Flexible Query Answering on Graph-modeled Data*

Federica Mandreoli, Riccardo Martoglia,
Giorgio Villani
DII
University of Modena and Reggio Emilia, Italy
{firstname.lastname}@unimore.it

Wilma Penzo
DEIS
University of Bologna, Italy
wilma.penzo@unibo.it

ABSTRACT

The largeness and the heterogeneity of most graph-modeled datasets in several database application areas make the query process a real challenge because of the lack of a complete knowledge of the vocabulary used, as well as of the information about the structural relationships between the data.

To overcome these problems, flexible query answering capabilities are an essential need. In this paper we present a general model for supporting approximate queries on graph-modeled data. Approximation is both on the vocabularies and the structure. The model is general in that it is not bound to a specific graph data model, rather it gracefully accommodates labeled directed/undirected data graphs with labeled/unlabeled edges. The query answering principles underlying the model are not compelled to a specific data graph, instead they are founded on properties inferable from the data model the data graph conforms to. We complement the work with a ranking model to deal with data approximations and with an efficient top-k retrieval algorithm which smartly accesses ad-hoc data structures and generates the most promising answers in an order correlated with the ranking measures. Experimental results prove the good effectiveness and efficiency of our proposal on different real world datasets.

1. INTRODUCTION

Graph-based data models have recently gained much popularity as powerful means for data representation in several database application areas, e.g., in biological databases [8], Web-scattered data [12], personal information management (PIM) systems [22], dataspaces [5]. In most of these domains, largeness and heterogeneity are common features which characterize the datasets. These peculiarities make it impractical to exactly query the data due to the lack of a complete knowledge of the vocabulary used, as well as of the information about how data is organized. One of the most elusive goals of research is thus establishing flexible query mechanisms which allow users to easily query graph-based data and to get useful results.

One way to achieve flexibility in query formulation is to adopt a keyword-based query model [1, 3, 4, 9, 10, 11, 14, 19]. This solution has the merit of eliminating structures in the query, thus lightening the user from the burden of knowing the relationships occurring between the data. Furthermore, it easily applies to heterogeneous scenarios where multiple schemas coexist.

On the other hand, a keyword-based approach suffers from an inherently limited capability in the semantics it can express. We are all familiar with how difficult is to translate a complex request in a set of keywords and most of all to get precise answers from this means. For instance, let us consider the graph-based data in Fig. 1, which represents a very small portion of the DBLP graph[1] in a RDF-like style, and let us look at Query 1. By following a simple keyword search query model [9, 14, 16], and assuming for simplicity to disregard the 'Sivastava' misspelling and the condition on the year of publication, a user would translate Query 1 into the set of terms `Q1={title, paper, author, Srivastava}`. The main drawback of this kind of approach is that terms are allowed to occur anywhere in the data. This means that, for instance, papers having `Srivastava` as reviewer would appear in the result, even if they obviously are not relevant for the query.

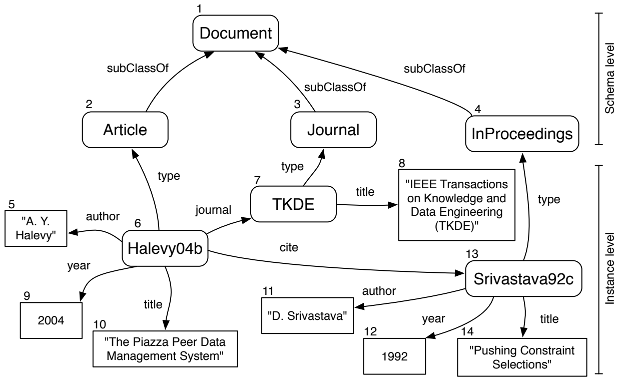
From the above example it is evident the need of supporting semantic query capabilities which go beyond keyword-based search when searching for knowledge [12]. Essentially, the user should be enabled to include varying degrees of structure in her queries, so that she can better specify her own needs according to the partial knowledge of the schema she may have. Furthermore, it is of fundamental importance to get *meaningful* answers, i.e., answers made up of data related in a significant way.

These new demands pose new challenges as to the definition of appropriate and effective models for query specification and query answering, as well as of efficient algorithms and data structures to support their applicability.

Only few recent works have dealt with these issues so far [5, 8, 12, 18, 21]. However, all of them are either tailored to specific domains or they focus on specific aspects of the problem. For instance, the papers [5, 8] propose a partial solution to the lack of expressiveness of the keyword-based approach by allowing the contextualization of value terms (e.g., `Srivastava`) into label terms (e.g., `author`). However, they do

*This work is partially supported by the Italian Council co-funded project NeP4B

[1] Available at `http://sw.deri.org`



Query 1: The title of the papers authored by someone whose name sounds like 'Sivastava' published before 2005.
Query 2: The documents related in some way to the 'Piazza' paper published in the TKDE journal.

Figure 1: Reference graph and two query samples

not allow the specification of relationships between terms. Thus, for instance, Query 2 in our reference example can not be expressed. NAGA [12] and TALE [18], instead, adopt a graph-based query language which enables the formulation of queries with additional semantic information, namely the one expressed through the specification of links (i.e., relationships) between terms. Their answer model is then based on approximate graph matching. However, in complex domains, approximation techniques like those adopted in [18] are not adequate, since they are limited to syntactic considerations on data linkage, completely disregarding the semantics underlying the connections between the data. This aspect has important implications on the meaningfulness of answers, since, for instance in biological databases, entities can be related in several different ways, each one representing a different biological phenomenon [8]. In this respect, NAGA [12] presents a more powerful query language, but it does not delve into details as to the efficiency of the answering process. [21] presents a flexible query model which is targeted for relational and XML data. Thus, it does not investigate the problem of handling labeled edges.

Finally, as to the problem of querying heterogeneous data, up to now most of the efforts have been spent on the problem of approximating the structure of a query [2, 18, 20], often giving little attention to support vagueness on the vocabulary used. Rather, it has been proved that label ambiguity is a very frequent issue, because people hardly choose the same term for a single well-known object [7]. For this purpose, some works [5, 22] adopt term expansion techniques, and follow a query relaxation approach. However, they do not explicitly deal with vague specification on data links.

In this paper we address the problem of *approximate query answering* on graph-modeled data, namely finding data subgraphs which are *similar* to a query graph. Our approach is semantics-driven in that we admit only approximations which lead to *semantically meaningful* results.

Our contributions are as follows:

- we present a flexible model for supporting approximate queries on graph-modeled data. The data model is general in that it is not bound to a specific graph data model, rather it gracefully accommodates several data models from other approaches, namely supporting labeled directed/undirected data graphs with labeled/unlabeled edges. This results in a general framework which flexibly satisfies a wide variety of application-specific needs (Section 2);
- we support approximate queries in a two-fold fashion: 1) by considering node/edge label mismatches; 2) through the structural relaxation of relationships between data. As for the latter point, a key contribution is the introduction of the notion of *Semantic Relatedness* relation for nodes in a data graph as a fundamental benchmark for the application of *semantically meaningful* relaxations only (Section 3);
- we complement the query answering model with a general ranking model which identifies some interesting properties of scoring functions useful to support efficiently the evaluation of approximations. Furthermore, the model adopts a distinct-node set semantics which privileges the most compact answers and avoids to overwhelm the user with quasi-redundant results (Section 4);
- we present an instantiation of the Semantic Relatedness relation for a RDF-like data model. This is generated by means of a set of rewriting rules on graph edges which are not compelled to a specific data graph, rather they rely on properties of the data model the data graph conforms to (Section 5);
- we provide an efficient top-k retrieval algorithm which adopts the principles of the Threshold Algorithm (TA) [6] and smartly accesses ad-hoc data structures to generate the top-k answers (Section 6);
- we present an extensive experimental evaluation which proves the effectiveness and the efficiency of our proposal on different real world datasets (Section 7).

Finally, we compare our proposal with the literature and provide concluding remarks in Section 8.

2. THE DATA MODEL

Graphs provide a natural way to represent a wide range of data for different applications, from relational information to XML documents, from biological databases to dataspace systems, only to mention a few.

In this paper we deal with the problem of flexible query answering on graph-modeled data. To this end, we do not want to limit the generality of the approach to a specific data model. Instead, the data model we adopt is a general model where data is represented as a connected graph allowing parallel edges (also known as multigraphs) and node and edge labels. In this way, we are able to cover most graph-based data models like the widely adopted standard RDF and interesting proposals recently introduced in the literature for various purposes (e.g. [5, 8, 12, 14]).

A data graph essentially represents a portion of the real world through entities, values, and relationships between them. In the following, we denote as L the set of all possible labels partitioned in literal values L_L (e.g. `2004`, `"A. Y. Halevy"`) and concept labels L_C (e.g. `year`, `author`, `cite`). Moreover, L_C also includes the empty label ϵ which is assigned to unlabeled nodes and edges.

