Keyword Proximity Search in Complex Data Graphs*

Konstantin Golenberg, Benny Kimelfeld and Yehoshua Sagiv
The Selim and Rachel Benin School of Engineering and Computer Science
The Hebrew University of Jerusalem, Edmond J. Safra Campus
Jerusalem 91904, Israel
{konstg01,bennyk,sagiv}@cs.huji.ac.il

ABSTRACT

In keyword search over data graphs, an answer is a non-redundant subtree that includes the given keywords. An algorithm for enumerating answers is presented within an architecture that has two main components: an *engine* that generates a set of candidate answers and a *ranker* that evaluates their score. To be effective, the engine must have three fundamental properties. It should not miss relevant answers, has to be efficient and must generate the answers in an order that is highly correlated with the desired ranking. It is shown that none of the existing systems has implemented an engine that has all of these properties. In contrast, this paper presents an engine that generates *all* the answers with provable guarantees. Experiments show that the engine performs well in practice. It is also shown how to adapt this engine to queries under the *OR* semantics. In addition, this paper presents a novel approach for implementing rankers destined for eliminating redundancy. Essentially, an answer is ranked according to its individual properties (relevancy) and its intersection with the answers that have already been presented to the user. Within this approach, experiments with specific rankers are described.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms

Keywords

Keyword proximity search, information retrieval on graphs, subtree enumeration by height, approximate top-k answers, redundancy elimination

*This research was supported by The Israel Science Foundation (Grant 893/05) and by a grant from Google.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

1. INTRODUCTION

In IR (information retrieval), the goal is to find documents that are relevant to the given keywords. Even when applying IR to textual documents in XML format, the goal is similar, namely, to search for relevant fragments (e.g., sections, paragraphs, etc.) rather than whole documents [6, 7, 22]. In other words, the XML elements of a textual document do not constitute a semistructured database[1] [20], but rather show how to break the document into meaningful fragments of text that may be relevant to the given keywords.

In keyword search over databases, the focus is not merely on finding relevant fragments of text, but also on discovering the semantic relationships that hold between the keywords. For example, suppose that we are searching for the keywords `Jones` and `Compilers` in a university database. Just looking for occurrences of these keywords wherever they might appear in the database is quite meaningless. Instead, we would like to know how `Jones` and `Compilers` are related to each other. For example, we might discover that `Jones` is the teacher of the course named `Compilers`; another answer might be that `Jones` is the author of a book that has the keyword `Compilers` in its title. Note that these are two distinct relationships between the given keywords.

Keyword search in databases is performed over a graph in which nodes are associated with keywords and edges describe semantic relationships. For example, if the database is as an XML document, then it can be naturally represented as a graph (rather than a tree, because we want to include semantic relationships that are described by ID references). If the database is relational, then the nodes correspond to the tuples and the edges connect pairs of tuples that can be joined by using a foreign key. A search query is formulated merely as a list of keywords and it does not convey any structural information. Answers to the query are subtrees (of the data graph) that contain occurrences of the given keywords. The tree structure of an answer describes the relationship between the keywords.

A system that supports keyword proximity search over complex data graphs faces two main challenges that are evident, for example, in the Mondial database[2] which is highly cyclic. The first is the standard challenge of IR, namely, to find the most relevant answers. The second, which arises in our type of search and especially in complex data graphs,

[1]Only a few elements of textual documents, such as author name, can be viewed as objects of a semistructured database.
[2]The Mondial database contains geographic data and is available at `http://www.dbis.informatik.uni-goettingen.de/Mondial/`.

is to deal with the *repeated-information* problem. That is, a system should not overwhelm the user with repeated (redundant) information. While the literature contains a lot of research on the first problem, it ignores the second. The main reason is that past systems have dealt with rather simple data graphs, e.g., DBLP and IMDB (the Internet Movie Database). Note that related problems are those of *novelty* and *first-story* detection in a stream of textual documents [2,21,26]. However, these problems face an inherently different notion of redundancy, namely, one that is based on similarity and containment of textual data in separate documents. Next, we consider the first challenge.

The major obstacle in dealing with the first challenge is the huge number of answers, i.e., subtrees that contain the given keywords. Typically, only a small fraction of those answers are deemed relevant by users. The first step in dealing with this problem is to require an answer to be *reduced* (i.e., non-redundant), which means that every proper subtree (of the answer) has fewer occurrences of the keywords (compared to the answer itself). But even under this definition, the number of answers can be very large. We have discovered, for example, that when searching the DBLP bibliography with names of two researchers who wrote many papers, there are literally tens of thousands of answers that connect the two authors through different chains of citations (e.g., the first author wrote a paper that cites another paper that cites a paper of the second author).

Ranking is used in order to cope with the large number of answers. Typically, fewer semantic connections (e.g., fewer edges) indicate a stronger affinity between the keywords. Thus, we actually want to perform *keyword proximity search,* that is, to find occurrences (of the keywords) that are close to each other in the data graph. Consequently, the size (or weight) of an answer plays a prominent role in the ranking function. Ideally, the system should enumerate the answers in ranked order, but it is unlikely that it can be done efficiently because just finding the smallest answer (known as the *Steiner-tree* problem) is intractable. Furthermore, it seems impossible to translate practical ranking measures (e.g., those used in [3, 8, 12, 19]) into weights on the nodes and edges of data graphs.

To circumvent the above problem, systems for keyword proximity search [3, 8, 12, 19] process queries by means of two components: an *engine* that generates *candidate* answers and a *ranker* that determines a score for each of the generated answers. At any given time, the process may stop and return the *top-k* answers among those that have already been generated. Furthermore, the search can be resumed in order to produce more answers if the user wants to see them. In this paper, we present a system that follows this architecture.

The effectiveness of the above approach is based on the ability of the engine to efficiently generate the answers in an order that is correlated with the ranking measures. In this respect, especially when considering large and complicated (e.g., highly cyclic) graphs, none of the engines used in existing systems is acceptable. For example, the engines of [1,8–10] evaluate all possible queries (up to some limit on the size) that may connect the given keywords. A large number of these queries may have an empty result, which causes the whole process to be inefficient. Furthermore, these engines cannot take weights into account, which leads to a rather limited correlation with the ranking measures. As another example, the engines of [3,4,12,23] may miss highly relevant answers (while producing less relevant answers instead), as we show in Section 3.2. Therefore, a robust system cannot rely on any of the above engines.

More formally, an effective engine for generating answers must satisfy three conditions. First, it must be able to generate all the answers. Second, the answers must be enumerated in an order that is correlated with the final ranking. Third, the engine must be efficient. The suitable criterion of efficiency is a polynomial bound on the delay between answers [11]. Note that none of the implemented engines enumerates with polynomial delay. Algorithms that satisfy all of the three conditions are presented in [15]. However, these algorithms do not lend themselves easily to a practical implementation, since they use complex subroutines for approximating Steiner trees.

In this paper, we describe a search engine that is based on an incremental algorithm for enumerating subtrees in a 2-*approximate order* [5, 15] by increasing height (i.e., the approximation ratio is 2). This algorithm runs with polynomial delay and can find all the top-k answers. Our experiments show that this engine is practical and, furthermore, there is a good correlation between the generated order of answers and the desired order according to the inverse of the weight.

After generating the candidate answers, the ranker sorts them according to a more sophisticated ranking function, e.g., one that is based on the total weight and additional IR measures. However, this ignores the repeated-information problem, i.e., the second challenge that we now illustrate. Consider again the example of the university database and suppose that the query contains the three keywords `Jones`, `compilers` and `CS`. There are many relevant connections between pairs of the keywords, e.g., `Jones` teaches `compilers`, `Jones` wrote a book about `compilers`, `Jones` is a staff member of `CS`, `compilers` is a course in `CS`, `Jones` is the head of `CS`, etc. But then, many combinations of these connections are also relevant answers. Hence, a system that is not aware of this problem will overwhelm the user with all the possible combinations, without being able to rank them correctly.

Our ranker employs a novel technique for eliminating repeated information. The main idea is that the ranking function (which is initially based on total weight) dynamically *adapts* itself, during the ranking procedure, to address redundancy. More particularly, after selecting the top-i answers, the ranking function *penalizes* the other answers if they contain information that is *similar* to what has already been given. We consider several policies for determining the penalty.

Given the absence of (and difficulty in producing) standard benchmarks for this type of search, we devised a novel methodology for evaluating the effectiveness of methods that deal with the repeated-information problem. Essentially, we measure the reduction in the redundancy and compare it with the loss in the score (where the latter is taken in the standard manner, i.e., ignoring information repetition). Our experiments and evaluation methodology indicate that our approach is effective in the sense that repeated information can be eliminated with a relatively low drop in the score of the top answers. Furthermore, one of the proposed alternatives to determining penalties for repeated information is better than the others. Note that methodologies for detecting redundancy have been proposed in the context of XML

Countries

	code	name	capital	government
t_1	B	Belgium	BRU	Monarchy
t_2	NOR	Norway	OSL	Monarchy

Organizations

	name	headq	#members
t_3	EU	BRU	25
t_4	ESA	PAR	17

Cities

	code	name	country	population
t_5	ANT	Antwerp	B	455,148
t_6	BRU	Brussels	B	141,312
t_7	OSL	Oslo	NOR	533,050

Members

	country	organization
t_8	B	ESA
t_9	B	EU
t_{10}	NOR	ESA

Figure 1: A small portion of the Mondial relational database

retrieval [13]. However, it is not clear how their methodologies can be applied in the context of keyword proximity search. For instance, in XML retrieval, redundancy means that both an element and one of its sub-elements are retrieved. Obviously, the notion of redundancy in keyword proximity search is much more complicated.

