Efficient and Progressive Group Steiner Tree Search

Rong-Hua Li
Shenzhen University
Shenzhen, China
rhli@szu.edu.cn

Lu Qin
University of Technology
Sydney, Australia
Lu.Qin@uts.edu.au

Jeffrey Xu Yu
CUHK
Hong Kong, China
yu@se.cuhk.edu.hk

Rui Mao
Shenzhen University
Shenzhen, China
mao@szu.edu.cn

ABSTRACT

The Group Steiner Tree (GST) problem is a fundamental problem in database area that has been successfully applied to keyword search in relational databases and team search in social networks. The state-of-the-art algorithm for the GST problem is a parameterized dynamic programming (DP) algorithm, which finds the optimal tree in $O(3^k n + 2^k(n \log n + m))$ time, where k is the number of given groups, m and n are the number of the edges and nodes of the graph respectively. The major limitations of the parameterized DP algorithm are twofold: (i) it is intractable even for very small values of k (e.g., $k = 8$) in large graphs due to its exponential complexity, and (ii) it cannot generate a solution until the algorithm has completed its entire execution. To overcome these limitations, we propose an efficient and progressive GST algorithm in this paper, called PrunedDP. It is based on newly-developed optimal-tree decomposition and conditional tree merging techniques. The proposed algorithm not only drastically reduces the search space of the parameterized DP algorithm, but it also produces progressively-refined feasible solutions during algorithm execution. To further speed up the PrunedDP algorithm, we propose a progressive A^*-search algorithm, based on several carefully-designed lower-bounding techniques. We conduct extensive experiments to evaluate our algorithms on several large scale real-world graphs. The results show that our best algorithm is not only able to generate progressively-refined feasible solutions, but it also finds the optimal solution with at least two orders of magnitude acceleration over the state-of-the-art algorithm, using much less memory.

CCS Concepts: Information systems→Graph-based database models

Keywords: Group Steiner Tree; DP; A^*-search Algorithm

1. INTRODUCTION

The Group Steiner Tree (GST) problem is a well-known combinatorial optimization problem that has been extensively studied in both research and industry communities [28, 18, 8, 22]. Given a weighted graph G and a set of labels P, where each node in G is associated with a set of labels. The GST problem seeks to find the minimum-weight connected tree (i.e., the top-1 connected tree with smallest weight) from G that covers all the given labels in P [28].

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2882903.2915217

The GST problem has a number of applications in the database and data mining communities. For example, the keyword search problem in relational databases has been formulated as a GST problem [8, 6]. Specifically, a relational database can be modeled as a graph, where each node denotes a tuple and each edge represents a foreign key reference between two tuples. Each edge is associated with a weight, representing the strength of the relationship between two tuples (nodes). The keyword search problem aims to find a set of connected nodes (tuples) that covers all the given keywords with minimum total weight of all induced edges. Clearly, the optimal solution for such a keyword search problem is a minimum-weight connected tree that covers the given keywords, and therefore it is an instance of the GST problem. Most existing keyword search systems such as BANKS-I [3], BANKS-II [19], BLINKS [17], S-TAR [20] and DPBF [8, 6] are based on approximate or exact GST search techniques. As indicated in [6], the GST based method (e.g., DPBF) is the most effective algorithm for keyword search in relational databases. Another notable application of the GST problem is to find a team of experts from a social network [22], also called the team formulation problem. Specifically, there is a set of experts that form a social network, and each expert is associated with a set of skills. Given a set of skills, the team formulation problem is to find a team of experts with all the skills at the minimum communication cost. In [22], the team is modeled as a connected tree covering all the given skills, and the goal is to find such a connected tree with minimum weight, where the weight measures the communication cost. Obviously, such a team formulation problem is an instance of the GST problem. Recently, the GST based method has been recognized as a standard technique to find a team of experts in social networks [30]. Many existing team search systems such as [22], [2] and [25] are based on the GST search technique.

Given its large number of applications, devising an efficient algorithm to solve the GST problem is crucial. Unfortunately, the GST problem is known to be NP-hard, and it even cannot admit a polynomial approximate algorithm with constant approximation ratio [18]. Thus, there is no hope to find the optimal GST (or constant approximate GST) from a graph within polynomial time unless P=NP.

Previous studies have adopted two main types of approach to solving the GST problem. First, many approximation solutions [5, 13, 3, 19, 17] have been proposed to solve the GST problem. In the theoretical computer science community, several LP (linear programming) based approximation algorithms [5, 13] have been developed, where the approximation ratio of these algorithms is polylogarithmic. Since these algorithms need to invoke the LP procedure, they are very hard to handle medium-sized graphs. In the database community, many practical approximation algorithms such as BANKS-I [3], BANKS-II [19], and BLINKS [17] have been devised for the keyword search application. The approximation ratio of these algorithms is $O(k)$, where k is the number of given labels. However, as shown in [6], all these approxima-

tion algorithms are still inefficient for large graphs due to the high time and space complexity. Second, faster parameterized dynamic programming (DP) algorithm has been devised [8] which takes $O(3^k n + 2^k(n \log n + m))$ time and $O(2^k n)$ space, where m and n are the number of the edges and nodes of the graph respectively. Unlike the approximate algorithms, the parameterized DP algorithm is able to find the optimal solution in reasonable time when k is very small. As indicated in [6], the parameterized DP algorithm significantly outperforms the above mentioned approximation algorithms for the keyword search application. The major limitations of the parameterized DP algorithm are twofold. First, due to the exponential time and space complexity, the parameterized DP algorithm quickly becomes impractical even for very small k (e.g., $k = 8$) in large graphs. Second, it cannot generate a solution until the algorithm has completed its entire execution, thus making the algorithm costly in practice.

Against this background, we propose an efficient and progressive GST algorithm, called PrunedDP. The PrunedDP algorithm quickly returns an initial solution, and then improves the quality of the solution that is found so far and achieves progressively better bounds, as the algorithm searches more of the search space, until an optimal solution is generated. The strength of the PrunedDP algorithm is founded on two main components. First, we propose a new approach to constructing a feasible solution for each state of the parameterized DP algorithm, and then keep refining those solutions during algorithm execution. Second, we develop two novel techniques, called optimal-tree decomposition and conditional tree merging, to reduce the search space of the parameterized DP algorithm. The general idea of these techniques is to obtain the optimal tree by merging several optimal subtrees with weights smaller than one half of the optimal solution. Moreover, we discover that there is no need to expand a state in the DP procedure by merging two optimal subtrees if their total weight is larger than two over three of the optimal solution. Armed with these techniques, the PrunedDP algorithm not only produces progressively-refined feasible solutions during algorithm execution, but it also drastically prunes the search space of the parameterized DP algorithm. Thus, the time and space overhead of the algorithm is significantly reduced.

To further speed up the PrunedDP algorithm, we propose a progressive A^*-search algorithm, called PrunedDP++, based on several carefully-designed lower-bounding techniques (more details can be found in Section 4). The PrunedDP++ algorithm is built on the PrunedDP algorithm. But unlike the PrunedDP algorithm, PrunedDP++ makes use of the A^*-search strategy to select the most promising state to expand, and thus many unpromising states can be pruned. We conduct extensive experiments on several large scale real-world graphs to evaluate the proposed algorithms. The results show that our algorithms are able to find progressively-refined feasible solutions during algorithm execution. Moreover, the reported feasible solutions can be quickly refined to a near-optimal solution in less than 10 seconds in large-scale graphs (≥ 10 million nodes) even for $k = 8$. In addition, when $k = 8$, our best algorithm finds the optimal solution in around 20 seconds in a large-scale graph, while the state-of-the-art algorithm cannot get the optimal solution within one hour. In general, the PrunedDP algorithm is around one order of magnitude faster than the state-of-the-art algorithm, and the PrunedDP++ algorithm can further achieves at least one order of magnitude acceleration over the PrunedDP algorithm, using much less memory.

The main contributions of this paper are summarized as follows.

- We propose an efficient and progressive GST algorithm, called PrunedDP, based on newly-developed optimal-tree decomposition and conditional tree merging techniques. The striking features of the PrunedDP algorithm are twofold: (i) the algorithm can generate progressively-refined feasible solutions during algorithm execution, and (ii) it finds the optimal solution one order of magnitude faster than the state-of-the-art algorithm.
- To further reduce the search space of the PrunedDP algorithm, we propose a progressive A^*-search algorithm, called PrunedDP++, based on several carefully-devised lower-bounding techniques. The PrunedDP++ algorithm also reports progressively-refined feasible solutions, and is at least one order of magnitude faster than the PrunedDP algorithm, using much less memory.
- We conduct comprehensive experiments on several large scale real-world graphs (≥ 10 million nodes) to evaluate our algorithms, and the results confirm our theoretical findings.

Notation	Meaning
$G = (V, E)$	the graph
S	the set of labels
S_v	the set of labels associated with node v
P	the given label sets
$f^*(P)$	the optimal solution for the given label sets P
(v, X)	a state of the DP algorithm
$T(v, X)$	the minimum-weight tree rooted at v covering labels X
$f^*_T(v, X)$	the weight of $T(v, X)$
$\tilde{v}_p$	the virtual node corresponding to label p
V_P	the set of virtual nodes corresponding to labels P
$R(\tilde{v}_i, \tilde{v}_j, \bar{X})$	the minimum-weight route that starts from $\tilde{v}_i$, ends at $\tilde{v}_j$, and passes through all virtual nodes in $V_{\bar{X}}$
$W(\tilde{v}_i, \tilde{v}_j, \bar{X})$	the weight of the route $R(\tilde{v}_i, \tilde{v}_j, \bar{X})$
$R(\tilde{v}_i, \bar{X})$	the minimum-weight route that starts from $\tilde{v}_i$ and passes through all virtual nodes in $V_{\bar{X}}$
$W(\tilde{v}_i, \bar{X})$	the weight of the tour $R(\tilde{v}_i, \bar{X})$
$R(v, \bar{X})$	a tour starts from v and passes through all nodes in $V_{\bar{X}}$
$\tilde{R}(v, \bar{X})$	the minimum-weight tour over all $R(v, \bar{X})$
$f^*_{\tilde{R}}(v, \bar{X})$	the weight of the tour $\tilde{R}(v, \bar{X})$
$\pi_1(v, X)$	the one-label lower bound for a state (v, X)
$\pi_{t_1}(v, X)$	the first type of tour-based lower bound for a state (v, X)
$\pi_{t_2}(v, X)$	the second type of tour-based lower bound for a state (v, X)

Table 1: Summary of notations

The rest of this paper is organized as follows. In Section 2, we formulate the problem and briefly review the parameterized DP algorithm. In Section 3, we propose the PrunedDP algorithm and develop the optimal-tree decomposition and conditional tree merging techniques. The PrunedDP++ algorithm and several nontrivial lower bounding techniques are proposed in Section 4. Finally, we review related work and conclude this paper in Section 6 and Section 7 respectively.

2. PRELIMINARIES

Let $G = (V, E)$ be a weighted graph, where V and E denote the set of nodes and edges respectively. Let $n = |V|$ and $m = |E|$ be the number of nodes and edges respectively. For each edge $e = (v_i, v_j) \in E$, $w(v_i, v_j)$ denotes the weight of e. For each node $v \in V$, there is a set of labels, denoted by S_v, associated with v. Let $S = \bigcup_{v \in V} S_v$ be the set of all labels. For each label $a \in S$, we let $V_a \subseteq V$ be the group (set) of nodes where each node in V_a contains the label a. Note that by this definition, each label corresponds to a group. Based on these notations, the Group Steiner Tree (GST) problem [28] is formulated as follows.

The GST problem. Given a weighted and labelled graph $G = (V, E)$ and a subset of labels P ($P \subseteq S$), the GST problem seeks to find the minimum-weight connected tree (i.e., the top-1 connected tree with smallest weight) from G that includes all the labels in P.

For convenience, in the rest of this paper, we assume without loss of generality that the graph $G = (V, E)$ is a connected graph. This is because if the graph is disconnected, we can solve the GST problem in each maximal connected component of the graph, and then pick the best connected tree as the answer.

It is well known that the GST problem is NP-hard [28], and is even inapproximable within a constant approximation factor by a polynomial algorithm [18]. Therefore, there is no hope to compute the GST (or find an approximate GST within constant preformation ratio) in polynomial time unless P=NP. In the literature, the state-of-the-art exact algorithm for the GST problem is a parameterized dynamic programming (DP) algorithm which runs in $O(3^k n + 2^k(n \log n + m))$ time [8], where $k = |P|$ denotes the number of given labels. Below, we briefly review such a parameterized DP algorithm, as it forms the basis for devising our progressive GST algorithms.

The parameterized DP algorithm. In the parameterized DP algorithm, a state, denoted by (v, X), corresponds to a connected tree rooted at v that covers all labels in X. Let $T(v, X)$ be the minimum-weight connected tree rooted at v that includes all labels in X, and $f_T^*(v, X)$ be the weight of $T(v, X)$ (i.e., the total weight of the edges in $T(v, X)$). Clearly, by these notations, the optimal weight of a state (v, X) is $f_T^*(v, X)$. Table 1 summarizes the important notations used in this paper.

Based on the definition of the state, the state-transition equation of the parameterized DP algorithm proposed in [8] is given by

$$f_T^*(v, X) = \min\{\min_{(v,u)\in E}\{f_T^*(u, X) + w(v, u)\}, \min_{\substack{(X=X_1\cup X_2)\\ \wedge(X_1\cap X_2=\emptyset)}}\{f_T^*(v, X_1) + f_T^*(v, X_2)\}\}. \quad (1)$$

As shown in Eq. (1), the optimal weight of the state (v, X) could be obtained by either of the following two cases. The first case is that the optimal connected tree, denoted by $T(v, X)$, could be obtained by growing an edge (v, u) from the minimum-weight subtree of $T(v, X)$ rooted at u including all labels in X, where u is a child node of v (i.e., the first part of Eq. (1)). We refer to the state expansion operation in this case as the *edge growing* operation. The second case is that $T(v, X)$ could be obtained by merging two optimal subtrees rooted at v that includes all labels in X_1 and X_2 respectively, where $X_1 \cup X_2 = X$ and $X_1 \cap X_2 = \emptyset$ (i.e., the second part of Eq. (1)). Similarly, we refer to the state expansion operation in this case as the *tree merging* operation.

Initially, for each node v with labels S_v, we have $f_T^*(v, \{p\}) = 0$, for each label $p \in S_v$. By Eq. (1), Ding et al. [8] proposed a parameterized DP algorithm based on the best-first search strategy to compute the minimum $f_T^*(v, P)$ over all $v \in V$. Specifically, the parameterized DP algorithm uses a priority queue $\mathcal{Q}$ to maintain the states (e.g., (v, X)), in which the weight of a state is the priority of that state. Initially, the state $(v, \{p\})$ for each $p \in S_v$ is inserted into $\mathcal{Q}$. Then, the algorithm iteratively pops the smallest-weight state from $\mathcal{Q}$, and applies Eq. (1) to expand such a state. For each expanded state, if it is already in $\mathcal{Q}$, the algorithm updates its weight using the smaller weight. Otherwise, the algorithm inserts it into $\mathcal{Q}$. The algorithm terminates until the popped state covers all the labels in P. Clearly, by the best-first search strategy, the optimal state is always popped later than any intermediate state from the priority queue, and thus the weight of the popped intermediate state must be smaller than the optimal solution.

