A Personal Perspective on Keyword Search over Data Graphs*

Yehoshua Sagiv
The Rachel and Selim Benin School of Computer Science and Engineering
Edmond J. Safra Campus
The Hebrew University of Jerusalem
Givat Ram, Jerusalem 91904, Israel
sagiv@cs.huji.ac.il

ABSTRACT

Theoretical and practical issues pertaining to keyword search over data graphs are discussed. A formal model and algorithms for enumerating answers (by operating directly on the data graph) are described. Various aspects of a system are explained, including the object-connector-property data model, how it is used to construct a data graph from an XML document, how to deal with redundancies in the source data, what are duplicate answers, implementation and GUI. An approach to ranking that combines textual relevance with semantic considerations is described. It is argued that search over data graphs is inherently a two-dimensional process, where the goal is not just to find particular content but also to collect information on how the desired data may be semantically connected.

Categories and Subject Descriptors: H.2.4[Database Management]: Systems—*Textual databases*; H.2.5[Database Management]: Heterogeneous Databases; H.3.3 [Information Storage and Retrieval]: Info. Search and Retrieval—*search process*

General Terms: Algorithms, Design, Human Factors

Keywords: Keyword search, data graph, enumeration algorithm, information retrieval, graph search

1. INTRODUCTION

Keyword search over databases may have started in order to deal with traditional databases that also have free text, but nowadays it faces broader challenges. Traditionally, database systems provide query languages for finding answers at a very fine granularity with a high degree of precision. Information retrieval deals with documents that are typically produced by different people. Traditional keyword search returns whole documents that are relevant to some extent. Databases require a considerable effort to organize the data at both the logical and physical level, in order to effectively support query languages. Information retrieval can be applied more easily to a collection of documents.

Data sources (and not just documents) are being generated at an exponential rate in a manner that "does not allow top-down design (or even overall design)" [3]. In many cases, those sources also include free text. Traditional query language are not effective, because there is no overall design and the data is inherently heterogeneous. Conventional information retrieval lacks the ability to take advantage of the available data semantics. Thus, new paradigms are needed, and some synergy of query languages and information retrieval is a promising approach.

Graphs are a good way of representing data that is created in an autonomous, distributed way. Most data models (such as relational databases, XML and RDF) can be easily represented as graphs. Nodes can store data at a variable granularity (that is not necessarily uniform across the graph), and their contents could be semistructured or even textual. Edges stand for semantic relationships between pairs of nodes.

Keyword search over data graphs enjoys the advantage of a very simple way of querying (just a set of terms, as in information retrieval). It also provides the main feature of database query languages, namely, ability to produce results with some semantic information, because answers are subtrees or subgraphs and, hence, have a structure.

There are three major issues to deal with when designing a system for keyword search over data graphs. First, it is not easy to design algorithms for evaluating search queries, because an answer is a subtree or a subgraph, rather than just a connected set of tuples. Second, answers must be ranked. Substantial work has been done in information retrieval on ranking and there are effective methods with foundations that are both theoretical and empirical (e.g., language models [32]). Ranking based on the relative importance and relevance of entities and relationships is not well understood. It is even less clear how to combine textual and semantic rankings. The third issue includes various aspects of a practical implementation, such as efficiency, scalability and human factors (e.g., a suitable GUI for displaying answers that are subtrees or subgraphs). The rest of the paper covers these important issues.

This paper offers a personal perspective and is not intended to be a survey. It is organized as follows. Section 2 proffers a retrospect on how keyword search over data graph might have evolved out of earlier research. Section 3

*This work was supported by the Israel Science Foundation (Grant No. 1632/12).

EDBT/ICDT '13, March 18–22 2013, Genoa, Italy

defines the formal notions of data graphs, queries and answers. It also discusses algorithms for enumerating answers with an emphasis on three important properties: correctness (i.e., ability to generate all the answers), efficiency and ability to enumerate in some desired order. The algorithms we discuss are of the type that works directly on the data graph (as in [5, 19]), rather than by first generating expressions and then evaluating them (e.g., [4, 17]). Section 4 is about the system we have implemented. In particular, we describe the object-connector-property data model and how it is used to construct a data graph from an XML document. Data graphs can also be built easily from other types of data (e.g., relational databases and RDF), but we prefer to do that from XML. Section 4 also describes how to deal with redundancies in the source XML document (which are likely in real-world datasets) and how to define duplicate answers. In addition, Section 4 discusses various aspects of the implementation and the GUI. Section 5 deals with the issue of ranking. It describes our particular approach to a ranking that combines relevance of the text with semantic importance, where the former depends on the query at hand whereas the latter does not. In Section 6, we argue that finding information in a data graph is inherently a two-dimensional search. That is, we have to find not just particular answers but also relevant patterns (i.e., expressions) that may indicate the semantic structures (rather than specific contents) of the information we are looking for. We conclude in Section 7.

2. A RETROSPECT

The aim of keyword search in information retrieval is to find relevant documents. Database queries are geared toward providing full and accurate answers to precise queries. The universal relation (e.g., [10, 34, 35]) might be perceived as an early attempt to combine the two. In that paradigm, a query is just a set of terms, which are attribute names. The goal is translate those terms into a relational query over the database. In retrospect, the universal-relation approach has not been widely used in practice, because the emphasis was on finding *the* correct translation of the attribute names to a relational query. This approach might have been more successful if it had been oriented toward finding relational queries that are (just) relevant to the given attribute names. An early system for keywords search over XML is [9]. That system also suffered from the desire to determine *the* correct way of answering a query and was limited to a data in the form of tree (rather than a graph).

A major obstacle in the development of a universal-relation interface was the need to handle incomplete information. In other words, even if one is able to translate correctly a set of attributes into a join of several relations, the result may be empty due to missing data. The notion of *full disjunctions* [11], also called *maximal answers*, was developed to tackle this problem, and several algorithmic solutions were found (e.g., [8,33]). The difference between a full answer and a maximal answer is that the former is a set that includes exactly one tuple from each relation, whereas the latter is a maximal set consisting of at most one tuple from each relation. In both cases, all the tuples of the set must be join consistent. Hence, an answer is maximal if one cannot add any tuple (from a relation not present in the answer) and still have a join consistent set. However, even a solution to the problem of incomplete information is not quite sufficient when dealing with heterogeneous data that is created in an autonomous, distributed manner, as we explain next.

In the database way of finding information, we first formulate a query that gives the correct relationship between the entities of interest and then perform some selections. For example, if we want to find whether Smith is a teacher of Jones, we first need to join several relations to get the correct connection between students and their teachers. Then, we select tuples where the teacher name is Smith and the student name is Jones. Common optimizations may reverse the order and do selections before joins. Nonetheless, in principle, we formulate a query that is good not just for Smith and Jones, but for any pair of a teacher and a student—all we have to do is a simple substitution that replaces the terms "Smith" and "Jones" with some other names. This is drastically different from keyword search that zooms in, at first, on documents that are relevant to Smith and Jones, and only then does it become apparent whether Smith is a teacher of Jones. Of course, it may not be easy to find the particular information we are looking for (namely, whether Smith is a teacher of Jones) by inspecting documents that are relevant to these two individuals. If Smith indeed teaches Jones, we are likely to find rather quickly concrete evidence to this fact (especially if we add the term "teach" to the query). But how can we be sure that Smith does not teach Jones? Inspecting all relevant documents (and verifying that none of them pertains to the fact that Smith teaches Jones) may require too much time. This is one indication to the need for a two-dimensional search that we mentioned in the introduction and is further explained in Section 6.