The rest of the paper is organized as follows. In the next section, we describe the formal setting and the basic architecture of our search engine. In Section 3, we give the algorithm for enumerating answers with polynomial delay in a 2-approximate order by increasing height. For completeness of the presentation, the underlying basic procedure (presented in [15] and based on [17,24,25]) of our algorithm is described in the appendix. Section 4 describes the repeated-information problem and our ranking techniques. The implementation and experimentation are described in Section 5. In Section 6, we describe how to adapt our algorithm to enumerating answers that contain some of the given keywords (rather than all of them), i.e., searching under the OR semantics [19]. We conclude in Section 7.

2. DATA MODEL

2.1 Data Graphs

In keyword proximity search over databases, the goal is to discover occurrences of the keywords as well as semantic relationships between them. Toward this end, a database is modeled as a graph.[3] Nodes correspond to either XML elements or relational tuples (depending on the data model), and edges represent semantic relationships. To simplify the presentation, there is also a node for each keyword (instead of assuming that keywords are associated with the nodes that represent elements or tuples). Formally, a *data graph* is a directed graph that has two types of nodes: *structural* nodes and *keyword* nodes (or just *keywords* for short).

As an example, Figure 2 shows a data graph (where keyword nodes are shown as words). Both relational and XML data can be represented as graphs. When representing an XML document as a graph, the XML elements correspond to structural nodes. Edges represent either element nesting or ID references (which must be taken into account when answering queries, since they represent semantic relationships). The text is represented by a keyword node for each word w. For all structural nodes v that correspond to XML elements containing w, there is an edge from v to w. In the case of relational data, the structural nodes represent tuples. The edges are from tuples to the keywords contained in those tuples and also from each tuple that includes a foreign key to the tuple that has that key value. 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 positive weights (which can be determined according to a variety of IR techniques). The *length* of a path in the graph is the sum of the weights along the path. The *height* of a 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.

2.2 Queries and Answers

In our model, a query is simply a set of keywords and an answer is a non-redundant subtree that contains all the given keywords. Formally, we consider a data graph G. A *query* is a set Q that consists of at least two keywords from G. An *answer* to Q is called a *Q-subtree* and is defined as follows. A Q-subtree is any subtree T of G that is *reduced* with respect to (abbr. w.r.t.) Q, namely, T contains all the keywords of Q but has no proper subtree that also contains all of them. Note that subtrees are directed, i.e, a subtree T of G has a node r, called the *root*, such that every other node of T is reachable from r through a unique directed path. Given a data graph G and a query Q, we use $\mathcal{A}(G, Q)$ to denote the set of all answers of Q. The following is an alternative characterization of Q-subtrees that is important for understanding the ideas discussed in this paper.

Observation 2.1. *A subtree T of G is a Q-subtree if and only if the set of leaves of T is exactly Q and the root of T has at least two children.*

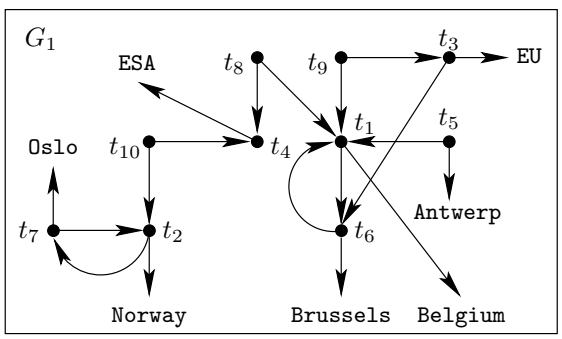

Figure 2: Data graph G_1

[3]Representing data as a tree might be sufficient for searching a corpus of textual documents in XML format, but not when performing keyword search over databases.

EXAMPLE 2.2. *The data graph G_1 of Fig. 2 is a tiny portion of the Mondial relational database that is depicted in Figure 1. (Translating a relational database into a data graph is done similarly to [1, 3, 9]). Let the query Q consist of the keywords* `Belgium`, `EU` *and* `Brussels`, *i.e.,* $Q = \{\texttt{Belgium}, \texttt{Brussels}, \texttt{EU}\}$. *The only two answers are the Q-subtrees A_1 and A_2 shown in Fig. 3.*

The answer A_1 states that Belgium is a member of the European Union and its capital is Brussels. A_2 is a similar yet different answer—the European Union is located in Brussels, which is the capital of Belgium. Clearly, both answers are relevant to the given keywords.

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. So, the answers are A'_1 and A'_2 of Figure 4.*

2.3 Answer Generators and Rankers

We study an architecture of search engines that comprises two major components. A database is transformed into a data graph G in an offline manner. Then, given a query Q, the following two components are invoked.

- **Answer Generator.** The answer generator produces N_i Q-subtrees that are *candidate answers.* In principle, this is an efficient algorithm for generating the top Q-subtrees, under some simple scoring function (which, in our case, is based on the height of the subtrees).
- **Ranker.** This component operates over the candidate answers. It generates a set of N_f ($< N_i$) final answers which are ranked according to a more complex ranking function.

In the next section, we describe the answer generator. Later, in Section 4, we describe the ranker.

3. GENERATING ANSWERS

3.1 Ranking, Enumerations and Efficiency

Given a data graph G and a query Q, the goal is to *enumerate* the set of answers $\mathcal{A}(G, Q)$, i.e., to print each answer exactly once. Ideally, answers should be enumerated in ranked order. However, it is not practical to do that, because even if the IR measures (that determine the rank) could be translated into weights on nodes and edges, one would still have to solve the intractable Steiner-tree problem. As explained earlier, the conventional approach is to enumerate according to an alternative, more tractable ranking order that is highly correlated with the final rank. The top answers are chosen after calculating the final rank for each answer that was actually generated. Formally, the alternative rank is determined by applying a function h to each answer A. In other words, $h(A)$ denotes the score (or grade) that is assigned to A. In IR, a higher score is usually considered as better than a lower one. But in keyword proximity search, "smaller is better," i.e., a smaller subtree indicates a stronger relationship between the keywords. Therefore, the top-ranked answer, according to h, is the one that has the minimum value of h and the ranking order is by increasing values of h.

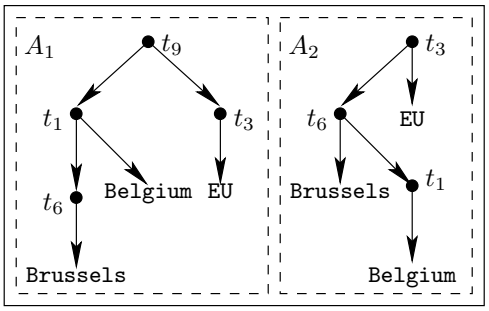

Figure 3: Answers A_1 and A_2

Even for a simple definition of h, efficient enumeration might not be possible. Therefore, we also need to consider the formal notion of enumeration in an approximate order, which is defined as follows [15]. Enumerating all the answers in a *θ-approximate order* means that if one answer precedes another, then the first is worse than the second by at most a factor of θ. More formally, the sequence of all answers $A_1, \ldots, A_n$ is in a θ-approximate order if $\theta \cdot h(A_j) \geq h(A_i)$ for all $1 \leq i \leq j \leq n$.

Note that there are two related notions. First, we say that an answer A is a *θ-approximation* of the *optimal* (i.e., top-ranked) answer if for all answers $A' \in \mathcal{A}(G, Q)$, it holds that $\theta \cdot h(A') \geq h(A)$. By definition, if $\mathcal{A}(G, Q) = \emptyset$, then $\perp$ is a θ-approximation of the optimal answer. Second, Fagin et al. [5] introduced the notion of a θ-approximation of the top-k answers as any set *AppTop* consisting of $\min(k, |\mathcal{A}(G, Q)|)$ answers, such that for all $A \in$ *AppTop* and $A' \in \mathcal{A}(G, Q) \setminus$ *AppTop*, it holds that $\theta \cdot h(A') \geq h(A)$.

The efficiency of an enumeration algorithm is measured in terms of the delay between printing each pair of consecutive answers. We say that an algorithm E enumerates with *polynomial delay* [11] if 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 $p(n)$. Note that 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 all, then the algorithm must stop after a polynomial number of steps.

The following is an important observation. An algorithm that enumerates in a θ-approximate order with polynomial delay can also find efficiently a θ-approximation of the top-k answers, by stopping the execution after printing the kth answer (and the running time is linear in k and polynomial in the input size). In particular, it can find efficiently a θ-approximation of the optimal answer.

In this paper, we explore the approach of enumerating answers by increasing height. So, $h(A)$ is the height of A, that is, the weight of the longest path from the root to a leaf. This approach has already been used in BANKS [3,12]. But, as discussed below, their algorithms miss answers and have exponential delays in the worst case.

3.2 Existing Approaches

In recent years, several systems [1,3,4,8–10,12,19,23] have implemented some form of keyword proximity search. These systems (and also the approach of searching for "information units" on the Web [18]) use an engine for generating answers. Some systems [1,4,9,10,23] present the answers in the order

of generating them, whereas other systems [3,8,12,19] apply functions (e.g., based on IR techniques or link analysis) to determine the final ranking order.

The algorithms used for implementing the engines of [1, 8–10, 19] are based on a *relational* approach. That is, they store the data in a relational database and use the schema to extract all join expressions that can potentially generate answers. Since these expressions are produced (and evaluated) in the order of increasing size, so are the answers. This approach, however, has two major drawbacks. The first is inefficiency. A join expression does not necessarily produce answers for the specific data at hand. Therefore, this approach may evaluate a huge number (i.e., exponential in the size of the data) of expressions without generating even one answer. So, it cannot enumerate with polynomial delay. The inefficiency is intensified when queries have relatively many keywords or the schema is complicated (e.g., highly cyclic as in the case of the Mondial database). The second drawback is that this approach cannot incorporate weights on the nodes and edges of the data graph. Conceivably, one may define weights on the schema so that the system can take into account the weighted size of each join expression. But it is impossible to differentiate between answers that are generated by the same expression or to use ranking measures that depend on the particular data. Hence, this approach has a very limited ability to correlate between the actual order of generating answers and the desired ranking.