Definition 1 (Data Graph). *Data is represented as a connected directed labeled multi-graph* $\mathcal{G} = (N, E, L_N, L_E)$

where N is a set of nodes, $E \subseteq N \times N$ is a multiset of directed edges, $L_N \subseteq L$ and $L_E \subseteq L_C$ are sets of node and edge labels, respectively. Each node $n \in N$ is assigned a label $\lambda(n) \in L_N$ and each edge $e \in E$ is assigned a label $\lambda(e) \in L_E$. Nodes having labels in L_L are called value nodes whereas nodes with labels in L_C are called entity nodes.

It is worth noting that the support to parallel edges, which not all models include (e.g. [18]), is instead fundamental in many contexts such as RDF and OWL based data as well as dataspaces, where the edge semantics depends not only from the involved nodes but also from the edge properties.

The graph which will be used as reference example in the following is shown in Fig. 1, and it describes the relationship between two publications[2]. The graph conforms to a data model which most of the RDF-like data can be translated into. The model distinguishes between the instance level and the schema level. More precisely, it specializes the data graph model introduced in Def. 1 in the following way: an entity node, depicted as a rounded rectangle, is an RDF class, a class instance or any other resource, while a value node, depicted as a rectangle, is an RDF literal. For simplicity, in Fig. 1 we show only a small portion of a potentially large and heterogeneous data graph which, as discussed in the Introduction, in most domains represents the collection of several different data sources. Thus, for instance, scientific papers are possibly described by concepts such as `Paper`, `Article`, `Publication`, `InProceedings`, aso.

Given a data graph $\mathcal{G}$, we introduce the notion of path which will be used in the following. A *path* is a sequence of consecutive edges in a graph and the *length* is the number of edges traversed. A sample of path of length 3 in the graph depicted in Fig. 1 is $(n_6, n_2) - (n_2, n_1) - (n_1, n_3)$. Notice that the involved edges are not necessarily required to have the same direction.

3. THE QUERY MODEL

Our main aim is to go beyond the keyword-based approach without necessarily requiring to precisely use the graph vocabulary and structure in query formulation. To this end, we propose a query model which allows users to specify query graphs exploiting whatever partial knowledge they have. This is particularly useful not only when the user does not know the structure but also for querying heterogeneous data sources. As far as the queried graphs can be partially unknown to the user, queries can also contain imprecise specifications both about the node and edge labels they are searching for, and in the relationships among nodes. Our matching mechanism will respect whatever query constraints are given by dealing with the possibly contained label and structure ambiguities.

3.1 Query Specification

A query is a multigraph connecting entity nodes and predicates on values.

The query specification mechanism we propose allows the formulation of graphs expressing as much topology and annotation as the user can, including none at all. To this end, users can annotate any node or edge with the wildcard “any label”, ‘#’, when they do not know the label at all. Indeed, the symbol ‘#’ may be substituted by any of the labels in L. Moreover, users are allowed to specify general connections between nodes by means of undirected edges.

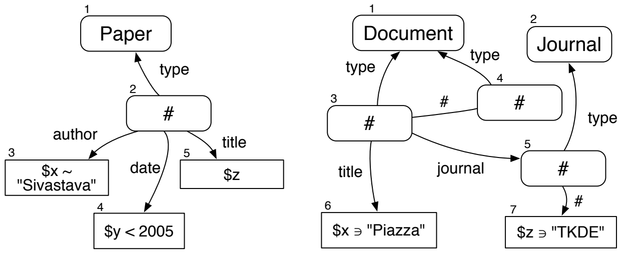


Figure 2: Query 1 (left) and Query 2 (right)

Definition 2 (Query Graph). *A query is a tuple $q = (N, E, L_N, L_E, V, C)$ where $E = E_d \cup E_u$, (N, E_d) is a directed multi-graph, (N, E_u) is an undirected multi-graph, (N, E, L_N, L_E) is a connected labeled multi-graph, V is a set of variables, and C is a set of conditions on V.*

$L_E \subseteq L_C$ and each $e \in E$ is assigned a label $\lambda(e) \in L_E \cup \{\#\}$. $L_N \subseteq L_C$ and each $n \in N$ is assigned a label $\lambda(n) \in L_N \cup \{\#\} \cup V$. Nodes having labels in $L_N \cup \{\#\}$ are called entity nodes while nodes having labels in V are called variable nodes. Each condition in C has the form $var\langle op\rangle v$ where $var \in V$, $\langle op\rangle$ is an operator, and $v \in L_L$ is a value. Possible operators are the relational ones, i.e., $=, <, \leq, >, \geq$, and the two operators $\ni$ and $\sim$. The semantics of the $\ni$ operator is the usual term containment in a text, whereas the $\sim$ operator expresses similarity between literals.

Fig. 2 depicts the graphs of two queries specified in Fig. 1. Notice that they both make use of the symbol ‘#’ to denote the user’s absence of knowledge about which entity nodes the specified terms are related to in the data (e.g., `Paper` and `author` in Query 1), and which relationships occur between such entity nodes (e.g., between the two document instances in Query 2). None of them finds an exact match on the reference graph because of node/edge label and structural mismatches: e.g. `Paper` and `date` in Query 1, the connection between n_1 and n_3 in Query 2. In the following, we propose a query answering semantics which effectively deals with such ambiguities.

3.2 The Query Answering Semantics

Our query answering model relies on approximate subgraph matching. In particular, approximation on data is performed in a two-fold fashion. As we allow label ambiguities, where the exact label names are unknown, the first kind of approximation we consider are node/edge label mismatches. Moreover, as users are not required to know the graph topology, we also support structural approximations.

Label Approximation

The degree of mismatch between concept labels is quantified by means of a distance function

$$d_L : L_C \cup \{\#\} \times L_C \cup \{\#\} \rightarrow [0, 1]$$

For any pair of labels, d_L is a symmetric function which returns a value ranging from exact match (0) to total mismatch (1). Obviously, for all labels $l \in L_C$, $d_L(\#, l) = 0$, $d_L(l, l) = 0$ and $d_L(\epsilon, l) = 1$. As far as its definition is concerned, we let the users customize the label matching

[2] For ease of reference, nodes are univocally identified by the integer numbers i shown on the left upper corner and will be referenced as n_i.

method that best suits the application needs. In our opinion, d_L should follow the principle of inter-substitutivity: the more l_i can be substituted by l_j in the same context and vice-versa the smaller $d_L(l_i, l_j)$ should be. For instance $d_L(\texttt{Paper}, \texttt{Article})$ should be lower than $d_L(\texttt{Paper}, \texttt{Journal})$. Possible alternatives range from purely syntactic approaches, such as the well known edit distance and its variants, to semantic approaches. To this extent, one possible solution is to correlate labels through knowledge-based distances, which take advantage of linguistic information extracted from external knowledge sources such as the WordNet (WN) thesaurus[3]. Here, we propose an adaptation of the Leacock-Chodorow [13] distance, which compares the WN hypernymy hierarchies of two given disambiguated labels l_i and l_j:

$$d_L(l_i, l_j) = \begin{cases} \frac{nlinks(l_i,l_j)}{2 \cdot H} & \text{if } \exists\ LCA(l_i, l_j) \\ 1 & \text{otherwise} \end{cases} \qquad (1)$$

where $nlinks(l_i, l_j)$ is the number of links connecting l_i and l_j in the hypernymy hierarchy, $LCA(l_i, l_j)$ is the Least Common Ancestor (LCA) between l_i and l_j, and H is the height of the hierarchy (16 in WordNet).

Structural Approximation

Exact graph matching maps each node adjacency specified in the query graph to exactly one data graph edge. A possible solution to support structural approximation is the purely topological approach which relaxes adjacency constraints by allowing node/edge insertions in the data graph.

But, since in a connected data graph all nodes are pairwise connected by at least one path, in order to cope with the possibly large amount of approximations, some syntactic properties on the involved labels and/or on the missed exact matches should be checked. This is the approach adopted, for instance, in [12, 18].

However, it should be noted that the fact that nodes are topologically connected, does not necessarily imply that they are meaningfully related. Rather, sometimes such an assumption would lead to useless results. For instance, notice that 2004 is not the year of publication of the paper `Srivastava92c` in Fig. 1 and thus the node pair (n_9, n_{13}) should not be considered when approximating Query 1. On the other hand, `Srivastava92c` is an instance of the `Document` class and the node pair (n_{13}, n_1) is thus useful for Query 2.

Essentially, we are concerned with graph-modeled data where graph nodes represent entities in the real world. Therefore, we can leverage on the graph topology semantics to decide whether any pair of nodes is meaningfully related and, in case, to annotate such a linkage. This leads to the notion of *Semantic Relatedness* on a data graph $\mathcal{G}$ as that relation that only contains node pairs in $\mathcal{G}$ which are meaningfully related. To our knowledge, the requirement that the elements satisfying a query must be meaningfully related has been first introduced in [4] for labeled keyword search over XML documents. In that context, meaningfully related nodes are used for conjunctive query answering. In our context, instead, they represent the extremes to be taken into account when approximating node adjacency.

