A Practically Efficient Algorithm for Generating Answers to Keyword Search Over Data Graphs*†

Konstantin Golenberg[1] and Yehoshua Sagiv[2]

1 **The Hebrew University of Jerusalem, Jerusalem, Israel**
2 **The Hebrew University of Jerusalem, Jerusalem, Israel**

Abstract

In keyword search over a data graph, an answer is a non-redundant subtree that contains all the keywords of the query. A naive approach to producing all the answers by increasing height is to generalize Dijkstra's algorithm to enumerating all acyclic paths by increasing weight. The idea of *freezing* is introduced so that (most) non-shortest paths are generated only if they are actually needed for producing answers. The resulting algorithm for generating subtrees, called *GTF*, is subtle and its proof of correctness is intricate. Extensive experiments show that GTF outperforms existing systems, even ones that for efficiency's sake are incomplete (i.e., cannot produce all the answers). In particular, GTF is scalable and performs well even on large data graphs and when many answers are needed.

1998 ACM Subject Classification H.3.3 [Information Storage and Retrieval] Information Search and Retrieval – Search process, H.2.4 [Database Management] Systems – Query processing

Keywords and phrases Keyword search over data graphs, subtree enumeration by height, top-k answers, efficiency

Digital Object Identifier 10.4230/LIPIcs.ICDT.2016.23

1 Introduction

Keyword search over data graphs is a convenient paradigm of querying semistructured and linked data. Answers, however, are similar to those obtained from a database system, in the sense that they are succinct (rather than just relevant documents) and include semantics (in the form of entities and relationships) and not merely free text. Data graphs can be built from a variety of formats, such as XML, relational databases, RDF and social networks. They can also be obtained from the amalgamation of many heterogeneous sources. When it comes to querying data graphs, keyword search alleviates their lack of coherence and facilitates easy search for precise answers, as if users deal with a traditional database system.

In this paper, we address the issue of efficiency. Computing keyword queries over data graphs is much more involved than evaluation of relational expressions. Quite a few systems have been developed (see [2] for details). However, they fall short of the degree of efficiency and scalability that is required in practice. Some algorithms sacrifice *completeness* for the sake of efficiency; that is, they are not capable of generating all the answers and, consequently, may miss some relevant ones.

We present a novel algorithm, called *Generating Trees with Freezing* (GTF). We start with a straightforward generalization of Dijkstra's shortest-path algorithm to the task of

* This work was supported by the Israel Science Foundation (Grant No. 1632/12).
† The full version of this paper appears in [5].

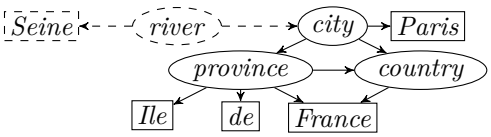

Figure 1 A snippet of a data graph.

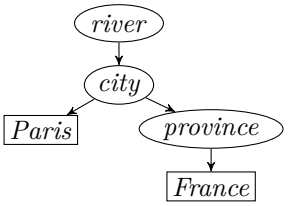

Figure 2 Redundant subtree.

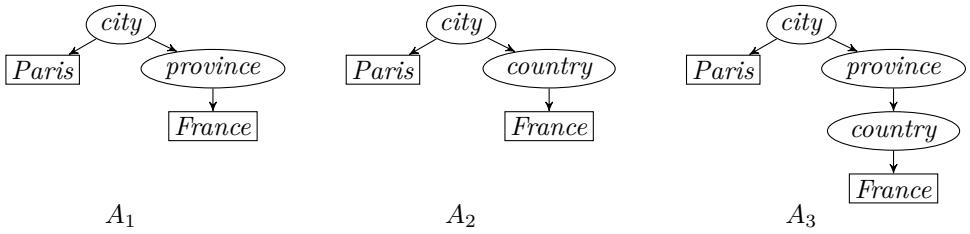

Figure 3 Answers.

constructing all simple (i.e., acyclic) paths, rather than just the shortest ones. Our main contribution is incorporating the *freezing* technique that enhances efficiency by up to one order of magnitude, compared with the naive generalization of Dijkstra's algorithm. The main idea is to avoid the construction of most non-shortest paths until they are actually needed in answers. Freezing may seem intuitively clear, but making it work involves subtle details and requires an intricate proof of correctness.

Our main theoretical contribution is the algorithm GTF, which incorporates freezing, and its proof of correctness. Our main practical contribution is showing experimentally (in Section 5 and Appendix B of [5]) that GTF is both more efficient and more scalable than existing systems. This contribution is especially significant in light of the following. First, GTF is complete (i.e., it does not miss answers); moreover, we show experimentally that not missing answers is important in practice. Second, the order of generating answers is by increasing height. This order is commonly deemed a good strategy for an initial ranking that is likely to be in a good correlation with the final one (i.e., by increasing weight).

2 Preliminaries

We model data as a directed graph G, similarly to [1]. Data graphs can be constructed from a variety of formants (e.g., RDB, XML and RDF). Nodes represent entities and relationships, while edges correspond to connections among them (e.g., foreign-key references when the data graph is constructed from a relational database). We assume that text appears only in the nodes. This is not a limitation, because we can always split an edge (with text) so that it passes through a node. Some nodes are for keywords, rather than entities and relationships. In particular, for each keyword k that appears in the data graph, there is a dedicated node. By a slight abuse of notation, we do not distinguish between a keyword k and its node – both are called *keyword* and denoted by k. For all nodes v of the data graph that contain a keyword k, there is a directed edge from v to k. Thus, keywords have only incoming edges.

Figure 1 shows a snippet of a data graph. The dashed part should be ignored unless explicitly stated otherwise. Ordinary nodes are shown as ovals. For clarity, the type of each node appears inside the oval. Keyword nodes are depicted as rectangles. To keep the figure

small, only a few of the keywords that appear in the graph are shown as nodes. For example, a type is also a keyword and has its own node in the full graph. For each oval, there is an edge to every keyword that it contains.

Let $G = (V, E)$ be a directed data graph, where V and E are the sets of nodes and edges, respectively. A directed path is denoted by $\langle v_1, \ldots, v_m \rangle$. We only consider *rooted* (and, hence, directed) subtrees T of G. That is, T has a unique node r, such that for all nodes u of T, there is exactly one path in T from r to u. Consider a query K, that is, a set of at least two keywords. A *K-subtree* is a rooted subtree of G, such that its leaves are exactly the keywords of K. We say that a node $v \in V$ is a *K-root* if it is the root of some K-subtree of G. It is observed in [1] that v is a K-root if and only if for all $k \in K$, there is a path in G from v to k. An *answer* to K is a K-subtree T that is *non-redundant* (or *reduced*) in the sense that no proper subtree T' of T is also a K-subtree. It is easy to show that a K-subtree T of G is an answer if and only if the root of T has at least two children. Even if v is a K-root, it does not necessarily follow that there is an answer to K that is rooted at v (because it is possible that in all K-subtrees rooted at v, there is only one child of v).

