Advanced Query Processing in Databases

DING, Bolin

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Systems Engineering and Engineering Management

August 2007

Abstract of thesis entitled:
Advanced Query Processing in Databases
Submitted by DING, Bolin
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in August 2007

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 thesis, we study processing an l-keyword query, $p_1, p_2, \cdots, p_l$, against a relational database which can be modelled 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 $O(3^l n + 2^l((l + \log n)n + m))$, 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.

Acknowledgement

First, I wish to thank my supervisor, Professor Prof. YU Xu, Jeffrey, for his numerous insightful comments in this project, his help on improving the quality of the text, and in general for his support. Also thank Professor LIN Xuemin, from The University of New South Wales, for his comments and encourage in this project. Finally, I am grateful to my friend and colleague, QIN Lu, in SEEM Department at CUHK (and also Renmin University previously). He helped implement algorithm RIU and BANKS-II to complete the experiments for this project.

This work is dedicated to my parents, my supervisor, and my friends.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Summary

This thesis is mainly based on [13] and [37]. [13] introduced the preliminary works of this thesis. It was presented as a regular paper in the 23nd International Conference on Data Engineering (ICDE 2007), and recieved the Best Student Paper Award (shared by 2). [37] showed a demo based on the algorithms introduced in this thesis. The first chapter introduces the overview of our research and the thesis, and the summary of results.

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, 11]. The recent studies on supporting

IR style queries in RDBMS include *DBXPlore* [1], *IR-Style* [23], *DISCOVER* [24], *ObjectRank* [4], *BANKS-I* [7], and *BANKS-II* [36]. All consider a RDBMS as a graph where nodes represent tuples/relations and edges represent foreign key references among tuples cross relations. With a keyword query, users can find the connections among the tuples stored in relations without the needs 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 1.1 (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 1.1 (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. Nodes may also have weights defined in a similar way. For simplicity, we only show edge weights in Figure 1.1 (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

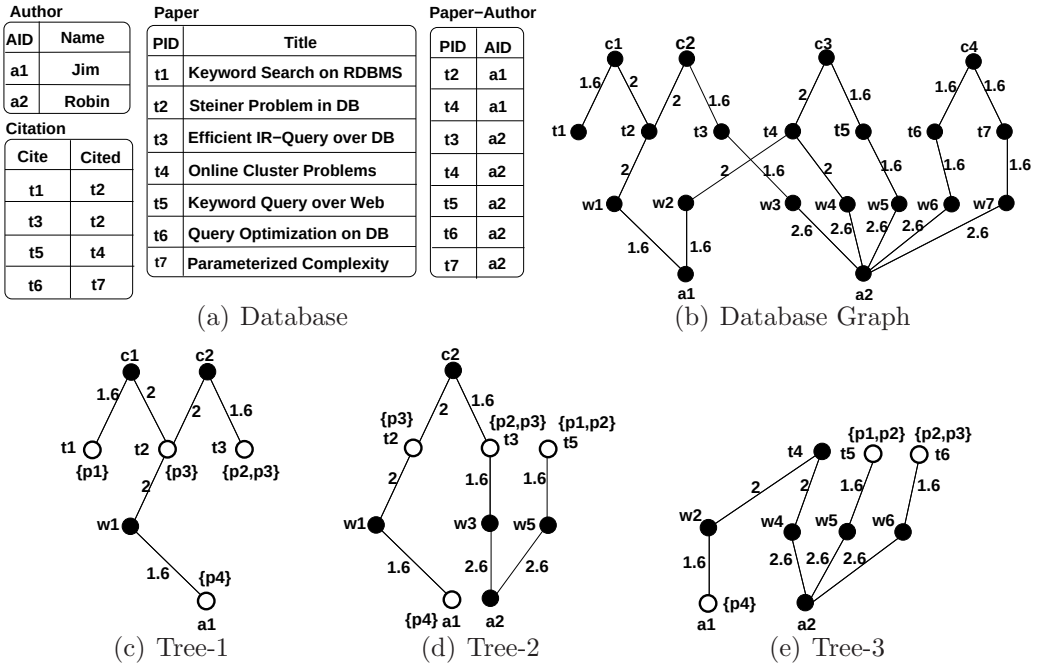


Figure 1.1: A Motivation Example

database, and consequently the database graph. Figure 1.1 (c)-(e) show 3 possible connected trees, `Tree-1`, `Tree-2`, and `Tree-3`, 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.

`Tree-3` implies that that `Jim` (a_1) wrote a paper t_4, and the co-author Robin (a_2) of it writes other two papers t_5 and t_6 with keywords p_1, p_2, and p_3.

The total weights on the 3 trees are 10.8, 15.6, and 16.6. The answer with a smaller total weight is ranked higher, because it represents stronger and more concise relationship among keywords. □

1.1 Overview of Research

In the literature, the reported approaches that support keyword queries in RDBMS can be categorized into two types, *relation-based* and *tuple-based*.

The relation-based approaches aim at processing a keyword query with SQL, by utilizing the schema information in RDBMS. Here, a schema graph is constructed over the relational schema, where a node represents a relation and an edge represents a foreign key reference between two relations. For a given keyword query, based on this graph, *Candidate Networks* and corresponding SQL statements are generated to handle keyword queries. Advanced techniques were proposed to reduce SQL processing cost. Such systems include *DBXPlore* [1], *DISCOVER* [24], *IR-Style* [23], and *SPARK* [32].

On the other hand, the tuple-based approaches aim at processing a keyword query by utilizing the structural information (foreign-key references) in RDBMS, and the weights associated with tuples and foreign-key reference. Here, a database graph $G(V, E)$ is constructed over tuples in RDBMS, where a node represents a tuple and an edge represents foreign-key reference between two tuples. This graph is node/edge-weighted. The node/edge (tuple/foreign key) weights can be assigned using [8, 4]. With this weighted graph, the tuple-based approaches find top-k min-cost connections (trees) in this graph for a keyword query. Such systems include *RIU* [31], *BANKS-I*, [7], and *BANKS-II* [36].

In this thesis, we focus ourselves on the tuple-based approaches, because we note some databases may have very simple schemas of only two or three tables, but consist of very complicated reference structures of foreign keys. In this case, the relation-based approaches cannot fully make use of the structural information behind foreign keys at the tuple level because of the feature of

SQL. Generally speaking, tuple-based approaches allow us to look into the keyword query problem over RDBMS in a more delicate manner, and to design more efficient algorithms.

Like other tuple-based approaches, we assume the existence of the node/edge-weighted database graph $G(V, E)$ in the main memory. The memory-consumption of this materialized graph is small enough to allow the maintenance of it in the main memory, which we will discuss later in this thesis.

Given a l-keyword query, $p_1, p_2, \cdots p_l$, we study finding top-k min-cost connected trees that contain all l keywords at least once in the database graph. We use *GST-1* to denote the min-cost connected tree, and *GST-k* to denote the top-k min-cost connected trees. This problem (*GST-1/k*) is also well known as the *(top-k) min-cost group Steiner tree*.

The min-cost group Steiner tree problem is an important (and of course NP-hard) problem in the areas of network optimization and graph theory. It is at least as hard as the Set Cover problem, and thus cannot be solved with constant performance ratio in polynomial time. Both of its approximation algorithms [33, 25, 6, 22, 9, 10, 19, 35, 16, 27] and approximability [26, 18, 21, 20] are studied intensively.

In the scenario of a l-keyword query over the database $G(V, E)$ with n nodes and m edges, we identify a characteristics of the (top-k) min-cost group Steiner tree problem: l **is a small number, and** G **is a sparse large graph (**$l \ll \log n < n < m \ll n^2$**)**. Because of this characteristics, we aim to design $O(g(l)f(n, m))$ algorithms, where $f(n, m)$ is a low-degree polynomial function, and $g(l)$ can be any function, say 2^l or $l!$. In general, this kind of algorithms is of special interests, when some parameters of the problem instances are small (refer to Downey and Fellows' book about Parameterized Complexity [14]).

1.2 Summary of Results and Thesis Outline

The material in this thesis is organized as follows:

Chapter 2 gives problem statement and discusses the characteristics of the problem. Chapter 3 reviews the related work.

In Chapter 4, based on dynamic programming, we propose an algorithm to find approximate *GST-1*. For fixed $0 < d < l$, its performance ratio is $O(\sqrt[d]{l})$, time complexity is $O(d(3^l n + 2^l m) + \alpha)$, and space complexity is $O(2^l nd)$. Here, $O(\alpha)$ is the time to compute all-pairs shortest paths in database graph $G(V, E)$. When $d = l$, it can find optimal *GST-1*.

In Chapter 5, we propose an exact algorithm to find optimal *GST-1* (or min-cost group Steiner) with time complexity $O(3^l n + 2^l((l + \log n)n + m))$ and space complexity of $O(2^l n)$. It is achieved by improving the dynamic programming algorithm introduced in Chapter 4 with a best-first strategy. Note: this parameterized algorithm works efficiently on the condition that l is small (even when the database graph $G(V, E)$ is very large), and we are not solving the problem in a general setting where l can be any large (in the order of n). This algorithm can be extended to find *GST-k* (top-k min-cost group Steiner trees) in a progress manner. That is, we needn't compute/sort all group Steiner trees and then find *GST-k*. We discuss our *GST-k* approach, and show its advantages in Chapter 5. Some other important issues about our algorithms like handling undirected/directed graphs, weight schema supported, and graph maintenance, are discussed at the end of Chapter 5.

In Chapter 6, we give our experimental studies. It is shown that our approach outperforms existing approaches with high quality and efficiency.

Finally, Chapter 7 concludes the thesis.

□ **End of chapter.**

Chapter 2

Problem Statement

Summary

This chapter models the keyword query and define the *GST-1/k* problem.

Database graph: Given an RDBMS, $\mathcal{DB}$, upon a relational schema $\mathcal{R}$ with foreign key references. We define a weighted database graph, $G(V, E)$, where $V(G)$ is the set of tuples (nodes) in $\mathcal{DB}$, and $E(G)$ is the set of edges. An edge, $(u, v) \in E(G)$, represents a foreign key reference between two tuples, u and v, if u has a foreign key matching the primary key attributes of v, or v has a foreign key matching the primary key attributes of u.

The graph is an undirected graph[1] 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 may be two distinct edges, (u, v) or (v, u), in $E(G)$. Graph $G(V, E)$ is weighted, with a node weight $w_v(u)$ for every node $u \in V$ and an edge weight $w_e((v, u))$ for every edge $(v, u) \in E(G)$, both of which are non-negative numbers. Below, let $n = |V(G)|$ and $m = |E(G)|$.

[1]We assume $G(V, E)$ is undirected in the following part. Discussion about how to handle directed graphs will appear in Section 5.4, Chapter 5.

l-keyword query: Consider a l-keyword query, i.e. a set of keywords, $p_1, \cdots, p_l$, against a database graph G. There is a set of nodes, denoted as V_i ($\subseteq V(G)$), that contain the keyword p_i, for $i = 1, \cdots, l$. We call V_i a *group* for the keyword p_i or simply a *group*. Note: 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. All groups can be obtained with either the symbol-table techniques [1] or the full text index techniques [24].

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

$$s(T) = \sum_{(v,u) \in E(T)} w_e((v,u)) \tag{2.1}$$

where $w_e((v,u))$ is the weight of edge (v,u), such that a lower cost represents a tighter relationship.

Remark 2.1: Weighting Edges *Let $N(v)$ be a set of neighbors of v, and $|N(v)|$ be the size of $N(v)$. We use $w_e(\cdot)$ in Equation (2.2) to weight edges in an undirected graph.*

$$w_e((v,u)) = \log_2(1 + \max\{|N(v)|, |N(u)|\}) \tag{2.2}$$

Note: $w_e((v,u)) = w_e((u,v))$ in an undirected graph. In a directed graph, let $N_{in}(v)$ be the set of nodes that reference to v. We use Equation (2.3) and Equation (2.4), which were also used in [36], to weight directed edges. In details, for a foreign key reference from u to v, the edge weight for (u,v) is given Equation (2.3), and the edge weight for (v,u) is given Equation (2.4).

$$w_e((u,v)) = 1 \tag{2.3}$$

$$w_e((v,u)) = \log_2(1 + N_{in}(v)) \tag{2.4}$$

The semantic captured by Equation (2.2)-(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). We will address how to enhance Equation (2.1) to handle node weights as well as edge weights later in Section 5.4, Chapter 5. □

Top-k group Steiner tree problem (GST-k): Given a l-keyword query, to find the top-k min-cost group Steiner trees, $T_1, T_2,, \cdots, T_k$, ranked with the cost function s defined in Equation (2.1), such that $s(T_1) \leq s(T_2) \leq \cdots \leq s(T_k)$. Example 1.1 shows an example of *GST-3*.