Definition 3 (Semantic Relatedness (SR)). *Given a data graph $\mathcal{G}$, the Semantic Relatedness relation SR is a binary relation on the nodes in $\mathcal{G}$. It is a multiset of node pairs such that $(n, n') \in N \times N$ belongs to SR iff n and n' are* meaningfully related. *Each $e = (n, n') \in SR$ is assigned a label $\lambda(e) \in L_C$, a path $p(e)$ in $\mathcal{G}$ connecting n with n', and an approximation cost $c(e) \geq 0$.*

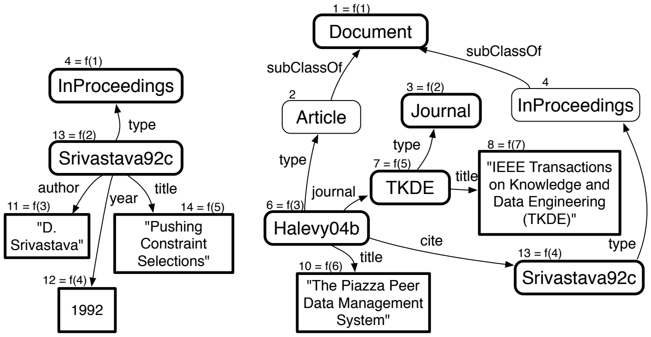


Figure 3: Embeddings for Query 1 (left) and for Query 2 (right)

In other words, each instance $e \in SR$ is a "virtual" edge which $p(e)$ approximates in $\mathcal{G}$, and $c(e)$ expresses the cost to approximate e with $p(e)$. It follows that $E \subseteq SR$ and, since no structural approximation is required for each edge $e \in E$, for each of them the approximation cost is $c(e) = 0$.

While the above SR definition abstracts from specific data models, Section 5 shows an example of how this relation can be customized for the RDF-like data model previously introduced.

Query Answers

Definition 4 (Query answer). *Let $\mathcal{G}$ be a data graph, $SR \subseteq N \times N$ be an SR relation over $\mathcal{G}$, and $q = (N^q, E^q, L_N^q, L_E^q, V^q, C^q)$ be a query. An SR-answer to q is an* approximate embedding *$\mathcal{E}^{SR} = (f, g)$ where f is an injective* node-assignment function *$f : N^q \rightarrow N$ such that*

- *for every entity node n, $f(n)$ is an entity node and $d_L(\lambda(n), \lambda(f(n))) < 1$;*
- *for every variable node n, $f(n)$ is a value node;*
- *for every condition $c = \lambda(n)\langle op \rangle v$ in C, $\lambda(f(n))\langle op \rangle v$ holds with a certain grade $s(c)$ which is called the* score *of c. In particular, relational operators have a Boolean semantics and thus $s(c)$ must be 1 whereas operators $\ni$ and $\sim$ return values $s(c) \in [0, 1]$ and it must be $s(c) > 0$;*

and g is an injective edge-assignment function *$g : E^q \rightarrow SR$ such that for every $e = (n, n') \in E^q$*

- *$d_L(\lambda(e), \lambda(g(e))) < 1$;*
- *if e is a directed edge, i.e. $e \in E_d^q$, then $g(e) = (f(n), f(n'))$, otherwise*
- *$g(e) = (f(n), f(n'))$ or $g(e) = (f(n'), f(n))$.*

The two kinds of approximation are used to deal with node mismatches and adjacency misses in query answers. In particular, f allows node label approximations while g allows both edge label approximations and adjacency approximations. To this end, notice that any exact embedding is an approximate embedding. Each embedding $\mathcal{E}^{SR}$ of a query graph q defines a subgraph of $\mathcal{G}$ which consists of the set of

[3] http://wordnet.princeton.edu

nodes in $f(N^q)$ connected through the paths in $g(E^q)$. The subgraphs depicted in Fig. 3 are defined by one plausible approximate embedding for each of the two query samples. The data nodes in the range of f are depicted in bold line and the query node image is shown on the left upper corner of each data node rectangle. For instance, $4 = f(1)$ means that data node n_4 is assigned to query node n_1.

For ease of presentation, the subgraph shown for Query 1 only contains label approximations: `Paper` is mapped to `inProceedings` and `date` to `year`. Dually, the embedding shown for Query 2 only deals with structural approximations: the data path $(n_6, n_2) - (n_2, n_1)$ approximates the query edge (n_3, n_1) whereas $(n_{13}, n_4) - (n_4, n_1)$ is for (n_4, n_1). Recall that, by definition of g, it is assumed that the data pairs (n_6, n_1) and (n_{13}, n_1) are in SR and that their labels (approximately) match. Indeed, `Halevy04b` as well as `Srivastava92c` are certainly documents. Also, it is worth noting that the subgraph makes explicit the relationship between $f(n_3)$ and $f(n_4)$: `Halevy04b` cites `Srivastava92c`.

Finally, since our primary focus is on approximate graph matching, the above definition does not delve into the specific functions $s(\cdot)$ used to evaluate conditions containing the IR-style operators $\ni$ and $\sim$. Our approach is general and completely application-independent, and those measures that best fit the nature of the data and the specific application needs can be easily integrated (e.g.,IR-style TF/IDF scores, edit distance).

4. THE RANKING MODEL

In a very frequent scenario where a large number of approximate embeddings are returned for a given query, we are concerned with finding the *top-ranked answers*. To this end, we introduce a ranking model which measures answer goodness by applying a scoring function $\mathcal{S}$ to each embedding $\mathcal{E}^{SR}$. In this section, we first argue about the use of SR in the ranking process. Then, we introduce a general ranking scheme which defines the fundamental properties of $\mathcal{S}$. Finally, we instantiate $\mathcal{S}$ to a specific scoring function which satisfies additional interesting properties.

4.1 Reducing SR for ranking purposes

Given a query graph, any set of data nodes matching the query nodes can be involved in more embeddings as to the way they are connected. Some of them could not be semantically different. In other words, SR can contain multiple occurrences of the same pair of data nodes which share the same label and only differ in the approximating path. Therefore, they essentially represent the same relationship. The ranking model we propose adopts a *distinct-node set semantics*, that is only one of the above embeddings is considered for ranking purposes. In particular, the model operates on a reduced version of SR where only the "less expensive" node pairs are maintained.

Definition 5 (Reduced SR). *Let $\mathcal{G}$ be a data graph and $SR \subseteq N \times N$ be a semantic relatedness relation. The reduced version of SR, denoted as $\overline{SR}$, contains each node pair $e = (n, n') \in SR$ such that for all $e' = (n, n') \in SR$ such that $d_L(\lambda(e), \lambda(e')) = 0$ then $c(e) \leq c(e')$.*

Therefore, given a semantic relatedness relation SR over a data graph $\mathcal{G}$ and a query q, a top-k query returns the k approximate embeddings $\mathcal{E}^{\overline{SR}}$ with the best scores $\mathcal{S}(\mathcal{E}^{\overline{SR}})$.

We have several reasons for adopting a distinct-node set semantics. First, the semantics of query answering is preserved as the set of answers built over any semantic relatedness relation is the same as the one obtained from its reduced version. Moreover, it is an intuitive and clean semantics. Second, we privilege the less expensive embeddings which, in some way, correspond to the most compact answers [9, 12]. Indeed, any edge approximation cost can be interpreted as the weakness of the connection between the involved nodes and thus it can even translate into the path length as shown in Section 5. On the other hand, each answer carries very little additional information from the rest of the answers overlapping both on the data nodes and on the connection semantics. Thus, by adopting a distinct-node set semantics we avoid to overwhelm users with quasi-redundant results. The last reason is more technical: it reduces the amount of data required for query answering.

4.2 Scoring functions

The scoring function $\mathcal{S}$ applied to an embedding $\mathcal{E}$[4] combines the approximations occurring at both data nodes and data edges, including the approximations on query conditions, and it returns a score in $[0, 1]$ such that the higher is $\mathcal{S}(\mathcal{E})$ the higher is the overall approximation required for the matching.

Definition 6 (Answer Score). *Given a query $q = (N, E, L_N, L_E, V, C)$ and an embedding $\mathcal{E} = (f, g)$ for q, the score of $\mathcal{E}$ is defined as:*

$$\mathcal{S}(\mathcal{E}) = \sigma(\mathcal{S}_f(\mathcal{E}), \mathcal{S}_g(\mathcal{E}), \mathcal{S}_c(\mathcal{E})) \tag{2}$$

where $\mathcal{S}_f$, $\mathcal{S}_g$, and $\mathcal{S}_c$ are functions which evaluate the approximations on data nodes, on data edges, and on query conditions, respectively, and σ is an increasingly monotone aggregation function having range $[0, 1]$.

The approximations components on data nodes and data edges $\mathcal{S}_f(\mathcal{E})$ and $\mathcal{S}_g(\mathcal{E})$, as well as the approximation component on query conditions $\mathcal{S}_c(\mathcal{E})$, are defined below.

