Finding Top-k Min-Cost Connected Trees in Databases

Bolin Ding\(^1\)  Jeffrey Xu Yu\(^1\)  Shan Wang\(^2\)  Lu Qin\(^1\)  Xiao Zhang\(^2\)  Xuemin Lin\(^3\)

\(^1\) The Chinese University of Hong Kong, China, \{blding,yu.luqin\}@se.cuhk.edu.hk
\(^2\) Key Laboratory of Data Engineering and Knowledge Engineering, MOE of China, Renmin University of China, China, \{swang,zhangxiao\}@ruc.edu.cn
\(^3\) The University of New South Wales & NICTA, Australia, lxue@ece.unsw.edu.au

Abstract

It is widely realized that the integration of database and information retrieval techniques will provide users with a wide range of high quality services. In this paper, we study processing an \(l\)-keyword query, \(p_1,p_2,\cdots,p_l\), against a relational database which can be modeled as a weighted graph, \(G(V,E)\). Here \(V\) is a set of nodes (tuples) and \(E\) is a set of edges representing foreign key references between tuples. Let \(V_i \subseteq V\) be a set of nodes that contain the keyword \(p_i\). We study finding top-k minimum cost connected trees that contain at least one node in every subset \(V_i\), and denote our problem as GST-k. When \(k = 1\), it is known as a minimum cost group Steiner tree problem which is NP-Complete. We observe that the number of keywords, \(l\), is small, and propose a novel parameterized solution, with \(l\) as a parameter, to find the optimal GST-1, in time complexity \(O(3^n + 2^l(l + \log n)n + m)\), where \(n\) and \(m\) are the numbers of nodes and edges in graph \(G\). Our solution can handle graphs with a large number of nodes. Our GST-1 solution can be easily extended to support GST-k, which outperforms the existing GST-k solutions over both weighted undirected/directed graphs. We conducted extensive experimental studies, and report our finding.

1 Introduction

Over decades, sophisticated database techniques have been developed to provide users with effective and efficient ways to access structural data managed by DBMS using SQL. At the same time, due to the rapid growth of hypertext data available on Web, advanced information retrieval techniques have been developed to allow users to use keyword queries (a set of keywords) to access unstructured data that users are most likely interested in, using ranking techniques. It is widely realized that the integration of information retrieval (IR) and database (DB) techniques will provide users with a wide range of high quality services [3, 2, 10]. The recent studies on supporting IR style queries in RDBMS include DBXPLore [1], IR-Style [17], DISCOVER [18], ObjectRank [4], BANKS-I [6], and BANKS-II [25]. All consider a RDBMS as a graph where nodes represent tuples/relations and edges represent foreign key references among tuples cross relations. And with a keyword query, users can find the connections among the tuples stored in relations without the need of knowing the relational schema imposed by RDBMS. We show a motivation example.

Example 1.1: Consider a database for citations among research papers written by authors. Figure 1 (a) shows such a database with 4 tables: Author, Paper, Paper-Author, and Citation. The Author table is with an author-id (AID) and an author name (Name). The Paper table is with a paper-id (PID) and a title (Title). The Paper-Author table specifies the relationship between a paper and an author using paper-id and author-id. PID and AID in the table Paper-Author are foreign key references to PID and AID in the Paper table and the Author table, respectively. The Citation table specifies the citation between two papers, and both attributes, Cite and Cited, are foreign key references to PID in the table Paper.

Weighted Database Graph: This database can be represented as a database graph in Figure 1 (b). Nodes are tuple identifiers. Edges represent foreign key references between two tuples. Nodes and edges are weighted. An edge is more important if it has a smaller weight. A node that has many links with others has relative small possibility of having a close relationship to any of them [4], and thus edges incident on it have large weights. For simplicity, we only show edge weights in Figure 1 (b).

Query/Answer: Given a 4-keyword query: Keyword \((p_1)\), Query \((p_2)\), DB \((p_3)\), and Jim \((p_4)\). It tries to find the possible relationship (foreign key references) among the 4 keywords in the database, and consequently the database graph. Figure 1 (c) and (d) show 2 possible connected trees, Tree-1 and Tree-2, as answers, containing all 4 keywords. Tree-1 shows that Jim \((a_1)\) writes a paper, \(t_2\), which is cited by two papers, \(t_1\) and \(t_3\). Here, \(t_1\) contains keyword \(p_1\), and \(t_3\) contains keywords \(p_2\) and \(p_3\). Tree-2 shows that Jim \((a_1)\) writes a paper \(t_2\) which is cited by \(t_3\) with keywords \(p_2\) and \(p_3\), and the author of \(t_3\) \(a_2\), writes another paper \(t_5\) with keywords, \(p_1\) and \(p_2\). The total weights on the 2 trees are 10.8 and 15.6. The answer with a smaller total weight is ranked higher, because it represents stronger
the existence of a node/edge weighted graph. Approaches like other tuple-based approaches, we assume the existence of a materialized graph is small, which we will discuss later.

In a graph, node/edge weights can be assigned using [7, 4]. Because of [6], and [4], we cannot fully make use of node/edge weights at tuple level. Processing a keyword query by utilizing the weights associated with nodes (tuples) and edges (foreign-key reference) in a graph. Node/edge weights can be assigned using [7, 4]. With the weighted graph, the tuple-based approaches find top-k minimum cost connections (trees) among tuples for a keyword query. Such systems include DBXPlore [1], IR-Style [17], and DISCOVER [18]. On the other hand, the tuple-based approaches aim at processing a keyword query by utilizing the weights associated with nodes (tuples) and edges (foreign-key reference) in a graph. Node/edge weights can be assigned using [7, 4]. With the weighted graph, the tuple-based approaches find top-k minimum cost connections (trees) among tuples for a keyword query. Such systems include RIU [22], BANKS-I, [6], and BANKS-II [25].

It is worth of noting that the relation-based approaches cannot fully make use of node/edge weights at tuple level because of SQL. To handle node/edge weights, the tuple-based approaches assume the existence of a materialized graph over the database. The memory-consumption of this materialized graph is small, which we will discuss later.