In this thesis, 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 $\mathcal{DB}$. The group Steiner tree problem has the following characteristics in the scenario of l-keyword queries over RDBMS.

Remark 2.2: Characteristics of Keyword Queries

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_u (with foreign key) to R_v (with primary key). A tuple in R_u can reference to at most one tuple in relation R_v, and can reference to at most r tuples if the database has r relations in total. Therefore, $m \leq rn$, since there are n nodes in total. Note the number of relations $r \ll n$. □

□ **End of chapter.**

Chapter 3

Literature Review

Summary

This chapter reviews the related work.

3.1 Hardness Results

Recall the inputs of the min-cost group Steiner tree problem (or *GST-1*) are: a weighted graph $G(V, E)$ with n nodes and m edges, and l groups (keywords) $V_1, \cdots, V_l \subseteq V$. It is proved to be NP-complete [33], because it is a generalized minimum set cover problem.

An *β-approximation* is β times of the optimal solution's value. In the min-cost group Steiner tree problem, let T be a tree found by an algorithm, and T^* be the optimal. Then T is an β-approximation iff $s(T)/s(T^*) \leq \beta$. Parameter β is also called *performance ratio*.

It was first proved by Ihler that *GST-1* (as well as the minimum set cover problem) is inapproximable within a constant performance ratio by a polynomial algorithm [26]. The first lower bound of the polynomial-time performance ratio $\Omega(\log l)$ was established by Feige in [18]. Recently, [21] and [20] studied

the hardness of approximation (or *inapproximability*) of *GST-1*. [21] pointed out this problem admits no $\log^{2-\epsilon} l$-approximation (in polynomial time) even when $G(V, E)$ is a tree. On trees, this bound is nearly tight with the logsquared approximation, $O(\log n \log l)$, currently known in [10, 19].

3.2 Approximation Algorithms

Because of the hardness of this problem, the existing reported studies aim at approximating the cost of the optimal *GST-1*. The approximation algorithms can be categorized into three types: spanning and cleanup, d-star Steiner tree or i-level tree, and randomized rounding. We discuss them in brief below.

3.2.1 Spanning and Cleanup

This technique was inspired by the minimum spanning tree algorithm. In [33], Reich and Widmayer presented an algorithm which spans a tree T_v from an arbitrary node v in one group step-by-step until it covers at least one node in each group, and then achieves a low-cost tree by cleaning up the redundant nodes. This algorithm's performance ratio is unbounded.

In [25], Ihler presented the first approximation algorithm with the bounded performance ratio, based on the idea behind Reich and Widmayer's work [33]. Ihler's algorithm enumerates the starting node v over all nodes in every group, V_i, for $i = 1, \cdots, l$, and run Reich and Widmayer's algorithm [33] for each v. The performance ratio was proved to be $O(l)$.

3.2.2 d-Star Tree or d-Level Tree

Generally, some edges in $G(V, E)$ may violate the triangle inequality w.r.t. edge weight $w_e(\cdot)$ (2.2). Clearly, the optimal group Steiner will not contain any such edge, because such an

edge can be replaced with a sequence of edges to improve the solution. Therefore, we can suppose G is a *metric closure*, where each edge represents a shortest path between two nodes (we can run an all-pair shortest-path algorithm to transform G into a metric closure).

A *d-star tree* in metric closure 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 $(1 \leq i \leq l)$. The main contribution of [22] by Helvig et al. was to prove the d-star Steiner tree can be used to approximate the optimal (min-cost) group Steiner tree.

Theorem 3.1:[22][9] *Let T^* be the optimal group Steiner tree over l groups in metric closure G, and for any fixed $d > 0$, let T be the d-star Steiner tree with the minimum cost. Then $s(T) \leq O(\sqrt[d]{l})s(T^*)$.* □

However, it is still NP-hard to find the optimal d-star tree for any fixed $d > 1$. Fortunately, an $O(d \log^{d-1} l)$-approximation for the optimal d-star tree can be achieved in polynomial time [22]. Therefore, the performance ratio is $O(d\sqrt[d]{l} \log^{d-1} l)$ in total. In fact, in an earlier time, Bateman et al. [6] used a special case 2-star tree $(d = 2)$ to solve this problem.

Charikar et al. [9] proposed an algorithm to approximate the optimal group Steiner tree based on the i-level tree, which is a similar concept in a directed graph as d-star tree in an undirected graph. The performance ratio was improved to be $O(d^2\sqrt[d]{l})$.

3.2.3 Randomized Rounding

The results of Bartal [5] are used to reduce the graph problem to the tree case (G is a tree), with an $O(\log n \log \log n)$ factor loss in the performance ratio bound. This factor was later improved by Fakcharoenphol et al. to an essentially tight bound $O(\log n)$ in [17].

Unfortunately, *GST-1* is still a very hard problem in trees (refer to Section 3.1), but when $G(V, E)$ is a tree, we can model it as an integer linear program, with an indicator variable $x(e)$ to indicate whether each edge e is in group Steiner tree T. The objective function is the the weighted sum of $\{x(e) : e \in E(G)\}$, and the constraints require that $\{x(e)\}$ support a unit flow (consider $x(e)$ as the edge capacity of G) individually from each group V_i $(i = 1, \cdots, l)$ to the root of G. In the LP relaxation, $x(e)$ is allowed to lie in $[0, 1]$. Let $\{x^*(e) : e \in E(G)\}$ denote an optimal LP solution, and let $s^* = \sum_{e \in E(G)} w_e(e)x^*(e)$.

Randomized rounding [19, 10] works as follows: Every edge e is chosen with probability $x(e)/x(f)$ independently, where f is the "father edge" of e. Let T be the connected component of the chosen edges that includes the root of G. It can be proved that the expected cost of T, $\mathbf{E}[s(T)]$, is s^*. What's more, for each $i = 1, \cdots, l$, the probability that at least one node of V_i is covered by some T is $\Omega(1/\log n)$. So if repeating rounding for $O(\log n \log l)$ times, and combining all the component T in each iteration, we can get an $O(\log n \log l)$-approximation with constant probability.

Therefore, the performance ratio is $O(\log^2 n \log l)$ in total.

3.3 Existing Systems

In the above section, we discussed the approximation algorithms to *GST-1*. As also pointed in [31], all these algorithms can not be efficiently used to compute *GST-k*, because they all need to compute/sort group Steiner trees, in order to find *GST-k*. They cannot terminate any early and report *GST-k* in a progress manner. Below, we introduce existing systems for *GST-k*: *RIU* (Retrieve Information Unit) [31], *BANKS-I* [7], and its successor *BANKS-II* [36]. Their *GST-1* / *GST-k* algorithms are relatively simpler, but effective in practice.

3.3.1 RIU (Retrieve Information Unit)

RIU was proposed by Li et al. in [31]. It adopted the spanning and cleanup strategy as given in [25] to find *GST-k* incrementally. Given a set of groups, $V_1, V_2, \cdots, V_l$. Let $\mathcal{V} = V_1 \cup V_2 \cup \cdots \cup V_l$. The *RIU* algorithm computes *GST-k* based on a set of trees, $\mathcal{M} = \{G_1, G_2, \cdots\}$. Here, G_i is represented by a pair, $(V(G_i), E(G_i))$. Initially, *RIU* constructs $\mathcal{M}$ as $\mathcal{M} = \{(\{v_1\}, \emptyset), (\{v_2\}, \emptyset), \cdots)\}$ for all $v_i \in \mathcal{V}$. In each iteration of the spanning step, *RIU* selects a tree $G_i \in \mathcal{M}$ and an edge (u, v), where $u \in V(G_i)$ and $v \notin V(G_i)$ as a target. Next, it examines whether there is a $G_j \in \mathcal{M}$ such that $v \in V(G_j)$ and $u \notin V(G_j)$. If G_j does not exist in G, it updates G_i by adding (u, v) into G_i in. If G_j exists, it deletes G_i and G_j from $\mathcal{M}$, and inserts a new graph, G_{ij} into $\mathcal{M}$ by merging the two graphs G_i and G_j. If the updated G_i or G_{ij} contains all keywords, *RIU* will output it. The first such G_i or G_{ij} is the top-1 min-cost group Steiner tree, and the k-th such a tree is the k-th min-cost group Steiner tree. There are many possible ways to select G_i and an edge (u, v). In [31], Li et al. proposed two 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.

3.3.2 BANKS

BANKS-I [7] and BANKS-II [36] studied computing *GST-k* over a weighted directed graph. The techniques used can support undirected graphs as well.

Both adopted the 1-star tree strategy [22]. In brief, the *GST-1* 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 actually an edge in the metric closure of G. So *BANKS-I* and *BANKS-II* use the 1-star group Steiner tree to approximate *GST-1*. *BANKS-I* and *BANKS-II* use 1-star 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 techniques [7] and *BANKS-II* uses bidirectional expansion techniques [36] to improve the efficiency of *BANKS-I*.

The *BANKS-I* algorithm [7] is outlined below. Given a l keyword-query, $p_1, p_2, \cdots, p_l$. It first obtains l groups, V_1, V_2, $\cdots, V_l$. Let $\mathcal{V} = \cup_{i=1}^{l} V_i$. The total number of nodes in $\mathcal{V}$ is $|\mathcal{V}|$. Next, it constructs trees, T, from $\mathcal{V}$ as leaf nodes, where every leaf node of T is taken from one of the groups, V_i, for $i = 1, \cdots, l$. The process is done in a backward fashion. Initially, every node in $\mathcal{V}$ is considered as a zero-length path. In each iteration, *BANKS-I* selects a path, from all existing paths, to expand, if the path is the shortest one, with the smallest sum of edge weights, among all possible path expansions. While expanding paths from leaf nodes backwards, paths will merge into trees, and a group Steiner tree is formed when all of its l leaves are taken from all l groups. With a cost function, all the group Steiner trees, each of which is formed from l shortest paths, are sorted, and thus *GST-k* can be obtained.

BANKS-II [36] significantly improves *BANKS-I* in terms of efficiency. It conducts searching using a bidirectional expansion strategy. In brief, in addition to the backward strategy used in *BANKS-I*, it also attempts to form a potential tree, T, by forward expanding the root node of T down to the leaf nodes, using the same shortest path strategy, in every iteration. The bidirectional expansion techniques reduce the size search space. But, in many cases, *BANKS-II* produces a lower-quality performance ratio than that of *BANKS-I*, because *BANKS-II* may

Algorithms	**Methodology**	**Performance Ratio**	**Time Complexity**
Reich et al. [33]	Spanning and Cleanup	unbounded	$O(l(m + n \log n))$
Ihler et al. [25]	Spanning and Cleanup	$O(l)$	$O(ln(m + n \log n))$
Bateman et al. [6]	2-Star Tree	$O(\sqrt{l} \log l)$	$O(\alpha + n^2 l^2 \log l)$
Helvig et al. [22]	d-Star Tree	$O(d \sqrt[d]{l} \log^{d-1} l)$	$O(\alpha + (nl)^d)$
Charikar et al. [9]	d-Level Tree	$O(d^2 \sqrt[d]{l})$	$O(\alpha + n^d l^{2d})$
Charikar et al. [10]	Randomized Rounding	$O(\log^2 n \log l)$	polynomial
Garg et al. [19]	Randomized Rounding	$O(\log^2 n \log l)$	polynomial
RUI [31]	Spanning and Cleanup	$O(l)$	$O(ln(m + n \log n))$
BANKS [7, 36]	1-Star Tree	$O(l)$	$O(n^2 \log n + nm)$
Our solution	Dynamic Programming	1 (**optimal**)	$O(3^l n + 2^l((l + \log n)n + m))$

Table 3.1: Approximation and Time Complexity of GST-1 Solutions

miss some shortest paths when it uses bidirectional expansion techniques to speed up.

3.4 Summary

Table 3.1 summarizes performance ratio and time complexity for *GST-1* algorithms, including ours. Note $O(\alpha)$ is the time needed to find all-pairs shortest paths in graph (APSP), i.e., to transform G into a metric closure.