The engines of [3, 4, 12, 15, 23] are based on a *graph* approach, that is, they apply algorithms that operate directly on the data graph and generate subtrees. Overall, this approach leads to more efficient engines than those mentioned earlier. However, it is a subtle task to find all the answers and avoid the generation of redundant subtrees. In fact, none of the algorithms of [3, 4, 12, 23] manages to do that. The work of [4, 23] uses a different algorithm from those employed by [3, 12], but in both cases the underlying approach is similar. First, they take an algorithm that finds the top-ranked answer, which is either a minimal-height subtree [3, 12] or a minimal-weight subtree [4, 23]. Then they repeatedly apply that algorithm to generate more answers. In each application, they generate a subtree that is required to include a new node or, in the case of [3, 12], new occurrences of the keywords. There are two problems with this approach. First, as we show below, there could be more than one answer that contains the same occurrences of the keywords and the same new node. But this approach can find only one of those answers. Second, this approach inevitably generates redundant (i.e., non-reduced) subtrees; for example, a subtree that consists of a previously generated answer and one additional edge that connects that answer to a node that does not have any keyword from the query. In [3, 12], they test whether a generated subtree is redundant and if so, they discard it. In [4, 23], the issue of redundancy is not mentioned.

In summary, the engines of [3, 4, 12, 23] have two major flaws. First, they may miss very relevant (i.e., small or short) answers while generating less relevant ones. Second, in the algorithms of [3, 12], redundant answers can cause a long delay (i.e., proportional to the number of answers that have already been generated). In [4, 23], the delay is exponential in the size of the query (but polynomial in the size of the data graph, because they generate at most one subtree for each node of the graph and do not eliminate redundant subtrees).

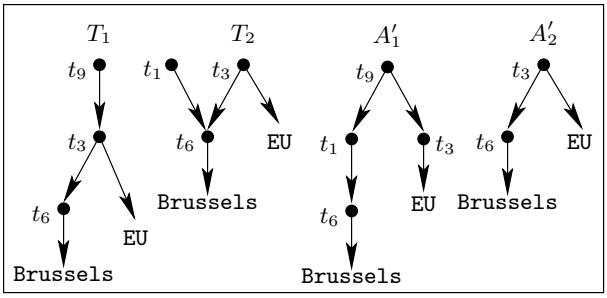

Figure 4: Two answers and a subtree

Thus, the algorithms of [3, 4, 12, 23] do not generate all the answers and do not run with polynomial delay. In comparison, the algorithms of [15] enumerate all the answers with polynomial delay and in the order of increasing weight (or an approximation thereof). But these algorithms are based on repeated computations of Steiner trees (or complex subroutines for approximating Steiner trees) and, therefore, do not lend themselves easily to a practical implementation.

We now use examples to illustrate the above problems and show why enumerating subtrees by height or weight is subtle. Consider Figure 4 and recall from Example 2.2 that A'_1 and A'_2 are the answers to the query {`Brussels`, `EU`}, given the data graph G_1 of Figure 2. For simplicity, we assume that edges and nodes have weights of 1 and 0, respectively.

The algorithms of [3, 12] attempt to find the answers A'_1 and A'_2, but may fail to do so. These algorithms apply backward Dijkstra-style (i.e., shortest-path) iterators that start from the keywords and meet at their ancestors. So, given the keywords `Brussels` and `EU`, these iterators meet at the nodes t_3 and t_9. For each ancestor v, the algorithms of [3, 12] produce only *one* (minimal-height) answer by combining shortest paths that connect the keywords to v. So, these algorithms might compute the subtree T_1 of Figure 4 for t_9. But the root of T_1 has a single child and so it is deleted in order to obtain a smaller (and, hence, more meaningful) subtree that contains all the given keywords. The end result is that only one answer, A'_2, is found and the second answer, A'_1, is completely missed.

In [4,23], answers are undirected subtrees. For each node v of the graph, the algorithm of [4,23] generates (at most) one answer by finding a minimal-weight subtree that includes v and the keywords. So, the first answer is A'_2. Then the algorithm produces two subtrees that have a weight of 4, namely, T_1 and T_2 (which should be viewed in Figure 4 as undirected). Neither T_1 nor T_2 is reduced, but the algorithm of [4, 23] does not check whether the generated subtrees are redundant. The answer A'_1 is not produced because for all nodes v of A'_1, there are smaller subtrees that contain v and the keywords (namely, A'_2, T_1 and T_2). The algorithm of [4, 23] continues to generate larger subtrees until it prints all the top-k. But all those subtrees are redundant except for the first. Moreover, this algorithm misses the answer A'_1.

As said above, another problem with the approach of [3, 12] is that the delay may be very large, that is, exponential in the size of the input. Consider, for example, the graph G_2 of Figure 5. (In this particular example, we follow the formalism of [3, 4, 12, 23] and represent different occurrences of the same keyword by distinct nodes.) Suppose that the query consists of the keywords $k_1, \ldots, k_n$. Each keyword k_i, where $1 \leq i \leq n-1$, has two occurrences in the graph

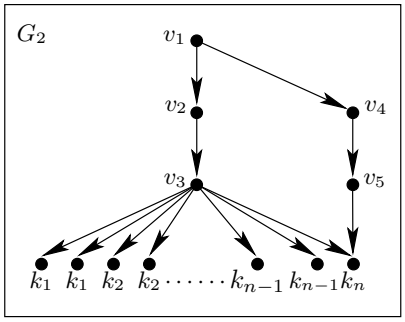

Figure 5: A case with an exponential delay

G_2 while k_n has a single occurrence. For node v_2 as the root, the algorithms of [3,12] generate 2^{n-1} minimal-height subtrees—one for each combination of choosing nodes for the keywords. But all of those subtrees are discarded, because their root v_2 has a single child. When v_1 is reached, there are two choices for the shortest path from v_1 to k_n. One path goes through v_2 and the other passes through v_4. A wrong choice (namely, the first path) will result once again in 2^{n-1} redundant subtrees. Thus, the algorithms may "get stuck" generating an exponential number of redundant subtrees and, in the process, miss an exponential number of equal-height subtrees that are not redundant.

On the graph of Figure 5, the algorithm of [4,23] generates at most $2n$ subtrees that have v_3 as the root and keyword occurrences in the leaves. Then it continues to generate bigger subtrees while missing smaller ones (and many of the generated subtrees are redundant).

3.3 Our Approach

In a sense, our algorithm combines the approaches of [3,12] and [15]. As in [3, 12], we apply shortest-path iterators to find the first answer, namely, a minimal-height subtree that contains the keywords. As in [15], we use the method of Lawler [17] to generate all the remaining answers without redundancies. This means that we have to solve the minimal-height-subtree problem subject to constraints. This is not an easy problem. In fact, we have only been able to find a 2-approximation algorithm, which is considerably more intricate than just using shortest-path iterators.

The algorithms of [14–16] and the one presented here are the only ones that can enumerate all the answers with polynomial delay. However, the algorithms of [14,16] do not have a guarantee on the order (although heuristics are proposed) and the one of this paper is simpler and much faster than that of [15]. Moreover, it lends itself easily to a practical implementation. Compared to other work in this area, our algorithm is the only one that is provably efficient and correct, as well as the only one that has been tested on highly cyclic graphs and on queries with many keywords.

In this section, we consider a data graph G and a query Q. We give an algorithm for enumerating all the Q-subtrees of G in a 2-approximate order (by increasing height) with polynomial delay. Our approach is based on an algorithm of [15] that reduces the above enumeration problem to an approximation problem under constraints (by adapting a procedure of Lawler [17]). The detailed algorithm, called GenerateAnswers, is described in the appendix for completeness of exposition. It uses two types of constraints: *inclusion constraints* and *exclusion constraints.* An instance of each type is just a set of edges. A Q-subtree satisfies a set I of inclusion constraints and a set E of exclusion constraints if it includes all the edges of I and none of E. The following is an important observation [15]. GenerateAnswers can be executed so that it only generates sets of inclusion constraints that have a special form that is defined next.

A subtree P of a graph G is a *partial answer* (abbr. PA) w.r.t. a query Q if all the leaves of P are keywords of Q. Note, however, that P is not necessarily a reduced subtree w.r.t. Q for two reasons: it may include only some of the keywords of Q and its root may have a single child. A set of inclusion constraints I is *proper* w.r.t. a query Q if the edges of I form either one PA or two node-disjoint PAs.

GenerateAnswers is a general-purpose algorithm that enumerates with polynomial delay if we can design an efficient (i.e., polynomial-time) algorithm that implements the subroutine QSubtree(G, Q, I, E). Doing so depends on the specific problem at hand and is not easy. Algorithms for QSubtree(G, Q, I, E) are given in [15] for enumerating Q-subtrees by increasing weight. In this paper, the goal is to enumerate by increasing height (rather than weight) and our algorithm for QSubtree(G, Q, I, E) is completely different from those of [15]. In particular, our algorithm is substantially more efficient and easier to implement.

In the next two sections, we give a polynomial-time algorithm for the subroutine QSubtree(G, Q, I, E). This algorithm finds a 2-approximation of a minimal-height Q-subtree of G that satisfies a proper set I of inclusion constraints and a set E of exclusion constraints. In other words, the algorithm computes a Q-subtree A of G that satisfies both I and E, such that the height of A is at most twice the height of any other Q-subtree of G that satisfies both I and E.

As shown in [15], it follows that GenerateAnswers enumerates all the Q-subtrees of G with polynomial delay and in a 2-approximate order by increasing height.

