 2, A_Q has a minimum cost of k for a run for p. Hence, the minimum cost of edit operations needed to obtain T_p from T_q, for *any* $q \in L(R)$, must be k. Therefore the approximation distance from p to $\theta(Q)$ is k. $\square$

We next consider the complexity of approximate matching. For all the complexity-related proofs in this paper, we make the assumption that any occurrence of "_" in a regular expression — denoting the disjunction of all constants in $\Sigma \cup \{\texttt{type}\}$ — has been rewritten to $a_1|a_2|...|a_n|\texttt{type}$ where $\Sigma = \{a_1, \dots, a_n\}$.

Lemma 4. *A_Q has at most $2|R|$ states and $4|R|^2|\Sigma|$ transitions, and can be constructed in $O(|R|^3|\Sigma|)$ time.*

PROOF. From [16], we have that M_R contains at most $2|R|$ states and $4|R|$ transitions. Deletions add at most $|R|$ more transitions, giving $5|R|$ transitions in total. The subsequent removal of ϵ-transitions may result in A_Q having at most $25|R|^2$ transitions. Insertions add at most $2|R||\Sigma|$ transitions, and substitutions add at most $25|R|^2|\Sigma|$ transitions. However, since a directed, labelled multi-graph with $2|R|$ nodes and $|\Sigma|$ distinct labels can have at most $4|R|^2|\Sigma|$ edges, this is also the bound for the number of transitions in A_Q. From [18], we have that the construction of A_Q can be performed in $O(|R|^3|\Sigma|)$ time. $\square$

Proposition 2. *Let $G = (V_G, E_G, \Sigma)$ be a graph and Q be a single-conjunct query using regular expression R over alphabet $\Sigma \cup \{\texttt{type}\}$. The approximate answer of Q on G can be found in time $O(|R|^2|V_G|(|R||\Sigma||E_G| + |V_G| \log(|R||V_G|)))$.*

PROOF. Let A_Q be the approximate automaton constructed from R, and H be the product graph constructed from A_Q and G. Lemma 3 shows that traversing H correctly yields all approximate answers of Q. Lemma 4 tells us that A_Q has at most $2|R|$ states and $4|R|^2|\Sigma|$ transitions. Therefore H has at most $2|R||V_G|$ nodes and $4|R|^2|E_G||\Sigma|$ edges. If we assume that H is sparse (which is highly likely), then running Dijkstra's algorithm on each node of a graph with node set N and edge set A can be done in time $O(|N||A| + |N|^2 \log |N|)$. So, for graph

H, the combined time complexity is $O(|R|^3|V_G||E_G||\Sigma|+|R|^2|V_G|^2\log(|R||V_G|))$ which is equal to $O(|R|^2|V_G|(|R||\Sigma||E_G|+|V_G|\log(|R||V_G|)))$. □

As a corollary, it is easy to see that the data complexity is $O(|V_G||\Sigma||E_G|+|V_G|^2\log(|V_G|))$ and the query complexity is $O(|R|^3)$. The space complexity is dominated by the space requirements of H given in the proof above.

3.3. Incremental Evaluation

The above evaluation can also be accomplished "on-demand" by incrementally constructing the edges of H as required, thus avoiding precomputation and materialisation of the entire graph H. This is performed by calling a function `Succ` with a node (s,n) of H. The function returns a set of transitions $\xrightarrow{d}(p,m)$, such that there is an edge in H from (s,n) to (p,m) with cost d.

We list function `Succ` below, where the function `nextStates`(A_Q,s,a) returns the set of states in A_Q that can be reached from state s on reading input a, along with the cost of reaching each.

```
Function Succ($s,n,A_Q,G$)
    Input: state $s$ of $A_Q$ and node $n$ of $G$
    Output: set of transitions which are successors of $(s,n)$ in $H$
(1) $W \leftarrow \emptyset$
(2) for $(n,a,m)\in E_G$ and $(p,d)\in$ nextStates$(A_Q,s,a)$ do
(3)     add the transition $\xrightarrow{d}(p,m)$ to $W$
(4) return $W$
```

Lemma 5. *The transition $\xrightarrow{d}(p,m)$ is returned by* `Succ`(s,n,A_Q,G) *iff* $(s,n)\xrightarrow{a,d}(p,m)$ *is in* $H=A_Q\times G$.

Proof. By the definition of `Succ`, the transition $\xrightarrow{d}(p,m)$ is added to W if and only if $(n,a,m)\in E_G$ and $(p,d)\in$ `nextStates`(A_Q,s,a). By the definition of H, the presence of an edge labelled a from n to m in G and of a transition labelled a from s to p in A_Q results in an edge $(s,n)\xrightarrow{a,d}(p,m)$ in H. □

For incremental evaluation, a set `visited`$_R$ is maintained, storing tuples of the form (v,n,s), representing the fact that node n of G was visited in state s of A_Q having started the traversal from node v. Also maintained is a priority queue `queue`$_R$ containing tuples of the form (v,n,s,d,f), ordered by increasing values of d, where d is the approximation distance associated with visiting node n in state s having started from node v, and f is a flag denoting whether the tuple is final or non-final, with the latter being the initial value for f.

Recalling that Q has the form (X,R,Y), we begin by enqueueing the initial tuple $(v,v,s_0,0,f)$, if X is some node v, or enqueueing a set of initial tuples otherwise, one for each node v of G. We maintain a list `answers`$_R$ containing tuples of the form (v, n, d), where d is the smallest approximation distance of this answer tuple to Q and ordered by non-decreasing value of d. This list is used to avoid returning as an answer (v, n, d') for any $d' \geq d$. It is initialised to the empty list.

We then call function `getNext` shown below to return the next query answer, in order of non-decreasing approximation distance from Q. We see that `getNext` repeatedly dequeues the first tuple of $\texttt{queue}_R$, (v, n, s, d, f), adding (v, n, s) to $\texttt{visited}_R$ if the tuple is not final, until $\texttt{queue}_R$ is empty.

After dequeueing a tuple (v, n, s, d, f), we check to see whether the tuple is a final one; if not, we enqueue $(v, m, s', d + d', f)$ for each transition $\xrightarrow{d'} (s', m)$ returned by $\texttt{Succ}(s, n, A_Q, G)$ such that $(v, m, s') \notin \texttt{visited}_R$. If s is a final state, its annotation matches n, and the answer (v, n, d') has not been generated before for some d', then we add the final weight function for s to d, mark the tuple as final, and enqueue the tuple.

On the other hand, if a dequeued tuple is a final one and the answer (v, n, d') has not been generated before for some d', the triple (v, n, d) is returned after being added to $\texttt{answers}_R$.

We note that the maximum size of each of $\texttt{visited}_R$ and $\texttt{queue}_R$ is $2|R||V_G|^2$, and that the size of $\texttt{answers}_R$ will never exceed $|V_G|^2$. For $\texttt{queue}_R$ this result follows from the fact that, as in Dijkstra's shortest path algorithm, we assume that, for each combination of nodes v and n and state s, at most one tuple (v, n, s, d, f) is enqueued by using the priority queue's 'decrease key' operation.

```
Function getNext($X, R, Y, A_Q, G$)
     Input: query conjunct $(X, R, Y)$
     Output: triple $(v, n, d)$, where $v$ and $n$ are instantiations of $X$ and $Y$
(1)  while nonempty($\texttt{queue}_R$) do
(2)      $(v, n, s, d, f) \leftarrow$ dequeue($\texttt{queue}_R$)
(3)      if $f \neq$ 'final' then
(4)          add $(v, n, s)$ to $\texttt{visited}_R$
(5)          foreach $\xrightarrow{d'} (s', m) \in \texttt{Succ}(s, n, A_Q, G)$ s.t. $(v, m, s') \notin \texttt{visited}_R$
             do
(6)              enqueue($\texttt{queue}_R$, $(v, m, s', d + d', f)$)
(7)          if s is a final state $s_f$ and its annotation matches $n$ and
             $\nexists d'.(v, n, d') \in \texttt{answers}_R$ then
(8)              enqueue($\texttt{queue}_R$, $(v, n, s, d + \xi[s]$, 'final'$)$)
(9)      else
(10)         if $\nexists d'.(v, n, d') \in \texttt{answers}_R$ then
(11)             append $(v, n, d)$ to $\texttt{answers}_R$
(12)             return $(v, n, d)$
(13) return null
```

The following theorem shows that our incremental evaluation algorithm, represented by getNext, is correct: that is, given a single-conjunct query Q and a graph G, it returns the approximate answer of Q on G.

Theorem 1. *Let Q be a query with single conjunct (X, R, Y), and G a graph. Let $\texttt{visited}_R$, $\texttt{queue}_R$ and $\texttt{answers}_R$ be initialised as described above and* getNext *be called repeatedly until it returns null. When* getNext *returns null, then (1) $(v, n, d) \in \texttt{answers}_R$ if and only if (v, n, d) is in the approximate answer of Q on G, and (2) if $\texttt{answers}_R[i] = (v, n, d)$ and $\texttt{answers}_R[j] = (v', n', d')$, for non-negative integers i and j with $i < j$, then $d \leq d'$.*

Proof. Throughout, we define H as being the product automaton of the graph G and the approximate automaton A_Q constructed for Q; and s_0 and s_f indicate an initial state and final state in A_Q, respectively.

Part (1): ($\Leftarrow$) By Lemma 3, we need to show that if the minimum cost of *any* run from (s_0, v) to (s_f, n) is d, for some $v, n \in V_G$, then $(v, n, d) \in \texttt{answers}_R$.

We first show that if the minimum cost of any run in H from (s_0, v) to (s, n) (where s may or may not be a final state) is d, then tuple (v, n, s, d, f) is added to $\texttt{queue}_R$ before any tuple (v, n', s', d', f), where $d' > d$, is dequeued from $\texttt{queue}_R$. Assume that r is a minimum cost run in H from (s_0, v) to (s, n). The proof proceeds by induction on the number of transitions in r having non-zero cost.

Basis: For the base case, there are no transitions with non-zero cost in r, so the cost of r is zero. By definition we have that the tuple $(v, v, s_0, 0, f)$, possibly as one of a set of initial tuples, is enqueued in $\texttt{queue}_R$. When one of these zero-cost tuples is dequeued at line (2), we can see, by Lemma 5 and the invocation of $\texttt{Succ}(s_0, v, A_Q, G)$ at line (5), that all tuples representing the successive transitions in H will be enqueued in $\texttt{queue}_R$ at line (6); each of these tuples subsequently undergoes the same process. Because run r ends with (s, n), the tuple $(v, n, s, 0, f)$, where f is 'non-final', will be added to $\texttt{queue}_R$.

Now assume that, at some point, (v, n', s', d', f), where $d' > 0$ and f is 'non-final', is added to $\texttt{queue}_R$. As $\texttt{queue}_R$ is a priority queue ordered by non-decreasing values of cost, it is straightforward to see that (v, n', s', d', f) will not be dequeued before tuple $(v, n, s, 0, f)$ is enqueued.

Induction: For the inductive step, suppose that there is an $n \geq 0$ such that, for all $m \leq n$, if a minimum cost run in H of cost k from (s_0, v) to (u, w), say, has m transitions of non-zero cost, then tuple (v, w, u, k, f) will be placed on $\texttt{queue}_R$ before any tuple (v, w', u', k', f), $k' > k$, is dequeued from $\texttt{queue}_R$.

Now consider a minimum-cost run r of cost k which contains $n+1$ transitions with non-zero cost. Let run r be from (s_0, v) to (s, n), and let transition t be the last transition in r labelled with a cost $c > 0$. Hence, r can be viewed as run $r' \cdot r''$, where t is the first transition on r''. Clearly, the number of transitions with a non-zero cost in r' is n, and the cost of r' is $k - c$. Let (u, w) be the state in H at which r' ends and r'' starts. By the induction hypothesis, we know that the tuple $(v, w, u, k - c, f)$ will have been placed on $\texttt{queue}_R$ before any tuple $(v, w', u', k' - c, f)$, where $k' > k$, is dequeued from $\texttt{queue}_R$.

Suppose, in the worst case, that there is already a tuple (v, n', s', k', f), $k' > k$, on $\texttt{queue}_R$. Since $k - c < k'$, tuple $(v, w, u, k - c, f)$ will be dequeued before (v, n', s', k', f). Suppose that transition t is form (u, w) to (x, y) in H. When the tuple $(v, w, u, k-c, f)$ is dequeued at line (2), we know from Lemma 5 that invoking $\texttt{Succ}(u, w, A_Q, G)$ at line (5), will, at line (6), enqueue all tuples representing successive transitions in H, including the tuple (v, y, x, k, f).

If $y = n$ and $x = s$, the proof is complete. If not, tuple (v, y, x, k, f) will be dequeued before (v, n', s', k', f) since $k < k'$ and $\texttt{queue}_R$ is a priority queue. Because the remaining transitions on r'' are of cost zero, it is easy to see that (v, n, s, k, f) will be added to $\texttt{queue}_R$ before any (v, n', s', k', f) is dequeued.

So for a minimum cost run of cost d from (s_0, v) to (s_f, n), where s_f is a final state, we know that tuple (v, n, s_f, k, f) will also be *dequeued* from $\texttt{queue}_R$ before any tuple (v, n', s', d', f), $d' > d$, is dequeued (because $\texttt{queue}_R$ is a priority queue). The (v, n, s) triple is then added to $\texttt{visited}_R$ at line (4), enqueued as a 'final' tuple at line (8), and once again dequeued before any tuple at greater cost. This time the dequeued tuple results in the tuple (v, n, d) being added to $\texttt{answers}_R$ at line (11).

($\Rightarrow$) We show that if $(v, n, d) \in \texttt{answers}_R$, then the minimum cost of any run in H from (s_0, v) to (s_f, n), for some final state s_f, is d. The result then follows by Lemma 3. The proof is by contradiction.

Suppose that a triple $(v, n, d) \in \texttt{answers}_R$ but that the minimum cost of a run in H from (s_0, v) to (s_f, n) is $d' < d$. As (v, n, d) was added to $\texttt{answers}_R$ at line (11), we know, by line (7), that the triple (v, n, d') was not added to $\texttt{answers}_R$ prior to the tuple (v, n, s, d, f) being dequeued from $\texttt{queue}_R$ at line (2). There are only two possibilities which could give rise to this state.

The first possibility is that the tuple (v, m, s, d, f) was dequeued *before* the tuple (v, m, s, d', f) was enqueued in $\texttt{queue}_R$. However, as we have seen in ($\Leftarrow$) above, (v, m, s, d, f) cannot be dequeued before (v, m, s, d', f) is enqueued. Thus, we have a contradiction.