It will be show our solution can find optimal *GST-1*. Meanwhile, it is important to note that all algorithms including ours attempt to find *GST-k* with small performance ratio. But, it is difficult to measure the theoretical bounds of performance ratio for the i-th min-cost group Steiner tree when $i > 1$.

□ **End of chapter.**

Chapter 4

Algorithm Based on Height-Bounded Trees

Summary

This chapter introduces an approximate algorithm to find *GST-1*. For fixed $0 < d < l$, its performance ratio is $O(\sqrt[d]{l})$, time complexity is $O(d(3^l n + 2^l m) + \alpha)$, and space complexity is $O(2^l nd)$. Here, $O(\alpha)$ is the time to compute all-pairs shortest paths in database graph $G(V, E)$. When $d = l$, it can find optimal *GST-1*.

All the reported works deal with *GST-1* (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 2.2). So in the time complexity given in Table 3.1, 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 [25, 6, 22, 9, 10, 19] is at least $O(n^2)$, which makes them difficult to be efficiently applied to a large graph G with

millions of nodes. The exemption is [33], but its performance ratio is unbounded.

In this chapter, we treat the input size, l, m, and n differently, and propose our first parameterized algorithm, allowing l to appear as an exponential in the time complexity. It considers the height-bounded trees, and finds $O(\sqrt[d]{l})$-approximation of *GST-1* in time $O(d(3^l n + 2^l m) + \alpha)$ for fixed height $d < l$. It also finds the optimal (performance ratio=1) *GST-1* in time $O(l(3^l n + 2^l m) + \alpha)$. Here, $O(\alpha)$ is the time needed to find all-pairs shortest paths in graph (APSP).

The theory of parameterized complexity is formally presented in [14]. The fixed-parameter tractability of 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 thesis, we treat the number of keywords, l, as a parameter. To our best knowledge, the min-cost Steiner tree problem, as a special case of the min-cost group Steiner tree problem, i.e. $|V_i| = 1$ for $1 \leq i \leq l$, was proved to be fixed-parameter tractable in [15]. 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-1*'s optimal substructure.

In the follow part of this thesis, given a l keyword query against a database graph G, let $\mathbf{P}$ be the entire set of keywords $\mathbf{P} = \{p_1, p_2, \cdots, p_l\}$, and let $\mathbf{p}$, $\mathbf{p}_1$ and $\mathbf{p}_2$ denote a non-empty subset of $\mathbf{P}$, i.e., $\mathbf{p}, \mathbf{p}_1, \mathbf{p}_2 \subseteq \mathbf{P}$.

Based on similar ideas, we will show how to improve this algorithm presented in this chapter to an $O(3^l n + 2^l((l+\log n)n + m))$ one (lower than $O(n^2)$ for fixed l) to find the optimal *GST-1* in Chapter 5.

4.1 Naive Dynamic Programming Algorithm

We give a dynamic programming solution by taking the heights, h, of trees as stages, and find the optimal *GST-1* by expanding the trees with heights $h = 0, 1, 2, \cdots$, until the *GST-1* is found. The optimal *GST-1* for a set of keywords $\mathbf{p}$, with a certain height h, is found from the optimal solutions to *GST-1*, for subsets of keywords $\mathbf{p}_1$, such that $\mathbf{p}_1 \subseteq \mathbf{p}$, with heights $\leq h$.

More formally, let $T(v, \mathbf{p}, h)$ be a tree with the minimum cost rooted at node v, with height $\leq h$, containing a set of keywords $\mathbf{p}$. In other words, it is a min-cost h-star tree with keywords $\mathbf{p}$. Every single node v in G containing a non-empty set of keywords, $\mathbf{p}$ ($\subseteq \mathbf{P}$), is a rooted tree with zero height, $h = 0$. We denote it $T(v, \mathbf{p}, 0)$. Such a tree does not have any edges, and therefore the cost of the tree $T(v, \mathbf{p}, 0)$ is zero (refer to Equation (2.1)) as given below.

$$T(v, \mathbf{p}, 0) = 0 \tag{4.1}$$

Here, we abuse the notations a bit: the left side is the tree, and the right side is the cost of the tree that appears on the left. Note $T(v, \emptyset, 0) = \infty$. In general, the (minimum) cost of $T(v, \mathbf{p}, h)$, for any $h > 0$, is given below in Equation (4.2).

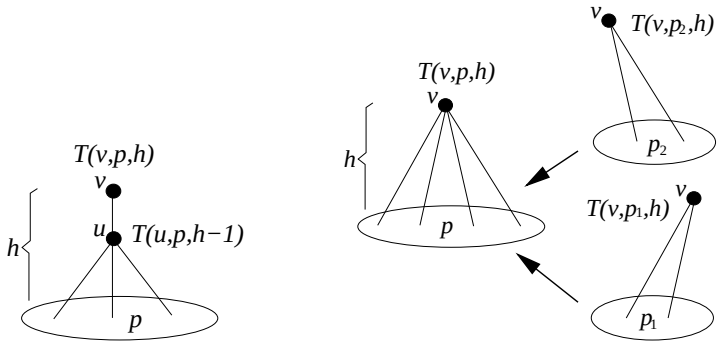
$$T(v, \mathbf{p}, h) = \min\{\mathsf{T}_{\mathsf{g}}(v, \mathbf{p}, h), \mathsf{T}_{\mathsf{m}}(v, \mathbf{p}, h), T(v, \mathbf{p}, h-1)\} \tag{4.2}$$

where

$$\mathsf{T}_{\mathsf{g}}(v, \mathbf{p}, h) = \min_{u \in N(v)} \{(v, u) \oplus T(u, \mathbf{p}, h-1)\} \tag{4.3}$$

$$\mathsf{T}_{\mathsf{m}}(v, \mathbf{p}_1 \cup \mathbf{p}_2, h) = \min_{\mathbf{p}_1 \cap \mathbf{p}_2 = \emptyset} \{T(v, \mathbf{p}_1, h) \oplus T(v, \mathbf{p}_2, h)\} \tag{4.4}$$

Operation $\oplus$ is to merge two trees into a new tree, and $N(v)$ is a set of neighbors of v, i.e., $N(v) = \{u \mid (v, u) \in E(G)\}$ in the database graph G. As a special case, if v contains some additional keywords $\mathbf{p}'$, then the left side of Equation(4.3) should be $\mathsf{T}_{\mathsf{g}}(v, \mathbf{p} \cup \mathbf{p}', h)$. We explain Equation (4.2)-(4.4) below.



(a) Tree Grow ($\mathsf{T_g}$) (b) Tree Merge ($\mathsf{T_m}$)

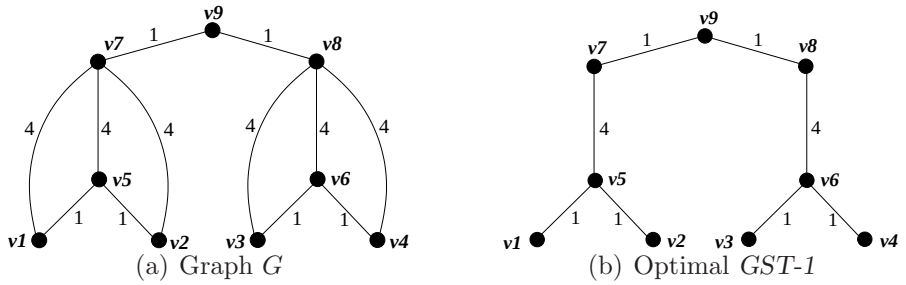
Figure 4.1: Optimal Substructure

In Equation (4.2), tree $T(v, \mathbf{p}, h)$ is constructed from either $\mathsf{T_g}(v, \mathbf{p}, h)$ or $\mathsf{T_m}(v, \mathbf{p}, h)$. The last term in Equation (4.2) is to explicitly state that $T(v, \mathbf{p}, h)$ is the tree with the minimum cost among all $T(v, \mathbf{p}, h')$ for $h' \leq h$. If the cost of $\mathsf{T_g}(v, \mathbf{p}, h)$ is minimum, $T(v, \mathbf{p}, h)$ is constructed as $\mathsf{T_g}(v, \mathbf{p}, h)$. Otherwise, $T(v, \mathbf{p}, h)$ is constructed as $\mathsf{T_m}(v, \mathbf{p}, h)$. $\mathsf{T_g}(v, \mathbf{p}, h)$ is for a case, called *tree grow*, as illustrated in Figure 4.1 (a), where the degree of the root is 1; $\mathsf{T_m}(v, \mathbf{p}, h)$ is for another case, called *tree merge*, as illustrated in Figure 4.1 (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 $T(v, \mathbf{p}, h)$ (or $T(v, \mathbf{p}_1 \cup \mathbf{p}_2, h)$). Equation (4.3)-(4.4) request that the cost, $s(T(v, \mathbf{p}, h-1)) + w_e(v, u)$ (or $s(T(v, \mathbf{p}_1, h)) + s(T(v, \mathbf{p}_2, h))$), 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, $\mathbf{P} = \{p_1, p_2, \cdots, p_l\}$. 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 rooted at v with height bounded by h can be computed as $T(v, \mathbf{P}, h)$ using Equation (4.1) – Equation (4.4).* □

Proof: Theorem 4.1 can be proved by the induction on h



(a) Graph G (b) Optimal *GST-1*

Figure 4.2: An Example

and $|\mathbf{p}|$. We need to show that, $T(v, \mathbf{p}, h)$ be a tree with the minimum cost rooted at node v, with height $\leq h$, containing a set of keywords $\mathbf{p}$, i.e. Equation (4.1) – Equation (4.4) are correct.

When $h = 0$, $T(v, \mathbf{p}, h)$ is gotten from Equation (4.1), and is obviously correct. When $\mathbf{p} = \varnothing$, $T(v, \mathbf{p}, h)$ is also zero for any h from the definition. Suppose $T(v, \mathbf{p}, h)$ can be correctly computed from Equation (4.1) – Equation (4.4) for $h \leq d$ or $|\mathbf{p}| \leq c$. To accomplish the proof, we need prove $T(v, \mathbf{p}, h)$ is correct when $h = d + 1$ or $|\mathbf{p}| = c + 1$.

Consider two cases of the min-cost tree $T(v, \mathbf{p}, h)$:

i) When the degree of root v in $T(v, \mathbf{p}, h)$ is 1, it must be expanded from min-cost tree $T(u, \mathbf{p}, h - 1)$ by adding one edge (v, u) (tree grow case - Figure 4.1 (a)). From the induction, we know $T(u, \mathbf{p}, h - 1)$ is correct, and thus according to Equation (4.3), $T(v, \mathbf{p}, h)$ can be correctly computed.

ii) When the degree of root v in $T(v, \mathbf{p}, h)$ larger than 1, $T(v, \mathbf{p}, h)$ can be decomposed into two trees $T(v, \mathbf{p}_1, h)$ and $T(v, \mathbf{p}_2, h)$ at the same root v (tree merge case - Figure 4.1 (b)). Both of them are min-cost (otherwise $T(v, \mathbf{p}, h)$ is not min-cost), and are correctly computed from the induction ($|\mathbf{p}_1|, |\mathbf{p}_2| < |\mathbf{p}| = c + 1$). Therefore, according to Equation (4.4), $T(v, \mathbf{p}, h)$ can be correctly computed.

The above to two cases show $T(v, \mathbf{p}, h)$ is correct. □

Algorithm 1 *DPH-1*

input: database graph $G(V, E)$, the set of keywords $\mathbf{P}$, groups $V_1, \cdots, V_l$, height d

output: *GST-1* with height bounded by d

1: $h \leftarrow 0$;
2: compute $T(v, \mathbf{p}, 0)$ for every $v \in V(G)$ that contains any keywords in $\mathbf{P}$ (Equation (4.1));
3: **while** $h \leq d$ **do**
4: $h \leftarrow h + 1$;
5: **for** each $v \in V(G)$ **do**
6: $\mathcal{T} \leftarrow \{T(v, \mathbf{p}, h-1)\}$;
7: for all possible $\mathbf{p}$, compute $\mathsf{T}_\mathsf{g}(v, \mathbf{p}, h)$ from its neighbors (Equation (4.3)), and insert them into $\mathcal{T}$;
8: compute all possible $\mathsf{T}_\mathsf{m}(v, \mathbf{p}_1 \cup \mathbf{p}_2, h)$ from two trees in $\mathcal{T}$ (Equation (4.4)); and insert the new trees into $\mathcal{T}$;
9: **for** each $v \in V(G)$ **do**
10: $T(v, \mathbf{p}, h) \leftarrow \min\{\mathsf{T}_\mathsf{g}(v, \mathbf{p}, h), \mathsf{T}_\mathsf{m}(v, \mathbf{p}_1 \cup \mathbf{p}_2, h), T(v, \mathbf{p}, h-1)\}$;
11: **return** $\min_{v \in V}\{T(v, \mathbf{P}, h)\}$;

One more point need be clarified: Consider Equation (4.3) using Figure 4.1(a). The node v may appear in the tree $T(u, \mathbf{p}, h-1)$, and thus $(v, u) \oplus T(u, \mathbf{p}, h-1)$ 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 $T(v, \mathbf{p}, h)$. It is similar for Equation (4.4) and Figure 4.1(b).

