# Finding and Approximating Top-k Answers in Keyword Proximity Search[*]

[Extended Abstract]

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

## ABSTRACT

Various approaches for keyword proximity search have been implemented in relational databases, XML and the Web. Yet, in all of them, an answer is a $Q$-fragment, namely, a subtree $T$ of the given data graph $G$, such that $T$ contains all the keywords of the query $Q$ and has no proper subtree with this property. The rank of an answer is inversely proportional to its weight. Three problems are of interest: finding an optimal (i.e., top-ranked) answer, computing the top-$k$ answers and enumerating all the answers in ranked order. It is shown that, under data complexity, an efficient algorithm for solving the first problem is sufficient for solving the other two problems with polynomial delay. Similarly, an efficient algorithm for finding a $\theta$-approximation of the optimal answer suffices for carrying out the following two tasks with polynomial delay, under query-and-data complexity. First, enumerating in a $(\theta+1)$-approximate order. Second, computing a $(\theta+1)$-approximation of the top-$k$ answers. As a corollary, this paper gives the first efficient algorithms, under data complexity, for enumerating all the answers in ranked order and for computing the top-$k$ answers. It also gives the first efficient algorithms, under query-and-data complexity, for enumerating in a provably approximate order and for computing an approximation of the top-$k$ answers.

**Categories and Subject Descriptors:** F.2.2 [**Nonnumerical Algorithms and Problems**]: Computations on discrete structures; H.3.3 [**Information Search and Retrieval**]: Search process; H.2.4 [**Systems**]: Query processing

**General Terms:** Algorithms

**Keywords:** Keyword proximity search, top-k answers, approximations of top-k answers

[*]This research was supported by The Israel Science Foundation (Grants 96/01 and 893/05).

## 1. INTRODUCTION

Typically, the aim of keyword search is to find chucks of text that are relevant to the keywords. Sometimes, there is also an underlying structure that can be taken into account when ranking answers, e.g., PageRank [3]. In *keyword proximity search,* however, not only does the structure influence the ranking process—it is also part of the answer. As an example, when searching for the keywords "Vardi" and "Databases" in DBLP, one answer might be a paper on databases written by Vardi while another could be a journal on databases that includes a paper with Vardi as an author. Only the structure, as opposed to just the occurrences of the keywords and the surrounding text, can indicate the full meaning of the answer. Typically, the structure can be represented as a graph, where keywords constitute some of the nodes, while other nodes as well as the edges indicate the structure. An answer is a subtree that includes the keywords of the query. By assigning weights to edges and possibly to nodes, answers that represent different types of associations between the keywords can be ranked apart.

Several systems for keyword proximity search have been implemented in recent years. DBXplorer [1], BANKS [2, 14] and DISCOVER [11] support keyword search in relational databases. These systems represent data as a graph, where nodes correspond to tuples and keywords. Edges connect tuples to each other, according to foreign keys, and also connect tuples with the keywords appearing in those tuples. XKeyword [12] implements a similar approach for XML. In the "information unit" approach of [20] for searching the Web, the aim is somewhat different—representing an answer as a collection of pages that include the given keywords, rather than just a single page. Although these systems use different techniques, it was shown in [16] that they actually solve variants of the same graph problem.

The above systems implement algorithms that are based on heuristics, in an attempt to reduce the search space. In some of these systems, the efficiency is further enhanced at the cost of generating the answers in an "almost," rather than the exact, ranked order. In the worst case, however, the runtime is exponential, even if the size of the query is bounded and there are only a few answers.[1] The rationale for using a heuristic approach is the intractability of finding the top-ranked answer (because the Steiner-tree problem is NP-hard). However, two questions naturally arise. First, if

[1]The algorithm used in BANKS is an exception, since it has a polynomial delay, under data complexity, but it neither finds all the answers nor guarantees an approximate order.

the size of the query is bounded, the Steiner-tree problem is tractable [8, 5]. So, can the top-$k$ answers be computed efficiently under data complexity? Second, can the vast amount of approximation algorithms for Steiner-trees (e.g., [4, 10, 9, 18, 25, 21]) be used in order to find an approximation (as defined in [7]) of the top-$k$ answers? In this paper, we show that the answer to both questions is positive.

We measure the runtime of an enumeration algorithm in terms of the *delay,* that is, the length of time between generating two successive answers. An algorithm is efficient if it enumerates answers with polynomial delay (i.e. polynomial in the size of the input). Our results hold within the formal framework defined in Section 2 and are divided into two cases: exact answers and approximations; in the first case, the query is assumed to be of a fixed size. For both cases, we show that an efficient algorithm $\mathcal{A}$ for finding just one answer is sufficient for enumerating all the answers with polynomial delay. The enumeration is either in ranked order or in an approximate order, depending on whether $\mathcal{A}$ computes the top-ranked answer or only an approximation thereof. It follows that either the top-$k$ answers or their approximations can be computed in time that is linear in $k$ and polynomial in the size of the input.

Our first result (i.e., enumerating in ranked order) is based on three reductions. We start by adapting the general procedure of Lawler [19] to our case, thereby reducing the problem of enumerating in ranked order to the problem of finding an optimal answer under constraints. In Section 4, we show how to do this reduction so that the optimization problem is a tractable one. In Section 5, we describe algorithms for computing a supertree that includes some node-disjoint subtrees; these algorithms use a reduction to finding Steiner trees. In Section 6, we reduce the optimization problem of Section 4 to the problem of finding a minimal supertree.

For our second result, we show that Lawler's procedure can be used for enumerating in an approximate order. It is also necessary, however, to modify the sequence of reductions, as described in Section 7.

There are actually three variants of each of our two results. The formal framework (Section 2) and the exposition of the results (Section 3) are described for all three variants, but the algorithms themselves are given only for the directed case (i.e., answers are directed subtrees).

## 2. THE FORMAL SETTING

### 2.1 Enumerations and Top-k Answers

An instance of an *enumeration problem* consists of an input $x$ and a finite set $\mathcal{A}(x)$ of *answers.* An *enumeration algorithm* $E$ prints all the answers of $\mathcal{A}(x)$ without repetitions. We assume that there is an underlying *ranking function* that maps answers to positive real numbers. The rank of an answer $a$ is denoted by $\mathit{rank}(a)$. Suppose that the algorithm $E$ enumerates the sequence $a_1, \dots, a_n$. If $\mathit{rank}(a_i) \geq \mathit{rank}(a_j)$ holds for all $1 \leq i \leq j \leq n$, then the enumeration is in *ranked order.* The delay of $E$ is the length of time (i.e., execution cost) between printing two successive answers. There is also a delay at the beginning, i.e., until the first answer is printed, and at the end, i.e., until the algorithm terminates after printing the last answer. Formally, the *ith delay* ($1 \leq i \leq n$) starts immediately after printing the $(i-1)$st answer (or at the beginning of the execution if $i = 1$) and ends when the $i$th answer is printed; the $(n+1)$st delay is

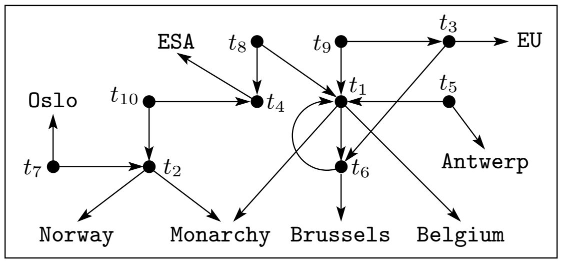

Figure 1: Data graph $G_1$

the delay at the end. We say that $E$ enumerates with *polynomial delay* [13] if there is a polynomial in the size of the input that bounds the $i$th delay for all $i$.

There are two related problems. First, finding an *optimal* (i.e., *top-ranked*) answer, that is, printing an answer that has the highest rank (or $\perp$ if there is no answer at all). Second, finding the *top-k* answers (or all the answers if they are fewer than $k$). An optimal answer is found efficiently if the runtime is polynomial in the size of the input $x$. Typically, an algorithm for finding the top-$k$ answers is considered efficient if the runtime is polynomial in the size of the input $x$ and the value of $k$ (rather than the size of its binary representation). Note that the runtime is at least linear in $k$ and, in principle, $k$ may be larger than the size of $x$.

Given an algorithm $E$ that enumerates in ranked order with polynomial delay, we can obtain an efficient algorithm for finding the top-$k$ answers by stopping the execution after generating $k$ answers. In particular, we can use $E$ for finding efficiently an optimal answer. There are two important advantages to such an approach. First, the running time is always linear in $k$ (and polynomial in $x$). Second, $k$ need not be known in advance (hence, the user can decide whether more answers are required, based on the first ones).