The second possibility is that, at line (5), the invocation of $\texttt{Succ}(s_i, v_i, A_Q, G)$, for some state s_i and some node v_i, did not return a transition $\xrightarrow{d''} (s_f, n)$, for some cost d'', where $d'' \leq d'$, and hence the tuple (v, m, s, d', f) was never enqueued at line (6). But we know, from Lemma 5, that $\texttt{Succ}$ returns all and only transitions that occur in H. Thus, we have a contradiction.

Therefore, either $(v, n, d) \notin \texttt{answers}_R$ or the minimum cost of any run in H from (s_0, v) to (s_f, n) is d.

<u>Part (2)</u>: From Part (1), we have that the tuple (v, n, s, d, f) — representing the minimum cost run in H from (s_0, v) to (s, n) — is dequeued from $\texttt{queue}_R$ before any tuple (v, n', s', d', f), where $d < d'$, is dequeued.

Suppose we have two runs, r and r'; suppose run r is from (s_{r0}, v_r) to (s_{rf}, n_r) of minimum cost d_r and run r' from (s'_{r0}, v'_r) to (s'_{rf}, n'_r) of minimum cost d'_r, where s_{rf} and s'_{rf} are final states. It is then straightforward to see that if the triple (v_r, n_r, d_r) had been added as the i^{th} item in $\texttt{answers}_R$ as a result of completely traversing run r, and the triple (v'_r, n'_r, d'_r) had been added as the j^{th} item in $\texttt{answers}_R$ as a result of completely traversing run r', for some $i < j$,

$$
\begin{array}{llll}
\text{(Subproperty)} & (1)\ \dfrac{(a,\texttt{sp},b)\ (b,\texttt{sp},c)}{(a,\texttt{sp},c)} & & (2)\ \dfrac{(a,\texttt{sp},b)\ (X,a,Y)}{(X,b,Y)} \\[2ex]
\text{(Subclass)} & (3)\ \dfrac{(a,\texttt{sc},b)\ (b,\texttt{sc},c)}{(a,\texttt{sc},c)} & & (4)\ \dfrac{(a,\texttt{sc},b)\ (X,\texttt{type},a)}{(X,\texttt{type},b)} \\[2ex]
\text{(Typing)} & (5)\ \dfrac{(a,\texttt{dom},c)\ (X,a,Y)}{(X,\texttt{type},c)} & & (6)\ \dfrac{(a,\texttt{range},c)\ (X,a,Y)}{(Y,\texttt{type},c)}
\end{array}
$$

Figure 6: RDFS Inference Rules

then $d_r \leq d'_r$. □

3.4. Ontology-Based Relaxation of Single-Conjunct Queries

The work in [11] considered ontology-based relaxation of conjunctive queries in the setting of the RDF/S data model and showed that query relaxation can be naturally formalised using *RDFS entailment*. The entailment was characterised by the derivation rules given in Figure 6, grounded in the semantics developed in [20, 21]. The work in [12] extended ontology-based relaxation to CRPQs, using an automaton-based approach. Here, we revisit the work of [12], giving full details and formally proving its correctness and complexity.

For RDF/S graphs G_1 and G_2, [11] states that $G_1 \models_{\texttt{rule}} G_2$ if G_2 can be derived from G_1 by iteratively applying the rules of Figure 6. The *closure* [21] of an RDF/S graph G under these rules is denoted by $\mathrm{cl}(G)$.

In the formalisation of RDF [21], infinite sets I of IRIs and L of RDF literals are assumed. The elements in $I \cup L$ are called RDF *terms*. Given a set of variables V disjoint from I and L, a *triple pattern* is a triple $(v_1, v_2, v_3) \in (I \cup V) \times (I \cup V) \times (I \cup V \cup L)$.

As described in Section 2, in this paper we assume that the data graph G and the ontology K are separate graphs, such that the nodes representing classes in V_G also appear as nodes in V_K. We also assume that query evaluation takes place on the graph given by restricting $\mathrm{cl}(G \cup K)$ to the nodes of $V_G \cup V_K$ and the edges labelled with labels from $\Sigma \cup \{\texttt{type}\} \cup V_{Prop}$. We call this the *closure of the data graph G with respect to the ontology K* and denote it by $closure_K(G)$. For example, the closure of the data graph of Figure 1 with respect to the ontology in Figure 2 is illustrated in Figure 7.

Terminology Note: Henceforth in this paper, we use the term 'graph' to mean 'data graph', unless otherwise stated. We use the term 'closure of the data graph G' to mean 'closure of the data graph G with respect to the ontology K' if the ontology K can be inferred from the context.

We recall that the edges of K have labels from the set $\{\texttt{sc}, \texttt{sp}, \texttt{dom}, \texttt{range}\}$. As in [12], we assume that the subgraphs of the ontology K induced by edges labelled sc and sp are acyclic, and that K is equal to its *extended reduction* [11]. These restrictions are necessary for associating an unambiguous cost

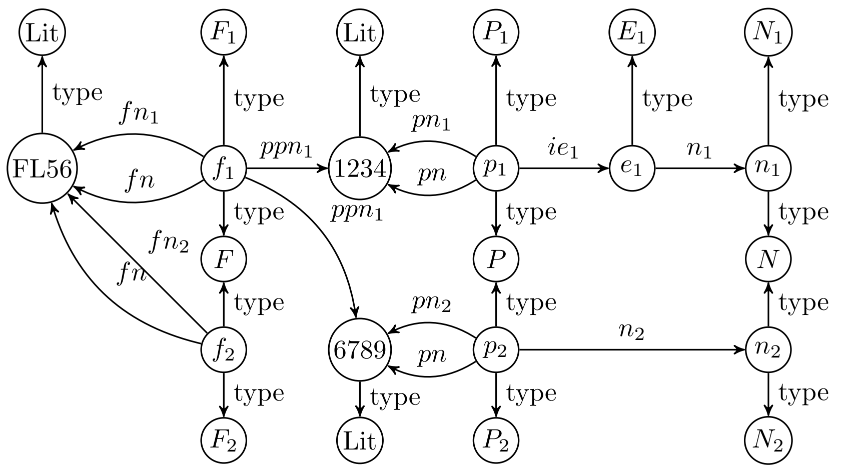

Figure 7: Closure of graph G in Figure 1 with respect to the ontology in Figure 2 where: F denotes *Flight*, P denotes *Person*, E denotes *Employee*, N denotes $NationalInsurancenumber$, fn denotes *flightNumber*, *ppn* denotes *passengerPassportNumber*, *pn* denotes *passportNumber*, *ie* denotes *isEmployee* and n denotes $nationalInsuranceNumber$.

with queries, so that query answers can be returned to users in order of increasing cost (see more details below, an illustrative example in Example 8, and full details in [11]).

The extended reduction of an ontology K, denoted by $\texttt{extRed}(K)$, is given by $\mathrm{cl}(K) - D$, where D is defined as follows: D is the set of triples in $\mathrm{cl}(K)$ that can be derived using rules (1) or (3) in Figure 6, or rules (e1), (e2), (e3) or (e4) in Figure 8. We note that, because $\mathrm{cl}(K)$ is closed with respect to the edge labels `sp` and `sc`, and also that the subgraphs induced by each of `sp` and `sc` are acyclic, the set D is uniquely defined[9].

As discussed in [11], although the rules of Figure 8 are not sound for RDFS entailment, using $\texttt{extRed}(K)$ allows us to perform what were termed *direct* relaxations in [11] which correspond to the "smallest" relaxation steps. This is necessary for associating an unambiguous cost to query answers, so that they can be returned incrementally in order of increasing relaxation cost[10]. In particular, we consider the cost of applying rule 2, 4, 5, or 6 of Figure 6 to be, respectively, c_{r2}, c_{r4}, c_{r5} or c_{r6}, each of which is a positive integer. (Since queries and data graphs cannot contain `sc` and `sp`, rules 1 and 3 are inapplicable to them, although of course they are used in computing $\mathrm{cl}(G \cup K)$.)

[9]Note also that removing `sp` and `sc` edges using rules (1) and (3) of Figure 6 is equivalent to forming the transitive reduction of $\mathrm{cl}(K)$ with respect to these labels (which is unique when the subgraphs they induce are acyclic).

[10]It is observed in [11] that $\texttt{extRed}(K)$ may not be logically equivalent to K. However, it is shown in that paper (Proposition 7) that the direct relaxations of any triple pattern using K can still be obtained from $\texttt{extRed}(K)$.

$$(e1)\ \frac{(b,\texttt{dom},c)\ (a,\texttt{sp},b)}{(a,\texttt{dom},c)} \qquad (e2)\ \frac{(b,\texttt{range},c)\ (a,\texttt{sp},b)}{(a,\texttt{range},c)}$$

$$(e3)\ \frac{(a,\texttt{dom},b)\ (b,\texttt{sc},c)}{(a,\texttt{dom},c)} \qquad (e4)\ \frac{(a,\texttt{range},b)\ (b,\texttt{sc},c)}{(a,\texttt{range},c)}$$

Figure 8: Additional rules used to compute the extended reduction of an RDFS ontology.

The set of variables mentioned in a triple pattern t is denoted by $\texttt{var}(t)$. Let t_1 and t_2 be normalised triple patterns (see Definition 4) such that $t_1, t_2 \notin \text{cl}(G \cup K)$, and $\texttt{var}(t_2) = \texttt{var}(t_1)$. Then t_1 *relaxes to* t_2, denoted $t_1 \leq t_2$, [11] if $(\{t_1\} \cup G \cup K) \models_{\texttt{rule}} t_2$. Note that when applying the rules of Figure 6 to triple patterns, rather than (ground) triples, a, b and c must be instantiated to RDF terms, while X and Y can be instantiated to either RDF terms or variables.

Given data graph G, ontology K and triple patterns t_1 and t_2, let G_1 and G_2 be the sets of triples in the closure of G that are 'matched' by t_1 and t_2, respectively. Then it can be shown that $t_1 \leq t_2$ if and only if $(G_1 \cup K) \models_{\texttt{rule}} G_2$. From now on in this section we assume that all triple patterns have been normalised, and likewise all triple forms of queries and paths.

A *graph pattern* P is a set of triple patterns. The set of variables mentioned in P is denoted by $\texttt{var}(P)$. Let P_1 and P_2 be graph patterns such that $\texttt{var}(P_2) = \texttt{var}(P_1)$ and for all $t_1 \in P_1$ and $t_2 \in P_2$, $t_1, t_2 \notin \text{cl}(G \cup K)$. Then P_1 *relaxes to* P_2, denoted $P_1 \leq P_2$, if for all $t_1 \in P_1$ there is a $t_2 \in P_2$ such that $t_1 \leq t_2$ and for all $t_2 \in P_2$ there is a $t_1 \in P_1$ such that $t_1 \leq t_2$. The relaxation relation is reflexive and transitive.

Example 6. Consider the ontology K described in Example 1 and shown in Figure 2. Let query Q comprise the single conjunct $(X, R, \text{'FL56'})$, where X is a variable, 'FL56' is a constant, and $R = (ppn_1^- \cdot fn_1)$. Recall that K contains the triples $(fn_1, \texttt{dom}, F_1)$ and $(F_1, \texttt{sc}, F)$. There is only a single $q \in L(R)$, namely $q = ppn_1^- fn_1$. Recalling the definition of a 'triple form' in Section 3.1, consider the following normalised triple form T of (Q, q):

$$(W_1, ppn_1, X), (W_1, fn_1, \text{'FL56'})$$

Notice that a triple form of (Q, q) is a graph pattern. Let P be the following graph pattern:

$$(W_1, ppn_1, X), (W_1, \texttt{type}, F)$$

Then T relaxes to P since $(W_1, fn_1, \text{'FL56'}) \leq (W_1, \texttt{type}, F_1)$ by rule 5, and $(W_1, \texttt{type}, F_1) \leq (W_1, \texttt{type}, F)$ by rule 4.

Note that, because of our requirement that variables be preserved when performing relaxation, rules 4, 5 and 6 can only be applied to the first or last triple

[11]For notational simplicity we assume that the parameters G and K are implicit.

pattern of a triple form of a sequence of labels, and even then only when a constant is present (as above). So, for example, $(ppn_1, \texttt{range}, Lit) \in K$ but we cannot apply rule 6 to the triple pattern (W_1, ppn_1, X) to relax it to $(X, \texttt{type}, Lit)$ because variable W_1 is lost in the process. We make use of this restriction in our algorithm for computing relaxed answers in Section 3.5 below. $\square$

We previously defined the exact semantics of single-conjunct regular path queries in Section 3.1. We now define the *relaxed semantics* of such queries.

Definition 11. Given a query Q with single conjunct (X, R, Y) and the closure of a data graph G with respect to an ontology K, $closure_K(G)$, let θ be a (Q, G)-matching. We use the notation $\theta(Q)$ to denote $(\theta(X), R, \theta(Y))$. A semipath p in $closure_K(G)$ *r-conforms* to $\theta(Q)$ if there is a $q \in L(R)$, a triple form T_q of $(\theta(Q), q)$ and a triple form T_p of p such that $T_q \leq T_p$. $\square$

Note that a path in $closure_K(G)$ can r-conform to a query on the basis of a triple pattern t relaxing to a triple pattern t' such that the constants in t and t' differ, due to applications of rules 5 and 6 (for example, the triple patterns $(W_1, fn_1, \text{‘FL56’})$ and $(W_1, \texttt{type}, F)$ in the previous example).