3.4 Finding a Minimal-Height Q-Subtree

We first assume that there are no constraints and describe the algorithm MinHeightSubtree(G, V, F) that returns a minimal-height subtree with all the nodes of V and a root that is not in F. Formally, the input is a graph G and two sets of nodes V and F. The output is a subtree T that has a minimal-height among all subtrees $\hat{T}$ of G that satisfy the following three conditions. First, $\hat{T}$ includes all the nodes of V. Second, all the leaves of $\hat{T}$ are in V. Third, the root of $\hat{T}$ is not in F. If there is no such subtree, then $\perp$ is returned. We call the output T a *minimal-height subtree of G w.r.t. V and F*. We need to use this algorithm even if V contains structural nodes, i.e., nodes that are not keywords. But we do that only if V has at least two nodes.

We use the same idea as in BANKS [3]. The algorithm works as follows. We start by reversing the directions of the edges of G. Then we assign to each node $v \in V$ an iterator that implements Dijkstra's single-source shortest-path algorithm. Recall that Dijkstra's algorithm uses a priority queue that initially contains just the source node v. Each node in the queue is marked with its shortest distance from v via paths that use only nodes that have already been removed from the queue. In each iteration, the top node of the queue (i.e., the one with the shortest distance from v) is removed and its neighbors are marked with new values and added to the queue (if they are not already there). So, each node $v \in V$ has a shortest-path iterator.

We perform a stepwise execution of these iterators as follows. In each step, we choose the iterator that has the smallest value at the top of its queue and do one more iteration. The following can be proven. The first node r that is removed from the queues of all the iterators is the root of a minimal-height subtree that includes all the nodes of V. When $F \neq \emptyset$, we continue until the first node r, such that $r \notin F$, is removed from the queues of all the iterators. If r does not exist, then there is no subtree T of G, such that T includes all the nodes of V and has a root that is not in F.

After finding the root r, we can construct a minimal-height subtree T of G that includes all the nodes of V as follows. We arbitrarily choose a node $u \in V$ and let T consist initially of all the edges of a shortest path from u to r (note that if r itself is in V, then we can initialize T to r). We then iterate through the remaining nodes $v \in V$ and for each v, we traverse the shortest path from v to r and add the edges along the way to T, until we get to a node that is already in T. Note that all the leaves of the resulting subtree T are in V. However, the root r of T may have only one child for two reasons. The first one is due to the restriction that r cannot be in some given set of nodes. The second reason is that the nodes of V are not necessarily keywords and so, in T, some nodes of V could be descendants of other nodes of V. If, on the other hand, $F = \emptyset$ and V is equal to a set of keywords Q, then the above construction yields a minimal-height Q-subtree, if it exists.

3.5 Approximating a Constrained Q-Subtree

We assume that the set of exclusion constraints has already been handled by removing all its edges from the data graph. Thus, in this section, we consider a data graph G, a query Q and a proper set of inclusion constraints I. We describe how to find a Q-subtree that satisfies I. By a detailed analysis, we show that in some cases we can find a minimal-height Q-subtree whereas in other cases we only obtain a 2-approximation thereof.

We start with an example. Consider the data graph G of Figure 6. Suppose that I consists of the two PAs P_1 and P_2 that are surrounded by dotted lines. We are looking for a subtree T that contains all the edges of P_1 and P_2. Hence, T cannot include the edge (v_{12}, v_8), because it must include another edge that enters into node v_8, namely, (v_5, v_8). However, T may include edges that enter into the roots of P_1 and P_2, e.g., the edge (v_4, v_5), even if those edges emanate from other nodes of P_1 and P_2. This example motivates the construction of G^e as follows.

G^e is obtained by removing edges that cannot be in any Q-subtree that satisfies I. Formally, G^e is the result of deleting all edges e of G, such that

- $e \notin I$ and
- e enters into a non-root node v of some PA of I.

Note that edges that emanate from the PAs of I are not removed (unless they enter into non-root nodes of those PAs).

Observe that a non-root node of a PA of I cannot be the root of a subtree that satisfies I. For example, node v_2 in Figure 6 cannot be the root, because the edge (v_1, v_2) that enters into v_2 must be in any subtree that satisfies the PA P_1. Thus, we define the set F as follows.

F contains all the nodes that cannot be the root of any Q-subtree that satisfies I. Formally, F is the set of all the nodes of the PAs of I, except for the roots of those PAs.

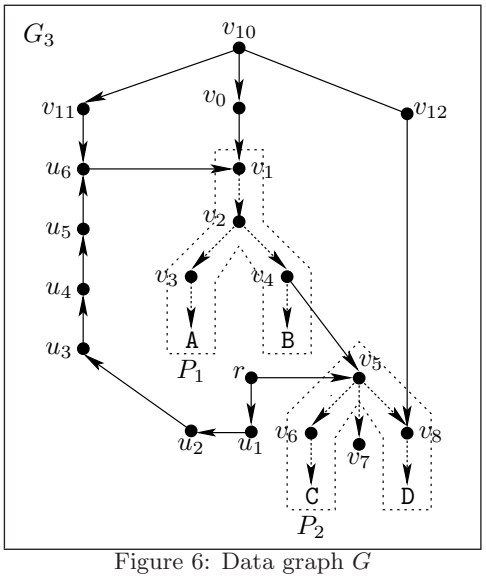

Figure 6: Data graph G

The first step in finding a Q-subtree that satisfies I is to execute MinHeightSubtree(G^e, Q, F). Let T be the result, that is,

$$T = \text{MinHeightSubtree}(G^e, Q, F). \quad (1)$$

It is easy to see the following. First, if $T = \perp$, then no Q-subtree satisfies I. Second, if $T \neq \perp$, then T is a subtree of G that satisfies I and contains all the keywords of Q. Furthermore, if the root of T has at least two children, then T is in fact a minimal-height Q-subtree that satisfies I. So in this case, we are done. But as described below, there are other cases too.

Suppose that in the example of Figure 6, Q is exactly the set $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ comprising the keywords that appear in the two PAs P_1 and P_2. Then we get the subtree that is rooted at v_1 and, other than the edges of P_1 and P_2, it only includes the edge (v_4, v_5). This subtree is not reduced, because its root has a single child. Earlier we argued that if a subtree is not reduced, then we can simply delete the root and get a meaningful answer. However, it cannot be done here, because if we ignore the inclusion constraint imposed by the edge (v_1, v_2) and simply delete that edge, then we will generate a Q-subtree that has already been enumerated earlier.[4] Thus, in order to avoid generating duplicate Q-subtrees while guaranteeing that all Q-subtrees are enumerated, we must look for a minimal-height subtree that has a root with at least two children and includes P_1 and P_2. In Figure 6, there is only one such Q-subtree and its root is node r. In general, however, the existence of a subtree that satisfies I and has a root with only one child does not necessarily imply that there is also a Q-subtree that satisfies I. For example, suppose that r did not exist in Figure 6.

So, we are now faced with the situation that is depicted in Figure 7, where edges are shown as solid lines while paths are dotted lines. We started with the PAs P_1 and P_2 that appear in the upper-left corner of that figure and we got the minimal-height subtree T that appears in the upper-right

[4]Technically, if the enumeration is in an approximate order, then it is possible for that Q-subtree to appear later rather than earlier.

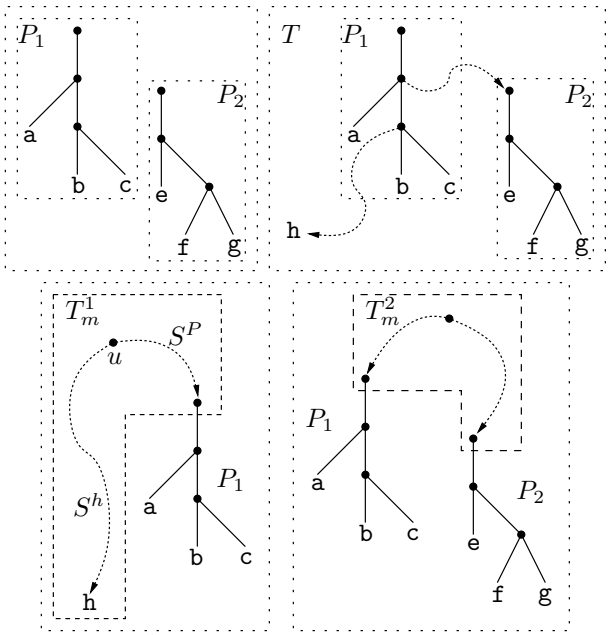

Figure 7: Approximating a minimal-height Q-subtree

corner. The root of T has only one child and, hence, both the root and its child must be in a PA of I, namely, P_1 in this case. Moreover, all the nodes of the second PA, P_2, as well as the keywords (e.g., h) that are in neither P_1 nor P_2, must be descendants of a non-root node of P_1.

Generally, the situation depicted in Figure 7 may occur only when I consists of either one or two PAs, such that at least one of them has a root with a single child. We now show how to transform the subtree T of Figure 7 into a Q-subtree that is a 2-approximation of the minimal-height Q-subtree that satisfies I (or determine that no such Q-subtree exists).

There are two cases to consider, depending on whether the following holds: In a minimal-height Q-subtree A_m that satisfies I, there is a path from the root to a keyword that does not use any edge of I. The two cases are shown at the bottom of Figure 7. Essentially, A_m must contain a subtree that looks either like T_m^1 or like T_m^2. Note, however, that T_m^1 and T_m^2 are not mutually exclusive; that is, A_m may have one part that matches T_m^1 and another part that fits the structure of T_m^2. T_m^1 describes the following situation.

- One keyword h is reachable from the root of A_m through a path S^h that does not use any edge of I; and
- A PA of I, namely P_1, is reachable from the root of A_m through a path S^P that does not include any edge that appears on the path S^h.

The above phrasing of the two conditions is a bit complicated, because it includes the possibility that the root of P_1 is also the root of A_m.