Definition 7 (Approximation Components). *Given a query $q = (N, E, L_N, L_E, V, C)$ and an embedding $\mathcal{E} = (f, g)$ for q, the scores of the approximations on data nodes, on data edges, and on conditions C in $\mathcal{E}$ are defined as:*

$$\begin{aligned} \mathcal{S}_f(\mathcal{E}) &= \sigma_f(\{d_f(n, f(n)) | n \in N_e\}) \\ \mathcal{S}_g(\mathcal{E}) &= \sigma_g(\{d_g(e, g(e)) | e \in E\}) \\ \mathcal{S}_c(\mathcal{E}) &= \sigma_c(\{d_c(c) | c \in C\}) \end{aligned}$$

where $N_e \subseteq N$ is the set of entity nodes. The symbols d_f, d_g, and d_c denote distance functions ranging from 0 to 1 and quantifying the approximation between each query node and its corresponding data node, each query edge and its mapped instance in $\overline{SR}$, and each condition and the matching value, respectively. σ_f, σ_g, and σ_c are increasingly monotone aggregation functions whose range is $[0, 1]$.

In order to ensure an intuitively "coherent" semantics in this context, some general properties should be guaranteed by the above functions. σ must satisfy the commutativity and associativity properties, 0 should be the identity element (i.e. $\sigma(d, 0) = d$) and 1 should be the annihilating

[4] In the following we assume that $\overline{SR}$ has been fixed and use $\mathcal{E}$ in place of $\mathcal{E}^{\overline{SR}}$.

element (i.e. $\sigma(d, 1) = 1$). The same applies to σ_f, σ_g, and σ_c. The last two conditions imply, from one hand, that the contribution of exact matches is absorbed in the evaluation of the respective score, which is representative of the actual approximations only, from the other hand, that any value of total mismatch leads to an overall total mismatch. The above properties also guarantee that, when $\mathcal{E}$ is an exact match, $\mathcal{S}(\mathcal{E}) = 0$, which confirms the intuition that no approximation is required in that case.

The general ranking scheme above can be tailored to the model presented so far by instantiating Eq. 2 to one specific scoring function as follows:

$$\begin{aligned} S(\mathcal{E}) \quad = \quad & \frac{\alpha}{n_e}\sum_{i=1}^{n_e} d_L(\lambda(n_i), \lambda(f(n_i))) + \\ & \frac{\beta}{2m}\sum_{j=1}^{m}(d_L(\lambda(e_j), \lambda(g(e_j))) + \frac{c(g(e_j))}{MC}) + \\ & \frac{\gamma}{n_c}\sum_{i=1}^{n_c}(1 - s(c_i)) \end{aligned} \qquad (3)$$

where α+β+γ=1 and MC is a normalizing constant corresponding to the maximum cost in SR. The function defines the score of an embedding $\mathcal{E}$ to a given query q by summing up the contribution of the n_e entity nodes, of the m edges, and of the n_c conditions in q. In particular, d_f uses the distance function d_L between node labels; d_g depends both on d_L for edge labels and on the approximation costs in SR to evaluate adjacency mismatches; d_c is instantiated to the inverse of the function $s(\cdot)$ used for condition evaluation.

It should be noted that the scoring function $S(\mathcal{E})$ instantiated above satisfies two interesting properties:

Cost-based graph-distance semantics. Recall that S is applied on embeddings which refer to $\overline{SR}$. This means that the computation of the latter factor of the edge approximation component in Eq. 3 can be reduced to the shortest-path problem, since the cost associated to the approximation of a given edge e is the lowest one among all possible costs associated to alternative approximations for e.

Match-distributive semantics. The overall score of an embedding $\mathcal{E}$ can be computed in a distributive way. This means that all matching paths contribute independently to the score computation, even if they may share some common edges. This is different from other approaches which consider the single contribution of each edge only once (e.g., [3]). As already evidenced in [9], this property has important implications on the complexity of the score computation. As a matter of fact, in our model this semantics allows for the precomputation of the approximation costs in $\overline{SR}$, and these can be combined as independent parts, thus disregarding the repeated contribution of possible overlapping paths.

We take advantage of these properties to efficiently support query answering and ranking in our model, as shown in Section 6.

5. A RDF-LIKE INSTANTIATION OF SR

In this section, we show a possible instantiation of SR for the RDF-like data model introduced in Section 2.

Such an instantiation relies on the notion of type. More precisely, data edges are grouped together on the basis of the kind of relationship they represent. We assume the existence of five edge types: *property*, *type*, *isA*, *isPartOf*, and *domRel*. The type *property* is assigned to each edge between an instance node and a value node. A sample of *property* edge is the edge (n_6, n_5) in the data graph in Fig. 1. *isA* is an acyclic transitive relation which expresses a hierarchical relationship between two classes while a *type* relation is used to link one entity node to a class it belongs to. All the edges in Fig. 1 labeled *subClassOf* and *type* are of type *isA* and *type*, respectively. *isPartOf* concerns the membership of an instance to another instance and, finally, *domRel* denotes any relationship which can be established between instances. The data graph in Fig. 1 contains two *domRel* edges: (n_6, n_7) and (n_6, n_{13}). In the following, we adopt the notation $\tau(e)$ to denote the type of the edge e.

The construction of SR on any data graph $\mathcal{G} = (N, E, L_N, L_E)$ conforming to the RDF-like data model is performed in an incremental fashion through a set of rewriting rules founded the type semantics and the graph topology. Similarly to [17] but for different purposes, the rewriting system starts from a set of axiomatic rules for edges in E, and recursively adds new node pairs by exploiting a set of extension rules. Each rule has a left-hand part which states preconditions and a right-hand part which specifies the properties of the node pair added to SR. The following axiomatic rule initializes SR with the set of edges in E[5]:

$$e \in E \quad \rightarrow \quad \{e | c(e) = 0, \tau_{SR}(e) = \tau_{\mathcal{G}}(e), p(e) = e\}$$

while the following rules are used to associate a label to each edge:

$$\begin{aligned} \tau_{\mathcal{G}}(e) = \textit{isA} \quad &\rightarrow \quad \lambda_{SR}(e) = \texttt{isA} \\ \tau_{\mathcal{G}}(e) = \textit{isPartOf} \quad &\rightarrow \quad \lambda_{SR}(e) = \texttt{isPartOf} \\ \tau_{\mathcal{G}}(e) = \textit{type} \quad &\rightarrow \quad \lambda_{SR}(e) = \texttt{type} \\ \tau_{\mathcal{G}}(e) = \textit{property} \quad &\rightarrow \quad \lambda_{SR}(e) = \lambda_{\mathcal{G}}(e) \\ \tau_{\mathcal{G}}(e) = \textit{domRel} \quad &\rightarrow \quad \lambda_{SR}(e) = \lambda_{\mathcal{G}}(e) \end{aligned}$$

Notice that whenever the node semantics is carried by the type, the node is assigned a default label. Then, SR is extended by means of the following rules:

For all $e = (x, y), e' = (y, z) \in SR$:
(r1) $\tau(e) = \tau(e') = \textit{isA}$ OR $\tau(e) = \textit{type}, \tau(e') = \textit{isA}$
OR $\tau(e) = \tau(e') = \textit{isPartOf} \rightarrow$
$\{e'' = (x, z) | \tau(e'') = \tau(e), \lambda(e'') = \lambda(e),$
$p(e'') = p(e) - p(e'), c(e'') = c(e) + c(e') + 1\}$

For all p_1, p_2 properties, $e = (x, y) \in SR$:
(r2) $\tau((p_1, p_2)) = \textit{isA}, \lambda(e) = pd(p_1).\lambda \rightarrow$
$\{e' = (x, y) | \tau(e') = pd(p_2).\tau, \lambda(e') = pd(p_2).\lambda,$
$p(e') = p(e), c(e') = c(e) + 1\}$

For all p properties, $e = (x, y), e' = (y, z) \in SR$:
(r3) $\tau((p, \texttt{aTrans})) = \textit{type}, \lambda(e) = pd(p).\lambda, \lambda(e') = pd(p).\lambda \rightarrow$
$\{e'' = (x, z) | \tau(e'') = pd(p).\tau, \lambda(e'') = pd(p).\lambda,$
$p(e'') = p(e) - p(e'), c(e'') = c(e) + c(e') + 1\}$

For all p properties, cl classes, $e = (x, y), (p, cl) \in SR$:
(r4) $\tau((p, cl)) = \textit{domain}, \lambda(e) = pd(p).\lambda \rightarrow$
$\{e' = (x, cl) | \tau(e') = \textit{type}, \lambda(e') = \texttt{type},$
$p(e') = p(e), c(e') = c(e) + 1\}$
(r5) $\tau((p, cl)) = \textit{range}, \lambda(e) = pd(p).\lambda \rightarrow$
$\{e' = (y, cl) | \tau(e') = \textit{type}, \lambda(e') = \texttt{type},$
$p(e') = p(e), c(e') = c(e) + 1\}$

[5] We use subscripts $\mathcal{G}$ and SR to distinguish the properties in the graph and in the semantic relatedness relationship, respectively.

In particular, in rule (r1) the path associated to the newly added edge is the concatenation of the two involved paths and the cost actually represents the difference between the length of $p(e'')$ and the length of a direct connection between x and z, i.e. 1.