To summarize thus far, a database query is good not just for finding a particular unit of information (such as "Smith teach Jones")—it can actually be used (with minor modification) for getting every fact of the same type (e.g., that one person is the teacher of another). This characteristic (along with the need to enforce some constraints) is the cause of why a good database design is rather elusive. It is even harder to preserve this property when dealing with data integration from heterogeneous sources and data exchange. Nevertheless, a lot of database research has been fueled by the belief that a database must have that characteristic at all cost. Is it really needed? Undoubtedly, a database of a bank must have this property. However, in an era where the amount of information grows exponentially, it is unrealistic to expect that all databases will have that property. Keyword search is an attractive alternative, because it concentrates on specific terms (e.g., "Smith" and "Jones") without trying to find a way of answering all queries of the same type (e.g., does one person teach another?). However, keyword search is oblivious to the semantic structure of the data and, therefore, is rather limited in its ability to concisely retrieve precise information. In this paper, we describe ongoing work that aims at amalgamating the two paradigms: keyword search and database queries.

3. FORMAL MODEL AND ALGORITHMS

In this section, we describe the formal notion of a data graph and define what are answers to keyword queries. We also discuss several algorithms for generating answers.

3.1 Formal Data Graphs

Formally, a directed *data graph* has two types of nodes: *structural* nodes and *keyword* nodes. The former corre-

sponds to the basic components of the data. For example, in a data graph derived from a relational database, the structural nodes are tuples. The edges correspond to foreign-key references among tuples. A data graph may also be derived from an XML document. In this case, the structural nodes may correspond to elements of the XML document (alternatively, there could also be a separate structural node for each attribute). The edges indicate the relationships that are derived either from id references or nesting (i.e., an element appearing inside another one). Each structural node has some associated text, namely, its *content*. In the case of a data graph derived from a relational database, the content of a structural node is the same as that of its corresponding tuple. That content comprises both the attribute names and their values. When the data graph is obtained from an XML document, the content associated with a structural node is that of its corresponding element, but excluding the content of sub-elements (or attributes that are represented by other nodes). Each structural node also has a *label*. There is no need to assume that a label uniquely identifies a node. The edges have neither contents nor labels.

The keyword nodes are for the terms that comprise the contents of the structural nodes. Each keyword has its own node. We use the keyword itself to identify its node. From each structural node s, there is an edge to every keyword node k, such that k appears in the content of s. By a slight abuse of terminology, we may refer to a "keyword node" just as "keyword."

As an example, Figure 1 shows a data graph (where keyword nodes are shown as words). This data graph is derived from a small part of the Mondial relational database,[1] so its structural nodes correspond to tuples (and their labels are tuple identifiers). Note that in our model, each keyword is represented by a single node and all its occurrences point to that node (hence, this is not a restriction). Moreover, keywords have only incoming edges.

The nodes and edges of a data graph have nonnegative weights The *length* of a path in the graph is the sum of the weights along the path. The *height* of a directed subtree T is the maximum length of any path from the root of T to a leaf. The *weight* of T is the sum of weights of all the nodes and edges of T.

3.2 Queries and Answers

In our model, a *query* is a set of keywords (or terms) Q. We assume that a query has at least two keywords. An *answer* to Q is a directed subtree T of the data graph G, such that T contains all the keywords of Q and is non-redundant. The occurrences of the keywords are represented by their corresponding nodes. Thus, the subtree T must include the nodes for all the keywords of Q. Non-redundancy of T means that T is *reduced* with respect to Q, namely, T contains the nodes for all the keywords of Q, but has no proper directed subtree that also includes all of them.

Since an answer T is a directed subtree, it follows that T has a node r, called the *root*, such that every other node of T is reachable from r through a unique directed path. Alternatively, we can characterize an answer as a subtree T of G, such that the root has at least two children and the set of leaves is exactly Q.

Example 1. Consider the data graph G_1 of Fig. 1. Let the query Q consist of the keywords `Belgium`, `EU` and `Brussels`, that is, $Q = \{\texttt{Belgium}, \texttt{Brussels}, \texttt{EU}\}$. The only two answers are the subtrees A_1 and A_2 shown in Fig. 2.

Now, consider the query $Q' = \{\texttt{Brussels}, \texttt{EU}\}$. A_1 and A_2 are *not* answers to Q', because each one contains a proper subtree that also includes the given keywords.

Following [20, 22], we can also view a data graph G as undirected. In this case, there are two types of answers to a query Q. First, an *undirected answer* is an undirected subtree that contains all the keywords and is reduced (i.e., has no proper undirected subtree that also contains all of them). Second, a *strong answer* is an undirected answer, such that all its keyword nodes are leaves (that is, we cannot use a keyword node in order to connect two structural nodes). When we just say "answer," we usually mean "directed answer" as defined earlier.

3.3 Generating Answers

Given a data graph G and a query Q, our goal is to enumerate the answers to Q by increasing weight. Thus, we follow the principle that a smaller subtree indicates a stronger relationship between the keywords.

For efficiency, we would like the enumeration to be with *polynomial delay* [18], which means the following. There is a polynomial $p(n)$, where n is the size of the input (i.e., G and Q), such that the time needed to produce the next answer (since the previous one was printed) is always bounded by $p(n)$. The *initial* delay (i.e., the running time until the first answer is printed) and the *final* delay (i.e., the time interval between printing the last answer and terminating) are also bounded by $p(n)$. In particular, if there are no answers at

[1] `http://www.dbis.informatik.uni-goettingen.de/Mondial/#SQL`

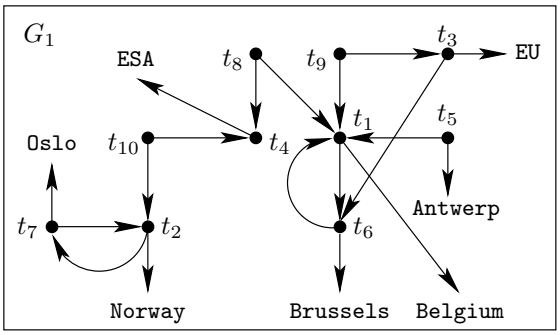

Figure 1: Data graph G_1

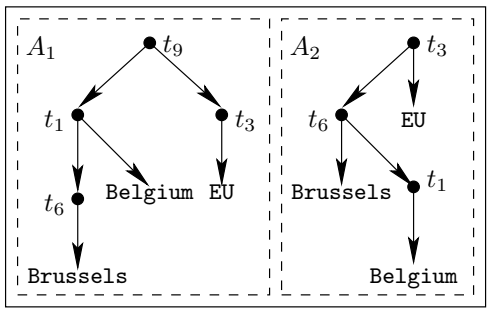

Figure 2: Answers A_1 and A_2

all, then the algorithm must stop after a polynomial number of steps. To improve efficiency, we may want to consider enumeration in an approximate order. This type of enumeration is formally defined in [22].

An important property of an algorithm for enumerating answers is the ability to generate all of them. It is challenging to design an algorithm that does not miss answers. In fact, this property is not realized by most of the existing algorithms that work directly on the data graph (see [12] for more details about this issue).

In summary, an algorithm for enumerating answers should have three important properties: *correctness*, *efficiency* and *order*. The first means that an enumeration algorithm should be capable of generating all the answers. The second refers to the guaranteed running time (e.g., polynomial delay). The third means that the enumeration should be in some desirable order (e.g., by increasing weight).

We would like to realize all of these three properties both theoretically and practically. However, some comprises might be necessary, especially when considering the practical point of view.

The work of [21, 23] was the first to give algorithms that are correct and provably efficient (i.e., run with polynomial delay); however, the enumeration is in an arbitrary order. This work includes algorithms for all three variants of answers (i.e., directed, undirected and strong).