The naive dynamic programming algorithm, for *GST-1*, is outlined in Algorithm 1, called *DPH-1*. It is a straightforward implementation of Equation (4.1)-(4.4).

Theorem 4.2: *Algorithm 1 can correctly compute the optimal GST-1 with height bounded by d.* □

Example 4.1: A database graph with edge weights is shown in Figure 4.2 (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-1* with cost 14 is shown in Fig-

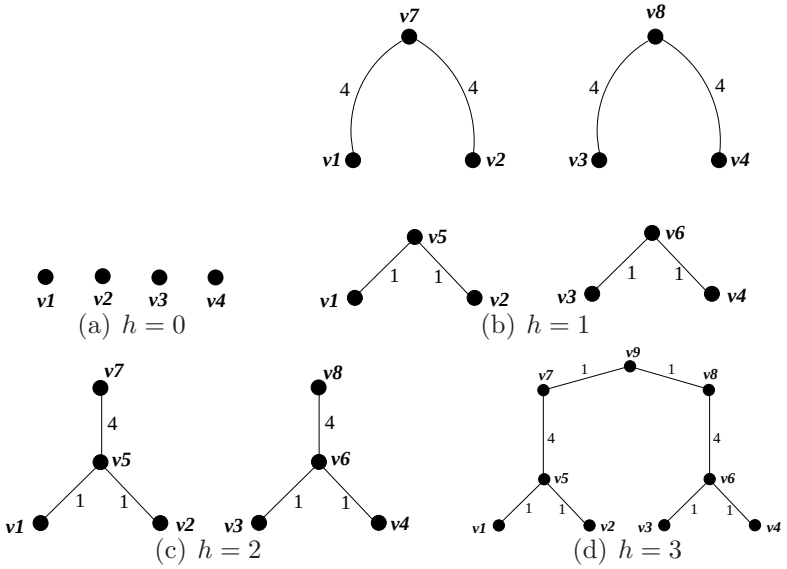


Figure 4.3: A Naive DP Solution: *DPH-1*

ure 4.2 (b). Figure 4.3 (a)-(c) show the intermediate results based on *DPH-1* when $h = 0, 1, 2$, and Figure 4.3 (d) shows the final result. Note: by employing 1-star tree technique, *BANKS-I/II* can only find the tree $(v_9(v_7(v_1v_2))(v_8(v_3v_4)))$ with cost 18 as *GST-1* answer (in Figure 4.3 (d), replace $\{(v_7, v_5), (v_5, v_1), (v_5, v_2)\}$ with $\{(v_7, v_1), (v_7, v_2)\}$, and replace $\{(v_8, v_6), (v_6, v_3), (v_6, v_4)\}$ with $\{(v_8, v_3), (v_8, v_4)\}$). □

4.2 Performance Ratio and Complexity

Note Algorithm 1 finds optimal *GST-1* with height bounded by d, which is an approximation of *GST-1*. Based on the results of [22] and [9], Algorithm 1 can achieve high performance ratio for *GST-1* for small d. We can also prove when d is large enough $(d = l)$, Algorithm 1 can achieve optimal *GST-1*.

Corollary 4.3: *For fixed d, Algorithm 1 can get the* $O(\sqrt[d]{l})$*-approximation of GST-1.* □

This result is directly gotten from Theorem 3.1 and Theorem 4.2. Note: usually database graph $G(V, E)$ is not a metric closure, but we can run an all-pair shortest-path algorithm to transform into a metric closure.

Theorem 4.4: *When $d = l$, Algorithm 1 can get the optimal GST-1.* □

Proof: Again, assume $G(V, E)$ is a metric closure, the above theorem is not hard to be proved. Over a metric closure $G(V, E)$, there is no 2-degree node in the optimal *GST-1*, otherwise, we can replace the two edges incident on it with another edge, to reduce the cost. What's more, all the leaves in the optimal *GST-1* must contain some keywords. Therefore, the depth of the optimal *GST-1* is at most l (there are l keywords), and it can be found by Algorithm 1 when $d = l$. □

The following part is the running time/space analysis of Algorithm 1.

Theorem 4.5: *Algorithm 1 consumes $O(d(3^l n + 2^l m) + \alpha)$ time and $O(2^l nd)$ space, where $O(\alpha)$ is the time to compute all-pairs shortest paths in $G(V, E)$.* □

Proof: Time Complexity: First, $O(\alpha)$ the time to transform $G(V, E)$ into a metric closure. This is done in the preprocessing procedure, and can be omitted if we do not require the bound of performance ratio in Corollary 4.3 and Theorem 4.4.

To compute $T_g(v, \mathbf{p}, h)$ (line 7), for node v and all $\mathbf{p} \subseteq \mathbf{P}$, $O(2^l \cdot |N(v)|)$ is needed where $|N(v)|$ is the number of neighbors of node v. To compute $T_m(v, \mathbf{p}_1 \cup \mathbf{p}_2, h)$ for node v for all possible combinations of $\mathbf{p}_1$ and $\mathbf{p}_2$, consider a subset $\mathbf{p}_1 \subseteq \mathbf{P}$ where $|\mathbf{p}_1| = i$ and $|\mathbf{P}| = l$. There are $\binom{l}{i}$ ways of selecting a subset $\mathbf{p}_1$ of size i out of l from $\mathbf{P}$, and there are 2^{l-i} ways of selecting another subset $\mathbf{p}_2$ disjoint with $\mathbf{p}_1$.

So the total time complexity is as follows.

$$\begin{aligned} & O(n(\Sigma_{v\in V(G)}2^l|N(v)| + n\Sigma_{i=0}^{l}\binom{l}{i}2^{l-i})) \\ = \; & O(n(2^l m + 3^l n)) = O(3^l n^2 + 2^l nm) \end{aligned}$$

Space Complexity: $T(v, \mathbf{p}, h)$ 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 $T_g(v, \mathbf{p}, h)$ is constructed, and record $\mathbf{p}_1$ and $\mathbf{p}_2$ with which $T_m(v, \mathbf{p}_1 \cup \mathbf{p}_2, h)$ is constructed. $T(v, \mathbf{p}, h)$ can be reconstructed recursively when needed. Therefore, the space needed for $T(v, \mathbf{p}, h)$ is bounded by $O(1)$. The total space required is $O(2^l nd)$.

Note if we store the subtree explicitly in $T(v, \mathbf{p}, h)$, the space complexity becomes $O(2^l ndl)$, since the size of each subtree is bounded by $O(l)$ (there are at most l leaves). □

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-1* in Chapter 5, based on similar ideas of *DPH-1*.

□ **End of chapter.**

Chapter 5

Best-First Dynamic Programming Algorithm

Summary

This chapter introduces an exact algorithm to find the optimal *GST-1* (or min-cost group Steiner) with time complexity $O(3^l n + 2^l((l + \log n)n + m))$ and space complexity of $O(2^l n)$. Discussions on how to extend it to solve *GST-k* and some other important issues also appear here.

The algorithm *DPH-1* (Algorithm 1) shows the main idea of dynamic programming algorithm with which the optimal *GST-1* (with height bounded) can be computed.

In this chapter, 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 $T(v, \mathbf{P})$ 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.

Equation (4.1) – Equation (4.4), with height h, are rewritten to Equation (5.1) – Equation (5.4), without height h, respectively. In brief, $T(v, \mathbf{p})$ is a tree with the minimum cost, rooted

at v, containing a set of keywords $\mathbf{p} \subseteq \mathbf{P}$. Below, Equation (5.1) shows that the primitive trees, $T(v, \mathbf{p})$, which is single node tree, rooted at v, and contains keyword set $\mathbf{p}$ in v, are with a zero cost.

$$T(v, \mathbf{p}) = 0 \tag{5.1}$$

Note $T(v, \emptyset) = 0$. Like Equation (4.2), Equation (5.2) shows the general case for a tree with more than one nodes.

$$\begin{aligned} T(v, \mathbf{p}) &= \min(\mathsf{T}_{\mathsf{g}}(v, \mathbf{p}), \mathsf{T}_{\mathsf{m}}(v, \mathbf{p})) && (5.2) \\ \text{where } \mathsf{T}_{\mathsf{g}}(v, \mathbf{p}) &= \min_{u \in N(v)} \{(v, u) \oplus T(u, \mathbf{p})\} && (5.3) \\ \mathsf{T}_{\mathsf{m}}(v, \mathbf{p}_1 \cup \mathbf{p}_2) &= \min_{\mathbf{p}_1 \cap \mathbf{p}_2 = \emptyset} \{T(v, \mathbf{p}_1) \oplus T(v, \mathbf{p}_2)\} && (5.4) \end{aligned}$$

We omit further explanation of Equation (5.1)-(5.4) because they share high similarity with Equation (4.1)-(4.4). The only difference is that they do not specify height h. Again, we can prove their correctness by the induction on $|\mathbf{p}|$. The optimal substructure holds here too.

Theorem 5.1: *Given an edge-weighted undirected graph G, and a set of l keywords, $\mathbf{P} = \{p_1, p_2, \cdots, p_l\}$. 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 rooted at v can be computed using Equation (5.1) – Equation (5.4).* □

Proof: The proof can be obtained using the similar ways as the proof sketched for Theorem 4.1. □

Note the omittance of parameter h makes the proof of Theorem 5.1 (the correctness of Equation (5.1) – Equation (5.4)) simpler than Theorem 4.1, while making the algorithm *DPBF-1* (Algorithm 2) to compute Equation (5.1) – Equation (5.4) more complicated than the algorithm *DPH-1* (Algorithm 1) to compute Equation (4.1) – Equation (4.4). We will discuss the algorithm *DPBF-1*, which can achieve the promised time/space complexity, in the following part.