The path S^h must exist in the graph G^n that is obtained from G by deleting all the nodes of the PAs of I, except for the roots of those PAs. The path S^P, on the other hand, may exist in the graph G^e that was defined earlier. This actually means that if I has two PAs, then either one of them may appear first on the path S^P; as a special case, the root of either one of those PAs could be the root of T_m^1.

The above analysis implies that finding T_m^1, if it exists, is done as follows.

1. For each keyword h of Q that is not in any PA of I, compute the minimal-height subtree T^h that includes h and the root of T. The computation is done subject to the following conditions.
   - The root u of T^h cannot be any non-root node of some PA of I.
   - The path from u to h must exist in G^n while the path from u to the root of T is in G^e. We can modify the algorithm MinHeightSubtree of Section 3.4 to handle this restriction: the iterator for h should use G^n while the iterator for the root of T uses G^e.
2. Among all subtrees found in the first step above, let T_m^1 be the one with the minimal height. If no subtree was found, then $T_m^1 = \perp$.

We now discuss the second case shown at the bottom of Figure 7. This case applies only when I has two PAs P_1 and P_2. Moreover, it is the only case that applies if P_1 and P_2 include all the keywords of Q. In this case, we should find a minimal-height subtree T_m^2 that includes the roots of P_1 and P_2. If both roots have a single child, then the paths from the root of T_m^2 to the roots of P_1 and P_2 cannot use any edge of P_1 and P_2. Thus, we should look for T_m^2 in the graph G^n. Next, suppose that the root of P_2 has two children (recall that the root of T is that of P_1 and it has a single child). In this case, the root of P_2 can actually be the root of T_m^2, even if it has the same children in both P_2 and T_m^2. So, we build a new graph G' as follows. We remove from G all the edges that enter into non-root nodes of P_2 and are not in P_2 itself (i.e., P_2 is handled as in the construction of G^e). We also remove all the non-root nodes of P_1 (i.e., P_1 is handled as in the construction of G^n). Now, we compute the minimal-height subtree T_m^2 of G' that includes the roots of P_1 and P_2, subject to the restriction that the root of T_m^2 is not any non-root node of P_2. If no subtree T_m^2 was found, then $T_m^2 = \perp$.

Let T_m be the shorter subtree among T_m^1 and T_m^2 (by definition, $\perp$ has an infinite height). If $T_m = \perp$, then there is no Q-subtree that satisfies I. Otherwise, we do the following.

Let $\hat{T}$ be the graph that comprises the nodes and edges of T and T_m. We obtain a Q-subtree A from $\hat{T}$ as follows. The root of A is the root of T_m and, so, we remove from $\hat{T}$ all the edges that enter into that root. Next, for all nodes $v \in \hat{T}$, if v has two incoming edges, then we remove the edge that originated from T. Finally, we remove all redundant nodes, i.e., nodes that have no descendant keywords. It can be shown that A is a reduced subtree of G w.r.t. Q that includes all the edges of I. The procedure QSubtree of Figure 8 summarizes the construction of the Q-subtree A.

We now sketch the proof that the height of A is at most twice the height of any Q-subtree that satisfies I. Let A_m be a minimal-height Q-subtree that satisfies I. From the construction of A, we can show that

$$height(A) \leq height(T) + height(T_m). \quad (2)$$

T has the minimal height among all subtrees of G that include the edges of I and the keywords of Q. Hence,

$$height(T) \leq height(A_m). \quad (3)$$

Furthermore, as argued above, A_m must include paths that match those of either T_m^1 or T_m^2. The computation of each

Algorithm QSUBTREE$(G, Q, I, \emptyset)$

1: $T \leftarrow$ MINHEIGHTSUBTREE(G^e, Q, F)
2: **if** T is $\perp$ or T is a Q-subtree **then**
3: **return** T
4: **if** not all keywords of Q are in the PAs of I **then**
5: construct the tree T_m^1
6: **if** I consists of two PAs **then**
7: construct the tree T_m^2
8: $T_m \leftarrow$ minimal-height subtree among T_m^1 and T_m^2 (or $\perp$ if neither one exists)
9: **if** T_m is not $\perp$ **then**
10: construct a Q-subtree A from T and T_m
11: **return** A
12: **return** $\perp$

Figure 8: A 2-approximation algorithm

T_m^i guarantees that it has a minimal height. T_m is the shorter tree among T_m^1 and T_m^2. Thus,

$$height(T_m) \leq height(A_m). \quad (4)$$

From Equations 2, 3, and 4, we get

$$height(A) \leq 2\, height(A_m). \quad (5)$$

Thus, A is indeed a 2-approximation and, by a result of [15], the following theorem follows.

THEOREM 3.1. *Given a data graph G and a query Q, all Q-subtrees can be enumerated in a 2-approximate order by increasing height with polynomial delay.*

The delay of the algorithm GENERATEANSWERS of Figure 14 (when the subroutine QSUBTREE implements the above algorithm) is $\mathcal{O}(n_i q(e+n \log n))$, where n and e are the number of nodes and edges, respectively, of G, q is the number of keywords in Q and n_i is the number of nodes in the ith answer. For comparison, the delay of the algorithm of [15] that enumerates by exact weight is $\mathcal{O}(n_i(4^q n + 3^q e \log n))$.

4. RANKING ANSWERS

In this section, we describe the ranker. This component processes the N_i Q-subtrees that are the output of the generator. The result is an ordered list of N_f Q-subtrees.

4.1 The Repeated-Information Problem

The standard way of generating top answers in search applications is to devise a *scoring function* that aims at quantifying the relevance of each answer to the user. Then, this function is applied to a set of candidates (which, in our case, is the output of the answer generator) and the answers with the top scores are selected. This approach is adopted, e.g., in [3, 8, 12, 19]. Nevertheless, when applying keyword search to complex data graphs (e.g., the Mondial database that contains many different relationships among objects) this approach is not effective at all. The main reason is that repeated information in relevant answers requires the user to browse through a huge number of answers in order to actually obtain information. This is illustrated next.

Consider the application of the query "Germany, France, Brussels" over the Mondial database. There are several interesting connections between Germany and France, e.g., a common border, both belong to the European Union (EU) and both contain the Rhine river. There are also relevant interesting connections between France and Brussels, e.g., Brussels hosts the headquarters of the EU (in which France is a member) and Brussels is the capital city of a country that borders France. Each of these connections is relevant and, consequently, the combination of one connection between the first two keywords and a second connection between the last two keywords is typically a relevant answer. But presenting all of these combinations to the user, as the top-ranked answers, is simply ineffective, as the user needs to browse through a large number of answers in order to gain a relatively small amount of information. We call this the *repeated-information* problem.

4.2 The Scoring Functions

In our implementation, a data graph is obtained from an XML document (in particular, edges correspond to nesting of elements and ID references). Hence, we assume that each structural node v has a unique ID and a tag, denoted by $\text{tag}(v)$, that are derived from the corresponding XML element. Next, we give some formal definitions.

Our approach to ranking answers is relative rather than absolute. That is, the criteria for choosing the nth answer in the ranked order depend also on the answers that appear in the first $n-1$ positions. In particular, we use three functions. The first is the *absolute relevance,* denoted by *arl*, that assigns to a Q-subtree A a score $arl(A)$ based just on the properties of A. The second function is the *redundancy penalty,* denoted by *rpt*, that has two arguments: the set $\mathcal{A}_i$ of the Q-subtrees produced thus far (in the first i positions) and an additional Q-subtree A'. The redundancy penalty $rpt(A', \mathcal{A}_i)$ measures the amount of information in A' that has already appeared in the first i answers. The third function is the *score,* denoted by S, and its arguments are $arl(A')$ and $rpt(A', \mathcal{A}_i)$. The Q-subtree in position $i+1$ is the one that yields the highest value of $S(arl(A'), rpt(A', \mathcal{A}_i))$ among all the Q-subtrees A' that have not yet appeared in the ranked order (i.e., $A' \notin \mathcal{A}_i$).

The function *arl* may involve IR and graph measures, as done in existing systems [3,8,12,19]. In our experimentation, $arl(A)$ is inversely proportional to the total weight of A. The total weight is the sum of weights of the nodes and the edges of A. In our experiments, the weight of each node was set to 1. Next, we describe how to assign weights to edges.

The experiments led us to the conclusion that the weight of an edge should reflect the amount of information that the edge bears. As an example, membership in a large organization (e.g., United Nations) provides little information (because almost every two countries are connected by their membership in the UN). Therefore, we devised the following formula that was fine tuned during the experiments. First, a few definitions. Consider a node v and a tag t. The *out degree of v with respect to t*, denoted by $out(v \rightarrow t)$, is the number of edges that lead from v to nodes that have the tag t. Similarly, the *in degree* of v *with respect to* t, denoted by $in(t \rightarrow v)$, is the number of edges that lead from nodes with the tag t to v. Now, consider an edge $e = (v_1, v_2)$, where the tags of v_1 and v_2 are t_1 and t_2, respectively. The weight $w(e)$ of the edge e is defined as follows.

$$w(e) = \log\left(1 + \alpha\, out(v_1 \rightarrow t_2) + (1-\alpha)\, in(t_1 \rightarrow v_2)\right).$$

Thus, the edge carries more information if there are a few edges that emanate from v_1 and lead to nodes that have the same tag as v_2, or a few edges that enter v_2 and emanate

from nodes with the same tag as v_1. We used the constant $\alpha = 0.1$, based on the experiments. An exception to the above formula is an edge that corresponds to an ID reference (i.e., a single reference, as opposed to IDREFS), which represents a many-to-one or one-to-one relationship. This is a strong semantic connection, and the experiments led to the conclusion that the weight of such an edge should be 0.