The work of [22] was the first to give algorithms that realize all of those three properties. That is, they enumerate all answers, with polynomial delay and by increasing weight. Since this necessarily involves solving the Steiner-tree problem, it is essential to assume that the query is of fixed size. However, the exponential factor (in the polynomial that bounds the delay) is of the form c^k, where c is a small constant and k is the number of keywords in the query. Practically, this is much better when compared to processing relational queries for which the exponential factor is of the form n^q, where n is the size of the database and q is the size of the relational query. In [22], they also give algorithms for enumerating in an approximate order by weight. These algorithm achieve polynomial delay without assuming that the query is of fixed size. All three types of answers (i.e., directed, undirected and strong) are considered in [22].

Since enumeration by increasing weight is not easy to achieve in practice, an alternative approach, which was introduced in [5, 19], is to enumerate (directed subtrees) by increasing height. The rationale is that enumerations by increasing weight and height are reasonably correlated. Therefore, we can achieve an enumeration that is close to the desired order as follows. Suppose that we want n answers. We enumerate by increasing height a larger number of answers (say $10n$), and then sort them by increasing weight and take the first n. This approach was adopted in [12]. Their algorithm enumerates answers in a 2-approximate order by increasing height with polynomial delay. In contrast to [5, 19], the algorithm of [12] does not miss answers and enumerates with polynomial delay. The work of [12] considers only directed answers.

3.4 Practically Efficient Algorithms

We have mentioned that the important properties of an enumeration algorithm are efficiency and ability to generate all answers in an order that is close to the desired one. However, an algorithm with a theoretical guarantee of efficiency (e.g., polynomial delay) is not necessarily better in practice than algorithms without such a guarantee. The reason is that theoretical efficiency is typically for the worst case, whereas in practice we measure (in experiments) the average behavior.

As mentioned earlier, a fast algorithm for generating directed answers was introduced in [5]. It uses Dijkstra's shortest paths iterators—one per keyword of the query Q. When these iterators (which start at the keyword nodes) meet in a structural node v, then we should examine the directed subtree T consisting of the shortest paths from the keywords of Q to v. If the root of T has at least two children, then it is a directed answer. This is a fast algorithm in practice, even though its delay is not always polynomial. A more severe problem with this algorithm is the inability to generate all answers. In particular, that algorithm cannot generate an answer T rooted at v, such that some paths of T from the keyword nodes of Q to v are not the shortest. The algorithm of [12] also uses shortest paths iterators as in [5], but it does so within the Lawler-Murty's procedure [26, 31] and couples those iterators with an additional algorithm for finding a directed subtree having a root with at least two children, when the shortest-path iterators do not produce such a subtree. Thus, the algorithm of [12] is both correct and theoretically efficient. However, we have developed another correct algorithm that is considerably faster in practice, even though its worst-case delay is exponential. That algorithm enumerates all directed answers by increasing height and is given in [15]. Next, we briefly describe its main ideas.

First, we can modify Dijkstra's algorithm so that it enumerates all simple (i.e., acyclic) paths by increasing weight from s to t. To do that, the queue has to store paths rather than nodes. A path is represented by a linked list in reverse order from its last node u to s. In each iteration, we remove the shortest path p from the queue. Let u be the last node of p. We create new paths by adding the outgoing edges of u at the end of p, and insert them into the queue provided that they are acyclic. This is a rather simple modification of Dijkstra's algorithm that was already presented in [36]. Now, we can modify the algorithm of [5] by using an all-simple-paths (rather than a shortest-path) iterator from each keyword node of Q. When paths from all the keywords of Q meet at a node r, we have to check that they create a subtree and that r has at least two children. If so, we have a directed answer. Even this straightforward generalization of Dijkstra's is substantially more efficient than the algorithm of [12]. Its efficiency can be enhanced by the following idea.

For each keyword node k of Q, we apply the following idea. At each node u, only the first path from k that reaches u can continue to the neighbors of u. Paths from k that reach u subsequently are frozen at u. Those paths are unfrozen and continue to the neighbors of u only when discovering that u is on a path from k to a node r, such that r is reachable from all the keywords of Q (hence, r could potentially be the root of a directed answer). This is just a rough overview and the actual algorithm is quite intricate; for a full description, see [15].

3.5 Parallel Algorithms

In the era of ubiquitous multi-core computers, using serial algorithms is a gross under utilization of the resources at hand. However, designing an effective parallel algorithm

is not an easy task. In this section, we give an overview of the work of [13] on optimizing and paralleling ranked enumeration. As mentioned earlier, the algorithm of [12] uses Lawler-Murty's procedure [26, 31]. The techniques of [13] actually apply to that procedure, rather than to a specific algorithm. Hence, the work of [13] is applicable to any ranked enumeration that uses Lawler-Murty's procedure.

Lawler-Murty's procedure works as follows. Essentially, it is a reduction of an enumeration problem to a sequence of optimization problems (i.e., each one entails finding the optimal solution) under constraints. The procedure stores solutions to those optimization problems in a priority queue. In each iteration, the top T of the queue is removed and printed, namely, T is the next element (e.g., an answer to a given query) in the ranked enumeration. In addition, several new optimization problems under constraints are created from T (in a way that also incorporates the constraints under which T was obtained). We call them the *spawned* optimization problems. Each one of those problems is solved and its solution is inserted into the queue before proceeding with the next iteration.

To apply Lawler-Murty's procedure to a specific ranked enumeration, one has to determine the type of constraints to be used and to supply a subroutine S for solving the optimization problem under those constraints. In typical applications, S is iterative and has the property of a *progressive lower bound*, which means the following. At the end of each iteration, S provides a progressively better lower bound on the cost (e.g., weight or height) of the optimal solution.

The techniques of [13] use progressive lower bounds as follows. When the lower bound l of a spawned optimization problem P exceeds the value v at the top of the queue, P is no longer *relevant* to the current iteration. When all the active spawned optimization problems are no longer relevant, the next iteration can start. In particular, the solution at the top of the queue can be removed and printed, even though not all the spawned problems of the previous iteration have yet been solved.

A thread pool is used for solving spawned optimization problems. The *main* task (which has a dedicated thread) monitors the progressive lower bounds of the active spawned problems and decides if it is safe to continue with the next iteration. If so, the main task removes and prints the top T of the queue, and also creates the spawned optimization problems implied by T. This technique is called *relevance monitoring*.

The work of [13] also introduces the idea of *freezing* a spawned optimization problem. This is instrumental in not overloading the thread pool. When the progressive lower bound l of an optimization problem P exceeds the current value v at the top of the queue, P can be frozen. The computation of P should resume when the top of the queue becomes greater than l. Note that we want to freeze an optimization problem only when there is no available thread to work on it.

Combining the techniques of relevance monitoring and freezing in an effective parallel algorithm is not straightforward. In particular, subtle synchronization is needed between the various threads, the frozen optimization problems and the queue, and it should be done without either incurring too much overhead or doing excessive locking (which may reduce the degree of parallelism). The details are in [13].

An additional technique of [13] is *early* freezing. The idea is to create spawned optimization problem from a solution T when T is inserted into the queue (rather than when it is removed). Those spawned problems are frozen upon creation. Early freezing is needed in order to eliminate a situation of not having enough spawned optimization problems to work on (which means that the thread pool is underutilized). For example, in the context of parallelizing the algorithm of [12], this situation is especially likely when the query has a small number of keywords and, hence, each answer creates only a few spawned optimization problems. Early freezing entails a significant change in the logic of Lawler-Murty's procedure, and requires a more subtle synchronization than the parallel variant (mentioned earlier) that uses freezing. The experiments of [13] show that early freezing delivers a speedup that is close to linear when using eight cores.