Example 7. Consider the query Q_3 from Example 3, the graph G of Figure 1 and the ontology K of Figure 2. Suppose that the second conjunct is used in a single-conjunct query Q_4 as follows:

$$Y \leftarrow RELAX(Y, pn_1^-.\texttt{type}, P_1)$$

Using matching θ_1 that matches Y to ‘1234’, semipath $(\text{‘1234’}, pn_1^-, p_1, \texttt{type}, P_1)$ r-conforms to $\theta_1(Q_4)$ since it matches the query exactly. Using matching θ_2 that matches Y to ‘6789’, semipath $(\text{‘6789’}, pn_2^-, p_2, \texttt{type}, P_2)$ r-conforms to $\theta_2(Q_4)$ because the normalised triple form $(W, pn_1, \text{‘6789’}), (W, \texttt{type}, P_1)$ for $\theta_2(Q_4)$ relaxes to the triple form $(W, pn, \text{‘6789’}), (W, \texttt{type}, P)$ which is a triple form for the semipath $(\text{‘6789’}, pn^-, p_2, \texttt{type}, P)$ in $closure_K(G)$. $\square$

Example 8. Consider again the ontology K described in Example 1 and shown in Figure 2. Recall that K contains the triples $(fn_1, \texttt{dom}, F_1)$ and $(F_1, \texttt{sc}, F)$. If we did not use the extended reduction of K, then K could also include the triple $(fn_1, \texttt{dom}, F)$. Now, given a query conjunct $(X, fn_1, \text{‘FL56’})$, we could apply rule 5 in order to relax $(X, fn_1, \text{‘FL56’})$ to $(X, \texttt{type}, F_1)$ with cost c_{r5} and to $(X, \texttt{type}, F)$, also with cost c_{r5}. However, the cost of relaxing $(X, fn_1, \text{‘FL56’})$ to $(X, \texttt{type}, F)$ should really be $c_{r5} + c_{r4}$, reflecting the cost of using rule 5 to relax $(X, fn_1, \text{‘FL56’})$ to $(X, \texttt{type}, F_1)$ followed by the cost of using rule 4 to relax $(X, \texttt{type}, F_1)$ to $(X, \texttt{type}, F)$. The extended reduction of K does not contain the triple $(fn_1, \texttt{dom}, F)$ because of applying rule e3 in reverse. $\square$

We now consider the cost of applying relaxations in order to be able to return answers ordered by increasing cost. For this we need the notion of *direct relaxation* [11]. The *direct relaxation relation*, which we denote here by $\preceq_R$, was defined in [11] to be the reflexive, transitive reduction of the relaxation relation $\leq$[12]. The *direct relaxations* of a triple pattern t (i.e., the triple patterns t' such that $t \preceq_R t'$) are the result of the smallest steps of relaxation (and the *indirect relaxations* of a triple pattern t are the triples t' such that $t \leq t'$ and $t \not\preceq_R t'$).

It is shown in [11] that a single application of each of the rules 2, 4, 5, 6 of Figure 6 to a triple pattern t and a triple $o \in \texttt{extRed}(K)$ yields precisely the direct relaxations of t with respect to K. We now extend this to graph patterns:

Given graph patterns P_1 and P_2, we say that P_1 *directly relaxes* to P_2, denoted $P_1 \preceq_R P_2$, if $P_1 = \{t_1\} \cup P$ and $P_2 = \{t_2\} \cup P$, for some (possibly empty) graph pattern P, and $t_1 \preceq_R t_2$. The *cost* of the direct relaxation is the cost of applying the rule that derives t_2 from t_1. The cost of a sequence of direct relaxations is the sum of the costs of each relaxation in the sequence.

Definition 12. Given ontology $K = \texttt{extRed}(K)$, graph $G = closure_K(G)$, semipath p in G, query Q with single conjunct (X, R, Y), sequence of labels $q \in L(R)$, (Q, G)-matching θ, triple form T_q for $(\theta(Q), q)$, and triple form T_p for p such that $T_q \leq T_p$:

- the *relaxation distance* from p to $(\theta(Q), q)$ is the minimum cost of any sequence of direct relaxations which yields T_p from T_q; the cost of the empty sequence of direct relaxations (so that T_q is already a triple form of p) is zero;
- the *relaxation distance* from p to $\theta(Q)$ is the minimum relaxation distance from p to $(\theta(Q), q)$ for any sequence of labels $q \in L(R)$;
- the *relaxation distance* of $\theta(Q)$, denoted $rdist(\theta, Q)$, is the minimum relaxation distance to $\theta(Q)$ from any semipath p that r-conforms to $\theta(Q)$;
- the *relaxed answer* of Q on G is a list of pairs $(\theta(vars), rdist(\theta, Q))$, such that there is a semipath in G that r-conforms to $\theta(Q)$, ranked in order of non-decreasing relaxation distance;
- the *relaxed top-k answer* of Q on G is a list containing the first k tuples in the relaxed answer of Q on G.

$\square$

3.5. Computing the relaxed answer

We now describe how the relaxed answer can be computed, starting from the weighted NFA M_R that recognises $L(R)$ which was described in Section 3.2.

Given a query Q with single conjunct (X, R, Y), a weighted automaton $M_R = (S, \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \delta, s_0, S_f, \xi)$ that does not contain ϵ-transitions, and ontology K such that $K = \texttt{extRed}(K)$, we construct as described below

[12]Once again for notational simplicity, we view the parameters G and K as being implicit.

the automaton $M_R^K = (S', \Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}, \tau, S_0, S'_f, \xi')$ of M_R with respect to K. The set of states S' includes the states in S as well as any new states defined below. S_0 and S'_f are sets of initial and final states, respectively, with S_0 including the initial state s_0 of M_R, S'_f including all final states S_f of M_R, and both possibly including additional states as defined below. We obtain the *relaxed automaton* M_Q^K by annotating each state in S_0 and S'_f either with a constant or with the wildcard symbol $*$, depending on whether X and Y in Q are constants or variables. We recall that ξ (and initially ξ') is the final weight function mapping each state in S_f to a non-negative integer. The transition relation τ includes the transitions in δ as well as any transitions specified by the rules defined below. The rules below are applied repeatedly until no new transitions or states arise. The process terminates because of our assumption that the subgraphs of K induced by edges labelled sc and sp are acyclic:

- (rule 2) For each transition $(s, a, d, t) \in \tau$ (respectively $(s, a^-, d, t) \in \tau$) and triple $(a, \mathtt{sp}, b) \in K$, there is a transition $(s, b, d + c_{r2}, t)$ (resp. $(s, b^-, d + c_{r2}, t)$) in τ.

- (rule 4 (i)) If there is a transition $(s, \mathtt{type}, d, t) \in \tau$ where $t \in S'_f$ and $(c, \mathtt{sc}, c') \in K$ such that t is annotated with c, there is a final state t' annotated with c' in S'. State t' has the same set of outgoing transitions as t. For each transition $(s, \mathtt{type}, d, t) \in \tau$, there is a transition $(s, \mathtt{type}, d + c_{r4}, t')$ in τ.

- (rule 4 (ii)) If there is a transition $(s, \mathtt{type}^-, d, t) \in \tau$ where $s \in S_0$ and $(c, \mathtt{sc}, c') \in K$ such that s is annotated with c, there is an initial state s' annotated with c' in S'. State s' has the same set of incoming transitions as s. For each transition $(s, \mathtt{type}^-, d, t) \in \tau$, there is a transition $(s', \mathtt{type}^-, d + c_{r4}, t)$ in τ.

- (rule 5) If there is a transition $(s, a, d, t) \in \tau$ (resp. $(s, a^-, d, t) \in \tau$) where $t \in S'_f$ (resp. $s \in S_0$) and $(a, \mathtt{dom}, c) \in K$, there is a final state t' (resp. initial state s') annotated with c in S'. State t' has the same set of outgoing transitions as t (resp. s' has the same set of incoming transitions as s). For each transition $(s, a, d, t) \in \tau$ (resp. $(s, a^-, d, t) \in \tau$), there is a transition $(s, \mathtt{type}, d + c_{r5}, t')$ (resp. $(s', \mathtt{type}^-, d + c_{r5}, t)$) in τ.

- (rule 6) If there is a transition $(s, a, d, t) \in \tau$ (resp. $(s, a^-, d, t) \in \tau$) where $s \in S_0$ (resp. $t \in S'_f$) and $(a, \mathtt{range}, c) \in K$, there is an initial state s' (resp. final state t') annotated with c in S'. State s' has the same set of incoming transitions as s (resp. t' has the same set of outgoing transitions as t). For each transition $(s, a, d, t) \in \tau$ (resp. $(s, a^-, d, t) \in \tau$), there is a transition $(s', \mathtt{type}^-, d + c_{r6}, t)$ (resp. $(s, \mathtt{type}, d + c_{r6}, t')$) in τ.

We call the initial and final states specified by rules 4, 5 and 6 *cloned* states.

Example 9. Consider once again the ontology K shown in Figure 2 and the query Q_4 from Example 7:

$$Y \leftarrow RELAX(Y, pn_1^-.\mathtt{type}, P_1)$$

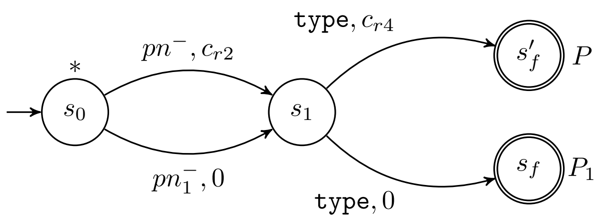

Figure 9: Relaxed automaton M_Q^K for conjunct $(Y, pn_1^-.\texttt{type}, P_1)$.

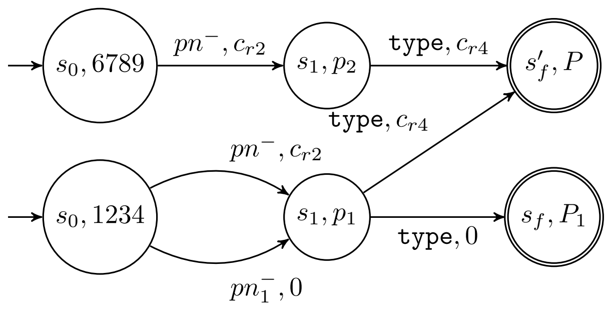

Figure 10: A subautomaton of the product automaton H.

The relaxed automaton M_Q^K initially comprises the states $\{s_0, s_1, s_f\}$ and two transitions labelled with cost zero between them, as shown in the lower part of Figure 9. From Figure 2, we see that K contains the triples $(pn_1, \texttt{sp}, pn)$ and $(P_1, \texttt{sc}, P)$. If we apply the transformation for rule 2 to the transition labelled $pn_1^-, 0$ and the triple $(pn_1, \texttt{sp}, pn) \in K$, then we add the transition labelled pn^-, c_{r2} from s_0 to s_1. If we now apply the transformation for rule 4 to the transition labelled $\texttt{type}, 0$ and the triple $(P_1, \texttt{sc}, P) \in K$, then a new final state s'_f, annotated with P is added, as is the transition labelled $\texttt{type}, c_{r4}$ from s_1 to s'_f.

Now consider the closure of the graph G shown in Figure 7 with respect to the ontology K shown in Figure 2. A subautomaton of the product automaton H of M_Q^K and $closure_K(G)$ is shown in Figure 10.

For a matching θ_1 that maps Y to '1234', it is easy to see that there are four runs in H from the initial state $(s_0, 1234)$ to a final state, with costs 0, c_{r2}, c_{r4} and $c_{r2} + c_{r4}$. Hence the relaxation distance of the answer '1234' is 0. For a matching θ_2 that maps Y to '6789', there is only one run in H from $(s_0, 6789)$ to a final state, with cost $c_{r2} + c_{r4}$, so the relaxation distance of the answer '6789' is $c_{r2} + c_{r4}$. □

The following proposition shows firstly the correctness of the construction and traversal of the product automaton, H; and secondly that the relaxation distance from a semipath in a graph G to the matchings for a single-conjunct query Q is equal to the minimum cost of a run in H.

Proposition 3. *Let Q be a query comprising a single conjunct (X, R, Y). Let $M_Q^K = (S', \Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}, \tau, S_0, S_f, \xi)$ be the relaxed automaton for query Q and ontology $K = \texttt{extRed}(K)$, where the ϵ-transitions have been removed from M_Q^K. Let $G = closure_K(G)$ be a graph and H be the product automaton of M_Q^K and G. Let θ be a (Q, G)-matching such that $\theta(X) = v_0$ and $\theta(Y) = v_n$. (1) There is a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G that r-conforms to $\theta(Q)$ if and only if there is a run $r = ((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}), l_n, c_n, (s_n, v_n))$ in H, where $s_0 \in S_0$ and $s_n \in S_f$. (2) The relaxation distance of $\theta(Q)$ is given by $c_1 + \cdots + c_n$ if and only if r is of minimum cost over all runs from (s_0, v_0) to (s_n, v_n) for any $s_0 \in S_0$ and $s_n \in S_f$.*

Proof. Part (1): $(\Rightarrow)$ Let p be a semipath $(v_0, l_1, \ldots, l_n, v_n)$ in G that r-conforms to $\theta(Q)$. Hence there is a $q \in L(R)$, a triple form T_q of $(\theta(Q), q)$ and a triple form T_p of p such that $T_q \leq T_p$. Since $q \in L(R)$, we know that there is a run in M_R corresponding to T_q. We therefore need to show that there is a run in M_Q^K corresponding to T_p. The proof proceeds by induction on the number of direct relaxations required to yield T_p from T_q.

Basis: When the number of direct relaxations applied to T_q is zero, $T_q = P_0 = T_p$. Thus, there is a run for T_p in M_Q^K, which corresponds to a run in M_R.

Induction: For the inductive step, assume that there is an $n \geq 0$ such that, for all $m \leq n$, if m direct relaxations are used in a relaxation sequence which yields T_p from T_q, there is a corresponding run in M_Q^K.

Now assume that $n + 1$ direct relaxations are required to produce T_p from T_q. Let such a sequence be given by $S = P_0 \preceq_R \cdots \preceq_R P_n \preceq_R P_{n+1}$, where $P_0 = T_q$ and $P_{n+1} = T_p$. From the induction hypothesis, we know that there is a run r_n in M_Q^K corresponding to the sequence of direct relaxations given by the sequence $S' = P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$.

We consider in turn each type of direct relaxation operation induced by the rules given in Figure 6 which can be used to produce P_{n+1} from P_n. We show that, corresponding to each such operation, is a transition in M_Q^K which, when traversed, gives rise to a run for T_p in M_Q^K. In all cases below, d denotes the cost of the transition.

Rule 2: Suppose that P_{n+1} is produced by applying rule 2 to the triple $(a, \texttt{sp}, b) \in K$ and the triple pattern $f = (W_{m-1}, a, W_m)$ in P_n which results in f being replaced by (W_{m-1}, b, W_m) in P_{n+1}, where W_{m-1} and W_m are variables or constants. Suppose that f is matched by the transition (h, a, d, s) in r_n. By construction, M_Q^K has a transition $(h, b, d + c_{r2}, s)$ and hence replacing (h, a, d, s) by $(h, b, d + c_{r2}, s)$ in r_n yields a run for P_{n+1}.

On the other hand, if P_{n+1} is produced by applying rule 2 to (W_m, a, W_{m-1}) in P_n, where the matching transition in r_n is (h, a^-, d, s), then (W_m, b, W_{m-1}) is in P_{n+1} and there is a transition $(h, b^-, d + c_{r2}, s)$ in M_Q^K.