### 2.2 Approximations

Sometimes, for the sake of efficiency, we are interested in *approximations.* The quality of an approximation is determined by an *approximation ratio* $\theta > 1$ ($\theta$ may be a function of the input $x$). Given an input $x$, a $\theta$-*approximation* of an optimal answer is any answer $\mathit{app} \in \mathcal{A}(x)$, such that $\theta \cdot \mathit{rank}(\mathit{app}) \geq \mathit{rank}(a)$ for all answers $a \in \mathcal{A}(x)$; if $\mathcal{A}(x) = \emptyset$, then $\perp$ is a $\theta$-approximation. Following Fagin et al. [7], a $\theta$-approximation of the top-$k$ answers is any set $\mathit{AppTop}$ consisting of $\min(k, |\mathcal{A}(x)|)$ answers, such that $\theta \cdot \mathit{rank}(a) \geq \mathit{rank}(a')$ holds for all $a \in \mathit{AppTop}$ and $a' \in \mathcal{A}(x) \setminus \mathit{AppTop}$.

Enumerating all the answers in a $\theta$-*approximate order* means that if one answer precedes another, then the first one is at most $\theta$ times worse than the other one. More formally, the sequence of all answers $a_1, \dots, a_n$ is in a $\theta$-approximate order if $\theta \cdot \mathit{rank}(a_i) \geq \mathit{rank}(a_j)$ for all $1 \leq i \leq j \leq n$. An algorithm that enumerates in a $\theta$-approximate order with polynomial delay can also find efficiently a $\theta$-approximation of the top-$k$ answers, by stopping the execution after printing $k$ answers. In particular, it can find efficiently a $\theta$-approximation of an optimal answer.

### 2.3 The Data Model

A *data graph* $G$ consists of a set $\mathcal{V}(G)$ of *nodes* and a set $\mathcal{E}(G)$ of *edges.* Nodes are divided into two types: $\mathcal{S}(G)$ is the set of *structural* nodes and $\mathcal{K}(G)$ is the set of *keyword* nodes (or *keywords* for short). Unless explicitly stated otherwise,

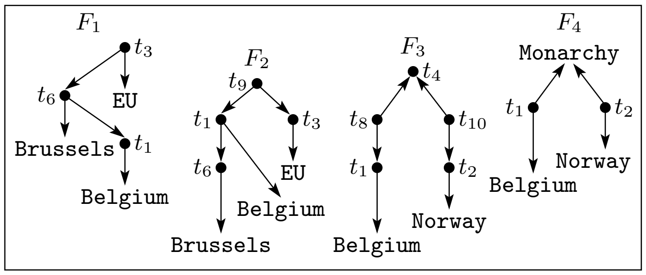

Figure 2: Four fragments of $G_1$

edges are directed, i.e., an edge is a pair $(n_1, n_2)$ of nodes. Keywords have only incoming edges, while structural nodes may have both incoming and outgoing edges. Hence, no edge can connect two keywords. These restrictions imply that $\mathcal{E}(G) \subseteq \mathcal{S}(G) \times \mathcal{V}(G)$. For example, consider the data graph $G_1$ depicted in Figure 1. Filled circles represent structural nodes and keywords are written in typewriter font.

The *weight* function $w_G$ assigns a positive weight $w_G(e)$ to every edge $e \in \mathcal{E}(G)$. The weight of the data graph $G$, denoted by $w(G)$, is the sum of the weights of all its edges, i.e., $w(G) = \sum_{e \in \mathcal{E}(G)} w_G(e)$. In the examples of this paper, we assume that all edges have an equal weight and, hence, that weight is not shown explicitly.

An *undirected subtree* of a data graph $G$ is a connected subgraph without cycles, assuming that edges are undirected. A *directed 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. We use $root(T)$ to denote the root of $T$ and $leaves(T)$ to denote the set of all the leaves of $T$, i.e., the set of all nodes that have no outgoing edges.

Consider a subset $U$ of the nodes of a data graph $G$, where $|U| > 1$. A directed (respectively, undirected) subtree $T$ of $G$ is *reduced* w.r.t. $U$ if $T$ contains $U$, but no proper directed (respectively, undirected) subtree of $T$ also contains $U$. Note that if $U$ consists of keyword nodes, then a directed subtree $T$ is reduced w.r.t. $U$ if and only if $leaves(T) = U$ and $root(T)$ has more than one child. A directed (respectively, undirected) *Steiner tree* for $U$ is a minimal-weight directed (respectively, undirected) subtree of $G$ that contains $U$.

## 2.4 Fragments

A *query* is simply a set $Q$ of keywords. Given a data graph $G$, an answer to $Q$ is a *$Q$-fragment*, namely, a reduced subtree of $G$ w.r.t. $Q$. There are three types of $Q$-fragments: directed $Q$-fragments (DQFs), undirected $Q$-fragments (UQFs) and *strong* $Q$-fragments (SQFs). Fragments of the later type (i.e., SQFs) are UQFs that have keywords only in their leaves (in other words, only structural nodes are incident to two or more edges).

As an example, consider again the data graph $G_1$ of Figure 1. Figure 2 depicts four subtrees $F_1, \ldots, F_4$ of $G_1$. Let $Q_1$ be the query $\{\texttt{Belgium}, \texttt{Brussels}, \texttt{EU}\}$ and $Q_2$ be the query $\{\texttt{Belgium}, \texttt{Norway}\}$. $F_1$ and $F_2$ are directed $Q_1$-fragments of $G_1$. The $Q_2$-fragment $F_3$ is strong but not directed. The $Q_2$-fragment $F_4$ is undirected but neither directed nor strong. Note that $F_4$ is also an undirected $Q_3$-fragment, where $Q_3 = \{\texttt{Belgium}, \texttt{Norway}, \texttt{Monarchy}\}$.

We take the common approach (e.g., [11, 20, 12, 1]) that the rank of an answer is inversely proportional to its weight. Thus, for a $Q$-fragment $F$, we define $rank(F) = w(F)^{-1}$. In particular, a top-ranked $Q$-fragment has the minimum weight. Since we always assume that queries have at least two keywords, the weight of a $Q$-fragment is not zero.

## 3. A SUMMARY OF THE MAIN RESULTS

Our main results are algorithms for enumerating DQFs, UQFs and SQFs. In this section, we summarize their complexity. First, note that optimal answers correspond to Steiner trees. Specifically, the top-ranked DQF and UQF are directed and undirected Steiner trees, respectively, while the top-ranked SQF corresponds to a *group* Steiner tree. Finding Steiner trees is intractable under *query-and-data* complexity, that is, when the size of the query is unbounded. In many cases, however, *data complexity* [22] is more suitable for measuring the efficiency of algorithms for answering queries. Under this notion, the size of the query $Q$ is assumed to be fixed (note that the number of answers could still be exponential in the size of the data graph).

The following theorem shows that, under data complexity, all three types of $Q$-fragments can be enumerated in ranked order with polynomial delay. The algorithms for DQFs, UQFs and SQFs are reductions to algorithms that solve the corresponding Steiner-tree problems; hence, any algorithm for Steiner trees can be used. In [15], we show how the algorithm of Dreyfus and Wagner [5] for finding undirected Steiner trees can be adapted to the directed and the group variants. Our results for enumerating in ranked order use the running times given in [15].

We assume that the data graph $G$ has $n$ nodes and $e$ edges. The number of keywords in the query $Q$ is denoted by $q$. We use $n_i$ to denote the number of nodes in the $i$th generated $Q$-fragment (i.e., the one that is printed at the end of the $i$th delay). Note that for the last delay (which ends when the algorithm terminates), we assume that $n_i = 0$.

THEOREM 3.1. *Given a data graph $G$ and a query $Q$, either all DQFs, all UQFs or all SQFs can be enumerated in ranked order with polynomial delay, under data complexity. The $i$th delay is $\mathcal{O}\left(n_i(4^q n + 3^q e \log n)\right)$.*

COROLLARY 3.1. *Under data complexity, the top-$k$ answers (DQFs, UQFs or SQFs) can be efficiently computed.*

The literature has a lot of work on efficient approximation algorithms for the different types of Steiner trees (see [6] for a detailed review). The next theorem shows that these algorithms can be used for enumerating answers (i.e., either DQFs, UQFs or SQFs) in approximately ranked order with polynomial delay, under query-and-data complexity. Specifically, given an approximation algorithm for Steiner trees, $f$ denotes its running time and $\theta$ denotes its approximation ratio. We assume that both $f$ and $\theta$ are monotonically non-decreasing functions of their arguments; that is, if $n_1 \le n_2$, $e_1 \le e_2$ and $q_1 \le q_2$, then $f(n_1, e_1, q_1) \le f(n_2, e_2, q_2)$ and $\theta(n_1, e_1, q_1) \le \theta(n_2, e_2, q_2)$. Hence, $n$, $e$ and $q$ can indeed be used as the three arguments of $f$ and $\theta$ in the following theorem and corollary.

THEOREM 3.2. *If a $\theta$-approximation of the optimal answer can be found in time $f$, then all answers can be enumerated in a $(\theta + 1)$-approximate ranked order with polynomial delay. The $i$th delay is $\mathcal{O}\left(n_i(f + e \log n)\right)$.*

COROLLARY 3.2. *Under query-and-data complexity, if a $\theta$-approximation of the optimal answer can be found efficiently, then a $(\theta + 1)$-approximation of the top-$k$ answers can be efficiently computed.*

In the remainder of this paper, we describe only the algorithms for enumerating DQFs. Accordingly, in the sequel, "subtrees" are always "directed subtrees." The conclusion discusses briefly the algorithms for UQFs and SQFs.