The above rules applied to the very small data graph shown in Fig. 1 add three edges to SR:

e	$\tau(e)$	$\lambda(e)$	$p(e)$	$c(e)$
(n_6, n_1)	*type*	`type`	$(n_6, n_2) - (n_2, n_1)$	1
(n_7, n_1)	*type*	`type`	$(n_7, n_3) - (n_3, n_1)$	1
(n_{13}, n_1)	*type*	`type`	$(n_{13}, n_4) - (n_4, n_1)$	1

The approach we adopted in introducing the above rules is conservative as it is exclusively based on types and does not make any assumption on the involved labels. On the other hand, to adopt a "naïve" approach for the SR expansion could bring to misleading results. For instance, the author of a paper which cites another paper is not the author of the cited paper. Thus, we prefer to not overwhelm users with wrong results while accepting few false negatives. One way to overcome this problem is to adopt a fine grain semantic analysis of the chain of node and edge labels in order to state whether the starting node and the ending node are semantically related or not. This study is out of the scope of this paper and will be dealt with in our future work.

In the following, we extend the RDF-like model and the related rewriting system in order to take into account, besides the subclass hierarchies, other meaningful class properties which can be defined in OWL and RDFS. In particular, we focus on the *property* and *domRel* edges which have been excluded in the previous model. For ease of reading, in the following we will refer to the *property* and *domRel* edges by using the generic term "property". Properties are meta-level classes whose domains and ranges are defined through the newly added edge types *domain* and *range* and whose hierarchies are defined through *isA* edges. Moreover, whenever OWL descriptions are available, we consider the acyclic transitive property characteristic and allow this characteristic to be stated through the edge type *type* and the newly added class `aTrans`. For instance, the triples (`cite`, `type`, `aTrans`), (`cite`, `domain`, `Document`), and (`cite`, `range`, `Document`) could be used to define the property `cite`. Given this extended model, the rewriting system is extended with rules (r2)-(r5) above. Function $pd(\cdot)$ associates any meta-level class defining a property with the property it defines (the label $pd(\cdot).\lambda$ and the type $pd(\cdot).\tau$). To this end, we assume that the bound between each property defined at meta-level and its use at instance level is made through the involved labels which must be the same. Rule (r2) states that all node pairs related by one property are also related by its super-properties, rule (r3) extends rule (r1) to any transitive property while rule (r4) and (r5) exploit the property domain and range to add new class instances. Finally, it is worth noting that the cost associated to each edge added through rules (r2), (r4), and (r5) is not related to the path length but rather to the number of node pairs used to infer them.

THEOREM 1 (CONVERGENCE). *SR is finite and unique.*

PROOF. First notice that the rewriting system is monotone and finitely terminating. Moreover, it is also locally confluent. It follows that the rewriting system is globally confluent. Therefore SR is finite and unique. □

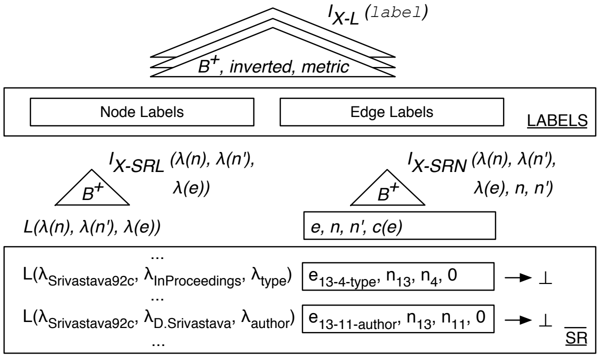

$I_{X\text{-}L}$ (label)
B^+, inverted, metric
Node Labels
Edge Labels
LABELS
$I_{X\text{-}SRL}$ ($\lambda(n)$, $\lambda(n')$, $\lambda(e)$)
B^+
$I_{X\text{-}SRN}$ ($\lambda(n)$, $\lambda(n')$, $\lambda(e)$, n, n')
B^+
$L(\lambda(n), \lambda(n'), \lambda(e))$
e, n, n', $c(e)$
...
$L(\lambda_{Srivastava92c}, \lambda_{InProceedings}, \lambda_{type})$
$e_{13\text{-}4\text{-}type}, n_{13}, n_4, 0$
→ ⊥
...
$L(\lambda_{Srivastava92c}, \lambda_{D.Srivastava}, \lambda_{author})$
$e_{13\text{-}11\text{-}author}, n_{13}, n_{11}, 0$
→ ⊥
...
$\overline{SR}$

Figure 4: An overview of the index structures

Finally, we did not include the two property characteristics, namely inverse and symmetric, due to the termination problem. Indeed, a repeated application of any rewriting rule exploiting such characteristics would add new instances to SR identical to some of the already included instances in SR, but paths and costs. Thus, the process would never terminate. However, as we adopt a distinct-node set semantics, we can directly compute $\overline{SR}$ instead of computing SR and, then, reducing it. In this case, we can add the above instances only once, the first time as it can be shown that the cost of the instances added afterwards would be higher.

6. DATA STRUCTURES AND ALGORITHMS

In this section, we introduce the data structures and the algorithms implementing the model presented so far.

6.1 Index Structures

Fig. 4 shows a graphical overview of the adopted index structures. Specifically, we exploit two families of indices: label (top part of Fig. 4) and $\overline{SR}$ indices (bottom). Let us first focus on the latter, as they are particularly crucial to the algorithm workings.

Indeed, the most expensive operation related with query processing is to check whether two data nodes are semantically related or not under the label constraints specified in the query. Our aim is to reduce space and time complexity of search and a common approach to do this is to perform some offline operations. To this end, we decided to create $\overline{SR}$ indices that would store the reduced version of the semantic relatedness relation. We cluster all the node pairs $e = (n, n') \in \overline{SR}$ sharing the same labels $\lambda(n)$, $\lambda(n')$, and $\lambda(e)$ together with the related costs $c(e)$. The result is a collection of lists where each list $L(\lambda(n), \lambda(n'), \lambda(e))$ is a sequence of entries $(e, n, n', c(e))$ ordered for ascending $c(e)$ and accessible through the tuple $(\lambda(n), \lambda(n'), \lambda(e))$.

As to the implementation, instead of designing new index structures, we preferred to consider existing solutions that can support the $\overline{SR}$ efficiently. We propose to rely on the strong bases of existing relational systems: the `SR-RV(Start,End,Edge, LStart,LEnd,LEdge,C)` relation stores each node pair in $\overline{SR}$, the starting node identifier in `Start`, the ending node identifier in `End` and the edge identifier in `Edge`, together with their labels (`LStart`, `LEnd`, and `LEdge`), and the cost (`C`). Then we build a B+-tree, denoted as I_{X-SRL} in Fig. 4, that clusters first on `(LStart,LEnd,LEdge)` and then on `C`. In this way, we can take advantage of sequential I/O accesses as tree leaf pages are linked and records in them are ordered on increasing values of C. Fig. 4 shows the contents of two of the L lists in $\overline{SR}$ for our running example. Notice that, as each node n, each edge e is also referenced in the indices through a unique numeric identifier; however, in order to make the example more readable, we employ a start, end and edge label subscript. For the same reason, we represent numeric label ids with λ and the label value as subscript.

Furthermore, we envisage a series of ad-hoc indices which efficiently support additional searches on SR-RV and on labels. More specifically, random accesses on the node identifiers as required by the following algorithm are implemented with B+-trees (index I_{X-SRN} in Fig. 4). Differently from I_{X-SRL}, which is used to determine the relevant lists L and thus points to the lists' starting elements, such an index is accessed also through a starting and ending node identifier (n and n', respectively) and directly points to all relevant entries inside such lists.

Finally, let us consider the structures supporting searches inside labels (upper part of Fig. 4). Range and containment search of value labels is supported by B+tree and inverted indices. Other indices are also available for approximate searches. For instance, a metric index on the disambiguated version of the labels in SR-RV allows label distance computations. All the different types of label indices are uniformly accessed through the requested label value label and are denoted in figure as I_{X-L}; they return the identifiers λ of the data labels matching the one required.

Note that, for sake of generality, the indices described in this section implement the general data model; however, they can be straightforwardly extended with the features which are specific to each particular instantiation of $\overline{SR}$, such as type information.

6.2 An Algorithm for Top-k Querying

In this section we propose a querying algorithm that efficiently generates the top-k answers in an order that is correlated with the ranking measure. The algorithm builds from the principles of the Threshold Algorithm (TA) proposed in [6], which has been proven optimal in terms of number of items visited, assuming the aggregation function is monotone. Before presenting the complete algorithm and discussing the enhancements it presents over other TA-derived algorithms, we will first give an intuition of its working, also by means of an example.

For the sake of simplicity of presentation, we first consider oriented queries and exact matches for nodes' and edges' labels and for conditions, then we will show how to relax this assumption and deal with approximate and/or unspecified labels as well as with undirected edges. Therefore, initially $S(\mathcal{E}) = \frac{\beta}{2m}\sum_{j=1}^{m}\frac{c(g(e_j))}{MC}$ and each query edge $e_i = (n_{S(i)}, n_{E(i)})$, for $i \in [1,m]$, is associated with a sorted list $L(\lambda(n_{S(i)}), \lambda(n_{E(i)}), \lambda(e_i))$. This is easily done by searching for the requested labels in I_{X-L}, and then by accessing I_{X-SRL} in order to retrieve the relevant lists. Then, the algorithm basically performs sorted access in parallel to each of the m sorted lists. As a node pair $e = (n, n')$ is seen in some list (object in TA), it does random access to the other lists to compute answers. For each answer $\mathcal{E}$, the algorithm computes its score $S(\mathcal{E})$ and if it is one of the k lowest, then it remembers $(\mathcal{E}, S(\mathcal{E}))$. After each answer computation step, the algorithm computes the score $lBound$ of the set of the next node pairs under sorted access to the m sorted lists as they were a solution, and stops the process whenever at least k answers have been seen whose grade is smaller than $lBound$. Indeed, as lists are ordered, $lBound$ represents the best score of any solution which could be computed in the following steps.