Figure 3 shows three answers to the query $\{France, Paris\}$ over the data graph of Figure 1. The answer A_1 means that the city Paris is located in a province containing the word France in its name. The answer A_2 states that the city Paris is located in the country France. Finally, the answer A_3 means that Paris is located in a province which is located in France.

Now, consider also the dashed part of Figure 1, that is, the keyword *Seine* and the node *river* with its outgoing edges. There is a path from *river* to every keyword of $K = \{France, Paris\}$. Hence, *river* is a K-root. However, the K-subtree of Figure 2 is not an answer to K, because its root has only one child.

For ranking, the nodes and edges of the data graph have positive weights. The *weight* of a path (or a tree) is the sum of weights of all its nodes and edges. The rank of an answer is inversely proportional to its weight. The *height* of a tree is the maximal weight over all paths from the root to any leaf (which is a keyword of the query). For example, suppose that the weight of each node and edge is 1. The heights of the answers A_1 and A_3 (of Figure 3) are 5 and 7, respectively. In A_1, the path from the root to *France* is a minimal (i.e., shortest) one between these two nodes, in the whole graph, and its weight is 5. In A_3, however, the path from the root (which is the same as in A_1) to *France* has a higher weight, namely, 7.

3 The GTF Algorithm

3.1 The Naive Approach

Consider a query $K = \{k_1, \ldots, k_n\}$. In [1], they use a backward shortest-path iterator from each keyword node k_i. That is, starting at each k_i, they apply Dijkstra's shortest-path algorithm in the opposite direction of the edges. If a node v is reached by the backward iterators from all the k_i, then v is a K-root (and, hence, might be the root of some answers). In this way, answers are generated by increasing height. However, this approach can only find answers that consist of shortest paths from the root to the keyword nodes. Hence, it misses answers (e.g., it cannot produce A_3 of Figure 3).

Dijkstra's algorithm can be straightforwardly generalized to construct all the simple (i.e., acyclic) paths by increasing weight. This approach is used[1] in [11] and it consists of two parts: path construction and answer production. Each constructed path is from

[1] They used it on a small *summary* graph to construct database queries from keywords.

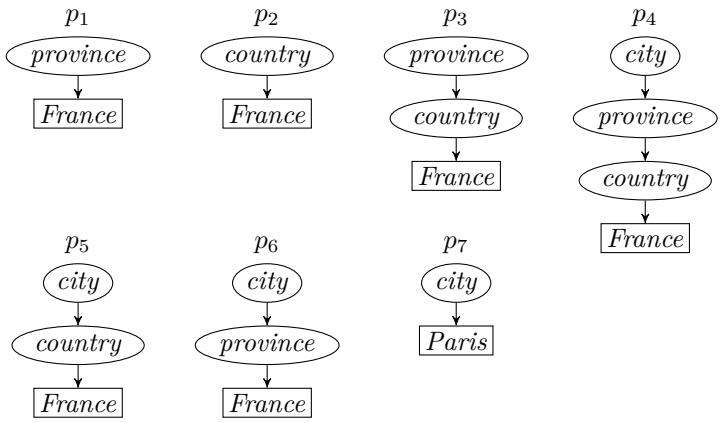

Figure 4 Paths to keywords in the graph snippet of Figure 1.

some node of G to a keyword of K. Since paths are constructed backwards, the algorithm starts simultaneously from all the keyword nodes of K. It uses a single priority queue to generate, by increasing weight, all simple paths to every keyword node of K. When the algorithm discovers that a node v is a K-root (i.e., there is a path from v to every k_i), it starts producing answers rooted at v. This is done by considering every combination of paths $p_1, \ldots, p_n$, such that p_i is from v to k_i $(1 \leq i \leq n)$. If the combination is a non-redundant K-subtree of G, then it is produced as an answer. It should be noted that in [11], answers are subgraphs; hence, every combination of paths $p_1, \ldots, p_n$ is an answer. We choose to produce subtrees as answers for two reasons. First, in the experiments of Section 5, we compare our approach with other systems that produce subtrees. Second, it is easier for users to understand answers that are presented as subtrees, rather than subgraphs.

The drawback of the above approach is constructing a large number of paths that are never used in any of the generated answers. To overcome this problem, the next section introduces the technique of *freezing*, thereby most non-minimal paths are generated only if they are actually needed to produce answers. Section 3.3 describes the algorithm *Generating Trees with Freezing* (GTF) that employs this technique.

To save space (when constructing all simple paths), we use the common technique known as *tree of paths*. In particular, a path p is a linked list, such that its first node points to the rest of p. As an example, consider the graph snippet of Figure 1. The paths that lead to the keyword *France* are p_1, p_2, p_3, p_4, p_5 and p_6, shown in Figure 4. Their tree of paths is presented in Figure 5.

Since we build paths backwards, a data graph is preprocessed to produce for each node v the set of its *parents*, that is, the set of nodes v', such that (v', v) is an edge of the data graph. We use the following notation. Given a path p that starts at a node v, the extension of p with a parent v' of v is denoted by $v' \rightarrow p$. Note that v' is the first node of $v' \rightarrow p$ and v is the second one.

3.2 Incorporating Freezing

The general idea of freezing is to avoid the construction of paths that cannot contribute to production of answers. To achieve that, a non-minimal path p is frozen until it is certain that p can reach (when constructed backwards) a K-root. In particular, the first path that reaches a node v is always a minimal one. When additional paths reach v, they are frozen

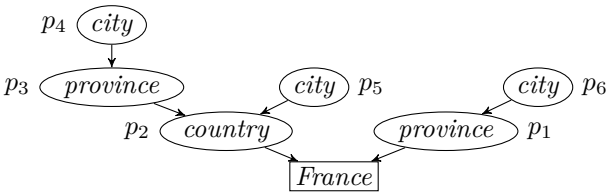

Figure 5 Tree of paths.

there until v is discovered to be on a path from a K-root to a keyword node. The process of answer production in the GTF algorithm remains the same as in the naive approach.

We now describe some details about the implementation of GTF. We mark nodes of the data graph as either *active*, *visited* or *in-answer*. Since we simultaneously construct paths to all the keywords (of the query $K = \{k_1, \ldots, k_n\}$), a node has a separate mark for each keyword. The marks of a node v are stored in the array $v.marks$, which has an entry for each keyword. For a keyword k_i, the mark of v (i.e., $v.marks[k_i]$) means the following. Node v is *active* if we have not yet discovered that there is a path from v to k_i. Node v is *visited* if a minimal path from v to k_i has been produced. And v is marked as *in-answer* when we discover for the first time that v is on a path from some K-root to k_i.

If $v.marks[k_i]$ is *visited* and a path p from v to k_i is removed from the queue, then p is *frozen* at v. Frozen paths from v to k_i are stored in a dedicated list $v.frozen[k_i]$. The paths of $v.frozen[k_i]$ are *unfrozen* (i.e., are moved back into the queue) when $v.marks[k_i]$ is changed to *in-answer*.