## 4. THE BASIC ALGORITHM

Lawler [19] generalized an algorithm of Yen [23, 24] to a procedure for computing the top-$k$ solutions of discrete optimization problems. The algorithm DQFSEARCH of Figure 3 is an adaptation of Lawler's procedure to enumerating all DQFs, given a query $Q$ and a data graph $G$. In this algorithm, a *constraint* is an edge. A DQF $F$ *satisfies* a set $I$ of *inclusion* constraints and a set $E$ of *exclusion* constraints if it has all the edges of $I$ and none of the edges of $E$. The algorithm employs a subroutine FRAGMENT$(G, Q, I, E)$ that returns a DQF of $G$ that satisfies both $I$ and $E$. If no such DQF exists, then $\perp$ is returned. The following sections describe two options for implementing this subroutine. One returns a minimal (i.e., optimal) DQF satisfying $I$ and $E$ and the other returns an approximation of such a DQF. We now describe how DQFSEARCH works, assuming that FRAGMENT returns a minimal DQF.

The algorithm DQFSEARCH uses a priority queue *Queue*. An element in *Queue* is a triplet $\langle I, E, F\rangle$, where $I$ and $E$ are sets of inclusion and exclusion constraints, respectively, and $F$ is the top-ranked DQF that satisfies both $I$ and $E$. Priority of $\langle I, E, F\rangle$ in *Queue* is based on the rank of the DQF $F$; that is, the top element of *Queue* is a triple $\langle I, E, F\rangle$, such that $rank(F)$ is maximal. The operation *Queue*.**removeTop**() removes the top element and returns it to the caller. We assume that operations on the priority queue take logarithmic time in the size of *Queue*, i.e., polynomial time in the size of the input.

The algorithm DQFSEARCH$(G, Q)$ starts by finding, in Line 2, a DQF $F$ that satisfies the empty sets of inclusion and exclusion constraints. Thus, $F$ is the top-ranked DQF. In Line 4, the triplet $\langle\emptyset, \emptyset, F\rangle$ is inserted into *Queue*. In the main loop of Line 5, the top triplet $\langle I, E, F\rangle$ is removed from *Queue* in Line 6. In principle, the algorithm should add to *Queue* the second DQF, in ranked order, among all those satisfying both $I$ and $E$. But instead of doing that, the algorithm partitions the set comprising all the DQFs satisfying $I$ and $E$, except for the top-ranked DQF $F$, and adds the top-ranked DQF of each partition to *Queue*. The partitioning is done as follows. Let $e_1, \ldots, e_k$ be the edges of $F$ that are not in $I$. The $i$th partition $(1 \le i \le k)$ is obtained by adding the first $i-1$ edges $e_1, \ldots, e_{i-1}$ to $I$ and adding the $i$th edge $e_i$ to $E$, thereby creating the new sets of constraints $I_i$ and $E_i$, respectively. In Line 11, these sets are given as arguments to FRAGMENT and the top-ranked DQF $F_i$ that satisfies $I_i$ and $E_i$ is returned. In Line 13, $\langle I_i, E_i, F_i\rangle$ is added to *Queue*, provided that $F_i \ne \perp$. Note that $F_1, \ldots, F_k$ are all different from $F$, since they satisfy constraints that $F$ itself does not satisfy. The DQF $F$ is printed at the end of the iteration, in Line 14.

DQFSEARCH$(G, Q)$ enumerates all DQFs of $G$. The delay and the order of the enumeration are determined by the implementation of FRAGMENT. Lawler's procedure [19] was devised for the purpose of finding the top-$k$ answers. The following theorem shows that our adaptation of that procedure indeed enumerates in ranked order provided that FRAGMENT returns optimal answers. Interestingly, DQFSEARCH$(G, Q)$ can also enumerate in a $\theta$-approximate order if FRAGMENT returns $\theta$-approximations. The theorem also specifies the delay in terms of the running time of FRAGMENT. Recall that $n$ and $e$ are the number of nodes and edges of $G$, respectively, $q$ is the number of keywords in $Q$ and $n_i$ is the number of nodes in the $i$th generated DQF. Note that there are at most $2^e$ DQFs, i.e., $i \le 2^e$.

```
Algorithm:  DQFSearch
Input:      A data graph $G$, a query $Q$
Output:     All DQFs

 1: $Queue \leftarrow$ an empty priority queue
 2: $F \leftarrow$ Fragment$(G, Q, \emptyset, \emptyset)$
 3: if $F \ne \perp$ then
 4:    $Queue$.insert$(\langle\emptyset, \emptyset, F\rangle)$
 5: while $Queue \ne \emptyset$ do
 6:    $\langle I, E, F\rangle \leftarrow Queue$.removeTop()
 7:    let $\{e_1, \ldots, e_k\} = \mathcal{E}(F) \setminus I$
 8:    for $i = 1$ to $k$ do
 9:       $I_i \leftarrow I \cup \{e_1, \ldots, e_{i-1}\}$
10:       $E_i \leftarrow E \cup \{e_i\}$
11:       $F_i \leftarrow$ Fragment$(G, Q, I_i, E_i)$
12:       if $F_i \ne \perp$ then
13:          $Queue$.insert$(\langle I_i, E_i, F_i\rangle)$
14:    print$(F)$
```

Figure 3: Basic algorithm for enumerating DQFs

THEOREM 4.1. *Consider a data graph $G$ and a query $Q$.*

- *The algorithm* DQFSEARCH$(G, Q)$ *enumerates all the DQFs of $G$ in ranked order if* FRAGMENT$(G, Q, I, E)$ *returns a minimal DQF that satisfies $I$ and $E$.*
- *If the subroutine* FRAGMENT$(G, Q, I, E)$ *returns a $\theta$-approximation of the minimal DQF that satisfies $I$ and $E$, then the algorithm* DQFSEARCH$(G, Q)$ *enumerates in a $\theta$-approximate ranked order.*
- *If the subroutine* FRAGMENT$(G, Q, I, E)$ *terminates in $T(n, e, q)$ time, then* DQFSEARCH$(G, Q)$ *enumerates with $\mathcal{O}\left(n_i \left(T(n, e, q) + \log(n \cdot i) + n_i\right)\right)$ delay.*

### 4.1 Partial Fragments

The task at hand is to efficiently implement FRAGMENT. Handling exclusion constraints is easy and can be done by deleting the edges of $E$ from the data graph $G$; hence, we ignore these constraints in the rest of the paper. But handling inclusion constraints is generally intractable. It can be shown that it is NP-complete to decide whether there is any DQF (regardless of its rank) that satisfies a single inclusion constraint, even if the query $Q$ has only two keywords.

We circumvent this difficulty by modifying the algorithm DQFSEARCH so that it only generates inclusion constraints of a special form that is defined as follows. Consider a data graph $G$ and a query $Q$. A *partial fragment* (PF) is any directed subtree of $G$. A set of PFs $\mathcal{P}$ is called a *set of PF constraints* if the PFs in $\mathcal{P}$ are pairwise node disjoint. If a subtree $T$ of $G$ includes all the PFs of $\mathcal{P}$, then $T$ is actually

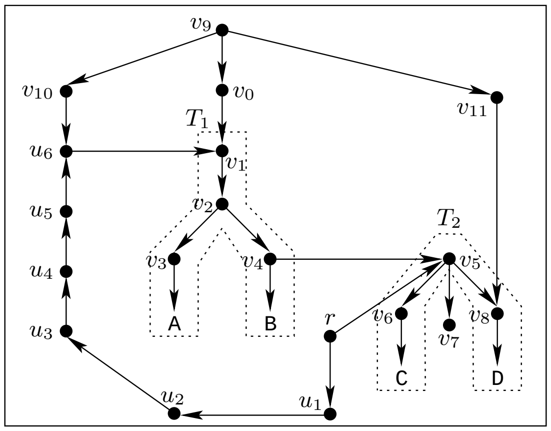

Figure 4: Data graph $G_2$

a *G-supertree* of $\mathcal{P}$ and we also say that $T$ *satisfies* $\mathcal{P}$. We use *leaves*$(\mathcal{P})$ to denote the set of leaves in the PFs of $\mathcal{P}$.

Proposition 4.1. *The algorithm* DQFSearch *can be executed so that for every generated set of inclusion constraints $I$, the subgraph of $G$ induced by the edges of $I$ forms a set of PF constraints $\mathcal{P}$, such that leaves$(\mathcal{P}) \subseteq Q$ and $\mathcal{P}$ has at most two PFs.*

There are two types of PFs. A *reduced PF* (RPF) has a root with at least two children whereas a *nonreduced PF* (NPF) has a root with only one child. We emphasize that, as a special case, a single node is considered an RPF. Note that an RPF, but not an NPF, is reduced w.r.t. the set of its leaves. We use $\mathcal{P}_{|\text{RPF}}$ and $\mathcal{P}_{|\text{NPF}}$ to denote the set of all the RPFs of $\mathcal{P}$ and the set of all the NPFs of $\mathcal{P}$, respectively.