For instance, going back to our data graph (Fig. 1) and Query 1 (Fig. 2), we will simplify the query so to avoid the need for approximate label matching, a feature which is not essential to understand the algorithm core. Let us consider only the first three query nodes (n_1, n_2 and n_3) and modify their labels to InProceedings, Srivastava92c and D. Srivastava, respectively. The algorithm proceeds in the following way:

1. the data label ids matching the nodes' and edges' labels are found by means of I_{X-L};
2. $\overline{SR}$ is then queried through I_{X-SRL}: the first edge (query nodes n_2-n_1) searches for ($\lambda(n) = \lambda_{Srivastava92c}$, $\lambda(n') = \lambda_{InProceedings}$, $\lambda(e) = \lambda_{type}$) and is associated to the first list shown in figure (L_1 for short), while, in a similar way, the second edge is associated to the second list (L_2 for short);
3. sorted access is performed on L_1 and a node pair (n_{13},n_4) is extracted from the list;
4. all answers involving the extracted pair are efficiently constructed by performing random accesses in the other lists; for the second query edge, index I_{X-SRN} is queried for $n = n_{13}$ (being this the starting node for the second edge too);
5. finally, results are generated by combining the matches for each of the query edges; in our simple case, exactly one answer $\mathcal{E}$ is found, its score $S(\mathcal{E})$ is computed and, since the lists do not contain additional data, the algorithm stops and returns the required answer.

Algorithm 1 answerQuery(q,k)

```
1: for all $i \in [1,m]$ do        // query edges
2:    $C_i \leftarrow$ newCursor($L(\lambda(n_{S(i)}), \lambda(n_{E(i)}), \lambda(e_i))$)
3: i=0;
4: while ($i \leftarrow$ getNextCursor($i$)) $> 0$ do
5:    $(e, n, n', c(e)) \leftarrow C_i$.next()
6:    computeAnswers($i, e, n, n', c(e), 0, \emptyset, \emptyset$)
7:    $lBound \leftarrow \sum_{j=1}^{m} C_j$.peekCost()
8:    if $|Ans| \geq k$ and $lBound \geq Ans[k].dist$ then
9:       abort answer computation
10: output $Ans$
```

The algorithm will now be presented in detail. The pseudocode is shown in Algorithm 1. Given a query q and the number k of results to be returned, a cursor C_i is used to traverse each list $L(\lambda(n_{S(i)}), \lambda(n_{E(i)}), \lambda(e_i))$ (lines 1-2). In particular, after line 4 selects cursor C_i by calling algorithm getNextCursor() (Algorithm 3), C_i advances through the next() function which returns the next node pair $e = (n, n')$, together with its cost $c(e)$. Then, (some of) the answers originating from e are computed (Algorithm 2) and the process stops as soon as the top k answers have been found (lines 7-9). An answer is a triple $(nList, eList, dist)$ where $nList[1, \ldots, n]$ is a list of data nodes, one for each query node n_i, which encodes the node-assignment function f, $eList[1, \ldots, m]$ is a list of node pairs in $\overline{SR}$, one for each query edge e_i, which encodes the edge-assignment function g, and $dist$ is the score $S(\mathcal{E})$ of the approximate embedding $\mathcal{E} = (f, g)$. Both list entries are initialized with node placeholders and updated while computing the answer. The top-k answers are maintained in a list Ans sorted on increasing $dist$ and initialized as an empty list. Algorithm 2 recursively computes each answer originating from the current query edge e_i.

Algorithm 2 computeAnswers($i, e, n, n', cost, curD, nList, eList$)

```
1: $nList[S(i)] = n$
2: $nList[E(i)] = n'$
3: update($eList, i, e$)
4: $curD \leftarrow curD + cost$
5: $\bar{\imath} \leftarrow$ getNextQueryEdge($i$)
6: if $\bar{\imath} < 0$ then    // answer completed
7:     if $Ans[k].dist > curD$ OR $|Ans| < k$ then
8:         $Ans$.add(($nList, eList, curD$))
9:     return
10: if $|Ans| \geq k \wedge curD + C_{\bar{\imath}}$.peekCost()) $\geq Ans[k].dist$ then
        // abort computation of current answer
11:     return
12: else    // continue answer computation recursively
13:     while (($\bar{e}, \bar{n}, \overline{n'}, \overline{cost}) \leftarrow C_{\bar{\imath}}$.seek($eList[\bar{\imath}].n_S, eList[\bar{\imath}].n_E$))
            $\neq NULL$ do
14:         computeAnswers($\bar{\imath}, \bar{e}, \bar{n}, \overline{n'}, \overline{cost}, curD$, $nList$, $eList$)
15:     return
```

Whenever the current answer computation process is completed by the newly entering data node pair $e = (n, n')$, the answer is added to Ans if Ans contains less than k entries or if its distance is no greater than the k-th one (lines 1-9). Otherwise, it can be interrupted for two reasons: the lower bound of the answers which will be computed in the following exceeds the pruning threshold $Ans[k].dist$ (lines 10-11) or no matching object in the selected list $C_{\bar{\imath}}$ is found (lines 13-15). In particular, $C_{\bar{\imath}}$.seek() usually performs indexed random accesses through the node identifiers available in $eList[\bar{\imath}]$, where $\bar{\imath}$ is the current query edge selected at line 5. More precisely, each time a new data node pair is added to the current solution, the update() function does not limit itself to update $eList[i]$, but it also uses n and n' to update the entries corresponding to e_i's adjacent edges. Then, getNextQueryEdge() selects at each call the next unvisited query edge which has been (partially) bounded as an update() side effect and $C_{\bar{\imath}}$.seek() uses such a key.

Summing up, the algorithm operates in a more challenging scenario than other TA-derived algorithms present in the literature (such as [9]) and thus introduces several enhancements and modifications w.r.t. them:

- each object in one list conceptually joins with more than one object in each of the others. Therefore, after an object is seen under sorted access, all answers involving such node pair should be computed and they can be potentially a large number, differently from most of the TA-derived algorithms (e.g. [9]) where at most one answer is returned. For this reason, a pruning threshold is applied not only to decide if the generated answers are sufficient, but also during answer computation to avoid completing the computation of useless answers;
- the fact that an object has been already seen under random access does not mean that all the solutions involving it have been generated. Therefore, the number of random accesses can be very high. In order to reduce it, a memory buffer structure Buf_i can be internally associated to each cursor C_i to collect all already seen objects in C_i. Buffers are organized by n, n' or (n, n'). In this way, C_i.seek() would first look in Buf_i and, only when the searched information is not found, it would search in the corresponding list and update Buf_i with the retrieved objects, in order to speed up subsequent searches;
- as far as the cursor selection is involved, differently from [9] where only round-robin accesses (`ROUND_ROBIN`) are considered, Algorithm 3 includes additional cursor selection strategies whose goal is to try to build better ranked answers as soon as possible. The `NEXT_BEST` mode chooses the cursor minimizing the variation of the pruning threshold. Moreover, as our lists are not equally sized, we can start building answers from the most selective edges by prioritizing the cursors whose size is the smallest one (`MAX_SEL`).

Algorithm 3 getNextCursor($current$)

```
$next \leftarrow -1$
if $Mode$ =ROUND_ROBIN then
    $next \leftarrow$ pick $i$ from $[1, \ldots, m]$ in round robin starting from
        $current$ such that $C_i$.size()> 0
else if $Mode$ =NEXT_BEST then
    $next \leftarrow$ find $i \in [1, \ldots, m]$ minimizing ($C_i$.nextCost()
        $-C_i$.curCost()) such that $C_i$.size()> 0
else if $Mode$ =MAX_SEL then
    $next \leftarrow$ find $i \in [1, \ldots, m]$ minimizing $C_i$.size() such that
        $C_i$.size()> 0
return $next$
```

The algorithm can be easily extended to handle the previously shown complete scoring function. In particular, differently from other TA-derived algorithms [9] which do not always satisfy those advanced requirements, the way the cursors C_i are organized effortlessly provides the flexibility needed to handle undirected edges, approximate labels' matching and more complex query predicates. In this case, each query label is associated with more than one data label: these are the labels similar to the query label w.r.t. d_L. The same applies to the labels on the query edges. Similarly, predicates are solved with B+tree and inverted indices to retrieve the matching labels. In this way, each query edge is associated to more than one cursor. The same applies to undirected edges which are associated to at most two cursors. Therefore, getNextCursor() selects among all the instantiated cursors. Note that, due to the way lists are constructed, the objects of each cursor share the same labels. Thus, fixed a query edge $e_i = (n_{S(i)}, n_{E(i)})$, it is sufficient to associate each cursor with three values, D^L_S, D^L_E and D^L_E, expressing the distance of the starting node, ending node and edge labels, respectively, from the query ones. As to answer ranking, $nList$ and $eList$ entries are simply extended with appropriate d_L fields in order to compute the distance of each answer.

