Finding Group Steiner Trees in Graphs with both Vertex and Edge Weights

Yahui Sun
School of Computing,
National University of Singapore
yahuisun@outlook.com

Xiaokui Xiao
School of Computing,
National University of Singapore
xkxiao@nus.edu.sg

Bin Cui
School of EECS & MOE,
Peking University
bin.cui@pku.edu.cn

Saman Halgamuge
School of Electrical, Mechanical and Infrastructure Engineering,
University of Melbourne
saman@unimelb.edu.au

Theodoros Lappas
School of Business,
Stevens Institute of Technology,
New Jersey, United States
tlappas@stevens.edu

Jun Luo
School of Computer Science and Engineering, Nanyang Technological University
junluo@ntu.edu.sg

ABSTRACT

Given an undirected graph and a number of vertex groups, the *group Steiner tree* problem is to find a tree such that (i) this tree contains at least one vertex in each vertex group; and (ii) the sum of vertex and edge weights in this tree is minimized. Solving this problem is useful in various scenarios, ranging from social networks to knowledge graphs. However, most existing work focuses on solving this problem in vertex-unweighted graphs, and rare work has been done to solve this problem in graphs with both vertex and edge weights. Here, we develop several algorithms to address this issue. Initially, we extend two algorithms from vertex-unweighted graphs to vertex- and edge-weighted graphs. The first one has no approximation guarantee, but often produces good solutions in practice. The second one has an approximation guarantee of $|\Gamma| - 1$, where $|\Gamma|$ is the number of vertex groups. Since the extended $(|\Gamma| - 1)$-approximation algorithm is too slow when all vertex groups are large, we develop two new $(|\Gamma| - 1)$-approximation algorithms that overcome this weakness. Furthermore, by employing a dynamic programming approach, we develop another $(|\Gamma| - h + 1)$-approximation algorithm, where h is a parameter between 2 and $|\Gamma|$. Experiments show that, while no algorithm is the best in all cases, our algorithms considerably outperform the state of the art in many scenarios.

PVLDB Reference Format:
Yahui Sun, Xiaokui Xiao, Bin Cui, Saman Halgamuge, Theodoros Lappas, and Jun Luo. Finding Group Steiner Trees in Graphs with both Vertex and Edge Weights. PVLDB, 14(6): XXX-XXX, 2021.
doi:10.14778/3447689.3447698

PVLDB Artifact Availability:
The C++ source codes, datasets, and the supplement have been made available at https://github.com/YahuiSun/GroupSteinerTree.

1 INTRODUCTION

Given an undirected graph G and a number of vertex groups, a *group Steiner tree* is a tree in G such that (i) this tree contains at least one vertex in each vertex group; and (ii) the total weight of vertices and edges in this tree is minimized. Finding group Steiner trees is an interesting problem that has applications in various scenarios, including team formation in social networks (*e.g.*, [25, 30, 39]), information retrieval in relational databases (*e.g.*, [11, 14, 27, 28]), design of very-large-scale integrated circuits (*e.g.*, [19, 22, 23, 33]), and pathway identification in metabolic networks (*e.g.*, [17]). Most existing work on finding group Steiner trees (*e.g.*, [19, 22, 23, 25, 28, 33]), however, focuses on the case where vertex weights are omitted, *i.e.*, only edge weights are considered when calculating the total weight of a tree. Nevertheless, there are useful applications that require finding group Steiner trees in graphs with both vertex and edge weights. We describe an example as follows.

Consider a social network where vertices and edges represent experts and collaborations between experts, respectively. Each vertex is associated with some skills and a weight value representing the hiring cost of expert. Each edge is associated with a weight value representing the distance between two experts. Suppose that we are to find a team of experts for performing a task that requires a set Γ of skills. For this purpose, we can find a group Steiner tree for $|\Gamma|$ vertex groups such that each vertex group is the set of vertices that are associated with a specific skill in Γ. Since this tree contains at least one vertex in each vertex group, the vertices in this tree represent a team of experts who collectively have all the required skills for performing the task. Since the total weight of vertices and edges in this tree is minimized, we can strike a good trade-off between the cost of hiring these experts and their closeness to each other, which could affect the efficiency of their collaboration [25, 30, 39]. We can adjust this trade-off by regulating vertex and edge weights, depending on whether we prefer lowering the hiring cost or reducing the distances between experts. We visualize an example in the DBLP [2] network in Figure 1.

Some exact algorithms [14, 28] have been developed for finding optimal group Steiner trees. The most advanced exact algorithms, *i.e.,* PrunedDP and PrunedDP++ in [28], rely on techniques that only hold in vertex-unweighted scenarios (details in the supplement [6]), and thus do not suit finding optimal group Steiner trees in graphs with both vertex and edge weights. The other simpler exact algorithms, *i.e.,* DPBF in [14] and Basic in [28], can perform this task. However, due to the NP-hardness of the group Steiner tree problem [23, 34], these algorithms have exponential time complexities with

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447698

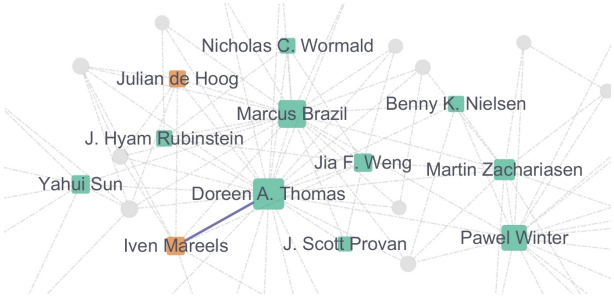


Figure 1: A snapshot of the DBLP [2] network, where green and orange experts have the skills of *Steiner trees* and *power grids*, respectively. The thick edge highlights a team of two experts who collectively have these two skills. By minimizing vertex and edge weights of experts, we can lower the cost of hiring experts, and reduce the distances between experts.

respect to the number of vertex groups: $|\Gamma|$. As a result, even though $|\Gamma|$ is often limited in practice (*e.g.,* in team formation scenarios where $|\Gamma|$ is the number of skills for performing a task), it is still too slow to use these algorithms to find optimal group Steiner trees in some cases. Therefore, it is also preferable to develop non-exact algorithms that find sub-optimal group Steiner trees.

Most existing non-exact algorithms ignore vertex weights (*e.g.*, [19, 23, 25, 33]). To the best of our knowledge, the algorithm in [8] is the only existing non-exact algorithm that considers vertex weights. It achieves an approximation guarantee of $O(\log|V|\log|\Gamma|)$, where $|V|$ is the number of vertices. However, it has a large time complexity of $|V|^{\tilde{O}(tw(G)^2)}$, where $\tilde{O}(x) = O(x \cdot \text{polylog}(x))$, and $tw(G)$ is the treewidth of the input graph G, which often ranges from dozens to hundreds for real graphs [31]. As a consequence, the algorithm in [8] is of theoretical interest only. Hence, there is a need for more practical algorithms for finding group Steiner trees in graphs with both vertex and edge weights.

To address the above issue, we make the following contributions.

- We extend a heuristic algorithm [25] from vertex-unweighted graphs to vertex- and edge-weighted graphs (Section 3). The extension, dubbed exENSteiner, has a time complexity of
$$O\Big(|\Gamma| \cdot (|E| + |V|\log|V| + |\Gamma|\log|\Gamma| + |\Gamma||V|)\Big),$$
where $|E|$ is the number of edges.
- Since exENSteiner has no approximation guarantee, we further extend a $(|\Gamma|-1)$-approximation algorithm [23]. The extension, dubbed exIhlerA (Section 4.1), has a time complexity of
$$O\Big(|g_{min}| \cdot (|E| + |V|\log|V| + |\Gamma||V|)\Big),$$
where $|g_{min}|$ is the size of the smallest vertex group.
- When $|g_{min}|$ is large, exIhlerA is too slow to be used. To address this issue, we propose a new $(|\Gamma|-1)$-approximation algorithm, dubbed FastAPP (Section 4.2), which has a time complexity of
$$O\Big(|\Gamma| \cdot (|E| + |V|\log|V|)\Big).$$
- FastAPP does not dominate the extensions on practical solution qualities. To attain this dominance while maintaining a high efficiency, we propose another $(|\Gamma|-1)$-approximation algorithm, dubbed ImprovAPP (Section 4.3). It has a time complexity of
$$O\Big(|\Gamma| \cdot \Big(|E| + |V|\log|V| + |g_{min}| \cdot (|V| + \log|\Gamma|)\Big)\Big),$$
which is close to $O(|\Gamma| \cdot (|E| + |V|\log|V|))$ in practice (we explain this in Section 7.3).
- In addition, by employing an exact algorithm for group Steiner trees, *i.e.,* DPBF in [14], we propose a $(|\Gamma|-h+1)$-approximation algorithm, dubbed PartialOPT (Section 5), where $h \in [2, |\Gamma|]$ is a tunable parameter. The time complexity of PartialOPT is
$$O\Big(|g_{min}| \cdot \Big(|\Gamma||V| + 3^h|V| + 2^h(|E| + h|V| + |V|\log|V|)\Big)\Big).$$
To our knowledge, it provides the tightest polynomial-time approximation guarantee to date for finding group Steiner trees in treewidth-unbounded graphs with vertex and edge weights.
- We evaluate our algorithms using real datasets (Section 7), and show that, while no algorithm is the best in all cases, our algorithms considerably outperform the state of the art in many scenarios. In particular, (i) our algorithms scale well to $|\Gamma|$, and thus support the exact algorithms [14, 28] that have exponential time complexities with respect to $|\Gamma|$; and (ii) ImprovAPP has a similar speed with FastAPP, and combines superior efficiency and solution quality when it is too slow to find optimal solutions.