By Proposition 4.1, the subroutine Fragment should find a DQF that satisfies a given set $\mathcal{P}$ of PF constraints, where *leaves*$(\mathcal{P})$ is a subset of $Q$. Without loss of generality, we can add to $\mathcal{P}$ every keyword in $Q \setminus$ *leaves*$(\mathcal{P})$ as a single-node RPF. Thus, from now on, when looking for a DQF that satisfies a set $\mathcal{P}$ of PF constraints, we assume that *leaves*$(\mathcal{P}) = Q$, where $Q$ is the given query. Note that by Proposition 4.1, $\mathcal{P}$ has at most two NPFs.

Suppose that $T$ is a $G$-supertree of $\mathcal{P}$. Note that $T$ is not necessarily a DQF, even if *leaves*$(T) =$ *leaves*$(\mathcal{P}) = Q$, since the root of $T$ may have a single child. Furthermore, if the root belongs to some NPF of $\mathcal{P}$, then it cannot be deleted in order to obtain a DQF. As an example, Figure 4 shows the data graph $G_2$ and two PFs, $T_1$ and $T_2$, that are surrounded by dotted polygons. Let $Q$ be the query $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$ and $\mathcal{P}$ be the set of PF constraints $\{T_1, T_2\}$. The minimal $G_2$-supertree of $\mathcal{P}$ comprises the nodes and edges of $T_1$ and $T_2$ as well as the edge $(v_4, v_5)$. Note, however, that this supertree is not a DQF, since its root $v_1$ has only one child. It can be shown that the only DQF that satisfies $\mathcal{P}$ is obtained by choosing $r$ to be the root, connecting $r$ to $T_1$ using the path $u_1 \to \cdots \to u_6 \to v_1$ and connecting $r$ to $T_2$ with the edge $(r, v_5)$. As another example, suppose that $T_3$ is the subtree of $G_2$ that is obtained by adding the edge $(v_0, v_1)$ to $T_1$ and let $\mathcal{P}' = \{T_2, T_3\}$. Although there is a $G_2$-supertree of $\mathcal{P}'$, no DQF of $G$ satisfies $\mathcal{P}'$.

Nevertheless, our algorithms for finding DQFs under PF constraints require finding minimal supertrees and approximations thereof. In the next section, we show how it is done.

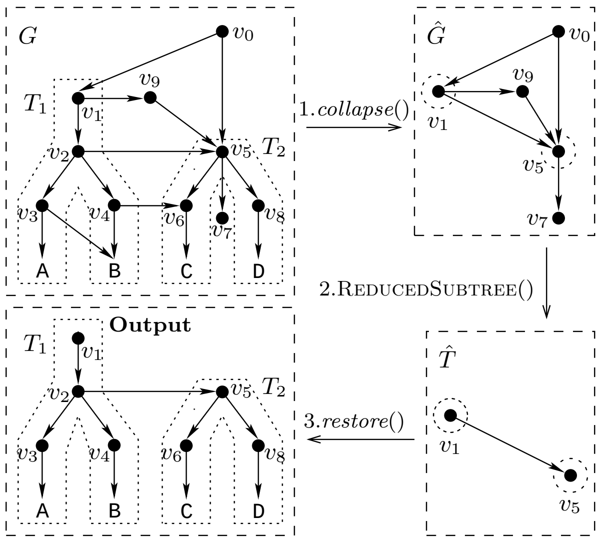

Figure 5: Execution example of Supertree

## 5. COMPUTING SUPERTREES

In this section, we consider a graph $G$ and a set $\mathcal{T}$ of PF constraints. Recall that $\mathcal{T}$ consists of subtrees of $G$ that are pairwise node disjoint. We give algorithms for finding a minimal $G$-supertree of $\mathcal{T}$ and an approximation thereof. Both algorithms are based on an algorithm for finding a *reduced G-supertree* $T$ of $\mathcal{T}$, namely, $T$ is a subtree of $G$ that includes all the PFs of $\mathcal{T}$ and has no proper subtree that also includes these PFs. It should be emphasized that the algorithms of this section are applied even when the set $\mathcal{T}$ of PF constraints does not necessarily correspond to a query, that is, the leaves of $\mathcal{T}$ need not be keywords.

Finding a reduced $G$-supertree of $\mathcal{T}$ is a three-step process. First, $G$ is transformed into a new graph $\hat{G}$ by collapsing each PF of $\mathcal{T}$ into a single node. Second, in the new graph $\hat{G}$, we find a reduced subtree $\hat{T}$ w.r.t. the nodes that correspond to the PFs of $\mathcal{T}$. Third, the PFs of $\mathcal{T}$ are restored in $\hat{T}$. We start by describing the operations *collapse* and *restore*.

### 5.1 Collapsing and Restoring Subtrees

Consider a data graph $G$ and a set $\mathcal{T}$ of PF constraints. We delete from $G$ all edges $e$, such that $e$ enters a non-root node of some PF $T \in \mathcal{T}$ and $e$ is not an edge of $T$. The reason for deleting these edges is that they cannot be included in any $G$-supertree of $\mathcal{T}$. Thus, from now on we assume that if an edge of $G$ enters a non-root node of some $T \in \mathcal{T}$, then that edge belongs to $T$.

The graph *collapse*$(G, \mathcal{T})$ is the result of collapsing all the subtrees of $\mathcal{T}$ in $G$ and it is obtained as follows.

- Delete[2] from $G$ all the non-root nodes of PFs of $\mathcal{T}$. (The roots of all the PFs of $\mathcal{T}$ remain in the collapsed graph, because these PFs are pairwise node disjoint.)

- For all deleted edges $(u, v)$, such that $u$ is a non-root node of a PF $T \in \mathcal{T}$ but $(u, v)$ is not an edge of $T$, add the edge $(\mathit{root}(T), v)$ to *collapse*$(G, \mathcal{T})$. The weight of the edge $(\mathit{root}(T), v)$ is the minimum among the weights of all the edges of $G$ from nodes of $T$ to $v$. In particular, if $(\mathit{root}(T), v)$ is already an edge of $G$, then

[2]When deleting a node, its incident edges are also removed.

```
Algorithm:  SUPERTREE
Input:      A data graph $G$, a set of PF constraints $\mathcal{T}$
Output:     A reduced $G$-supertree of $\mathcal{T}$

1: $\hat{G} \leftarrow collapse(G, \mathcal{T})$
2: $R \leftarrow \{root(T) \mid T \in \mathcal{T}\}$
3: if $\exists v \in \mathcal{V}(\hat{G})$ s.t. each $r \in R$ is reachable from $v$ then
4:     $\hat{T} \leftarrow$ REDUCEDSUBTREE$(\hat{G}, R)$
5:     return $restore_G(\hat{T}, \mathcal{T})$
6: else
7:     return $\perp$
```

Figure 6: Finding a reduced $G$-supertree of $\mathcal{T}$

its weight is decreased if there is another edge with a smaller weight from a non-root node of $T$ to $v$.

As an example, the top part of Figure 5 shows how two node-disjoint subtrees $T_1$ and $T_2$ are collapsed. These subtrees are surrounded by dotted polygons. In this figure, weights of edges are not specified and we assume that they are equal. (Note that the edges $(v_3, \texttt{B})$ and $(v_4, v_6)$ are actually deleted before collapsing $\mathcal{T}$, due to the assumption stated above.)

Next, we describe the inverse operation of collapse, namely, restoring the subtrees of $\mathcal{T}$ in a subgraph $\hat{H}$ of $collapse(G, \mathcal{T})$. The data graph $restore_G(\hat{H}, \mathcal{T})$ is defined as follows.

- Add all the edges and nodes of $\mathcal{T}$ to $\hat{H}$.
- Consider each PF $T$ of $\mathcal{T}$. For each edge $(root(T), v)$ of $\hat{H}$, replace this edge with $(t, v)$, where $(t, v)$ has the minimal weight among all edges of $G$ from nodes of $T$ to $v$. If there are several such nodes $t$, an arbitrary one is chosen. Note that according to this definition, the edge $(root(T), v)$ is left unchanged if $t = root(T)$.

As an example, the bottom part of Figure 5 shows how the collapsed subtrees $T_1$ and $T_2$ are restored in $\hat{T}$.

## 5.2 Approximate and Minimal Supertrees

Consider a data graph $G$ and a set $\mathcal{T}$ of PF constraints. A reduced $G$-supertree of $\mathcal{T}$ can be obtained as follows. First, collapse all the subtrees of $\mathcal{T}$. Note that only the roots of the PFs of $\mathcal{T}$ remain in the resulting graph $\hat{G}$. Next, find a subtree $\hat{T}$ of $\hat{G}$ that is reduced w.r.t. the set $R$ comprising the roots of the PFs of $\mathcal{T}$. Finally, restore the subtrees of $\mathcal{T}$ in $\hat{T}$ to obtain a reduced $G$-supertree of $\mathcal{T}$.