Next, we explain how the redundancy penalty is computed. The main idea is to compare two Q-subtrees A_1 and A_2 by inspecting the information they provide for every pair of keywords k and k' of Q. The more pairs of keywords with identical or similar information in A_1 and A_2, the higher the degree of redundancy between the two answers. So, the next question is when do A_1 and A_2 provide identical or similar information about k and k'? Intuitively, if both answers state that France and Germany are members of the EU, then they provide identical information about these two keywords. And if one answers says that both countries are members of the EU whereas the other says that they are members of the UN, then these two facts are similar, because they describe they same type of relationship, namely, membership in an organization. Next, we formalize the above intuition.

Consider two subtrees T_1 and T_2 of a data graph G. We say that T_1 and T_2 are *similar* (or *isomorphic*), denoted by $T_1 \sim T_2$, if there exists a bijection $\varphi : \mathcal{V}(T_1) \to \mathcal{V}(T_2)$, such that for all nodes v and v' of T_1 *(1)* if v is structural, then $\text{tag}(\varphi(v)) = \text{tag}(v)$, *(2)* if v is a keyword, then $v = \varphi(v)$, *and (3)* if (v, v') is an edge of T_1, then $(\varphi(v), \varphi(v'))$ is an edge of T_2. As a special case, if $\varphi(v) = v$ for all nodes v of T_1, then T_1 and T_2 are actually the same subtree of G (i.e., $T_1 = T_2$) and in this case we say that T_1 and T_2 are *identical*.

Now, consider two Q-subtrees A_1 and A_2. Let $\{k, k'\}$ be an unordered pair of keywords of Q. We use $A_i[k, k']$ to denote the reduced subtree of A_i with respect to $\{k, k'\}$. Clearly, A_1 and A_2 provide the same information about k and k' if they are identical, namely, $A_1[k, k'] = A_2[k, k']$. Intuitively, A_1 and A_2 convey similar information about k and k' if $A_1[k, k'] \sim A_2[k, k']$, because isomorphism preserves the type of relationship between the two keywords.

We define a similarity function $sim(\cdot, \cdot)$ as follows. If $A_1[k, k'] = A_2[k, k']$, then $sim(A_1[k, k'], A_2[k, k'])$ is equal to 1. If $A_1[k, k'] \neq A_2[k, k']$ and $A_1[k, k'] \sim A_2[k, k']$, then $sim(A_1[k, k'], A_2[k, k']) = c$ for some fixed value $0 \leq c \leq 1$.

Next, we consider the set $\mathcal{A}_n = \{A_1, \ldots, A_n\}$ of the first n answers that have already been generated in ranked order, and let $\hat{A}$ be another answer.

There are two versions of defining the redundancy penalty of $\hat{A}$ with respect to $\mathcal{A}_n$. In the first, we sum for all $A_j \in \mathcal{A}_n$, the similarity scores of $\hat{A}[k, k']$ and $A_j[k, k']$, and then sum over all unordered pairs $\{k, k'\}$ of keywords. That is,

$$rpt_s(\hat{A}, \mathcal{A}_n) = \sum_{\{k,k'\} \in Q} \sum_{j=1}^{n} sim(\hat{A}[k, k'], A_j[k, k']) \,. \qquad (6)$$

In the second version, we take the maximum (and, as in the first version, compute the sum over all unordered pairs of keywords).

$$rpt_m(\hat{A}, \mathcal{A}_n) = \sum_{\{k,k'\} \in Q} \max_{1 \leq j \leq n} sim(\hat{A}[k, k'], A_j[k, k']) \qquad (7)$$

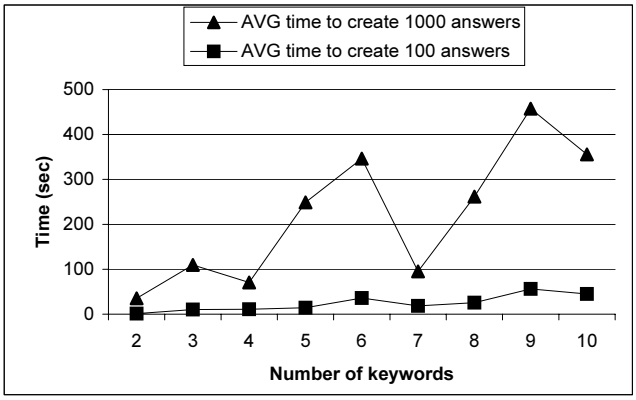

Figure 9: Average running times

Finally, the score S of $\hat{A}$ relative to $\mathcal{A}_n$ is

$$S(arl(\hat{A}), rpt(\hat{A}, \mathcal{A}_n)) = \Big(arl(\hat{A})^{-1} + \epsilon \cdot rpt(\hat{A}, \mathcal{A}_n)\Big)^{-1} \qquad (8)$$

where ϵ is the *penalty factor*. We substitute either rpt_s or rpt_m for rpt in the above formula. Note that $arl(\hat{A})^{-1}$ is the weight of $\hat{A}$.

5. IMPLEMENTATION AND EXPERIMENTATION

The aim of the experiments was threefold. To verify the efficiency of the algorithm, evaluate the correlation between ranking by a 2-approximation of the height vs. the exact weight, and determine the effect of the redundancy penalty on the quality of the ranking.

The experiments used queries that ranged is size from 2 to 10 keywords. For each size, we evaluated four different queries and took the average. The first experiment tested the running time of generating 100 and 1000 answers. The height of an answer was computed according to the weights that are described in the previous section. Figure 9 shows the results. For 3 keywords, as an example, it took about a 100 sec. to generate 1000 answers. Naturally, the running time increases with the size of the query. The following should be borne in mind when comparing the efficiency of our algorithm with other systems. In related work, there are no experiments on complex graphs, such as the Mondial, and the faster algorithms miss answers.

The second experiment tested the correlation between the ranked order generated by the algorithm of Section 3 and the desired ranking according to *arl* (i.e., the inverse of the weight). We generated 1000 answers and then sorted them according to *arl*. Figure 10 shows the position of the nth answer, according to *arl*, in the initial ranked order. As could be expected, the correlation is better for queries of small size, because a Q-subtree with many keywords is more likely to have a large height but an overall small weight.

The third experiment tested the effect of correcting the score by adding the redundancy penalty. We first explain how we measured this effect. Consider a set $\mathcal{A}_n$ of n Q-subtrees for a query that has m keywords. There are $\binom{m}{2}$ unordered pairs $\{k, k'\}$ of keywords. Hence, n Q-subtrees can carry at most $n\binom{m}{2}$ nonsimilar subtrees of the form $A_i[k, k']$, where $A_i \in \mathcal{A}_n$ and $\{k, k'\} \subseteq Q$.

The *non-identicality count* of $\mathcal{A}_n$, denoted by $nic(\mathcal{A}_n)$,

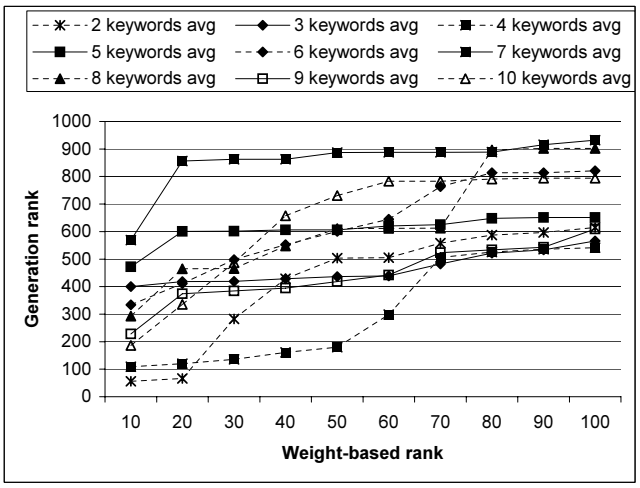

Figure 10: Correlation between ranking by a 2-approximation of the height vs. the exact weight

is the number of elements in the set $Y = \{A_i[k, k'] \mid A_i \in \mathcal{A}_n \text{ and } k, k' \in Q\}$. The *non-similarity count* of $\mathcal{A}_n$, denoted by $nsc(\mathcal{A}_n)$, is the number of elements (i.e., subtrees) in the set that is obtained from Y by eliminating isomorphic subtrees (in other words, it is the size of the maximal subset of Y that does not contain any pair of similar subtrees). Note that $nsc(\mathcal{A}_n) \leq nic(\mathcal{A}_n) \leq n\binom{m}{2}$.

In the third experiment, we tried to figure out what is the best policy of penalizing answers for repeated information. Ideally, we would like to maximize the non-similarity and non-identicality counts while preserving the quality of the ranking. In other words, after adding the penalty (and re-ranking), the average absolute score (i.e., *arl*) of the top-ranked answers should remain about the same, but the non-similarity and non-identicality counts of those answers need to increase substantially.