2 PROBLEM FORMULATION

We consider an undirected graph $G(V, E, w, c)$, where V is the set of vertices, E is the set of edges, w is a function which maps each vertex $i \in V$ to a nonnegative value $w(i)$ that we refer to as vertex weight, and c is a function which maps each edge $e \in E$ to a non-negative value $c(e)$ that we refer to as edge weight. We also consider a set Γ of vertex groups such that each vertex group $g \in \Gamma$ is a subset of vertices, *i.e.*, $g \subseteq V$. Notably, vertex groups may overlap with each other. We aim to address the following problem.

Problem 1 (Vertex- and Edge-Weighted Group Steiner Tree [14]). *Given an undirected graph $G(V, E, w, c)$ and a set Γ of vertex groups, the vertex- and edge-weighted group Steiner tree problem asks for a tree $G'(V', E')$, $V' \subseteq V, E' \subseteq E$ such that (i) $g \cap V' \neq \emptyset$ for all $g \in \Gamma$ (i.e., this tree contains at least one vertex in each group in Γ), and (ii) the regulated weight of this tree, namely,*

$$c_\lambda(G') = (1-\lambda)\sum_{v \in V'} w(v) + \lambda \sum_{e \in E'} c(e) \quad (1)$$

is minimized, where $\lambda \in [0, 1]$ is a regulating weight.

We note that Problem 1 is NP-hard even when G is a tree and all vertices have zero weights [23, 34]. We refer to its special case where all vertices have zero weights as the *vertex-unweighted group Steiner tree* problem [33].

We assume that $|\Gamma| \geq 2$, since Problem 1 is trivial when $|\Gamma| = 1$. Moreover, without loss of generality, we assume that G is connected. If G is not connected, then we can solve Problem 1 in G as follows: first, we obtain a solution in each maximal connected component of G separately; and then, we evaluate all the obtained solutions, and return the one with the minimum regulated weight.

3 AN EXTENDED HEURISTIC ALGORITHM

The ENSteiner algorithm in [25] is a heuristic algorithm for finding vertex-unweighted group Steiner trees. It employs (i) a transformation [16] from group Steiner trees to Steiner trees in vertex-unweighted graphs; and (ii) a 2-approximation algorithm [38] for

finding vertex-unweighted Steiner trees. In this section, we extend ENSteiner for finding vertex- and edge-weighted group Steiner trees. First, in Sections 3.1 and 3.2, we extend the transformation and the 2-approximation algorithm, respectively. Then, in Section 3.3, we extend ENSteiner. Our extension is dubbed exENSteiner.

3.1 From group Steiner trees to Steiner trees

The *Steiner tree* problem in graphs with vertex and edge weights is a special case of Problem 1 where each vertex group contains exactly one vertex. We introduce this problem as follows.

Problem 2 (Vertex- and Edge-Weighted Steiner Tree [24]). *Given a connected undirected graph $G_t(V_t, E_t, w_t, c_t)$ and a set $T_t \subseteq V_t$ of vertices that we refer to as compulsory vertices, the vertex- and edge-weighted Steiner tree problem asks for a tree $G'_t(V'_t, E'_t)$, $V'_t \subseteq V_t, E'_t \subseteq E_t$ such that (i) $T_t \subseteq V'_t$ (i.e., all compulsory vertices are in this tree), and (ii) the weight of this tree, namely,*

$$c(G'_t) = \sum_{v \in V'_t} w_t(v) + \sum_{e \in E'_t} c_t(e) \tag{2}$$

is minimized.

We refer to the special case of Problem 2 where all vertex weights are zero as the *vertex-unweighted Steiner tree* problem [15]. We observe that any instance of Problem 1 can be transformed to an equivalent instance of Problem 2, as shown as follows. We put the proof of this transformation in the supplement [6].

Theorem 1. *Let $G(V, E, w, c)$ be a connected undirected graph, and Γ be a set of vertex groups. Let $G_t(V_t, E_t, w_t, c_t)$ be a connected undirected graph, and $T_t \subseteq V_t$ be a set of compulsory vertices. Based on G and Γ, we construct G_t and T_t in the following way:*

(1) Initialize $V_t = V$, $E_t = E$, $T_t = \emptyset$, $w_t = (1-\lambda)w$, and $c_t = \lambda c$.

(2) For each vertex group $g \in \Gamma$, (i) add a dummy vertex v_g into T_t and V_t, such that $w_t(v_g) = 0$, and (ii) add dummy edges (v_g, j) for all $j \in g$ into E_t, such that $c_t(v_g, j) = M$, and M is a constant satisfying

$$M > (1-\lambda) \sum_{v \in V} w(v) + \lambda \sum_{e \in E_{MST}} c(e), \tag{3}$$

and E_{MST} is the set of edges in a Minimum Spanning Tree of G.

Let Θ_{G_t} be an optimal solution to the vertex- and edge-weighted Steiner tree problem in G_t, and $\Theta^{non}_{G_t}$ be the non-dummy part of Θ_{G_t}. Then, there is an optimal solution to the vertex- and edge-weighted group Steiner tree problem in G, namely, Θ_G, that has the same sets of vertices and edges with $\Theta^{non}_{G_t}$.

This transformation is modified from the existing method [16] of transforming the vertex-unweighted group Steiner tree problem to the vertex-unweighted Steiner tree problem. The difference is that we incorporate vertex weights into this transformation. Moreover, the value of M in the existing method is unspecified and vaguely described as a large value. Due to this un-rigorousness, it may be difficult to decide the usefulness of this transformation in practice, *e.g.*, if G contains large weights, then it is difficult to decide whether there is a feasible M that guarantees the correctness of this transformation. We address this issue by specifying the bound of M. In practice, we can set M as $1 + (1-\lambda)\sum_{v \in V} w(v) + \lambda \sum_{e \in E} c(e)$ by traversing all vertices and edges in $O(|V| + |E|)$ time.

Example. In Figure 2, G contains five vertices $v_1, \ldots, v_5$, whose weights are $\{w(v_1), \ldots, w(v_5)\} = \{2, 1, 1, 1, 1\}$. Given the three vertex groups, we transform G to G_t, which contains three dummy vertices: $T_t = \{v_{d1}, v_{d2}, v_{d3}\}$. The transformed vertex weights are: $\{w_t(v_1), \ldots, w_t(v_5), w_t(v_{d1}), w_t(v_{d2}), w_t(v_{d3})\} = (1-\lambda) \cdot \{2, 1, 1, 1, 1, 0, 0, 0\}$. In addition, we have $M > 6$. The red thick edges in the figure highlight (i) the optimal solution to the vertex- and edge-weighted group Steiner tree problem in G, *i.e.*, Θ_G, and (ii) the optimal solution to the vertex- and edge-weighted Steiner tree problem in G_t, *i.e.*, Θ_{G_t}.

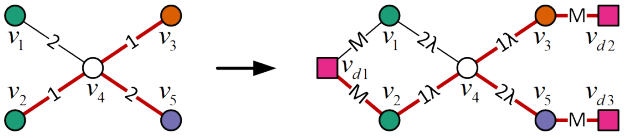


Figure 2: An example of the transformation in Theorem 1, where the graph on the left (*resp.*, right) is G (*resp.*, G_t). There are three vertex groups: $g_1 = \{v_1, v_2\}$, $g_2 = \{v_3\}$ and $g_3 = \{v_5\}$.

3.2 The LANCET algorithm

The 2-approximation algorithm in [38] is designed for finding vertex-unweighted Steiner trees. Here, we extend it to find vertex- and edge-weighted Steiner trees. Our extension is dubbed LANCET, *i.e.*, the lowest weight path concatenating algorithm.

LANCET uses the concept of *lowest weight paths (LWPs)*, which is defined as follows. An LWP between two vertices t_1 and t_2 in a graph $G(V, E, w, c)$ is a simple path that (i) has t_1 and t_2 as its two endpoints and (ii) minimizes the total (not λ-regulated) weight of vertices and edges in this path. We can find the LWPs from a vertex $s \in V$ to the other vertices in G as follows. First, for each edge $(u, v) \in E$, we redefine its weight as: $c'(u, v) = c(u, v) + w(u)/2 + w(v)/2$. After that, we employ Dijkstra's algorithm [13] to find the shortest paths from s to the other vertices for the above new edge weights. These shortest paths are the LWPs from s to the other vertices for the original vertex and edge weights.

Description of LANCET**.** Algorithm 1 shows the pseudo code of LANCET. The algorithm randomly selects a compulsory vertex $i_{rand} \in T_t$ (Line 1), and initializes the following (Line 2):

- The set of connected vertices: $V_1 = \{i_{rand}\}$.
- The set of unconnected compulsory vertices: $V_2 = T_t \setminus i_{rand}$.
- An empty tree $\Theta(V_\Theta, E_\Theta) = \emptyset$.
- An empty min Fibonacci heap (priority queue) [18] $Q = \emptyset$.