Algorithm 2 *DPBF-1*

input: database graph G, the set of keywords $\mathbf{P}$, 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 ← ∅;
3: for each v ∈ V(G) do
4:    if v contains keywords p then
5:       enqueue T(v, p) = 0 into Q_T;
6: while Q_T ≠ ∅ do
7:    dequeue Q_T to T(v, p);
8:    return T(v, p) if p = P;
9:    for each u ∈ N(v) do
10:      if T(v, p) ⊕ (v, u) < T(u, p) then
11:         T(u, p) ← T(v, p) ⊕ (v, u);
12:         update Q_T with the new T(u, p);
13:   p_1 ← p;
14:   for each p_2 s.t. p_1 ∩ p_2 = ∅ do
15:      if T(v, p_1) ⊕ T(v, p_2) < T(v, p_1 ∪ p_2) then
16:         T(v, p_1 ∪ p_2) ← T(v, p_1) ⊕ T(v, p_2);
17:         update Q_T with the new T(v, p_1 ∪ p_2);
```

5.1 An Efficient Best-First Algorithm

Based on (5.1)-(5.4), we outline the best-first strategy dynamic programming algorithm, called *DPBF-1*, in Algorithm 2.

In *DPBF-1*, for simplicity and brevity, we use $T(v, \mathbf{p})$ for the tree structure and its cost. Recall in Equation (5.1) – Equation (5.4), 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 $T(v, \mathbf{p})$ into the queue Q_T and Q_T is updated to maintain the increasing order of costs of trees. *Dequeue* remove the top tree $T(v, \mathbf{p})$ in Q_T. *Update* operation first enqueues $T(v, \mathbf{p})$ if it does not exist in Q_T, and update Q_T to maintain

the increasing order of costs.

Remark 5.2: *Let $T(v, \mathbf{p})$ be the tree at the top of Q_T. It is important to know that $T(v, \mathbf{p})$ is with the minimum cost among all trees rooted at v, containing the same set of keywords $\mathbf{p}$. This is because tree grow and tree merge can only get trees with larger costs. Equation (5.2) is ensured by Q_T.* □

Initialization and Outline: In *DPBF-1*, it first initializes Q_T to be empty (line 2). In line 3-5, it enqueues $T(v, \mathbf{p})$ into Q_T if node v contains a subset of keywords $\mathbf{p}$ ($\subseteq \mathbf{P}$) (Equation (5.1)). 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 $T(v, \mathbf{p})$ from Q_T, which is with the smallest cost in all trees in Q_T. Note: $T(v, \mathbf{p})$ is rooted at node v. If $T(v, \mathbf{p})$ contains the entire set of keywords $\mathbf{P}$ ($\mathbf{p} = \mathbf{P}$), the algorithm will return $T(v, \mathbf{p})$ and terminate (line 8: $T(v, \mathbf{p})$ is optimal here because of Remark 5.2).

Tree Grow: In line 9-12, the algorithm considers the neighbors, u, of the node v, which is the root of the tree $T(v, \mathbf{p})$ just dequeued. It attempts to reduce the cost of trees rooted at u, by utilizing the cost information associated with the keywords $\mathbf{p}$ that $T(v, \mathbf{p})$ has. Here, a neighbor $u \in N(v)$ 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 $T(v, \mathbf{p}) \oplus (v, u)$ has a smaller cost than $T(u, \mathbf{p})$. If yes, the tree $T(u, \mathbf{p})$ will be updated to $T(v, \mathbf{p}) \oplus (u, v)$, and the cost of $T(u, \mathbf{p})$ becomes smaller. Note $T(u, \mathbf{p})$ may or may not exist in Q_T. If not, $T(u, \mathbf{p}) = T(v, \mathbf{p}) \oplus (u, v)$ 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 $\mathbf{p}'$ is similar. The line 9-12 handles the case of *tree grow* (Equation (5.3)).

Tree Merge: The case of *tree merge* (Equation (5.4)) is handled

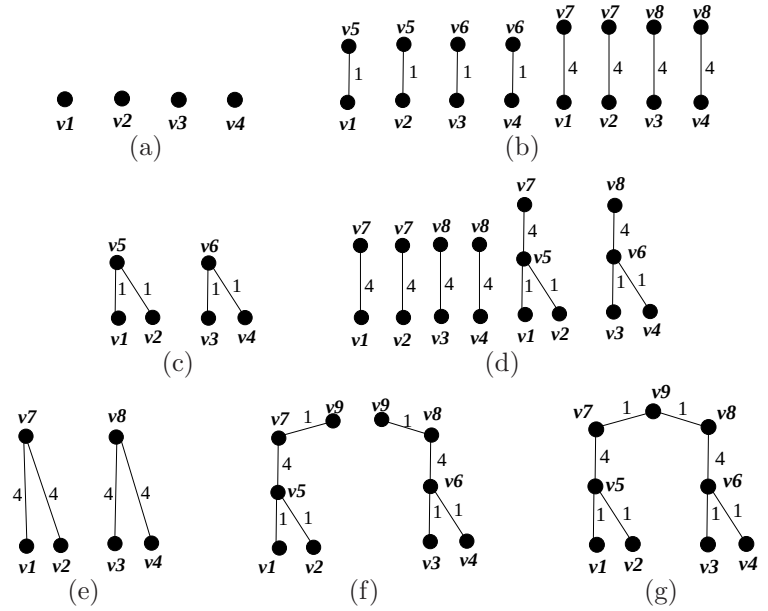


Figure 5.1: A Best-First DP Solution: *DPBF-1*

in line 13-17. There are many trees rooted at the same node v containing different subsets of keywords, $\mathbf{p}_1$, $\mathbf{p}_2$, the algorithm considers every possible disjoint pair of $\mathbf{p}_1$ and $\mathbf{p}_2$, and tries to reduce the cost of $T(v, \mathbf{p}_1 \cup \mathbf{p}_2)$. In line 15, it checks if the tree $T(v, \mathbf{p}_1) \oplus T(v, \mathbf{p}_2)$ has a smaller cost than $T(v, \mathbf{p}_1 \cup \mathbf{p}_2)$. If yes, the tree $T(v, \mathbf{p}_1 \cup \mathbf{p}_2)$ will be updated to $T(v, \mathbf{p}_1) \oplus T(v, \mathbf{p}_2)$, and the cost of $T(v, \mathbf{p}_1 \cup \mathbf{p}_2)$ becomes smaller. Note $T(v, \mathbf{p}_1 \cup \mathbf{p}_2)$ may or may not exist in Q_T. If not, $T(v, \mathbf{p}_1 \cup \mathbf{p}_2) = T(v, \mathbf{p}_1) \oplus T(v, \mathbf{p}_2)$ is enqueued into Q_T. Then Q_T will be updated. In both cases, Q_T ensures the increasing order of costs.

Theorem 5.3: *Algorithm 2 (DPBF-1) can correctly compute the optimal GST-1.* □

Proof: It will be shown in Section 5.4.1 that Algorithm 2 is an analogue of Dijkstra's algorithm for the shortest path problem in the space composed of (partial) Steiner trees $\{T(v, \mathbf{p})\}$. So the proof here is very similar to the proof of Dijkstra's algorithm.

As in the description of Algorithm 2, we abuse the notations a bit: let $T(v, \mathbf{p})$ be either a tree rooted at node v and containing keywords $\mathbf{p}$, or its cost $s(T(v, \mathbf{p}))$.

Let $T^*(v, \mathbf{p})$ be the min-cost tree rooted at node v and containing keywords $\mathbf{p}$. We need to prove, based on Equation (5.1) – Equation (5.4), Algorithm 2 terminates with $T(v, \mathbf{p}) = T^*(v, \mathbf{p})$ for all $T(v, \mathbf{p})$'s dequeued. Therefore, the $T(v, \mathbf{P})$ returned in line 8 is the optimal *GST-1*.

We first prove the following two properties of Algorithm 2.

No-tree property: If there is no tree rooted at node v and containing keywords $\mathbf{p}$, then $T(v, \mathbf{p}) = T^*(v, \mathbf{p}) = \infty$. This is directly gotten from the definitions.

Upper-bound property: We always have $T(v, \mathbf{p})$ is a upper bound of $T^*(v, \mathbf{p})$, i.e. $T(v, \mathbf{p}) \geq T^*(v, \mathbf{p})$, in Algorithm 2. Initially from line 5 (according to Equation (5.1)), this is true. Line 11 (according to Equation (5.3)) and line 16 (according to Equation (5.4)) update $T(v, \mathbf{p})$ by constructing a new tree in the tree-grow case and the tree-merge case respectively. Since both $T^*(v, \mathbf{p})$ and $T(v, \mathbf{p})$ are trees rooted at v and containing keywords vp, and $T^*(v, \mathbf{p})$ is the min-cost one[1], we have $T(v, \mathbf{p}) \geq T^*(v, \mathbf{p})$.

Now we use the following **loop invariant** to prove this theorem: *At line 7 of each iteration of the while loop of line 6-17,* $T(v, \mathbf{p}) = T^*(v, \mathbf{p})$, *for* $T(v, \mathbf{p})$ *dequeued from* Q_T.

Initialization: Initially, from Equation (5.1), the invariant is trivially true.

Maintenance: We wish to show that at line 7, $T(v, \mathbf{p}) = T^*(v, \mathbf{p})$, for every $T(v, \mathbf{p})$ dequeued from Q_T. *For the purpose of contradiction*, let $T(\bar{v}, \bar{\mathbf{p}})$ be the first tree to be dequeued from Q_T, such that $T(\bar{v}, \bar{\mathbf{p}}) \neq T^*(\bar{v}, \bar{\mathbf{p}})$.

There must be some tree rooted at $\bar{v}$ and containing keywords $\bar{\mathbf{p}}$, for otherwise $T^*(\bar{v}, \bar{\mathbf{p}}) = \infty$ by the no-tree property,

[1]Actually, we will prove $T(v, \mathbf{p})$ is also min-cost.

which would violate our assumption that $T(\bar{v}, \bar{\mathbf{p}}) \neq T^*(\bar{v}, \bar{\mathbf{p}})$. We examine the min-cost tree $T^*(\bar{v}, \bar{\mathbf{p}})$ using the following procedure:

1: $v \leftarrow \bar{v}$, $\mathbf{p} \leftarrow \bar{\mathbf{p}}$;
2: **while** true **do**
3: Let $v_1, \cdots, v_c$ be the children of v in $T^*(v, \mathbf{p})$
4: Let $\mathbf{p}_1, \cdots, \mathbf{p}_c$ be the sets of keywords contained in v and v's subtrees rooted at $v_1, \cdots, v_c$;
5: **if** there exists some $\mathbf{p}_i$ s.t. $T(v, \mathbf{p}_i)$ has not been dequeued from Q_T **then**
6: **if** $T(v_i, \mathbf{p}_i)$ has not been dequeued from Q_T **then**
7: $v \leftarrow v_i$, $\mathbf{p} \leftarrow \mathbf{p}_i$;
8: **else**
9: **terminate with** v and $\mathbf{p}_i$;
10: **else**
11: **terminate with** v and $\mathbf{p}$;
12: **terminate if** $i = k$;

The above procedure will finally terminate, because each leaf of $T^*(v, \mathbf{p})$ is a 0-cost tree, which was enqueued into Q_T in line 3-5 of Algorithm 2, has been dequeued already (Q_T is a priority queue). When it terminates, there are two cases:

1. Terminate at line 9 (corresponding to tree-grow): Consider $T(v, \mathbf{p}_i)$ and $T(v_i, \mathbf{p}_i)$. Note $T(v_i, \mathbf{p}_i)$ has been dequeued from Q_T, thus

$$T(v_i, \mathbf{p}_i) = T^*(v_i, \mathbf{p}_i).$$

Because i) in line 9-12 of Algorithm 2, $T(v, \mathbf{p}_i)$ has been updated as $T(v_i, \mathbf{p}_i) \oplus (v_i, v)$, and ii) in the min-cost tree $T^*(\bar{v}, \bar{\mathbf{p}})$, $T^*(v, \mathbf{p}_i) = T^*(v_i, \mathbf{p}_i) \oplus (v_i, v)$ (note v_i is a child of v in $T^*(\bar{v}, \bar{\mathbf{p}})$), we must have

$$T(v, \mathbf{p}_i) = T(v_i, \mathbf{p}_i) \oplus (v_i, v) = T^*(v_i, \mathbf{p}_i) \oplus (v_i, v) = T^*(v, \mathbf{p}_i).$$

For $T^*(v, \mathbf{p}_i)$ is a subtree of $T^*(\bar{v}, \bar{\mathbf{p}})$, we have

$$T^*(v, \mathbf{p}_i) \leq T^*(\bar{v}, \bar{\mathbf{p}}).$$

Therefore, $T(v, \mathbf{p}_i) = T^*(v, \mathbf{p}_i) \leq T^*(\bar{v}, \bar{\mathbf{p}}) \leq T(\bar{v}, \bar{\mathbf{p}})$ (by the upper-bound property). But because both $T(v, \mathbf{p}_i)$ and $T(\bar{v}, \bar{\mathbf{p}})$ are not dequeued from Q_T, we have $T(\bar{v}, \bar{\mathbf{p}}) \leq T(v, \mathbf{p}_i)$. These two inequalities give $T(v, \mathbf{p}_i) = T^*(v, \mathbf{p}_i) = T^*(\bar{v}, \bar{\mathbf{p}}) = T(\bar{v}, \bar{\mathbf{p}})$, which violates our assumption.

2. Terminate at line 11 (corresponding to tree-merge): The following discussion is similar to case 1. Consider $T(v, \mathbf{p})$ and $T(v, \mathbf{p}_1), \cdots, T(v, \mathbf{p}_c)$. Note $T(v, \mathbf{p}_1), \cdots, T(v, \mathbf{p}_c)$ have been dequeued from Q_T, thus for $i = 1, \cdots, c$,

$$T(v, \mathbf{p}_i) = T^*(v, \mathbf{p}_i).$$