Rule 4: Suppose that P_{n+1} is produced by applying rule 4 to the triple $(c, \texttt{sc}, c') \in K$ and the triple pattern $f = (W_m, \texttt{type}, c)$ in P_n which results in f being replaced by $(W_m, \texttt{type}, c')$ in P_{n+1}, where W_m is a variable. Suppose that f is matched by the transition $(h, \texttt{type}, d, s)$ in r_n, where s is annotated with c. By the induction hypothesis and the fact that f is the last triple pattern in P_n, s

must be a final state. By construction, M_Q^K has a transition $(h, \mathtt{type}, d+c_{r4}, s')$ where $s' \in S_f$ and is annotated with c', and hence replacing $(h, \mathtt{type}, d, s)$ by $(h, \mathtt{type}, d+c_{r4}, s')$ in r_n yields a run for P_{n+1}.

On the other hand, if P_{n+1} is produced by applying rule 4 to $f = (c, \mathtt{type}, W_m)$ in P_n, where W_m is a variable and where the matching transition in r_n is $(s_0, \mathtt{type}^-, d, h)$ (where, by the induction hypothesis and the fact that f must be the first triple pattern in P_n, we have that s_0 must be an initial state), then $(c', \mathtt{type}, W_m)$ is in P_{n+1} and there is a transition $(s'_0, \mathtt{type}^-, d+c_{r4}, h)$, where s'_0 is annotated with c', in M_Q^K.

Rule 5: Suppose that P_{n+1} is produced by applying rule 5 to the triple $(a, \mathtt{dom}, c) \in K$ and the triple pattern f in P_n. Because applying relaxation requires that the variables between pairs of triple patterns remain the same, it must be the case that f is of the form (W, a, z), where $z = \theta(Y)$ or $z = v_n$ (respectively $z = \theta(X)$ or $z = v_0$), and W is a variable. Hence, f must be matched by a transition t which leads to a final state (respectively starts from an initial state) in r_n; thus we have $t = (h, a, d, s)$ (respectively (s_0, a^-, d, h)) where $s \in S_f$ (respectively $s_0 \in S_0$). From the application of rule 5, we also have that f is replaced by $(W, \mathtt{type}, c)$ in P_{n+1}. By construction, M_Q^K has a transition $(h, \mathtt{type}, d+c_{r5}, s')$ (respectively $(s'_0, \mathtt{type}^-, d+c_{r5}, h)$) where $s' \in S_f$ (respectively $s'_0 \in S_0$) and is annotated with c, and hence replacing (h, a, d, s) (respectively (s_0, a^-, d, h)) by $(h, \mathtt{type}, d + c_{r5}, s')$ (respectively $(s'_0, \mathtt{type}^-, d + c_{r5}, h)$) in r_n yields a run for P_{n+1}.

Rule 6: Suppose that P_{n+1} is produced by applying rule 6 to the triple $(a, \mathtt{range}, c) \in K$ and the triple pattern f in P_n. Because applying relaxation requires that the variables between pairs of triple patterns remain the same, it must be the case that f is of the form (z, a, W), where $z = \theta(X)$ or $z = v_0$ (respectively $z = \theta(Y)$ or $z = v_n$), and W is a variable. Hence, f must be matched by a transition t which starts from an initial state (respectively leads to a final state) in r_n; thus we have $t = (s_0, a, d, h)$ (respectively (h, a^-, d, s)) where $s_0 \in S_0$ (respectively $s \in S_f$). From the application of rule 6, we also have that f is replaced by $(W, \mathtt{type}, c)$ in P_{n+1}. By construction, M_Q^K has a transition $(s'_0, \mathtt{type}^-, d+c_{r6}, h)$ (respectively $h, \mathtt{type}, d+c_{r6}, s')$ where $s'_0 \in S_0$ (respectively $s' \in S_f$) and is annotated with c, and hence replacing (s_0, a, d, h) (respectively (h, a^-, d, s)) by $(s'_0, \mathtt{type}^-, d + c_{r6}, h)$ (respectively $h, \mathtt{type}, d + c_{r6}, s')$ in r_n yields a run for P_{n+1}.

Thus, we have shown that, in all cases, there is a run in M_Q^K corresponding to T_p. By the construction of H from M_Q^K and G, if we have the run (s_0, l_1, c_1, s_1) $, \ldots, (s_{n-1}, l_n, c_n, s_n)$ in M_Q^K and the semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G, we have a run $r = ((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}), l_n, c_n, (s_n, v_n))$ in H, where $s_0 \in S_0$ and $s_n \in S_f$.

($\Leftarrow$) Let r be a run $((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}), l_n, c_n, (s_n, v_n))$ in H, where $s_0 \in S_0$ and $s_n \in S_f$. By the construction of H from M_Q^K and G, there must be a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G and a run (s_0, l_1, c_1, s_1), $\ldots, (s_{n-1}, l_n, c_n, s_n)$ in M_Q^K. Let T_p be a triple form of p.

We know that there is a run from $s_0 \in S_0$ to $s_n \in S_f$ in M_Q^K corresponding to T_p. By the construction of M_Q^K, we also know that the transitions added to the transition relation τ correspond to the relaxation operations induced by the rules in Figure 6. Thus, each transition in any run in M_Q^K corresponds to either a transition in M_R or a relaxation of one of the transitions in M_R. By definition, we also know that every run in M_R corresponds to an acceptance of a sequence of labels $q \in L(R)$.

We therefore need to show that every run in M_Q^K corresponds to a sequence of direct relaxations of the triple form of some sequence of labels $q \in L(R)$; i.e. that a run in M_Q^K corresponds to T_p.

For the purposes of the proof, we assume that each transition in M_Q^K is assigned a *derivation number*. Each transition in M_R is assigned a derivation number of zero. Then, each application of rule 2, 4, 5 or 6 will result in a new transition t', derived from an existing transition t, where the derivation number of t' is the derivation number of t plus one. The *derivation length* of a run r is then the sum of the derivation numbers of the transitions comprising r. The proof proceeds by induction on the derivation length of a run.

Basis: For the base case, we consider runs having a derivation length of zero; i.e. runs only containing transitions with a derivation number of zero, which are present in M_R. Thus, $T_q = P_0 = T_p$ (no relaxation has occurred), and every run in M_R corresponds to T_q, as in Definition 8.

Induction: For the inductive step, assume that there is an $n \geq 0$ such that, for all $m \leq n$, if m is the derivation length of a run r in M_Q^K, then r corresponds to a sequence of m direct relaxations which yields T_p from T_q.

Now let r_{n+1} be a run in M_Q^K with derivation length $n+1$. We need to show how r_{n+1} corresponds to a triple form representing a sequence of $n+1$ direct relaxations, given by $T_q = P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n \preceq_R P_{n+1} = T_p$.

Since r_{n+1} has derivation length $n+1$, there must be a transition t in r_{n+1} with a non-zero derivation number λ. Below, we consider each rule that may have given rise to t in M_Q^K. In all cases, d indicates the cost of the transition.

Rule 2: Suppose that $t = (h, b, d, s)$. Because t was added using rule 2, there must be a transition $t' = (h, a, d - c_{r2}, s)$ in M_Q^K for some $(a, \texttt{sp}, b) \in K$, with a derivation number of $\lambda - 1$. Replacing t in r_{n+1} by t' gives rise to a run r_n with a derivation length of n. From the induction hypothesis, there is a sequence of n direct relaxations $P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$ and a triple pattern $f = (W_{m-1}, a, W_m)$ in P_n matched by t'. Applying rule 2 to f will give rise to a triple form P_{n+1} corresponding to a sequence of $n+1$ direct relaxations having been applied to q.

By an analogous process where $t = (h, b^-, d, s)$, we can show that its matching triple pattern (W_m, b, W_{m-1}) is in P_{n+1}.

Rule 4: Suppose that $t = (h, \texttt{type}, d, s'_n)$ where $s'_n \in S_f$ and is annotated with c'. Because t was added using rule 4, there must be a transition $t' = (h, \texttt{type}, d - c_{r4}, s_n)$ in M_Q^K for some $(c, \texttt{sc}, c') \in K$, with a derivation number of $\lambda - 1$, and where $s_n \in S_f$ and is annotated with c. Replacing t in r_{n+1} by t' gives rise to a run r_n with a derivation length of n. From the induction hypothesis, there is a sequence of n direct relaxations $P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$

and a triple pattern $f = (W_m, \texttt{type}, c)$ in P_n matched by t'. Applying rule 4 to f will give rise to a triple form P_{n+1} corresponding to a sequence of $n+1$ direct relaxations having been applied to q.

By an analogous process where $t = (s'_0, \texttt{type}^-, d, h)$, and $s'_0 \in S_0$ and is annotated with c', we can show that its matching triple pattern $(c', \texttt{type}, W_m)$, where W_m is a constant, is in P_{n+1}.

Rule 5: Suppose that $t = (h, \texttt{type}, d, s'_n)$ (respectively $(s'_0, \texttt{type}^-, d, h)$) where $s'_n \in S_f$ (respectively $s'_0 \in S_0$). Because t was added using rule 5, there must be a transition $t' = (h, a, d - c_{r5}, s_n)$ (respectively $(s_0, a^-, d - c_{r5}, h)$) in M_Q^K for some $(a, \texttt{dom}, c) \in K$, with a derivation number of $\lambda - 1$, and where $s_n \in S_f$ (respectively $s_0 \in S_0$). Replacing t in r_{n+1} by t' gives rise to a run r_n with a derivation length of n. From the induction hypothesis, there is a sequence of n direct relaxations $P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$ and a triple pattern f in P_n matched by t'. Because applying relaxation requires that the variables between pairs of triple patterns remain the same, and from the fact that f is matched by t' which leads to a final state (respectively starts from an initial state), it must be the case that f is of the form (W, a, z), where $z = \theta(Y)$ or $z = v_n$ (respectively $z = \theta(X)$ or $z = v_0$), and W is a variable. Applying rule 5 to f will give rise to a triple form P_{n+1} corresponding to a sequence of $n+1$ direct relaxations having been applied to q.

Rule 6: Suppose that $t = (s'_0, \texttt{type}^-, d, h)$ (respectively $(h, \texttt{type}, d, s'_n)$) where $s'_0 \in S_0$ (respectively $s'_n \in S_f$). Because t was added using rule 6, there must be a transition $t' = (s_0, a, d - c_{r6}, h)$ (respectively $(h, a^-, d - c_{r6}, s_n)$) in M_Q^K for some $(a, \texttt{range}, c) \in K$, with a derivation number of $\lambda - 1$, and where $s_0 \in S_0$ (respectively $s_n \in S_f$). Replacing t in r_{n+1} by t' gives rise to a run r_n with a derivation length of n. From the induction hypothesis, there is a sequence of n direct relaxations $P_0 \preceq_R P_1 \preceq_R \cdots \preceq_R P_n$ and a triple pattern f in P_n matched by t'. Because applying relaxation requires that the variables between pairs of triple patterns remain the same, and from the fact that f is matched by t' which starts from an initial state (respectively leads to a final state), it must be the case that f is of the form (z, a, W), where $z = \theta(X)$ or $z = v_0$ (respectively $z = \theta(Y)$ or $z = v_n$), and W is a variable. Applying rule 6 to f will give rise to a triple form P_{n+1} corresponding to a sequence of $n+1$ direct relaxations having been applied to q.

Part (2): ($\Rightarrow$) Let θ be a matching mapping X to v_0 and Y to v_n in G, r be a run $((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}), l_n, c_n, (s_n, v_n))$ in H, where $s_0 \in S_0$ and $s_n \in S_f$, and let the relaxation distance of $\theta(Q)$ be given by $c_1 + \cdots + c_n$. By definition, we know that the relaxation distance of $\theta(Q)$ is the minimum relaxation distance to $\theta(Q)$ from any semipath p that r-conforms to $\theta(Q)$. By the construction of H from M_Q^K and G, there is a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G and a run $(s_0, l_1, c_1, s_1), \ldots, (s_{n-1}, l_n, c_n, s_n)$ in M_Q^K. From Part (1), we know that p r-conforms to $\theta(Q)$. Thus, we have that $c_1 + \cdots + c_n$ is the minimum relaxation distance to $\theta(Q)$ from p.

Also by definition, we have that $c_1 + \cdots + c_n$ is the minimum relaxation distance from p to $(\theta(Q), q)$ for any sequence of labels $q \in L(R)$ and that, for any such q, $c_1+\cdots+c_n$ is the minimum cost of any sequence of direct relaxation operations which yields p from q. Thus, there can be no run of cost less than $c_1+\cdots+c_n$ for p in H as this would contradict the fact that p is of relaxation distance $c_1+\cdots+c_n$ from $\theta(Q)$. Hence, r is a minimum cost run over all runs from (s_0, v_0) to (s_n, v_n).

($\Leftarrow$) We assume that r is a minimum cost run in H over all runs from some (s_0, v_0) to some (s_n, v_n), where $s_0 \in S_0$ and $s_n \in S_f$. This means, by construction, that the minimum cost of any run is $c_1+\cdots+c_n$. Also by the construction of H, there is a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G. But Part (1) shows us that p r-conforms to $\theta(Q)$. Thus, the relaxation distance of $\theta(Q)$ is $c_1+\cdots+c_n$. □

The following two propositions show firstly an upper bound for the size of the relaxed automaton, M_Q^K; and secondly that the relaxed answer of a single-conjunct query Q on the closure of a graph G can be computed in time that is polynomial in the size of Q, G and the ontology K.

Proposition 4. *Let Q be a query comprising a single conjunct (X, R, Y), $M_Q^K = (S', \Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}, \tau, S_0, S_f, \xi)$ be the relaxed automaton for Q and ontology $K = \mathtt{extRed}(K)$, where $K = (V_K, E_K)$. M_Q^K has at most $2|R|(|V_K|+1)$ states and $2|R|^2(|E_K||V_K| + 8|E_K| + 8)$ transitions.*

Proof. From [16] and [18], we have that $M_R = (S, \Sigma \cup \{\mathtt{type}\}, \delta, s_0, S_f, \xi)$ contains at most $2|R|$ states and $4|R|$ transitions. The subsequent removal of ϵ-transitions as described in [18] may result in at most $16|R|^2$ transitions. We recall that M_Q^K initially consists of all states S in M_R and all these transitions.

We know that each node in the set V_K is either a class node or a property node in K. From the construction of M_Q^K — in particular, by the application of rules 4, 5 and 6 — we can see that, for each class node in V_K, at most *one* new state, corresponding to the class node, is added for any existing state s, provided that $s \in S_0$ or $s \in S_f$. Hence, we can see that no more than $|V_K|$ new states may be added for each of the original states in S, which results in at most $2|R||V_K|$ new states in total. Thus, M_Q^K has at most $2|R|(|V_K|+1)$ states.