Then, it concatenates the LWPs between vertices in V_1 and V_2 as follows (Lines 3-11). First, for every vertex $i \in V_2$, the algorithm finds the LWPs from i to all vertices in V_t, and stores these LWPs in a lookup table (Line 3). Then, the algorithm pushes into Q the LWPs from the vertices in V_2 to $V_1 = \{i_{rand}\}$, with the weights of these LWPs as priorities (Line 4). Subsequently, the algorithm iteratively pops out the top entry $LWP_{min}(V_{min}, E_{min})$ in Q (Lines 5-6), and processes it in four steps (Lines 7-10):

(1) Merge $LWP_{min}(V_{min}, E_{min})$ into Θ.
(2) Update the set of connected vertices: V_1.
(3) Update the set of unconnected compulsory vertices: V_2.
(4) For each vertex $i \in V_2$ and each newly connected vertex j, identify the LWP between i and j, i.e., $LWP_{i \to j}$, from the lookup table. Let $LWP_{i \to *}$ be the LWP in Q that starts from i. If $LWP_{i \to j}$ has a smaller weight than $LWP_{i \to *}$, then replace $LWP_{i \to *}$ with $LWP_{i \to j}$ in Q (Line 10). In other words, for each vertex $i \in V_2$, Q keeps track of the minimum-weight LWP from i to V_1.

Algorithm 1 The LANCET algorithm

Input: a graph $G_t(V_t, E_t, w_t, c_t)$, a set of vertices $T_t \subseteq V_t$
Output: a Steiner tree: $\Theta(V_\Theta, E_\Theta)$

1: Randomize $i_{rand} \in T_t$
2: Initialize $V_1 = \{i_{rand}\}$, $V_2 = T_t \setminus i_{rand}$, $\Theta = Q = \emptyset$
3: Find and store $LWP_{i \to j} \mid \forall i \in V_2, \forall j \in V_t$
4: $enqueue(Q, LWP_{i \to i_{rand}}) \mid \forall i \in V_2$
5: **while** $V_2 \neq \emptyset$ **do**
6: $\quad dequeue(Q, LWP_{min}(V_{min}, E_{min}))$
7: $\quad \Theta = \Theta \cup LWP_{min}(V_{min}, E_{min})$
8: $\quad V_1 = V_1 \cup V_{min}$
9: $\quad V_2 = V_2 \setminus V_{min}$
10: $\quad update(Q, LWP_{i \to V_1}) \mid \forall i \in V_2$
11: **end while**
12: Return $\Theta(V_\Theta, E_\Theta)$

LANCET iterates this process until all the compulsory vertices are connected. In the end, it returns Θ (Line 12).

Example of LANCET. Consider G_t in Figure 2 (details in Section 3.1). Suppose that LANCET initializes $V_1 = \{v_{d1}\}$ and $V_2 = \{v_{d2}, v_{d3}\}$. Then, it pushes into Q two paths as follows: $LWP_{v_{d2} \to v_{d1}}$ as $\{v_{d2} \to v_3 \to v_4 \to v_2 \to v_{d1}\}$, and $LWP_{v_{d3} \to v_{d1}}$ as $\{v_{d3} \to v_5 \to v_4 \to v_2 \to v_{d1}\}$. It concatenates $LWP_{v_{d2} \to v_{d1}}$. After this concatenation, $V_1 = \{v_{d2}, v_3, v_4, v_2, v_{d1}\}$ and $V_2 = \{v_{d3}\}$. Subsequently, it updates and concatenates $LWP_{v_{d3} \to V_1}$ as $\{v_{d3} \to v_5 \to v_4\}$. After that, $V_2 = \emptyset$. LANCET returns the highlighted red tree in G_t.

Approximation guarantee of LANCET**.** LANCET is an extension of the 2-approximation algorithm in [38]. The main difference is that the 2-approximation algorithm concatenates shortest paths for finding vertex-unweighted Steiner trees, while LANCET concatenates lowest weight paths for finding vertex- and edge-weighted Steiner trees. Another difference is that the ratio of 2 does not hold for LANCET. We show the approximation guarantee of LANCET as follows. We put the proof of this guarantee in the supplement [6].

Theorem 2. LANCET *has a sharp approximation guarantee of* $|T_t| - 1$ *for solving Problem 2.*

Time complexity of LANCET**:**

$$O\Big(|T_t| \cdot (|E_t| + |V_t| \log |V_t|)\Big).$$

The details of this time complexity are in the supplement [6].

3.3 The exENSteiner algorithm

Here, we present exENSteiner, which is an extension of ENSteiner in [25]. The difference is that we replace the existing transformation [16] and the 2-approximation algorithm [38] in ENSteiner with our extended transformation and LANCET, respectively.

Description of exENSteiner**.** Algorithm 2 shows the pseudo code of exENSteiner. The algorithm first transforms G and Γ to G_t and T_t based on Theorem 1 (Line 1). Then, it employs LANCET to find a Steiner tree in G_t: $\Theta_t(V_{\Theta_t}, E_{\Theta_t})$ (Line 2). Recall that LANCET initially considers a random dummy vertex as connected, and then iteratively concatenates the minimum-weight LWPs between connected vertices and unconnected dummy vertices. Suppose that the first concatenated LWP connects two dummy vertices v_{d1} and

Algorithm 2 The exENSteiner algorithm

Input: a graph $G(V, E, w, c)$, a set of vertex groups Γ, a regulating weight λ
Output: a group Steiner tree: $\Theta(V_\Theta, E_\Theta)$

1: $G(V, E, w, c)$ & $\Gamma \to G_t(V_t, E_t, w_t, c_t)$ & T_t (Theorem 1)
2: $\Theta_t(V_{\Theta_t}, E_{\Theta_t}) = LANCET(G_t, T_t)$
3: $\Theta(V_\Theta, E_\Theta) = \Theta_t \setminus \cup_{g \in \Gamma} v_g$
4: $\Theta = MST(\Theta)$

v_{d2}. Since (i) dummy vertices only connect non-dummy vertices via dummy edges; and (ii) the weight of each dummy edge is M, which is larger than the (λ-regulated) total weight of a Minimum Spanning Tree (MST) of G (see Equation (3)), the first concatenated LWP contains two dummy edges, and v_{d1} and v_{d2} are leaves of this LWP, and the weight of this LWP is smaller than $3M$, *i.e.,* any path that contains more than two dummy edges has a larger weight. Moreover, this LWP connects at least one non-dummy vertex. As a result, each of the later concatenated LWPs contains only one dummy edge and one (newly connected) dummy vertex, which is a leaf, since any path that contains more than one dummy edge would have a larger weight. Thus, all dummy vertices are leaves of the Steiner tree found by LANCET. Based on this fact, exENSteiner removes dummy vertices and edges from Θ_t, and produces a feasible solution to the group Steiner tree problem in G: Θ (Line 3). It returns the MST that spans the vertices in Θ (Line 4).

Example of exENSteiner. Consider the example in Section 3.1 (*i.e.,* Figure 2). First, exENSteiner transforms G and Γ to G_t and T_t. Then, it employs LANCET to find the highlighted red tree in G_t as Θ_t (details in Section 3.2). It removes dummy vertices and edges from Θ_t, and gets the highlighted red tree in G as Θ. It returns this tree.

Time complexity of exENSteiner**:**

$$O\Big(|\Gamma| \cdot (|E| + |V| \log |V| + |\Gamma| \log |\Gamma| + |\Gamma||V|)\Big).$$

The details of this time complexity are also in the supplement [6].

4 THREE ($|\Gamma| - 1$)-APPROXIMATION ALGORITHMS

The above exENSteiner has no approximation guarantee for finding group Steiner trees. To address this issue, here, we develop three ($|\Gamma| - 1$)-approximation algorithms. First, in Section 4.1, we extend an existing algorithm that is for vertex-unweighted graphs. Our extension is dubbed exIhlerA. Since exIhlerA is too slow when $|g_{min}|$ is large, we propose a fast algorithm, dubbed FastAPP, in Section 4.2. FastAPP does not dominate exENSteiner or exIhlerA on practical solution qualities. Thus, in Section 4.3, we propose an improved algorithm, dubbed ImprovAPP, that dominates the above algorithms on practical solution qualities, and scales well in practice.

4.1 An extended ($|\Gamma| - 1$)-approximation algorithm

Here, we extend the ($|\Gamma| - 1$)-approximation algorithm in [23], which we refer to as IhlerA. Our extension is dubbed exIhlerA.

Description of exIhlerA**.** Algorithm 3 shows the pseudo code of exIhlerA. It initializes an empty graph G_{min}, and sets $c_\lambda(G_{min}) = \infty$

Algorithm 3 The exIhlerA algorithm

Input: a graph $G(V, E, w, c)$, a set of vertex groups Γ, a regulating weight λ

Output: a group Steiner tree: $\Theta(V_\Theta, E_\Theta)$

1: Initialize $G_{min} = \emptyset$; $c_\lambda(G_{min}) = \infty$
2: Find the smallest group g_{min} in Γ
3: **for** each vertex $i \in g_{min}$ **do**
4: Find $LWP_{\lambda ig} \mid \forall g \in \Gamma \setminus g_{min}$
5: $G_i = \cup_{g \in \Gamma \setminus g_{min}} LWP_{\lambda ig}$
6: $G_{min} = \min_{c_\lambda}\{G_{min}, G_i\}$
7: **end for**
8: Return $\Theta = MST(G_{min})$

(Line 1). Then, it identifies the smallest group g_{min} in Γ (Line 2), and processes every vertex $i \in g_{min}$ as follows (Lines 4-6):

(1) For every vertex group $g \in \Gamma \setminus g_{min}$, find the regulated lowest weight path between i and g: $LWP_{\lambda ig}(V_{\lambda ig}, E_{\lambda ig})$, *i.e.,* the simple path that contains i and at least one vertex in g, and the regulated weight of this path, namely,

$$c_\lambda(LWP_{\lambda ig}) = (1-\lambda)\sum_{v \in V_{\lambda ig}} w(v) + \lambda \sum_{e \in E_{\lambda ig}} c(e), \quad (4)$$

is minimized. Such paths can be found by invoking Dijkstra's algorithm to find non-regulated lowest weight paths (see Section 3.2) in $G''(V, E, (1-\lambda)w, \lambda c)$.