The algorithm SUPERTREE of Figure 6 formally describes the above process. Note that in Line 3, a simple test is performed in order to determine whether $\hat{G}$ has a reduced subtree $\hat{T}$ w.r.t. $R$. If the test is positive, $\hat{T}$ is obtained by executing the procedure REDUCEDSUBTREE$(\hat{G}, R)$ in Line 4.

Figure 5 depicts the graph $G$ and the subtrees $T_1$ and $T_2$. This figure shows an execution of SUPERTREE for the input consisting of $G$ and $\mathcal{T} = \{T_1, T_2\}$. In the first step, $\hat{G}$ is obtained from $G$ by collapsing the subtrees $T_1$ and $T_2$. The second step constructs the subtree $\hat{T}$ of $\hat{G}$ that is reduced w.r.t. the set of roots $\{v_1, v_5\}$. Finally, in the third step, $T_1$ and $T_2$ are restored in $\hat{T}$ and the result is returned.

The proof of correctness of SUPERTREE is omitted, due to a lack of space. Essentially, it is based on the following observation. If $\hat{T}$ is a reduced subtree of $\hat{G}$ w.r.t. $R$, then restoring the subtrees of $\mathcal{T}$ in $\hat{T}$ yields a reduced $G$-supertree of $\mathcal{T}$. Actually, we can also show the following. If $\hat{T}$ is a $\theta$-approximation of the directed Steiner tree for $R$ in $\hat{G}$, then $restore_G(\hat{T}, \mathcal{T})$ is a $\theta$-approximation of the minimal $G$-supertree of $\mathcal{T}$. In particular, if $\hat{T}$ is a directed Steiner tree for $R$ in $\hat{G}$, then $restore_G(\hat{T}, \mathcal{T})$ is a minimal $G$-supertree of $\mathcal{T}$. Thus, we use the algorithm SUPERTREE of Figure 6 as the basis for creating two new algorithms that use existing algorithms from the literature as described below.

We assume that APPROX-DIRECTEDSTEINER$(\hat{G}, R)$ is an existing algorithm that accepts as input a data graph $\hat{G}$, with $\hat{n}$ nodes and $\hat{e}$ edges, and a set $R$ of $t$ nodes of $\hat{G}$. It returns a $\theta(\hat{n}, \hat{e}, t)$-approximation of a directed Steiner tree for $R$ in $G$ and its running time is $f(\hat{n}, \hat{e}, t)$.[3] As discussed in Section 3, we assume that both $\theta$ and $f$ are monotonically non-decreasing functions. The second existing algorithm, DIRECTEDSTEINER$(\hat{G}, R)$, returns a directed Steiner tree for $R$ in $G$ and its running time is $\mathcal{O}(3^t \hat{n} + 2^t \hat{e} \log \hat{n})$.[4]

The algorithm APPROX-SUPERTREE$(G, \mathcal{T})$ is obtained by using APPROX-DIRECTEDSTEINER$(\hat{G}, R)$, in Line 4 of Figure 6, instead of REDUCEDSUBTREE$(\hat{G}, R)$. The following theorem shows that APPROX-SUPERTREE$(G, \mathcal{T})$ returns a $\theta(n, e, t)$-approximation of the minimal $G$-supertree of $\mathcal{T}$.

THEOREM 5.1. *Consider a data graph $G$ and a set $\mathcal{T}$ of PF constraints. Suppose that $G$ has $n$ nodes and $e$ edges, and let $\mathcal{T}$ have $t$ PFs.* APPROX-SUPERTREE$(G, \mathcal{T})$ *returns a $G$-supertree that is reduced w.r.t. $\mathcal{T}$ and is a $\theta(n, e, t)$-approximation of the minimal $G$-supertree of $\mathcal{T}$. If no $G$-supertree of $\mathcal{T}$ exists, then $\perp$ is returned. The running time is $\mathcal{O}(te + f(n, e, t))$.*

The algorithm MIN-SUPERTREE$(G, \mathcal{T})$ is obtained by using DIRECTEDSTEINER$(\hat{G}, R)$, in Line 4 of Figure 6, instead of REDUCEDSUBTREE$(\hat{G}, R)$. The following is a corollary of Theorem 5.1, since the algorithm DIRECTEDSTEINER$(\hat{G}, R)$ returns a 1-approximation.

COROLLARY 5.1. *Consider a data graph $G$ and a set $\mathcal{T}$ of PF constraints. Let $G$ have $n$ nodes and $e$ edges, and let $\mathcal{T}$ have $t$ PFs.* MIN-SUPERTREE$(G, \mathcal{T})$ *returns a minimal $G$-supertree of $\mathcal{T}$ if one exists, or $\perp$ otherwise. The running time is $\mathcal{O}(3^t n + 2^t e \log n)$.*

## 6. MINIMAL CONSTRAINED ANSWERS

In this section, we show how to find efficiently a minimal DQF that satisfies a set of PF constraints, assuming that the size of the query $Q$ is bounded (otherwise, the problem is intractable). The algorithm MIN-FRAGMENT$(G, \mathcal{P})$ of Figure 7 accepts as input a data graph $G$ and a set $\mathcal{P}$ of PF constraints, such that $leaves(\mathcal{P}) = Q$. It returns a minimal DQF that satisfies $\mathcal{P}$. If MIN-FRAGMENT is used instead of FRAGMENT in Figure 3, then DQFSEARCH enumerates all the DQFs in ranked order with polynomial delay. (Recall that the arguments of FRAGMENT are converted to those of MIN-FRAGMENT as described in Section 4.1.)

We use the following notation. For a data graph $G$ and a subset $U \subseteq \mathcal{V}(G)$, we denote by $G - U$ the induced subgraph of $G$ that consists of the nodes of $\mathcal{V}(G) \setminus U$ and all the edges

[3]We assume that, if necessary, the approximation algorithm is modified so that it returns a reduced subtree w.r.t. $R$, which can be done in linear time.

[4]The specified running time can be obtained by implementing DIRECTEDSTEINER as described in [15].

```
Algorithm:  MIN-FRAGMENT
Input:      A data graph $G$, a set of PF constraints $\mathcal{P}$, such that $leaves(\mathcal{P}) = Q$
Output:     A minimal DQF that satisfies $\mathcal{P}$, or $\perp$ if none exists

1: if $\mathcal{P}$ consists of a single RPF $T$ then
2:     return $T$
3: if $\mathcal{P}$ consists of a single NPF then
4:     return $\perp$
5: $F_{\min} \leftarrow \perp$    /* — note that the weight of $\perp$ is $\infty$ — */
6: let $\mathcal{T}$ be obtained from $\mathcal{P}$ by replacing each $N \in \mathcal{P}_{|\mathrm{NPF}}$ with the PF consisting of $root(N)$
7: for all subsets $\mathcal{T}_{\mathrm{top}}$ of $\mathcal{T}$, such that $|\mathcal{T}_{\mathrm{top}}| \geq 2$ do
       /* — Construct the supertree $T_{top}$ — */
8:     $G_1 \leftarrow G - \cup_{N \in \mathcal{P}_{|\mathrm{NPF}}} (\mathcal{V}(N) \setminus \{root(N)\}) - \cup_{T \in \mathcal{T} \setminus \mathcal{T}_{\mathrm{top}}} \mathcal{V}(T)$
9:     $T_{\mathrm{top}} \leftarrow$ MIN-SUPERTREE$(G_1, \mathcal{T}_{\mathrm{top}})$
10:    if $T_{\mathrm{top}} \neq \perp$ then
           /* — Construct the DQF $F$ — */
11:        let $T^{+}_{\mathrm{top}}$ be the union of $T_{\mathrm{top}}$ and all $N \in \mathcal{P}_{|\mathrm{NPF}}$, such that $root(N) \in \mathcal{V}(T_{\mathrm{top}})$
12:        let $\mathcal{P}'$ be the set of all the PFs of $\mathcal{P}$ that are not contained in $T^{+}_{\mathrm{top}}$
13:        let $G_2$ be obtained from $G$ by removing all the incoming edges of $root(T^{+}_{\mathrm{top}})$
14:        $F \leftarrow$ MIN-SUPERTREE$(G_2, \mathcal{P}' \cup \{T^{+}_{\mathrm{top}}\})$
           /* — Update $F_{min}$ — */
15:        $F_{\min} \leftarrow \min\{F_{\min}, F\}$
16: return $F_{\min}$
```

Figure 7: Finding a minimal DQF under a set of PF constraints

of $G$ between these nodes. If $G_1$ and $G_2$ are subgraphs of $G$, then $G_1 \cup G_2$ denotes their *union*, namely, the subgraph $G'$ that consists of all the nodes and edges that appear in either $G_1$ or $G_2$; in other words, $G'$ satisfies $\mathcal{V}(G') = \mathcal{V}(G_1) \cup \mathcal{V}(G_2)$ and $\mathcal{E}(G') = \mathcal{E}(G_1) \cup \mathcal{E}(G_2)$.