Because i) in line 13-17 of Algorithm 2, $T(v, \mathbf{p})$ has been updated as $T(v, \mathbf{p}_1) \oplus T(v, \mathbf{p}_2) \oplus \cdots \oplus T(v, \mathbf{p}_c)$, and ii) in the min-cost tree $T^*(\bar{v}, \bar{\mathbf{p}})$, $T^*(v, \mathbf{p}) = T^*(v, \mathbf{p}_1) \oplus T^*(v, \mathbf{p}_2) \oplus \cdots \oplus T^*(v, \mathbf{p}_c)$, we must have

$$\begin{aligned} T(v, \mathbf{p}) &= T(v, \mathbf{p}_1) \oplus T(v, \mathbf{p}_2) \oplus \cdots \oplus T(v, \mathbf{p}_c) \\ &= T^*(v, \mathbf{p}_1) \oplus T^*(v, \mathbf{p}_2) \oplus \cdots \oplus T^*(v, \mathbf{p}_c) \\ &= T^*(v, \mathbf{p}). \end{aligned}$$

For $T^*(v, \mathbf{p})$ is a subtree of $T^*(\bar{v}, \bar{\mathbf{p}})$, we have

$$T^*(v, \mathbf{p}) \leq T^*(\bar{v}, \bar{\mathbf{p}}).$$

Therefore, $T(v, \mathbf{p}) = T^*(v, \mathbf{p}) \leq T^*(\bar{v}, \bar{\mathbf{p}}) \leq T(\bar{v}, \bar{\mathbf{p}})$ (by the upper-bound property). But because both $T(v, \mathbf{p})$ and $T(\bar{v}, \bar{\mathbf{p}})$ are not dequeued from Q_T, we have $T(\bar{v}, \bar{\mathbf{p}}) \leq T(v, \mathbf{p})$. These two inequalities give $T(v, \mathbf{p}) = T^*(v, \mathbf{p}) = T^*(\bar{v}, \bar{\mathbf{p}}) = T(\bar{v}, \bar{\mathbf{p}})$, which violates our assumption.

These two cases shows the equality $T(v, \mathbf{p}) = T^*(v, \mathbf{p})$ is maintained now and at all times thereafter.

Termination: At termination (line 8 of Algorithm 2), $\mathbf{p} = \mathbf{P}$ which, along with our earlier loop invariant $T(v, \mathbf{p}) = T^*(v, \mathbf{p})$, implies that $T(v, \mathbf{P})$ output here is the optimal *GST-1*. □

Note: by deleting line 8 of Algorithm 2, we can find $T(v, \mathbf{p}) = T^*(v, \mathbf{p})$ for all v's and $\mathbf{p}$'s, and output them in the increasing order of cost. The correctness can be easily verified by generalizing the proof above. This idea will be used in our top-k algorithm *DPBF-k* (to find *GST-k*) introduced in Section 5.3.

Example 5.1: We explain *DPBF-1* (Algorithm 2) with the same database graph in Figure 4.2 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 5.1 (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 (Equation (5.1)). Figure 5.1 (b) shows Q_T after the first 4 iterations of the **while** statement in *DPBF-1*. Here, all 4 trees in Figure 5.1 (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 5.1 (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 5.1 (b)) are dequeued. The costs of the trees in Figure 5.1 (c) are both 2, so they are ranked as the first two in Q_T. Figure 5.1 (d) shows the first 6 trees in Q_T after the next 2 iterations. Here, the first 4 trees in Figure 5.1 (d) are the 5-th to 8-th trees in Figure 5.1 (b), and the last trees in Figure 5.1 (d) are newly constructed from the 2 trees in 5.1 (c) based on the case of tree grow. Figure 5.1 (e) shows the trees constructed from the first 4 trees in Q_T (Figure 5.1 (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 5.1 (d). Note: they share the same roots and contain the same keywords. Figure 5.1 (f)-(g) show the rest iterations based on *tree grow* and *tree merge*, respectively. The optimal *GST-1* is shown in Figure 5.1 (g). □

5.2 Time/Space Complexity

We analyze the time/space complexity of Algorithm 2 in this Section.

Theorem 5.4: *Algorithm 2 consumes* $O(3^l n + 2^l((l + \log n)n + m))$ *time and* $O(2^l n)$ *space.* □

Proof: **Time Complexity**: Let $T(v, \mathbf{p})$ be the minimum cost for a tree rooted at every $v \in V(G)$ containing a subset of keywords, $\mathbf{p} \subseteq \mathbf{P}$ where $l = |\mathbf{P}|$. There are totally n nodes, and 2^l subsets of $\mathbf{P}$. So the length of Q_T will be at most $2^l n$. Note: any $T(v, \mathbf{p})$ will be enqueued/dequeued into/from Q_T at most once in *DPBF-1*. With Fibonacci Heap [12], the cost for enqueue/update and dequeue are $O(1)$ and $O(\log 2^l n)$, respectively. The total cost for the dequeue is $O(2^l n(l + \log n))$, for all $2^l n$ number of $T(v, \mathbf{p})$'s.

Equation (5.3) is computed in line 9-12 to minimize $T(u, \mathbf{p})$, where $u \in N(v)$, using the information of $T(v, \mathbf{p})$. 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 $O(2^l \Sigma_{v \in V} |N(v)|) = O(2^l m)$. Q_T need be updated (in time $O(1)$), if a smaller cost for $T(u, \mathbf{p})$ is found in line 10. So the total time needed for line 9-12 is $O(2^l m)$.

Equation (5.4) is computed in line 13-17 to minimize $T(u, \mathbf{p}_1 \cup \mathbf{p}_2)$, for every pair of non-empty disjoint $\mathbf{p}_1$ and $\mathbf{p}_2$. Let $\mathbf{p}_1$

be $\mathbf{p}$ of $T(v, \mathbf{p})$ dequeued in this iteration. With $T(v, \mathbf{p}_1)$ in hand, it enumerates $T(v, \mathbf{p}_2)$. If a lower cost of $T(v, \mathbf{p}_1 \cup \mathbf{p}_2)$ is found (line 15), Q_T is updated. If $|\mathbf{p}_1| = i$ is fixed, the total number of possible $\mathbf{p}_1$ is $\binom{l}{i}$, and the total number of possible $\mathbf{p}_2$ is 2^{l-i}, so the number of comparisons in line 15 is bounded by $O(n \sum_{i=0}^{l} \binom{l}{i} 2^{l-i}) = O(3^l n)$. Because the time needed by an update of Q_T is $O(1)$ (with Fibonacci Heap [12]), the total time needed for line 13-17 is $O(3^l n)$.

So the time complexity is $O(3^l n + 2^l((l + \log n)n + m))$.

Space Complexity: $T(v, \mathbf{p})$ 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 $T_g(v, \mathbf{p})$ is constructed, and record $\mathbf{p}_1$ and $\mathbf{p}_2$ with which $T_m(v, \mathbf{p}_1 \cup \mathbf{p}_2)$ is constructed. $T(v, \mathbf{p})$ can be reconstructed recursively when needed. Therefore, the space needed for $T(v, \mathbf{p})$ is bounded by $O(1)$, and the total space required to store $T(v, \mathbf{p})$ if bounded by $O(2^l n)$. The maximum size of Q_T is also bounded by $O(2^l n)$. So the space complexity is $O(2^l n)$. □

5.3 Finding Top-k Group Steiner Trees

To compute *GST-k*, we propose algorithm *DPBF-k*, by just replacing line 8 in Algorithm 2 (*DPBF-1*) with:

if $\mathbf{p} = \mathbf{P}$ **then**
  $i \leftarrow i + 1$;
  output $T_i = T(v, \mathbf{p})$;
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_T in *DPBF-1*, we can find T_1, T_2, ... 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-1*, because the worst case for solving *GST-1* is to search all possible trees in $\{T(v, \mathbf{p})\}$, which is the same as for *GST-k*.

Note $\{T_1, \cdots, T_k\}$ output by algorithm *DPBF-k* is an approximation of *GST-k*. Suppose $\{T_1^*, \cdots, T_k^*\}$ is the optimal *GST-k*, by Theorem 5.3, we can only promise that $s(T_1) = s(T_1^*)$, but may have $s(T_i) \geq s(T_i^*)$ $(1 < i \leq k)$. The reason is that for example, T_2^* may have the same set of nodes as T_1^*, but with different edges. Such T_2^* cannot be found by *DPBF-k*, because $T_2 = T(u, \mathbf{P})$ must differ from $T_1 = T(v, \mathbf{P})$ on at least one node.

However, *DPBF-k* can promise that $\{T_1, \cdots, T_k\}$ contains all the nodes covered by $\{T_1^*, \cdots, T_k^*\}$ for any fixed k. It means all the nodes in the optimal top-k answers will be output by *DPBF-k*. This feature of *DPBF-k* satisfies the need of users, because users will not miss any "desired" nodes (representing, e.g., *tuples*, and *webpages*) in the list of top-k answers (recall an answer is a tree, consisting of tuples in the database). Formally, we have the following theorem.

Theorem 5.5: *Suppose* $\{T_1, \cdots, T_k\}$ *is the GST-k output by algorithm DPBF-k, and* $\{T_1^*, \cdots, T_k^*\}$ *is optimal GST-k. Then for any fixed* k*, if* $s(T_k^*) < s(T_k)$ *(*$\{T_1, \cdots, T_k\}$ *is not the optimal GST-k), we have*

$$\bigcup_{1 \leq i \leq k} V(T_i) \supseteq \bigcup_{1 \leq i \leq k} V(T_i^*). \tag{5.5}$$

□

Proof: To prove this theorem by contradiction, we assume that: there exists a node

$$v \in \bigcup_{1 \leq i \leq k} V(T_i^*),$$

but

$$v \notin \bigcup_{1 \leq i \leq k} V(T_i). \tag{5.6}$$

Without of loss of generality, suppose $v \in V(T_j^*)$ for some $1 \leq j \leq k$. Because of the optimality of $\{T_1^*, \cdots, T_k^*\}$, we must have

$$s(T_j^*) \leq s(T_k^*) < s(T_k).$$

Consider the tree $T(v, \mathbf{P})$. From Theorem 5.3, $T(v, \mathbf{P})$ can be correctly computed, and because of the definition of $T(v, \mathbf{P})$ (min-cost tree rooted at v, containing all keywords $\mathbf{P}$), we have

$$s(T(v, \mathbf{P})) \leq s(T_j^*).$$

Thus, we have

$$s(T(v, \mathbf{P})) < s(T_k),$$

which means $T(v, \mathbf{P})$ must have been output by algorithm *DPBF-k*. This contradicts to our assumption (Equation (5.6)), and completes the proof. □

Our algorithm *DPBF-1* can be also applied in the algorithmic framework proposed in [29], to find the optimal *GST-k*. Theorem 5.5 can be used to further improve the performance of exact top-k algorithms in this framework. Because it points out that although algorithm *DPBF-k* cannot output the optimal *GST-k*, it can output the set of nodes, which the trees in the optimal *GST-k* consist of. With this information, we need only run the exact top-k algorithm in a subgraph of database graph G.

5.4 Other Important Issues

In this section, we discuss several additional issues: (1) comparison with an unpublished work (independently developed by Benny Kimelfeld and Yehoshua Sagiv [28]), (2) handling general cost functions with node/edge weights, (3) handling keyword

queries with logical operators, (4) handling directed database graph, (5) the size and the maintenance of database graph, and (6) indexing and pruning strategy.