(2) Combine $LWP_{\lambda ig}$ for every $g \in \Gamma \setminus g_{min}$ to form a graph G_i.

(3) If the regulated weight of G_{min} is larger than that of G_i, then update G_{min} to G_i.

After processing all the vertices in g_{min}, the algorithm returns the MST that spans the vertices in G_{min} (Line 8).

Example of exIhlerA. We use G in Figure 2 as an example (details in Section 3.1). Suppose that exIhlerA selects $g_2 = \{v_3\}$ as g_{min}. Then, $\Gamma \setminus g_{min}$ contains g_1 and g_3. $LWP_{\lambda v_3 g_1}$ is the path $\{v_3 \rightarrow v_4 \rightarrow v_2\}$, and $LWP_{\lambda v_3 g_3}$ is the path $\{v_3 \rightarrow v_4 \rightarrow v_5\}$. exIhlerA combines these two paths as G_{v_3}, which is the highlighted red tree in G. exIhlerA updates G_{min} to be this tree. It returns this tree.

Approximation guarantee of exIhlerA**.** The difference between IhlerA [23] and exIhlerA is that IhlerA combines shortest paths in Line 5 for finding vertex-unweighted group Steiner trees, while exIhlerA combines LWPs for finding vertex- and edge-weighted group Steiner trees. Like IhlerA, exIhlerA has a guarantee of $|\Gamma| - 1$. The proof is in the supplement [6].

Theorem 3. exIhlerA *has a sharp approximation guarantee of* $|\Gamma| - 1$ *for solving Problem 1.*

Time complexity of exIhlerA**:**

$$O\Big(|g_{min}| \cdot (|E| + |V| \log |V| + |\Gamma||V|)\Big).$$

First, it initializes G_{min} in $O(1)$ time. It identifies g_{min} (Line 2) at a cost of $O(|\Gamma|)$. For each $i \in g_{min}$, it finds the regulated lowest weight path between i and each vertex group in $\Gamma \setminus g_{min}$ (Line 4) in $O(|E| + |V| \log |V| + |\Gamma||V|)$ time (by first using Dijkstra's algorithm to find the regulated LWPs from i to all vertices, and then evaluating the regulated LWPs from i to each vertex in each vertex group in $\Gamma \setminus g_{min}$). It combines LWPs as G_i (Line 5) in $O(|\Gamma||V|)$ time. It updates G_{min} (Line 6) at a cost of $O(|V| + |E|)$. After the loop, it derives the MST (Line 8) in $O(|E| + |V| \log |V|)$ time [32].

Algorithm 4 The FastAPP algorithm

Input: a graph $G(V, E, w, c)$, a set of vertex groups Γ, a regulating weight λ

Output: a group Steiner tree: $\Theta(V_\Theta, E_\Theta)$

1: Find the smallest group g_{min} in Γ
2: Find and store $LWP_{\lambda ig} \mid \forall i \in V, g \in \Gamma \setminus g_{min}$
3: Initialize $i_{min} = \emptyset$; $cost(i_{min}) = \infty$
4: **for** each vertex $i \in g_{min}$ **do**
5: **if** $cost(i_{min}) > \max\{c_\lambda(LWP_{\lambda ig}) \mid \forall g \in \Gamma \setminus g_{min}\}$ **then**
6: $i_{min} = i$
7: $cost(i_{min}) = \max\{c_\lambda(LWP_{\lambda ig}) \mid \forall g \in \Gamma \setminus g_{min}\}$
8: **end if**
9: **end for**
10: $G_{min} = \cup_{g \in \Gamma \setminus g_{min}} LWP_{\lambda i_{min} g}$
11: Return $\Theta = MST(G_{min})$

4.2 A fast ($|\Gamma| - 1$)-approximation algorithm

The above exIhlerA is too slow to be implemented when $|g_{min}|$ is large. To address this issue, here, we propose FastAPP, which uses a different approximation approach from exIhlerA.

First, recall that the regulated lowest weight path between a vertex i and a vertex group g, namely, $LWP_{\lambda ig}(V_{\lambda ig}, E_{\lambda ig})$, is a simple path that contains i and at least one vertex in g, and the regulated weight of this path (see Equation (4)) is minimized. We observe that the regulated lowest weight paths between g and every vertex can be found in $O(|E| + |V| \log |V|)$ time via the following two steps. First, add a dummy vertex v_g into V, such that $w(v_g) = 0$, and add dummy edges (v_g, j) for all $j \in g$ into E, such that $c(v_g, j) = 0$. Second, use Dijkstra's algorithm to find the non-regulated lowest weight paths (see Section 3.2) between v_g and the other vertices in $G''(V, E, (1-\lambda)w, \lambda c)$. These paths correspond to the regulated lowest weight paths between g and every vertex in G.

Unlike exIhlerA that employs Dijkstra's algorithm $|g_{min}|$ times, we can employ Dijkstra's algorithm $|\Gamma| - 1$ times to find $LWP_{\lambda ig}$ between every $i \in V$ and every $g \in \Gamma \setminus g_{min}$ for achieving the guarantee of $|\Gamma| - 1$. However, this change is not enough for completely removing $|g_{min}|$ from the time complexity of exIhlerA, due to the cost of $O(|g_{min}| \cdot |\Gamma||V|)$ for building G_i in Line 5 of exIhlerA. FastAPP completely removes $|g_{min}|$ from its time complexity by using a different approximation approach from exIhlerA and the previous work [23]. In particular, FastAPP does not build G_i when enumerating $i \in g_{min}$. Instead, it finds $i \in g_{min}$ that minimizes the maximum regulated weight of the regulated lowest weight paths between i and each vertex group in $\Gamma \setminus g_{min}$.

Description of FastAPP. Algorithm 4 shows the pseudo code of FastAPP. First, it finds g_{min} in Γ (Line 1). Then, it finds and stores $LWP_{\lambda ig}$ (as well as $c_\lambda(LWP_{\lambda ig})$) between every vertex $i \in V$ and every vertex group $g \in \Gamma \setminus g_{min}$ (Line 2). It initializes a vertex i_{min}, and considers the cost of i_{min} as infinity (Line 3). It processes every vertex $i \in g_{min}$ as follows (Lines 5-8). If the cost of i_{min} is larger than the maximum regulated weight of $LWP_{\lambda ig}$ for all $g \in \Gamma \setminus g_{min}$ (Line 5), then it updates i_{min} to i (Line 6), and updates the cost of i_{min} to this maximum weight (Line 7). After the loop, it combines $LWP_{\lambda i_{min} g}$ for every $g \in \Gamma \setminus g_{min}$ to form a graph G_{min} (Line 10). It returns the MST that spans the vertices in G_{min} (Line 11).

Example of FastAPP. Consider G in Figure 2 (details in Section 3.1). Suppose that FastAPP selects g_2 as g_{min}. It processes $v_3 \in g_{min}$ as follows. $LWP_{\lambda v_3 g_1}$ is the path $\{v_3 \rightarrow v_4 \rightarrow v_2\}$, and $LWP_{\lambda v_3 g_3}$ is the path $\{v_3 \rightarrow v_4 \rightarrow v_5\}$. Suppose that $\lambda \neq 0$. Then, it calculates $\max\{c_\lambda(LWP_{\lambda v_3 g}) \mid \forall g \in \Gamma \setminus g_{min}\}$ as $c_\lambda(LWP_{\lambda v_3 g_3})$, and updates i_{min} to v_3. It builds G_{min} by merging $LWP_{\lambda v_3 g_1}$ and $LWP_{\lambda v_3 g_3}$, which induces the highlighted red tree in G. It returns this tree.

Approximation guarantee of FastAPP**.** Like exIhlerA, FastAPP has a guarantee of $|\Gamma| - 1$. The proof is in the supplement [6].

Theorem 4. FastAPP *has a sharp approximation guarantee of* $|\Gamma| - 1$ *for solving Problem 1.*

Time complexity of FastAPP**:**

$$O\Big(|\Gamma| \cdot (|E| + |V| \log |V|)\Big).$$

First, the algorithm finds g_{min} (Line 1) at a cost of $O(|\Gamma|)$. Then, it finds and stores the regulated lowest weight paths (Line 2) in $O(|\Gamma|(|E| + |V| \log |V|))$ time. It initialize i_{min} (Line 3) in $O(1)$ time. Subsequently, it conducts a loop with $|g_{min}|$ iterations (Line 4). In each iteration, it finds $\max\{c_\lambda(LWP_{\lambda ig}) \mid \forall g \in \Gamma \setminus g_{min}\}$ from the pre-stored information in Line 2 in $O(|\Gamma|)$ time. As a result, the time complexity of updating i_{min} (Lines 5-8) is $O(|\Gamma|)$. Since $|g_{min}| \leq |V|$, it conducts the above loop in $O(|\Gamma||V|)$ time. After the loop, it builds G_{min} (Line 10) in $O(|\Gamma||V|)$ time. Then, it finds and returns an MST (Line 11) in $O(|E| + |V| \log |V|)$ time.