As shown in [8], the parameterized DP algorithm works well for keyword search applications when both the parameter k and the graph size are small. However, as discussed below, the parameterized DP algorithm becomes impractical when the parameter k and the graph size are large.

Limitations of the parameterized DP algorithm. First, the time complexity of the parameterized DP algorithm relies on the exponential factor 3^k. Thus, the algorithm only works well for very small values of k (e.g., $k = 4$ used in [8]) in large graphs. On the other hand, the space complexity of the parameterized DP algorithm is $O(2^k n)$. Clearly, when k is large, the algorithm will quickly run out of memory in large graphs. Indeed, as shown in our experiments, the parameterized DP algorithm does not work in large graphs with relatively large k values (e.g., $k = 8$). Second, the algorithm only generates a solution (i.e., the optimal solution) when it terminates. However, in many practical applications (e.g., keyword search), users may prefer a sub-optimal solution in less time, rather than wait for the algorithm to find the optimal solution.

The above discussions motivate us to answer the following question: can we develop an algorithm that is not only more efficient than the parameterized DP algorithm, but also reports sub-optimal solutions during execution (not only when the algorithm terminates)? In the following sections, we will propose several novel algorithms to achieve those goals.

3. THE PROGRESSIVE GST ALGORITHM

In this section, we first propose a basic progressive GST algorithm, called Basic, based on the parameterized DP algorithm in [8]. Then, we propose a novel algorithm, called PrunedDP, to reduce the search space of the Basic algorithm based on newly-developed optimal-tree decomposition and conditional tree merging techniques.

3.1 The Basic algorithm

We first introduce a progressive search framework, and then present the Basic algorithm following this framework.

The progressive search framework. Roughly speaking, a progressive search algorithm works in rounds, reporting a sub-optimal and feasible solution with smaller error guarantees in each round, until in the last round the optimal solution is obtained. More specifically, we refer to a search algorithm as a progressive algorithm when it satisfies the following two properties.

- **Progressive property.** The algorithm works in rounds. In each round, the algorithm returns a feasible solution, and reports the maximum error (with respect to the optimal solution) it may have.
- **Monotonic property.** The maximum error of a feasible solution does not increase after each round. In other words, the feasible solution reported in each round should be a refinement of the previous solution. Until in the last round, the optimal solution is obtained.

Clearly, the progressive property is desirable for many applications, because with the progressive property, we can always obtain a feasible solution and also know its bound to the optimal solution whenever we interrupt the algorithm. The monotonic property is also desirable, because the maximum error in each round is guaranteed to be non-increasing, and thus the algorithm can obtain a refined solution from one round to the next.

The basic progressive GST algorithm. Following the progressive search framework, we devise a progressive GST algorithm, called Basic, based on the parameterized DP algorithm in [8]. The general idea of Basic is that we first construct a feasible solution for an intermediate state (v, X) of the parameterized DP algorithm, then keep refining the feasible solution, and also compute the bound for that feasible solution.

Before proceeding further, we first introduce a preprocessing procedure for any query P and graph G, which will be used in all the proposed algorithms. Specifically, for a label $p \in P$, we create a virtual node $\tilde{v}_p$, and create an undirected edge $(\tilde{v}_p, v)$ with zero weight for each $v \in V$ that includes a label p. Then, we compute the single-source shortest path from $\tilde{v}_p$ to all the other nodes in the graph G. For each label $p \in P$, we independently perform this process. Let $dist(v, \tilde{v}_p)$ be the shortest-path distance between v and the virtual node $\tilde{v}_p$. For a query P, we can pre-compute all those shortest paths and $dist(v, \tilde{v}_p)$ by invoking the Dijkstra algorithm $k = |P|$ times, which takes $O(k(m + n \log n))$ time complexity.

After the above preprocessing, we can then construct a feasible solution as follows. First, for each state (v, X), let $\bar{X} = P \backslash X$

Algorithm 1 Basic(G, P, S)

Input: $G = (V, E)$, label set S, and the query label set P.
Output: the minimum weight of the tree that covers P.

1: $\mathcal{Q} \leftarrow \emptyset$; $\mathcal{D} \leftarrow \emptyset$;
2: $best \leftarrow +\infty$; /* maintain the weight of the best feasible solution. */
3: **for all** $v \in V$ **do**
4: **for all** $p \in S_v$ **do**
5: $\mathcal{Q}.push((v, \{p\}), 0)$;
6: **while** $\mathcal{Q} \neq \emptyset$ **do**
7: $((v, X), cost) \leftarrow \mathcal{Q}.pop()$;
8: **if** $X = P$ **then return** $cost$;
9: $\mathcal{D} \leftarrow \mathcal{D} \cup \{(v, X)\}$; $\bar{X} \leftarrow P \backslash X$;
10: $T'(v, \bar{X}) \leftarrow \emptyset$;
11: **for all** $p \in \bar{X}$ **do**
12: $T'(v, \bar{X}) \leftarrow T'(v, \bar{X}) \cup$ Shortest-Path$(v, \tilde{v}_p)$;
13: $\tilde{T}(v, P) \leftarrow \mathsf{MST}(T'(v, \bar{X}) \cup T(v, X))$;
14: $best \leftarrow \min\{best, f_{\tilde{T}}(v, P)\}$;
15: Report the approximation ratio for $best$;
16: **for all** $(v, u) \in E$ **do**
17: update$(\mathcal{Q}, \mathcal{D}, best, (u, X), cost + w(v, u))$;
18: **for all** $X' \subseteq \bar{X}$ **and** $(v, X') \in \mathcal{D}$ **do**
19: update$(\mathcal{Q}, \mathcal{D}, best, (v, X' \cup X), cost + \mathcal{D}.cost(v, X'))$;
20: **return** $+\infty$;

21: **Procedure** update$(\mathcal{Q}, \mathcal{D}, (v, X), cost)$
22: **if** $(v, X) \in \mathcal{D}$ **then return**;
23: **if** $cost >= best$ **then return**;
24: **if** $X = P$ **then** $best \leftarrow \min\{best, cost\}$;
25: **if** $(v, X) \notin \mathcal{Q}$ **then** $\mathcal{Q}.push((v, X), cost)$;
26: **if** $cost < \mathcal{Q}.cost((v, X))$ **then** $\mathcal{Q}.update((v, X), cost)$;

(i.e., $\bar{X}$ be the complementary set of X with respect to the set P). For a node v and label set $\bar{X}$, there are $|\bar{X}|$ shortest paths from v to $\tilde{v}_p$ for all $p \in \bar{X}$. We merge all those $|\bar{X}|$ shortest paths, resulting in a tree denoted by $T'(v, \bar{X})$. Second, we unite the tree $T(v, X)$ (corresponding to the state (v, X)) and $T'(v, \bar{X})$, and then compute the minimum spanning tree (MST) of the united result, i.e., $\mathsf{MST}(T'(v, \bar{X}) \cup T(v, X)))$. Let $\tilde{T}(v, P) \triangleq \mathsf{MST}(T'(v, \bar{X}) \cup T(v, X)))$. Clearly, $\tilde{T}(v, P)$ is a feasible solution, as it is a tree that covers all the labels in P.

Let $f_{\tilde{T}}(v, P)$ be the weight of the tree $\tilde{T}(v, P)$. Then, $f_{\tilde{T}}(v, P)$ is an upper bound of the optimal solution. On the other hand, we can always use $f^*_T(v, X)$ as a lower bound for the optimal solution. This is because by the best-first DP algorithm, the optimal solution is popped later from the priority queue than any computed intermediate state (v, X), and thus we have $f^*_T(v, X) \leq f^*(P)$, where $f^*(P)$ is the optimal solution for the GST query P. As a result, for any intermediate state (v, X), we can always report a feasible solution $\tilde{T}(v, P)$ and its approximation ratio $f_{\tilde{T}}(v, P)/f^*_T(v, X)$.

The algorithm is detailed in Algorithm 1. In Algorithm 1, we use a tuple $((v, X), cost)$ to denote the state (v, X), where $cost$ denotes the weight of the tree that corresponds to the state (v, X). Similar to the parameterized DP algorithm [8], Algorithm 1 follows the best-first search strategy. In each round, the algorithm pops the best state (v, X) from the priority queue $\mathcal{Q}$ (line 7), and uses a set $\mathcal{D}$ to maintain all the states that have been computed (i.e., the optimal weight of each state in $\mathcal{D}$ has been calculated). Based on (v, X), we construct a feasible solution $\tilde{T}(v, P)$ as described above (lines 10-13).

To make the algorithm progressive, we use a variable $best$ to record the best feasible solution found so far, and then keep refining $best$ in each round, and also report the approximation ratio for $best$ using the method described above (lines 14-15). In lines 16-19, the algorithm expands and updates the states based on the state-transition equation of the parameterized DP algorithm (i.e., Eq.(1)). Note that unlike the parameterized DP algorithm, in line 23, the Basic algorithm makes use of the current best solution for pruning the unpromising states. This pruning technique not only decreases the running time of the parameterized DP algorithm, but also reduces the space overhead as well (because the unpromising states will not be inserted into the priority queue $\mathcal{Q}$). Consequently, Basic is not just a progressive algorithm, it can also improve the efficiency of the parameterized DP algorithm. However, although the Basic algorithm is more efficient than the parameterized DP algorithm, it still needs to search a large number of states to find the optimal solution. Below, we propose a novel algorithm which drastically reduces the search space of the Basic algorithm.

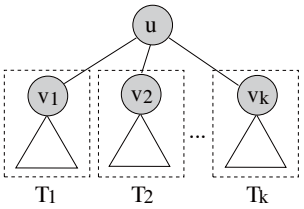


Figure 1: Illustration of the optimal-tree decomposition.

3.2 The PrunedDP algorithm

Recall that in the Basic algorithm, to obtain the optimal solution $f^*(P)$ for a query P, the algorithm must compute the optimal weights of all the intermediate states that are smaller than $f^*(P)$ in terms of the best-first search strategy. An immediate question is: can we avoid computing all such optimal weights to get $f^*(P)$? Below, we propose a new algorithm based on optimal-tree decomposition and conditional tree merging techniques to achieve this goal. Specifically, let $T^*(P)$ be the optimal tree with weight $f^*(P)$. For simplicity, we assume that $T^*(P)$ contains at least one edge. We then have the following optimal-tree decomposition theorem. Proofs of all the theorems and lemmas in this paper can be found in Appendix A.1.

THEOREM 1. ***Optimal-Tree Decomposition Theorem**: Assume that each edge in the graph G has a positive weight. Then, for the optimal tree $T^*(P)$, there always exists a node $u \in T^*(P)$ such that (i) the tree $T^*(P)$ rooted at u has k ($k \geq 1$) subtrees $T_1, T_2, \cdots, T_k$, and (ii) each subtree T_i (for $i \in \{1, 2, \cdots, k\}$) has a weight smaller than $f^*(P)/2$.*

Fig. 1 illustrates the idea of the optimal-tree decomposition, where every subtree rooted at v_i, for $i = 1, \cdots, k$, has a weight smaller than $f^*(P)/2$. According to Theorem 1, the optimal tree can always be decomposed into several optimal subtrees with weights smaller than $f^*(P)/2$. This result motivates us to devise a two-stage algorithm to compute the optimal tree. In the first stage, we compute all the optimal subtrees that have weights smaller than $f^*(P)/2$ by the best-first DP algorithm. Then, in the second stage, we merge the results obtained from the first stage to get the optimal tree. This is because, after the first stage, all the optimal subtrees with weights smaller than $f^*(P)/2$ have been generated, thus by Theorem 1, we can obtain the optimal tree via merging the optimal subtrees. To achieve this, we can apply the same method as used in the best-first DP algorithm to merge the optimal subtrees. Specifically, we can obtain the optimal tree by invoking two operations: (i) *edge growing*, and (ii) *tree merging*. For example, in Fig. 1, the optimal tree rooted at u can be obtained by growing an edge (v_i, u) from each optimal subtree rooted at v_i, and then merging all those edge-grown subtrees. Thus, without loss of optimality, for each optimal subtree generated in the first stage, we must perform an edge growing operation. Then, for every two subtrees, if they can be merged (i.e., share the same root), we need to perform a tree merging operation. Clearly, the optimal tree can always be obtained by invoking these two operations.

Since the second stage of our algorithm invokes the same operations that are used in the best-first DP algorithm, we can integrate

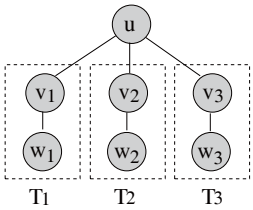


Figure 2: Illustration of the optimality of Theorem 2.

them into the first stage. In particular, when we pop a state from the priority queue in the best-first DP algorithm, we expand this state only when its weight is smaller than $f^*(P)/2$. Clearly, by adding this constraint to expand the states, all the optimal subtrees with weights smaller than $f^*(P)/2$ can be computed. Moreover, every subtree, which is obtained by invoking an edge growing operation on a computed subtree, is generated; and every subtree that is obtained by invoking a tree merging operation on two computed subtrees is also generated. After getting all these subtrees, we merge two subtrees if they can form a feasible solution (i.e., covers all labels in P), and then select the best one as the optimal tree.

Interestingly, we discover that there is no need to generate every subtree obtained by invoking the tree merging operation on two computed subtrees. Without loss of optimality, we can add an additional constraint in the tree merging operation to avoid generating unpromising subtrees in the best-first DP algorithm. Our technique is based on the following conditional tree merging theorem.

THEOREM 2. ***Conditional Tree Merging Theorem**: Without loss of optimality, to expand a state (v, X) by a tree merging operation in the best-first DP algorithm, we can merge two subtrees $T(v, X)$ and $T(v, X')$ for $X' \subset P \backslash X$ only when the total weight of these two subtrees is no larger than $2/3 \times f^*(P)$.*

By Theorem 2, we can further reduce a number of states generated in the best-first DP algorithm without loss of optimality, because any intermediate state (not a feasible solution) with a weight larger than $2f^*(P)/3$ does not need to be generated by the tree merging operation. It is important to note that this does not mean that all the states with weights larger than $2f^*(P)/3$ will not be generated by our algorithm, because a state with weight larger than $2f^*(P)/3$ may be generated by the edge growing operation, and also it may be generated by merging two *complementary* states (i.e., (v, X) and $(v, \bar{X})$ with $X = P \backslash X$) to form a feasible solution.

The optimality of Theorem 2. It should be noted that there is a factor of 2/3 in Theorem 2. Clearly, a small factor is better than a large one when pruning unpromising states. Thus, without loss of optimality, we strive to find the minimum factor to reduce the number of states generated by the best-first DP algorithm. However, interestingly, we find that the factor 2/3 is optimal. Below, we give an example in which the factor cannot be smaller than 2/3. Consider an optimal tree rooted at u as shown in Fig. 2. The optimal tree has three subtrees which are denoted by T_i for $i = \{1, 2, 3\}$ with roots v_i respectively. Suppose that the weight of each edge in this tree is 1. Note that by our algorithm, the optimal solution in this example must be obtained by merging two subtrees in which one of them must have a weight no smaller than $2f^*(P)/3$. Moreover, in this example, by our algorithm, we must invoke the tree merging operation to obtain a subtree that has a weight no smaller than $2f^*(P)/3$. This is because, we cannot obtain such a subtree by growing an edge from an optimal subtree with a weight smaller than $f^*(P)/2$. Therefore, in this example, we cannot set the factor to be a value smaller than $2/3$, otherwise we cannot generate such a subtree and thus cannot guarantee the correctness of the algorithm. As a result, the factor 2/3 given in Theorem 2 is optimal.