Figure 3: An answer to "`Paris, Strasbourg`"

4. THE SYSTEM

In this section, we describe practical aspects of keyword search over data graphs. We start with a description of our data model. Then we explain how to build a data graph from an XML document. We discuss the issues of how to handle data redundancies in the source XML document and how to eliminate duplicate answers. Finally, we describe the implementation of our system and the GUI. This section is an overview of the work done in [14].

4.1 The OCP Model

An essential feature for keyword search over data graphs is a GUI that facilitates quick understanding of what an answer means. Showing just an XML fragment or a set of tuples makes it very hard to grasp the essence of an answer. The *object-connector-property* model was develop in [1] for rendering XML fragments that are answers to keyword search over data graphs. Figure 3 shows an answer to the query "`Paris, Strasbourg`." Gray rectangles represent objects and red circles depict connectors. Each object has a *title*, which is shown in blue, and a *type*, which appears inside parentheses. For example, the top rectangle in Figure 3 represents the object entitled `Strasbourg` that has the type `city`. A connector has only a type, which is written in red next to the circle. Occurrences of the query keywords are underlined in green. Edges represent connections (i.e., relationships) between objects. An *implicit connection* is just an edge between two objects, and its meaning is clear from the context. For example, the edge connecting `Strasbourg` and `France` simply means that the former is a city of the latter. An *explicit connection* between a pair of objects is represented by a pair of edges that go through a connector. The type of the connector indicates the meaning of the relationship. For example, in Figure 3, the pair of edges

```
<!ELEMENT country (name,ethnicgroups*,border*,province*)>
<!ATTLIST country car_code ID #IMPLIED
          capital IDREF #IMPLIED
          memberships IDREFS #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT ethnicgroups (#PCDATA)>
<!ATTLIST ethnicgroups percentage #PCDATA #REQUIRED>
<!ELEMENT border EMPTY>
<!ATTLIST border country IDREF #REQUIRED
          length CDATA #REQUIRED>
<!ELEMENT province (name,city*)>
<!ATTLIST province id ID #REQUIRED>
<!ELEMENT city (name)>
<!ATTLIST city id ID #REQUIRED
          country IDREF #REQUIRED>
<!ELEMENT organization (name)>
<!ATTLIST organization id ID #REQUIRED>
<!ELEMENT sea (name)>
<!ATTLIST sea id ID #REQUIRED
          country IDREFS #IMPLIED>
```

Figure 4: A snippet of the Mondial DTD

connecting `France` and `Paris` through the connector `capital` means that the latter is the capital of the former.

The OCP model assumes that the source data is given as an XML document. The first task is to identify the objects, connectors and properties of that document by examining the DTD. A set of rules for doing that was first developed in [1]. Here, we describe the approach of [14], which uses fewer, simpler rules as follows.

- An object is any XML element with an ID attribute.
- A connector is either one of the following two.
  - An element that has either an IDREF or IDREFS attribute, and does not have an ID attribute.
  - An IDREF or IDREFS attribute.
- A property is either one of the following two.
  - An element such that none of its attributes is either ID, IDREF or IDREFS (i.e., an element which is neither an object nor a connector).
  - An attribute that is neither IDREF nor IDREFS.

Example 2. Consider the DTD that is shown in Figure 4 (which is a part of the DTD of the Mondial XML document[2]). For each component (i.e., element or attribute) we want to determine whether it is an object, a connector or a property. The element `country` is an object, because its attribute `car_code` is defined as ID in the DTD. Similarly, the elements `province`, `city`, `organization` and `sea` have the attribute `id` that is defined as ID; hence, all of them are objects. The attributes `capital` and `memberships` of the element `country` are connectors, because the former is defined as IDREF and the latter—as IDREFS. Similarly, the attribute `country` is a connector in all of the elements where it appears (i.e., `border`, `city` and `sea`), because each of those occurrences is defined as either IDREF or IDREFS. The element `border` is a connector, because its attribute `country` is defined as IDREF and it has no other attribute that is defined as ID. The elements `name` and `ethnicgroups` are properties, because they have neither an ID, an IDREF nor an IDREFS attribute. Finally, the attributes `car_code`, `percentage` and (all the occurrences of) `id` are properties, because they are defined as neither IDREF nor IDREFS.

[2] `http://www.dbis.informatik.uni-goettingen.de/Mondial/#XML`

```
1  <country car_code="IL" capital="cty-Israel-Jerusalem"
           memberships="org-Interpol org-UN">
2    <name>Israel</name>
3    <ethnicgroups percentage="82">Jewish</ethnicgroups>
4    <border country="RL" length="79"/>
5    <province id="prov-cid-cia-Israel-2">
6      <name>Central</name>
7      <city id="cty-Israel-Jerusalem" country="IL">
8        <name>Jerusalem</name>
9      </city>
10   </province>
11 </country>
12 <country car_code="RL" memberships="org-UN">
13   <name>Lebanon</name>
14   <border country="IL"/>
15 </country>
16 <organization id="org-Interpol">
17   <name>Interpol</name>
18 </organization>
19 <organization id="org-UN">
20   <name>UN</name>
21 </organization>
22 <sea id="sea-Mittelmeer" country="IL">
23    <name>Mediterranean Sea</name>
24 </sea>
```

Figure 5: A fragment of the Mondial document

Looking now at the XML fragment of Figure 5 that conforms to the DTD of Figure 4, we can see that lines 12–15 define a particular object that has two properties: `car_code` and `name`. That object also embeds two connectors: `memberships` and `border`, where the latter has a connector of its own, namely, its attribute `country`.

As already mentioned, each object, connector and property has a *type*, which is just the name of the corresponding element or attribute in the DTD. In the above example, the type of (either the object or connector) `country` is `country`, the type of the property `name` is `name`, etc. Each object has a *title*, which is the value of one of its properties. As a heuristic, the title is the value of a property having a type such as `name`, `header`, etc; however, for some XML documents, this heuristic may have to be modified. For example, lines 12–15 of Figure 5 define an object of type `country` that has the title `Lebanon` (which is the value of its property `name`). It is possible that some objects lack any title. In the next section, we use the types and titles in lieu of the labels of the formal data graph of Section 3.1.

The above example illustrates several points. First, both objects and connectors may have properties (e.g., `name` and `length` are properties of an object and a connector, respectively). Second, properties can be nested (e.g., `percentage` is a sub-property of `ethnicgroups`). Third, a connector may nest another connector (e.g., the connector `border` contains the connector `country`). Obviously, objects may be nested (e.g., `province` is a sub-object of `country`).

4.2 Constructing the Initial Data Graph

We now describe how to build a directed data graph from a given XML document, after we have determined the objects, connectors and properties using the rules of Section 4.1. This construction is taken from [14] and it comprises two phases. In this section, we show how to build the *initial data graph*. The second phase of adding more edges is discussed in the next section. Note that here we describe a

concrete implementation of the abstract notion of a data graph that was given in Section 3.1. We actually describe only the construction of the structural nodes and their edges. The keyword nodes are discussed in Section 4.6. Thus, by a slight abuse of terminology, when we say "node," we mean "structural node."

Due to the tree structure of an XML document, each element and attribute has (at most) one parent. To emphasize that this notion of a parent is defined by the XML document, we call it the *XML parent* and define it formally as follows. For an attribute a, the XML parent is the element to which a belongs. For an element e, the XML parent (if it exists) is the immediate super-element s in which e is nested. The notion of an XML parent implies naturally the meaning of an *XML child*.