4.3 An improved $(|\Gamma| - 1)$-approximation algorithm

The above FastAPP does not dominate the extended algorithms on practical solution qualities. Here, we develop ImprovAPP, which dominates the above algorithms on practical solution qualities, while having a high efficiency in practice. The main difference between ImprovAPP and the above two $(|\Gamma| - 1)$-approximation algorithms is that, when processing each $i \in g_{min}$, ImprovAPP constructs a feasible solution tree by concatenating lowest weight paths in a similar way with LANCET.

Description of ImprovAPP. Algorithm 5 shows the pseudo code of ImprovAPP. First, it finds g_{min} in Γ (Line 1). Then, it finds and stores $LWP_{\lambda ig}$ between every $i \in V$ and every $g \in \Gamma \setminus g_{min}$ (Line 2). It initializes an empty tree Θ_{min}, and considers $c_\lambda(\Theta_{min}) = \infty$ (Line 3). It processes every $i \in g_{min}$ as follows (Lines 5-14).

It produces a feasible solution tree through a concatenation process similar to LANCET (Lines 5-13). In particular, it initializes (Line 5) the set of connected vertices: $V_c = \{i\}$; the set of unconnected vertex groups: $\Gamma_{uc} = \Gamma \setminus g_{min}$; an empty tree Θ_i; and an empty min Fibonacci heap Q. It pushes into Q the regulated lowest weight paths between each unconnected vertex group and i, with the regulated weights of these paths as priorities (Line 6). It iteratively pops out the top entry $LWP_{\lambda v_{top} g_{top}}$ in Q (Lines 7-8), and processes $LWP_{\lambda v_{top} g_{top}}$ as follows (Lines 9-12). It merges $LWP_{\lambda v_{top} g_{top}}$ into Θ_i (Line 9), and updates V_c and Γ_{uc} (Lines 10-11). For every newly connected vertex j and every unconnected vertex group $g \in \Gamma_{uc}$, it identifies $LWP_{\lambda jg}$. Let LWP_{*g} be the path in Q that connects g. If the regulated weight of $LWP_{\lambda jg}$ is smaller than that of LWP_{*g}, then it updates LWP_{*g} to $LWP_{\lambda jg}$ in Q (Line 12). It iterates this process until Γ_{uc} is empty. Then, Θ_i becomes a feasible solution tree. It uses Θ_i to update Θ_{min} (Line 14). After enumerating all vertices in g_{min}, Θ_{min} becomes a feasible solution tree. It is not guaranteed that Θ_{min} is an MST that spans the vertices in Θ_{min} (we provide an example in the supplement [6]). Thus, it updates Θ_{min} to be an MST that spans the vertices in Θ_{min} (Line 16). It proceeds to refine Θ_{min} (Lines 17-29) as follows.

The refinement of Θ_{min} is based on the concept of *unique-group leaves*. To explain, observe that in an optimal solution, every leaf is contained by one or more groups, one of which must be *unique* in the sense that this leaf does not share this group with any other vertex in the optimal solution. Otherwise, we could remove this leaf from the optimal solution to obtain another feasible solution with a smaller weight. In a solution, we refer to a leaf with at least one unique group as a *unique-group leaf* of this solution. Meanwhile, it is not guaranteed that all leaves of Θ_{min} are such unique-group leaves (we provide an example in the supplement [6]). Motivated by this, ImprovAPP removes non-unique-group leaves from Θ_{min} as follows (Lines 17-29). First, it initializes a max Fibonacci heap [18] $Q_{max} = \emptyset$ (Line 17). For each non-unique-group leaf v of Θ_{min}, it pushes v into Q_{max} with a priority of $(1 - \lambda)w(v) + \lambda c(v, v_{adj})$ (Line 19), where

Algorithm 5 The ImprovAPP algorithm

Input: a graph $G(V, E, w, c)$, a set of vertex groups Γ, a regulating weight λ
Output: a group Steiner tree: $\Theta(V_\Theta, E_\Theta)$

1: Find the smallest group g_{min} in Γ
2: Find and store $LWP_{\lambda ig} \mid \forall i \in V, g \in \Gamma \setminus g_{min}$
3: Initialize $\Theta_{min} = \emptyset$; $c_\lambda(\Theta_{min}) = \infty$
4: **for** each vertex $i \in g_{min}$ **do**
5: Initialize $V_c = \{i\}$, $\Gamma_{uc} = \Gamma \setminus g_{min}$, $\Theta_i = Q = \emptyset$
6: $enqueue(Q, LWP_{\lambda ig}) \mid \forall g \in \Gamma_{uc}$
7: **while** $\Gamma_{uc} \neq \emptyset$ **do**
8: $dequeue(Q, LWP_{\lambda v_{top} g_{top}}(V_{v_{top} g_{top}}, E_{v_{top} g_{top}}))$
9: $\Theta_i = \Theta_i \cup LWP_{\lambda v_{top} g_{top}}(V_{v_{top} g_{top}}, E_{v_{top} g_{top}})$
10: $V_c = V_c \cup V_{v_{top} g_{top}}$
11: $\Gamma_{uc} = \Gamma_{uc} \setminus g_{top}$
12: $update(Q, LWP_{*g}) \mid \forall g \in \Gamma_{uc}$
13: **end while**
14: $\Theta_{min} = \min_{c_\lambda}\{\Theta_{min}, \Theta_i\}$
15: **end for**
16: $\Theta_{min} = MST(\Theta_{min})$
17: Initialize $Q_{max} = \emptyset$
18: **for** each non-unique-group leaf v of Θ_{min} **do**
19: $enqueue(Q_{max}, v)$
20: **end for**
21: **while** $Q_{max} \neq \emptyset$ **do**
22: $dequeue(Q_{max}, v_{top})$
23: **if** v_{top} is a non-unique-group leaf **then**
24: $\Theta_{min} = \Theta_{min} \setminus (v_{top}, v_{topadj})$
25: **if** v_{topadj} is a non-unique-group leaf **then**
26: $enqueue(Q_{max}, v_{topadj})$
27: **end if**
28: **end if**
29: **end while**
30: Return $\Theta = \Theta_{min}$

v_{adj} is the adjacent vertex of v in Θ_{min}. Then, it iteratively pops the top entry v_{top} from Q_{max} (Lines 21-22), and checks whether v_{top} is a non-unique-group leaf (because a non-unique-group leaf in Q_{max} could become a unique-group leaf after other leaves are removed from Θ_{min} in preceding iterations). If v_{top} is a non-unique-group leaf (Line 23), it removes the edge (v_{top}, v_{topadj}) from Θ_{min} (Line 24), where v_{topadj} is the adjacent vertex of v_{top} in Θ_{min}. After this removal, v_{topadj} may become a non-unique-group leaf. In this case, ImprovAPP pushes v_{topadj} into Q_{max} (Line 26). This iteration process ends when Q_{max} becomes empty. After this refinement, it returns Θ_{min} (Line 30).

Example of ImprovAPP. Consider G in Figure 2 (details in Section 3.1). If ImprovAPP selects g_2 as g_{min}, then it processes $v_3 \in g_{min}$ as follows. It initializes Θ_{v_3} as empty. $LWP_{\lambda v_3 g_1}$ is the path $\{v_3 \rightarrow v_4 \rightarrow v_2\}$, and $LWP_{\lambda v_3 g_3}$ is the path $\{v_3 \rightarrow v_4 \rightarrow v_5\}$. It pushes these two paths into Q. It merges $LWP_{\lambda v_3 g_1}$ into Θ_{v_3}. Then, it updates LWP_{*g_3} in Q as the path $\{v_4 \rightarrow v_5\}$, and merges this path into Θ_{v_3}. The resulting Θ_{v_3} is the highlighted red tree in G. It updates Θ_{min} to be this tree. It returns this tree.

Approximation guarantee of ImprovAPP**.** ImprovAPP keeps the guarantee of $|\Gamma| - 1$. We put the proof in the supplement [6].

Theorem 5. ImprovAPP *has a sharp approximation guarantee of* $|\Gamma| - 1$ *for solving Problem 1.*

Time complexity of ImprovAPP**:**

$$O\Big(|\Gamma| \cdot \Big(|E| + |V| \log |V| + |g_{min}| \cdot (|V| + \log |\Gamma|)\Big)\Big).$$

First, it finds g_{min} (Line 1) at a cost of $O(|\Gamma|)$. Then, it finds $LWP_{\lambda ig}$ between every $i \in V$ and every $g \in \Gamma \setminus g_{min}$ (Line 2) in $O(|\Gamma|(|E| + |V| \log |V|))$ time. It initialize Θ_{min} (Line 3) in $O(1)$ time. It conducts a for loop with $|g_{min}|$ iterations (Line 4). In each iteration, it does the initialization (Line 5) in $O(|\Gamma|)$ time. It takes $O(|\Gamma|)$ time to push the lowest weight paths into Q (Line 6). Then, it concatenates lowest weight paths using a while loop with $O(|\Gamma|)$ iterations (Lines 7-13) as follows. It pops out $LWP_{\lambda v_{top} g_{top}}$ in $O(\log |\Gamma|)$ time (Line 8). It merges $LWP_{\lambda v_{top} g_{top}}$ into Θ_i and updates V_c and Γ_{uc} (Lines 9-11), which takes $O(|V| + |\Gamma|)$ time throughout the while loop. Since the cost of decreasing the key of an element in a min Fibonacci heap is $O(1)$, updating the lowest weight paths in Q (Line 12) takes $O(|\Gamma||V|)$ time throughout the while loop. It updates Θ_{min} using Θ_i (Line 14) in $O(|V|)$ time. After enumerating every $i \in g_{min}$, it identifies the MST (Line 16) in $O(|E| + |V| \log |V|)$ time [32].