Since there are at most $|E_K|$ edges in K with label sp, rule 2 adds at most $16|R|^2|E_K|$ transitions to M_Q^K. Rules 4, 5 and 6 can collectively be applied no more than $|E_K|$ times. Each application results, in the worst case, in $|R|$ transitions being added for each of the $2|R||V_K|$ new, cloned states, giving rise to at most $2|R|^2|V_K||E_K|$ transitions. Thus, overall M_Q^K has at most $2|R|^2(|E_K||V_K| + 8|E_K| + 8)$ transitions. □

Proposition 5. *Let $K = (V_K, E_K)$ be an ontology such that $k = \mathtt{extRed}(K)$, $G = (V_G, E_G)$ be a graph such that $G = closure_K(G)$, and Q be a single-conjunct query using regular expression R over alphabet $\Sigma \cup \Sigma^- \cup \{\mathtt{type}, \mathtt{type}^-\}$. The relaxed answer of Q on G can be found in time $O(|R|^2|V_K|^2|V_G|(|R||E_K||E_G| + |V_G|\log(|R||V_K||V_G|)))$.*

PROOF. Let M_Q^K be the relaxed automaton constructed from R and K, and H be the product automaton constructed from M_Q^K and G. Proposition 3 shows that traversing H yields all relaxed answers to Q. Proposition 4 tells us that M_Q^K has at most $2|R|(|V_K|+1)$ states and $2|R|^2(|E_K||V_K|+8|E_K|+8)$ transitions. Therefore H has at most $2|R||V_G|(|V_K|+1)$ nodes and $2|R|^2|E_G|(|E_K||V_K|+8|E_K|+8)$ edges. If we assume that H is sparse (which is highly likely), then running Dijkstra's algorithm on each node of a graph with node set N and edge set A can be done in time $O(|N||A|+|N|^2\log|N|)$. So, for graph H, the combined time complexity is $O(|R|^3|E_K||V_K|^2|V_G||E_G|+|R|^2|V_K|^2|V_G|^2\log(|R||V_K||V_G|))$ which is equal to $O(|R|^2|V_K|^2|V_G|(|R||E_K||E_G|+|V_G|\log(|R||V_K||V_G|)))$. $\square$

As a corollary, it is easy to see that the data complexity is $O(|V_K|^2|V_G|(|E_K||E_G|+|V_G|\log(|V_K||V_G|)))$ and the query complexity is $O(|R|^3)$. The space complexity is dominated by the space requirements of H given in the proof above.

3.6. Incremental Evaluation of RELAX Conjuncts

In order to compute the relaxed answers to a single-conjunct regular path query incrementally, we can use the `getNext` function from Section 3.2 along with the same initialisations of the algorithm's variables. The only difference is that the `Succ` function now uses the automaton M_Q^K rather than the automaton A_Q.

3.7. Multi-Conjunct Queries

Recall the general form of a CRPQ Q from Section 2:

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

where any of the conjuncts may have the APPROX or RELAX operator applied to them. Let θ be a (Q,G)-matching. If conjuncts $i_1,\ldots,i_j$, $j \le n$, have APPROX or RELAX applied to them, the *distance* from θ to Q, $dist(\theta,Q)$, is defined as

$$dist(\theta,(X_{i_1},R_{i_1},Y_{i_1})) + \cdots + dist(\theta,(X_{i_j},R_{i_j},Y_{i_j}))$$

where $dist(\theta,(X_{i_k},R_{i_k},Y_{i_k}))$ is the approximation distance if conjunct i_k has APPROX applied to it and is the relaxation distance if i_k has RELAX applied to it.

Let $\theta(Z_1,\ldots,Z_m)=(a_1,\ldots,a_m)$. θ is a *minimum-distance* matching if for all (Q,G)-matchings ϕ such that $\phi(Z_1,\ldots,Z_m)=(a_1,\ldots,a_m)$, $dist(\theta,Q)\le dist(\phi,Q)$.

The *answer* of Q on G is the list of pairs $(\theta(Z_1,\ldots,Z_m), dist(\theta,Q))$, for some minimum-distance matching θ, ranked in order of non-decreasing distance. The *top-k answer* of Q on G comprises the first k tuples in the answer of Q on G.

The query Q can be evaluated by joining the answers arising from the evaluation of each of its conjuncts. For each conjunct with APPROX or RELAX applied to it we can use the techniques described in previous sections to incrementally compute a relation r_i with scheme $(X_i, Y_i, Distance)$. A query evaluation tree can be constructed for Q, consisting of nodes denoting join operators and nodes representing conjuncts of Q. Since the answers for single conjuncts are ordered by non-decreasing distance, pipelined execution of any rank-join operator (see e.g. [22]) can be used to output the answers to Q in order of non-decreasing distance. If the conjuncts of Q are acyclic, the evaluation can be accomplished in polynomial time [23], since there are a fixed number of head variables in Q and we process leaf nodes (denoting conjuncts) in a bottom-up manner (beginning at the leaf nodes) by executing a sequence of semi-joins.

4. The FLEX Operator

We now combine the use of both query approximation and query relaxation within one integrated FLEX operator that applies both techniques at the same time. This aims to allow greater ease of querying for users, in that they do not need to be aware of the ontology structure and to identify explicitly which parts of their overall query are amenable to relaxation and which to approximation. As we will see below, it also allows answers to be returned that cannot be obtained by applying only APPROX or RELAX to individual query conjuncts.

Definition 13. Let $K = \texttt{extRed}(K)$ be an ontology. A *flex* operation is either an edit operation on a symbol in $\Sigma \cup \Sigma^-$ or a direct relaxation using K[13].

Example 10. Referring to the same data graph and ontology as in the examples in Section 2, the user may pose the following query which uses the new FLEX operator to apply both approximation and relaxation simultaneously to both conjuncts:

$$Y \leftarrow FLEX(\text{‘FL56’}, fn_1.ppn_1.pn_1^-, Y), FLEX(Y, n_1.type, N_1)$$

By replacing fn_1 by fn_1^- and inserting ie_1 after pn_1^- (at a cost of $c_s + c_i$), the result e_1 is returned. By replacing fn_1 by fn_1^-, relaxing pn_1^- to pn^-, replacing n_1 by n_2, and relaxing N_1 to N, (at an overall cost of $2c_s + c_{r2} + c_{r4}$), the result p_2 is returned. We note that result p_2 could not have been returned by applying only APPROX or RELAX to the two conjuncts — it requires the FLEX operator in order to be returned. □

Definition 14. Let $K = \texttt{extRed}(K)$ be an ontology, $G = closure_K(G)$ a graph, p a semipath in G, Q a query with single conjunct (X, R, Y), θ a (Q, G)-matching, $q \in L(R)$, T_q a triple form for $(\theta(Q), q)$, and T_p a triple form for

[13]Edit operations on labels in $\{\texttt{type}, \texttt{type}^-\}$ are not allowed for FLEX because, as we will see below, allowing such edits would require multiple rounds of approximation and relaxation to be applied to yield a final automaton, rather than a simple two-step process that we describe below.

p. We write $T_q \preceq T_p$, if T_q can be transformed to T_p (up to variable renaming) by a sequence of flex operations. The *cost* of the sequence of flex operations is the sum of the costs of each operation. The *distance* from p to $(\theta(Q), q)$ is the minimum cost of any sequence of flex operations which yields T_p from T_q. The cost of the empty sequence of flex operations (so T_q is already a triple form of p) is zero. The *distance* from p to $\theta(Q)$ is the minimum distance from p to $(\theta(Q), q)$ for any string $q \in L(R)$. □

Definition 15. Given ontology $K = \mathit{extRed}(K)$, graph $G = \mathit{closure}_K(G)$, single-conjunct query Q to which FLEX has been applied, and (Q, G)-matching θ, the *distance* of $\theta(Q)$, denoted $\mathit{dist}(\theta, Q)$, is the minimum distance to $\theta(Q)$ from any semipath p in G. The *answer* of Q on G is a list of pairs $(\theta(vars), \mathit{dist}(\theta, Q))$, where θ is a (Q, G)-matching, ranked in order of non-decreasing distance. The *top-k answer* of Q on G is a list containing the first k tuples in the answer of Q on G. □

Example 11. Consider the conjunct $(\text{‘FL56’}, fn_1.ppn_1.pn_1^-, Y)$ of the query from Example 10. There is only one sequence q of labels denoted by the regular expression of the conjunct, so a triple form of q is:

$$(\text{‘FL56’}, fn_1, X_1), (X_1, ppn_1, X_2), (X_2, pn_1^-, Y)$$

Replacing fn_1 by fn_1^- and inserting ie_1 after pn_1^- gives rise to

$$(\text{‘FL56’}, fn_1^-, X_1), (X_1, ppn_1, X_2), (X_2, pn_1^-, X_3), (X_3, ie_1, Y)$$

with cost $c_s + c_i$. This will match the semipath p from ‘FL56’ to e_1 in the graph of Figure 1, thereby instantiating Y to e_1. Since p cannot be matched by q with fewer flex operations, the distance from p to q is $c_s + c_i$. For example, relaxing pn_1^- to pn^- will also instantiate Y to e_1, but at a cost of $c_s + c_i + c_{r2}$. □

The answer of a single-conjunct query Q to which FLEX has been applied on a graph G can be computed by using an automaton A_Q^K constructed from the approximate automaton A_Q and ontology K which captures both approximation and relaxation with respect to K, as follows:

Step 1: We construct a weighted automaton M_R from R (in which all weights are zero), and then the approximate automaton A_Q, using essentially the same process as described in Section 3.2 (we describe the distinction after the following example).

Step 2: We construct the relaxed automaton A_Q^K from A_Q, applying relaxation using rules 2, 4, 5 and 6 from Figure 6, following the same process as described in Section 3.5.

Example 12. Figure 11 shows the automaton corresponding to the conjunct $(\text{‘FL56’}, fn_1.ppn_1.pn_1^-.Y)$ of the query from Example 10 (only those transitions that contribute to finding answers in the data graph are shown).

The transition with cost c_s results from replacing fn_1 with fn_1^- and the transition with cost c_i results from inserting ie_1. The transition with cost c_{r2}

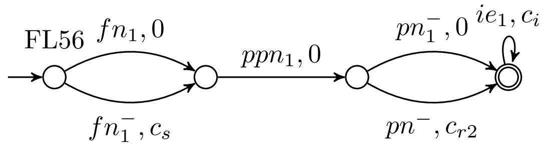

Figure 11: Automaton for conjunct (‘FL56’, $fn_1.ppn_1.pn_1^-, Y$).

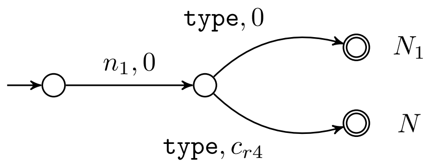

Figure 12: Automaton for conjunct ($Y, n_1.\texttt{type}, N_1$).

results from applying rule (2) from Figure 6 to the transition for pn_1^- and the triple $(pn_1, \texttt{sp}, pn)$ in K.

The automaton for conjunct $(Y, n_1.\texttt{type}, N_1)$ is shown in Figure 12. In this case, the $\texttt{type}$ transition with cost c_{r4} has been added as a result of applying rule (4) to the other $\texttt{type}$ transition using triple $(N_1, \texttt{sc}, N)$ from K, which results in a cloned final state annotated with N. □

The answer to a single-conjunct CRPQ Q is obtained by constructing and traversing the weighted product automaton, H, of A_Q^K with the closure of the data graph $G = (V_G, E_G)$, viewing each node in V_G as both an initial and final state: The process is the same as described in Section 3.2. To evaluate Q on G, if X is a node $v \in V_G$, a shortest path traversal of H is undertaken starting from each state (s_0, v) such that $s_0 \in S_0$. If X is a variable, these shortest path traversals are undertaken for each $v \in V_G$. In each case, the answers to Q on G are given by the bindings for Y found from the final states reached during the traversal of H.

As stated earlier, the automaton A_Q^K is constructed by applying relaxation to the approximate automaton A_Q. However, in contrast to Definition 9 in Section 3.2, the edit operations used in the construction of A_Q are confined to those on labels in $\Sigma \cup \Sigma^-$, i.e., we do not allow edits to the labels in $\{\texttt{type}, \texttt{type}^-\}$. Henceforth in this section, we make this assumption about the approximate automaton A_Q. Allowing edits to the labels in $\{\texttt{type}, \texttt{type}^-\}$ would require multiple rounds of approximation and relaxation to be applied to yield a final automaton, rather than the simple two-step process described above. Determining an upper bound for the number of rounds of approximation/relaxation that would be needed for the construction of the automaton to reach a fixed point is an open problem. This is illustrated in the following example.

Example 13. Suppose we have the query Q with single conjunct $(?X, e.\texttt{type}, c)$, the labels $a, b, e \in \Sigma$, and the triples $(c, \texttt{sc}, c') \in K$ and $(a, \texttt{sp}, b) \in K$. The two

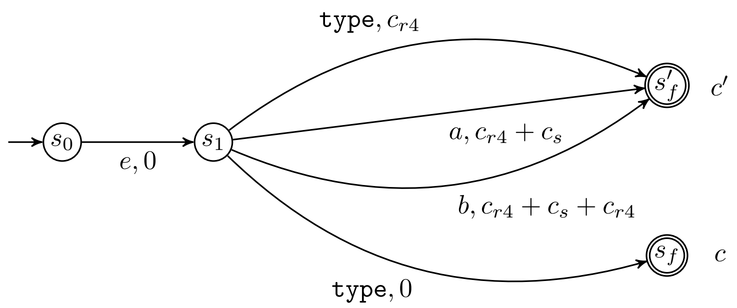

Figure 13: Automaton for $(X, e.\texttt{type}, \text{‘c’})$.

transitions with cost zero in Figure 13 arise from the original query Q. Approximation would add various transitions not shown in Figure 13. Then in the relaxation phase, we apply rule 4(i) which creates a final state s'_f, annotated with c', and a transition $(s_1, \texttt{type}, c_{r4}, s'_f)$. However, the presence of $(s_1, \texttt{type}, c_{r4}, s'_f)$ means we are able to apply more edit operations to the automaton to produce, say, the transition $(s_1, a, c_{r4} + c_s, s'_f)$ by replacing `type` with a. But, by once again applying relaxation operations to the automaton, a new transition $(s_1, b, c_{r4} + c_s + c_{r2}, s'_f)$ induced by rule 2 would be created. The resulting automaton is shown in Figure 13 (only those transitions and states explicitly mentioned are shown).