Our goal is to find a minimal DQF that satisfies the given set $\mathcal{P}$ of PF constraints. Recall that we cannot simply apply the algorithm MIN-SUPERTREE of Section 5.2, unless all the PFs of $\mathcal{P}$ are actually RPFs. As discussed at the end of Section 4.1, even if we find a minimal $G$-supertree $T$ of $\mathcal{P}$, a minimal DQF that satisfies $\mathcal{P}$ may be very different from $T$ or may not exist at all. Consequently, the algorithm of Figure 7 is more intricate than MIN-SUPERTREE.

The main idea of the algorithm is to convert $\mathcal{P}$ into an equivalent set $\mathcal{P}' \cup \{T^{+}_{\mathrm{top}}\}$, such that $T^{+}_{\mathrm{top}}$ is an RPF with the same root as some minimal DQF that satisfies $\mathcal{P}$. We then compute, in Line 14 of Figure 7, a minimal $G_2$-supertree of $\mathcal{P}' \cup \{T^{+}_{\mathrm{top}}\}$, where $G_2$ is obtained from $G$ by deleting the edges that enter into $root(T^{+}_{\mathrm{top}})$ (thereby guaranteeing that $root(T^{+}_{\mathrm{top}})$ is the root of the computed supertree).

In order to understand how we compute $T^{+}_{\mathrm{top}}$, consider a minimal DQF $F'$ that satisfies $\mathcal{P}$. Suppose that for all NPFs $N \in \mathcal{P}$, we delete from $F'$ all the nodes of $N$ except for $root(N)$. Next, we delete all the nodes that are not reachable from $root(F')$. We define the following.

- $T'_{\mathrm{top}}$ is the subtree of $F'$ that remains after the above deletions.
- $T'^{+}_{\mathrm{top}}$ is the subtree of $F'$ that is the union of $T'_{\mathrm{top}}$ and all the NPFs $N \in \mathcal{P}$, such that $root(N)$ is in $T'_{\mathrm{top}}$.
- $\mathcal{T}'_{\mathrm{top}}$ consists of all PFs $T'$, such that $T'$ is either
  - an RPF of $\mathcal{P}$ that is included in $T'^{+}_{\mathrm{top}}$, or
  - $root(N)$, where $N$ is an NPF of $\mathcal{P}$ that is included in $T'^{+}_{\mathrm{top}}$.
- The subgraph $G'_1$ is obtained from $G$ by deleting all nodes $n$, such that $n$ appears in a PF of $\mathcal{P}$ but not in any PF of $\mathcal{T}'_{\mathrm{top}}$.

The following can be shown. First, $root(T'^{+}_{\mathrm{top}})$ has at least two children. Second, $\mathcal{T}'_{\mathrm{top}}$ has at least two PFs. Third, $T'_{\mathrm{top}}$ is a minimal $G'_1$-supertree of $\mathcal{T}'_{\mathrm{top}}$. Fourth, if we replace the subtree $T'_{\mathrm{top}}$ of $F'$ with any minimal $G'_1$-supertree of $\mathcal{T}'_{\mathrm{top}}$, the result remains a minimal DQF that satisfies $\mathcal{P}$.

Thus, we can compute $T'_{\mathrm{top}}$ if we know the set $\mathcal{T}'_{\mathrm{top}}$, which is not the case. In the algorithm MIN-FRAGMENT of Figure 7, the main loop (Line 7) iterates over all subsets of $\mathcal{T}$ that could be the value of $\mathcal{T}'_{\mathrm{top}}$; in each iteration, the variable $\mathcal{T}_{\mathrm{top}}$ is assigned one possible subset. (The number of these subsets is bounded by a constant assuming that the size of the query is fixed.) Note that $\mathcal{T}$ is obtained from $\mathcal{P}$, in Line 6, by replacing each NPF of $\mathcal{P}$ with its root. In Line 8, $G_1$ is obtained from $G$ by deleting all the nodes of $\mathcal{P}$ that do not appear in $\mathcal{T}_{\mathrm{top}}$. In Line 9, MIN-SUPERTREE finds a minimal $G_1$-supertree $T_{\mathrm{top}}$ that satisfies $\mathcal{T}_{\mathrm{top}}$. It is important to compute the supertree $T_{\mathrm{top}}$ in the graph $G_1$, rather than the original graph $G$, because $T_{\mathrm{top}}$ cannot use edges and nodes that appear in $\mathcal{P}$ but not in $\mathcal{T}_{\mathrm{top}}$.

Note that there could be many minimal $G_1$-supertrees of $\mathcal{T}_{\mathrm{top}}$, but Line 9 computes just one such supertree, namely, $T_{\mathrm{top}}$. In Line 11, $T_{\mathrm{top}}$ is expanded to $T^{+}_{\mathrm{top}}$ by adding the nodes and edges of all NPFs $N \in \mathcal{P}$, such that $root(N)$ is already in $T_{\mathrm{top}}$. Note that $T^{+}_{\mathrm{top}}$ is an RPF.

We are now looking for a DQF $F$ that has the same root as the RPF $T^{+}_{\mathrm{top}}$. Note that $F$ must satisfy $T^{+}_{\mathrm{top}}$ as well as all the PFs of $\mathcal{P}$ that are not contained in $T^{+}_{\mathrm{top}}$. Thus, the set $\mathcal{P}'$ is constructed in Line 12. To guarantee that $F$ has the same root as $T^+_{\text{top}}$, Line 13 removes from $G$ all the incoming edges of this root and the result is $G_2$. Line 14 finds a minimal $G_2$-supertree $F$ that satisfies $\mathcal{P}' \cup \{T^+_{\text{top}}\}$. The algorithm Min-Fragment returns the minimal $F$ among all those found in the loop of Line 7.

The following theorem uses the running time of Min-Supertree from Corollary 5.1. Note that Theorem 3.1 follows as a direct corollary, since $p \leq q$.

Theorem 6.1. *Consider a data graph $G$ with $n$ nodes and $e$ edges. Let $Q$ be a query with $q$ keywords and $\mathcal{P}$ be a set of $p$ PF constraints, such that leaves$(\mathcal{P}) = Q$.* Min-Fragment$(G, \mathcal{P})$ *returns either a minimal DQF that satisfies $\mathcal{P}$, or $\perp$ if there is no such DQF. The running time is $\mathcal{O}\left(4^p n + 3^p e \log n\right)$.*

Proof. Suppose that $G$ has a minimal DQF $F'$ that satisfies $\mathcal{P}$. We will prove that the algorithm returns a DQF that satisfies $\mathcal{P}$ and has a weight that is not larger than that of $F'$. As defined above, $F'$ has a subtree $T'_{\text{top}}$ that satisfies the PFs of $\mathcal{T}'_{\text{top}}$. Consider the iteration of the loop of Line 7 in which $\mathcal{T}_{\text{top}}$ is assigned the set $\mathcal{T}'_{\text{top}}$. Let $T_{\text{top}}$ be the minimal $G_1$-supertree of $\mathcal{T}_{\text{top}}$ that is returned in Line 9 during that iteration. Observe that $T_{\text{top}}$ and $T'_{\text{top}}$ are not necessarily the same, but the weight of $T_{\text{top}}$ is not larger than that of $T'_{\text{top}}$.

Since $T_{\text{top}}$ is a $G_1$-supertree of $\mathcal{T}_{\text{top}}$, its root is either a root of some PF of $\mathcal{T}_{\text{top}}$ or a node that does not appear in $\mathcal{P}$. Furthermore, $T_{\text{top}}$ has none of the edges of the NPFs of $\mathcal{P}$, because $G_1$ has none of these edges. Thus, when $T_{\text{top}}$ is expanded to $T^+_{\text{top}}$, the root must have at least two children.

It remains to show that Line 14 computes a DQF that satisfies $\mathcal{P}$ and has a weight that is not larger than that of $F'$. Consider the graph $H$ that is obtained from $T^+_{\text{top}}$ by adding all the edges of $F'$ that are not in $T'^+_{\text{top}}$, i.e., the edges of the set $\mathcal{E}(F') \setminus \mathcal{E}(T'^+_{\text{top}})$. The graph $H$ has the following properties. First, the weight of $H$ is not larger than that of $F'$. Second, $H$ includes all the PFs of $\mathcal{P}$. Third, the root of every PF of $\mathcal{P}$ is reachable in $H$ from the root of $T^+_{\text{top}}$. Fourth, edges of $H$ that enter non-root nodes of PFs of $\mathcal{P}$ must belong to $\mathcal{P}$. Note that these properties continue to hold even if we remove from $H$ all the edges that enter the root of $T^+_{\text{top}}$ (if $H$ indeed has such edges). It thus follows that Line 14 computes a subtree that satisfies all the PFs of $\mathcal{P}$, has a root with at least two children and its weight is not larger than that of $F'$ (because $H$ has such a subtree).

We have shown that the algorithm returns a minimal DQF that satisfies $\mathcal{P}$ if indeed there is such a DQF. The above proof also indicates that if the output is not $\perp$, then a DQF that satisfies $\mathcal{P}$ is returned. Thus, the algorithm is correct.