Then, it refines Θ_{min} (Lines 17-29). We use a hash to record, for each vertex v in Θ_{min}, the groups in Γ that v belongs to. In addition, we use a hash to record the number of vertices in Θ_{min} that is in each group. The construction of these hashes takes $O(|\Gamma||V|)$ time. Since it takes $O(|\Gamma|)$ time to check whether a leaf is a non-unique-group leaf (by examining the number of vertices in Θ_{min} that are in each group that this leaf belongs to), ImprovAPP pushes non-unique-group leaves into Q_{max} (Lines 18-20) in $O(|\Gamma||V|)$ time. Then, the removal of each non-unique-group leaf from Θ_{min} (Lines 22-28) takes $O(|\Gamma| + \log |V|)$ time. The reason is as follows. Popping the top entry from Q_{max} (Line 22) takes $O(\log |V|)$ time. Checking whether v_{top} is a non-unique-group leaf (Line 23) incurs $O(|\Gamma|)$ cost. If v_{top} is a non-unique-group leaf, it takes $O(1)$ time to remove v_{top} from Θ_{min} (Line 24). After that, it takes $O(|\Gamma|)$ time to check if the neighbor of v_{top} in Θ_{min} is a non-unique-group leaf (Line 25), and if it is, then ImprovAPP inserts it into Q_{max} (Line 26) in $O(1)$ time. Thus, the total cost of Lines 17-29 is $O(|\Gamma||V| + |V| \log |V|)$.

Algorithm 6 The PartialOPT algorithm

Input: a graph $G(V, E, w, c)$, a set of vertex groups Γ, a regulating weight λ, a tunable parameter $h \in [2, |\Gamma|]$
Output: a group Steiner tree: $\Theta(V_\Theta, E_\Theta)$

1: Initialize $\Theta = \emptyset$; $c_\lambda(\Theta) = \infty$
2: Find the smallest vertex group g_{min} in Γ
3: **for** each vertex $i \in g_{min}$ **do**
4: Let Γ_1 be a set of vertex groups that contains $\{i\}$ and the first $h-1$ vertex groups in $\Gamma \setminus g_{min}$;
let Γ_2 be another set of vertex groups that contains $\{i\}$ and the last $|\Gamma| - h$ vertex groups in $\Gamma \setminus g_{min}$
5: $\Theta_i^h = DPBF(G, \Gamma_1, \lambda)$
6: **if** $\Gamma_2 = \{\{i\}\}$ **then**
7: $\Theta_i^{|\Gamma|} = \{i\}$
8: **else**
9: $\Theta_i^{|\Gamma|} = exIhlerA(G, \Gamma_2, \lambda)$
10: **end if**
11: $G_i = \Theta_i^h \cup \Theta_i^{|\Gamma|}$
12: $\Theta_i = MST(G_i)$
13: Execute Lines 17-29 in ImprovAPP to refine Θ_i
14: $\Theta = \min_{c_\lambda}\{\Theta, \Theta_i\}$
15: **end for**
16: Return Θ

5 A $(|\Gamma| - h + 1)$-APPROXIMATION ALGORITHM

Here, we present PartialOPT, which achieves a guarantee of $|\Gamma| - h + 1$, where $h \in [2, |\Gamma|]$ is a tunable parameter. The main idea behind PartialOPT is similar to the idea of the $(|\Gamma| - h + 1)$-approximation algorithm in [23] for solving the vertex-unweighted group Steiner tree problem. The idea is: when enumerating every vertex $i \in g_{min}$, find a tree to connect i with $h-1$ vertex groups in $\Gamma \setminus g_{min}$ optimally, and find another tree to connect i with the other $|\Gamma| - h$ vertex groups in $\Gamma \setminus g_{min}$ sub-optimally using a $(|\Gamma| - 1)$-approximation algorithm, and then combines these two trees as a solution.

However, the $(|\Gamma| - h + 1)$-approximation algorithm in [23] has a large time complexity of $O(3^h \cdot |V|^{h+3})$, and is too slow to be used. The reason is that, when finding a tree to connect i with $h-1$ vertex groups optimally, it enumerates $O(|V|^{h-1})$ sets of $h-1$ vertices such that every set covers the $h-1$ vertex groups, and for every set, it uses an exact Steiner tree algorithm in [15] to find the minimum-weight tree that connects i with this set of vertices.

In comparison, PartialOPT has a smaller time complexity, since PartialOPT employs an exact group Steiner tree algorithm in [14] to connect i with $h-1$ vertex groups optimally.

Description of PartialOPT**.** Algorithm 6 shows the pseudo code of PartialOPT. It first initializes an empty tree Θ, and sets $c_\lambda(\Theta) = \infty$ (Line 1). Then, it identifies $g_{min} \in \Gamma$ (Line 2), and processes each vertex i in g_{min} (Lines 4-14) in seven steps:

(1) Construct a set Γ_1 of vertex groups that contains $\{i\}$ and the first $h-1$ vertex groups in $\Gamma \setminus g_{min}$, and a set Γ_2 of vertex groups that contains $\{i\}$ and the last $|\Gamma| - h$ vertex groups in $\Gamma \setminus g_{min}$.

(2) Invoke DPBF [14] to derive an optimal solution Θ_i^h to the group Steiner tree problem with Γ_1 being the set of vertex groups.
(3) If $\Gamma_2 = \{\{i\}\}$, then let $\Theta_i^{|\Gamma|} = \{i\}$. Otherwise, execute exIhlerA to compute $\Theta_i^{|\Gamma|}$ with Γ_2 being the set of vertex groups. The reason why we use exIhlerA, but not FastAPP or ImprovAPP, here is that exIhlerA incurs a smaller time complexity here. In particular, since Γ_2 contains $\{i\}$, exIhlerA incurs a time complexity of $O((|\Gamma| - h)|V| + |E| + |V| \log |V|)$ here.
(4) Merge Θ_i^h and $\Theta_i^{|\Gamma|}$ to form a graph G_i.
(5) Derive the MST, Θ_i, that spans the vertices in G_i.
(6) Refine Θ_i using Lines 17-29 in ImprovAPP.
(7) If Θ_i has a smaller regulated weight than Θ, then let $\Theta = \Theta_i$.

After the enumeration, PartialOPT returns Θ.

Example of PartialOPT. Take G in Figure 2 as an example. Let $h = 2$. If PartialOPT selects g_2 as g_{min}, then it processes $v_3 \in g_{min}$ as follows. First, it builds $\Gamma_1 = \{\{v_3\}, g_1\}$ and $\Gamma_2 = \{\{v_3\}, g_3\}$. It uses DPBF to find $\Theta_{v_3}^h = \{(v_3, v_4), (v_4, v_2)\}$. Then, it uses exIhlerA to find $\Theta_{v_3}^{|\Gamma|} = \{(v_3, v_4), (v_4, v_5)\}$. It merges $\Theta_{v_3}^h$ and $\Theta_{v_3}^{|\Gamma|}$ as G_{v_3}, which is the highlighted red tree in G. It returns this tree.

Approximation guarantee of PartialOPT**.** PartialOPT has the following guarantee. The proof is in the supplement [6].

Theorem 6. PartialOPT *has a sharp approximation guarantee of* $|\Gamma| - h + 1$ *for solving Problem 1.*

Time complexity of PartialOPT**:**

$$O\Big(|g_{min}| \cdot \Big(|\Gamma||V| + 3^h|V| + 2^h(|E| + h|V| + |V| \log |V|)\Big)\Big).$$

The details of this time complexity are in the supplement [6].

6 RELATED WORK

Group Steiner tree algorithms. Reich and Widmayer [33] first study the vertex-unweighted group Steiner tree problem. Several algorithms have been developed for solving this problem since then (*e.g.*, [14, 19, 22, 23, 25, 28]), in which ENSteiner [25] and IhlerA [23] (*i.e.*, the $(|\Gamma| - 1)$-approximation algorithm in [23]) are two state-of-the-art scalable non-exact algorithms. The reason is that the other algorithms either achieve tight (often poly-logarithmic) approximation guarantees at the cost of large time complexities (*e.g.*, [19, 22]), or find optimal solutions via dynamic programming approaches (*e.g.*, [14, 28]). Ding *et al.* [14] further study the vertex- and edge-weighted group Steiner tree problem. Their DPBF algorithm can solve this problem to optimality. Recently, Li *et al.* [28] develop some powerful pruning techniques to enhance the efficiency of DPBF for solving the vertex-unweighted group Steiner tree problem. Their Basic algorithm can solve the vertex- and edge-weighted group Steiner tree problem to optimality as well. More recently, Chalermsook *et al.* [8] study a special case of this problem where (i) a root vertex in the optimal solution tree is known; and (ii) all edge weights are zero. They point out that some recursive greedy algorithms [9, 10, 22] lead to new algorithms that can achieve an approximation guarantee of $O(\log^2 |\Gamma|)$ for solving this special case. Since these algorithms cannot achieve guarantees in polynomial time, they further propose an algorithm that achieves a guarantee of $O(\log |V| \log |\Gamma|)$ in polynomial time when the treewidth of graph is bounded. We can apply this algorithm to non-rooted graphs with both vertex and edge weights by (i) enumerating all vertices as possible root vertices; and (ii) dividing every edge into two new edges with a new vertex in the middle, and then giving new vertices the weights of divided edges, while setting the weights of new edges to zero. However, it is too slow to use this algorithm. Specifically, this algorithm has a large time complexity of $|V|^{\tilde{O}(tw(G)^2)}$, where $\tilde{O}(x) = O(x \cdot \text{polylog}(x))$, and $tw(G)$ is the treewidth of the input graph G, which often ranges from dozens to hundreds for real graphs [31]. This motivates us to develop more practical algorithms for finding vertex- and edge-weighted group Steiner trees.