In this paper, we focus ourselves on the tuple-based approaches. Like other tuple-based approaches, we assume the existence of a node/edge weighted graph $G(V,E)$ in the main memory. Given a $l$-keyword query, $p_1, p_2, \ldots, p_l$, we study finding top-k minimum cost connected trees that contain all $l$ keywords at least once in the database graph. We use GST-1 to denote the minimum cost connected tree, and GST-k to denote the top-k minimum cost connected trees. This problem is also well known as finding (top-k) minimum cost group Steiner tree.

**Contributions of this paper:** (1) We identify the characteristics of the group Steiner tree problem, for $l$-keyword query, over the database graph $G$, where the numbers of nodes and edges are $n$ and $m$. The characteristics are: $l$ is a small number, and $G$ is a sparse graph with a large number of nodes. In brief, $l \ll \log n$, and $m \ll n^2$. (2) We propose a parameterized solution, with $l$ as a parameter. We can obtain the optimal $GST-I$ with the worst-case time complexity $O(3^n n^2((l + \log n) n + m))$ and space complexity of $O(2^n n)$. Note: this parameterized algorithm works efficiently on the condition that $l$ is small, and we are not solving the problem in a general setting where $l$ can be any large. (3) We extend our solution to find $GST-k$ in a progress manner. That is, we needn’t compute/sort all group Steiner trees and then find $GST-k$. Our approach outperforms existing approaches with high quality and efficiency. (4) Our approach supports undirected graphs as well as directed graphs, with node/edge weights. We can handle the class of additive cost functions to score group Steiner trees.

**Organization:** Section 2 gives problem statement and discusses the characteristics of the problem. Section 3 reviews the related work. Section 4 provides a new parameterized solution for $GST-I$. It is achieved by a dynamic programming with a best-first strategy. We will also discuss how to extend it for $GST-k$ solutions and show the advantages of our algorithms. Discussions on undirected/directed graphs, weight schema supported, and graph maintenance will also be given at the end of this section. Experimental studies are given in Section 5. Finally, Section 6 concludes the paper.

## 2 Problem Statement

Given an RDBMS, $DB$, upon a relational schema $R$ with foreign key references. We define a weighted database graph, $G(V,E)$, where $V$ is the set of tuples (nodes) in $DB$, and $E$ is the set of edges. An edge, $(u, v) \in E$, represents a foreign key reference between two nodes (tuples), $u$ and $v$, if $u$ has a foreign key matches the primary key attributes of $v$, or $v$ has a foreign key matches the primary key attributes of $u$. The graph is an undirected graph if the direction, either from foreign key to primary key or from primary key to foreign key, is not the main concern. Otherwise, the graph is a directed graph where there are two edges, $(u, v)$ and $(v, u)$, in $E$ such that $(u, v) \neq (v, u)$. Graph $G(V,E)$ is weighted, with a node-weight $w_v(v)$ for every node $v \in V$ and an edge-weight $w_e(e)$ for every edge $e \in E$, both of which are non-negative numbers. Below, we use $V(G)$ and $E(G)$ to denote the set of nodes and the set of edges of the graph $G$, respectively. Let $n = |V(G)|$ and $m = |E(G)|$.

Consider a $l$-keyword query, i.e., a set of keywords, $p_1, \ldots, p_l$, against a database graph $G$. There is a set of nodes, denoted $V_l (\subseteq V(G))$, that contain the keyword $p_i$, for
$i = 1, \cdots, l$. We call $V_i$ a group of the keyword $p_i$ or simply a group. Note: a node is a tuple with several attributes, a node contains a keyword if the keyword appears in any of the attributes of the corresponding tuple, and a node may contain several keywords. There are $l$ groups, $V_1, \cdots, V_l$. All groups can be obtained with either the symbolizable techniques [1] or the full text index techniques [18].

**Minimum group Steiner tree problem (GST-1):** Given a $l$-keyword query, to find the minimum cost connected tree $T$, such that $V(T) \cap V_i \neq \emptyset$ for $i = 1, \cdots, l$. For brevity, the cost of such a tree $T$, is given below.

$$s(T) = \sum_{e \in E(T)} w_e(e)$$ \hfill (1)

Here, let $N(v)$ be a set of neighbors of $v$, and $|N(v)|$ be the size of $N(v)$. We use $w_e$ in Eq. (2) to weight edges in an undirected graph.

$$w_e((v, u)) = \log_2(1 + \max(|N(v)|, |N(u)|))$$ \hfill (2)

Note: $w_e((v, u)) = w_e((u, v))$. We use Eq. (3) and Eq. (4), which were used in [25], to weight edges in a directed graph. In detail, for a foreign key reference from $u$ to $v$, the edge weight for $(u, v)$ is given Eq. (3), and the edge weight for $(v, u)$ is given Eq. (4).

$$w_e((u, v)) = 1$$ \hfill (3)

$$w_e((v, u)) = \log_2(1 + N_{in}(v))$$ \hfill (4)

where $N_{in}(v)$ is the number of nodes that reference to $v$. We will address how to enhance Eq.(1) to handle node weights as well as edge weights later in Section 4.4.

The semantic captured by Eq.(2)-(4) is that: If a node has more neighbors, an edge that is incident on it reflects a weaker relationship between tuples (see Example 1.1).

**Top-k group Steiner tree problem (GST-k):** Given a $l$-keyword query, to find the top-k minimum cost group Steiner trees, $T_1, T_2, \cdots, T_k$, ranked with a cost function $s$, such that $s(T_1) \leq s(T_2) \leq \cdots \leq s(T_k)$. Example 1.1 shows an example of GST-2.

In this paper, we study finding GST-1 and GST-k, for a $l$-keyword query, upon a database graph $G$, which is constructed from $r$ tables, $R_1, R_2, \cdots, R_r$, in the underneath $DB$. Let $n$ and $m$ be the numbers of nodes and edges in $G$. The characteristics of our problem are given in Remark 2.1.