Based on the above theoretical results, we are now ready to present the PrunedDP algorithm. Note that to make the algorithm progressive, we can apply the same approach developed in the Basic algorithm to produce progressively-refined results. In addition, a final issue is that Theorem 1 and Theorem 2 rely on the optimal solution $f^*(P)$ of the GST problem which is unknown before the algorithm terminates. However, it is easy to verify that if we use an upper bound of $f^*(P)$ to replace $f^*(P)$ in Theorem 1 and Theorem 2, the results still hold. Therefore, in our algorithm, we can use the weight of the best feasible solution found so far as an upper bound of $f^*(P)$. Since the best feasible solution is progressively improved, the bound becomes tighter and tighter, as the algorithm searches more of the search space, and thus the pruning power of the algorithm is progressively enhanced.

Algorithm 2 PrunedDP(G, P, S)

Input: $G = (V, E)$, label set S, and the query label set P.
Output: the minimum weight of the tree that covers P.

```
1: Q ← ∅; D ← ∅;
2: best ← +∞; /* maintain the weight of the best feasible solution. */
3: for all v ∈ V do
4:     for all p ∈ S_v do
5:         Q.push((v, {p}), 0);
6: while Q ≠ ∅ do
7:     ((v, X), cost) ← Q.pop();
8:     if X = P then return cost;
9:     D ← D ∪ {(v, X)}; X̄ ← P\X;
10:    T'(v, X̄) ← ∅;
11:    for all p ∈ X̄ do
12:        T'(v, X̄) ← T'(v, X̄) ∪ Shortest-Path(v, ṽ_p);
13:    T̃(v, P) ← MST(T'(v, X̄) ∪ T(v, X));
14:    best ← min{best, f_T̃(v, P)};
15:    Report the approximation ratio for best;
16:    if (v, X̄) ∈ D then
17:        update(Q, D, best, (v, P), cost + D.cost(v, X̄));
18:        continue;
19:    if cost < best/2 then
20:        for all (v, u) ∈ E do
21:            update(Q, D, best, (u, X), cost + w(v, u));
22:        for all X' ⊂ X̄ and (v, X') ∈ D do
23:            cost' ← D.cost(v, X');
24:            if cost + cost' ≤ 2/3 × best then
25:                update(Q, D, best, (v, X' ∪ X), cost + cost');
26: return +∞;
```

The algorithm is detailed in Algorithm 2. Note that PrunedDP follows the same framework as the Basic algorithm to produce progressively-refined feasible solutions (see lines 10-15 of Algorithm 2). But drastically different from Basic, PrunedDP is based on the pruned DP-search strategy developed in Theorem 1 and Theorem 2. Specifically, in lines 16-18, the algorithm merges two *complementary* computed states once they can form a feasible state (a state that covers all query labels P), and then invokes the same update procedure as used in Algorithm 1 to update the feasible state. Then, in lines 19-25, the algorithm expands the state (v, X) only if its weight is smaller than $best/2$. This is because, according to the results in Theorem 1, the algorithm does not need to expand a state with weight no smaller than $f^*(P)/2$. In addition, in line 24, the algorithm merges two optimal subtrees only if their total weight is smaller than $2/3 \times best$ based on the *conditional tree merging* theorem. Clearly, the correctness of the algorithm can be guaranteed by Theorem 1 and Theorem 2.

Cost analysis of the PrunedDP algorithm. It is worth noting that the worst-case time and space complexity of the PrunedDP algorithm is the same as that of the Basic algorithm. However, compared to Basic, PrunedDP can prune a large number of unpromising states using *optimal-tree decomposition* and *conditional tree merging* techniques, and thus drastically reduces both the running time and memory overhead. In the experiments, we show that PrunedDP is one order of magnitude faster than the Basic algorithm, using much less memory.

4. THE PROGRESSIVE A^* ALGORITHM

Recall that the PrunedDP algorithm finds the optimal solution in a pruned search space based on the best-first DP search strategy. To further speed up the PrunedDP algorithm, we propose a novel progressive algorithm, called PrunedDP++, based on the A^*-search strategy over the pruned search space. Below, we first propose several lower bounding techniques, which are crucial to devise the PrunedDP++ algorithm.

4.1 Lower bounds construction

To devise an A^*-search algorithm for the GST problem, the key is to establish an effective lower bound for each state (v, X) in the search space [16, 7]. It is important to note that according to the A^*-search theory [16, 7], the lower bound in our problem denotes the bound from the current state (v, X) to its goal state (v, P). Thus, for a state (v, X), we need to design a lower bound for the weight of the optimal subtree $T(v, \bar{X})$ (i.e., $f_T^*(v, \bar{X})$), where $\bar{X} = P \backslash X$. In the following, we propose several effective lower bounds that are constructed via relaxing the constraints of the optimal subtree $T(v, \bar{X})$.

One-label lower bound. Note that the optimal subtree $T(v, \bar{X})$ must cover all the labels in $\bar{X}$. Here we develop a lower bound, called one-label lower bound, by relaxing such a *label covering* constraint. Our idea is that we consider the optimal connected tree rooted at v that covers only one label in $\bar{X}$, where $\bar{X} = P \backslash X$. Specifically, let $T(v, \{x\})$ be the optimal connected tree rooted at v that covers a label $x \in \bar{X}$, and $f_T^*(v, \{x\})$ be the weight of $T(v, \{x\})$. Further, we let $f_1^*(v, \bar{X})$ be the weight of the maximum-weight tree over all $T(v, \{x\})$ for $x \in \bar{X}$. Then, we define the one-label lower bound $\pi_1(v, X)$ as follows.

$$\pi_1(v, X) \triangleq f_1^*(v, \bar{X}) = \max_{x \in \bar{X}} \{f_T^*(v, \{x\})\}. \quad (2)$$

The following lemma shows that $\pi_1(v, X)$ is indeed a valid lower bound for a state (v, X).

LEMMA 1. *For any state (v, X), we have $\pi_1(v, X) \leq f_T^*(v, \bar{X})$.*

Note that by definition, $f_T^*(v, \{x\})$ is equal to the shortest-path distance between the node v and the virtual node $\tilde{v}_x$ (i.e., $f_T^*(v, \{x\}) = dist(v, \tilde{v}_x)$), which can be obtained by invoking the same preprocessing procedure used in Algorithms 1 and 2. Therefore, for any state (v, X), the time complexity for computing $\pi_1(v, X)$ is $O(|\bar{X}|)$ after preprocessing.

Tour-based lower bounds. Recall that the optimal tree $T(v, \bar{X})$ must be a connected tree. By relaxing such a *connected tree* constraint, here we develop two novel tour-based lower bounds.

The first type of the tour-based lower bound is constructed as follows. First, for a given set of labels P with $k = |P|$, we create a virtual node $\tilde{v}_p$ for each label $p \in P$. Let V_P denotes the set of all virtual nodes. Then, we create an edge $(\tilde{v}_p, v)$ with zero weight for each $v \in V$ that includes a label p. After that, we can obtain a new graph, called the label-enhanced graph. Second, we compute the single-source shortest path from each virtual node $\tilde{v}_p$ to all the other nodes in the label-enhanced graph. It is worth mentioning that this procedure is different from the preprocessing procedure used for constructing the feasible solution in Algorithm 1 and Algorithm 2, and also for computing the one-label lower bound. Here we add all virtual edges simultaneously for all the virtual nodes, whereas in the previous preprocessing procedure, we independently process each virtual node. Third, for $\bar{X} \subseteq P$, we let $V_{\bar{X}}$ be the set of all virtual nodes that correspond to the labels in $\bar{X}$. For any $i, j \in \bar{X}$ and $\bar{X} \subseteq P$, we let $R(\tilde{v}_i, \tilde{v}_j, \bar{X})$ be the minimum-weight route that starts from $\tilde{v}_i$, ends at $\tilde{v}_j$, and passes through all the virtual nodes in $V_{\bar{X}}$. Further, we let $W(\tilde{v}_i, \tilde{v}_j, \bar{X})$ be the weight of the route $R(\tilde{v}_i, \tilde{v}_j, \bar{X})$.

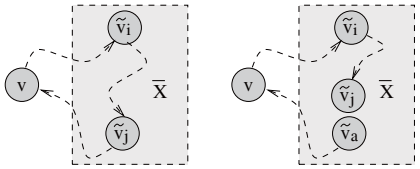


Figure 3: Illustration of the tour-based lower bounds.

Based on these notations, for a state (v, X) of the GST problem, we can construct a tour $R(v, \bar{X}) \triangleq (v \sim R(\tilde{v}_i, \tilde{v}_j, \bar{X}) \sim v)$ that starts from v, ends at v, and covers all labels in $\bar{X}$. Fig. 3(a) illustrates such a tour. Note that for each virtual-node pair $(\tilde{v}_i, \tilde{v}_j)$, we have a tour $R(v, \bar{X})$. To construct the lower bound, we intend to find the minimum-weight tour. Specifically, we define $\tilde{R}(v, \bar{X})$ as the minimum-weight tour over all tours $R(v, \bar{X})$ for all $\tilde{v}_i, \tilde{v}_j \in V_{\bar{X}}$. Further, we let $f_{\tilde{R}}^*(v, \bar{X})$ be the weight of the optimal tour $\tilde{R}(v, \bar{X})$. Formally, we have

$$f_{\tilde{R}}^*(v, \bar{X}) = \min_{i,j \in \bar{X}} \{dist(v, \tilde{v}_i) + W(\tilde{v}_i, \tilde{v}_j, \bar{X}) + dist(\tilde{v}_j, v)\}, \quad (3)$$

where $dist(v, \tilde{v}_i)$ denotes the shortest-path distance between v and $\tilde{v}_i$. For a state (v, X), we define the first type of tour-based lower bound using the following formula:

$$\pi_{t_1}(v, X) \triangleq f_{\tilde{R}}^*(v, \bar{X})/2. \quad (4)$$

With this definition, we show that $\pi_{t_1}(v, X)$ is a valid lower bound.

LEMMA 2. *For any state (v, X), we have $\pi_{t_1}(v, X) \leq f_T^*(v, \bar{X})$.*

With Lemma 2, we successfully construct a lower bound for any state (v, X) of the GST problem. The remaining question is how to compute such a lower bound efficiently. According to Eq. (3), the most challenging task is to compute $W(\tilde{v}_i, \tilde{v}_j, \bar{X})$ for all $i, j \in \bar{X}$. Below, we propose a DP algorithm to tackle this challenge.

For any $i, j \in \bar{X}$ and $\bar{X} \subseteq P$, we define $(\tilde{v}_i, \tilde{v}_j, \bar{X})$ as a state in our DP algorithm, denoting a route that starts from $\tilde{v}_i$, ends at $\tilde{v}_j$, and passes through all the virtual nodes in $V_{\bar{X}}$. Then, the recursive equation of the DP algorithm is given as follows.

$$W(\tilde{v}_i, \tilde{v}_j, \bar{X}) = \min_{p \in \bar{X} \backslash \{j\}} \{W(\tilde{v}_i, \tilde{v}_p, \bar{X} \backslash \{p\}) + dist(\tilde{v}_p, \tilde{v}_j)\}, \quad (5)$$

where $dist(\tilde{v}_p, \tilde{v}_j)$ denotes shortest path distance between two virtual nodes $\tilde{v}_p$ and $\tilde{v}_j$ in the label-enhanced graph. We explain Eq. (5) as follows. The optimal route w.r.t. the state $(\tilde{v}_i, \tilde{v}_j, \bar{X})$ can be obtained by expanding the optimal sub-route w.r.t. the state $(\tilde{v}_i, \tilde{v}_p, \bar{X} \backslash \{j\})$ with a path $\tilde{v}_p \sim \tilde{v}_j$. Initially, we have $W(\tilde{v}_i, \tilde{v}_i, \{\tilde{v}_i\}) = 0$, for all $\tilde{v}_i \in V_P$.

By Eq. (5), we can implement the DP algorithm using the best-first search strategy. Algorithm 3 details the DP algorithm. By invoking Algorithm 3, we can compute $W(\tilde{v}_i, \tilde{v}_j, \bar{X})$ for all $i, j \in \bar{X}$ and $\bar{X} \subseteq P$. Below, we analyze the complexity of Algorithm 3.

THEOREM 3. *The time and space complexity of Algorithm 3 is $O(2^k k^3 + k(m + n \log n))$ and $O(2^k k^2 + n + m)$ respectively, where $k = |P|$ is the number of given labels.*

PROOF. First, the shortest paths for all pairs $(\tilde{v}_j, \tilde{v}_p)$ can be pre-computed in $O(k(m + n \log n))$ time using the Dijkstra algorithm with a Fibonacci heap. Second, there are $2^k k^2$ states in total, thus the total cost for popping the top element from the priority queue is $O(2^k k^2 \log(2^k k^2)) = O(2^k k^3)$ using the Fibonacci heap (line 7). Also, for each state, the cost used in lines 10-15 is $O(k)$ using the Fibonacci heap and pre-computed shortest-path distances. Therefore, the total cost used in lines 10-15 is

Algorithm 3 AllPaths(G, P)

1: $Q \leftarrow \emptyset$; $D \leftarrow \emptyset$;
2: **for all** $p \in P$ **do**
3: Create a virtual node $\tilde{v}_p$ for p;
4: Create an edge $(\tilde{v}_p, v)$ with zero weight for each $v \in V$ including a label p;
5: $Q.push((\tilde{v}_p, \tilde{v}_p, \{\tilde{v}_p\}), 0)$;
6: **while** $Q \neq \emptyset$ **do**
7: $((\tilde{v}_i, \tilde{v}_j, \bar{X}), cost) \leftarrow Q.pop()$;
8: $W(\tilde{v}_i, \tilde{v}_j, \bar{X}) \leftarrow cost$;
9: $D \leftarrow D \cup \{(\tilde{v}_i, \tilde{v}_j, \bar{X})\}$;
10: **for all** $p \in P$ **and** $p \notin \bar{X}$ **do**
11: $\widetilde{X} \leftarrow \bar{X} \cup \{p\}$;
12: $\widetilde{cost} \leftarrow cost + dist(\tilde{v}_j, \tilde{v}_p)$;
13: **if** $(\tilde{v}_i, \tilde{v}_p, \widetilde{X}) \in D$ **then continue**;
14: **if** $(\tilde{v}_i, \tilde{v}_p, \widetilde{X}) \notin Q$ **then** $Q.push((\tilde{v}_i, \tilde{v}_p, \widetilde{X}), \widetilde{cost})$;
15: **if** $\widetilde{cost} < Q.cost(\tilde{v}_i, \tilde{v}_p, \widetilde{X})$ **then** $Q.update((\tilde{v}_i, \tilde{v}_p, \widetilde{X}), \widetilde{cost})$;

$O(2^k k^3)$. Putting it all together, the time complexity of Algorithm 3 is $O(2^k k^3 + k(m + n \log n))$. For the space complexity, we need to maintain the priority queue $\mathcal{Q}$ and the set of all the computed states D which takes $O(2^k k^2)$. We also need to maintain the graph and the all-pair shortest-path distances between the virtual nodes which take $O(m + n + k^2)$ in total. As a result, the space complexity of Algorithm 3 is $O(2^k k^2 + n + m)$. □

It is worth mentioning that for a given query P, we can first invoke Algorithm 3 to pre-compute all the $W(\tilde{v}_i, \tilde{v}_j, \bar{X})$ for all $i, j \in \bar{X}$ and $\bar{X} \subseteq P$ in $O(2^k k^3 + k(m + n \log n))$ time. Moreover, the shortest-path distances $dist(v, \tilde{v}_i)$ and $dist(\tilde{v}_j, v)$ can also be pre-computed within $O(k(m + n \log n))$ time by invoking the Dijkstra algorithm. Consequently, for each state (v, X), we can compute $\pi_{t_1}(v, X)$ in $O(|\bar{X}|)$ time by Eq. (3) and Eq. (4).