We now describe the execution of GTF on the graph snippet of Figure 1, assuming that the query is $K = \{France, Paris\}$. Initially, two paths $\langle France\rangle$ and $\langle Paris\rangle$, each consisting of one keyword of K, are inserted into the queue, where lower weight means higher priority. Next, the top of the queue is removed; suppose that it is $\langle France\rangle$. First, we change $France.marks[France]$ to *visited*. Second, for each parent v of *France*, the path $v \to France$ is inserted into the queue; namely, these are the paths p_1 and p_2 of Figure 4. We continue to iterate in this way. Suppose that now $\langle Paris\rangle$ has the lowest weight. So, it is removed from the queue, $Paris.marks[Paris]$ is changed to *visited*, and the path p_7 (of Figure 4) is inserted into the queue.

Now, let the path p_1 be removed from the queue. As a result, $province.marks[France]$ is changed to *visited*, and the path $p_6 = city \to p_1$ is inserted into the queue. Next, assume that p_2 is removed from the queue. So, $country.marks[France]$ is changed to *visited*, and the paths $p_3 = province \to p_2$ and $p_5 = city \to p_2$ are inserted into the queue.

Now, suppose that p_3 is at the top of the queue. So, p_3 is removed and immediately frozen at *province* (i.e., added to $province.frozen[France]$), because $province.marks[France] = visited$. Consequently, no paths are added to the queue in this iteration. Next, assume that p_6 is removed from the queue. The value of $city.marks[France]$ is changed to *visited* and no paths are inserted into the queue, because *city* has no incoming edges.

Now, suppose that p_7 is at the top of the queue. So, it is removed and $city.marks[Paris]$ is changed to *visited*. Currently, both $city.marks[Paris]$ and $city.marks[France]$ are *visited*. That is, there is a path from *city* to all the keywords of the query $\{France, Paris\}$. Recall that the paths that have reached *city* so far are p_6 and p_7. For each one of those paths p, the following is done, assuming that p ends at the keyword k. For each node v of p, we change the mark of v for k to *in-answer* and unfreeze paths to k that are frozen at v. Doing it for p_6 means that $city.marks[France]$, $province.marks[France]$ and $France.marks[France]$ are all changed to *in-answer*. In addition, the path p_3 is removed from $province.frozen[France]$

and inserted back into the queue. We act similarly on p_7. That is, *city.marks*[*Paris*] and *Paris.marks*[*Paris*] are changed to *in-answer*. In this case, there are no paths to be unfrozen.

Now, the marks of *city* for all the keywords (of the query) are *in-answer*. Hence, we generate answers from the paths that have already reached *city*. As a result, the answer A_1 of Figure 3 is produced. Moreover, from now on, when a new path reaches *city*, we will try to generate more answers by applying **produceAnswers**$(\mathcal{P}, p)$.

3.3 The Pseudocode of the GTF Algorithm

The GTF algorithm is presented in Figure 6 and its helper procedures in Figure 7. The input is a data graph $G = (V, E)$ and a query $K = \{k_1, \ldots, k_n\}$. The algorithm uses a single priority queue Q to generate, by increasing weight, all simple paths to every keyword node of K. For each node $v \in V$, there is a flag *isKRoot* that indicates whether v has a path to each keyword of K. Initially, that flag is **false**. For each node $v \in V$, the set of the constructed paths from v to the keyword k is stored in $v.paths[k]$, which is initially empty. Also, for all the keywords of K and nodes of G, we initialize the marks to be *active* and the lists of frozen paths to be empty. The paths are constructed backwards, that is, from the last node (which is always a keyword). Therefore, for each $k \in K$, we insert the path $\langle k \rangle$ (consisting of the single node k) into Q. All these initializations are done in lines 1–9 (of Figure 6).

The main loop of lines 10–37 is repeated while Q is not empty. Line 11 removes the best (i.e., least-weight) path p from Q. Let v and k_i be the first and last, respectively, nodes of p. Line 12 freezes p provided that it has to be done. This is accomplished by calling the procedure **freeze**(p) of Figure 7 that operates as follows. If the mark of v for k_i is *visited*, then p is frozen at v by adding it to $v.frozen[k_i]$ and **true** is returned; in addition, the main loop continues (in line 13) to the next iteration. Otherwise, **false** is returned and p is handled as we describe next.

Line 15 checks if p is the first path from v to k_i that has been removed from Q. If so, line 16 changes the mark of v for k_i from *active* to *visited*. Line 17 assigns **true** to the flag *relax*, which means that (as of now) p should spawn new paths that will be added to Q.

The test of line 18 splits the execution of the algorithm into two cases. If v is a K-root (which must have been discovered in a previous iteration and means that for every $k \in K$, there is a path from v to k), then the following is done. First, line 19 calls the procedure **unfreeze**(p, Q) of Figure 7 that unfreezes (i.e., inserts into Q) all the paths to k_i that are frozen at nodes of p (i.e., the paths of $\bar{v}.frozen[k_i]$, where $\bar{v}$ is a node of p). In addition, for all nodes $\bar{v}$ of p, the procedure **unfreeze**(p, Q) changes the mark of $\bar{v}$ for k_i to *in-answer*. Second, line 20 tests whether p is acyclic. If so, line 21 adds p to the paths of v that reach k_i, and line 22 produces new answers that include p by calling **produceAnswers** of Figure 7. The pseudocode of **produceAnswers**$(v.paths, p)$ is just an efficient implementation of considering every combination of paths $p_1, \ldots, p_n$, such that p_i is from v to k_i $(1 \leq i \leq n)$, and checking that it is an answer to K. (It should be noted that GTF generates answers by increasing height.) If the test of line 20 is **false**, then the flag *relax* is changed back to **false**, thereby ending the current iteration of the main loop.

If the test of line 18 is **false** (i.e., v has not yet been discovered to be a K-root), the execution continues in line 26 that adds p to the paths of v that reach k_i. Line 27 tests whether v is now a K-root and if so, the flag *isKRoot* is set to **true** and the following is done. The nested loops of lines 29–32 iterate over all paths p' (that have already been discovered) from v to any keyword node of K (i.e., not just k_i). For each p', where k' is the last node of p' (and, hence, is a keyword), line 31 calls **unfreeze**(p', Q), thereby inserting into Q all the paths to k' that are frozen at nodes of p' and changing the mark (for k') of those nodes to

Algorithm: *GTF (Generate Trees with Freezing)*
Input: $G = (V, E)$ is a data graph
K is a set of keyword nodes
Output: Answers to K

```
1: Q ← an empty priority queue
2: for v ∈ V do
3:    v.isKRoot ← false
4: for v ∈ V and k ∈ K do
5:    v.paths[k] ← ∅
6:    v.frozen[k] ← ∅
7:    v.marks[k] ← active
8: for k ∈ K do
9:    Q.insert(⟨k⟩)
10: while Q is not empty do
11:    p ← Q.remove()
12:    if freeze(p) then
13:       continue
14:    v ← first(p)
15:    if v.marks[p.keyword] = active then
16:       v.marks[p.keyword] ← visited
17:    relax ← true
18:    if v.isKRoot = true then
19:       unfreeze(p, Q)
20:       if p has no cycles then
21:          v.paths[p.keyword].add(p)
22:          produceAnswers(v.paths, p)
23:       else
24:          relax ← false
25:    else
26:       v.paths[p.keyword].add(p)
27:       if for all k ∈ K, it holds that v.paths[k] ≠ ∅ then
28:          v.isKRoot ← true
29:          for k ∈ K do
30:             for p′ ∈ v.pahts[k] do
31:                unfreeze(p′, Q)
32:             remove cyclic paths from v.paths[k]
33:          produceAnswers(v.paths, p)
34:    if relax then
35:       for v′ ∈ parents(v) do
36:          if v′ is not on p or v′ → p is essential then
37:             Q.insert(v′ → p)
```

Figure 6 The GTF algorithm.