Finally, notice that any unspecified label could be dealt with as a special case of an approximate label, as # matches with all the data labels. However, the number of data labels is usually very high and thus the management of all the instantiated cursors would be impractical. For this reason, we decided to implement ad-hoc lists which maintain the node pairs in $\overline{SR}$ ordered on increasing values of cost and which are accessible through the specified labels. In this way, each query edge is associated with one cursor also when it includes one or more unspecified labels.

7. EXPERIMENTAL EVALUATION

We performed a thorough analysis of both the effectiveness of our query answering model and the efficiency of the presented algorithms on several RDF-based real world datasets. In this section we present a selection of the most significant results we obtained.

7.1 Experimental Setting

Among the several collections we employed to test our approach, we selected four of them, Russia[6], DBLP-S, DBLP-L and DBLP-XL, for the following discussion, as they allow us to completely stress the system from all the required perspectives. Russia describes several information about the country's cities and people, while DBLP-S, DBLP-L and DBLP-XL are extracted from the well known DBLP scientific bibliography data, enriched in its RDF version with several interconnections, such as between papers by the same authors. Russia and DBLP-S are not very big in size, consisting in 1012 nodes / 1613 edges and 4373 nodes / 7779 edges, respectively, which we employed for testing the system effectiveness. In particular, the Russia dataset has a very detailed schema, with over 500 different and richly interconnected concepts and properties, while DBLP is a typical data-centric collection, presenting nearly 100 schema elements and a large number of instances. On the other hand, we selected the DBLP-L collection with the aim of stressing the approach from an efficiency point of view: the collection shares the schema of DBLP-S and consists of 445345 nodes and 763976 edges. Moreover, for additional efficiency and scalability analyses, we also employed a DBLP-XL dataset, consisting of 4563896 nodes and 7548976 edges. For both Russia and DBLP, we considered a set of significant queries, named R1 to R5 (Russia) and Q1 to Q4 (DBLP). The queries are designed to be increasingly more complex, both in terms of number of nodes and unspecified labels, and thus to test the system on increasingly more demanding cases. See the next section for more details. In order to apply label similarities, the concept labels are disambiguated with the STRIDER [15] structural disambiguation system, which in a completely automatic way associates the most probable meaning to each of the labels; the data labels having a similarity higher than a specified similarity threshold are associated to the query labels. The algorithm and data structures are implemented in Java 1.6, exploit MySQL 5.0 and Oracle BerkeleyDB 3.2 Java Edition storage engines and indices. All the experiments are executed on an Intel Core2 Duo 2.4Ghz OSX workstation, equipped with 2GB RAM and a 160GB SATA disk.

7.2 Effectiveness Evaluation

Tab. 1 presents a first selection of the effectiveness results we obtained for Russia (upper part of the table) and DBLP-S (lower part), along with an overview of the numeric features of the employed queries (number of nodes, number of edges, number of "any labels" #, number of expected answers). Queries typically require both structural and semantic approximations to be correctly solved. The table shows different measures and comparisons: from left to right, we compute the number of queries that would be necessary to obtain all the expected results from an exact matching approach, the number of queries that would have to be submitted to specific web search portals (only applicable to DBLP) and the precision (i.e. percentage of relevant retrieved answers w.r.t. the retrieved ones) of our approach in different settings. We also compare our precision to the one of an exact match and of a naïve approach. While the exact match approach clearly has a very limited flexibility, a naïve approach (similarly to [4]) is somehow opposed: the naïve results are the ones that would have been retrieved by computing all node matches and connecting them in the data in all possible ways, disregarding our SR information.

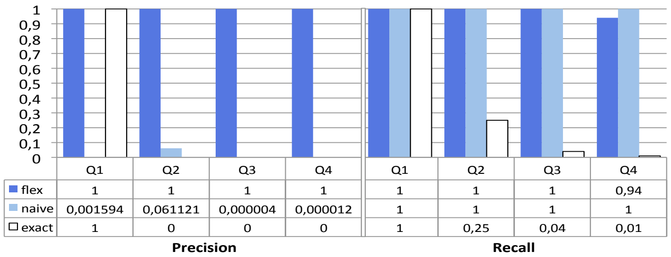
				answers	equiv queries			precision	
Query	#n	#e	#any	#exp	#exact	#web	P	naive	exact
					Russia dataset				
R1	3	2	1	1	1	n/a	1	0,200	1
R2	3	2	1	4	13	n/a	1	0,308	n/a
R3	3	2	1	4	689	n/a	1	0,308	n/a
R4	4	3	2	4	637	n/a	1	0,044	n/a
R5	5	4	3	4	49	n/a	1	0,267	n/a
				answers	equiv queries		precision	precision (lowT)	
Query	#n	#e	#any	#exp	#exact	#web	P	P	P@10
					DBLP-S Dataset				
Q1	3	2	1	12	1	3	1	0,355	1
Q2	3	2	1	36	4	2	1	1	1
Q3	5	4	3	10	25	n/a	1	0,893	1
Q4	6	6	4	148	64	41	1	0,754	1

Table 1: Effectiveness results - Russia and DBLP-S

Let us examine the results for Russia. Query R1 is the most simple, "The authors that studied at the University of Kazan": the query does not require approximations, thus both the exact approach and ours return the correct answer, Lev Tolstoj, with a precision of 1. The naïve approach, however, builds a larger number of answers, since it connects other authors to the required university through paths that are not semantically relevant (precision 0.2). The other queries require different approximations: R2 generically asks for "People that studied at the University of Kazan", R3 "People semantically connected (# edge) to University of Kazan", R4 "The public spaces semantically connected (# edge) to Lev Tolstoj", R5 "The public spaces that lie in the same city as Vosstaniye Square". Notice that for all of them we achieve perfect precision; this is made possible by exploiting the SR rules outlined in the previous sections and, for the label similarity computations, a "safely" high similarity threshold setting, nonetheless also allowing us to obtain perfect recall (i.e. percentage of relevant retrieved answers w.r.t. existing relevant ones) levels, not shown in table due to lack of space. Instead, the other approaches are either not applicable (no retrieved exact matches) or very inaccurate. Also, an approach which "rewrites" our query to all possible exact queries would be almost infeasible due to the excessive growth of the number of equivalent queries.

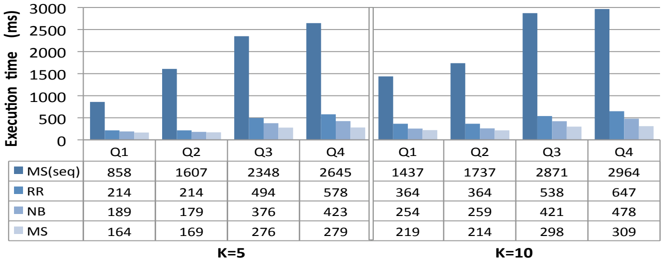
These positive results are also confirmed by the DBLP queries, specifically: Q1 "Articles of year 1997", Q2 "Papers (generically) of year 1997", Q3 "Titles of papers of date 1990 and semantically connected (# edge) to STACS conference" and Q4 "Titles of documents created by a person who has also created a document in 1994". In this case, we can also see that retrieving all the relevant answers through the DBLP search engines on the web would, again, require a significant work from the user (see the relevant column table). See also Fig. 5, which gives full details on precision and recall figures in this setting and compares our results ("flex" for short) to the naïve and exact ones. Very high precision and recall levels (typically perfect, while in Q4 recall is less than 1 since "person" matches with "author" but

[6] Publicly available at http://www.rdfdata.org



	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
flex	1	1	1	1	1	1	1	0,94
naive	0,001594	0,061121	0,000004	0,000012	1	1	1	1
exact	1	0	0	0	1	0,25	0,04	0,01

Figure 5: Effectiveness comparison - DBLP-S



	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
MS(seq)	858	1607	2348	2645	1437	1737	2871	2964
RR	214	214	494	578	364	364	538	647
NB	189	179	376	423	254	259	421	478
MS	164	169	276	279	219	214	298	309

Figure 6: Query Execution time for DBLP-L

not with "editor", employing the standard semantic threshold) are achieved by us; instead, while the exact approach is good for query Q1, it is inappropriate for the others, and the naïve generally has perfect recall but very low precision.

Finally, back to Tab. 1, we present a small but representative sample of a specific effectiveness evaluation we performed on our ranking model and function. In particular, we simulated a more rich and "noisy" answer set to DBLP queries by significantly lowering the semantic approximation threshold employed for label match ("lowT" section of the table) and, together with precision P, we computed precision at recall level 10, P@10. As shown in table, even if P is globally lower (for instance, "title" is now also similar to label "note"), the function proves to be effective in discriminating the irrelevant answers and keeping them low in the ranking, with perfect P@10 levels.