**Remark 2.1:** 1) $n$ is large, because the number of tuples in the corresponding database is large. 2) $l$ is small ($l \ll \log n$), say $l = 6$, because users do not usually use many keywords to query. 3) $m \ll n^2$ (database graph $G$ is sparse). We explain why $G$ is sparse below. Suppose there is a foreign key reference from relation $R_a$ (with foreign key) to $R_b$ (with primary key). A tuple in $R_a$ can reference to at most one tuple in relation $R_b$, and can reference to at most $r$ tuples if the database has $r$ relations in total. Therefore, $m \leq r \cdot n$, since there are $n$ nodes in total. Note: $r \ll n$.

### 3 Existing Solutions

In the literature, the group Steiner tree problem, or GST-1, is proved to be NP-Complete [23] by reducing it to minimum set cover problem. Here, the number of groups, $l$, and the graph $G(V, E)$ can be any large. The existing reported studies aim at approximating the minimum cost of GST-1 within bounded performance ratio. Let $T_o$ be a tree found by an algorithm, and $T_n$ be the optimal. The performance ratio is defined as $s(T_o)/s(T_n) (\geq 1)$. It is also known that GST-1 is inapproximable within a constant performance ratio by a polynomial algorithm [20]. The lower bound of performance ratio is $O(\ln l)$ [14].

Table 1 shows performance ratio and time complexity for GST-1 algorithms. Note $O(\alpha)$ is the time needed to find all-pairs shortest paths in graph. We discuss them in brief below. The approximation algorithms can be categorized into three types: spanning and cleanup, $d$-star Steiner tree, and $i$-level tree.

**Spanning and Cleanup:** This technique was inspired by the minimum spanning tree algorithm. It spans a tree $T_o$ from an arbitrary vertex $v$ in one group step-by-step until it covers at least one node in every group, and then achieves a low score tree by cleaning up the redundant vertices.

**$d$-Star Tree or $i$-Level Tree:** The two methods are based on a metric closure, $G^+$, which is a complete graph of a graph $G$ where $V(G^+) = V(G)$, and the weight of each edge $(u, v)$ in $G^+$ is equal to the weight-sum on the shortest path from $u$ to $v$ in $G$. A $d$-star tree in graph $G^+$ is a rooted subtree with depth at most $d$. A $d$-star Steiner tree is a $d$-star tree containing at least one node from each group $V_i$ ($1 \leq i \leq l$). In [16], Helvig et al. consider a general case, and use optimal $d$-star Steiner tree in $G^+$ to approximate Steiner tree in $G$. In [5], Bateman et al. consider a special case $d = 2$. In [8], Charikar et al. proposed an approximation algorithm based on $i$-level tree, which is a similar concept in a directed graph as $d$-star tree.

Above, we discussed the solutions to GST-1. As also pointed in [22], all the algorithms [19, 5, 16, 8, 23] can not be directly used to compute GST-k, because they all need to compute/ sort all group Steiner trees, in order to find the minimum cost GST-k. They cannot terminate any early and report GST-k in a progress manner [22]. Below, we introduce two existing approaches for GST-k. RIU [22], and BANKS-I [6] and its successor BANKS-II [25].

**RIU (Retrieve Information Unit [22])** adopted the spanning and cleanup strategy as given in [19] to find GST-k incrementally. [22] proposed two spanning strategies, namely, minimum edge-based strategy and balanced MST strategy. We denote the former and the latter as RIU-E and RIU-T, respectively. RIU-E achieves high efficiency but with worse performance ratio, whereas RIU-T achieves better performance ratio at the expenses of low efficiency.
Methodology

Time Complexity

Performance Ratio

GST-1 Algorithms  Methodology  Performance Ratio  Time Complexity  Time Complexity with Fixed $l$

Reisch and Waldmayer [25]  Spanning and Cleanup  unbounded  $O(\cdot(n + n \log n))$  $O(n + n \log n)$

Batean et al. [19]  Spanning and Cleanup  $O(1)$  $O(\cdot n \cdot (n + n \log n))$  $O(n \cdot (n + n \log n))$

Batov et al. [5]  2-Star Tree  $O(1 + \ln \phi \cdot \sqrt{l})$  $O(\cdot n \cdot \log l)$  $O(\cdot n \cdot l)$

Helv et al. [16]  d-Star Tree  $O((2d - 2 + (m(2d))^2) \cdot \sqrt{l})$  $O(\cdot n \cdot (n \cdot l)^3)$  $O(\cdot n \cdot n^3)$

Chandar et al. [8]  d-Level Tree  $O((s - 1)^2 \cdot l^2)$  $O(n^2 \cdot l^2)$  $O(n^2 \cdot l^2)$

Ri [22]  Spanning and Cleanup  $O(1)$  $O(n \cdot (n + n \log n))$  $O(n \cdot (n + n \log n))$

BANKS-I [6], BANKS-II [25]  1-Star Tree  $O(1)$  $O((3^i + 2 \cdot (l + \log n) + m))$  $O(n \cdot \log n + m)$

This paper  Dynamic Programming  $1$ (optimal)  $O((3^i + 2 \cdot (l + \log n) + m))$  $O(n \cdot \log n + m)$

Table 1. Approximation and Time Complexity of GST-1 Solutions

BANKS-I [6] and BANKS-II [25] studied GST-k over a weighted directed graph. The techniques used can support undirected graphs as well. Both adopted the d-star tree strategy [16]. In brief, the GST-I found by BANKS-I and BANKS-II is a tree constructed by combining shortest-paths from the leaves to the root in $G$, each of which is an edge in metric closure $G^+$. So they actually used the 1-star tree on $G^+$ to approximate GST-I. Moreover, they used the shortest path strategy to approximate GST-k under a hypothesis that GST-k are also obtained by combining shortest-paths form leaves to the root. The main difference between BANKS-I and BANKS-II is that BANKS-I uses backward expanding search [6] and BANKS-II uses bidirectional expansion [25] to improve the efficiency. The bidirectional expansion technique reduces the size search space. But in many cases, BANKS-II produces a low-quality performance ratio than that of BANKS-I, because bidirectional expansion may miss some shortest paths.