We will first show the difference between our algorithm *DPBF-1/k* and the solution in [28], and then show that we can handle keyword queries with logical opeators in the database graph G as a weighted undirected/directed graph with both node-weights and edge-weights. We will give the class of cost functions we can support with our dynamic programming algorithms. Finally, we propose the strategies for indexing and pruning.

5.4.1 Comparison with an Unpublished Work

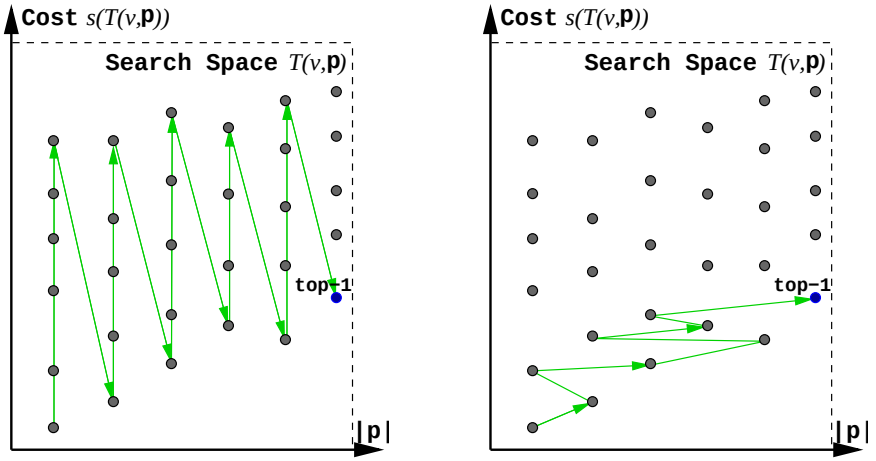
Benny Kimelfeld and Yehoshua Sagiv [28] proposed a *GST-1* algorithm also based on Equation (5.1) – Equation (5.4), independently. To compare their solution with ours, consider $\{T(v, \mathbf{p})\}$, the set of all (partial) Steiner trees, as the *search space*. Our task is to explore this space under the guidance of Equation (5.1) – Equation (5.4), to find $T(v, \mathbf{P})$ with the minimum cost. Because of the absence of parameter h, the exploration must follow such an order.

Requirement of the Order to Compute $T(v, \mathbf{p})$: *IF* $T(v', \mathbf{p}')$ *is a subtree of* $T(v, \mathbf{p})$, *THEN* $T(v', \mathbf{p}')$ *must be computed earlier than* $T(v, \mathbf{p})$.

So as shown in Figure 5.2, there are two possible orders satisfying this requirement.

1. The ascending order of $|\mathbf{p}|$ (size of keywords set $\mathbf{p}$): The algorithm in [28] is based on this order. It visits nearly the whole search space for every input, because the set of all keyword $\mathbf{P}$ has the largest size.

2. The ascending order of $s(T(v, \mathbf{p}))$ (cost of tree $T(v, \mathbf{p})$): Our *DPBF-1* is based on this order. It follows a shortcut to *GST-1*, and visits only the necessary portion of the



(a) Ascending Order of $|\mathbf{p}|$ (b) Ascending Order of $s(T(v, \mathbf{p}))$

Figure 5.2: Two possible orders to compute $T(v, \mathbf{p})$

search space. In this view of point, our *DPBF-1* is an analogue of Dijkstra's algorithm in the space $\{T(v, \mathbf{p})\}$, where each $T(v, \mathbf{p})$ is considered as a node, and relationships specified by Equation (5.1) – Equation (5.4) are considered as weighted edges (Equation (5.3) and Equation (5.4) specify two types of edges respectively). We aim to find the shortest path from $\{T(v, \emptyset) : v \in V(G)\}$ to $\{T(v, \mathbf{P}) : v \in V(G)\}$.

Therefore, our *DPBF-1* can correct compute *GST-1*, and always explore a smaller portion of search space than the algorithm in [28] does.

5.4.2 General 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-1/k* can support any additive cost function $s'(T)$ in the form shown in Equation (5.7)-(5.9), and

any nonnegative weights $w_v(v)$ and $w_e(e)$.

$$s'(T) = (1 - \lambda) \cdot s'_v(T) + \lambda \cdot s'_e(T). \tag{5.7}$$

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,

$$s'_v(T) = \sum_{v \in V(T) \cap \mathcal{V}} w_v(v) \tag{5.8}$$

$$s'_e(T) = \sum_{e \in E(T)} w_e(e) \tag{5.9}$$

$s'_v(T)$ is the total node weight of those nodes in T that also appear in $\mathcal{V}$, where $\mathcal{V}$ is the set of nodes in G that contain at least one keyword. $s'_e(T)$ is the total edge weight in T. Note Equation (2.1) is a special case of Equation (5.7) with $\lambda = 1$. Handling node-weight in *GST-1* (or *GST-k*) does not increase the complexity, as can be sensed in the discussions in Section 5.2, Chapter 5. Node weights can be assigned using the exsiting approaches given in [8, 4].

It should be mentioned that there are many other possible ranking methods. Comparison between alternatives need involve large-scale user studies, which is beyond the scope of this thesis.

5.4.3 Keyword Queries with Logical Operators

The l keyword query, with a set of keywords, p_1, p_2, $\cdots$, p_l, can be considered as a conjunctive query, $p_1 \wedge p_2 \wedge \cdots \wedge p_l$, because it needs to find a group Steiner tree containing every keyword. We can handle a keyword query with logical operators $\vee$, as well as $\wedge$. Note $p_1 \vee p_2$ means the group Steiner tree may contain either p_1 or p_2.

Simply put, to answer a keyword query in the general form: $p_1 \otimes_1 p_2 \otimes_2 \cdots \otimes_{l-1} p_l$, where $\otimes_i = \wedge$ or $\vee$, we only need:

1. compute a collection of keywords set

   $\mathcal{P} = \{\mathbf{p} : \mathbf{p} \subseteq \mathbf{P},\ \mathbf{p}$ satisfies query $p_1 \otimes_1 p_2 \otimes_2 \cdots \otimes_{l-1} p_l\}$.

   For query $(p_1 \wedge p_2) \vee p_3$, e.g., $\mathcal{P} = \{\{p_1, p_3\}, \{p_2, p_3\}\}$.

2. replace line-8 of Algorithm 2 (*DPBF-2*) with:
   return $T(v, \mathbf{p})$ if $\mathbf{p} \in \mathcal{P}$;

   Top-k algorithm *DPBF-k* can be extended in a similar way.

Using bit-wise implementation, the time complexity of algorithm *DPBF-1/k* is unchanged.

5.4.4 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-1/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.

5.4.5 Graph Size and Graph Maintenance

As other tuple-based approaches [7, 36], 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 2.2, G is sparse. So the memory consumption for the materialized database graph G is small. Following is the quantitative analysis.

Suppose that a node needs 6 bytes to uniquely identify a tuple (node) in $\mathcal{DB}$ (2 bytes for unique relation identifiers and 4 bytes for unique tuple identifiers in a relation). And suppose that each node is associated with a data structure for (out-going) edges where an edge is occupied 4 bytes. The memory consumption for

G becomes, $6n + 4\overline{r}n$ where $\overline{r}$ is the average number of tuples that a tuple in DB references to using foreign key references. Note $\overline{r} < r$ based on Remark 2.2. In other words, the byte consumption per node is $6 + 4\overline{r}$. Consider a real database graph G for `DBLP` in year 2004 [30]. The number of nodes is $n =$ 1900K, and the number of edges is $m =$ 5400K. The average number of outgoing edges per node is $\overline{r} = 3$. 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. Given a node identifier, v, $H_v(v)$ finds its location, and $H_e(v)$ finds all u that reference to v. When insert/delete a tuple or a foreign key in $\mathcal{DB}$, hash structures H_v and H_e are also updated.

5.4.6 Indexing and Pruning Strategy

We can speedup the algorithm *DPBF-1* based on these two observations: (i) The number of non-empty $T(v, \mathbf{p})$ is very small. (ii) The cost of $T(v, \mathbf{p})$ is monotone with respect to $\mathbf{p}$, i.e. $s(T(v, \mathbf{p})) \leq s(T(v, \mathbf{p}')) \Leftrightarrow \mathbf{p} \subseteq \mathbf{p}'$. Based on the observations, we speedup Algorithm *DPBF-1* with indexing and pruning.

Indexing: Theoretically, the size of search space is $O(2^l \cdot n)$. But in practice, the number of $T(v, \mathbf{p})$ searched by the algorithm *DPBF-1* is not large. In our extensive experimental studies, we found that the number of $T(v, \mathbf{p})$ accessed in Algorithm 2 is only about 10% - 40% of the whole search space, even when $k = 100$. This is because that there is only a small portion of nodes in database graph that is closely related to a user-given keyword.

An index is built on node, say v, which assists us to efficiently identify $\mathbf{p}_2$ such that $T(v, \mathbf{p}_2)$ is non-empty (line 14 in Algorithm

2). Because only the non-empty $T(v, \mathbf{p}_2)$ is useful in the Tree Merge process (line 14-17). Without such an index, we need always enumerate up to 2^l possible values for $\mathbf{p}_2$. Such index can be easily implemented with hash table and double-link list.

Pruning: Recall $T(v, \mathbf{p})$ is defined as a tree with the minimum cost, rooted at v, containing keywords $\mathbf{p} \subseteq \mathbf{P}$. The following lemma is obvious from the definition.

Lemma 5.6: $T(v, \mathbf{p}) \leq T(v, \mathbf{p}')$ *if* $\mathbf{p} \subseteq \mathbf{p}'$. □

Based on this lemma, the following steps can be added below line 7 of Algorithm 2.

for each $\mathbf{p}' \supset \mathbf{p}$ such that $T(v, \mathbf{p}') \in Q_T$ **do**
  if $T(v, \mathbf{p}') = T(v, \mathbf{p})$ **then**
    goto 7;

The intuition of the above prune strategy is: for certain $T(v, \mathbf{p})$, if there is another tree $T(v, \mathbf{p}')$ containing more keywords but with the same cost, then just discard $T(v, \mathbf{p})$. $T(v, \mathbf{p}')$ can be used to accomplish the Tree-Grow / Tree-Merge process instead of $T(v, \mathbf{p})$ in some iteration later (when $T(v, \mathbf{p}')$ is dequeued), because $\mathbf{p} \supset \mathbf{p}'$ and $T(v, \mathbf{p}') = T(v, \mathbf{p})$.

With careful implementation to check whether there is any such $T(v, \mathbf{p}') \in Q_T$, the time complexity of algorithm *DPBF-1/k* can be unchanged.

□ **End of chapter.**

Chapter 6

Experimental Studies

Summary

This chapter introduces the experiments we conducted and analyzes the results.

We conducted extensive experimental studies to compare our parameterized solution, *DPBF*, with four algorithms, namely, *BANKS-I* [7], *BANKS-II* [36], *RIU-E* [31], and *RIU-T* [31]. 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 (Equation (2.1)), 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-1*, because *DPBF* outperforms *DPH-1*. We do not compare our results with the work in [25, 6, 22, 9, 10, 19], 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.

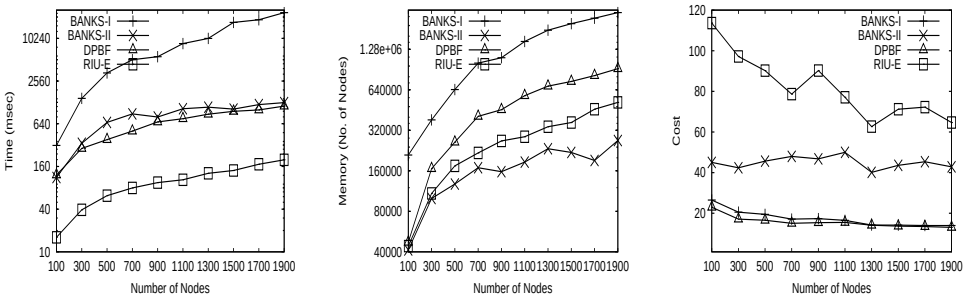
We used two real datasets, `DBLP` [30] and `MDB` [34]. The database schema of `DBLP` is outlined in Figure 1.1 (a). `DBLP` consists of $1,900K$ records for research papers archived up to year 2004. `MDB` consists of 1 million records for a movie recommendation system. It contains of 2,811,983 ratings entered by 72,916 users for 1,628 different movies.

6.1 Exp-1 Scalability

We first conducted a scalability test, because it is critical whether an algorithm can compute *GST-1* 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.