7.3 Efficiency Evaluation

Along with a good effectiveness, a graph query answering approach also necessarily needs to be very efficient, since the size of the managed graphs can be very high. In the following tests, we analyze the performance of our proposal on the large DBLP-L collection, considering query execution time and the number of index accesses performed by the different available cursor selection strategies and access modes. Note that since we are not focused on the efficiency of the preprocessing phases, such as index construction, we will not present an analysis of these performances.

Fig. 6 shows query execution time of queries Q1 to Q4 for the Round Robin (RR), Next Best (NB) and Max Sel (MS) cursor selection strategies, for both K=5 and K=10. As we can see, execution time reflects the query complexity, both considering the number of edges and the selectivity of the labels. Further, for all of them, the random accesses performed by `seek()` exploit indices; the sequential versions of these strategies proved significantly slower then their index-based counterparts, thus we present only the best performing one ("MS(seq)" in figure) as a baseline. Note that we also tested the performances of a memory buffer for caching index data accesses, as described in the algorithm discussion, however we do not present them since the achieved performance gain was not significant. First of all, we can see that the index-based execution time, even for the most complex queries, is low, less than 0.7 seconds for this large dataset (K=10). As to the cursor selection methods, RR generally proves to be the worse performing strategy and is particularly outperformed by the others in case of queries having many edges with different selectivity (such as Q3 and Q4). MS strategy is the most efficient in all situations (less than 0.3 seconds); NB performance is close but equally satisfying, also considering that, differently from MS, it does not require to know the cursors' size to work, thus possibly proving more versatile in some specific implementation scenarios. The scalability w.r.t. K also proves encouraging for all queries, with a computation time increase of only 10% to 30% in going from K=5 to K=10. Even if, due to lack of space, we are unable to show the complete figures, these scalability results are also confirmed by the other tests we performed for K values up to 100, on the queries retrieving a large number of results (such as Q4).

	%compl answers			#sorted accesses		
Query	**RR**	**NB**	**MS**	**RR**	**NB**	**MS**
Q1	43%	87%	93%	20	15	12
Q2	36%	76%	82%	25	18	16
Q3	10%	7%	41%	98	102	26
Q4	2%	8%	33%	105	67	23
	#random accesses - seq mode			**#random accesses - index mode**		
Query	**RR**	**NB**	**MS**	**RR**	**NB**	**MS**
Q1	22002	12385	10493	28	13	12
Q2	81003	37945	29455	31	16	13
Q3	172964	125881	92743	47	29	24
Q4	816162	249449	136068	412	154	58

Table 2: Details on index accesses for DBLP-L

Tab. 2 completes the picture by comparing the different strategies, for K=5, in terms of the percentage of completed answer computations and the number of required disk accesses, both sorted and random (for random access we show both sequential and standard indexed modes, for completeness). Notice the high percentage of completed answer computations, specifically for MS, meaning that the time spent in starting useless answer computations is minimized. Further, the MS strategy also provides the lowest number of disk accesses, thus justifying the previously examined figures.

We close our experimental discussion with a positive note on the scalability of the approach w.r.t. dataset size: by considering DBLP-XL (nearly 10 times larger then DBLP-L), execution time for Q1 to Q4 is less than tripled, thus proving a satisfying efficiency also from this point of view.

8. RELATED WORK AND CONCLUSIONS

The works which are more related to ours are [12, 21].

As far as we know, NAGA [12] is the first work which addresses the need of semantic query capabilities which go beyond keyword-based search as a key issue when searching for knowledge. Nevertheless, several differences exist with our work. NAGA relies on the YAGO Web-derived knowledge base [17] which consists of 16 millions facts extracted from semi-structured Web sources and ontologies. The work introduces a data model which is similar to ours. However, query formulation and answering is very different. First of all, the system does not allow for semantic approximations on query nodes' and edges labels. Then, edge approximation is expressed through the use of regular expressions over relationships as query edges' labels. Labels of matching paths must satisfy the given regular expression, thus following a pure syntactic approach. In very complex databases like those, for instance, in the biological field, finding data which exactly matches a complex regular expression may be a real chance. On the other hand, simple regular expressions, i.e., made of a single label, are not approximated (except for the `isA` relationship). Furthermore, answers to relatedness queries, i.e., queries containing edges labeled by the special keyword `connect`, return nodes which are connected through any path in the data. As discussed in Section 3.2, topological connection does not imply that the data retrieved is meaningfully related. To overcome these limitations, our model relies on the Semantic Relatedness relation SR to exclude misleading results. The ranking model proposed in [12] consists of a really valuable framework. However, it is orthogonal to ours since it follows a probabilistic approach which exploits some knowledge of the underlying dataset. Our ranking model instead relies on scoring functions which evaluate the approximations occurring at both data nodes and data edges. Finally, in [12] query processing is discussed to a limited extent, and only hints are given about the specific data structures and algorithms used to implement the system. The work in [21] is the first that exploits schema information to derive the concept of Meaningful Schema Pattern, with similar objectives as our Semantic Relatedness relation. However, the proposed solution is targeted for relational and XML data, and can be considered as an XML-like instantiation of our general model.

Much research efforts have focused on querying graph databases. Some works [8, 9, 11, 14, 16] have the main goal of investigating efficiency issues, and/or they are limited to keyword-based search. [18, 20] go beyond this query paradigm and they both support approximate subgraph matching. However, they only make syntactic considerations to evaluate the degree of structural approximations on query connections. In all these works the semantic relatedness of the connected data is not discussed. In [22] query relaxations are applied to malleable schemas. Approximations are achieved by query expansion techniques based on the discovery of correlations of attributes and relationships in the data. However, query relaxations are not concerned with entities' labels, and thus the user must know the schema to start the query. Furthermore, approximations on relationships are limited to edge substitution, thus not considering structural relaxations to paths. A further relevant work is [5] which emphasizes the need of supporting flexible query answering over heterogeneous data sources. However, the expressive power of predicate queries is limited. For instance, Query 2 in our reference example can not be expressed. Then, the work mainly focuses on indexing aspects, and approximations on queries are limited to the identification of synonyms.

In this paper we presented a model for supporting approximate queries on graph-modeled data. It is general and flexible enough to deal with data of different kinds besides the one shown in this paper, e.g., multimedia data. This does not affect the query answering mechanism proposed, rather it only impacts on the indices used to access the data.

The main strengths of our proposal are: 1) we abstract from a specific data model, rather our approach is general in that it gracefully accommodates several data models from other approaches; 2) we present a query answering framework which supports approximations on the vocabulary as well as on the structure of a query. Our approach is focused on a *flexible query matching* mechanism. This is orthogonal to approaches like e.g., [2, 22] which adopt a different perspective, by focusing on generating query relaxations to be matched exactly in the data. 3) As to structural approximation, we introduce the notion of Semantic Relatedness relation, which overcomes the limitations of simple topological approximations by allowing *semantically meaningful* relaxations only.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE*, 2002.

[2] S. Amer-Yahia, L. Lakshmanan, and S. Pandit. FleXPath: Flexible Structure and Full-Text Querying for XML. In *SIGMOD*, pages 83–94, 2004.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, 2002.

[4] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, 2003.

[5] X. Dong and A. Halevy. Indexing dataspaces. In *SIGMOD*, 2007.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, 2001.

[7] G. Furnas, T. Landauer, L. Gomez, and S. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, 1987.

[8] L. Guo, J. Shanmugasundaram, and G. Yona. Topology Search over Biological Databases. In *ICDE*, 2007.

[9] H. He, H. Wang, J. Yang, and P. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[10] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE Trans. Knowl. Data Eng.*, 18(4), 2006.

[11] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional Expansion For Keyword Search on Graph Databases. In *VLDB*, 2005.

[12] G. Kasneci, F. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. NAGA: Searching and Ranking Knowledge. In *ICDE*, 2007.

[13] C. Leacock and M. Chodorow. Combining Local Context and WordNet Similarity for Word Sense Identification. In C. Fellbaum, editor, *WordNet: An Electronic Lexical Database*, pages 256–283. MIT Press, 1998.

[14] G. Li, B. Ooi, J. Feng, J. Wang, and L. Zhou. EASE: An Effective 3-in-1 Keyword Search Method for Unstructured, Semi-structured and Structured Data. In *SIGMOD*, 2008.

[15] F. Mandreoli, R. Martoglia, and E. Ronchetti. Versatile Structural Disambiguation for Semantic-Aware Applications. In *CIKM*, 2005.

[16] A. Simitsis, G. Koutrika, and Y. Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *VLDB J.*, 17(1):117–149, 2008.

[17] F. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW*, 2007.

[18] Y. Tian and J. Patel. TALE: A Tool for Approximate Large Graph Matching. In *ICDE*, 2008.

[19] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD*, 2005.

[20] X. Yan, P. Yu, and J. Han. Substructure Similarity Search in Graph Databases. In *SIGMOD*, 2005.

[21] C. Yu and H. Jagadish. Querying Complex Structured Databases. In *VLDB*, 2007.

[22] X. Zhou, J. Gaugaz, W.-T. Balke, and W. Nejdl. Query relaxation using malleable schemas. In *SIGMOD*, 2007.