Linear programming: [9, 15] studied linear programming for approximate GST-I, which is difficult to be used for handling large data graphs.

It is important to note that all the bounds about performance ratio listed in Table 1 are for GST-I. It is difficult to determine the theoretical bounds of performance ratio for the $k$-th minimum cost group Steiner tree when $k > 1$.

4 A New Parameterized Solution

All the reported works deal with GST-I (or GST-k) for an approximate solution in a general setting where all $n$, $m$ and $l$ can be any large. Let’s reconsider the characteristics of the problem of $l$-keyword query processing: $n$ is large, $m \ll n^2$, and $l$ is small ($l \ll \log n$) (Remark 2.1). So in the time complexity given in Table 1, the $l$ components are less important ($l \ll \log n$), but the $n$ components are very important, even more important than the $m$ components ($m \ll n^2$). Regarding the $n$ components (for fixed $l$), the time complexity of all the algorithms [19, 5, 16, 8, 9, 15] is at least $O(n^2)$, which makes them difficult to be efficiently applied to a large graph $G$ with millions of nodes. The exception is [23], but its performance ratio is unbounded. In this section, we propose a parameterized algorithm using $l$ as a parameter. We find the optimal solution (performance ratio=1) of GST-I in time $O(3^i n + 2((l + \log n)n + m))$, treating input size, $l$, $m$ and $n$, differently. Our solution can be easily extended to solve GST-k.

The theory of parameterized complexity is formally presented in [12]. The fixed-parameter tractability [12] for a problem of size $n$, with respect to a parameter $l$, means it can be solved in time $O(f(l)g(n))$, where $f(l)$ can be any function (like $2^l$ or $l^l$) but $g(n)$ must be polynomial. Such an algorithm is called parameterized algorithm. In this paper, we treat the number of keywords, $l$, as a parameter. To our best knowledge, the minimum Steiner tree problem, as a special case of the group Steiner tree problem, i.e. $|V| = 1, 1 \leq i \leq l$, was proved to be fixed-parameter tractable in [13]. Our work presented here is the first to prove that the group Steiner tree problem is fixed-parameter tractable.

Our parameterized algorithm is based on dynamic programming because of GST-I’s optimal substructure [11].

Given a $l$ keyword query against a database graph $G$. Let $P$ be the entire set of keywords $P = \{p_1, p_2, \ldots, p_l\}$. We use $p, p_1$ and $p_2$ to denote a non-empty subset of $P$ such as $p, p_1, p_2 \subseteq P$.

4.1 Naive Dynamic Programming

We give a dynamic programming solution by taking the heights, $h$, of trees as stages, and find the optimal GST-I by expanding the trees with heights $h = 0, 1, 2, \ldots$, until the GST-I is found. The optimal GST-I for a subset of keywords $p$, with a certain height $h$, is found from the optimal solutions to GST-I, for subsets of keywords $p_1$, such as $p_1 \subseteq p$, with heights $\leq h$.

Let $T(v, p, h)$ be a tree with minimum cost rooted at node $v$, with height $\leq h$, containing a non-empty set of keywords $p$. Every single node $v$ in $G$ containing a non-empty set of keywords, $p (\subseteq P)$, is a rooted tree with zero height, $h = 0$. We denote it $T(v, p, 0)$. Such a tree does not have any edges, and therefore the cost of the tree $T(v, p, 0)$ is zero (refer to Eq. (1)) as given below.

$$T(v, p, 0) = 0$$

(5)

Here, the left side is the tree, and the right side is the cost of the tree that appears on the left. Note $T(v, 0, 0) = \infty$. In general, the (minimum) cost of $T(v, p, h)$, for any $h > 0$,
is given below in Eq. (6).

\[ T(v, p, h) = \min \{ T_g(v, p, h), T_m(v, p, h), T(v, p, h - 1) \} \tag{6} \]

where

\[ T_g(v, p, h) = \min_{u \in N(v)} \{ (v, u) \oplus T(u, p, h - 1) \} \tag{7} \]
\[ T_m(v, p_1 \cup p_2, h) = \min_{p_1, p_2} \{ T(v, p_1, h) \oplus T(v, p_2, h) \} \tag{8} \]

Here \( \oplus \) is an operation to merge two trees into a new tree, and \( N(v) \) is a set of neighbors of \( v \) such as \( N(v) = \{ u \mid (v, u) \in E(G) \} \) in the database graph \( G \). As a special case, if \( v \) contains some additional keywords \( p' \), then the left side of Eq. (7) should be \( T_g(v, p \cup p', h) \). We explain Eq. (6)-(8) below.

In Eq. (6), \( T(v, p, h) \) is constructed from either \( T_g(v, p, h) \) or \( T_m(v, p, h) \). The last term in Eq. (6) is to explicitly state that \( T(v, p, h) \) is the tree with the minimum cost among all \( T(v, p, h') \) for \( h' \leq h \). If the cost of \( T_g(v, p, h) \) is minimum, \( T(v, p, h) \) is constructed as \( T_g(v, p, h) \). Otherwise, \( T(v, p, h) \) is constructed as \( T_m(v, p, h) \). \( T_g(v, p, h) \) is for a case, called tree grow, as illustrated in Figure 2 (a), where the degree of the root is 1; \( T_m(v, p, h) \) is for another case, called tree merge, as illustrated in Figure 2 (b), where a tree is constructed from the merge of other two trees. Note there are possible different trees rooted at \( v \) which can be grown (or merged) to \( T(v, p, h) \) (or \( T(v, p_1 \cup p_2, h) \)). Eq. (7)-(8) request that the cost, \( s(T(v, p, h - 1)) + w_{e_{v}}(v, u) \) or \( s(T(v, p_1, h)) + s(T(v, p_2, h)) \), is minimized if there are alternatives to construct the same tree.