Note that the first type of tour-based lower bound needs to find the minimum-weight tour over all virtual-node pairs $(\tilde{v}_i, \tilde{v}_j)$. Below, we propose the second type of tour-based lower bound, which does not take a minimum operation over all virtual-node pairs.

Let $R(\tilde{v}_i, \bar{X})$ be the minimum-weight route that starts from $\tilde{v}_i$ and passes through all virtual nodes in $V_{\bar{X}}$. Further, we let $W(\tilde{v}_i, \bar{X})$ be the weight of the route $R(\tilde{v}_i, \bar{X})$. Then, for a state (v, X), we construct the second type of the tour-based lower bound, denoted by $\pi_{t_2}(v, X)$, using the following formula.

$$\pi_{t_2}(v, X) \triangleq \max_{i \in \bar{X}} \{(dist(v, \tilde{v}_i) + W(\tilde{v}_i, \bar{X}) + \min_{j \in \bar{X}} \{dist(\tilde{v}_j, v)\})\}/2. \quad (6)$$

Eq. (6) includes three parts: (i) the shortest-path distance between the node v to a virtual node $\tilde{v}_i$, (ii) the route $R(\tilde{v}_i, \bar{X})$, and (iii) the minimum shortest-path distance between the virtual nodes in $V_{\bar{X}}$ to the node v. The second type of tour-based lower bound is constructed by taking the maximum weight of the sum over these three parts for all $\tilde{v}_i \in V_{\bar{X}}$. Fig. 3 illustrates the idea of this tour-based lower bound, where $dist(\tilde{v}_a, v) = \min_{j \in \bar{X}} \{dist(\tilde{v}_j, v)\}$. It should be noted that the end point of the route $R(\tilde{v}_i, \bar{X})$ could be any node in $V_{\bar{X}}$. The following lemma shows that $\pi_{t_2}(v, X)$ is indeed a valid lower bound.

LEMMA 3. *For any state* (v, X), *we have* $\pi_{t_2}(v, X) \leq f_T^*(v, \bar{X})$.

To compute $\pi_{t_2}(v, X)$, we can adopt a similar method as used for computing $\pi_{t_1}(v, X)$. Specifically, on the one hand, all the shortest-path distances in Eq. (6) can be pre-computed. On the other hand, we have $W(\tilde{v}_i, \bar{X}) = \min_{j \in \bar{X}} \{W(\tilde{v}_i, \tilde{v}_j, \bar{X})\}$ by definition. Thus, all $W(\tilde{v}_i, \bar{X})$ can also be pre-computed via invoking Algorithm 3. As a result, for each state (v, X), we are able to calculate $\pi_{t_2}(v, X)$ in $O(|\bar{X}|)$ time.

Compared to $\pi_{t_1}(v, X)$, the lower bound $\pi_{t_2}(v, X)$ does not take a minimum operation over all virtual-node pairs. Instead, it takes a maximum operation over all $\tilde{v}_i \in V_{\bar{X}}$. Theoretically, it is very hard to compare the performance of these two lower bounds. In practice, we can combine $\pi_{t_1}(v, X)$ and $\pi_{t_2}(v, X)$ by taking the maximum operation to obtain a better tour-based lower bound $\pi_t(v, X)$, i.e., $\pi_t(v, X) = \max\{\pi_{t_1}(v, X), \pi_{t_2}(v, X)\}$. In addition, we can further combine the one-label and tour-based lower bounds to achieve a better lower bound by taking the maximum operation. Let $\pi(v, X) = \max\{\pi_1(v, X), \pi_t(v, X)\}$, then we have the following lemma.

LEMMA 4. *For any state* (v, X), *we have* $\pi(v, X) \leq f_T^*(v, \bar{X})$.

4.2 The PrunedDP++ algorithm

Armed with the above lower bounds, we are ready to design the PrunedDP++ algorithm, which finds progressively-refined feasible solutions using A^*-search over the pruned search space. Specifically, PrunedDP++ is built on the PrunedDP algorithm. But unlike PrunedDP, PrunedDP++ makes use of $f_T^*(v, X) + \pi(v, X)$ as the priority for each state (v, X) to perform best-first search, where $\pi(v, X)$ is the combined lower bound developed in Section 4.1. Based on such an A^*-search strategy, PrunedDP++ can always select the most promising state to expand, and thus a large number of unpromising states can be pruned. In the experiments, we will show that PrunedDP++ is at least one order of magnitude faster than PrunedDP. The detailed implementation of PrunedDP++ is outlined in Algorithm 4.

The PrunedDP++ algorithm first invokes Algorithm 3 to compute $W(\tilde{v}_i, \tilde{v}_j, \bar{X})$ and $W(\tilde{v}_i, \bar{X})$ for all $\tilde{v}_i, \tilde{v}_j \in V_{\bar{X}}$ and $\bar{X} \subseteq P$ (line 1). Then, the algorithm initializes the priority queue $\mathcal{Q}$ and the set $\mathcal{D}$, and performs the best-first search using the A^*-search strategy (lines 3-26). Note that in Algorithm 4, each state (v, X) is represented by a tuple $((v, X), cost, lb)$, where $cost$ and lb denote the weight and the priority of the state (v, X) respectively. For each state (v, X), lb is obtained by the sum of the weight and the lower bound of that state (i.e., $cost + \pi(v, X)$) in terms of the A^*-search strategy. In Algorithm 4, the lower bound $\pi(v, X)$ and priority lb can be computed by invoking the lb procedure (lines 37-47).

The algorithm also applies the same method as used in Basic and PrunedDP to generate progressively-refined feasible solutions. Unlike Basic and PrunedDP, for any feasible solution constructed by the state (v, X) in PrunedDP++, we can report $f_{\tilde{T}}(v, P)/(f_T^*(v, X) + \pi(v, X))$ as the approximation ratio, because $f_T^*(v, X) + \pi(v, X)$ is a lower bound of the optimal solution by the best-first search strategy. The general framework of PrunedDP++ is very similar to that of PrunedDP, thus we omit the details.

The major differences between PrunedDP++ and PrunedDP are twofold. First, for each state (v, X), PrunedDP++ uses the sum of the weight and the lower bound as the priority, whereas PrunedDP only adopts the weight as the priority. Second, in the update procedure (lines 28-36), PrunedDP++ needs to invoke the lb procedure (lines 37-47) to compute the lower bound and the priority of the expanded state (line 30), and then determines whether the state can be pruned or not (line 32). If the state cannot be pruned, the algorithm needs to update the weight and the lower bound (or the priority) of the state (line 35-36). The PrunedDP algorithm, however, does not need to compute and update the lower bound. In addition, it is important to note that in line 31 of Algorithm 4, for each expanded state, the algorithm must take the maximum over the computed lb and the lb of its parent state, to ensure the *consistency* of the lower bound. After this process, the correctness of the algorithm can be guaranteed, which will be analyzed in the following.

Algorithm 4 PrunedDP++(G, P, S)

Input: $G = (V, E)$, label set S, and the query label set P.
Output: the minimum weight of the connected tree that covers P.

```
1: AllPaths(G, P);
2: best ← +∞; /* maintain the weight of the current feasible solution. */
3: Q ← ∅; D ← ∅;
4: for all v ∈ V do
5:     for all p ∈ S_v do
6:         Q.push((v, {p}), 0, lb((v, {p}), 0, P); /* lb is the priority in Q. */
7: while Q ≠ ∅ do
8:     ((v, X), cost, lb) ← Q.pop(); /* pop the minimum lb element. */
9:     if X = P then return best ← cost;
10:    D ← D ∪ {((v, X), cost)}; X̄ ← P\X;
11:    T'(v, X̄) ← ∅;
12:    for all p ∈ X̄ do
13:        T'(v, X̄) ← T'(v, X̄) ∪ Shortest-Path(v, ṽ_p);
14:    T̃(v, P) ← MST(T'(v, X̄) ∪ T(v, X));
15:    best ← min{best, f_T̃(v, P)};
16:    Report the approximation ratio for best;
17:    if (v, X̄) ∈ D then
18:        update(Q, D, P, best, (v, P), cost + D.cost(v, X̄), lb);
19:        continue;
20:    if cost < best/2 then
21:        for all (v, u) ∈ E do
22:            update(Q, D, P, best, (u, X), cost + w(v, u), lb);
23:        for all X' ⊂ X̄ and (v, X') ∈ D do
24:            cost' ← D.cost(v, X');
25:            if cost + cost' ≤ 2/3 × best then
26:                update(Q, D, P, best, (v, X' ∪ X), cost + cost', lb);
27: return +∞;

28: Procedure update(Q, D, P, best, (v, X), cost, l̃b)
29: if (v, X) ∈ D then return;
30: lb ← lb((v, X), cost, P);
31: lb ← max{lb, l̃b};
32: if lb >= best then return;
33: if X = P then best ← min{best, cost};
34: if (v, X) ∉ Q then Q.push((v, X), cost, lb);
35: if lb < Q.lb((v, X)) then
36:     Q.update((v, X), cost, lb); /* update both cost and lb */

37: Procedure lb((v, X), cost, P)
38: X̄ ← P\X;
39: if X̄ = ∅ then
40:     lb ← 0;
41: else
42:     lb_1 ← min_{i,j∈X̄}{(dist(v, ṽ_i) + W(ṽ_i, ṽ_j, X̄) + dist(ṽ_j, v))}/2;
43:     lb_2 ← max_{i∈X̄}{(dist(u, ṽ_i) + W(ṽ_i, X̄) + min_{j∈X̄}{dist(ṽ_j, u)})}/2;
44:     lb ← max{lb_1, lb_2};
45: for all p ∈ X̄ do
46:     lb ← max{lb, dist(v, ṽ_p)};
47: return cost + lb;
```

Correctness analysis. Recall that the PrunedDP++ algorithm relies on two types of operations to expand a state (v, X): edge growing and tree merging. For the edge growing operation, we can obtain a successor state (u, X) by expanding an edge (v, u), while for the tree merging operation, we can get a successor state $(v, X \cup X')$ through merging with a state (v, X'). To show the correctness of PrunedDP++, we need to prove that the developed lower bounds satisfy the *consistent* property defined in the A^*-search theory [16, 7, 14] for these two state-expansion operations. Below, we first prove that $\pi_1(v, X)$ and $\pi_{t_1}(v, X)$ are consistent, and then we introduce a technique to make $\pi_{t_2}(v, X)$ consistent. Finally, we show that the combination of all these lower bounds is also consistent.

Specifically, Lemma 5 and Lemma 6 show that $\pi_1(v, X)$ and $\pi_{t_1}(v, X)$ are consistent, respectively.

LEMMA 5. *For any state (v, X), we have (i) $\pi_1(u, X) + w(v, u) \geq \pi_1(v, X)$, and (ii) $\pi_1(v, X \cup X') + f^*_T(v, X') \geq \pi_1(v, X)$, where $X \cap X' = \emptyset$.*

LEMMA 6. *For any state (v, X), we have (i) $\pi_{t_1}(u, X) + w(v, u) \geq \pi_{t_1}(v, X)$, and (ii) $\pi_{t_1}(v, X \cup X') + f^*_T(v, X') \geq \pi_{t_1}(v, X)$, where $X \cap X' = \emptyset$.*

Unfortunately, we find in the experiments that $\pi_{t_2}(v, X)$ is not consistent. However, we can use the following technique to make it consistent. In particular, for a state (v, X), let (u, X) and $(v, X \cup X')$ be its successor states by the edge growing and tree merging operations, respectively. Then, for the successor state (u, X), we set $\pi_{t_2}(u, X) = \max\{\pi_{t_2}(u, X), \pi_{t_2}(v, X) - w(v, u)\}$ as the new second type of tour-based lower bound. Likewise, for the successor state $(v, X \cup X')$, we set $\pi_{t_2}(v, X \cup X') = \max\{\pi_{t_2}(v, X \cup X'), \pi_{t_2}(v, X) - f^*_T(v, X')\}$. After this process, it is easy to verify that such a new tour-based lower bound is consistent. According to A^*-search theory, the consistent property also implies that the new bound is a valid lower bound [14]. For convenience, in the rest of this paper, we refer to this new lower bound as the second type of tour-based lower bound, and also denote it by $\pi_{t_2}(u, X)$. We notice that in A^*-search theory, a similar process for inconsistent lower bounds was applied in [26]. In Algorithm 4, we implement this process in line 31.

Finally, we show that the consistent property can be preserved by taking the maximum operation over all the devised consistent lower bounds. In particular, we have the following result.

LEMMA 7. *Let $\pi(v, X) = \max\{\pi_1(v, X), \pi_{t_1}(v, X), \pi_{t_2}(v, X)\}$. Then, for a state (v, X), we have (i) $\pi(u, X) + w(v, u) \geq \pi(v, X)$, and (ii) $\pi(v, X \cup X') + f^*_T(v, X') \geq \pi(v, X)$, where $X \cap X' = \emptyset$.*

Armed with Lemmas 5, 6, and 7, we conclude that Algorithm 4 can find the optimal solution for the GST problem by the optimality of the A^*-search theory [16, 7].

Cost analysis. Note that the worst-case time and space complexity of PrunedDP++ are no higher than those of the PrunedDP algorithm. In the experiments, we show that the PrunedDP++ algorithm can achieve at least one order of magnitude acceleration over the PrunedDP algorithm, using much less memory. This is because in the PrunedDP++ algorithm, the lb in line 32 of Algorithm 4 can be much larger than the weight used in PrunedDP. Moreover, lb increases as the algorithm searches more of search space. On the other hand, the weight of the best feasible solution $best$ progressively decreases, thus the pruning power of our algorithm becomes increasingly strong. If both lb and $best$ are close to optimal, most of the unpromising states will be pruned by our algorithm (line 32). Therefore, compared to PrunedDP, PrunedDP++ will generate a much smaller number of states, and thus both the time and space overhead will be drastically reduced.