The running time of Min-Fragment$(G, \mathcal{P})$ is dominated by the two calls to Min-Supertree in Lines 9 and 14. For each subset $\mathcal{T}_{\text{top}} \subseteq \mathcal{T}$ of size $k$, Min-Supertree is called once with the set $\mathcal{T}_{\text{top}}$ and once with the set $\mathcal{P}' \cup \{T^+_{\text{top}}\}$. The number of PFs in $\mathcal{P}' \cup \{T^+_{\text{top}}\}$ is at most $p - k + 1$. By Corollary 5.1, an upper bound on the total cost of Lines 9 and 14 is $\mathcal{O}(g(n, e, p))$, where

$$g(n,e,p) = \sum_{k=2}^{p} \binom{p}{k} \left(3^k n + 2^k e \log n + 3^{p-k+1} n + 2^{p-k+1} e \log n\right)$$
$$\leq 4n \sum_{k=0}^{p} \binom{p}{k} 3^k + 3e \log n \sum_{k=0}^{p} \binom{p}{k} 2^k = 4n4^p + 3e(\log n)3^p.$$

The specified runtime then follows immediately. □

## 7. MINIMAL-ANSWER APPROXIMATION

In this section, we show how to find an approximation of a minimal DQF under a set of PF constraints. Note that if, in the algorithm Min-Fragment of Figure 7, we simply replaced Min-Supertree with the algorithm Approx-Supertree of Section 5.2, the running time would still be exponential in the number of PFs of $\mathcal{P}$. Thus, we use the algorithm Approx-Fragment of Figure 8 that accepts as input the data graph $G$ and a set of PF constraints $\mathcal{P}$, such that *leaves*$(\mathcal{P}) = Q$. This algorithm finds a $(\theta + 1)$-approximation of a minimal DQF that satisfies $\mathcal{P}$, given that Approx-DirectedSteiner finds a $\theta$-approximation of a directed Steiner tree. By Proposition 4.1, we can assume that $\mathcal{P}$ has at most two NPFs. This fact is important in showing that Approx-Fragment has a polynomial runtime even if the size of the query $Q$ is unbounded.

Approx-Fragment starts by checking whether $\mathcal{P}$ has no RPFs and, if so, Min-Fragment is applied to the input. Otherwise, a $G$-supertree $T_{\text{app}}$ that satisfies $\mathcal{P}$ is constructed by the algorithm Approx-Supertree. If $T_{\text{app}}$ is either a DQF or $\perp$, then it is returned by the algorithm. If not, the root of $T_{\text{app}}$ has only one child (and, hence, *root*$(T_{\text{app}})$ must also be the root of some NPF of $\mathcal{P}$). We fix this problem by modifying $T_{\text{app}}$ at the cost of increasing the approximation ratio to $\theta + 1$. This is done as follows.

In the loop of Line 7, $G'$ is obtained from $G$ by deleting all edges $e$, such that $e$ is not in $\mathcal{P}$ and $e$ enters a non-root node of some PF of $\mathcal{P}$ (note that the deleted edges cannot be included in any subtree of $G$ that satisfies $\mathcal{P}$). The loop of Line 10 iterates over all RPFs $T_{\text{r}} \in \mathcal{P}$ and applies Min-Fragment to $G'$ and $\mathcal{P}_{|\text{NPF}} \cup \{T_{\text{r}}\}$. The following can be shown. The minimal subtree $T_{\min}$ among all those returned by Min-Fragment, during the iterations of the loop, has a root with at least two children and a weight that is not larger than that of the minimal DQF $F$ that satisfies $\mathcal{P}$. Thus, the weight of $T_{\text{app}} \cup T_{\min}$ is at most $\theta + 1$ times the weight of $F$. Note that $T_{\text{app}} \cup T_{\min}$ is not necessarily a tree (it might even contain directed cycles). However, it can be shown that $T_{\text{app}} \cup T_{\min}$ contains a DQF that satisfies $\mathcal{P}$.

The algorithm returns a DQF $F$ that is obtained from $G_{\text{app}} = T_{\text{app}} \cup T_{\min}$ as follows. Recall that the root of $T_{\text{app}}$ is reachable from the root of $T_{\min}$, since *root*$(T_{\text{app}})$ is also the root of some NPF of $\mathcal{P}$ (or else the algorithm terminates in Line 5). Therefore, *root*$(T_{\min})$ becomes the root of $F$ unless *root*$(T_{\min})$ is in some RPF $T \in \mathcal{P}$ and if so, *root*$(T)$ becomes the root of $F$. Lines 19–22 remove all redundant nodes and edges of $G_{\text{app}}$. The loop of Lines 20–21 iterates over all nodes $v$ of $G_{\text{app}}$ that have more than one incoming edge. Note that such a node $v$ has exactly two incoming edges: one belongs to $T_{\text{app}}$ and the other is from $T_{\min}$. We remove the first and leave the second. Line 22 deletes all structural nodes that have no descendant keywords.

Interestingly, the loop of Line 10 can be replaced with a single invocation of Min-Fragment by applying the following transformation to $G'$. First, collapse each RPF of $\mathcal{P}$ into a single node. Second, add a new keyword $v$ and edges from the collapsed RPFs to $v$. Min-Fragment is then called with the set $\mathcal{P}_{|\text{NPF}} \cup \{v\}$.

In the next theorem, the runtime analysis assumes that the above optimization is used. As explained in Section 5.2, Approx-Supertree uses Approx-DirectedSteiner and the latter finds a $\theta$-approximation of a directed Steiner tree. The approximation ratio $\theta$ and the running time $f$ are mono-

```
Algorithm:  Approx-Fragment
Input:      A data graph $G$, a set of PF constraints $\mathcal{P}$, such that $leaves(\mathcal{P}) = Q$
Output:     A $(\theta + 1)$-approximation of the minimal DQF that satisfies $\mathcal{P}$, or $\perp$ if none exists

 1: if $\mathcal{P}_{|\text{RPF}} = \emptyset$ then
 2:    return Min-Fragment$(G, \mathcal{P})$
    /* — Construct $T_{app}$ — */
 3: $T_{\text{app}} \leftarrow$ Approx-Supertree$(G, \mathcal{P})$
 4: if $T_{\text{app}} = \perp$ or $T_{\text{app}}$ is a DQF then
 5:    return $T_{\text{app}}$
    /* — Construct $T_{min}$ — */
 6: $G' \leftarrow G$
 7: for all $T \in \mathcal{P}$ do
 8:    remove from $G'$ all edges $(v, u)$, such that $(v, u) \notin \mathcal{E}(T)$ and $u \in \mathcal{V}(T) \setminus \{root(T)\}$
 9: $T_{\text{min}} \leftarrow \perp$   /* — note that the weight of $\perp$ is $\infty$ — */
10: for all $T_{\text{r}} \in \mathcal{P}_{|\text{RPF}}$ do
11:    $T \leftarrow$ Min-Fragment$(G', \mathcal{P}_{|\text{NPF}} \cup \{T_{\text{r}}\})$
12:    $T_{\text{min}} \leftarrow \min\{T_{\text{min}}, T\}$
13: if $T_{\text{min}} = \perp$ then
14:    return $\perp$
    /* — Remove redundant nodes and edges from $T_{min} \cup T_{app}$ — */
15: $r \leftarrow root(T_{\text{min}})$
16: if $r$ belongs to a subtree $T \in \mathcal{P}_{|\text{RPF}}$ then
17:    $r \leftarrow root(T)$
18: $G_{\text{app}} \leftarrow T_{\text{min}} \cup T_{\text{app}}$
19: remove from $G_{\text{app}}$ all incoming edges of $r$
20: for all nodes $v \in \mathcal{V}(G_{\text{app}})$ that have two incoming edges $e_1 \in \mathcal{E}(T_{\text{min}})$ and $e_2 \in \mathcal{E}(T_{\text{app}})$ do
21:    remove $e_2$ from $G_{\text{app}}$
22: delete from $G_{\text{app}}$ all structural nodes $v$, such that no keyword is reachable from $v$
23: return $G_{\text{app}}$
```

Figure 8: Finding an approximation of the minimal DQF under a set of PF constraints

tonically nondecreasing functions and, hence, $n$, $e$ and $q$ can be used as their arguments. The constant $c$ denotes the number of NPFs in $\mathcal{P}$; recall that $c \leq 2$ whenever Approx-Fragment is invoked in Line 11 of DQFSearch (Figure 3).

Theorem 7.1. *Consider a data graph $G$ with $n$ nodes and $e$ edges. Let $Q$ be a query with $q$ keywords and $\mathcal{P}$ be a set of PF constraints, such that $leaves(\mathcal{P}) = Q$ and $\mathcal{P}$ has at most $c$ NPFs. The algorithm* Approx-Fragment$(G, \mathcal{P})$ *finds a $(\theta+1)$-approximation of the minimal DQF that satisfies $\mathcal{P}$ in $\mathcal{O}(f + 4^c n + 3^c e \log n)$ time, where $\theta$ and $f$ are the approximation ratio and runtime, respectively, of* Approx-DirectedSteiner.