Theorem 4.1: Given an edge-weighted undirected graph \( G \), and a set of \( l \) keywords, \( P = \{ p_1, p_2, \ldots, p_l \} \). Let \( V_i \subseteq V \) be a group where \( v \in V_i \) contains \( p_i \), for \( i = 1, \ldots, l \). The optimal GST-I can be computed using Eq. (5) – Eq. (8) by examining heights \( h = 0, 1, 2, \ldots, \text{dia}(G) \), for all \( v \in V(G) \) and \( p \subseteq P \). (dia(G) is defined as \( G \)'s diameter).

Theorem 4.1 can be proved by the induction on \( h \) and \(|p|\). The optimal substructure holds as specified in Eq. (5)-(8). One more point need be clarified: Consider Eq. (7) using Figure 2(a). The node \( v \) may appear in the tree \( T(u, p, h - 1) \), and thus \((v, u) \oplus T(u, p, h - 1)\) contains \( v \) twice in the computing process. But if so, there definitely exists a tree rooted at \( v \) which contains \( v \) only once with a smaller cost. Therefore, a node appears only once in the resulting tree \( T(v, p, h) \). It is similar for Eq. (8) and Figure 2(b).

The naive dynamic programming algorithm, for GST-I, is outlined in Algorithm 1, called DPH-I. It is a straightforward implementation of Eq. (5)-(8).

Algorithm 1 DPH-I

\textbf{input:} database graph \( G \), the set of keywords \( P \), and groups \( V_1, \ldots, V_l \)

\textbf{output:} GST-I

1: let \( H \) be the graph diameter for \( G \);
2: \( h \leftarrow 0 \);
3: compute \( T(v, p, 0) \) for every \( v \in V(G) \) that contains any keywords in \( P \) (Eq. (5));
4: while \( h < H \) do
5: \( h \leftarrow h + 1 \);
6: for each \( v \in V(G) \) do
7: \( T \leftarrow \{ T(v, p, h - 1) \} \);
8: for all possible \( p \), compute \( T_g(v, p, h) \) from its neighbors (Eq. (7)), and insert them into \( T \);
9: compute all possible \( T_m(v, p_1 \cup p_2, h) \) from two trees in \( T \) (Eq. (8)); and insert the new trees into \( T \);
10: \( T(v, p, h) \leftarrow \) minimum cost tree rooted at \( v \) and containing \( p \) in \( T \) (Eq. (6));
11: sort the costs for all \( T(v, P, h) \), for \( v \in V(G) \);
12: return \( T(v, P, h) \) with minimum cost;

Example 4.1: A database graph with edge weights is shown in Figure 3 (a). Given a 4-keyword query: \( p_1, p_2, p_3, p_4 \). Suppose the four nodes, \( v_1, v_2, v_3, \) and \( v_4 \), contain \( p_1, p_2, p_3, \) and \( p_4, \) respectively. The optimal GST-I with cost 14 is shown in Figure 3 (b). Figure 4 (a)-(c) show the intermediate results based on DPH-I when \( h = 0, 1, 2 \), and Figure 4 (d) shows the final result. Note: by employing 1-star tree technique, BANKS-II can only find the tree \((v_9(v_7(v_1v_2)))(v_8(v_3v_4))\) with cost 18 as GST-I answer.

\[ \square \]

Time/Space Complexity: A straightforward implementation of DPH-I consumes time \( O(3^l n^{2} + 2^l mn) \) and space \( O(2^l n^2) \). To reduce the time complexity to promised \( O(3^l n + 2^l ((l + \log n)n + m)) \) and to reduce the space complexity to \( O(2^l n) \), we propose DPBF-I in Section 4.2, based on DPH-I.
4.2 A New Best-First DP Solution

The algorithm DPH-1 (Algorithm 1) shows the main idea of dynamic programming algorithm with which the optimal GST-1 can be computed. The main problem of DPH-1 is that it cannot determine whether the current smallest cost tree, $T(v, P, h)$, containing all keywords $P$, is the optimal, until $h = H = \text{dia}(G) = \mathcal{O}(n)$.

In this section, we present a novel dynamic programming solution with a best-first strategy. First, it does not rely on parameter height $h$. Second, it ensures the optimal GST-1 is the first $T(v, P)$ found containing all keywords. In other words, with the best-first strategy, the algorithm can terminate when it finds a connected tree containing all keywords.

Eq. (5) – Eq. (8), with height $h$, are rewritten to Eq. (9) – Eq. (12), without height $h$, respectively. In brief, $T(v, p)$ is a tree with minimum cost, rooted at $v$, containing a set of keywords $p \subseteq P$. Below, Eq. (9) shows that the primitive trees, $T(v, p)$, which is single node tree, rooted at $v$, and contains keyword set $p$ in $v$, are with a zero cost.

$$T(v, p) = 0 \quad (9)$$

Note $T(v, \emptyset) = \infty$. Like Eq. (6), Eq. (10) shows the general case for a tree with more than one nodes.

$$T(v, p) = \min(T_v, T_p, T_m(v, p)) \quad (10)$$

where

$$T_v = \min_{u \in N(v)} \{(u, v) \oplus T(u, p)\} \quad (11)$$

and

$$T_m(v, p, p_1 \cup p_2) = \min_{p_1, p_2 \in \emptyset} \{T(v, p_1) \oplus T(v, p_2)\} \quad (12)$$

We omit further explanation because they share high similarity with Eq. (5)-(8). The only difference is that they do not specify height $h$. Again, we can prove the following theorem by the induction on $|p|$. The optimal substructure holds here too.