Thus, for the purposes of the FLEX operator, we do not allow edit operations on `type` or $\texttt{type}^-$ labels. □

In Theorem 2 below, we show that using automaton A_Q^K is sufficient to find all sequences of labels generated by flex operations at distance k from a given query; i.e. that applying a second approximation step after the relaxation step does not (i) yield any additional answers, and (ii) yield any answers previously obtained at cost k, at some cost $j < k$.

We first have the following lemma that shows that for semipath p, sequence of labels $q \in L(R)$, and (Q, G)-matching θ such that the distance from p to $(\theta(Q), q)$ is k, there is a cost-k sequence of flex operations yielding triple form T_p from triple form T_q in which all edit operations precede all relaxation operations.

Lemma 6. *Let Q be a query comprising a single conjunct (X, R, Y), $K =$ $\texttt{extRed}(K)$ be an ontology, $G = (V_G, E_G)$ be the closure of a data graph with respect to K, p be a semipath $(v_0, l_1, \ldots, l_n, v_n)$ in G, θ be a (Q, G)-matching such that $\theta(X) = v_0$ and $\theta(Y) = v_n$, and $q \in L(R)$.*

Let T_p be a triple form for p and T_q a triple form for $(\theta(Q), q)$ such that the distance from p to $(\theta(Q), q)$ is k. There is a sequence of flex operations of cost k, yielding T_p from T_q, in which all the edit operations (applied to symbols in $\Sigma \cup \Sigma^-$) precede all the relaxation operations.

Proof. Let the sequence of flex operations of cost k, yielding T_p from T_q, consist of n flex operations, where $n \leq k$. Let this sequence, $flex_n$, be given by

$T_q = P_0 \preceq P_1 \preceq \cdots \preceq P_n = T_p$, in which the edit and relaxation operations have been applied in an arbitrary order. We need to show that there is an alternative sequence of flex operations, $flex'_n$, also of cost k, yielding T_p from T_q, and comprising the same operations as those in $flex_n$, but where all edit operations (considering only symbols in $\Sigma \cup \Sigma^-$) have been applied prior to all relaxation operations. That is, $flex'_n$ is given by $T_q = P_0 \preceq_A \cdots \preceq_A P_\lambda \preceq_R \cdots \preceq_R P_n = T_p$. The proof proceeds by induction on the number of flex operations applied to T_q.

Basis: For the base case, we assume no flex operations have been applied; hence, $T_q = P_0 = T_p$ and all edits precede all relaxations.

Induction: For the inductive step, suppose that there is an $n \geq 0$ such that, for all $m \leq n$, any sequence of m flex operations of cost k yielding T_p from T_q can be rewritten as a sequence (also yielding T_p from T_q at cost k) in which all edit operations precede all relaxation operations.

Now consider sequence $flex_{n+1}$ of $n+1$ flex operations in which the last is an edit operation. We show that this edit operation can be moved before all the relaxations. If there are no relaxations in the sequence, this is trivial, so assume there is at least one relaxation. By the induction hypothesis, the sequence up to P_n can be rewritten so that relaxations follow edits. Hence $flex_{n+1}$ can be rewritten as ψ given by $T_q = P_0 \preceq \cdots \preceq_R P_n \preceq_A P_{n+1} = T_p$ where $n+1$ flex operations have been applied to the sequence and the $n+1^{th}$ operation is an edit operation, denoted by op_E.

First suppose op_E is applied to a triple pattern present in T_q. Clearly, op_E can be applied to P_0 and hence can precede relaxations.

Next suppose op_E is applied to a triple pattern resulting from an edit operation. As this operation precedes all relaxations, op_E can follow it directly and hence it too can precede all relaxations.

In the case where op_E is an insertion operation, it is straightforward to see that the result also follows, as insertion is not dependent on the presence of any triple pattern and so can be placed before all relaxations.

Thus, we now need to consider the cases in which op_E is a *substitution* or *deletion* operation applied to a triple pattern t of triple form P_n, such that t was the result of having applied some relaxation operation op_R to some triple pattern t' in P_{n-1}. We show below that in fact these are not possible, given that the distance from p to $(\theta(Q), q)$ is k.

We note that t may only be in one of the following forms, depending on which relaxation operation was applied: (i) (W_{m-1}, a, W_m), where W_{m-1} and W_m are variables or constants, (ii) $(W, \texttt{type}, c)$, or (iii) $(c, \texttt{type}, W)$, where W is a variable and c a constant. As edit operations are not applied to the `type` label, we only need consider what happens when applying op_E to P_n if t is of the form (W_{m-1}, a, W_m). By definition, such a t could only have been produced as a result of applying the relaxation operation induced by Rule 2 to t' in P_{n-1}; thus, op_R may only ever be a Rule 2 relaxation operation.

Suppose that $t' = (W_{m-1}, b, W_m)$ and $t = (W_{m-1}, a, W_m)$, where there is a triple $(b, \texttt{sp}, a) \in K$. Let the cost of the sequence ψ be $k = C + c_{r2} + c_e$, where c_{r2} denotes the cost of op_R, c_e denotes the cost of op_E (and is thus either c_s or

c_d) and C is the cost of the remaining operators used in ψ. We now consider the following possibilities for op_E applied to t:

- *Substitution:* Suppose op_E substitutes t in P_n by (W_{m-1}, e, W_m) in P_{n+1}, for some $e \in \Sigma$. However, we could replace op_R by a substitution of b by e in t' in order to obtain (W_{m-1}, e, W_m) and hence T_p at a cost of $C + c_s < k$. This contradicts the assumption that the distance from p to $(\theta(Q), q)$ is k; hence, op_E cannot be substitution.

- *Deletion:* Suppose P_n is given by

  $$\cdots, (W_{m-2}, g, W_{m-1}), t = (W_{m-1}, a, W_m), (W_m, f, W_{m+1}), \cdots$$

  and op_E deletes t to obtain P_{n+1}, given by

  $$\cdots, (W_{m-2}, g, W_{m-1}), (W_{m-1}, f, W_{m+1}), \cdots$$

  However, we could replace the application of op_R on t' (in P_{n-1}) by deleting t' instead, in order to obtain a sequence yielding T_p at cost $C + c_d < k$. This contradicts the assumption that the distance from p to $(\theta(Q), q)$ is k; hence, op_E cannot be deletion.

Thus, we have shown that, for all allowable operations op_E, the sequence ψ can be rewritten to a sequence of the same cost in which all edit operations precede all relaxation operations. $\square$

Theorem 2. *Let Q be a query comprising a single conjunct (X, R, Y) and $K =$ $\texttt{extRed}(K)$ be an ontology. Let A_Q^K be the automaton constructed for Q as described above, where the ϵ-transitions have been removed. Let $G = (V_G, E_G)$ be the closure of a data graph with respect to K, H be the product automaton of G and A_Q^K, p be a semipath $(v_0, l_1, \ldots, l_n, v_n)$ in G, and θ be a (Q, G)-matching such that $\theta(X) = v_0$ and $\theta(Y) = v_n$. The distance from p to $\theta(Q)$ is k if and only if k is the minimum cost of a run for the sequence of labels comprising p from (s_0, v_0) to (s_n, v_n) in H, where s_0 is an initial state and s_n a final state in A_Q^K.*

PROOF. ($\Rightarrow$) By Lemma 6, we know that if the distance from p to $(\theta(Q), q)$ is k, for any $q \in L(R)$, then k is the minimum cost of any sequence of flex operations yielding the triple form T_p from the triple form T_q, where the flex operations have been applied in an analogous manner to the construction of A_Q^K.

The result then follows from the construction of A_Q^K, and by Lemma 3 (as A_Q^K contains A_Q as a subautomaton) and Proposition 3 (as A_Q^K contains M_Q^K as a subautomaton).

($\Leftarrow$) Let $r = ((s_0, v_0), l_1, c_1, (s_1, v_1)), \ldots, ((s_{n-1}, v_{n-1}), l_n, c_n, (s_n, v_n))$ be a minimum cost run of cost k in H for the sequence of labels $l_1, \ldots, l_n$ of p, where s_0 is an initial state and s_n a final state in A_Q^K. From the construction of H from A_Q^K and G, there must be a semipath $p = (v_0, l_1, \ldots, l_n, v_n)$ in G and a minimum cost run (s_0, l_1, c_1, s_1), $\ldots, (s_{n-1}, l_n, c_n, s_n)$ of cost k in A_Q^K, corresponding to T_p, a triple form of p. From the construction of A_Q^K, we also know that the transitions added to those transitions originally present in M_R correspond to flex operations. By definition, we also know that every run in M_R corresponds to an acceptance of a sequence of labels $q \in L(R)$. Let the triple form of such a q be T_q.

But, by Lemma 3 (for edit operations) and Proposition 3 (for relaxation operations), it follows that the minimum cost of any sequence of flex operations yielding T_p from T_q is k. The result then follows straightforwardly from Lemma 6. □

The following proposition shows that if FLEX has been applied to a single-conjunct query Q, the answers on the closure of a graph G can be computed in time that is polynomial in the size of Q, G and the ontology K.

Proposition 6. *Let $K = \texttt{extRed}(K)$ be an ontology, where $K = (V_K, E_K)$, $G = (V_G, E_G)$ be the closure of a data graph with respect to K, and Q be a single-conjunct query using regular expression R over alphabet $\Sigma \cup \Sigma^- \cup \{\texttt{type}, \texttt{type}^-\}$. If FLEX has been applied to Q, the answer of Q on G can be found in time $O(|R|^3|V_G||V_K||E_G||E_K|(|V_K| + |\Sigma|) + |R|^2|V_G|^2|V_K|^2(log|R||V_G||V_K|))$.*

Proof. Let A_Q^K be the automaton constructed from Q and K which captures both approximation and relaxation with respect to K, and H be the product graph constructed from A_Q^K and G. Lemma 3 and Proposition 3 show that traversing H correctly yields all answers to Q. Lemma 4 tells us that A_Q has at most $2|R|$ states and $4|R|^2|\Sigma|$ transitions.

By the construction of A_Q^K from A_Q, the application of rules 4, 5 and 6 results in at most one new state for each class node in V_K being added for any existing state s, where $s \in S_0$ or $s \in S_f$. Hence, we can see that no more than $|V_K|$ new states may be added for each of the original states in A_Q, resulting in at most $2|R||V_K|$ new states in total. Thus, A_Q^K has at most $2|R|(|V_K| + 1)$ states.

Since there are at most $|E_K|$ edges in K with label sp, rule 2 adds at most $4|R|^2|\Sigma||E_K|$ transitions to A_Q^K. Rules 4, 5 and 6 can collectively be applied no more than $|E_K|$ times. Each application results, in the worst case, in $|R|$ transitions being added for each of the $2|R||V_K|$ new, cloned states, giving rise to at most $2|R|^2|V_K||E_K|$ transitions. Thus, overall A_Q^K has at most $2|R|^2(|E_K||V_K| + 2|\Sigma| + |\Sigma||E_K|)$ transitions.

Therefore H has at most $2|R||V_G|(|V_K|+1)$ nodes and $2|R|^2|E_G|(|E_K||V_K|+ 2|\Sigma| + |\Sigma||E_K|)$ edges. If we assume that H is sparse (which is highly likely), then running Dijkstra's algorithm on each node of a graph with node set N and edge set A can be done in time $O(|N||A| + |N|^2 \log |N|)$. So, for graph H, the combined time complexity is $O(|R|^3|V_G||V_K||E_G|(|V_K||E_K|+|\Sigma|+|\Sigma||E_K|)+ |R|^2|V_G|^2|V_K|^2 \log(|R||V_G||V_K|))$ which simplifies to $O(|R|^3|V_G||V_K||E_G||E_K|(|V_K| +|\Sigma|) + |R|^2|V_G|^2|V_K|^2(log|R||V_G||V_K|))$.

□

As a corollary, it is easy to see that the data complexity is $O(|V_G||V_K||E_G||E_K|(|V_K|+|\Sigma|)+|V_G|^2|V_K|^2(\log|V_G||V_K|))$ and the query complexity is $O(|R|^3)$. The space complexity is dominated by the space requirements of H given in the proof above.

The above query evaluation can also be accomplished "on-demand" by incrementally constructing the edges of H as required, thus avoiding precomputation and materialisation of the entire graph H. The incremental evaluation process is the same as described earlier for the cases of approximation and relaxation, considered separately.

It is easy to show that if the ontology K is empty and there are no `type` edges in the data graph G, then FLEX semantics reduce to approximate matching of CRPQs, as in Section 3.2. Similarly, if only ontology-based relaxation is permitted, and the queries over G are limited to be simple conjunctive queries, then this reduces to the query processing semantics with ontology relaxation presented in [11].

4.1. Multi-Conjunct FLEX Queries and Comparison with APPROX/RELAX

A general FLEX query Q is of the form

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

where any conjuncts may in addition have the FLEX operator applied to them. Let θ be a (Q,G)-matching. If conjuncts $i_1,\ldots,i_j$, $j \leq n$, have FLEX applied to them, then the *distance* from θ to Q, $dist(\theta,Q)$, is defined as

$$dist(\theta,(X_{i_1},R_{i_1},Y_{i_1})) + \cdots + dist(\theta,(X_{i_j},R_{i_j},Y_{i_j}))$$

The definitions of *minimum-distance* matching, *answer* of Q on G, and *top-k answer* of Q on G are as in Section 3.7 for multi-conjunct APPROX/RELAX queries.

The evaluation of a multi-conjunct query FLEX Q can be undertaken incrementally in the same way as described in Section 3.7 for multi-conjunct APPROX/RELAX queries, joining the answers arising from the incremental evaluation of each of its conjuncts using a rank-join algorithm.

Considering the complexity of FLEX queries compared to APPROX/RELAX queries, Proposition 2, Proposition 5 and Proposition 6 state the relative complexities of evaluating APPROX, RELAX and FLEX single-conjunct CRPQs, from which it can be observed that FLEX has higher complexity than both APPROX and RELAX. This is to be expected as the automaton A_Q^K used to evaluate a FLEX single-conjunct CRPQ contains all the states and transitions that would appear in the approximate automaton derived directly from M_R for evaluating APPROX (limited to labels in $\Sigma \cup \Sigma^-$) as well as all the states and transitions in the relaxed automaton derived from M_R, for evaluating RELAX.