Remark. Note that some applications may require to find r connected trees with smallest weight (r is typically very small, e.g., $r \leq 50$). As discussed in [8], the parameterized DP algorithm can be adapted to find the top-r results. Specifically, the algorithm first finds the top-1 result, followed by top-2, top-3, and so on. Clearly, such an algorithm is rather costly, because it is expensive even for finding the top-1 result as analyzed in Section 2. Unlike the parameterized DP algorithm, all the proposed algorithms report progressively-refined results during execution, and the top-1 result is found until algorithms terminate. In the experiments, we find that our algorithms report many near-optimal solutions during execution, and thus we can select the best r results among them as the approximate top-r results. On the other hand, an efficient algorithm for finding the top-1 GST is sufficient for identifying the top-r GST with polynomial delay as shown in [21]. Thus, we can also apply the framework proposed in [21] to extend our algorithms to find the exact top-r results. As a consequence, we believe that our algorithms are also very useful for the applications that require to identify top-r results, albeit we mainly focus on finding the top-1 result (i.e., the minimum-weight connected tree) in this paper.

5. EXPERIMENTS

In this section, we conduct comprehensive experiments to evaluate the proposed algorithms. We implement four various progressive algorithms: Basic (Algorithm 1); PrunedDP (Algorithm 2); PrunedDP+, an A^*-search algorithm based on PrunedDP and the one-label lower bound; and PrunedDP++ (Algorithm 4). For a fair comparison, we use Basic as the baseline, because it produces progressively-refined results, and also is more efficient than the state-of-the-art parameterized DP algorithm, as discussed in Section 3.1. We evaluate the average query processing time, the average memory consumption, and the progressive performance of all algorithms. In all the experiments, the reported query processing time includes the time to pre-compute the shortest paths from each virtual node to the other nodes and the time to pre-compute $W(\tilde{v}_i, \tilde{v}_j, \bar{X})$ by invoking Algorithm 3. We only report the memory allocated for query processing as the memory overhead, without including the memory used to store the graph, because the graph size remains constant for all queries. In addition, we also implement BANKS-II [19], which is the widely-used approximation algorithm for GST search in the keyword search application, to compare the performance between our exact algorithms and existing approximation algorithms. All the experiments are conducted on a machine with two 3.46GHz Intel Xeon CPUs and 96GB main memory running Red Had Enterprise Linux 6.4 (64-bit), and all the algorithms are implemented in C++.

Datasets. We use two large scale real-world datasets, DBLP[1] and IMDB[2], which are two widely-used datasets for keyword search in graphs [8]. The DBLP dataset contains $15,825,211$ nodes and $19,609,604$ edges, while the IMDB datasets includes $30,407,193$ nodes and $46,971,820$ edges. Each node in these datasets consists of a set of labels (or keywords). Our main goal is to evaluate the performance of the GST search algorithms on these datasets. We also perform case studies on these datasets to evaluate the effectiveness of the proposed methods in the keyword search application.

Parameters and query generation. We vary two parameters in our experiments, namely, knum and kwf. knum is the number of labels in the query P (i.e., knum $= |P|$), and kwf is the average number of nodes containing each label in the query (i.e., the label frequency). knum is selected from 5, 6, 7, and 8 with a default value of 6, and kwf is selected from 200, 400, 800, and 1600 with a default value of 400. Unless otherwise specified, when varying a parameter, the values of the other parameters are set to their default values. Each query is generated by randomly selecting knum labels from the label set. In each test, we randomly generate 50 queries and report the average results over all of them.

We mainly perform six different experiments based on the above setting. In the first three experiments, we intend to test the query processing time and memory usage of various algorithms in a progressive manner. We vary the approximation ratio from 8 to 1 for each algorithm and report the average time/memory consumption with a decreasing approximation ratio during algorithm execution. In the fourth experiment, we aim to evaluate the progressive performance of various algorithms. In the last two experiments, we compare our best algorithm with the approximation algorithm BANKS-II, and conduct case studies to show the effectiveness of our approaches in the keyword search application. Additionally, we test the performance of our algorithms on graphs with different topologies, and also show the performance of our best algorithm for relatively large knum values. Due to space limit, these two additional experiments are reported in Appendix A.2.

Exp-1: Query processing time (vary knum). In this experiment, we vary knum from 5 to 8 and test the processing time of various algorithms with a decreasing approximation ratio. The results for the DBLP and IMDB datasets are shown in Fig. 4 and Fig. 5 respectively. Since the results on these two datasets are very similar, we focus on describing the results on the DBLP dataset. Specifically, from Fig. 4, we make the following observations.

First, all algorithms take a certain amount of initialization time to pre-compute the shortest paths from each virtual node to all the nodes in the graph, before reporting a feasible solution. PrunedDP++ also spends time pre-computing all $W(\tilde{v}_i, \tilde{v}_j, \bar{X})$ by invoking Algorithm 3. The initialization time increases when knum increases. The time to compute all $W(\tilde{v}_i, \tilde{v}_j, \bar{X})$ in PrunedDP++ can almost be omitted because such time cost is independent of the graph size as shown in Section 4.1.

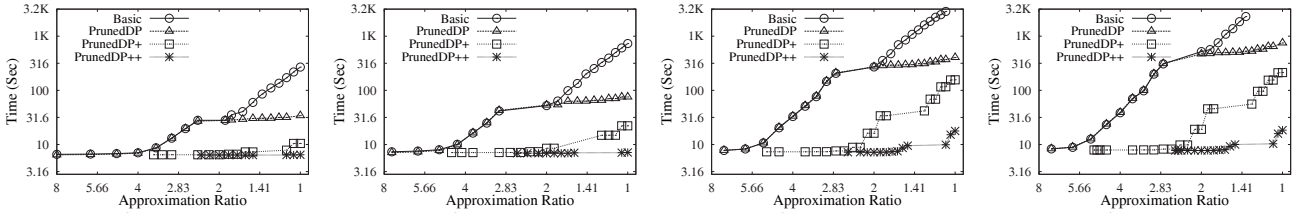
Second, the approximation ratio for each algorithm becomes better when more time is spent, until it finally reaches 1, which indicates that the optimal solution has been found. The processing time of Basic and PrunedDP are very similar when the approximation ratio is larger than 2 because, in PrunedDP, the weight of most intermediate states cannot reach $f^*(P)/2$ and they apply the same process to compute the weights of those states. However, when the approximation ratio becomes smaller, the processing time of Basic increases sharply, while the processing time of PrunedDP becomes stable. PrunedDP+ shows significant improvement over PrunedDP using the A^*-search strategy, and PrunedDP++ makes further improvement over PrunedDP+ using the novel tour-based lower bounding techniques. For example, when knum $= 6$, Basic, PrunedDP, PrunedDP+, and PrunedDP++ take 732.4 seconds, 75.5 seconds, 22.2 seconds, and 7.1 seconds respectively to find the optimal solution as shown in Fig. 4 (b). In general, PrunedDP++ is more than one order of magnitude faster than PrunedDP and more than two orders of magnitude faster than Basic. These results are consistent with our theoretical analysis in Sections 3 and 4.

Finally, to achieve a certain approximation ratio, the processing time for each algorithm increases with increasing knum. Taking PrunedDP as an example, to achieve an approximation ratio of 1 for knum $= 5, 6, 7, 8$, the algorithm spends 34.1 seconds (Fig. 4 (a)), 75.5 seconds (Fig. 4 (b)), 404.7 seconds (Fig. 4 (c)) , and 750.9 seconds (Fig. 4 (d)) respectively. When knum increases, the gap between PrunedDP++ and PrunedDP+ increases and the gap between PrunedDP+ and PrunedDP decreases. This is because when knum increases, the lower bound used in PrunedDP+ tends to be more loose, which degrades the performance of the PrunedDP+ algorithm.

Exp-2: Query processing time (vary kwf). In this experiment, we vary kwf from 200 to 1600, and test the processing time with a decreasing approximation ratio. The results on the DBLP and IMDB datasets are reported in Fig. 6 and Fig. 7 respectively. Again, the results on these datasets are very similar, so we focus on describing the results on the DBLP dataset. From Fig. 6, we make the following observations: First, when kwf increases, the processing time of Basic, PrunedDP, and PrunedDP+ tend to decrease. The reason is that when kwf is small, the weight of the optimal tree tends to be large, which may result in longer processing time for Basic, PrunedDP, and PrunedDP+. For example, when kwf decreases from 1600 to 200, to achieve an approximation ratio of 1, the processing time of PrunedDP increases from 55.3 seconds to 141.5 seconds. However, the performance of PrunedDP++ is not largely influenced by kwf of the query due to the effective tour-based lower bounding techniques. Second, when kwf increases, the gap between PrunedDP+ and PrunedDP++ decreases. For example, for kwf $= 200, 400, 800$, and 1600, to achieve an approximation ratio of 1, the processing time of PrunedDP+ is 10 times (Fig. 6 (a)), 3 times (Fig. 6 (b)), 2.5 times (Fig. 6 (c)), and 1.4 times (Fig. 6 (d)) of that of PrunedDP++, respectively. The reason is that, when kwf increases, the lower bound computed by PrunedDP+ tends

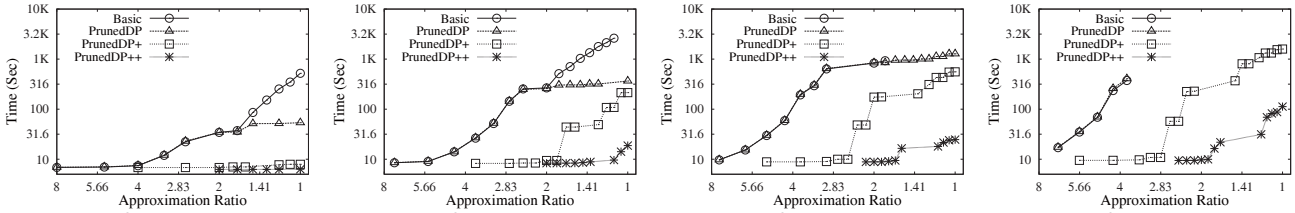
[1]http://dblp.dagstuhl.de/xml/

[2]http://www.imdb.com/interfaces/



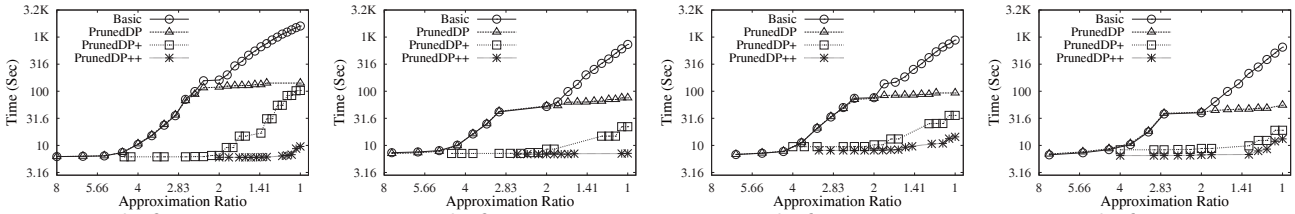
(a) knum = 5 (b) knum = 6 (c) knum = 7 (d) knum = 8

Figure 4: Query Processing Time vs. Approximation Ratio: Vary Number of Labels (knum) on DBLP



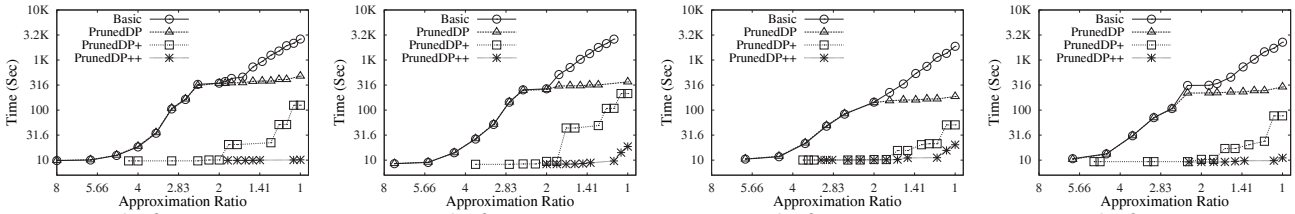
(a) knum = 5 (b) knum = 6 (c) knum = 7 (d) knum = 8

Figure 5: Query Processing Time vs. Approximation Ratio: Vary Number of Labels (knum) on IMDB



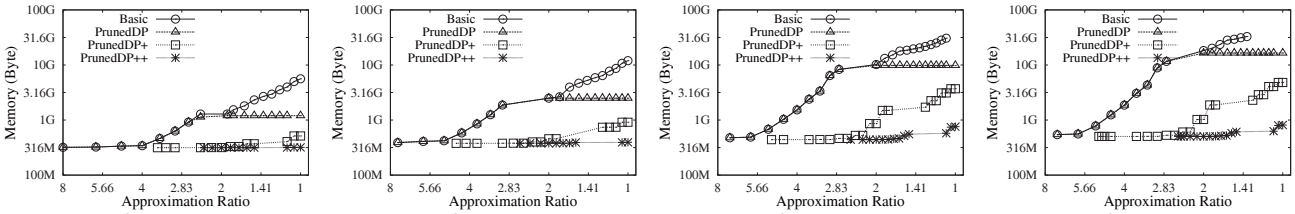
(a) kwf = 200 (b) kwf = 400 (c) kwf = 800 (d) kwf = 1600

Figure 6: Query Processing Time vs. Approximation Ratio: Vary Label Frequency (kwf) on DBLP



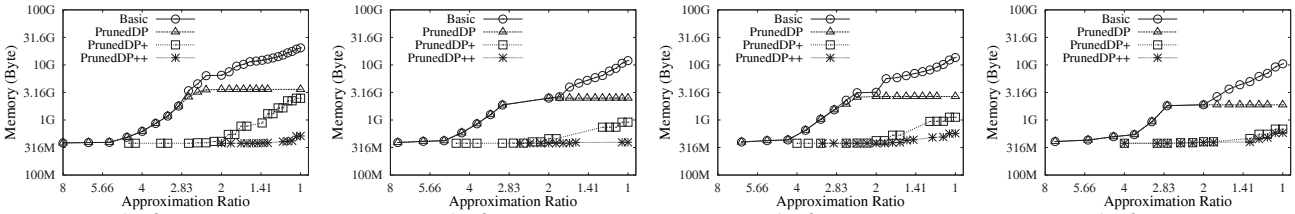
(a) kwf = 200 (b) kwf = 400 (c) kwf = 800 (d) kwf = 1600

Figure 7: Query Processing Time vs. Approximation Ratio: Vary Label Frequency (kwf) on IMDB



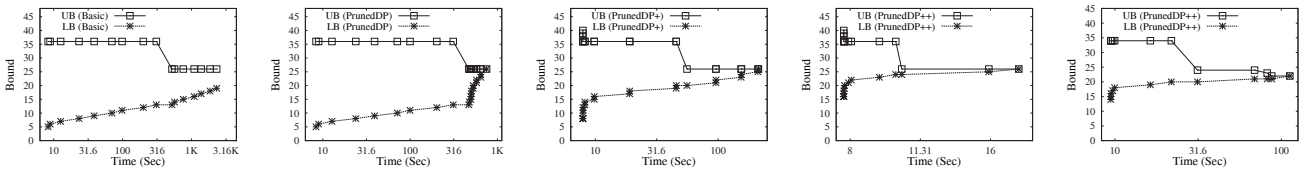
(a) knum = 5 (b) knum = 6 (c) knum = 7 (d) knum = 8

Figure 8: Memory Overhead vs. Approximation Ratio: Vary Number of Labels (knum) on DBLP



(a) kwf = 200 (b) kwf = 400 (c) kwf = 800 (d) kwf = 1600

Figure 9: Memory Overhead vs. Approximation Ratio: Vary Label Frequency (kwf) on DBLP



(a) Basic (DBLP) (b) PrunedDP (DBLP) (c) PrunedDP+ (DBLP) (d) PrunedDP++ (DBLP) (e) PrunedDP++ (IMDB)

Figure 10: Progressive Performance Testing (knum $= 8$, kwf $= 400$)

knum	kwf	Total Time BANKS-II	Appro Ratio BANKS-II	Total Time PrunedDP++	T_r PrunedDP++
5	400	88.0 secs	1.22	6.5 secs	6.4 secs
6	400	109.1 secs	1.40	7.1 secs	7.1 secs
7	400	131.2 secs	1.31	17.8 secs	9.9 secs
8	400	162.4 secs	1.38	18.4 secs	7.8 secs
6	200	110.8 secs	1.07	9.6 secs	6.4 secs
6	800	109.0 secs	1.20	14.4 secs	10.7 secs
6	1,600	109.1 secs	1.25	13.2 secs	6.8 secs

Table 2: Comparison with BANKS-II on DBLP

to be tighter and tighter, and thus the bounds of PrunedDP+ and PrunedDP++ tend to be similar.