**Theorem 4.2:** Given an edge-weighted undirected graph $G$, and a set of $l$ keywords, $P = \{p_1, p_2, \cdots, p_l\}$. Let $V_i \subseteq V$ be a group where $v \in V_i$ contains $p_i$, for $i = 1, \cdots, l$. The optimal GST-1 can be computed using Eq. (9) – Eq. (12) by examining for all $v \in V(G)$ and $p \subseteq P$. $\square$

**Algorithm 2 DPBF-1**

**Input:** database graph $G$, the set of keywords $P$, and groups $V_1, \cdots, V_l$

**Output:** GST-1

1. Let $Q_T$ be a priority queue sorted in the increasing order of costs of trees;
2. $Q_T \leftarrow \emptyset$;
3. for each $v \in V(G)$ do
4. if $v$ contains keywords $p$ then
5. enqueue $T(v, p)$ into $Q_T$;
6. while $Q_T \neq \emptyset$ do
7. dequeue $Q_T$ to $T(v, p)$;
8. return $T(v, p)$ if $p = P$;
9. for each $u \in N(v)$ do
10. if $T(v, p) \oplus (v, u) < T(u, p)$ then
11. $T(u, p) \leftarrow T(v, p) \oplus (v, u)$;
12. update $Q_T$ with the new $T(u, p)$;
13. $p_1 \leftarrow p$;
14. for each $p_2 \in p_1$ s.t. $p_1 \cap p_2 = \emptyset$ do
15. if $T(v, p_1) \oplus T(v, p_2) < T(v, p_1 \cup p_2)$ then
16. $T(v, p_1 \cup p_2) \leftarrow T(v, p_1) \oplus T(v, p_2)$;
17. update $Q_T$ with the new $T(v, p_1 \cup p_2)$;

4.2.1 Efficient Algorithm

Based on (9)-(12), we outline the best-first strategy dynamic programming algorithm, called DPBF-1, in Algorithm 2.

In DPBF-1, for simplicity and brevity, we use $T(v, p)$ for the tree structure and its cost. Recall in Eq. (9) – Eq. (12), the left side is the tree whereas the right side is the cost of the tree. We will also make it clear in due course.

DPBF-1 maintains trees in a priority queue $Q_T$, by the increasing order of costs of trees. The smallest cost tree is maintained at the top of the queue $Q_T$. The queue is manipulated with three operators: Enqueue, Dequeue, and Update. Enqueue inserts a tree $T(v, p)$ into the queue $Q_T$ and $Q_T$ is updated to maintain the increasing order of costs of trees. Dequeue removes the top tree $T(v, p)$ in $Q_T$. Update operation first enqueues $T(v, p)$ if it does not exist in $Q_T$, and update $Q_T$ to maintain the increasing order of costs.

**Remark 4.3:** Let $T(v, p)$ be the tree at the top of $Q_T$. It is important to know that $T(v, p)$ is with the minimum cost among all trees rooted at $v$, containing the same set of keywords $p$. Because tree grow and tree merge can only get trees with larger costs. Eq. (10) is ensured by $Q_T$. $\square$

**Initialization and Outline:** In DPBF-1, it first initializes $Q_T$ to be empty (line 2). In line 3-5, it enqueues $T(v, p)$ into $Q_T$ if node $v$ contains a subset of keywords $p$ ($\subseteq P$) (Eq. (9)). While the queue $Q_T$ is non-empty, in line 6-17, the algorithm repeats to dequeue/enqueue in the attempt to make all trees grow/merge individually to reach GST-1. It dequeues the top tree $T(v, p)$ from $Q_T$, which is with the smallest cost in all trees in $Q_T$. Note: $T(v, p)$ is rooted at
and tries to reduce the cost of words, containing different subsets of keywords, in line 13-17. There are many trees rooted at the same node when \( u \) may or may not contain keywords. First consider the case when \( u \) does not contain any keywords: In line 10, it checks if the tree \( T(u, p) \) may or may not exist in \( Q_T \). If not, \( T(u, p) = T(v, p) \oplus (u, v) \) is enqueued into \( Q_T \). Then \( Q_T \) will be updated. In both cases, \( Q_T \) ensures the increasing order of costs. The case when \( u \) does contain some keywords \( p' \) is similar. The line 9-12 handles the case of tree grow (Eq. (11)).

**Tree Merge:** The case of tree merge (Eq. (12)) is handled in line 13-17. There are many trees rooted at the same node \( v \) containing different subsets of keywords, \( p_1, p_2, \) the algorithm considers every possible disjoint pair of \( p_1 \) and \( p_2 \), and tries to reduce the cost of \( T(v, p_1 \cup p_2) \). In line 15, it checks if the tree \( T(v, p_1) \oplus T(v, p_2) \) has a smaller cost than \( T(v, p_1 \cup p_2) \). If yes, the tree \( T(v, p_1 \cup p_2) \) may or may not exist in \( Q_T \). If not, \( T(v, p_1) \oplus T(v, p_2) \) is enqueued into \( Q_T \). Then \( Q_T \) will be updated. In both cases, \( Q_T \) ensures the increasing order of costs.

**Example 4.2:** We explain \( DPBF-1 \) (Algorithm 2) with the same database graph in Figure 3 and keywords as Example 4.1. Recall the a 4-keyword query \( \{p_1, p_2, p_3, p_4\} \). And 4 nodes, \( v_1, v_2, v_3, \) and \( v_4 \), contain the four distinctive keywords, \( p_1, p_2, p_3, \) and \( p_4 \), respectively.

As shown in Figure 5 (a), after line 3-5 of \( DPBF-1 \), \( Q_T \) maintains four trees, \( T(v_1, \{p_1\}), T(v_2, \{p_2\}), T(v_3, \{p_3\}), \) and \( T(v_4, \{p_4\}) \). Their costs are zero, because they do not have edges (Eq. (9)). Figure 5 (b) shows \( Q_T \) after the first 4 iterations of the while statement in \( DPBF-1 \). Here, all 4 trees in Figure 5 (a) are dequeued, and 8 new trees are enqueued into \( Q_T \) based on the case of tree-grow. Their costs are 1, 1, 1, 1, 4, 4, 4, and 4. Figure 5 (c) shows the first two trees of \( Q_T \) after the next 4 iterations of the while statement. They are enqueued into \( Q_T \) based on the case of tree merge, after the first 4 trees in \( Q_T \) (Figure 5 (b)) are dequeued. The costs of the trees in Figure 5 (c) are both 2, so they are ranked as the first two in \( Q_T \). Figure 5 (d) shows the first 6 trees in \( Q_T \) after the next 2 iterations. Here, the first 4 trees in Figure 5 (d) are the 5-th to 8-th trees in Figure 5 (b), and the last trees in Figure 5 (d) are newly constructed from the 2 trees in 5 (c) based on the case of tree grow. Figure 5 (e) shows the trees constructed from the first 4 trees in \( Q_T \) (Figure 5 (d)) in the next 4 iterations based on the case of tree merge. But they are not enqueued into \( Q_T \), because they have higher costs than the 5-th and 6-th trees in Figure 5 (d). Note: they share the same roots and contain the same keywords. Figure 5 (f)-(g) show the rest iterations based on tree grow and tree merge, respectively. The optimal \( GST-I \) is shown in Figure 5 (g).