Procedure: **freeze**(p)

```
1: if first(p).marks[p.keyword] = visited then
2:     first(p).frozen[p.keyword].add(p)
3:     return true
4: else
5:     return false
```

Procedure: **unfreeze**(p, Q)

```
1: p' ← p
2: while p' ≠ ⊥ do
3:     v̄ ← first(p')
4:     if v̄.marks[p.keyword] ≠ in-answer then
5:         v̄.marks[p.keyword] ← in-answer
6:         for p'' ∈ v̄.frozen[p.keyword] do
7:             Q.insert(p'')
8:     p' ← predecessor(p')
```

Procedure: **produceAnswers**$(\mathcal{P}, p)$
Output: answers rooted at **first**(p)

```
1: P[p.keyword] ← {p}
2: iter ← new pathGroups(P)
3: while iter.hasNext() do
4:     P̄ ← iter.next()
5:     a ← combine all the paths in P̄
       /* next() ensures that the combination of all the paths in P̄ yields a
       tree (rather than a graph) */
6:     if the root of a has more than one
       child then
7:         output a
```

Figure 7 Helper procedures for the GTF algorithm.

in-answer. Line 32 removes all the cyclic paths among those stored at v. Line 33 generates answers from the paths that remain at v.

If the test of line 34 is **true**, the relaxation of p is done in lines 35–37 as follows. For each parent v' of v, the path $v' \to p$ is inserted into Q if either one of the following two holds (as tested in line 36). First, v' is not on p. Second, $v' \to p$ is essential, according to the following definition. The path $v' \to p$ is *essential* if v' appears on p and the section of $v' \to p$ from its first node (which is v') to the next occurrence of v' has at least one node u, such that $u.marks[k] = visited$, where the keyword k is the last node of p. Appendix A of [5] gives an example that shows why essential paths (which are cyclic) have to be inserted into Q.

Note that due to line 24, no cyclic path $p[v, k]$ is relaxed if v has already been discovered to be a K-root in a previous iteration. The reason is that none of the nodes along $p[v, k]$ could have the mark *visited* for the keyword k (hence, no paths are frozen at those nodes).

Observe that before v is known to be a K-root, we add cyclic paths to the array $v.paths$. Only when discovering that v is a K-root, do we remove all cyclic paths from $v.paths$ (in line 32) and stop adding them in subsequent iterations. This is lazy evaluation, because prior to knowing that answers with the K-root v should be produced, it is a waste of time to test whether paths from v are cyclic.

4 Correctness and Complexity of GTF

4.1 Definitions and Observations

Before proving correctness of the GTF algorithm, we define some notation and terminology (in addition to those of Section 2) and state a few observations. Recall that the data graph is $G = (V, E)$. Usually, a keyword is denoted by k, whereas r, u, v and z are any nodes of V.

We only consider directed paths of G that are defined as usual. If p is a path from v to k, then we write it as $p[v, k]$ when we want to explicitly state its first and last nodes. We say that node u is *reachable* from v if there is a path from v to u.

A *suffix* of $p[v, k]$ is a traversal of $p[v, k]$ that starts at (some particular occurrence of) a node u and ends at the last node of p. Hence, a suffix of $p[v, k]$ is denoted by $p[u, k]$. A *prefix*

of $p[v,k]$ is a traversal of $p[v,k]$ that starts at v and ends at (some particular occurrence of) a node u. Hence, a prefix of $p[v,k]$ is denoted by $p[v,u]$. A suffix or prefix of $p[v,k]$ is *proper* if it is different from $p[v,k]$ itself.

Consider two paths $p_1[v,z]$ and $p_2[z,u]$; that is, the former ends in the node where the latter starts. Their *concatenation*, denoted by $p_1[v,z] \circ p_2[z,u]$, is obtained by joining them at node z.

As already mentioned in Section 2, a positive *weight* function w is defined on the nodes and edges of G. The weight of a path $p[v,u]$, denoted by $w(p[v,u])$, is the sum of weights over all the nodes and edges of $p[v,u]$. A *minimal* path from v to u has the minimum weight among all paths from v to u. Since the weight function is positive, there are no zero-weight cycles. Therefore, a minimal path is acyclic. Also observe that the weight of a proper suffix or prefix is strictly smaller than that of the whole path.[2]

Let K be a query (i.e., a set of at least two keywords). Recall from Section 2 the definitions of K-root, K-subtree and height of a subtree. The *best height* of a K-root r is the maximum weight among all the minimal paths from r to any keyword $k \in K$. Note that the height of any K-subtree rooted at r is at least the best height of r.

Consider a nonempty set of nodes S and a node v. If v is reachable from every node of S, then we say that node $u \in S$ is *closest to* v if a minimal path from u to v has the minimum weight among all paths from any node of S to v.

Similarly, if every node of S is reachable from v, then we say that node $u \in S$ is *closest from* v if a minimal path from v to u has the minimum weight among all paths from v to any node of S.

In the sequel, line numbers refer to the algorithm GTF of Figure 6, unless explicitly stated otherwise. We say that a node $v \in V$ is *discovered as a K-root* if the test of line 27 is satisfied and $v.isRoot$ is assigned **true** in line 28. Observe that the test of line 27 is **true** if and only if for all $k \in K$, it holds that $v.marks[k]$ is either *visited* or *in-answer*. Also note that line 28 is executed at most once for each node v of G. Thus, there is at most one iteration of the main loop (i.e., line 10) that discovers v as K-root.

We say that a path p is *constructed* when it is inserted into Q for the first time, which must happen in line 37. A path is *exposed* when it is removed from Q in line 11. Observe that a path $p[v,k]$ may be exposed more than once, due to freezing and unfreezing.

▶ **Proposition 1.** *A path can be exposed at most twice.*

Proof. When an iteration exposes a path $p[v,k]$ for the first time, it does exactly one of the following. It freezes $p[v,k]$ at node v, discard $p[v,k]$ due to line 24, or extend (i.e., relax) $p[v,k]$ in the loop of line 35 and inserts the results into Q in line 37. Note that some relaxations of $p[v,k]$ are never inserted into Q, due to the test of line 36. Only if $p[v,k]$ is frozen at v, can it be inserted a second time into Q, in line 7 of the procedure **unfreeze** (Figure 7) that also sets $v.marks[k]$ to *in-answer*. But then $p[v,k]$ cannot freeze again at v, because $v.marks[k]$ does not change after becoming *in-answer*. Therefore, $p[v,k]$ cannot be inserted into Q a third time. ◀

In the next section, we sometimes refer to the mark of a node v of a path p. It should be clear from the context that we mean the mark of v for the keyword where p ends.