Vertex- and edge-weighted Steiner tree algorithms. Klein and Ravi [24] propose the first approximation algorithm for solving the vertex- and edge-weighted Steiner tree problem. Their algorithm has an approximation guarantee of $2 \ln |T_t|$. Guha and Khuller [20] later improve Klein and Ravi's algorithm, and their improvement has an approximation guarantee of $(1.35 + \varepsilon) \ln |T_t|$, for any constant $\varepsilon > 0$. Guha and Khuller point out that both Klein and Ravi's algorithm and their improvement are too slow to be implemented in practice, since both algorithms repeatedly find a tree that minimizes the ratio of the weight of this tree to the number of compulsory vertices that this tree connects. Consequently, Guha and Khuller [20] further propose an algorithm that has an approximation guarantee of $1.6103 \ln |T_t|$. This algorithm, which we refer to as GKA, is a state-of-the-art algorithm for solving the vertex- and edge-weighted Steiner tree problem, given that the more recent work focuses on solving this problem in special graphs, such as unit disk graphs [42], planar graphs [12], and graphs with compulsory leaf vertices [37]. We observe that, even though this algorithm is faster than the other ones, it still does not scale well to large graphs. Specifically, it requires finding the lowest weight paths between all pairs of vertices, which induces a time complexity of $O(|V_t| \cdot (|E_t| + |V_t| \log |V_t|))$. This motivates us to develop LANCET.

7 EXPERIMENTS

In this section, we conduct experiments on a computer with two Intel Xeon Gold 6240 processors and 395 GB RAM[1].

7.1 Datasets

We use three real datasets: Toronto, DBLP and MovieLens.

Toronto. We collect this dataset from the City of Toronto's Open Data Portal [5]. We use it to build the Toronto graph, where each vertex represents a road junction, and each edge represents a road segment. Each vertex is associated with the types of nearby facilities (*e.g.*, schools) and a value representing the nearby traffic count in January 2020. Each edge is associated with a value representing the length of the corresponding road segment. There are 46,073 vertices, 68,353 edges, and 35 types of facilities in total.

We use the nearby traffic count associated with vertex v as the weight of v, and the road length associated with edge e as the weight of e. We normalize all vertex and edge weights to the range of $[0, 1]$. We consider each vertex group in Γ as the set of vertices that are associated with a specific type of facilities. In this case, finding a group Steiner tree could be useful for property search, *e.g.*, for a user who wants to move to a region with low traffic noise and close to a library, a tennis court, and a school.

[1] **Our codes and datasets are at** https://github.com/YahuiSun/GroupSteinerTree

DBLP. It is the DBLP-Citation-network V12 dataset at the AMiner website [1]. We use it to build the DBLP graph, where each vertex represents an expert [41], and each edge between two vertices indicates that the two corresponding experts have co-authored publication(s). Each expert is associated with the number of citations that he or she gets and a set of research topics that his or her publications are on. We refer to each topic as a *skill*. There are 2,497,782 vertices, 12,786,329 edges, and 127,726 research topics in total.

For vertex v that represents an expert with x citations, we use $\ln(x+1)$ as the weight of v, and treat this weight as the hiring cost of the expert. For each edge e connecting two vertices u and v, we follow previous work [36] and set the weight of e as the pairwise Jaccard distance $c(e) = 1 - \frac{|V_u \cap V_v|}{|V_u \cup V_v|}$, where V_u and V_v are the sets of vertices adjacent to u and v, respectively. Such edge weights represent distances between experts in the team formation scenario [25, 29]. We normalize all vertex and edge weights to the range of $[0, 1]$. We consider each vertex group in Γ as the set of vertices that are associated with a specific skill. Then, finding a group Steiner tree could help form a team of experts to perform a task.

MovieLens. It is the MovieLens 25M dataset at the GroupLens website [3]. We use it to build the MovieLens graph, where each vertex represents a movie. Each movie is associated with the genres (*e.g.,* comedy) that this movie belongs to and a number of rating stars (1 to 5) that this movie gets from MovieLens users [4]. There is an edge between two vertices if there are users who give both corresponding movies 5 stars, which indicates that people who like one of these two movies may also like the other one. There are 62,423 vertices, 35,323,774 edges, and 19 genres in total.

For vertex v representing a movie that has an average star of x, we use $5 - x$ as the weight of v, *i.e.,* a small vertex weight indicates that a movie is highly rated. For edge e that connects two movies v and u, if there are y users who give both movies 5 stars, then we use $\frac{1}{y}$ as the weight of e, *i.e.,* a small edge weight indicates that people are likely to like both movies at the same time. We normalize all vertex and edge weights to the range of $[0, 1]$. We consider each vertex group in Γ as the set of vertices that are associated with a specific genre. Then, finding a group Steiner tree could be useful for recommending movies that are related to some certain genres.

7.2 Experiment settings

Algorithms. Except the proposed algorithms, we also implement six state-of-the-art algorithms as follows.

- GKA [20]: a $(1.6103 \ln |T_t|)$-approximation algorithm for finding vertex- and edge-weighted Steiner trees. The main idea of GKA is to greedily merge spiders (*i.e.,* trees having at most one vertex of degree more than two) that contain compulsory vertices.
- DPBF [14]: a dynamic programming algorithm that finds optimal vertex- and edge-weighted group Steiner trees and is widely used for information retrieval in databases (*e.g.,* [21, 26, 27]). The main idea of DPBF is to build an optimal group Steiner tree for covering $|\Gamma|$ vertex groups by dynamically merging optimal group Steiner trees for covering parts of these vertex groups.
- Basic [28]: another dynamic programming algorithm that can find optimal vertex- and edge-weighted group Steiner trees. The main idea of Basic is to first use the dynamically constructed trees in DPBF to build feasible solutions as upper bounds of the optimal solution, and then use these upper bounds to prune unprofitable ones of the dynamically constructed trees. Basic can progressively find sub-optimal solutions with quality guarantees before finding optimal solutions. We utilize this progressive nature, and let Basic return the first found solution such that the regulated weight of this solution is guaranteed to be not larger than r times the optimal regulated weight, where $r \geq 1$ is a parameter. When $r = 1$, Basic returns the optimal solution. When $r > 1$, Basic may not return the optimal solution.
- Basic+: an improvement of Basic. In particular, Basic+ uses the one-label lower bound in [28] to enhance the pruning process of Basic. Basic+ can also progressively find sub-optimal solutions with quality guarantees before finding optimal ones. We also use r to decide the returned solution of Basic+.
- ENSteiner [25] and IhlerA [23]: the vertex-unweighted versions of exENSteiner and exIhlerA, respectively.

Note that, we omit the algorithm in [8] since it incurs prohibitive computational costs, as we have discussed in Section 6. Furthermore, we observe that the PrunedDP and PrunedDP++ algorithms in [28] improve DPBF for finding vertex-unweighted group Steiner trees. In the supplement [6], we show that these two algorithms rely on techniques that do not hold in graphs with vertex weights. Thus, we do not implement these two algorithms here.

Parameters. We vary five parameters as follows.

- $|V|$: the number of vertices. We extract $|V|$ vertices (and the edges between these vertices) from the input data. In particular, we first randomly select a vertex v, and then perform a breadth first search starting from v and extract the first $|V|$ vertices encountered. Since the DBLP and MovieLens graphs are not connected, the breadth first search starting from v may not encounter $|V|$ vertices. In this case, we perform multiple breadth first searches in the above way, until $|V|$ vertices are extracted.
- $|\Gamma|$: the number of vertex groups. For Toronto (*resp.,* DBLP and MovieLens), a candidate vertex group is the set of vertices that are associated with a specific facility type (*resp.,* skill and genre). We select $|\Gamma|$ candidate vertex groups via two different approaches as follows. First, the *uniform* approach: we select $|\Gamma|$ candidate vertex groups uniformly at random. Second, the *non-uniform* approach: the probability of selecting candidate vertex group g is $\frac{|g|}{\sum_{g_x \in \Gamma_{can}} |g_x|}$, where Γ_{can} is the set of all candidate vertex groups. That is to say, the probability of selecting g is in proportion to the size of g. This corresponds to the fact that more common resources are often more frequently used (*e.g.,* there are more machine learning researchers than Steiner tree researchers, and there are also more tasks that require the skill of machine learning than Steiner tree). Since the DBLP and MovieLens graphs are not connected, there could be no feasible solution for some Γ. We regenerate Γ when such a case occurs.
- λ: the regulating constant between vertex and edge weights.
- r: the parameter of Basic and Basic+ (details are above).
- h: the parameter of PartialOPT.

Metrics. We evaluate three metrics as follows.

- $c_\lambda(G')$: the objective value of Problem 1.
- $c(G'_t)$: the objective value of Problem 2.
- t: the running time of algorithms.

7.3 Experiment results

We visualize the experiment results in Figures 3-8. For each set of parameters, we randomly generate 100 instances, and visualize the average metric values for comparison.