### 4.2.2 Time/Space Complexities

We analyze the complexity of \( DPBF-1 \) in this subsection.

**Time Complexity:** Let \( T(v, p) \) be the minimum cost for a tree rooted at every \( v \in V(G) \) containing a subset of keywords, \( p \subseteq P \) where \( l = |P| \). There are totally \( n \) nodes, and \( 2^l \) subsets of \( P \). So the length of \( Q_T \) will be at most \( 2^n \). Note: any \( T(v, p) \) will be enqueued/dequeued into/from \( Q_T \) at most once in \( DPBF-1 \). With Fibonacci Heap [11], the cost for enqueue/update and dequeue are \( O(1) \) and \( O(\log 2^n) \), respectively. The total cost for the dequeue is \( O(2^{2n}(l + \log n)) \), for all \( 2^n \) number of \( T(v, p) \).’s.

Eq. (11) is computed in line 9-12 to minimize \( T(u, p) \), where \( u \in N(v) \), using the information of \( T(v, p) \). The total number of possible \( u \) is bounded by \( O(|N(v)|) \) where \( |N(v)| \) is the number of neighbors of \( v \) (line 9). The total number of comparisons in line 10 is bounded by \( O(2^n \sum_{v \in V} |N(v)|) = O(2^nm) \). \( Q_T \) need be updated (in time \( O(1) \)), if a smaller cost for \( T(u, p) \) is found in line 10. So the total time needed for line 9-12 is \( O(2^nm) \).

Eq. (12) is computed in line 13-17 to minimize \( T(u, p_1 \cup p_2) \). For every pair of non-empty disjoint \( p_1 \) and \( p_2 \), it enumerates \( T(v, p_2) \). If a lower cost of \( T(v, p_1 \cup p_2) \) is found (line 15), \( Q_T \) is updated. The total number of possible \( p_2 \) is bounded by \( O(2^{l-|p_1|}) \), so the number of comparisons in line 15 is bounded by \( O(n \sum_{i=0}^{l-1} \binom{l}{i} 2^{l-i} = O(3^ln) \). Because the time needed by an update of \( Q_T \) is
Space Complexity: $T(v, p)$ represents a subtree in $G$. But, we do not need to store the whole tree in memory. We only need to record the edge $(v, u)$ from which $T_{p}(v, p)$ is constructed, and record $p_{1}$ and $p_{2}$ with which $T_{m}(v, p_{1} \cup p_{2})$ is constructed. $T(v, p)$ can be reconstructed recursively when needed. Therefore, the space needed for $T(v, p)$ is bounded by $O(1)$, and the total space required to store $T(v, p)$ if bounded by $O(2^{k} n)$. The maximum size of $Q_{2}$ is also bounded by $O(2^{k} n)$. So the space complexity is $O(2^{k} n)$.

4.3 Finding GST-k

To compute GST-k, we propose algorithm DPBF-k, by just replacing line 8 in DPBF-I with:

if $p = P$ then
    output $T(v, p)$; $i \leftarrow i + 1$;
    terminate if $i = k$;
Here, $i$ is initialized as 0. DPBF-k reports approximate answers for GST-k where $k > 1$, but the first one, $T_{1}$ is promised to be optimal. Moreover, due to the nature that the smallest cost tree is always kept at the top of the priority queue $Q_{2}$ in DPBF-I, we can find $T_{1}, T_{2}, \ldots, T_{k}$ in the increasing order of cost, i.e. $s(T_{1}) \leq s(T_{2}) \leq \cdots \leq s(T_{k})$, for GST-k. So no sorting is needed. The time complexity and space complexity for solving GST-k is the same as to solving GST-I, because the worst case for solving GST-I is to search all possible trees which is the same as for GST-k.

4.4 Discussions

Handling Directed Graph: Our algorithm can compute GST-k over a directed graph as well as undirected graphs as shown in our experimental studies. The only place that we need to change for dealing with directed graph is the treatment of neighbors in algorithm DPBF-I/k. To handle a directed graph $G$, $N(v)$ needs be refined as $N(v) = \{ u \mid (v, u) \in E(G) \}$, where $E(G)$ is a set of ordered pairs.

Graph Size and Graph Maintenance: As other tuple-based approaches [6, 25], we assume the existence of a materialized database graph $G(V, E)$ in memory. For each node and edge in $G$, we only maintain the IDs of relevant tuples in the main memory, and based on Remark 2.1, $G$ is sparse. So the memory consumption for the materialized database graph $G$ is small. Consider a real database graph $G$ for DBLP in year 2004 [21]. The number of nodes is $n = 1900K$, and the number of edges is $m = 5400K$. The memory for the materialized graph $G$ is less than 34MB.

Moreover, the database graph can be maintained dynamically using two additional hash structures, a node-hash $H_{v}$ and an edge-hash $H_{e}$. The point is: when database is updated, the materialized database graph in the memory can be also updated with nearly zero cost.