Considering the expressiveness of FLEX queries compared to APPROX / RELAX queries, it is easy to see that given any graph G, ontology K and CRPQ Q over G that has APPROX or RELAX applied to any of its conjuncts (with APPROX limited to labels in $\Sigma \cup \Sigma^-$) then any answer that is returned at distance k would also be returned, at the same or a lower distance, by a query that has the same conjuncts as Q but with FLEX in place of any occurrence of APPROX or RELAX. Again, this is because any automaton A_Q^K contains all the states and transitions that would appear in the approximate automaton derived from M_R for evaluating APPROX (limited to labels in $\Sigma \cup \Sigma^-$) as well as all the states and transitions that appear in the relaxed automaton derived from M_R. An answer that is returned using FLEX semantics may be at a lower distance than the same answer returned using APPROX/RELAX semantics if an APPROX were replaced by a FLEX in the query and if c_{r2} were less than c_s, because in this case a property relaxation (if applicable) would be less costly than substitution.

Conversely, there exist CRPQs that return answers using FLEX semantics which cannot be returned by any query under APPROX/RELAX semantics, as noted in Example 10.

As a final remark, we would argue that the availability of FLEX does not render APPROX and RELAX redundant. Firstly, FLEX does not apply edit operations to labels in $\{\texttt{type}, \texttt{type}^-\}$, whereas APPROX does. Secondly, the user may only want to consider in some given setting the application of (syntactic) edits — and hence use APPROX, or the application of (semantic) relaxations — and hence use RELAX.

5. Related Work

5.1. Graph-modelled data and graph query languages

The work described in this paper has considered the simple graph data model introduced in Section 2, and a general survey of graph data models can be found in [24]. Likewise, a survey of graph query languages can be found in [25], and we focus here on languages that support regular path queries and on flexible query processing for graph-structured data.

Using regular expressions to specify path queries on graph-structured data has been studied for over 20 years, having been introduced in the languages G, G+ and Graphlog [3, 26, 17]. More recently, conjunctive regular path queries are supported in NAGA [27], SPARQLeR [5], PSPARQL [28] and SPARQL 1.1 [7]. nSPARQL [6] adds nested regular expressions to SPARQL, showing that these are necessary in order to answer queries using the semantics of the RDFS vocabulary by directly traversing the RDF graph, without materialising the closure of the graph.

The Cypher query language of the Neo4j graph database system[14] can express a restricted form of regular path queries: concatenation and disjunction of single edge types, as well as variable length paths in which optional upper and lower bounds may be set; nested regular expressions are not allowed.

[14]http://neo4j.com/

Kasneci *et al.* [29] discuss the importance of finding relationships between nodes in a graph-modelled data structure; the nodes are connected via weighted edges which represent the relationships between the nodes. The authors present the STAR algorithm to determine relationships over large graphs. STAR returns approximations of the original (exact) query, by using the original query as template for an optimal Steiner tree. In addition, if the graph contains information pertaining to the classification hierarchy of the nodes, this is exploited by the algorithm.

Zhou *et al.* [30] explore the idea of achieving query relaxation by using malleable entity-relationship schemas, which contain multiple, overlapping definitions of data structures and attributes, and are intended to capture more fully the diverse semantics of a complex, heterogeneous domain. The approach is grounded in a statistics-based model, in contrast to our work.

5.2. Flexible querying in XML and semi-structured data

Work has been done on relaxing tree pattern queries for XML, e.g. in [31, 32, 33] and more recently in [34]. Liu *et al.* [34] use an XML schema to relax queries; [33] implements relaxation by utilising IR-style techniques; and [32] achieves query relaxation through the removal of conditions within the XPath expression.

Concerning query approximation, [35] considers querying semi-structured data using flexible matchings which allow paths whose edge labels contain those appearing in the query to be matched. Such semantics can be captured by transposition and insertion edit operations on edge labels. More generally, Grahne and Thomo [8, 9] explore approximate matching of single-conjunct regular path queries, using a weighted regular transducer to perform transformations to RPQs for approximately matching semistructured data. In other work [36], Grahne and Thomo introduce *preferential* RPQs where users can specify the relative importance of symbols appearing in the query by annotating them with weights.

Buratti *et al.* [37] discuss use of the notion of a cost-based edit distance to transform one path into another within an `XQuery FullText` expression. The answers are ranked according to a 'satisfaction' ratio, computed using a statistical scoring method for numeric values and similarity metrics calculated from an ontology for string values.

Within the context of combined XML and relational data, Yu *et al.* [38] discuss a flexible query model using the notion of a *schema summary*. This is a condensed description of the full database or XML document schema. The schema summary is used to construct a new model called the *Meaningful Summary Query* (MSQ). Such a query requires the user only to have knowledge of the schema summary. From there, the MSQ query is evaluated by making use of schema-matching semantics.

5.3. Flexible querying in SPARQL and RDF

There have been several proposals that use similarity measures to retrieve additional relevant answers to semantic web queries. For example, in iSPARQL [39] similarity measures are applied to resources (rather than the paths connecting resources), e.g. by using the edit distance between the names of the resources; in [40] similarity functions are applied to constants such as strings and numeric values; in [41] a structural similarity approach is proposed that exploits the graph structure of the data; and in [42, 43, 44] ontology-driven similarity measures are developed, using the RDFS ontology to retrieve additional answers and assign a score to them.

Elbassuoni *et al.* [45] propose an extension to SPARQL with keyword search capabilities. They also discuss the relaxation of triple patterns by replacing constants with variables. The larger the number of relaxations, the lower the rank of any answers arising from the (relaxed) query. The relaxation of keywords is achieved by employing IR-based techniques, which differs from our ontology-based approach to query relaxation.

Zauner *et al.* [46] present RPL (RDF Path Language), which allows conditional regular expressions to be expressed over the nodes and edges appearing on paths within RDF data. In contrast to our approach, there is no query relaxation or approximation.

Mandreoli *et al.* [47] allow edges in a query to match paths in a graph that have been semantically related, e.g. using RDFS. This requires the degree of relationship between pairs of nodes in the graph to have been established beforehand, whereas our approach obtains this information automatically from the accompanying ontology. Furthermore, our model allows for regular expression matching, which is not provided in [47].

5.4. User preferences and domain-specific systems

Within the context of human resources and employee recruitment, Mochol *et al.* [48] implement query relaxation by applying query-rewriting techniques to SeRQL queries posed over RDF data, in which specific terms are replaced with more general terms. The relationships between terms are derived from subclass-superclass relationships within an accompanying ontology.

User and domain preferences play an integral part in the query relaxation techniques described in [49, 50], in which aspects such as rewriting rules, preferences, and user-relevant dimensions are able to be configured and subsequently applied in order to yield the best results according to what the user deems important within a particular context. Meng *et al.* [51] explore query relaxation by utilising the personal preferences of the user in order to reduce the constraints of the original query. The contextual preferences of the user are obtained by using association-rule mining over the log of past queries. Flexible querying of RDF using preferences expressed as fuzzy sets is investigated by Buche *et al.* [52].

Meng *et al.* [53] propose the use of conjunctive selections on attributes of form-based web data, where value constraints can be relaxed according to their perceived importance to the user.

5.5. Graph matching

Zhang *et al.* [54] present the SAPPER model, which sets out to find all instances of a query graph within a large graph database. The model uses *edge edit distance* in order to return both exact and approximated subgraphs matching the original query, possibly containing missing edges (provided the edge edit distance does not exceed a certain threshold value). The TALE method of Tiang *et al.* [55] uses a heuristic algorithm to match important nodes within the graph first, after which the search is extended. Neither of these approaches uses queries with regular expressions, and neither implements any notion of query relaxation. Moreover, because TALE uses a heuristic algorithm, it does not guarantee to find all or even the best matches, unlike SAPPER and our framework in which all answers at a particular distance will always be returned.

Zhu *et al.* [56] also discuss approximate subgraph matching in large graph databases. They use an index, augmented with structural and attribute information, in order to accelerate the process of finding the best-matching answer graph to a query. Cheng *et al.* [57] propose a method based on indexes to locate and retrieve subgraphs in a graph database consisting of a large number of small graphs, which is especially prevalent in bioinformatics. Such indexing techniques could potentially be applied in our context of approximate/relaxed evaluation of CRPQs.

Varadarajan *et al.* [58] propose the GID framework for facilitating user queries on hyperlinked data, where answers are ranked according to a user's input criteria. In GID a query is a series of user-defined 'filters'. In contrast to our approach, this does not allow for path query approximation or relaxation.

Zou *et al.* [59] present gStore, which stores RDF data as a large graph rather than in an RDF repository. SPARQL queries issued to this system are transformed into subgraph matching queries. A potential interface to this system from our own Omega prototype [60] is an area of future investigation.

Other work on approximate graph matching includes [61] in which regular expressions are added as edge constraints on the graph patterns to be matched; [62] in which in top-k matches may be found without computing the entire graph using relevance functions based on factors such as social impact and distance; and [63] which defines a set of criteria preserving the topology of massive graphs to rectify the problem caused by pattern matching over a data graph whose structure differs greatly from the pattern, thereby resulting in matches which may be difficult to comprehend and analyse. This work too has synergies with our approach to flexible querying processing, since the techniques proposed could potentially be applied to the evaluation of approximated/relaxed CRPQs.

5.6. Our previous work

The work in [11] introduced a RELAX clause for querying RDF/S data which can be applied to those triple patterns of a query that the user would like to be matched flexibly. These triple patterns are successively made more general so that the overall query returns successively more general answers, at increasing costs from the exact form of the query (using the entailment rules of Figure 4). An essential aspect of this approach, which distinguishes it from earlier work on query relaxation, is that the answers to a query are ranked based on how closely they satisfy the original query.

The work in [10] discusses how approximate answers can be computed for CRPQs, based on edit operations such as insertions, deletions, inversions, substitutions and transpositions of edge labels being applied to a semipath.

The work in [12] combines within one querying framework both approximation (APPROX) and relaxation (RELAX) and applies it to CRPQs. In [64] we describe the implementation of a prototype system that supports this functionality. In [60] we describe a more mature implementation of these flexible querying capabilities.

Approximation and relaxation of CRPQs can be applied to the more pragmatic setting of the SPARQL 1.1 language [7] — specifically, to its property path queries — and in [65, 66] a query rewriting approach is presented for implementing such extensions.

6. Conclusions

We have considered approximation and relaxation of conjunctive regular path queries and have shown how these can be combined to support users in flexible querying of complex, irregular graph-structured data. Using the APPROX and RELAX operators, users can specify approximations and relaxations to be applied to selected conjuncts of their original query, as well as the relative costs of these. Also, using the FLEX operator, users can specify that both approximation and relaxation should be applied to a query conjunct. In all cases, query answers are returned incrementally, in polynomial time, ranked in order of increasing distance from the user's original query.

The paper makes two major contributions. Firstly, we have extended the work in [12] on CRPQ approximation and relaxation by giving full details of the algorithms and full proofs of the theoretical results. Secondly, we have proposed merging the APPROX and RELAX query operators from [12] into an integrated FLEX operator. This allows easier querying of complex heterogeneous data sets for users as they do not have to be aware of the ontology structure and do not have to identify explicitly which parts of their overall query may be amenable to relaxation. It may also allow more query results to be returned, as there exist CRPQs that return answers using FLEX semantics which cannot be returned by any CRPQ using APPROX/RELAX semantics. Along the way, we have introduced the notion of the 'triple form' of a sequence of edge labels, giving for the first time a uniform framework for handling both query approximation and query relaxation for CRPQs.

This paper has concentrated on theoretical aspects of approximation and relaxation of CRPQs. In practice, a graphical front-end could allow users to pose queries using forms, keywords, or even natural language, which the system would then translate into CRPQs (c.f. [64]). The user would select which approximation and relaxation operations, from the full range of operations supported by our framework, should be applied by the system to parts of their queries in order to approximate or to relax them. The user could select which parts of the ontology should be used for the relaxation operations (rather than the whole ontology) and which labels for the edit operations (rather then the full set of edge labels in the data graph). For the substitution operation, it would be straightforward to extend our framework to support finer-grained ranking of the substitution of one edge label by another, e.g. through the application of lexical or semantic similarity measures on edge labels (rather than assuming the same cost for all substitutions). Finally, in order to help users interpret answers to their queries, the system could provide a trace of the successive edits/relaxations applied by the system to the original query, and the answers arising from each modified query. Showing the sequence of changes by which the original query was approximated or relaxed could help the user decide whether the answers being returned are relevant to them.

As mentioned in Section 5.6, descriptions of implementations of APPROX and RELAX (but not FLEX) appear in [60, 65, 66]. In [60] we describe an NFA-based implementation of these flexible querying capabilities. We also report on a performance study of single-conjunct regular path queries, with approximation and relaxation, on several datasets sourced from the L4All system [64] and from YAGO [27]. This study showed that most of the approximated and relaxed queries executed quickly on all datasets. However, some of the approximated queries on YAGO either failed to terminate or did not complete in a reasonable amount of time. This was mainly due to a large number of intermediate results being generated when the `Succ` function returns a large number of transitions, which are then converted into tuples in `getNext` and added to $\texttt{queue}_R$. Two optimisations are explored in [60] for such queries, enabling several (but not all) of these APPROX queries to execute faster. In [65, 66] an alternative implementation approach based on query rewriting is presented, in the more pragmatic setting of SPARQL 1.1. Performance studies of multi-conjunct regular path queries, with approximation and relaxation, are undertaken on several datasets sourced from LUBM[15] and from YAGO. These studies show good performance for most of the queries trialled, apart from when Kleene closure ('*') and the wildcard symbol('_') appear within the same regular path query. Ongoing work includes the development of optimisation techniques to improve query evaluation in these prototype systems, drawing for example from recent work in [67, 68, 69, 70, 71, 72, 73, 14]. The FLEX operator is not yet supported by any implementation, and extending these prototypes to support it too is an area of future work.

Beyond centralised architectures, the area of distributed graph data processing is increasing in importance, with the aim of handling larger volumes of graph data than can be handled on a single server and to achieve horizontal scalability[16] [74, 75, 76, 77, 78]. Using distributed graph query processing techniques to enable flexible querying of larger volumes of complex, irregular graph-structured data is an important area of future work.

Finally, deeper investigation of the relationships between FLEX, APPROX and RELAX would be interesting, for example to determine if there are char-

[15]http://swat.cse.lehigh.edu/projects/lubm/

[16]http://thinkaurelius.com, https://github.com/apache/giraph

acteristics of a multi-conjunct CRPQ query, data graph or ontology that mean that FLEX will always return more results than any combination of APPROX or RELAX; and to investigate further integration of the automaton-based query evaluation approaches for APPROX, RELAX and FLEX that we have presented here, into a higher-level abstract framework of which APPROX, RELAX and FLEX are specific instances.

Acknowledgements: We thank the reviewers of this paper for their detailed and insightful comments, which helped to improve the paper substantially.