[2] For the proof of correctness, it is enough for the weight function to be non-negative (rather than positive) provided that every cycle has a positive weight.

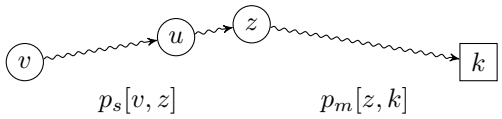

Figure 8 The path $\bar{p}[v, k]$.

4.2 The Proof

We start with an auxiliary lemma that considers the concatenation of two paths, where the linking node is z, as shown in Figure 8 (note that a wavy arrow denotes a path, rather than a single edge). Such a concatenation is used in the proofs of subsequent lemmas.

▶ **Lemma 2.** *Let k be a keyword of the query K, and let v and z be nodes of the data graph. Consider two paths $p_s[v, z]$ and $p_m[z, k]$. Let $\bar{p}[v, k]$ be their concatenation at node z, that is,*

$$\bar{p}[v, k] = p_s[v, z] \circ p_m[z, k].$$

Suppose that the following hold at the beginning of iteration i of the main loop (line 10).

1. *The path $p_s[v, z]$ is minimal or (at least) acyclic.*
2. *The path $p_m[z, k]$ has changed $z.marks[k]$ from active to visited in an earlier iteration.*
3. *$z.marks[k] =$ visited.*
4. *For all nodes $u \neq z$ on the path $p_s[v, z]$, the suffix $\bar{p}[u, k]$ is not frozen at u.*
5. *The path $\bar{p}[v, k]$ has not yet been exposed.*

Then, some suffix of $\bar{p}[v, k]$ must be on Q at the beginning of iteration i.

Proof. Suppose, by way of contradiction, that no suffix of $\bar{p}[v, k]$ is on Q at the beginning of iteration i. Since $\bar{p}[v, k]$ has not yet been exposed, there are two possible cases regarding its state. We derive a contradiction by showing that none of them can happen.

Case 1: Some suffix of $\bar{p}[v, k]$ is frozen. This cannot happen at any node of $\bar{p}[z, k]$ (which is the same as $p_m[z, k]$), because Condition 3 implies that $p_m[z, k]$ has already changed $z.marks[k]$ to *visited*. Condition 4 implies that it cannot happen at the other nodes of $\bar{p}[v, k]$ (i.e., the nodes u of $p_s[v, z]$ that are different from z).

Case 2: Some suffix of $\bar{p}[v, k]$ has already been discarded (in an earlier iteration) either by the test of line 36 or due to line 24. This cannot happen to any suffix of $\bar{p}[z, k]$ (which is the same as $p_m[z, k]$), because $p_m[z, k]$ has already changed $z.marks[k]$ to *visited*. We now show that it cannot happen to any other suffix $\bar{p}[u, k]$, where u is a node of $p_s[v, z]$ other than z. Note that $\bar{p}[v, k]$ (and hence $\bar{p}[u, k]$) is not necessarily acyclic. However, the lemma states that $p_s[v, z]$ is acyclic. Therefore, if the suffix $\bar{p}[u, k]$, has a cycle that includes u, then it must also include z. But $z.marks[k]$ is *visited* from the moment it was changed to that value until the beginning of iteration i (because a mark cannot be changed to *visited* more than once). Hence, the suffix $\bar{p}[u, k]$ could not have been discarded by the test of line 36. It is also not possible that line 24 has already discarded $\bar{p}[u, k]$ for the following reason. If line 24 is reached (in an iteration that removed $\bar{p}[u, k]$ from Q), then for all nodes x on $\bar{p}[u, k]$, line 19 has already changed $x.marks[k]$ to *in-answer*. Therefore, $z.marks[k]$ cannot be *visited* at the beginning of iteration i.

It thus follows that some suffix of $\bar{p}[v, k]$ is on Q at the beginning of iteration i. ◀

▶ **Lemma 3.** *For all nodes $v \in V$ and keywords $k \in K$, the mark $v.marks[k]$ can be changed from active to visited only by a minimal path from v to k.*

Proof. Suppose that the lemma is not true for some keyword $k \in K$. Let v be a closest node to k among all those violating the lemma with respect to k. Node v is different from k, because the path $\langle k \rangle$ marks k as *visited*. We will derive a contradiction by showing that a minimal path changes $v.marks[k]$ from *active* to *visited*.

Let $p_s[v, k]$ be a minimal path from v to k. Consider the iteration i of the main loop (line 10 in Figure 6) that changes $v.marks[k]$ to *visited* (in line 16). Among all the nodes of $p_s[v, k]$ in which suffixes of some minimal paths from v to k are frozen at the beginning of iteration i, let z be the first one when traversing $p_s[v, k]$ from v to k (i.e., on the path $p_s[v, z]$, node z is the only one in which such a suffix is frozen). Node z exists for the following three reasons.

- The path $p_s[v, k]$ has not been exposed prior to iteration i, because we assume that $v.marks[k]$ is changed to *visited* in iteration i and that change can happen only once.
- The path $p_s[v, k]$ is acyclic (because it is minimal), so a suffix of $p_s[v, k]$ could not have been discarded either by the test of line 36 or due to line 24.
- The path $p_s[v, k]$ (or any suffix thereof) cannot be on the queue at the beginning of iteration i, because v violates the lemma, which means that a non-minimal path from v to k must be removed from the queue at the beginning of that iteration.

The above three observations imply that a proper suffix of $p_s[v, k]$ must be frozen at the beginning of iteration i and, hence, node z exists. Observe that z is different from v, because a path to k can be frozen only at a node $\hat{v}$, such that $\hat{v}.marks[k] = visited$, whereas we assume that $v.marks[k]$ is *active* at the beginning of iteration i.

By the selection of v and $p_s[v, k]$ (and the above fact that $z \neq v$), node z does not violate the lemma, because $p_s[z, k]$ is a proper suffix of $p_s[v, k]$ and, hence, z is closer to k than v. Therefore, according to the lemma, there is a minimal path $p_m[z, k]$ that changes $z.marks[k]$ to *visited*. Consequently,

$$w(p_m[z, k]) \leq w(p_s[z, k]). \tag{1}$$

Now, consider the path

$$\bar{p}[v, k] = p_s[v, z] \circ p_m[z, k]. \tag{2}$$

Since $p_s[v, k]$ is a minimal path from v to k, Equations (1) and (2) imply that so is $\bar{p}[v, k]$.

We now show that the conditions of Lemma 2 are satisfied at the beginning of iteration i. In particular, Condition 1 holds, because $p_s[v, k]$ is acyclic (since it is minimal) and, hence, so is the path $p_s[v, z]$. Condition 2 is satisfied, because of how $p_m[z, k]$ is defined. Condition 3 holds, because we chose z to be a node where a path to k is frozen. Condition 4 is satisfied, because of how z was chosen and the fact that $\bar{p}[v, k]$ is minimal. Condition 5 is satisfied, because we have assumed that $v.marks[k]$ is changed from *active* to *visited* during iteration i.