For simplicity, we make two assumptions. First, if an element does not have an XML parent, then it is an object (i.e., it has an ID attribute).[3] Second, an element that is a connector cannot have a sub-element that is also a connector (but it may have an attribute which is a connector). The first assumption is rather natural, but the second is not always satisfied in real-world XML documents. The general case (namely, when the second assumption does not hold) is handled in [14].

In the data graph, there are structural nodes for all the objects and for some of the connectors. By a slight abuse of terminology, we do not distinguish between a node v and the object or connector for which v was created. That is, we may call v an object or a connector (rather than a node). We construct the data graph as follows. For each object o, we create a unique node and add a directed edge (o, o') for all XML children o' of o, such that o' is also an object. For a connector c, we do the following. If c is an element, we create a node for c and add a directed edge from the XML parent of c to c. If c is an attribute, we consider each IDREF i in the value of c. If i points to an object $\vec{o}$ that has the same type as c, then we add a directed edge from the XML parent of c to $\vec{o}$. If there is at least one i that does not satisfy that condition (namely, i points to an object of a different type), then we create a node v for c and add a directed edge from the XML parent of c to v. In addition, for each i, such that i points to an object $\vec{o}$ of a different type from that of c, we add a directed edge from v to $\vec{o}$.

We do not create nodes for properties. Since properties can be nested, we consider each maximal subtree t (of the XML parent hierarchy), such that t comprises only properties, and associate t with the XML parent of its root.

Example 3. In Example 2, we showed what are the objects, connectors and properties of the XML fragment of Figure 5. Figure 6 depicts the data graph constructed from that XML fragment. The nodes for objects are rectangles and those for connectors—ellipses. To avoid cluttering, only the title (but not the type) of an object is written inside its corresponding node. For a connector, its node includes the type. The solid edges are those created as explained earlier. The dashed edges will be discussed in the next section. Recall that lines 12–15 of Figure 5 define the object `Lebanon` of type `country`. A single node is created for `Lebanon` and its two properties (i.e., `car_code` and `name`). The object `Lebanon` embeds the connectors `memberships` and `border`. The connector `memberships` is an IDREF attribute that points to the object `UN` (which is defined in lines 19-21 of Figure 5). Since `memberships` is different from the type of `UN` (which is `organization`), we create a node for `memberships` and connect it by a directed edge to `UN`. Similarly, a node is created for the connector `border` of `Lebanon`. Line 14 of Figure 5 shows that `border` has the attribute `country`, which is a connector with a single IDREF that points to `Israel` (which is also of type `country`). Hence, we add a directed edge from (the node created for) `border` to `Israel` and no node has to be created for the connector `country` of line 14 of Figure 5. Note that a node of type `memberships` is created also for `Israel`, but it has two outgoing edges, because `Israel` is a member of two organizations. As a last example, lines 5–10 of Figure 5 define the object `Central` of type `province` that embeds the object `Jerusalem` of type `city`. Therefore, there is a directed edge from `Central` to `Jerusalem`.

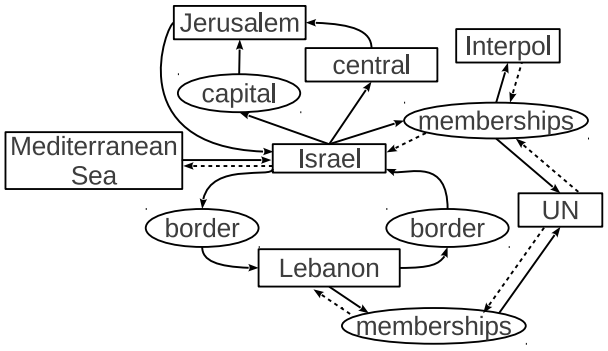

Figure 6: Data graph of the Mondial fragment

For simplicity, we assume that in a data graph that is constructed as described in this section, all the edges are between either two objects or an object and a connector; that is, there is no edge between two connectors. This is equivalent to requiring the following. If a connector c is an element of the given XML document, then each of its IDREF or IDREFS attributes a points only to objects that have the same type as a. We have also ignored the case that a connector has no outgoing edges and the possibility of some errors in the source XML document (e.g., an id reference that points to a non-existing element). As already mentioned, the general case is discussed in [14].

4.3 Adding Opposite IDREF Edges

The initial data graph (constructed in the previous section) cannot provide some answers that we would expect to get. For example, consider the data graph of Figure 6 with only the solid edges (which are those of the initial data graph). This graph does not have a subtree that includes `Israel`, `Lebanon`, `UN` and the two `memberships` nodes. So, when posing the query "`Israel`, `Lebanon`, `UN`" we would not get the answer that both `Israel` and `Lebanon` are members of the `UN`. A solution to this problem is to introduce edges in the opposite direction of existing ones. We could do it for all edges, but that may result in overwhelming the user with too many answers of a semantically weak nature. Therefore, we do it more judiciously as discussed in this section, based on the work of [14] (which also includes additional details that are not covered here).

[3]We need not assume that there is just one element without an XML parent.

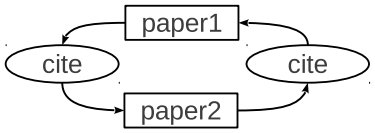

Figure 7: A data graph of papers and citations

Recall that an *explicit connection* between two objects o_1 and o_2 is a pair of edges (o_1, c) and (c, o_2) that pass through a connector c (note that in the initial data graph, the latter edge is due to an id reference). An *implicit connection* is a single edge (o_1, o_2) between two objects, and it is either an *implicit idref connection* or an *implicit nesting connection* depending on whether it is due to an id reference or nesting, respectively. When we just say "connection," we mean that it could be of any type.

For some of the connections that are not due to nesting, we add edges in the opposite direction. We use the data graph of Figure 6 as a running example. Recall that solid edges are those created for the initial data graph, as explained in the previous section. Dashed arrows are called *opposite idref edges* and we now describe the rules for creating them.

Suppose that e is an implicit idref connection from node o_1 to o_2. If the data graph does not already have a connection (using only solid edges) in the opposite direction, we add a directed edge from o_2 to o_1. For example, in Figure 6, we add the (dashed) edge from `Israel` to `Mediterranean Sea`. However, there is no need to add an edge from `Israel` to `Jerusalem`, because the data graph already has a connection from the former to the latter through the connector `capital`.

Now, suppose that there is an explicit connection from o_1 to o_2 consisting of the edges (o_1, c) and (c, o_2), namely, c is a connector. At first, we assume that the objects o_1 and o_2 have different types. We add the edges (o_2, c) and (c, o_1) if in the opposite direction, there is neither an implicit connection nor an explicit connection that uses the same connector type. For example, in Figure 6, for the explicit connection `Israel` → `memberships` → `Interpol`, we add the edges from `Interpol` to `memberships` and from `memberships` to `Israel`. However, for the explicit connection `Israel` → `capital` → `Jerusalem`, we do not add any edges, because there is already the implicit connection from `Jerusalem` to `Israel` in the initial data graph.

If o_1 and o_2 have the same type, then additional information is required. We need to determine whether the connection is *symmetric* or *asymmetric*. For example, for two countries, the connection that they have a common border is symmetric. That is, if we reverse the direction of the connection, its meaning remains the same. In contrast, for two papers, the connection that one cites the other is asymmetric. Namely, if the first paper cites the second, then the opposite connection is `cited_by` and not `cite`.