We performed many experiments in which we varied the following parameters. The value c that *sim* (the similarity function) assigns to isomorphic subtrees was set to either 0.1, 0.5 or 1; the value assigned to identical subtrees was always 1. The number of keywords was $2, 3, \ldots, 10$ (for each size, we used four different queries and took the average). The formula of the redundancy penalty was either rpt_m or rpt_s. Figure 11 gives results of four typical experiments. The horizontal axis is the penalty factor (i.e., the value of ϵ in Equation 8). In this experiment, we generated the first 1000 answers and then ranked them according to the score function S of Equation 8. Note that $\epsilon = 0$ means that the ranked order is according to the absolute relevance *arl*. For each value of ϵ, we calculated the following for the top 20 answers: the average value of S, the non-similarity count and the non-identicality count. The vertical axis in Figure 11 is a percentage that should be interpreted as follows. The graph called "Score" shows the average rank as a percentage of the one achieved when $\epsilon = 0$. The graph called "Connection (tags)" shows the non-similarity count, and the one called "Connection (objects)" is for the non-identicality count. For the last two graphs, the vertical value is the number of distinct connections among pairs of keywords (in the top 20 answers) as a percentage of the theoretical maximum, that is, $20\binom{m}{2}$, where m is the number of keywords. The words "Multiple" and "Once" mean that rpt_s and rpt_m were used,

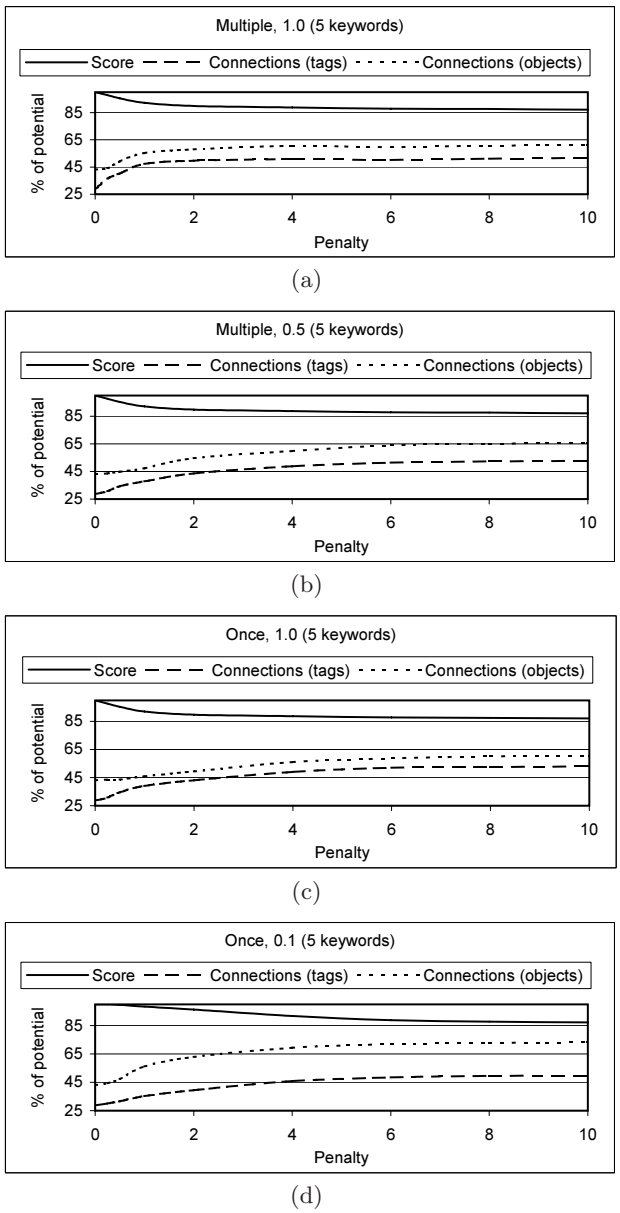

Figure 11: Drop in the score vs. increase in the number of distinct connections

respectively. The number following these words is the value of c.

Rather surprisingly, most of the experiments showed a similar phenomenon. As the penalty factor ϵ was increased, the non-similarity count and the non-identicality count increased sharply for a while and then leveled off. The drop in the average score was rather small during the initial phase of the big increase in the counts. Interestingly, the lowest tested value of c (i.e., 0.1) gave the best results. And rpt_m (i.e., penalizing just once) was better than rpt_s (i.e., penalizing for multiple occurrence of isomorphic subtrees).

The main conclusions from the experiments are as follows. First, our approach can effectively generate a ranked order that has less redundancy, and is able to do so without significantly lowering the relevance of the answers. In particular, penalizing just once (i.e., using rpt_m) and using

a low value for c (i.e., 0.1) is the best. This choice has the effect of a minimal drop in the average score (only a few percentage points) while the increase in the non-similarity and non-identicality counts is the best or close to the best across all the experiments. Moreover, the usage of rpt_m (rather than rpt_s) provides a finer control, by means of changing the value of ϵ, for tuning the tradeoff between the degree of redundancy removal and the decrease of relevancy.

6. EXTENSION TO OR SEMANTICS

Our algorithm for generating answers (Section 3) enumerates Q-subtrees according to the *AND* semantics, that is, an answer should contain all the keywords of Q. In this section, we show how to modify the algorithm so that the enumeration is based on the *OR* semantics, namely, an answer may include only some of the keywords. Formally, consider a data graph G and let Q be a set of keywords. A *Q-or-subtree* of G is a subtree that is reduced w.r.t. a subset of Q that has at least two keywords. Note that a Q-subtree is a special case of a Q-or-subtree. In order to enumerate all the Q-or-subtrees with polynomial delay in a 2-approximate order (by increasing height), we need to make the following changes.

Recall that the algorithm GenerateAnswers (Figure 14 in the appendix) uses a priority queue of elements of the form $\langle I, E, A\rangle$, where A is a 2-approximation of the minimal-height Q-subtree that satisfies the set of inclusion constraints I and the set of exclusion constraints E. In the new algorithm, A is a Q-or-subtree and there are two types of sets of inclusion constraints that are denoted by I and I^+, respectively. As previously defined, a Q-or-subtree A satisfies I if it includes all the edges of I. We also define that A satisfies I^+ if it has the edges of I^+ as well as (at least) one additional edge that is not in I^+. Note that if I^+ itself is a Q-or-subtree, then an answer A satisfies I^+ if and only if it has (at least) one keyword in addition to those in I^+.

When a triplet $\langle I, E, A\rangle$ is removed from the queue, new elements are inserted as previously. Additionally, the element $\langle I^+_{k-h+1}, E_{k-h+1}, A_{k-h+1}\rangle$ is also inserted into the queue, where I^+_{k-h+1} is exactly the set of edges of A, the set E_{k-h+1} is the same as E and A_{k-h+1} is a 2-approximation of the minimal-height Q-or-subtree that satisfies both I^+_{k-h+1} and E_{k-h+1}. Thus, the loop of Line 10 in Figure 14 now has $k-h+1$ iterations and, in the last iteration, a set of inclusion constraints of the new form I^+ is created. The only other case of creating a set of this form is in the first iteration after removing an element $\langle I^+, E, A\rangle$, i.e., an element that has a set of inclusion constraints of the new form.

Modifications are also needed in the subroutine QSubtree that should now compute a 2-approximation T of the minimal-height Q-or-subtree that satisfies the given set of inclusion constraints, which is of the form I or I^+. (As previously, we may assume that E is empty.) The subtree T is constructed as follows.

1. If the set of inclusion constraints is empty (and, hence, is of the form I), then for each pair of keywords of Q, we generate a minimal-height subtree of G that is reduced w.r.t. these two keywords. T is the subtree with the minimum height among all the generated ones.

2. If the set of inclusion constraints is of the form I^+, then it must be a Q-or-subtree. So, T is the minimal-height subtree among all those that contain the edges of I^+ and exactly one additional keyword that is not in I^+. (The root of T must have at least two children.)

3. For a nonempty set of the form I, there are several cases as follows. (a) If I is a Q-or-subtree, then it is the result T. (b) If I consists of a single PA that has a root with only one child, then T is the minimal-height Q-or-subtree among all those that contain the edges of I as well as one additional keyword that is not in I. The construction of T is the same as the one that generates the subtree T^1_m of Section 3. (c) If I has two PAs, then we do the following. First, we find the subtree T^2_m of Section 3. Second, for each keyword h that is not in I, we find (as described in Section 3) a 2-approximation T^h of the minimal-height Q^h-subtree that satisfies I, where Q^h comprises the keywords of I and h. T is the minimal-height subtree among T^2_m and all the T^h.

The OR semantics is used in [8, 19]. They also have a slightly different notion of a data graph, since they allow multiple occurrences of the same keyword. Essentially, nodes correspond to tuples that may contain many keywords; similarly, a keyword can appear in any number of nodes. There is a straightforward reduction from the model of [8,19] to the one used in this paper. Formally, a query Q comprises all the nodes that contain some of the keywords posed by the user. (Hence, Q may have nodes with outgoing edges.) The definition of an answer has to be modified, since the root may have only one child if it contains some of the keywords that appear in the query. Only minor changes are needed, in the above algorithm, in order to enumerate all the Q-or-subtrees when the data graph is of the type used in [8,19]. In particular, note that the above algorithm does not assume that the nodes of Q have no outgoing edges.

7. CONCLUSION

The architecture of a generator and a ranker is essential for overcoming the obstacles that arise when applying keyword search to complex data graphs. Not only is there a huge number of possible answers, but unlike standard IR settings, even a set of highly relevant answers might be overwhelmingly redundant due to the repeated-information problem.

We have presented an answer generator that is the first one with all the essential properties of polynomial delay, weight-correlated order (i.e., 2-approximate order by height) and a guarantee not to miss answers (with small weights). Our experiments show that, in addition to the provable properties, the answer generator is practically effective.

The ranker we presented is aimed at eliminating repeated information by incorporating *global* measures. Namely, it penalizes answers based on their degree of similarity to the ones that have already been given a final rank. Thus, the ranking function actually changes throughout the ranking process. We developed a methodology that compares the decrease in the absolute relevancy with the drop in the redundancy. This methodology enabled us to detect a specific ranking method that properly deals with the repeated-information problem at a low cost (i.e., a very small decrease in the absolute relevancy).

8. REPEATABILITY ASSESSMENT RESULT

All the results in this paper were verified by the SIGMOD repeatability committee.

9. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: enabling keyword search over relational databases. In *SIGMOD*, 2002.

[2] J. Allan, V. Lavrenko, and H. Jin. First story detection in TDT is hard. In *CIKM*, 2000.

[3] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.

[4] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, 2007.

[5] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[6] N. Fuhr and K. Großjohann. XIRQL: A query language for information retrieval in XML documents. In *SIGIR*, 2001.