By Lemma 2, a suffix of $\bar{p}[v, k]$ must be on the queue at the beginning of iteration i. This contradicts our assumption that a non-minimal path (which has a strictly higher weight than any suffix of $\bar{p}[v, k]$) changes $v.marks[k]$ from *active* to *visited* in iteration i. ◀

▶ **Lemma 4.** *For all nodes $v \in V$ and keywords $k \in K$, such that k is reachable from v, if $v.marks[k]$ is active at the beginning of an iteration of the main loop (line 10), then Q contains a suffix (which is not necessarily proper) of a minimal path from v to k.*

Proof. The lemma is certainly true at the beginning of the first iteration, because the path $\langle k \rangle$ is on Q. Suppose that the lemma does not hold at the beginning of iteration i. Thus, every minimal path $p[v, k]$ has a proper suffix that is frozen at the beginning of iteration i.

(Note that a suffix of a minimal path cannot be discarded either by the test of line 36 or due to line 24, because it is acyclic.) Let z be the closest node from v having such a frozen suffix. Hence, $z.marks[k]$ is *visited* and $z \neq v$ (because $v.marks[k]$ is *active*). By Lemma 3, a minimal path $p_m[z,k]$ has changed $z.marks[k]$ to *visited*. Let $p_s[v,z]$ be a minimal path from v to z. Consider the path

$$\bar{p}[v,k] = p_s[v,z] \circ p_m[z,k].$$

The weight of $\bar{p}[v,k]$ is no more than that of a minimal path from v to k, because both $p_s[v,z]$ and $p_m[v,k]$ are minimal and the choice of z implies that it is on some minimal path from v to k. Hence, $\bar{p}[v,k]$ is a minimal path from v to k.

We now show that the conditions of Lemma 2 are satisfied. Conditions 1–3 clearly hold. Condition 4 is satisfied because of how z is chosen and the fact that $\bar{p}[v,k]$ is minimal. Condition 5 holds because $v.marks[k]$ is *active* at the beginning of iteration i.

By Lemma 2, a suffix of $\bar{p}[v,k]$ is on Q at the beginning of iteration i, contradicting our initial assumption. ◀

▶ **Lemma 5.** *Any constructed path can have at most $2n(n+1)$ nodes, where $n = |V|$ (i.e., the number of nodes in the graph). Hence, the algorithm constructs at most $(n+1)^{2n(n+1)}$ paths.*

Proof. We say that $v_m \to \cdots \to v_1$ is a *repeated run* in a path $\bar{p}$ if some suffix (not necessarily proper) of $\bar{p}$ has the form $v_m \to \cdots \to v_1 \to p$, where each v_i also appears in any two positions of p. In other words, for all i $(1 \leq i \leq m)$, the occurrence of v_i in $v_m \to \cdots \to v_1$ is (at least) the third one in the suffix $v_m \to \cdots \to v_1 \to p$. (We say that it is the third, rather than the first, because paths are constructed backwards).

When a path $p'[v',k']$ reaches a node v' for the third time, the mark of v' for the keyword k' has already been changed to *in-answer* in a previous iteration. This follows from the following two observations. First, the first path to reach a node v' is also the one to change its mark to *visited*. Second, a path that reaches a node marked as *visited* can be unfrozen only when that mark is changed to *in-answer*.

Let $v_m \to \cdots \to v_1$ be a repeated run in $\bar{p}$ and suppose that $m > n = |V|$. Hence, there is a node v_i that appears twice in the repeated run; that is, there is a $j < i$, such that $v_j = v_i$. If the path $v_i \to \cdots \to v_1 \to p$ is considered in the loop of line 35, then it would fail the test of line 36 (because, as explained earlier, all the nodes on the cycle $v_i \to \cdots \to v_j$ are already marked as *in-answer*). We conclude that the algorithm does not construct paths that have a repeated run with more than n nodes.

It thus follows that two disjoint repeated runs of a constructed path $\bar{p}$ must be separated by a node that appears (in a position between them) for the first or second time. A path can have at most $2n$ positions, such that in each one a node appears for the first or second time. Therefore, if a path $\bar{p}$ is constructed by the algorithm, then it can have at most $2n(n+1)$ nodes. Using n distinct nodes, we can construct at most $(n+1)^{2n(n+1)}$ paths with $2n(n+1)$ or fewer nodes. ◀

▶ **Lemma 6.** *K-Roots have the following two properties.*

1. *All the K-roots are discovered before the algorithm terminates. Moreover, they are discovered in the increasing order of their best heights.*
2. *Suppose that r is a K-root with a best height b. If $p[v,k]$ is a path (from any node v to any keyword k) that is exposed before the iteration that discovers r as a K-root, then $w(p[v,k]) \leq b$.*

Proof. We first prove Part 1. Suppose that a keyword k is reachable from node v. As long as $v.marks[k]$ is *active* at the beginning of the main loop (line 10), Lemma 4 implies that the queue Q contains (at least) one suffix of a minimal path from v to k. By Lemma 5, the algorithm constructs a finite number of paths. By Proposition 1, the same path can be inserted into the queue at most twice. Since the algorithm does not terminate while Q is not empty, $v.marks[k]$ must be changed to *visited* after a finite time. It thus follows that each K-root is discovered after a finite time.

Next, we show that the K-roots are discovered in the increasing order of their best heights. Let r_1 and r_2 be two K-roots with best heights b_1 and b_2, respectively, such that $b_1 < b_2$. Lemma 3 implies the following for r_i $(i = 1, 2)$. For all keywords $k \in K$, a minimal path from r_i to k changes $r_i.marks[k]$ from *active* to *visited*; that is, r_i is discovered as a K-root by minimal paths. Suppose, by way of contradiction, that r_2 is discovered first. Hence, a path with weight b_2 is removed from Q while Lemma 4 implies that a suffix with a weight of at most b_1 is still on Q. This contradiction completes the proof of Part 1.

Now, we prove Part 2. As shown in the proof of Part 1, a K-root is discovered by minimal paths. Let r be a K-root with best height b. Suppose, by way of contradiction, that a path $p[v, k]$, such that $w(p[v, k]) > b$, is exposed before the iteration, say i, that discovers r as a K-root. By Lemma 4, at the beginning of iteration i, the queue Q contains a suffix with weight of at most b. Hence, $p[v, k]$ cannot be removed from Q at the beginning of iteration i. This contradiction proves Part 2. ◀

▶ **Lemma 7.** *Suppose that node v is discovered as a K-root at iteration i. Let $p_1[v', k']$ and $p_2[v, k]$ be paths that are exposed in iterations j_1 and j_2, respectively. If $i < j_1 < j_2$, then $w(p_1[v', k']) \leq w(p_2[v, k])$. Note that k and k' are not necessarily the same and similarly for v and v'; moreover, v' has not necessarily been discovered as a K-root.*

Proof. Suppose the lemma is false. In particular, consider an iteration j_1 of the main loop (line 10) that violates the lemma. That is, the following hold in iteration j_1.