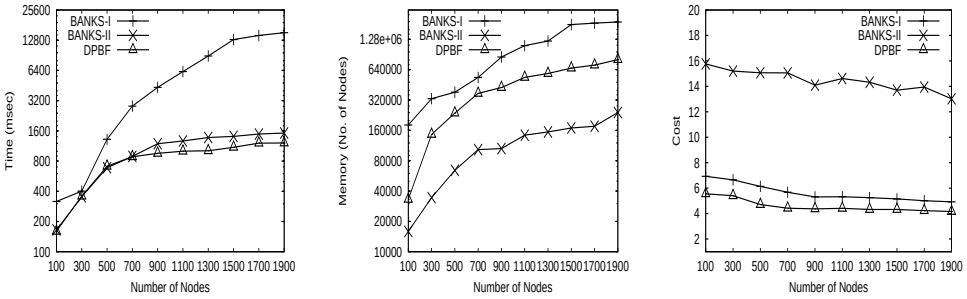
Results of tests on 10 undirected graphs are shown in Figure 6.1, *RIU-E* is the best in terms of processing time and is the worst in terms of cost, because it uses a simple heuristics to select edges when expanding. On the other hand, *BANKS-I* is the worst in terms of processing time but computes *GST-1* 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-1*, and outperforms *BANKS-I* and *BANKS-II* in terms of processing time. As shown in Figure 6.1 (b), our *DPBF* consumes less memory than that of *BANKS-I*.

Results of tests on 10 directed graphs are shown in Figure 6.2. *DPBF* finds the optimal *GST-1*, and outperforms *BANKS-I/II*.



(a) Time (b) Memory (c) Cost

Figure 6.1: Scalability over Undirected Graphs



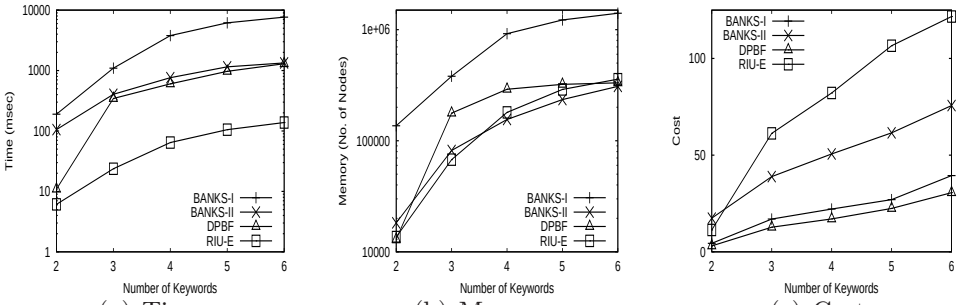
(a) Time (b) Memory (c) Cost

Figure 6.2: Scalability over Directed Graphs

We do not report *RIU-E* in Figure 6.2, because of its large average cost obtained (see Figure 6.1).

We do not include *RIU-T* in Figure 6.1-6.2, because it consumes much more processing time and memory than others to compute *GST-1* 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.



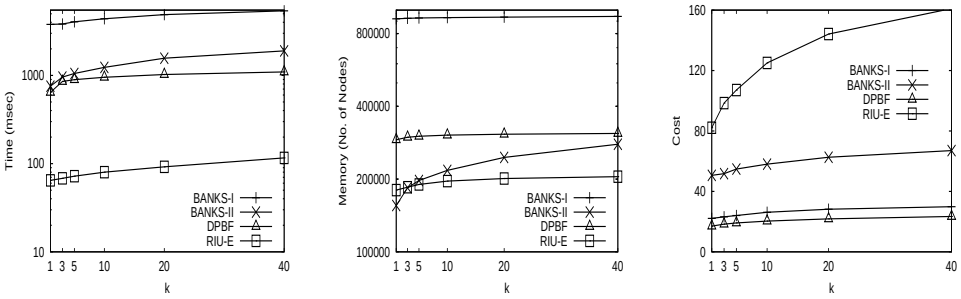
(a) Time (b) Memory (c) Cost

Figure 6.3: Varying l (Undirected Graph)

6.2 Exp-2 Number of Keywords

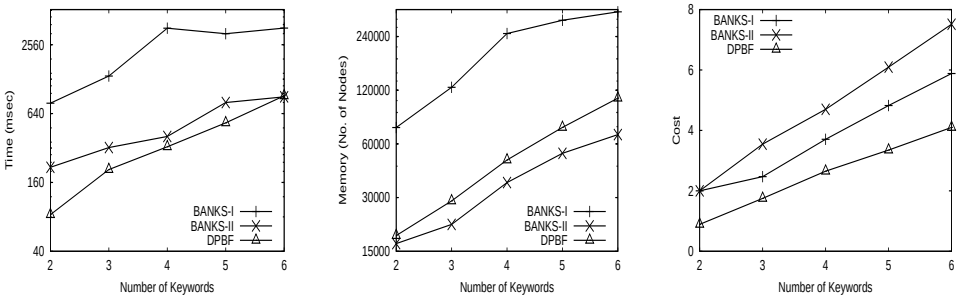
We vary the number of keywords l, from 2 to 6, to compute *GST-1*. For each l value, we randomly generate 100 queries to test. Results are shown in Figure 6.3. *DPBF* finds the optimal *GST-1*, and outperforms *BANKS-I*, *BANKS-II* and *RIU-E* in terms of cost. As shown in Figure 6.3, the processing 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-1* 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-1* is high. In Figure 6.3, the 100 queries are selected uniformly from different patterns.



(a) Time (b) Memory (c) Cost

Figure 6.4: Varying k (Undirected Graph)

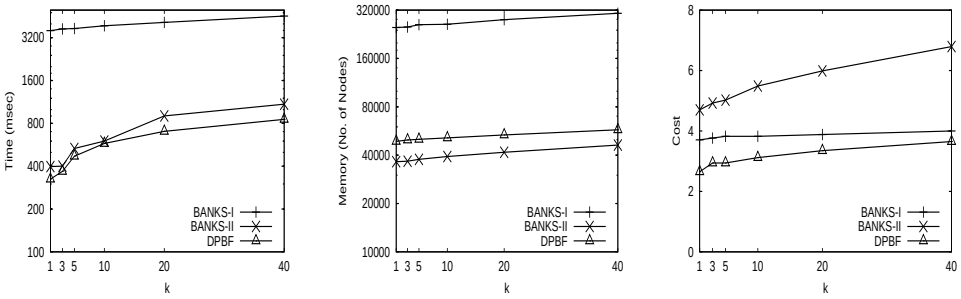


(a) Time (b) Memory (c) Cost

Figure 6.5: Varying l (MDB)

6.3 Exp-3 GST-k Testing

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 6.4. From Figure 6.4 (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-1*. Our *DPBF* outperforms the others in terms of cost, and *DPBF* outperforms *BANKS-I/II* in terms of processing time.



(a) Time (b) Memory (c) Cost

Figure 6.6: Varying k (MDB)

6.4 Exp-4 MDB (A Directed Graph Dataset)

We test `MDB` dataset [34] 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 6.5. Then, we fixed $l = 4$, and randomly generated 100 4-keyword queries to test *GST-k*, for $k = 1, 3, ...40$. The results are shown in Figure 6.6. We obtain similar results. Our *DPBF* outperforms the others in terms of both cost and processing time.

□ **End of chapter.**

Chapter 7

Conclusions

Summary

This chapter concludes the thesis.

In this thesis, we studied memory-based algorithms to find top-k min-cost group Steiner trees, denoted *GST-k*, for l-keyword queries, in a relational database which can be modelled as a graph G, with n nodes and m edges. We observed that l is small, and proposed two novel parameterized solutions to find the provably approximate / optimal *GST-1*. Both solutions can be extended to find approximate *GST-k*. We conducted extensive studies over large undirected/directed graphs, and confirmed that our algorithm can obtain the optimal *GST-1* with high efficiency, and achieve high quality (low performance ratio) and high efficiency for computing *GST-k*.

There are several research directions one may wish to follow as an extension of the work of this thesis.

- As is pointed out in Theorem 5.5, the property of the *GST-k* output by algorithm *DPBF-k* can be used to improve the framework for finding the optimal *GST-k* [29]. The currently best framework proposed in [29] requires recomputation to find the top-$(k+1)$ answer after finding the top-k

answer, which is inefficiently in practice. How to improve the theoretical time complexity and efficiency of the top-k algorithm is an interesting work.

- How to reduce the time complexity of the exact algorithm for *GST-1* is a challenging problem. Since *GST-1* is at least as hard as the Set Cover problem, the improvement on *GST-1* algorithm also implies an improvement on the Set Cover algorithm. Both problems are NP-Complete, but the exact/parameterized algorithms for them are interesting especially when some parameters of the problem in practice are bounded. Up to now, only the exact algorithm for the min-cost Steiner tree problem (a special case of *GST-1* when only one node contains the each keyword) is known to be improved to $O((3-\epsilon)^l n)$ in undirected graphs.

- Currently, our algorithms *DPH-1* and *DPBF-1/k* can only support additive cost functions. When the cost function is not additive, the keyword query cannot be modelled as a group Steiner tree problem any more. So it is interesting to study the relationship between the type of the cost function and the hardness of the problem. A solution to a larger class of cost functions is of great interest.

- Keyword query problem is studied in databases in this thesis. It is interesting to study the model and solution for the keyword query problem on the Web.

□ **End of chapter.**

Bibliography

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. of ICDE'02*, 2002.

[2] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the db/ir panel at SIGMOD 2005. *SIGMOD Record*, 34(4), 2005.

[3] R. Baeza-Yates and M. Consens. The continued saga of DB-IR integration. In *Proc. of VLDB'04 (tutorial)*, 2004.

[4] A. Balmin, V. Hristidis, and Y. Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *Proc. of VLDB'04*, 2004.

[5] Y. Bartal. On approximating arbitrary metrices by tree metrics. In *STOC*, pages 161–168, 1998.

[6] C. D. Bateman, C. S. Helvig, G. Robins, and A. Zelikovsky. Provably good routing tree construction with multi-port terminals. In *Proc. of ISPD'97*, 1997.

[7] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proc. of ICDE'02*, 2002.

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

[9] M. Charikar, C. Chekuri, T.-Y. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. In *SODA*, pages 192–200, 1998.

[10] M. Charikar, C. Chekuri, A. Goel, and S. Guha. Rounding via trees: Deterministic approximation algorithms for group steiner trees and k-median. In *Proc. of STOC'98*, 1998.

[11] S. Chaudhuri, R. Ramakrishnan, and G. Weikum. Integrating DB and IR techniques: What is the sound of one hand clapping? In *Proc. of CIDR'05*, 2005.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivesi, and Clifford. *Introduction to Algorithm (2nd Edition)*. The MIT Press, USA, 2001.

[13] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding top-k min-cost connected trees in databases. In *ICDE*, pages 836–845, 2007.

[14] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[15] S. Dreyfus and R. Wagner. The steiner problem in graphs. *Networks*, 1(1), 1972.

[16] G. Even and G. Kortsarz. An approximation algorithm for the group steiner problem. In *SODA*, pages 49–58, 2002.

[17] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *STOC*, pages 448–455, 2003.

[18] U. Feige. A threshold of $\ln n$ for approximating set cover. *Journal of ACM*, 45(4), 1998.

[19] N. Garg, G. Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group steiner tree problem. In *Proc. of SODA'98*, 1998.

[20] E. Halperin, G. Kortsarz, R. Krauthgamer, A. Srinivasan, and N. Wang. Integrality ratio for group steiner trees and directed steiner trees. In *SODA*, pages 275–284, 2003.

[21] E. Halperin and R. Krauthgamer. Polylogarithmic inapproximability. In *STOC*, pages 585–594, 2003.

[22] C. S. Helvig, G. Robins, and A. Zelikovsky. Improved approximation bounds for the group steiner problem. In *DATE*, pages 406–413, 1998.

[23] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *Proc. of VLDB'03*, 2003.

[24] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *Proc. of VLDB'02*, 2002.

[25] E. Ihler. Bounds on the quality of approximate solutions to the group steiner problem. In *Proc. of WG'90*, 1990.

[26] E. Ihler. The complexity of approximating the class steiner tree problem. In *WG*, pages 85–96, 1991.

[27] L. Jia, G. Lin, G. Noubir, R. Rajaraman, and R. Sundaram. Universal approximations for tsp, steiner tree, and set cover. In *STOC*, pages 386–395, 2005.

[28] B. Kimelfeld and Y. Sagiv. New algorithms for computing steiner trees for a fixed number of terminals. *http://www.cs.huji.ac.il/ bennyk/papers/steiner06.pdf*.

[29] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proc. of PODS'06*, pages 173–182, 2006.

[30] M. Ley. DBLP: Computer Science Bibliography. `http://dblp.uni-trier.de/xml/`.

[31] W.-S. Li, K. S. Candan, Q. Vu, and D. Agrawal. Query relaxation by structure and semantics for retrieval of logical web documents. *IEEE Trans. Knowl. Data Eng.*, 14(4), 2002.

[32] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, pages 115–126, 2007.

[33] G. Reich and P. Widmayer. Beyond steiner's problem: A vlsi oriented generalization. In *Proc. of WG'89*, 1989.

[34] J. Riedl and J. Konstan. MoveLens. `http://www.cs.umn.edu/Research/GroupLens`.

[35] A. Srinivasan. New approaches to covering and packing problems. In *SODA*, pages 567–576, 2001.

[36] K. Varun, P. Shashank, C. Soumen, S. Sudarshan, D. Rushi, and K. Hrishikesh. Bidirectional expansion for keyword search on graph databases. In *Proc. of VLDB'05*, 2005.

[37] S. Wang, Z. Peng, J. Zhang, L. Qin, S. Wang, J. X. Yu, and B. Ding. Nuits: A novel user interface for efficient keyword search over databases. In *VLDB*, pages 1143–1146, 2006.