References

[1] T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, D. Sheets, Tabulator: Exploring and analyzing linked data on the semantic web, in: Proc. 3rd Int. Semantic Web User Interaction Workshop, 2006.

[2] A. Singhal, Introducing the Knowledge Graph: things, not strings, in: Official Google Blog, https://googleblog.blogspot.co.uk/2012/05/introducing-knowledge-graph-things-not.html, 2012.

[3] I. F. Cruz, A. O. Mendelzon, P. T. Wood, A graphical query language supporting recursion, in: Proc. ACM SIGMOD, 1987, pp. 323–330.

[4] D. Calvanese, G. D. Giacomo, M. Lenzerini, M. Y. Vardi, Containment of conjunctive regular path queries with inverse, in: Proc. 7th Int. Conf. on Principles of Knowledge Representation and Reasoning, 2000, pp. 176–185.

[5] K. Kochut, M. Janik, SPARQLeR: Extended SPARQL for semantic association discovery, in: Proc. 4th European Semantic Web Conference, 2007, pp. 145–159.

[6] J. Pérez, M. Arenas, C. Gutierrez, nSPARQL: A navigational language for RDF, in: Proc. 7th Int. Semantic Web Conference, 2008, pp. 66–81.

[7] A. Seaborne, S. Harris(Editors), Sparql 1.1 query language, w3c recommendation 21 march 2013.
URL `http://www.w3.org/TR/sparql11-query/`

[8] G. Grahne, A. Thomo, Approximate reasoning in semi-structured databases, in: Proc. 8th Int. Workshop on Knowledge Representation meets Databases, 2001.

[9] G. Grahne, A. Thomo, Regular path queries under approximate semantics, Ann. Math. Artif. Intell. 46 (1-2) (2006) 165–190.

[10] C. A. Hurtado, A. Poulovassilis, P. T. Wood, Ranking approximate answers to semantic web queries, in: Proc. 6th European Semantic Web Conference, 2009, pp. 263–277.

[11] C. A. Hurtado, A. Poulovassilis, P. T. Wood, Query relaxation in RDF, Journal on Data Semantics X (2008) 31–61.

[12] A. Poulovassilis, P. T. Wood, Combining approximation and relaxation in semantic web path queries, in: Proc. 9th Int. Semantic Web Conference, 2010, pp. 631–646.

[13] T. Heath, M. Hausenblas, C. Bizer, R. Cyganiak, How to publish linked data on the web (tutorial), in: Proc. 7th Int. Semantic Web Conference, 2008.

[14] G. H. Fletcher, J. Peters, A. Poulovassilis, Efficient regular path query evaluation using path indexes, in: 19th Int. Conf. on Extending Database Technology, 2016, pp. 636–639.

[15] I. F. Ilyas, W. G. Aref, A. K. Elmagarmid, Supporting top-k join queries in relational databases, The VLDB Journal 13 (2004) 207–221.

[16] A. V. Aho, J. E. Hopcroft, J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

[17] A. O. Mendelzon, P. T. Wood, Finding regular simple paths in graph databases, SIAM J. Comput. 24 (6) (1995) 1235–1258.

[18] M. Droste, W. Kuich, H. Vogler, Handbook of Weighted Automata, 1st Edition, Springer Publishing Company, Incorporated, 2009.

[19] R. A. Wagner, M. J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.

[20] C. Gutierrez, C. Hurtado, A. O. Mendelzon, J. Perez, Foundations of semantic web databases, J. Comput. Syst. Sci. 77 (3) (2011) 520–541.

[21] P. Hayes, P. F. Patel-Schneider (Eds.), RDF 1.1 Semantics, W3C Recommendation, 2014.

[22] J. Finger, N. Polyzotis, Robust and efficient algorithms for rank join evaluation, in: Proc. ACM SIGMOD, 2009, pp. 415–428.

[23] G. Gottlob, N. Leone, F. Scarcello, The complexity of acyclic conjunctive queries, J. ACM 43 (3) (2001) 431–498.

[24] R. Angles, C. Gutierrez, Survey of graph database models, ACM Comput. Surv. 40 (1) (2008) 1:1–1:39.

[25] P. T. Wood, Query languages for graph databases, SIGMOD Rec. 41 (1) (2012) 50–60.

[26] M. P. Consens, A. O. Mendelzon, Expressing structural hypertext queries in GraphLog, in: Proc. 2nd Annual ACM Conference on Hypertext, 1989, pp. 269–292.

[27] G. Kasneci, M. Ramanath, F. Suchanek, G. Weikum, The YAGO-NAGA approach to knowledge discovery, SIGMOD Rec. 37 (4) (2009) 41–47.

[28] F. Alkhateeb, J.-F. Baget, J. Euzenat, Extending SPARQL with regular expression patterns (for querying RDF), J. Web Sem. 7 (2) (2009) 57–73.

[29] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, G. Weikum, STAR: Steiner-Tree approximation in relationship graphs, in: Proc. 25th Int. Conf. on Data Engineering, 2009, pp. 868–879.

[30] X. Zhou, J. Gaugaz, W.-T. Balke, W. Nejdl, Query relaxation using malleable schemas, in: Proc. ACM SIGMOD Int. Conf. on Management of Data, 2007, pp. 545–556.

[31] S. Amer-Yahia, S. Cho, D. Srivastava, Tree pattern relaxation, in: Proc. 8th Int. Conf. on Extending Database Technology, 2002, pp. 496–513.

[32] S. Amer-Yahia, L. V. S. Lakshmanan, S. Pandit, FleXPath: Flexible structure and full-text querying for XML, in: SIGMOD Conference, 2004, pp. 83–94.

[33] M. Theobald, R. Schenkel, G. Weikum, An efficient and versatile query engine for TopX search, in: Proc. 31st Int. Conf. on Very Large Data Bases, 2005, pp. 625–636.

[34] C. Liu, J. Li, J. X. Yu, R. Zhou, Adaptive relaxation for querying heterogeneous XML data sources, Information Systems 35 (6) (2010) 688–707.

[35] Y. Kanza, Y. Sagiv, Flexible queries over semistructured data, in: Proc. 20th ACM Symposium on Principles of Database Systems, 2001, pp. 40–51.

[36] G. Grahne, A. Thomo, W. W. Wadge, Preferentially annotated regular path queries, in: Proc. 11th Int. Conf. on Database Theory, 2007, pp. 314–328.

[37] G. Buratti, D. Montesi, Ranking for approximated XQuery Full-Text queries, in: Proc. 25th British National Conf. on Databases, 2008, pp. 165–176.

[38] C. Yu, H. V. Jagadish, Querying complex structured databases, in: Proc. 33rd Int. Conf. on Very Large Data Bases, 2007, pp. 1010–1021.

[39] C. Kiefer, A. Bernstein, M. Stocker, The fundamentals of iSPARQL: A virtual triple approach for similarity-based semantic web tasks, in: Proc. 6th Int. Semantic Web Conference, 2007, pp. 295–309.

[40] A. Hogan, M. Mellotte, G. Powell, D. Stampouli, Towards fuzzy query-relaxation for RDF, in: Proc. 9th Int. Conf. on The Semantic Web, 2012, pp. 687–702.

[41] R. De Virgilio, A. Maccioni, R. Torlone, A similarity measure for approximate querying over RDF data, in: Proc. of the Joint EDBT/ICDT 2013 Workshops, 2013, pp. 205–213.

[42] H. Huang, C. Liu, X. Zhou, Computing relaxed answers on RDF databases, in: Proc. 9th Int. Conf. on Web Information Systems Engineering, 2008, pp. 163–175.

[43] H. Huang, C. Liu, Query relaxation for star queries on RDF, in: Proc. 11th Int. Conf. on Web Information Systems Engineering, 2010, pp. 376–389.

[44] B. R. K. Reddy, P. S. Kumar, Efficient approximate SPARQL querying of web of linked data, in: Proc. 6th Int. Workshop on Uncertainty Reasoning for the Semantic Web, Vol. 654, 2010, pp. 37–48.

[45] S. Elbassuoni, M. Ramanath, R. Schenkel, M. Sydow, G. Weikum, Language-model-based ranking for queries on RDF-graphs, in: Proc. 18th ACM Conf. on Information and Knowledge Management, 2009, pp. 977–986.

[46] H. Zauner, B. Linse, T. Furche, F. Bry, A RPL through RDF: expressive navigation in RDF graphs, in: Proc. 4th Int. Conf. on Web Reasoning and Rule Systems, 2010, pp. 251–257.

[47] F. Mandreoli, R. Martoglia, G. Villani, W. Penzo, Flexible query answering on graph-modeled data, in: 12th Int. Conf. on Extending Database Technology, 2009, pp. 216–227.

[48] M. Mochol, A. Jentzsch, H. Wache, Suitable employees wanted? Find them with semantic techniques. Paper presented at the Workshop on Making Semantics Work For Business (2007).

[49] P. Dolog, H. Stuckenschmidt, H. Wache, Robust query processing for personalized information access on the semantic web, in: Proc. 7th Int. Conf. on Flexible Query Answering Systems, 2006, pp. 343–355.

[50] P. Dolog, H. Stuckenschmidt, H. Wache, J. Diederich, Relaxing RDF queries based on user and domain preferences, J. Intell. Inf. Syst. 33 (3) (2009) 239–260.

[51] X. Meng, Z. M. Ma, L. Yan, Providing flexible queries over web databases, in: Proc. 12th Int. Conf. on Knowledge-Based and Intelligent Information & Engineering Systems, 2008, pp. 601–606.

[52] P. Buche, J. Dibie-Barthélemy, H. Chebil, Flexible SPARQL querying of web data tables driven by an ontology, in: Proc. 8th Int. Conf. on Flexible Query Answering Systems, 2009, pp. 345–357.

[53] X. Meng, Z. M. Ma, L. Yan, Answering approximate queries over autonomous web databases, in: Proc. 18th Int. World Wide Web Conference, 2009, pp. 1021–1030.

[54] S. Zhang, J. Yang, W. Jin, SAPPER: Subgraph indexing and approximate matching in large graphs, Proc. VLDB Endow. 3 (1) (2010) 1185–1194.

[55] Y. Tian, J. M. Patel, TALE: A tool for approximate large graph matching, in: Proc. 24th Int. Conf. on Data Engineering, 2008, pp. 963–972.

[56] L. Zhu, W. K. Ng, J. Cheng, Structure and attribute index for approximate graph matching in large graphs, Inf. Syst. 36 (6) (2011) 958–972.

[57] J. Cheng, Y. Ke, W. Ng, Efficient query processing on graph databases, ACM Trans. Database Syst. 34 (1) (2009) 1–48.

[58] R. Varadarajan, V. Hristidis, L. Raschid, M. Vidal, L. D. Ibáñez, H. Rodríguez-Drumond, Flexible and efficient querying and ranking on hyperlinked data sources, in: Proc. 12th Int. Conf. on Extending Database Technology, 2009, pp. 553–564.

[59] L. Zou, J. Mo, L. Chen, M. T. Özsu, D. Zhao, gStore: Answering SPARQL queries via subgraph matching, Proc. VLDB Endow. 4 (8) (2011) 482–493.

[60] P. Selmer, A. Poulovassilis, P. T. Wood, Implementing flexible operators for regular path queries, in: Proc. 4th Int. Workshop on Querying Graph Structured Data, 2015, pp. 149–156.

[61] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, Adding regular expressions to graph reachability and pattern queries, in: Proc. 27th Int. Conf. on Data Engineering, 2011, pp. 39–50.

[62] W. Fan, X. Wang, Y. Wu, Diversified top-k graph pattern matching, Proc. VLDB Endow. 6 (13) (2013) 1510–1521.

[63] S. Ma, Y. Cao, W. Fan, J. Huai, T. Wo, Strong simulation: Capturing topology in graph pattern matching, ACM Trans. Database Syst. 39 (1) (2014) 4:1–4:46.

[64] A. Poulovassilis, P. Selmer, P. T. Wood, Flexible querying of lifelong learner metadata, IEEE Transactions on Learning Technologies 5 (2) (2012) 117–129.

[65] A. Calì, R. Frosini, A. Poulovassilis, P. T. Wood, Flexible querying for SPARQL, in: On the Move to Meaningful Internet Systems, 2014, pp. 473–490.

[66] R. Frosini, A. Cali, A. Poulovassilis, P. T. Wood, Flexible query processing for SPARQL, Semantic Web Journal, In press.

[67] R. Castillo, C. Rothe, U. Leser, RDFMatView: Indexing RDF data using materialized SPARQL queries, in: Proc. 6th Int. Workshop on Scalable Semantic Web Knowledge Base Systems, 2010.

[68] O. Hartig, R. Heese, The SPARQL query graph model for query optimization, in: Proc. of the 4th European Conf. on The Semantic Web, 2007, pp. 564–578.

[69] Z. Kaoudi, K. Kyzirakos, M. Koubarakis, SPARQL query optimization on top of DHTs, in: Proc. 9th Int. Semantic Web Conference, 2010, pp. 418–435.

[70] A. Koschmieder, U. Leser, Regular path queries on large graphs, in: Proc. 24th Int. Conf. on Scientific and Statistical Database Management, 2012, pp. 177–194.

[71] A. Loizou, P. T. Groth, On the formulation of performant SPARQL queries, CoRR abs/1304.0567.

[72] X. Wang, X. Ding, A. K. H. Tung, S. Ying, H. Jin, An efficient graph indexing method, in: Proc. 28th Int. Conf. on Data Engineering, 2012, pp. 210–221.

[73] N. Yakovets, P. Godfrey, J. Gryz, Towards query optimization for SPARQL property paths (arXiv:1504.08262, 30th April 2015).

[74] U. Kang, C. E. Tsourakakis, C. Faloutsos, PEGASUS: A peta-scale graph mining system implementation and observations, in: Proc. 9th Int. Conf. on Data Mining, 2009, pp. 229–238.

[75] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: a system for large-scale graph processing, in: Proc. ACM SIGMOD Int. Conf. on Management of data, 2010, pp. 135–146.

[76] S. Salihoglu, J. Widom, GPS: A graph processing system, in: Proc. 25th Int. Conf. on Scientific and Statistical Database Management, 2013, pp. 22:1–22:12.

[77] M. Sarwat, S. Elnikety, Y. He, G. Kliot, Horton: Online query execution engine for large distributed graphs, in: Proc. 28th Int. Conf. on Data Engineering, 2012, pp. 1289–1292.

[78] K. Zeng, J. Yang, H. Wang, B. Shao, Z. Wang, A distributed graph engine for web scale RDF data, Proc. VLDB Endow. 6 (4) (2013) 265–276.