- Node v has already been discovered as a K-root in an earlier iteration (so, there are no frozen paths at v).
- A path $p_1[v', k']$ is exposed in iteration j_1.
- A path $p_2[v, k]$ having a strictly lower weight than $p_1[v', k']$ (i.e., $w(p_2[v, k]) < w(p_1[v', k'])$) will be exposed after iteration j_1. Hence, a proper suffix of this path is frozen at some node z during iteration j_1.

For a given v and $p_1[v', k']$, there could be several paths $p_2[v, k]$ that satisfy the third condition above. We choose one, such that its suffix is frozen at a node z that is closest from v. Since v has already been discovered as a K-root, z is different from v.

Clearly, $z.marks[k]$ is changed to *visited* before iteration j_1. By Lemma 3, a minimal path $p_m[z, k]$ does that. Let $p_s[v, z]$ be a minimal path from v to z.

Consider the path

$$\bar{p}[v, k] = p_s[v, z] \circ p_m[z, k].$$

Since both $p_s[v, z]$ and $p_m[z, k]$ are minimal, the weight of their concatenation (i.e., $\bar{p}[v, k]$) is no more than that of $p_2[v, k]$ (which is also a path that passes through node z). Hence, $w(\bar{p}[v, k]) < w(p_1[v', k'])$.

We now show that the conditions of Lemma 2 are satisfied at the beginning of iteration j_1 (i.e., j_1 corresponds to i in Lemma 2). Conditions 1–2 clearly hold. Condition 3 is satisfied because a suffix of $p_2[v, k]$ is frozen at z. Condition 4 holds, because of the choice of z and the

fact $w(\bar{p}[v,k]) < w(p_1[v',k'])$ that was shown earlier. Condition 5 holds, because otherwise $\bar{p}[v,k]$ would be unfrozen and $z.marks[k]$ would be *in-answer* rather than *visited*.

By Lemma 2, a suffix of $\bar{p}[v,k]$ is on the queue at the beginning of iteration j_1. This contradicts the assumption that the path $p_1[v',k']$ is removed from the queue at the beginning of iteration j_1, because $\bar{p}[v,k]$ (and, hence, any of its suffixes) has a strictly lower weight. ◀

▶ **Lemma 8.** *For all nodes $v \in V$, such that v is a K-root, the following holds. If z is a node on a simple path from v to some $k \in K$, then $z.marks[k] \neq$ visited when the algorithm terminates.*

Proof. The algorithm terminates when the test of line 10 shows that Q is empty. Suppose that the lemma is not true. Consider some specific K-root v and keyword k for which the lemma does not hold. Among all the nodes z that violate the lemma with respect to v and k, let z be a closest one from v. Observe that z cannot be v, because of the following two reasons. First, by Lemma 6, node v is discovered as a K-root before termination. Second, when a K-root is discovered (in lines 27–28), all its marks become *in-answer* in lines 29–31.

Suppose that $p_m[z,k]$ is the path that changes $z.marks[k]$ to *visited*. Let $p_s[v,z]$ be a minimal path from v to z. Note that $p_s[v,z]$ exists, because z is on a simple path from v to k. Consider the path

$$\bar{p}[v,k] = p_s[v,z] \circ p_m[z,k].$$

Suppose that the test of line 10 is **false** (and, hence, the algorithm terminates) on iteration i. We now show that the conditions of Lemma 2 are satisfied at the beginning of that iteration. Conditions 1–2 of Lemma 2 clearly hold. Conditions 3–4 are satisfied because of how z is chosen. Condition 5 holds, because otherwise $z.marks[k]$ should have been changed to *in-answer*.

By Lemma 2, a suffix of $\bar{p}[v,k]$ is on Q when iteration i begins, contradicting our assumption that Q is empty. ◀

▶ **Theorem 9.** *GTF is correct. In particular, it finds all and only answers to the query K by increasing height within $2(n+1)^{2n(n+1)}$ iterations of the main loop (line 10), where $n = |V|$.*

Proof. By Lemma 5, the algorithm constructs at most $(n+1)^{2n(n+1)}$ paths. By Proposition 1, a path can be inserted into the queue Q at most twice. Thus, the algorithm terminates after at most $2(n+1)^{2n(n+1)}$ iterations of the main loop.

By Part 1 of Lemma 6, all the K-roots are discovered. By Lemma 8, no suffix of a simple path from a K-root to a keyword can be frozen upon termination. Clearly, no such suffix can be on Q when the algorithms terminates. Hence, the algorithm constructs all the simple paths from each K-root to every keyword. It thus follows that the algorithm finds all the answers to K. Clearly, the algorithm generates only valid answers to K.

Next, we prove that the answers are produced in the order of increasing height. So, consider answers a_1 and a_2 that are produced in iterations j_1' and j_2, respectively. For the answer a_i $(i = 1, 2)$, let r_i and h_i be its K-root and height, respectively. In addition, let b_i be the best height of r_i $(i = 1, 2)$.

Suppose that $j_1' < j_2$. We have to prove that $h_1 \leq h_2$. By way of contradiction, we assume that $h_1 > h_2$. By the definition of best height, $h_2 \geq b_2$. Hence, $h_1 > b_2$.

Let $p_2[r_2,k]$ be the path of a_2 that is exposed (i.e., removed from Q) in iterations j_2. Suppose that $p_1[r_1,k']$ is a path of a_1, such that $w(p_1[r_1,k']) = h_1$ and $p_1[r_1,k']$ is exposed in the iteration j_1 that is as close to iteration j_1' as possible (among all the paths of a_1 from r_1 to a keyword with a weight equal to h_1). Clearly, $j_1 \leq j_1'$ and hence $j_1 < j_2$.

We now show that $w(p_1[r_1, k']) < h_1$, in contradiction to $w(p_1[r_1, k']) = h_1$. Hence, the claim that $h_1 \leq h_2$ follows. Let i be the iteration that discovers r_2 as a K-root. There are two cases to consider as follows.

Case 1: $i < j_1$. In this case, $i < j_1 < j_2$, since $j_1 < j_2$. By Lemma 7, $w(p_1[r_1, k']) \leq w(p_2[r_2, k])$. (Note that we apply Lemma 7 after replacing v and v' with r_2 and r_1, respectively.) Hence, $w(p_1[r_1, k']) < h_1$, because $w(p_2[r_2, k]) \leq h_2$ follows from the definition of height and we have assumed that $h_1 > h_2$.

Case 2: $j_1 \leq i$. By Part 2 of Lemma 6, $w(p_1[r_1, k']) \leq b_2$. Hence, $w(p_1[r_1, k']) < h_1$, because we have shown earlier that $h_1 > b_2$.