Exp-3: Memory consumption. In this experiment, we evaluate the memory overhead of all the algorithms with a decreasing approximation ratio. To this end, we vary the parameters knum from 5 to 8 and kwf from 200 to 1600, respectively. We test all the algorithms on the DBLP dataset, and similar results can also be observed on the IMDB dataset. Fig. 8 and Fig. 9 depict the results with varying knum from 5 to 8 and varying kwf from 200 to 1600, respectively. Compared to Fig. 4 and Fig. 6, the curves for memory consumption for all the algorithms are very similar to those for query processing time under the same parameter settings. This is because both the memory and time overhead for each algorithm are roughly proportional to the number of states generated and, unsurprisingly, the curves are similar. In addition, PrunedDP++ uses less than 1GB memory to find the optimal solution even when knum =8. In the same setting, PrunedDP+ uses around 3.16GB memory and PrunedDP uses more than 10GB memory. The baseline algorithm (Basic) performs even worse, using more than 31.6GB memory to obtain a 1.41-approximation solution (see Fig. 8(e)). These results show that PrunedDP++ is indeed much more memory-efficient than the other algorithms.

Exp-4: Progressive Performance Testing. In this experiment, we test how well the reported feasible solutions are progressively improved during algorithm execution. To this end, we report the lower bounds denoted by LB and the weights of the feasible solutions denoted by UB (because it is an upper bound of the optimal solution) when a feasible solution is generated for all the algorithms. We set knum $= 8$ and kwf $= 400$, and similar results can also be obtained for the other values of knum and kwf. The results are shown in Fig. 10. As desired, for each algorithm, we can see that LB monotonically increases and UB monotonically decreases with increasing running time. In general, for each algorithm, the gap between the reported LB and UB becomes smaller with increasing running time. Clearly, if such a gap closes, the algorithm has found the optimal solution. However, the gap for the Basic algorithm (Fig. 10(a)) is relatively large compared to the other algorithms. Moreover, it dose not close within the one-hour time limit on the DBLP dataset. The gap between LB and UB for the PrunedDP algorithm (Fig. 10(b)) quickly closes on the DBLP dataset when the running time is larger than 316 seconds. The reason is as follows. After 316 seconds, the weight of most states in the PrunedDP algorithm may reach $f^*(P)/2$, thus such states will not be further expanded by the algorithm. As a consequence, the optimal solution can be quickly generated by merging these states. These results indicate that the PrunedDP algorithm exhibits much better progressive performance than the baseline algorithm. Furthermore, as can be seen from Fig. 10(c) and Fig. 10(d), PrunedDP+ shows much better progressive performance than PrunedDP, and PrunedDP++ can further achieve significant performance improvement over PrunedDP+ on the DBLP dataset. For example, the PrunedDP++ algorithm quickly narrows the gap between LB and UB in less than 10 seconds, and closes the gap within 20 seconds, whereas the PrunedDP+ algorithm does not close the gap in 100 seconds. We also show the result of PrunedDP++ on the IMDB dataset in Fig. 10(e). As can be seen, the PrunedDP++ algorithm results in a 2-approximation solution within 10 seconds, finds a near-optimal solution within 31.6 seconds, and obtains the optimal solution taking around 110 seconds. These results demonstrate that the PrunedDP++ algorithm exhibits excellent progressive performance in practice, which further confirm our theoretical findings.

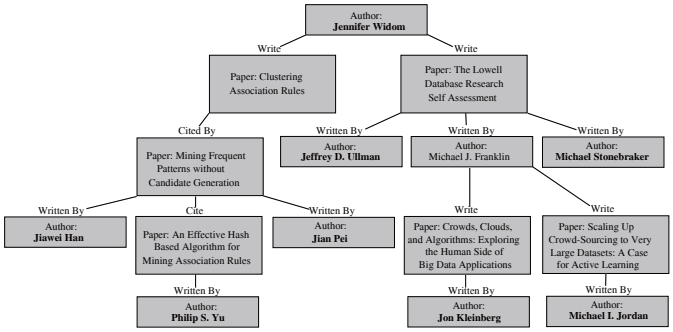


Figure 11: Case Study on DBLP using PrunedDP++ (knum $=$ 8, Processing Time $= 7.5$ Seconds)

Exp-5: Comparison with the Approximation Algorithm. In this experiment, we compare PrunedDP++ with the approximation algorithm BANKS-II [19]. We report the processing time of BANKS-II and PrunedDP++, as well as the approximation ratio of the best answer reported by BANKS-II. Since PrunedDP++ reports answers in a progressive manner, we also report the processing time of PrunedDP++ to generate an answer with at least the same quality as BANKS-II, and we denote such processing time as T_r. The results on the DBLP dataset are shown in Table 2, and similar results on IMDB dataset are shown in Appendix A.2. We test 7 different combinations of knum and kwf. As can be seen, PrunedDP++ is 7.4 to 15.4 times faster than BANKS-II, because BANKS-II typically needs to explore the whole graph to get an approximate answer while PrunedDP++ visits only a part of the graph to output the optimal answer. For example, with the default parameter setting (knum $= 6$ and kwf $= 400$), BANKS-II computes an 1.4-approximate answer in 109.1 seconds while PrunedDP++ outputs the optimal answer taking only 7.1 seconds. When knum $= 8$, although PrunedDP++ requires 18.4 seconds to get the optimal answer, it only needs 7.8 seconds to obtain the answer with at least the same quality as the answer returned by BANKS-II. In this case, PrunedDP++ is 20.8 times faster than BANKS-II to generate the answer with the same quality.

Exp-6: Case Study. In this experiment, we conduct a case study on the DBLP dataset to compare the effectiveness of the exact GST computed by PrunedDP++ and the approximate GST calculated

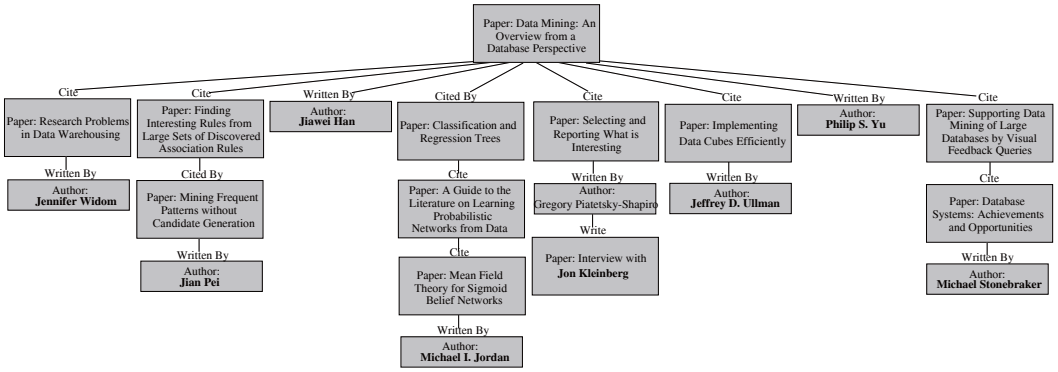


Figure 12: Case Study on DBLP using BANKS-II (knum $= 8$, Processing Time $= 151.6$ Seconds)

by the approximation algorithm BANKS-II [19]. An additional case study on the IMDB dataset can be found in Appendix A.2. We use the query $P = \{$"Jiawei Han", "Philip S. Yu", "Jeffrey D. Ullman", "Jian Pei", "Jennifer Widom", "Michael Stonebraker", "Jon Kleinberg", "Michael I. Jordan"$\}$, which includes 8 professor names. The answer reported by PrunedDP++ (taking 7.5 seconds) is shown in Fig. 11. As can be seen, the answer contains 14 edges, and the 8 professors are divided into two subgroups. The first subgroup contains "Jiawei Han", "Philip S. Yu", and "Jian Pei", in which "Jiawei Han" and "Jian Pei" collaboratively write a paper which cites a paper written by "Philip S. Yu". The second subgroup contains "Jeffrey D. Ullman", "Michael Stonebraker", 'Jon Kleinberg", "Michael I. Jordan", and "Jennifer Widom", in which "Jon Kleinberg" and "Michael I. Jordan" each collaborates with another professor "Michael J. Franklin" who further collaborates with "Jeffrey D. Ullman", "Michael Stonebraker", and "Jennifer Widom". The two subgroups are connected by "Jennifer Widom". The answer reported by BANKS-II (taking 151.6 seconds) is shown in Fig. 12. In this answer, a paper written by "Jiawei Han" and "Philip S. Yu" cites another five papers which is directly/indirectly related to five professor names in P, and there are 19 edges in total. Such a result does not exhibit any subgroup information. Obviously, the answer reported by PrunedDP++ is more compact and better captures the relationship among the 8 authors in P than the answer returned by BANKS-II. These results confirm the effectiveness of our approaches for the keyword search application.

6. RELATED WORK

The GST problem and its applications. The GST problem is a generalization of the traditional Steiner tree problem which was introduced by Reich and Widmayer in [28], motivated by an application in VLSI design. Recently, the GST problem has been widely used to keyword search in the database community [3, 19, 8], and it has also been applied to search for a team of experts in a social network [22]. From a computational point of view, the GST problem is known to be NP-hard [28]. In [18], Ihler showed that the GST problem cannot be approximated within a constant performance ratio by any polynomial algorithm unless P=NP. The lower bound of the approximation ratio is known to be $O(\ln k)$ [12], where k denotes the number of groups. The polylogarithmic approximation algorithms for the GST problem were obtained in [5, 13] based on the technique of linear programming relaxation. However, such approximation algorithms are very hard to handle large graphs, due to the high computational overhead of linear programming. To solve the GST problem optimally, Ding et al. [8] proposed a parameterized DP algorithm using the number of groups k as a parameter, which is a generalization of the well-known Dreyfus-Wagner algorithm for the traditional Steiner tree problem [9]. The parameterized DP algorithm was shown to be efficient for handling large graphs when the parameter k is very small (e.g., $k = 4$) [8]. However, the parameterized DP algorithm cannot provide progressively-refined solutions and it only works well for very small k values. Instead, in this paper, we propose several efficient and progressive algorithms to solve the GST problem. Our best algorithm is shown to be at least two orders of magnitude faster than the parameterized DP algorithm, using much less memory.

Anytime algorithms. Our work is also closely related to the anytime algorithms that have been studied in artificial intelligent and data mining communities [27, 32, 15, 29, 4]. For example, Mouaddib and Zilberstein [27] studied a knowledge-based anytime algorithm for real-time decision making applications. Hansen and Zhou [15] proposed an anytime heuristic search algorithm based on the technique of weighted heuristic search algorithm. Ueno et al. [29] proposed an anytime nearest neighbor classification algorithm for stream mining. Esmeir and Markovitch [10] studied an anytime decision tree induction algorithm for classification task. Mai et al. [4] proposed an anytime density-based clustering algorithm based on a lower-bounding technique for the similarity measure. Generally, all the above anytime algorithms are algorithms that can be terminated at any time to return an intermediate solution, and the quality of the intermediate solution can be improved progressively. However, unlike our progressive GST algorithm, all the above mentioned anytime algorithms do not focus on graph data. To the best of our knowledge, this is the first work to study progressive algorithms for the GST problem over large-scale graph data. It is worth mentioning that progressive algorithms were also studied in the computational geometry and database communities. For example, in [31], Zhang et al. proposed a progressive algorithm for computing the optimal location query problem in spatial databases. Alewijnes et al. [1] proposed several progressive algorithms for computational geometric problems. Once again, all of those algorithms do not focus on graph data. In addition, the term of progressive algorithms were also used in keyword search literature [23, 24]. However, substantially different from our progressive search algorithm, the algorithms proposed in [23, 24] are based on the classic threshold algorithm [11] to progressively identify the top-r answers.

7. CONCLUSION

In this paper, we first propose an efficient and progressive algorithm, called PrunedDP, based on two novel optimal-tree decomposition and conditional tree merging techniques. The PrunedDP algorithm is not only able to produce progressively-refined results during algorithm execution, but it is also one order of magnitude faster than the state-of-the-art algorithm. Second, to further speed up the PrunedDP algorithm, we propose a progressive A^*-search algorithm, named PrunedDP++, which can achieve at least one order of magnitude acceleration over the PrunedDP algorithm. Finally, we conduct extensive performance studies over several large-scale graphs (≥ 10 million nodes), and the results demonstrate the high efficiency and effectiveness, as well as excellent progressive performance of the proposed algorithms.

8. REFERENCES

[1] S. P. A. Alewijnse, T. M. Bagautdinov, M. de Berg, Q. W. Bouts, A. P. ten Brink, K. Buchin, and M. A. Westenberg. Progressive geometric algorithms. In *SOCG*, 2014.

[2] A. Anagnostopoulos, L. Becchetti, C. Castillo, A. Gionis, and S. Leonardi. Online team formation in social networks. In *WWW*, 2012.

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

[4] C. Böhm, J. Feng, X. He, and S. T. Mai. Efficient anytime density-based clustering. In *SDM*, 2013.

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

[6] J. Coffman and A. C. Weaver. An empirical performance evaluation of relational keyword search techniques. *IEEE Trans. Knowl. Data Eng.*, 26(1):30–42, 2014.

[7] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the ACM*, 32(3):505–536, 1985.

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

[9] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[10] S. Esmeir and S. Markovitch. Anytime induction of decision trees: An iterative improvement approach. In *AAAI*, 2006.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[12] U. Feige. A threshold of ln *n* for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[13] N. Garg, G. Konjevod, and R. Ravi. A polylogarithmic approximation algorithm for the group steiner tree problem. *J. Algorithms*, 37(1):66–84, 2000.

[14] A. V. Goldberg and C. Harrelson. Computing the shortest path: *A** search meets graph theory. In *SODA*, 2005.