If the explicit connection (from o_1 to o_2 of the same type) is symmetric, then we treat it as in the case where o_1 and o_2 have different types. For example, for the explicit connection `Lebanon` → `border` → `Israel`, we do not add any edges, because there is already an explicit connection in the opposite direction through `border`.

Figure 7 depicts a situation that occurs in the data graph for the DBLP dataset. There is an explicit connection from `paper1` to `paper2` through `cite`, which is asymmetric. In this case, we add opposite idref edges from `paper2` to `paper1` through the same `cite` connector. We do that even though there is already a connection from `paper2` to `paper1` through another `cite` connector. The new connection through the opposite idref edges is really `cited_by` (rather than `cite`). Therefore, when presenting an answer that uses those opposite idref edges, we show the connector as having the type `cited_by` (rather than `cite`). In other words, `cited_by` does not exist in the data graph—it is only created at the presentation level.

Determining whether a connection is symmetric or asymmetric can be done interactively by consulting some users. It can also be done automatically as explained in [14].

We end this section with the following observation. In the initial data graph, a node of a connector has exactly one incoming edge (from its XML parent). Therefore, if a node of a connector has any incoming opposite idref edges, then it has exactly one outgoing opposite idref edge; otherwise, it has no outgoing opposite idref edges.

4.4 Data Redundancies

Real-world data often has redundancies. The Mondial XML document has quite a few. For example, Figure 6 shows that the information about borders is stored twice. There is a connection through `border` from `Lebanon` to `Israel` and vice versa. This redundancy is rather easy to detect, because the same type of connector is used for both directions. As explained in Section 4.3, we do not create opposite idref edges in this case. Sometimes each direction is through a different type of connector. For example, the Mondial XML document stores for each country information about its memberships in organizations, and for each organization it lists its members. Only the former is shown in the XML fragment of Figure 5 and in the data graph of Figure 6. This kind of redundancy is harder to detect, because one direction uses connectors of type `memberships` whereas the other is through the type `members`. Nonetheless, [14] shows how to detect this redundancy automatically. Hence, there is actually no need to add opposite idref edges for connections through `memberships` when building the data graph for the full Mondial XML document (and the same is true for `members`).

There are also cases of redundancies involving different connections in the same direction. The Mondial XML document stores twice the same relationship between a river and the countries through which it flows. First, each `river` has an IDREFS attribute `country` that points to the relevant countries. Second, for each `river`, Mondial stores separately each country and those of its provinces that contain the river. In this case, it seems reasonable not to include the former in the data graph (i.e., to remove the attribute `country` of `river`). Automatically detecting this kind of redundancy can be done as explained in [14].

4.5 Eliminating Duplicates

We use a directed data graph. In the graphical interface (e.g., Figure 8), the direction of the edges is not shown explicitly, but is from top to bottom. Due to opposite idref edges, when we get the answer of Figure 8, there would also be a similar answer but in the opposite direction (i.e., the root is `Ukraine` and `Dnepr` is the leaf). These two answers are essentially the same. As in [5], we use an undirected semantics for answers. Since we employ an algorithm that generates directed subtrees, we need to eliminate duplicates.

Figure 8: An answer for the query "`Dnepr Ukrainian`"

An answer is removed if it is the same as a previous one when viewing edges as undirected. Duplicate elimination is done after removing the keyword nodes (which are attached to the data graph when generating answers, as explained in the next section).

Eliminating duplicates just on the basis of on undirected semantics may not be enough. For example, in Figure 6, the information that `Lebanon` and `Israel` have a common border is stored in both directions using different connectors (although both are of the same type). Therefore, our system offers also the option of eliminating duplicates while viewing two connectors as the same node if their types are identical. In this way, we would get only one answer (rather than two) with the information that `Lebanon` and `Israel` have a common border. Note that this would eliminate many redundancies among answers that include the path `Lebanon` → `border` → `Israel` (or its reversal). This idea can be extended to cases where two different types of connectors are deemed the same (e.g., `memberships` and `members`). An alternative is to build the data graph so that the same connection between two objects is represented in both directions by the same connector (i.e., same physical node).

It should be noted that although duplicate removal assumes that edges are undirected, we still get fewer answers compared with the case of assuming that the initial data graph is undirected (and generating strong answers). We believe that in this way, we quickly cull many semantically weak answers, even if there are examples where this is not necessarily the case. In summary, we judiciously add opposite idref edges (as explained in Section 4.3), but eliminate duplicates based on an undirected semantics. We believe that this approach strikes the right balance between inundating the user with a lot of semantically weak answers, on the one hand, and missing important ones on the other hand.

4.6 Implementation and GUI

The data graph is stored in a DBMS. When the system is started, a *skeleton* of the data graph is loaded into main memory. The skeleton consists of the structural nodes with only their internal ids (generated by the system) and their edges. When a query Q is given, nodes are created for its keywords. Those nodes are attached to the data graph, based on the information in the DBMS. That is, an edge is created from each node v of the data graph to a keyword k of Q, provided that the content of v contains k. Note that the keyword nodes that are needed for generating answers to Q are only those for the terms of Q.

When parsing the source XML document, during the construction of the data graph, we also create for each node v an XML fragment that describes the full content (i.e., text) of

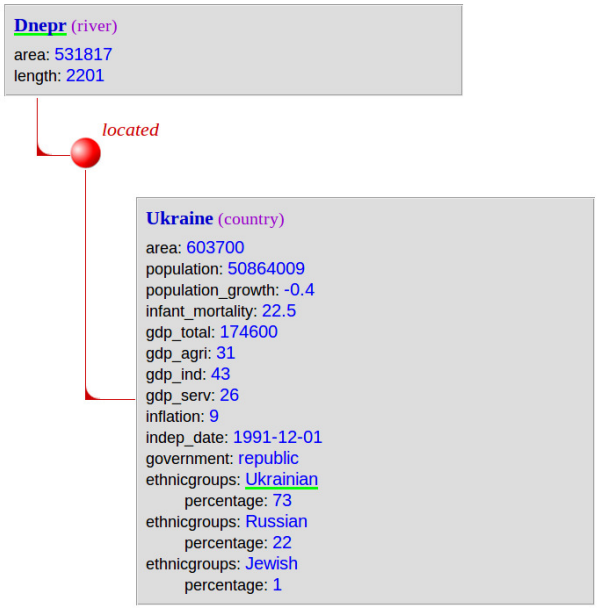

Figure 9: An expanded answer for "`Dnepr Ukrainian`"

v, that is, its type, title (if it exists), all the nested properties and their values, and any additional text (e.g., PCDATA). The XML fragment of each node is stored in the DBMS. Given an answer, we can easily combine the XML fragments of its nodes to obtain an effective graphical rendition.

Our approach of using *fat nodes*, by associating each object and connector with all its properties (even nested ones), results in a much smaller data graph, compared with having a separate node for each property. Thus, less main memory is needed and the traversal of the graph is faster when generating answers. As demonstrated in [2], fat nodes also facilitate an easy-to-understand graphical presentation of answers. The default display of an answer is illustrated in Figure 8. The keyword of the query are underlined in green. Only properties that include the keywords are shown. There is an option to expand an answer so that all its properties are displayed, as illustrated in Figure 9.

Recall that an answer is defined (in Section 3.2) as a reduced subtree with respect to the query. Next, we discuss cases where this definition does not yield (sufficiently) meaningful answers and explain how to handle them.

It is meaningless to show a connector without displaying also some of its adjacent objects. Actually, we need to display at least two adjacent objects, such that one of them is the XML parent of the connector. For example, in the data graph of Figure 6, there is a connector `memberships` with the adjacent objects `Israel`, `Interpol` and `UN`. Any meaningful information pertaining to that connector must include at least its XML parent `Israel` and one more adjacent object. Observe that if we display two adjacent objects, such that none is the XML parent, then the obtained information is rather meaningless. That is, `UN` ← `memberships` → `Interpol` only indicates that some country is a member of both `Interpol` and `UN`.