Thus, we have derived a contradiction and, hence, it follows that answers are produced by increasing height. ◀

▶ **Corollary 10.** *The running time of the algorithm GTF is* $O\left(kn(n+1)^{2kn(n+1)+1}\right)$*, where* n *and* k *are the number of nodes in the graph and keywords in the query, respectively.*

Proof. The most expensive operation is a call to **produceAnswers**$(v.paths, p)$. By Lemma 5, there are at most $(n+1)^{2n(n+1)}$ paths. A call to the procedure **produceAnswers**$(v.paths, p)$ considers all combinations of $k-1$ paths plus p. For each combination, all its k paths are traversed in linear time. Thus, the total cost of one call to **produceAnswers**$(v.paths, p)$ is $O\left(kn(n+1)(n+1)^{(k-1)2n(n+1)}\right)$. By Theorem 9, there are at most $2(n+1)^{2n(n+1)}$ iterations. Hence, the running time is $O\left(kn(n+1)^{2kn(n+1)+1}\right)$. ◀

5 Summary of the Experiments

In this section, we summarize our experiments. The full description of the methodology and results is given in Appendix B of [5]. We performed extensive experiments to measure the efficiency of GTF. The experiments were done on the Mondial[3] and DBLP[4] datasets.

To test the effect of freezing, we ran the naive approach (described in Section 3.1) and GTF on both datasets. We measured the running times of both algorithms for generating the top-k answers ($k = 100, 300, 1000$). We discovered that the freezing technique gives an improvement of up to about one order of magnitude. It has a greater effect on Mondial than on DBLP, because the former is highly cyclic and, therefore, has more paths (on average) between a pair of nodes. Freezing has a greater effect on long queries than short ones. This is good, because the bigger the query, the longer it takes to produce its answers. This phenomenon is due to the fact that the average height of answers increases with the number of keywords. Hence, the naive approach has to construct longer (and probably more) paths that do not contribute to answers, whereas GTF avoids most of that work.

In addition, we compared the running times of GTF with those of BANKS [1, 7], BLINKS [6], SPARK [9] and ParLMT [4]. The last one is a parallel implementation of [3]; we used its variant ES (early freezing with single popping) with 8 threads. BANKS has two versions, namely, MI-BkS [1] and BiS [7]. The latter is faster than the former by up to one order of magnitude and we used it for the running-time comparison.

GTF is almost always the best, except in two particular cases. First, when generating $1,000$ answers over Mondial, SPARK is better than GTF by a tiny margin on queries with 9 keywords, but is slower by a factor of two when averaging over all queries. On DBLP, however,

[3] http://www.dbis.informatik.uni-goettingen.de/Mondial/
[4] http://dblp.uni-trier.de/xml/

SPARK is slower than GTF by up to two orders of magnitude. Second, when generating 100 answers over DBLP, BiS is slightly better than GTF on queries with 9 keywords, but is 3.5 times slower when averaging over all queries. On Mondial, however, BiS is slower than GTF by up to one order of magnitude. All in all, BiS is the second best algorithm in most of the cases. The other systems are slower than GTF by one to two orders of magnitude.

Not only is our system faster, it is also increasingly more efficient as either the number of generated answers or the size of the data graph grows. This may seem counterintuitive, because our algorithm is capable of generating all paths (between a node and a keyword) rather than just the minimal one(s). However, our algorithm generates non-minimal paths only when they can potentially contribute to an answer, so it does not waste time on doing useless work. Moreover, if only minimal paths are constructed, then longer ones may be needed in order to produce the same number of answers, thereby causing more work compared with an algorithm that is capable of generating all paths.

GTF does not miss answers (i.e., it is capable of generating all of them). Among the other systems we tested, ParLMT [4] has this property and is theoretically superior to GTF, because it enumerates answers with polynomial delay (in a 2-approximate order of increasing height), whereas the delay of GTF could be exponential. In our experiments, however, ParLMT was slower by two orders of magnitude, even though it is a parallel algorithm (that employed eight cores in our tests). Moreover, on a large dataset, ParLMT ran out of memory when the query had seven keywords. The big practical advantage of GTF over ParLMT is explained as follows. The former constructs paths incrementally whereas the latter (which is based on the Lawler-Murty procedure [8, 10]) has to solve a new optimization problem for each produced answer, which is costly in terms of both time and space.

A critical question is how important it is to have an algorithm that is capable of producing all the answers. We compared our algorithm with BANKS. Its two versions only generate answers consisting of minimal paths and, moreover, those produced by BiS have distinct roots. BiS (which is overall the second most efficient system in our experiments) misses between 81% (on DBLP) to 95% (on Mondial) of the answers among the top-100 generated by GTF. MI-BkS misses much fewer answers, that is, between 1.8% (on DBLP) and 32% (on Mondial), but it is slower than BiS by up to one order of magnitude. For both versions the percentage of misses increases as the number of generated answers grows. This is a valid and significant comparison, because our algorithm generates answers in the same order as BiS and MI-BkS, namely, by increasing height.

6 Conclusions

We presented the GTF algorithm for enumerating, by increasing height, answers to keyword search over data graphs. Our main contribution is the freezing technique for avoiding the construction of (most if not all) non-minimal paths until it is determined that they can reach K-roots (i.e., potentially be parts of answers). Freezing is an intuitive idea, but its incorporation in the GTF algorithm involves subtle details and requires an intricate proof of correctness. In particular, cyclic paths must be constructed, although they are not part of any answer. For efficiency's sake, however, it is essential to limit the creation of cyclic paths as much as possible, which is accomplished by lines 24 and 36 of Figure 6.

Freezing is not merely of theoretical importance. Our extensive experiments (described in Section 5 and Appendix B of [5]) show that freezing increases efficiency by up to about one order of magnitude compared with the naive approach (of Section 3.1) that does not use it.

The experiments of Section 5 and Appendix B of [5] also show that in comparison to other systems, GTF is almost always the best, sometimes by several orders of magnitude.

Moreover, our algorithm is more scalable than other systems. The efficiency of GTF is a significant achievement especially in light of the fact that it is complete (i.e., does not miss answers). Our experiments show that some of the other systems sacrifice completeness for the sake of efficiency. Practically, it means that they generate longer paths resulting in answers that are likely to be less relevant than the missed ones.

The superiority of GTF over ParLMT is an indication that polynomial delay might not be a good yard stick for measuring the practical efficiency of an enumeration algorithm. An important topic for future work is to develop theoretical tools that are more appropriate for predicting the practical efficiency of those algorithms.

References

1. Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, 2002.
2. Joel Coffman and Alfred C. Weaver. An empirical performance evaluation of relational keyword search techniques. *IEEE Trans. Knowl. Data Eng.*, 26(1):30–42, 2014.
3. Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, 2008.
4. Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Optimizing and parallelizing ranked enumeration. *PVLDB*, 2011.
5. Konstantin Golenberg and Yehoshua Sagiv. A practically efficient algorithm for generating answers to keyword search over data graphs. *arXiv*, 2015. URL: `http://arxiv.org/abs/1512.06635`.
6. Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.
7. Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.
8. 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*, 1972.
9. Yi Luo, Wei Wang, Xuemin Lin, Xiaofang Zhou, Jianmin Wang, and Keqiu Li. SPARK2: Top-k keyword query in relational databases. *IEEE Trans. Knowl. Data Eng.*, 2011.
10. K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 1968.
11. Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE*, 2009.