[15] E. A. Hansen and R. Zhou. Anytime heuristic search. *J. Artif. Intell. Res. (JAIR)*, 28:267–297, 2007.

[16] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[17] H. He, H. Wang, J. Yang, and P. S. Yu. BLINKS: ranked keyword searches on graphs. In *SIGMOD*, 2007.

[18] E. Ihler. The complexity of approximating the class steiner tree problem. In *17th International Workshop, WG*, 1991.

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

[20] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. STAR: steiner-tree approximation in relationship graphs. In *ICDE*, 2009.

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

[22] T. Lappas, K. Liu, and E. Terzi. Finding a team of experts in social networks. In *KDD*, 2009.

[23] G. Li, C. Li, J. Feng, and L. Zhou. SAIL: structure-aware indexing for effective and progressive top-k keyword search over XML documents. *Inf. Sci.*, 179(21):3745–3762, 2009.

[24] G. Li, X. Zhou, J. Feng, and J. Wang. Progressive keyword search in relational databases. In *ICDE*, 2009.

[25] A. Majumder, S. Datta, and K. V. M. Naidu. Capacitated team formation problem on social networks. In *KDD*, 2012.

[26] L. Mero. A heuristic search algorithm with modifiable estimate. *Artificial Intelligence*, 23(1):13–27, 1984.

[27] A. Mouaddib and S. Zilberstein. Knowledge-based anytime computation. In *IJCAI*, 1995.

[28] G. Reich and P. Widmayer. Beyond steiner's problem: A VLSI oriented generalization. In *15th International Workshop,WG*, 1989.

[29] K. Ueno, X. Xi, E. J. Keogh, and D. Lee. Anytime classification using the nearest neighbor algorithm with applications to stream mining. In *ICDM*, 2006.

[30] X. Wang, Z. Zhao, and W. Ng. A comparative study of team formation in social networks. In *DASFAA*, 2015.

[31] D. Zhang, Y. Du, T. Xia, and Y. Tao. Progressive computation of the min-dist optimal-location query. In *VLDB*, 2006.

[32] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.

Acknowledgements. The work was supported in part by (i) NSFC Grants (61402292, U1301252, 61033009), NSF-Shenzhen Grants (JCYJ20150324140036826, JCYJ20140418095735561), and Startup Grant of Shenzhen Kongque Program (827/000065); (ii) ARC DE140100999 and ARC DP160101513; (iii) Research Grants Council of the Hong Kong SAR, China, 14209314; (iv) Guangdong Key Laboratory Project (2012A061400024). Dr. Rui Mao is a corresponding author.

A. APPENDIX

A.1 Missing proofs

THEOREM 1. ***Optimal-Tree Decomposition Theorem**: Assume that each edge in the graph G has a positive weight. Then, for the optimal tree $T^*(P)$, there always exists a node $u \in T^*(P)$ such that (i) the tree $T^*(P)$ rooted at u has k ($k \geq 1$) subtrees $T_1, T_2, \cdots, T_k$, and (ii) each subtree T_i (for $i \in \{1, 2, \cdots, k\}$) has a weight smaller than $f^*(P)/2$.*

PROOF. We can construct the optimal-tree decomposition using the following procedure. First, we randomly select a node from $T^*(P)$ as the root u. If all the subtrees of the tree rooted at u have weights smaller than $f^*(P)/2$, we are done. Otherwise, there must be a subtree T_i that has a weight no smaller than $f^*(P)/2$. Let v_i be the root of such a subtree T_i. Then, we move the root from u to v_i, and recursively execute the same procedure. Fig. 13 illustrates this movement procedure. Note that the movement procedure is nonreversible (i.e., we cannot move the root from v_i to u), because the subtree rooted at v_i (i.e., T_i) has a weight no smaller than $f^*(P)/2$ whereas the remaining part of the tree rooted at u has a weight smaller than $f^*(P)/2$. More specifically, let us consider the tree shown in Fig. 13 (left panel). Since the tree rooted at v_i has a weight no smaller than $f^*(P)/2$ and the weight of the edge (u, v_i) is positive, thus the weight of the left-part subtree rooted at u must be smaller than $f^*(P)/2$. As a result, the procedure cannot move back from v_i to u, and thus in each movement, the procedure must replace the root with a new node. Since the number of nodes in $T^*(P)$ is bounded, the movement procedure must be terminated in finite number of steps. Once the procedure terminates, we obtain a tree rooted at u that satisfies the conditions as stated in the theorem. □

THEOREM 2. ***Conditional Tree Merging Theorem**: Without loss of optimality, to expand a state (v, X) by a tree merging operation in the best-first DP algorithm, we can merge two subtrees $T(v, X)$ and $T(v, X')$ for $X' \subset P \backslash X$ only when the total weight of these two subtrees is no larger than $2/3 \times f^*(P)$.*

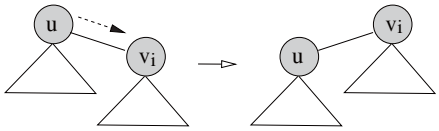


Figure 13: Illustration of the movement from a root u to v_i.

PROOF. To prove the theorem, we need to show that the optimal solution can still be obtained using the *conditional* tree merging strategy. By Theorem 1, we can assume, without loss of generality, that the optimal solution is a tree rooted at u with k subtrees, $T_1, \cdots, T_k$, whose roots are v_i respectively and weights are smaller than $f^*(P)/2$ (see Fig. 1). Let $\tilde{T}_i$ be the edge-grown subtree that is grown by T_i with an edge (v_i, u). Then, there are three different cases: (1) the weight of each $\tilde{T}_i$ is smaller than $f^*(P)/2$, (2) there is only one edge-grown subtree $\tilde{T}_i$ that has a weight no smaller than $f^*(P)/2$, and (3) there are two edge-grown subtrees and each one has a weight $f^*(P)/2$.

In case (1), we claim that the optimal tree can always be divided into two subtrees such that each of them has a weight no larger than $2f^*(P)/3$. In particular, we can obtain such a division using the following procedure. First, we pick the top-1 maximum-weight edge-grown subtree. If its weight is no smaller than $f^*(P)/3$, we are done. This is because, the union of the remaining edge-grown subtrees must have a weight no larger than $2f^*(P)/3$. Since the top-1 maximum-weight edge-grown subtree has a weight smaller than $f^*(P)/2$ (by the condition of case (1)), its weight is also smaller than $2f^*(P)/3$, and thus we obtain a feasible division. Otherwise, we pick the top-2 maximum-weight edge-grown subtree, and merge it with the top-1 maximum-weight edge-grown tree. If the merged edge-grown tree has a weight no smaller than $f^*(P)/3$, then we are done. Since the top-1 maximum-weight edge-grown subtree has a weight smaller than $f^*(P)/3$ in this case, the weight of the merged edge-grown tree must be smaller than $2f^*(P)/3$ (because the weight of the top-2 maximum-weight edge-grown subtree is also smaller than $f^*(P)/3$). On the other hand, the union of the remaining edge-grown subtrees must have a weight no larger than $2f^*(P)/3$. Thus, in this case, we obtain a feasible division. We can perform the same procedure recursively until we get a feasible division. Therefore, under the case (1), to expand a state (v, X) in the best-first DP algorithm, there is no need to merge two subtrees if the total weight of them is larger than $2f^*(P)/3$. This is because under the case (1), the optimal solution can always be obtained by merging two subtrees with weights no larger than $2f^*(P)/3$, and thus we do not need to generate a subtree that has a weight larger than $2f^*(P)/3$ via merging two subtrees.

In case (2), since there is an edge-grown subtree $\tilde{T}_i$ with weight no smaller than $f^*(P)/2$, thus by our best-first DP algorithm, the state corresponding to such a tree $\tilde{T}_i$ will not be expanded (see Theorem 1). On the other hand, the total weight of all the remaining edge-grown subtrees must be no larger than $f^*(P)/2$, and thus no larger than $2/3 \times f^*(P)$. In case (3), since both the two edge-grown subtrees have weights $f^*(P)/2$, there is no need to expand the states corresponding to these two subtrees by our algorithm. Putting it all together, the theorem is established. □

LEMMA 1. *For any state (v, X), we have $\pi_1(v, X) \leq f^*_T(v, \bar{X})$.*

PROOF. Since $\pi_1(v, X)$ denotes the weight of the optimal tree rooted at v that covers only one label in $\bar{X}$, $\pi_1(v, X)$ must be smaller than the weight of the optimal tree rooted at v that covers all labels in $\bar{X}$, which is exactly equal to $f^*_T(v, \bar{X})$. □

LEMMA 2. *For any state (v, X), we have $\pi_{t_1}(v, X) \leq f^*_T(v, \bar{X})$.*

PROOF. Recall that $f^*_T(v, \bar{X})$ denotes the optimal weight of a connected tree rooted at v that covers all the labels in $\bar{X}$. We can double every edge in this optimal tree $T(v, \bar{X})$, and thus obtain an Euler tour that starts from v and also ends at v. Clearly, the weight of such a tour should be no less than the weight of the optimal tour starting from v and ending at v, and covering all the labels in $\bar{X}$. As a result, we have $2f^*_T(v, \bar{X}) \geq f^*_{\tilde{R}}(v, \bar{X})$, and thereby the lemma is established. □

LEMMA 3. *For any state (v, X), we have $\pi_{t_2}(v, X) \leq f^*_T(v, \bar{X})$.*

PROOF. First, we can add the virtual nodes and virtual edges for the optimal tree $T(v, \bar{X})$, and thus result in a graph with weight $f^*_T(v, \bar{X})$. Then, we double the edges of such a graph, and obtain an Euler graph $\tilde{G}(v, \bar{X})$ with weight $2f^*_T(v, \bar{X})$. Clearly, in $\tilde{G}(v, \bar{X})$, we have an Euler tour that starts from v, ends at v, and passes through all edges in $\tilde{G}(v, \bar{X})$. For each virtual node $\tilde{v}_i \in V_{\bar{X}}$, we can always decompose such an Euler tour into three parts: (i) the path from v to $\tilde{v}_i$, (ii) the minimum-weight route $\tilde{R}$ that starts from $\tilde{v}_i$ and passes through all virtual nodes in $V_{\bar{X}}$, and (iii) the path from $\tilde{v}_j$ to v, where $\tilde{v}_j$ is the end node of the route $\tilde{R}$. By our definition, the total weight of these three parts must be no less than $dist(v, \tilde{v}_i) + W(\tilde{v}_i, \bar{X}) + \min_{j \in \bar{X}}\{dist(\tilde{v}_j, v)\}$. Since this result hold for every virtual node $\tilde{v}_i$, we can conclude that the weight of the Euler tour must be no less than $\max_{i \in \bar{X}}\{(dist(v, \tilde{v}_i) + W(\tilde{v}_i, \bar{X}) + \min_{j \in \bar{X}}\{dist(\tilde{v}_j, v)\})\}$, and therefore the lemma is established. □

LEMMA 4. *For any state (v, X), we have $\pi(v, X) \leq f^*_T(v, \bar{X})$.*

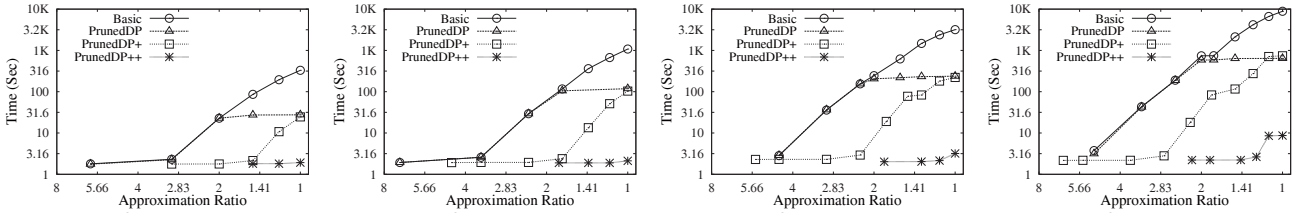
PROOF. Since $\pi(v, X) = \max\{\pi_1(v, X), \pi_{t_1}(v, X), \pi_{t_2}(v, X)\}$ and $\pi_1(v, X), \pi_{t_1}(v, X)$, and $\pi_{t_2}(v, X)$ are the lower bounds of $f^*_T(v, \bar{X})$, thus the lemma holds. □

LEMMA 5. *For any state (v, X), we have (i) $\pi_1(u, X) + w(v, u) \geq \pi_1(v, X)$, and (ii) $\pi_1(v, X \cup X') + f^*_T(v, X') \geq \pi_1(v, X)$, where $X \cap X' = \emptyset$.*

PROOF. By definition, to prove the lemma, we need to prove that (i) $f^*_1(u, \bar{X}) + w(v, u) \geq f^*_1(v, \bar{X})$, and (ii) $f^*_1(v, \bar{X}\backslash X') + f^*_T(v, X') \geq f^*_1(v, \bar{X})$. First, we have $f^*_1(u, \bar{X}) + w(v, u) = \max_{x \in \bar{X}}\{f^*_T(u, \{x\}) + w(v, u)\}$. Clearly, $f^*_T(u, \{x\}) + w(v, u)$ is the weight of a tree that is rooted at v and covers a label x. Since $f^*_T(v, \{x\})$ is the optimal weight over all such type of trees, we have $f^*_T(u, \{x\}) + w(v, u) \geq f^*_T(v, \{x\})$, and thereby we have $\max_{x \in \bar{X}}\{f^*_T(u, \{x\}) + w(v, u)\} \geq \max_{x \in \bar{X}}\{f^*_T(v, \{x\})\} = f^*_1(v, \bar{X})$. Second, since $f^*_T(v, X') \geq f^*_1(v, X')$, it is sufficient to show $f^*_1(v, \bar{X}\backslash X') + f^*_1(v, X') \geq f^*_1(v, \bar{X})$. By the definition of the one-label lower bound, the above inequality clearly holds. □

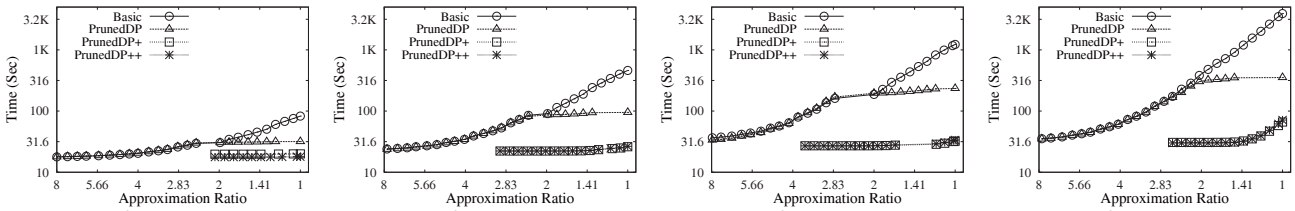
LEMMA 6. *For any state (v, X), we have (i) $\pi_{t_1}(u, X) + w(v, u) \geq \pi_{t_1}(v, X)$, and (ii) $\pi_{t_1}(v, X \cup X') + f^*_T(v, X') \geq \pi_{t_1}(v, X)$, where $X \cap X' = \emptyset$.*

PROOF. First, we prove that the case (i) holds. Recall that $f^*_{\tilde{R}}(u, \bar{X})$ is the optimal weight of the tour (denoted by $(u \sim R(\tilde{v}_i, \tilde{v}_j, \bar{X}) \sim u)$) that starts from u, ends at u, and passes through all the virtual nodes in $V_{\bar{X}}$. Based on this optimal tour, we can construct a new tour $v \sim u \sim R(\tilde{v}_i, \tilde{v}_j, \bar{X}) \sim u \sim v$. Clearly, the weight of this new tour is $f^*_{\tilde{R}}(u, \bar{X}) + 2 \times w(v, u)$. Note that such a new tour is tour that starts from v, ends at v, and passes through all the virtual nodes in $V_{\bar{X}}$. Since $f^*_{\tilde{R}}(v, \bar{X})$ is the optimal weight of the tour that starts from v, ends at v, and passes through all virtual nodes in $V_{\bar{X}}$, we have $f^*_{\tilde{R}}(u, \bar{X}) + 2w(v, u) \geq f^*_{\tilde{R}}(v, \bar{X})$. Since $\pi_{t_1}(u, X) = f^*_{\tilde{R}}(u, \bar{X})/2$ by definition, we have $\pi_{t_1}(u, X) + w(v, u) \geq \pi_{t_1}(v, X)$.