Cost Functions: Suppose the database graph $G$ is a node/edge-weighted graph. Let $w_{v}(v)$ and $w_{e}(e)$ be a node-weight and edge-weight for a node $v$, and an edge $e$ in $G$. DPBF-I/k can support any additive cost function $s'(T)$ in the form shown in Eq. (13)-(15), and any nonnegative weights $w_{v}(v)$ and $w_{e}(e)$.

$$s'(T) = (1 - \lambda) \cdot s_{v}'(T) + \lambda \cdot s_{e}'(T). \quad (13)$$

Here, the total costs for nodes and edges, $s_{v}'(T)$ and $s_{e}'(T)$, must be additive. And $\lambda \in [0, 1]$. For example,

$$s_{v}'(T) = \sum_{v \in V(T) \cap V} w_{v}(v) \quad (14)$$

$$s_{e}'(T) = \sum_{e \in E(T)} w_{e}(e) \quad (15)$$

Here, $s_{v}'(T)$ is the total node weight of those nodes in $T$ that also appear in $V$, where $V$ is the set of nodes in $G$ that contain at least a keyword. And $s_{e}'(T)$ is the total edge weight in $T$. Note Eq. (1) is a special case of Eq. (13) with $\lambda = 1$. Handling node-weight in GST-I (or GST-k) does not increase the complexity, as can be sensed in the discussions in Section 4.1. Node weights can be assigned using the approaches given in [7, 4].

5 Experimental Studies

We conducted extensive experimental studies to compare our parameterized solution, DPBF, with four algorithms, namely, BANKS-I [6], BANKS-II [25], RIU-E [22], and RIU-T [22]. We implemented all algorithms using C++. We used the default values of the parameters in the existing work, and tuned the parameters to get the better results, when needed.

We report our findings, using the total edge-weight of a tree as the cost (Eq. (1)), over undirected/directed graphs. All algorithms use the same weight scheme. Due to space limit, we do not report the tests on node/edge-weighted graphs, because they show the similarity with those on edge-weighted graphs. We do not report our DPH-I, because DPBF outperforms DPH-I. We do not compare our results with the work in [19, 5, 16, 8, 9, 15], because their time complexity is higher than $O(n^{2})$, which make them difficult to handle large database graphs.

We conducted all tests on a 3.4GHz CPU and 2G memory PC running XP. For each test, we selected at least 20 keyword queries, and report Processing Time (msec), Memory Consumption (in terms of the number of nodes), and Cost (the total edge-weight), on average.

We used two real datasets, DBLP [21] and MDB [24]. The database schema of DBLP is outlined in Figure 1 (a). DBLP
consists of 1,900K records for research papers archived up to year 2004. MDB consists of 1 million records for a movie recommendation system. It contains of 2,811,983 ratings entered by 72,916 users for 1,628 different movies.

**Exp-1 (Scalability):** We first conducted a scalability test, because it is critical whether an algorithm can compute GST-I for a large graph. We divide DBLP into 10 datasets, namely, 100K (up to 1982), 300K (up to 1987), 500K (up to 1993), 700K (up to 1996), 900K (up to 1997), 1100K (up to 1999), 1300K (up to 2000), 1500K (up to 2001), 1700K (up to 2002), and 1900K (up to 2004). We construct 10 edge-weighted undirected/directed graphs correspondingly, and then randomly select 20 4-keyword queries.

Results of tests on 10 undirected graphs are shown in Figure 6. RIU-E is the best in terms of processing time and is the worst in terms of cost, because it uses a simple heuristics to select edges when expanding. On the other hand, BANKS-I is the worst in terms of processing time but computes GST-I with a small average cost, which is very close to the optimal. BANKS-II significantly improves the efficiency of BANKS-I, but produces a larger average cost. Overall, our DPBF finds the optimal GST-I, and outperforms BANKS-I and BANKS-II in terms of processing time. As shown in Figure 6 (b), our DPBF consumes less memory than that of BANKS-I.

Results of tests on 10 directed graphs are shown in Figure 7. DPBF finds the optimal GST-I, and outperforms BANKS-II. We do not report RIU-E in Figure 7, because of its large average cost obtained (see Figure 6).

We do not include RIU-T in Figure 6-7, because it consumes much more processing time and memory than others to compute GST-I when the graph is large. For 300K, RIU-T takes more than 5 minutes for a 4-keyword query.

In the following experiments, we use the dataset 500K to test other settings, which is in favor of BANKS-I, because the processing time of BANKS-I increases significantly, when the number of nodes is large. We do not report our finding for the edge-weighted directed graph, because they show the similar results as those for undirected graphs.

**Exp-2 (Keywords):** We vary the number of keywords \( l \), from 2 to 6, to compute GST-I. For each \( l \) value, we randomly generate 100 queries to test. Results are shown in Figure 8. DPBF finds the optimal GST-I, and outperforms BANKS-I, BANKS-II and RIU-E in terms of cost. As shown in Figure 8, the processing time of DPBF is not significantly affected by \( l \) value, except a jump from \( l = 2 \) to 3. It is because, when \( l = 2 \), GST-I becomes the shortest path problem, can be solved by DPBF efficiently. Afterward processing time does not increase much while \( l \) increases. It indicates that our DPBF can support most user keyword queries when \( l \) is of a reasonable number.

We also test \( l \)-keyword queries of different keyword patterns (keywords are with low/medial/high frequency). It is found that, with low frequency keywords, the average cost is higher than those with high frequency keywords, and the processing time is longer. It is because that with low frequency keywords, the probability of obtaining a large GST-I is high. In Figure 8, the 100 queries are selected uniformly from different patterns.

**Exp-3 (GST-k):** We test GST-k, using the same 100 randomly-generated 4-keyword queries, that we used in Exp-2. We vary \( k \) from 1, 3, ... to 40, and report our results in Figure 9. From Figure 9 (a), all algorithms to be tested can compute GST-k in a progressive manner, and the processing time is not much more than computing GST-I. Our DPBF outperforms the others in terms of cost, and DPBF outperforms BANKS-III in terms of processing time.

**Exp-4 (MDB(Directed Graph)):** We test MDB dataset [24] as an edge-weighted directed graph. We first vary the number of keywords, \( l \), from 2 to 6, using 100 randomly generated \( l \)-keyword queries, for each \( l \). The results are shown in Figure 10. Then, we fixed \( l = 4 \), and randomly generated...
ACKNOWLEDGMENT: This work was supported by a grant of the Research Grants Council of the Hong Kong SAR, China (No. 418206), ARC discovery grant (DP0666428), UNSW FRG (RGP, PS08709), and the Research Grants of NSFC (No. 60473069, 60496325).

References