Hence, if a connector is either the root or a leaf of a generated answer A, we need to *augment* A before showing it to the user. First, suppose that the root of A is a connector

c. Recall that the root must have at least two children. If one of them is the XML parent of c, we discard the answer A, because it is more meaningful when the XML parent is the root (and due to opposite idref edges, there would be an equivalent answer that satisfies that). If the answer A is not discarded, we add the XML parent of c as the new root (and c becomes its child). Note that this augmentation violates the requirement that the root must have at least two children, but it is necessary to make the answer meaningful to the user.

Now, suppose that a leaf of the answer A is a connector c. If a solid edge (i.e., of the initial data graph) enters c, then we add all the outgoing solid edges of c and their adjacent objects become leaves instead of c. If an opposite idref edge (i.e., dashed arrow) enters c, then we add the only outgoing opposite idref edge of c and its adjacent object (which is actually the XML parent of c) becomes a new leaf instead of the connector c.

Sometimes, nodes added during augmentation already appear in the answer. These cases can be handled as explained in [14].

The discussion about augmentation already alludes to an important principle. That is, if a connector appears in an answer, both its incoming and outgoing edges should be of the same kind (i.e., all of them should be either solid or dashed edges). For example, consider the path `Interpol` → `memberships` → `UN` in Figure 6. A dashed (i.e., opposite idref) edge enters `memberships` and a solid one goes out. This is a rather meaningless path—all it says is that some country is a member of both `Interpol` and `UN`. Therefore, we do not want such paths in answers. We achieve this by requiring that the traversal of the data graph (during the process of generating answers) satisfies the following rule. If a path (which is part of an answer) enters a connector through a solid (resp., dashed) edge, then it must also leave through a solid (resp., dashed) edge. Since this rule is applied during the traversal of the graph, we do not even generate answers that violate it.

5. RANKING ANSWERS

In this section, we discuss the issue of how to assign weights to the nodes and edges of a data graph and how to do the final ranking of answers. It should be noted that lower weights are better, since (as mentioned in Section 3.3) we follow the principle that a smaller answer is more meaningful (and the size of an answer is actually its weight).

Two factors determine the weights: the semantics of the data and the text. The former refers to database aspects, namely, the relative importance of entities and the strength of the relationships among them (note that entities and relationships are objects and connections, respectively, in our model). The latter factor refers to the relevance of the text to a specific query, as viewed in information retrieval (IR).

Semantic considerations determine *static* weights that are independent of a specific query. Using a given query Q and the text associated with the nodes of the data graph, we should also derive *dynamic* weights that indicate relevance to Q. The weights of nodes and edges are used in two ways. First, when generating answers for a given query Q, we do it in a manner that uses the weights. As mentioned in Sections 3.3 and 3.4, enumeration by increasing height is a good strategy, because it can be done efficiently and has a good correlation with the desired final ranking. Second, those weights can be used in the final ranking, say by increasing weight (and, hence, "smaller is better" actually means "an answer with a smaller weight is better.") However, the final ranking of answers may also be done in a way that is not strictly determined by their weights.

A common theme for determining static weights is "prestige" [5]. It is tempting to interpret this notion as meaning a high degree in the data graph. That is, an object that has a high degree in the data graph (which means connections to many other objects) would be deemed more important (and, hence, should get a lower weight). But this is hardly a universal principle. For example, which is a more important organization—one with many members or one with only a few? An organization with many members may be important because it is viewed as having a wide influence, whereas an organization with only a few members could be deemed important because it is highly exclusive. Formulas for assigning static weights are given in [12, 29, 30].

Clearly, the first step is to assign the static weights to the structural nodes of the data graph and to the edges connecting them, because they do not depend on a specific query. Now, suppose that we are given a query Q. Two issues arise: how to determine dynamic weights and how to assign them; in particular, there is no obvious way of combining static and dynamic weights.

In [29, 30], dynamic weights are computed by using language models [32] with smoothing methods [37]. In addition, the text of each node is divided into three fields: *title*, *content* and *structure*. The first field is just the title of a node, if it exists (recall that objects, but not connectors, typically have titles). The second is the whole text associated with the node (including the nested properties). The third comprises the type of the node and those of its properties (that is, the names of the corresponding elements and attributes from the source XML document). The approach of dividing the text into fields bears some similarity to [16]. It makes it possible to determine the degree of importance of a particular keyword occurrence depending on where it appears.

Recall from Section 4.6 that only keyword nodes of the given query Q are attached to the skeleton of the data graph, when generating answers to Q. In [30], only the edges of those keyword nodes get dynamic weights. Consider an edge (v, k) from a node v that has an occurrence of the keyword k of Q to the node for k. The weight of (v, k) is determined by the relevance of v to the query Q (rather than just to k). That relevance is calculated by the language-model approach. In particular, distinct language models are created for the title and content fields and then combined together. A keyword that appears in the structure field is treated differently (see details in [30]), because language models are not effective in this case.

The final ranking of answers in [30] is not according to the weights of the generated answers. Instead, for each answer, the text from all its nodes is concatenated (while keeping distinct fields separately) and a new *lm-value* is computed based on language models. That lm-value is combined with the weight of an answer to yield a score that determines the final ranking.

An important issue is that of normalizing (into one scale) the static weights, the dynamic weights and the lm-values. Recall that the first are derived from semantic considerations whereas the latter two are obtained from language models. In particular, language models yield probabilities for which

the higher is better. The first task of the normalization is to invert those probabilities, since ranking is done by increasing score (i.e., the lower is better). The second task of the normalization is to make it meaningful to combine values obtained from language model with those derived from semantic considerations. Some formulas for doing that were developed in [30]. However, more research is needed to provide theoretical foundations for the normalization process.

In [30], as explained earlier, the dynamic weight of an edge (v, k) from a node v to a keyword k is determined only by the relevance of v to the given query Q. However, v is (at best) just one node of a relevant answer. In [29], a neighborhood of v, called a *virtual leaf*, is considered rather than merely v itself. In addition, the estimation of probabilities for a virtual leaf takes into account the distance of each term occurrence t from v, in a manner that resembles [27]. Note that the distance is the sum of static weights from v to the node containing t. In [29], the idea of virtual leaves is also extended to *virtual roots*, that is, nodes that might be roots of relevant answers.

In [7], they developed a framework for evaluating database keyword-search strategies and tested nine systems. Their framework consists of three datasets and fifty queries for each one of them. When compared with those nine systems, [30] achieved higher MAP (mean average precision) on each of the three datasets, and [29] was even better. The work of [6] reported the best performance that has been achieved so far on the evaluation framework of [7], but the authors of [6] have not yet published sufficient details to reproduce their results.

In [30], they also showed how to combine the system described in Section 4 with Lucene,[4] which is an open-source software for information retrieval. Thus, we can efficiently generate answers according to the ranking described in this section. In particular, [30] showed that there is no need to have edges entering keyword k from all the structural nodes that contain k. Instead, it is sufficient to do that just for the top-N nodes that contain k, where the ranking of those nodes is according to their relevance to the given query Q. That is, even when using small values of N, the decrease in the MAP is relatively minor, whereas the efficiency is substantially increased (because a smaller portion of the data graph has to be traversed). A similar result holds for the approach of [29].