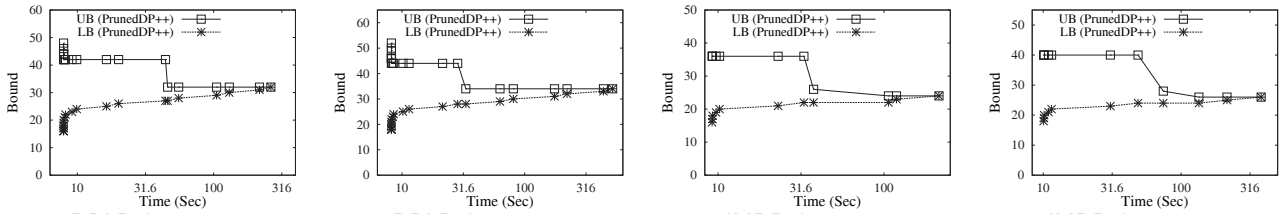
(a) knum $= 5$ (b) knum $= 6$ (c) knum $= 7$ (d) knum $= 8$

Figure 14: Query Processing Time vs. Approximation Ratio: Vary Number of Labels (knum) on LiveJournal



(a) knum $= 5$ (b) knum $= 6$ (c) knum $= 7$ (d) knum $= 8$

Figure 15: Query Processing Time vs. Approximation Ratio: Vary Number of Labels (knum) on RoadUSA



(a) DBLP (knum $= 9$) (b) DBLP (knum $= 10$) (c) IMDB (knum $= 9$) (d) IMDB (knum $= 10$)

Figure 16: Testing with Relatively Large knum for PrunedDP++ (kwf $= 400$)

Second, we show that the case (ii) also holds. Since $f_T^*(v, X') \geq f_{\bar{R}}^*(v, X')/2$, it is sufficient to show $f_{\bar{R}}^*(v, \bar{X}\backslash X') + f_{\bar{R}}^*(v, X') \geq f_{\bar{R}}^*(v, \bar{X})$. Let $v \sim R(\tilde{v}_i, \tilde{v}_j, \bar{X}\backslash X') \sim v$ and $v \sim R(\tilde{v}_i, \tilde{v}_j, X') \sim v$ be the optimal tours that starts from v, ends at v, and passes through all the virtual nodes in $V_{\bar{X}\backslash X'}$ and $V_{X'}$ respectively. We can merge these two tours and thus obtain a new tour that starts from v, ends at v, and passes through all the virtual nodes in $V_{\bar{X}}$. Since $f_{\bar{R}}^*(v, \bar{X})$ is the optimal weight over all such type of tours, we have $f_{\bar{R}}^*(v, \bar{X}\backslash X') + f_{\bar{R}}^*(v, X') \geq f_{\bar{R}}^*(v, \bar{X})$. Putting it all together, the lemma is established. □

LEMMA 7. *Let* $\pi(v, X) = \max\{\pi_1(v, X), \pi_{t_1}(v, X), \pi_{t_2}(v, X)\}$. *Then, for a state* (v, X)*, we have (i)* $\pi(u, X) + w(v, u) \geq \pi(v, X)$*, and (ii)* $\pi(v, X \cup X') + f_T^*(v, X') \geq \pi(v, X)$*, where* $X \cap X' = \emptyset$.

PROOF. First, since all the developed lower bounds are consistent (for $\pi_{t_2}(v, X)$, we can use the technique proposed in Section 4.2 to make it consistent), we can easily prove that the inequality $\pi(u, X) + w(v, u) \geq \pi(v, X)$ holds.

Second, we prove that $\pi(v, X \cup X') + f_T^*(v, X') \geq \pi(v, X)$ also holds. Below, we consider nine different cases.

Case (1): if $\pi(v, X \cup X') = \pi_1(v, X \cup X')$ and $\pi(v, X) = \pi_1(v, X)$, then the result clearly holds by Lemma 5.

Case (2): if $\pi(v, X \cup X') = \pi_1(v, X \cup X')$ and $\pi(v, X) = \pi_{t_1}(v, X)$, then we have $\pi_1(v, X \cup X') + f_T^*(v, X') \geq \pi_{t_1}(v, X \cup X') + f_T^*(v, X')$. Since $\pi_{t_1}(v, X)$ is consistent by Lemma 6, we have $\pi_{t_1}(v, X \cup X') + f_T^*(v, X') \geq \pi_{t_1}(v, X) = \pi(v, X)$.

Case (3): if $\pi(v, X \cup X') = \pi_1(v, X \cup X')$ and $\pi(v, X) = \pi_{t_2}(v, X)$, then we have $\pi_1(v, X \cup X') + f_T^*(v, X') \geq \pi_{t_2}(v, X \cup X') + f_T^*(v, X')$. Since $\pi_{t_2}(v, X)$ is consistent, we have $\pi_{t_2}(v, X \cup X') + f_T^*(v, X') \geq \pi_{t_2}(v, X) = \pi(v, X)$.

Similarly, for the other six cases, we can also prove that $\pi(v, X \cup X') + f_T^*(v, X') \geq \pi(v, X)$ holds. Putting it all together, the lemma is established. □

A.2 Additional experiments

Results on Different Graph Topologies. In this experiment, we test the performance of the proposed algorithms on graphs with different topologies. To this end, we use two additional publicly available datasets LiveJournal[3] and RoadUSA[4] to evaluate our algorithms. LiveJournal is a social network with a power-law degree distribution. It contains $4,847,572$ nodes and $42,851,237$ edges with a diameter 20. The average degree is 17.7 and the maximum degree is $20,333$. RoadUSA is the USA road network with a relatively uniform degree distribution. It contains $23,947,348$ nodes and $28,854,312$ edges with a diameter larger than $8,000$. The average degree is 2.4 and the maximum degree is 9. For each dataset, we generate labels for nodes randomly with the default label frequency. We vary knum from 5 to 8 and test the performance of the four algorithms Basic, PrunedDP, PrunedDP+, and PrunedDP++.

The results on the LiveJournal dataset are shown in Fig. 14. As can be seen, when knum increases, the processing time for all algorithms increases. When knum $= 8$ (Fig. 14 (d)), to reach an approximation ratio of 1, PrunedDP++ is two orders of magnitude faster than PrunedDP+ and PrunedDP, and three orders of magnitude faster than Basic. For example, Basic computes the optimal solution within $8,871.9$ seconds while PrunedDP++ only requires 8.7 seconds to calculate the optimal solution. As desired, PrunedDP+ is faster than PrunedDP. However, when the approximation ratio decreases to 1, the gap between PrunedDP+ and PrunedDP decreases. For example, for knum $= 7$ (Fig. 14 (c)), to achieve an approximation ratio of 2, PrunedDP+ only requires 16.2 seconds while PrunedDP needs 205.2 seconds. How-

[3]http://konect.uni-koblenz.de/networks/soc-LiveJournal1

[4]http://www.dis.uniroma1.it/challenge9/download.shtml

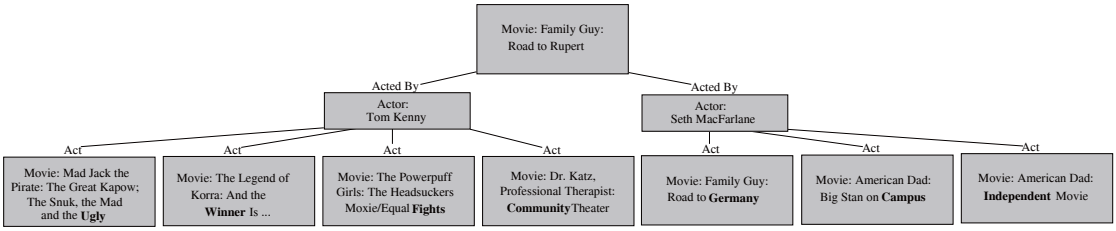


Figure 17: Case Study on IMDB using PrunedDP++ (knum $= 7$, Processing Time $= 40.9$ Seconds)

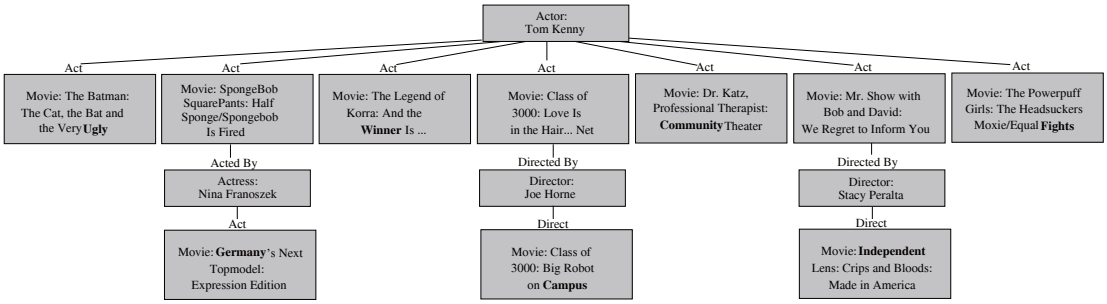


Figure 18: Case Study on IMDB using BANKS-II (knum $= 7$, Processing Time $= 252.7$ Seconds)

ever, to achieve an approximation ratio of 1, PrunedDP+ requires 169.7 seconds while PrunedDP needs 237.2 seconds.

The results on the RoadUSA dataset are shown in Fig. 15. Similar to the results on LiveJournal, PrunedDP++ is also the winner among all the algorithms on RoadUSA. However, different from the results on LiveJournal, the gap between PrunedDP++ and PrunedDP+ on RoadUSA is much smaller than that on LiveJournal. For example, when knum $= 6$, to reach an approximation of 1, PrunedDP++ is 48.8 times faster than PrunedDP+ in LiveJournal (Fig. 14 (b)), whereas PrunedDP++ is slightly faster than PrunedDP+ on RoadUSA (Fig. 15 (b)). This is because RoadUSA is a near planar graph, in which the difference between the one-label based lower bound and the tour-based lower bound is usually small. Instead, LiveJournal is a power-law graph, in which the one-label based lower bound is typically much smaller than the tour-based lower bound. These results indicate that the performance of PrunedDP++ is better on power-law graphs than that on near planar graphs.

Results on Relatively Large knum Values. In this experiment, we increase knum to 9 and 10 to test the scalability and progressive performance of the PrunedDP++ algorithm. Note that in many practical applications such as keyword search and team search problems, the number of given keywords is typically smaller than 10 [8, 6, 22, 30]. To the best of our knowledge, we are the first to handle knum > 6 to compute the exact GST on million-scale graphs. The experimental results on the DBLP dataset are shown in Fig. 16 (a) and Fig. 16 (b) for knum $= 9$ and knum $= 10$ respectively. As can be seen, when knum increases, the total processing time to compute the optimal answer increases. For example, PrunedDP++ requires 262.7 seconds and 520.6 seconds to compute the optimal answers for knum $= 9$ and knum $= 10$ respectively. Nevertheless, within 10 seconds, PrunedDP++ already generates answers with approximation ratios 1.31 and 1.29 for knum $= 9$ and knum $= 10$ respectively. The results on the IMDB dataset are similar to those on the DBLP dataset, which are shown in Fig. 16 (c) and Fig. 16 (d) for knum $= 9$ and knum $= 10$ respectively. As can be seen, when knum increases from 9 to 10, the processing time for PrunedDP++ increases from 214.7 seconds to 380.4 seconds. Likewise, within 37 seconds, PrunedDP++ generates an answer with approximation ratio1.083 for knum $= 9$, and within 72 seconds, PrunedDP++ generates an answer with approximation ratio 1.077 for knum $= 10$. These results indicate that the PrunedDP++ algorithm works well for relatively large knum values, and thus it is very useful for keyword search and team search related applications.

knum	kwf	Total Time BANKS-II	Appro Ratio BANKS-II	Total Time PrunedDP++	T_r PrunedDP++
5	400	211.3 secs	1.33	7.0 secs	6.9 secs
6	400	274.8 secs	1.22	18.7 secs	8.3 secs
7	400	332.1 secs	1.40	24.6 secs	18.0 secs
8	400	391.7 secs	1.45	112.4 secs	31.4 secs
6	200	275.4 secs	1.22	10.2 secs	10.1 secs
6	800	282.6 secs	1.14	20.3 secs	11.2 secs
6	1,600	266.0 secs	1.14	11.1 secs	9.8 secs

Table 3: Comparison with BANKS-II on IMDB

Additional Exp-5: Comparison with BANKS-II on IMDB. Here we compare PrunedDP++ and BANKS-II on the IMDB dataset. Table 3 reports the results on IMDB. Similar to the results on DBLP (Table 2), PrunedDP++ is 13.5 to 30.2 times faster than BANKS-II under all parameter settings on the IMDB dataset. For instance, under the default setting (knum $= 6$ and kwf $= 400$), BANKS-II computes an 1.22-approximate answer in 274.8 seconds while PrunedDP++ generates the optimal answer using only 18.7 seconds. When knum $= 8$, although PrunedDP++ takes 112.4 seconds to calculate the optimal answer, it only consumes 31.4 seconds to compute the answer with at least the same quality as the answer calculated by BANKS-II. These results further confirm the efficiency of the PrunedDP++ algorithm.

Additional Exp-6: Case Study. Here we report the case study on the IMDB dataset. In this case study, we use a seven keywords query $P = \{$ "Germany", "Campus", "winner", "independent", "fights", "ugly", "community" $\}$. With this query, the users may want to find some correlated actors/actresses who act movies with topics related to the keywords given in P. The answer reported by PrunedDP++ (taking 40.9 seconds) is shown in Fig. 17. This answer contains two actors who collaborate a movie and act movies with topics related to all the 7 keywords in P. Totally two actors and eight movies are included in this answer. The answer reported by BANKS-II (taking 252.7 seconds) is shown in Fig. 18. This answer contains an actor who acts seven movies directly/indirectly related to the seven keywords in P. Totally an actor, an actress, two directors, and ten movies are involved in this answer. Clearly, the answer reported by PrunedDP++ is more compact and may better meet the users' requirements than the answer reported by BANKS-II. These results further confirm the effectiveness of the proposed approach.