Proof. The running time easily follows from that of Min-Fragment, so we only prove correctness. Suppose that the algorithm is applied to $G$ and $\mathcal{P}$. As discussed above (and using the claim proven below), if the algorithm returns a DQF, then its weight is a $(\theta + 1)$-approximation of a minimal DQF that satisfies $\mathcal{P}$. So, we prove that the algorithm returns a DQF that satisfies $\mathcal{P}$ (if one exists) assuming that Line 6 is reached, since the other cases are easy.

Consider a minimal DQF $F$ of $G$ that satisfies $\mathcal{P}$. We claim that there is an RPF $T_{\text{r}} \in \mathcal{P}$ and a subtree $T$ of $F$, such that $T$ is a reduced supertree of $\mathcal{P}_{|\text{NPF}} \cup \{T_{\text{r}}\}$ and its root has at least two children. Assuming that $F$ exists, this claim implies that Line 15 is reached and the weight of $T_{\text{min}}$ is not larger than that of $F$. To prove the claim, consider a minimal subtree $T'$ of $F$ that has the same root as $F$ and includes all the NPFs of $\mathcal{P}$. We choose the RPF $T_{\text{r}}$ as follows. If $root(F)$ has more than one child in $T'$, then $T_{\text{r}}$ can be any RPF of $\mathcal{P}$. Otherwise, $T_{\text{r}}$ is an RPF that is reachable from $root(F)$ through an edge that is not in $T'$ (note that $T_{\text{r}}$ must exist or else $F$ is not a DQF). Now, we choose $T$ to be the minimal subtree of $F$ that satisfies $\mathcal{P}_{|\text{NPF}} \cup \{T_{\text{r}}\}$.

It remains to show that after executing Line 19–22, $G_{\text{app}}$ is a DQF that satisfies $\mathcal{P}$. Let $G^0_{\text{app}}$ be the initial value that is assigned to $G_{\text{app}}$ in Line 18 and consider the root $r$ that is computed in Lines 15–17. Two properties hold. First, $r$ has two children in $G^0_{\text{app}}$, because $r$ is either the root of an RPF (which is included in $G^0_{\text{app}}$) or the root of $T_{\text{min}}$ (which has two children). Second, all the PFs of $\mathcal{P}$ are reachable from $r$ in $G^0_{\text{app}}$; in proof, the root $r'$ of $T_{\text{app}}$ is also the root of some NPF of $\mathcal{P}$ and, hence, $r'$ is included in $T_{\text{min}}$, which is reachable from $r$.

The first property above clearly continues to hold after deleting (in Line 19) all the incoming edges of $r$. The same is true for the second property, because of the following case analysis. The root of $T_{\text{min}}$ is either a node of an RPF $R \in \mathcal{P}$, the root of an NPF of $\mathcal{P}$ or not in any PF of $\mathcal{P}$. In the first case, $r$ is the root of $R$, whereas in the other two cases, $r$ is the root of $T_{\text{min}}$. Thus, if an edge of $\mathcal{P}$ enters $r$, then $r$ is the root of one PF and also a node of another PF, contradicting the fact that the PFs of $\mathcal{P}$ are node disjoint.

In $G_{\text{app}}$, each non-root node $v$ of a PF $T \in \mathcal{P}$ has only one incoming edge, namely, the edge of $T$ that enters $v$. This follows from the facts that $T_{\text{app}}$ is a subtree that satisfies $\mathcal{P}$ and $T_{\text{min}}$ is a subtree of $G'$. Thus, edges of $\mathcal{P}$ are not removed in the loop of Line 20. Structural nodes of $\mathcal{P}$ are not deleted in Line 22, because $G^0_{\text{app}}$ satisfies $\mathcal{P}$, the edges of $\mathcal{P}$ are not deleted and all the leaves of $\mathcal{P}$ are keywords. If an edge $e$ is removed, then it enters a node $n$ of $T_{\min}$ but the edge itself does not belong to $T_{\min}$.[5] Hence, $n$ is still reachable from $r$ after the removal of $e$. We conclude that the two properties above continue to hold when performing the deletions of edges and nodes in Lines 20–22.

At the end of the algorithm, $G_{\text{app}}$ satisfies the following. First, $r$ is the root of $G_{\text{app}}$, because it has no incoming edges and all the nodes of $G_{\text{app}}$ are reachable from $r$. Second, each non-root node has just one incoming edge. Third, $r$ has at least two children (because if $r$ is the root of $T_{\min}$, then none of the edges of $T_{\min}$ is removed; and if $r$ is the root an RPF $T$, then its outgoing edges in $T$ are not deleted). Fourth, the leaves of $G_{\text{app}}$ are exactly the keywords of $Q$, since keywords are never deleted from $G_{\text{app}}$ and, after Line 22, every structural node has a descendant keyword. Fifth, all the PFs of $\mathcal{P}$ are included in $G_{\text{app}}$. Thus, upon termination, $G_{\text{app}}$ is a DQF that satisfies $\mathcal{P}$. $\square$

## 8. CONCLUSION

The main contribution of this paper is in showing that the answers (i.e., DQFs, UQFs and SQFs) of keyword proximity search can be enumerated in ranked order with polynomial delay, under data complexity. Furthermore, when a faster algorithm is more important than the exact ranked order, it is possible to enumerate in an approximate order with polynomial delay (under query-and-data complexity). It thus follows that the top-$k$ answers as well as approximations thereof (as defined in [7]) can be computed efficiently. In comparison, existing systems [11, 1, 12, 20] use algorithms that have an exponential runtime, in the worst case, even if the size of the query is fixed. Some systems [2, 14, 20] enumerate in an "almost" ranked order, but without proving the existence of any nontrivial approximation ratio (and the delay could still be exponential).

It might be argued that our algorithms cannot be effectively implemented in practical systems, because the Steiner-tree computation must be applied to large graphs. However, our algorithms are efficient reductions in the sense that the Steiner-tree computation dominates the runtime and is applied to graphs that are only slightly different from the original data graph. Any algorithm for enumerating answers, in either the exact or an approximate order, must also solve the corresponding Steiner-tree problem and, so, can be incorporated in our approach. In fact, our reductions show that it is not only necessary but also sufficient to concentrate on developing practically efficient algorithms for finding Steiner trees in large data graphs.

To avoid the "Steiner-tree bottleneck," one might try to develop efficient heuristics for enumerating in an "almost" ranked order, without any provable, nontrivial approximation ratio. The algorithms of [17] could be a good basis for developing such heuristics, since they are more efficient than those presented here (but they enumerate in an arbitrary order). In [17], there is also an algorithm for enumerating in ranked order, but it applies only to DQFs and data graphs without cycles.

We have presented detailed algorithms only for DQFs. In principle, one can reduce the enumeration of UQFs and SQFs to that of DQFs by adding back edges. This approach, however, has two drawbacks. First, it increases the delay (which is still polynomial). Second, it does not use the lower approximation ratios that are realizable for the undirected and group Steiner-tree problems. A better approach is a direct adaptation of the algorithms for DQFs to UQFs and SQFs. In [15], we describe algorithms for undirected and group Steiner trees that are needed for this approach.

We note that our algorithms can handle more general ranking functions than the one defined in this paper. In particular, nodes (and not just edges) can have weights; moreover, the rank of a graph can be defined by any function that has certain properties of monotonicity.

[5]Note that the only case where an edge of $T_{\min}$ might be deleted occurs in Line 19 if $r$ is the root of an RPF and a non-root node of $T_{\min}$.

## 9. REFERENCES

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

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

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks*, 30(1-7), 1998.

[4] M. Charikar, C. Chekuri, T. Y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. In *SODA*, 1998.

[5] S. Dreyfus and R. Wagner. The Steiner problem in graphs. *Networks*, 1, 1972.

[6] D.-Z. Du, J. Smith, and J. Rubinstein. *Advances in Steiner Trees*. Springer, 2000.

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

[8] J. Feldman and M. Ruhl. The directed steiner network problem is tractable for a constant number of terminals. In *FOCS*, 1999.

[9] N. Garg, G. Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group steiner tree problem. *J. Algorithms*, 37(1), 2000.

[10] C. S. Helvig, G. Robins, and A. Zelikovsky. An improved approximation scheme for the group Steiner problem. *Networks*, 37(1), 2001.

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

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

[13] D. Johnson, M. Yannakakis, and C. Papadimitriou. On generating all maximal independent sets. *Information Processing Letters*, 27, 1988.

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

[15] B. Kimelfeld and Y. Sagiv. New algorithms for computing Steiner trees for a fixed number of terminals. To be found in the first author's home page (`http://www.cs.huji.ac.il/~bennyk`).

[16] B. Kimelfeld and Y. Sagiv. Efficient engines for keyword proximity search. In *WebDB*, 2005.

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

[18] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. *Acta Inf.*, 15, 1981.

[19] 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.

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

[21] G. Robins and A. Zelikovsky. Improved Steiner tree approximation in graphs. In *SODA*, 2000.

[22] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, 1982.

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

[24] 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.

[25] A. Zelikovsky. An 11/6-approximation algorithm for the network steiner problem. *Algorithmica*, 9(5), 1993.