Two useful features were demonstrated in [2]. The first is to specify that a keyword of the query must appear in the structure field. This is done by adding the suffix ? to the keyword. For example, the query "`France, capital?`" would return the answer `France` $\rightarrow$ `capital` $\rightarrow$ `Pairs`. Hence, this is a form of question answering. The second feature is the ability to group several keywords together by enclosing them inside curly brackets. The meaning is that those keywords should appear together in the same node of an answer. In [30], it is shown that these two features can enhance the performance (i.e., increase the MAP).

6. TWO DIMENSIONAL SEARCH

As enunciated in [28], keyword search over data graph is inherently a two-dimensional process. A specific answer is characterized by its content as well as by its *pattern*. The latter refers to the tree structure of an answer, that is, the pattern that we get when considering only the type of each node (and ignoring the node's id and the rest of its content).

For example, consider the query "`Lebanon, Israel`." Many answers to this query have a similar flavor: each one states that both countries are members of some organization. All those answers have the pattern `country` $\leftarrow$ `memberships` $\leftarrow$ `organization` $\rightarrow$ `memberships` $\rightarrow$ `country`. It is pointless to show to the user many answers of this pattern if she is interested in something else. Some rudimentary features for dealing with this issue were demonstrated in [2]. For example, the user can specify that she is not interested in answers that have nodes of type `organization`.

A comprehensive approach would allow a user to seamlessly navigate across the two dimensions. Toward this end, we may want to allow a user (who has already seen some answers) to choose any option among the following three. First, getting the next best answer in the overall ranking order. Second, getting the next best answer of a particular pattern, which would be chosen from those of answers that have already been presented, and the user may impose some additional constraints (e.g., the pattern should include a specific node). Third, getting the best next answer of a patten that has not yet appeared among the presented answers; again, some constraints might be imposed (e.g., the pattern should not include a node of type `organization`).

Efficient algorithm for supporting the above options should be developed. A possible approach might be based on a synergy of algorithms that directly search for answers on the data graph and those that first generate patterns and then compute them. The problem with the latter is that many of the generated patterns may yield empty results. Recent work [24, 25] developed algorithms that alleviate this problem by taking into account *neighborhood* constraints that the schema satisfies.

There is also a need for some additional methods and features to support two-dimensional search; for example, techniques for summarization and aggregation of answers. More details are given in [28].

Another approach is to facilitate two-dimensional search by automatically diversifying the presented answers. In [12], they introduced the idea of *penalizing* answers due to similarity to those that have already been presented to the user. Thus, the ranking is dynamic in the sense that after printing each answer, the remaining ones are re-ranked by adding a factor that takes into account similarity to the answers that the user has already seen. The notion of similarity is based on common parts either in the patterns of two answers or in their particular trees (i.e., by taking into account the id's of nodes, rather than just their types).

7. CONCLUSIONS

Although work on keyword search over data graphs has already been done for (at least) a decade, there are still quite a few theoretical and practical issues that have to be tackled. Progress depends on a synergy of practical and theoretical research. It is essential to invest resources in creating real-world data graphs and developing evaluation frameworks.

8. ACKNOWLEDGMENTS

The author is deeply grateful to Konstantin Golenberg, Benny Kimelfeld and Yosi Mass without whom the work described in this paper would not have been done.

[4] `http://lucene.apache.org/`

9. REFERENCES

[1] H. Achiezra. Understanding Complex Answers of Queries on Graphs. Master's thesis, Hebrew University, Department of Computer Science, 2009.

[2] H. Achiezra, K. Golenberg, B. Kimelfeld, and Y. Sagiv. Exploratory keyword search on data graphs. In *SIGMOD Conference*, pages 1163–1166, 2010.

[3] M. S. Ackerman. The politics of design: Next generation computational environments. In K. Kraemer and M. Elliott, editors, *Computerization Movements and Technology Diffusion: From Mainframes to Ubiquitous Computing*, ASIS&T Monograph Series, 2006.

[4] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: enabling keyword search over relational databases. In *SIGMOD Conference*, page 627, 2002.

[5] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[6] V. Bicer, T. Tran, and R. Nedkov. Ranking support for keyword search on structured data using relevance models. In *CIKM*, pages 1669–1678, 2011.

[7] J. Coffman and A. C. Weaver. A framework for evaluating database keyword search strategies. In *CIKM*, pages 729–738, 2010.

[8] S. Cohen, I. Fadida, Y. Kanza, B. Kimelfeld, and Y. Sagiv. Full disjunctions: Polynomial-delay iterators in action. In *VLDB*, pages 739–750, 2006.

[9] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. Xsearch: A semantic search engine for xml. In *VLDB*, pages 45–56, 2003.

[10] R. Fagin, A. O. Mendelzon, and J. D. Ullman. A simplified universal relation assumption and its properties. *ACM Trans. Database Syst.*, 7(3):343–360, 1982.

[11] C. A. Galindo-Legaria. Outerjoins as disjunctions. In *SIGMOD Conference*, pages 348–358, 1994.

[12] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD Conference*, pages 927–940, 2008.

[13] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Optimizing and parallelizing ranked enumeration. *PVLDB*, 4(11):1028–1039, 2011.

[14] K. Golenberg and Y. Sagiv. The architecture of a system for keyword search over data graphs. Manuscript in preparation.

[15] K. Golenberg and Y. Sagiv. A practically efficient algorithm for generating all answers to search queries over data graphs. Manuscript in preparation.

[16] D. Hiemstra. Statistical language models for intelligent XML retrieval. In *Intelligent Search on XML Data*, pages 107–118, 2003.

[17] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[18] D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

[19] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[20] B. Kimelfeld and Y. Sagiv. Efficient engines for keyword proximity search. In *WebDB*, pages 67–72, 2005.

[21] B. Kimelfeld and Y. Sagiv. Efficiently enumerating results of keyword search. In *DBPL*, pages 58–73, 2005.

[22] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182, 2006.

[23] B. Kimelfeld and Y. Sagiv. Efficiently enumerating results of keyword search over data graphs. *Inf. Syst.*, 33(4-5):335–359, 2008.

[24] B. Kimelfeld and Y. Sagiv. Finding a minimal tree pattern under neighborhood constraints. In *PODS*, pages 235–246, 2011.

[25] B. Kimelfeld and Y. Sagiv. Extracting minimum-weight tree patterns from a schema with neighborhood constraints. In *ICDT*, 2013.

[26] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7), 1972.

[27] Y. Lv and C. Zhai. Positional language models for information retrieval. In *SIGIR*, pages 299–306, 2009.

[28] Y. Mass, M. Ramanath, Y. Sagiv, and G. Weikum. IQ: The case for iterative querying for knowledge. In *CIDR*, pages 38–44, 2011.

[29] Y. Mass and Y. Sagiv. Language models for virtual documents in data graphs. Unpublished manuscript.

[30] Y. Mass and Y. Sagiv. Language models for keyword search over data graphs. In *WSDM*, pages 363–372, 2012.

[31] K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3), 1968.

[32] J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In *SIGIR*, pages 275–281, 1998.

[33] A. Rajaraman and J. D. Ullman. Integrating information by outerjoins and full disjunctions. In *PODS*, pages 238–248, 1996.

[34] Y. Sagiv. Can we use the universal instance assumption without using nulls? In *SIGMOD Conference*, pages 108–120, 1981.

[35] Y. Sagiv. A characterization of globally consistent databases and their correct access paths. *ACM Trans. Database Syst.*, 8(2):266–286, 1983.

[36] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data. In *ICDE*, pages 405–416, 2009.

[37] C. Zhai and J. D. Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Trans. Inf. Syst.*, 22(2):179–214, 2004.