[7] N. Fuhr, M. Lalmas, S. Malik, and G. Kazai, editors. *4th International Workshop of the Initiative for the Evaluation of XML Retrieval*. Springer, 2006.

[8] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.

[9] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, 2002.

[10] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword proximity search on XML graphs. In *ICDE*, 2003.

[11] D. Johnson, M. Yannakakis, and C. Papadimitriou. On generating all maximal independent sets. *Info. Proc. Lett.*, 27, 1988.

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

[13] G. Kazai and M. Lalmas. extended cumulated gain measures for the evaluation of content-oriented XML retrieval. *ACM Trans. Inf. Syst.*, 24(4), 2006.

[14] B. Kimelfeld and Y. Sagiv. Efficiently enumerating results of keyword search. In *DBPL*, 2005.

[15] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, 2006.

[16] B. Kimelfeld and Y. Sagiv. Efficiently enumerating results of keyword search over data graphs. To appear in *Information Systems*, 2008.

[17] 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, 1972.

[18] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal. Retrieving and organizing web pages by "information unit". In *WWW*, 2001.

[19] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD Conference*, 2006.

[20] D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, and J. Widom. Querying semistructured heterogeneous information. In *DOOD*, 1995.

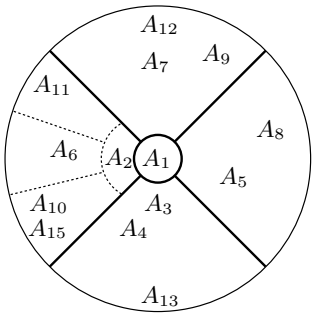

Figure 12: Illustration of GenerateAnswers

[21] N. Stokes and J. Carthy. Combining semantic and syntactic document classifiers to improve first story detection. In *SIGIR*. ACM, 2001.

[22] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for topx search. In *VLDB*, 2005.

[23] S. Wang, Z. Peng, J. Zhang, L. Qin, S. Wang, J. X. Yu, and B. Ding. NUITS: A novel user interface for efficient keyword search over databases. In *VLDB*, 2006.

[24] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17, 1971.

[25] J. Y. Yen. Another algorithm for finding the k shortest loopless network paths. In *Proc. 41st Mtg. Operations Research Society of America*, volume 20, 1972.

[26] Y. Zhang, J. P. Callan, and T. P. Minka. Novelty and redundancy detection in adaptive filtering. In *SIGIR*. ACM, 2002.

APPENDIX

A. THE BASIC ENUMERATION METHOD

A.1 The Main Idea

Lawler [17] generalized an algorithm of Yen [24, 25] to a procedure for computing the top-k solutions to discrete optimization problems. The algorithm GenerateAnswers of Figure 14 is an adaptation of Lawler's procedure to enumerating all Q-subtrees of a data graph G.

We start with an explanation of how Lawler's procedure works. The circle of Figure 12 contains the elements that we want to enumerate. Pictorially, elements that are closer to the center have a higher rank. So, A_1 is the first element that should be enumerated. An algorithm for finding the top-ranked element can be used for finding A_1. After printing A_1, the remainder of the circle is divided into disjoint subsets. Generally, the number of these subsets depends on A_1. In Figure 12, it is assumed that there are four subsets and they are shown as the slices separated by thick lines. Now, we should find the top-ranked element in each slice, thereby obtaining A_2, A_3, A_5 and A_7. These four elements are inserted into a priority queue. Clearly, the second element in the enumeration is the one at the top of the queue, namely, A_2. After printing A_2, the remainder of the slice (from which A_2 was taken) is divided into sub-slices, i.e., the ones separated by dotted lines. In each sub-slice, we find the top-ranked element and add it to the queue. Now, the queue has six elements, i.e., A_3, A_5, A_7 and the three new elements A_{10}, A_6 and A_{11}. The top element is

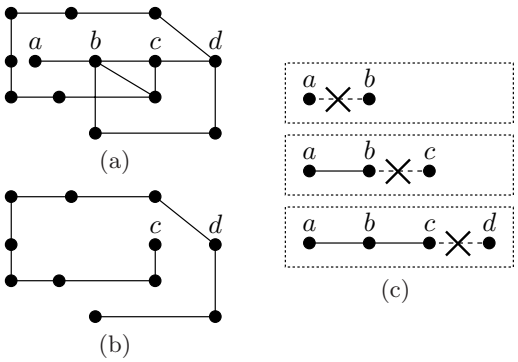

Figure 13: Illustration of Yen's procedure

removed from the queue, printed, its slice is divided into sub-slices, the top-ranked element in each sub-slice is added to the queue and so on.

There are two problems that should be solved when using Lawler's procedure. First, how to divide each slice into sub-slices. Second, how to find the top-ranked element in each slice. Lawler addressed the first problem but not the second.

As a concrete example of using Lawler's procedure, we consider Yen's original algorithm for enumerating (by increasing length) all simple paths between two given nodes. In Figure 13(a), the shortest path between nodes a and d has three edges and it goes through nodes b and c (Dijkstra's algorithm can be used for finding this path). So, the following three subsets (i.e., "slices") are created, as illustrated in Figure 13(c). 1. The subset of all paths between a and d that do not contain the edge (a, b). 2. The subset of all paths between a and d that contain the edge (a, b), but not the edge (b, c). 3. The subset of all paths between a and d that contain the edges (a, b) and (b, c), but not (c, d). In other words, each subset comprises all paths that contain some (possibly empty) prefix of the shortest path from a to d and do not contain the next edge that follows that prefix. Two things should be noted. First, the subsets are disjoint. Second, their union contains all paths between a and b except for the shortest one, which has already been enumerated.

Now, we have to find the shortest path in each subset. This can be done by a reduction to the regular shortest-path problem. As an example, suppose that we want to find the shortest path in the third subset above. Nodes a and b (and their incident edges) are removed from the graph. The edge (c, d) is also removed and, in the resulting graph that is shown in Figure 13(b), we find the shortest path between c and d. We add to that path the edges (a, b) and (b, c) in order to obtain the shortest path of the third subset.

A.2 Adaptation to Keyword Proximity Search

The algorithm GenerateAnswers of Figure 14 enumerates all the Q-subtrees of a data graph G. It is an adaptation of Lawler's procedure that is described above. We use elements of the form $\langle I, E, A\rangle$ in order to partition the search space. An element $\langle I, E, A\rangle$ represents the subspace comprising all the Q-subtrees that satisfy the set of inclusion constraints I and the set of exclusion constraints E, where A is a 2-approximation of the minimal-height Q-subtree in this subspace. Since I must be a proper set of inclusion constraints w.r.t. Q, the order of the edges is important and,

Algorithm GenerateAnswers(G, Q)

```
 1: Queue← an empty priority queue
 2: A←QSubtree(G, Q, ⟨⟩, ∅)
 3: if A ≠⊥ then
 4:    Queue.insert(⟨⟨⟩, ∅, A⟩)
 5: while Queue ≠ ∅ do
 6:    ⟨I, E, A⟩←Queue.removeTop()
 7:    print(A)
 8:    let I = ⟨e_1, ..., e_h⟩
 9:    ⟨e_1, ..., e_k⟩←Serialize(A, I)
10:    for i←1, ..., k − h do
11:       I_i←⟨e_1, ..., e_{h+i−1}⟩
12:       E_i←E ∪ {e_{h+i}}
13:       A_i←QSubtree(G, Q, I_i, E_i)
14:       if A_i ≠⊥ then
15:          Queue.insert(⟨I_i, E_i, A_i⟩)
```

Figure 14: Basic algorithm for enumerating Q-subtrees

therefore, we write a particular I as a sequence $\langle e_1, \ldots, e_k\rangle$, rather than as a set.

The procedure QSubtree(G, Q, I, E) is the one described in Section 3. It returns a 2-approximation of the minimal-height Q-subtree that satisfies I and E; if there is no such subtree, then it returns $\perp$.

The algorithm GenerateAnswers of Figure 14 starts in Line 1 by initializing an empty priority queue. In Line 2, QSubtree is called with the empty sets of inclusion and exclusion constraints, and it returns the minimal-height Q-subtree A. If A exists (i.e., $A \neq \perp$), then the triplet $\langle\langle\rangle, \emptyset, A\rangle$ is inserted into the priority queue (Line 4). The loop of Line 5 removes the top element $\langle I, E, A\rangle$ from the queue (i.e., the height of A is currently the minimum), prints A and then inserts new elements into the queue, in the loop of Line 10, as follows.

The first step is to arrange all the edges of A in a sequence $\langle e_1, \ldots, e_k\rangle$, such that the first h edges are exactly those of I (Line 8) and every prefix $\langle e_1, \ldots, e_i\rangle$ $(h \leq i < k)$ is a proper set of inclusion constraints (Line 9). Now, let S denote the set comprising all the Q-subtrees that satisfy both I and E, except for A itself. The loop of Line 10 partitions S into pairwise disjoint subsets $S_1, \ldots, S_{k-h}$ and inserts, into the queue, elements that represent these subsets. Note that if $h = k$, i.e., the set I already contains all the edges of A, then the loop of Line 10 is not executed at all. For $i = 1, 2, \ldots, k - h$, the subset S_i comprises all the Q-subtrees that satisfy I_i and E_i. The set I_i consists of the first $h+i-1$ edges in the sequence $\langle e_1, \ldots, e_k\rangle$ and, hence, it contains I. The set E_i contains E and the edge e_{h+i}. Thus, the subsets S_i are pairwise disjoint and their union is exactly S. In Line 13, a 2-approximation A_i of the minimal-height Q-subtree, among those in S_i, is computed and if it exists, then the triplet $\langle I_i, E_i, A_i\rangle$ is inserted into the queue.

The crux of applying GenerateAnswers to a specific enumeration problem is finding a polynomial-time implementation of QSubtree. This is done in Section 3.