The effectiveness of our extensions. exENSteiner and exIhlerA extend the existing ENSteiner and IhlerA, respectively. We compare these algorithms in Figure 3, where vertex groups are selected via the uniform approach, and the parameter settings are: for Toronto, $|V| = 46073$, $|\Gamma| = 8$, $\lambda = 0.33$; for DBLP, $|V| = 2497782$, $|\Gamma| = 6$, $\lambda = 0.33$; for MovieLens, $|V| = 2423$, $|\Gamma| = 6$, $\lambda = 0.33$. We observe that exENSteiner and exIhlerA produce better solutions than ENSteiner and IhlerA, respectively (particularly for Toronto). The reason is that exENSteiner and exIhlerA consider both vertex and edge weights, while ENSteiner and IhlerA ignore vertex weights. Note that, exENSteiner and exIhlerA are slightly slower than ENSteiner and IhlerA, respectively, since it is slightly slower to compute lowest weight paths than to compute shortest paths, due to the calculation of vertex weights. Nevertheless, it is reasonable to conclude that exENSteiner and exIhlerA have similar speeds with ENSteiner and IhlerA, respectively. Thus, exENSteiner and exIhlerA are more effective than ENSteiner and IhlerA for finding vertex- and edge-weighted group Steiner trees. Due to this reason, we do not use ENSteiner and IhlerA in the following main experiments.

The main experiment results. We present the main experiment results in Figures 4 and 5, where vertex groups are selected via the uniform and non-uniform approaches, respectively. We report the average $|g_{min}|$ in these experiments in Figure 6. Note that, $|g_{min}|$ is larger when vertex groups are selected non-uniformly.

Evaluating exact algorithms. We use three exact algorithms: Basic, Basic+ and DPBF. We observe that Basic+ is often faster than DPBF for finding optimal solutions (*e.g.,* Figures 4a-4c). In comparison, Basic often has a similar speed with DPBF (*e.g.,* Figure 4c). The difference between Basic and Basic+ is that Basic+ uses the one-label lower bound in [28] to enhance Basic. The above observation shows the effectiveness of this lower bound. Nevertheless, DPBF can be faster than Basic and Basic+ in some cases (*e.g.,* when $\lambda = 0$ in Figures 4g, 5g, 5i). The reason is that Basic and Basic+ find the lowest weight paths between vertices and vertex groups, and construct feasible solutions progressively, while DPBF does not find these paths or construct these feasible solutions.

Basic, Basic+ and DPBF are often faster when vertex groups are selected non-uniformly. For example, these algorithms are faster in Figure 5d than in Figure 4d. The reason is as follows. Let $T(v, \Gamma')$ be the minimum-weight tree that roots at vertex $v \in V$ and covers all vertex groups in $\Gamma' \subseteq \Gamma$. These algorithms enumerate $T(v, \Gamma')$ for different pairs of v and Γ', in an increasing order of the weight of $T(v, \Gamma')$, until an optimal solution is found. The sizes of non-uniformly selected vertex groups are often larger, which means that the optimal solution is often smaller and has a smaller weight, and as a result these algorithms often find optimal solutions after enumerating a smaller number of trees. Due to this reason, it is too slow to use these algorithms when $|V|$ is large in Figure 4b, where vertex groups are selected uniformly. Similarly, we do not use these algorithms to find optimal solutions in Figures 4e, 4h and 4k.

We note that Basic, Basic+ and DPBF do not scale well to $|\Gamma|$ (see Figures 4d, 4f, 5d-5f), since these algorithms have exponential time complexities with respect to $|\Gamma|$. As a result, in cases where $|\Gamma|$ is not small, it is often too slow to find optimal solutions, and thus required to accept sub-optimal solutions.

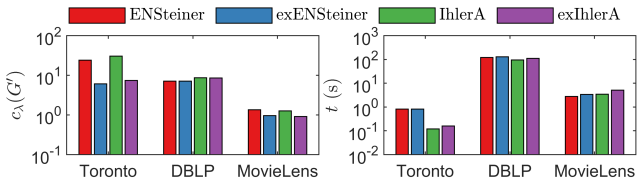


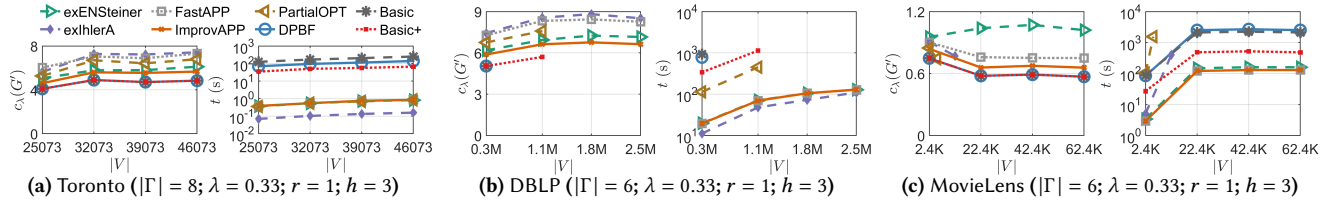
Figure 3: The effectiveness of our extensions.

Different from DPBF, Basic and Basic+ can find sub-optimal solutions with quality guarantees progressively. We let Basic and Basic+ return solutions with the guarantee of r. We vary r in Figures 4j-4l and 5j-5l. When $r = |\Gamma| - 1$, Basic and Basic+ achieve the same guarantee with ImprovAPP. Nevertheless, the solutions of Basic and Basic+ are often worse than the solutions of ImprovAPP when $r = |\Gamma| - 1$ (*e.g.,* Figures 4j, 4l, and 5j-5l). In particular, when $r = |\Gamma| - 1 = 5$ in Figure 5k, the solution weights of Basic and Basic+ are twice that of ImprovAPP, which means that the solutions of Basic and Basic+ are considerably worse than that of ImprovAPP. The reason is as follows. Based on the enumerated $T(v, \Gamma')$, Basic and Basic+ construct a feasible solution by directly merging lowest weight paths between v and every vertex group in $\Gamma \backslash \Gamma'$ into $T(v, \Gamma')$. This direct merging process often induces larger solution weights than a greedy merging process like that in ImprovAPP, *i.e.,* greedily and iteratively merging lowest weight paths between connected vertices and unconnected vertex groups.

Moreover, Basic and Basic+ are often slower than ImprovAPP when $r = |\Gamma| - 1$ (*e.g.,* Figures 4j-4k and 5j-5k). Particularly, when $r = |\Gamma| - 1 = 7$ in Figure 4j, Basic and Basic+ are an order of magnitude slower than ImprovAPP. The reason is that Basic and Basic+ enumerate $T(v, \Gamma')$, and achieve a guarantee of r if the weight of the best found solution is not larger than r times the weight of the enumerated $T(v, \Gamma')$. Basic and Basic+ have an exponential time complexity with respect to $|\Gamma|$ even when $r = |\Gamma| - 1$. In comparison, ImprovAPP does not enumerate $T(v, \Gamma')$ for achieving the guarantee of $|\Gamma| - 1$, and has a polynomial time complexity.

Note that, Basic or Basic+ cannot considerably outperform ImprovAPP on either efficiency or practical solution quality, while there are multiple scenarios where ImprovAPP considerably outperforms Basic and Basic+ on efficiency or practical solution quality (as described above). Thus, Basic and Basic+ often do not have superior efficiency or solution quality for finding sub-optimal solutions. As a result, it may be preferable to use non-exact algorithms in many cases. We evaluate non-exact algorithms as follows.

Evaluating non-exact algorithms. First, we observe that PartialOPT is often slow (*e.g.,* Figure 4b), since it employs DPBF $|g_{min}|$ times for connecting h vertex groups optimally. Thus, we do not use PartialOPT in the full DBLP and MovieLens graphs. Moreover, it is too slow to use exIhlerA when $|g_{min}|$ is large (*e.g.,* Figure 5b), since it employs Dijkstra's algorithm $|g_{min}|$ times. For this reason, t of exIhlerA decreases with $|\Gamma|$ (*e.g.,* Figure 4e), since $|g_{min}|$ decreases with $|\Gamma|$ (see Figure 6). Hence, exIhlerA is only useful when $|g_{min}|$ is small. In comparison, exENSteiner, FastAPP and ImprovAPP can be used when $|g_{min}|$ is large, since these algorithms employ Dijkstra's algorithm $|\Gamma| - 1$ times.



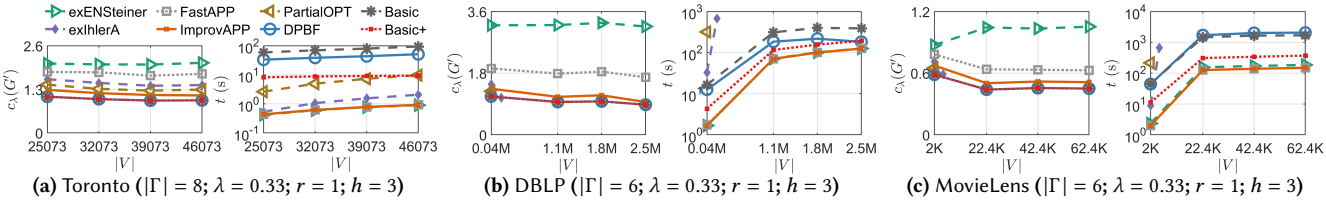
(a) Toronto ($|\Gamma| = 8$; $\lambda = 0.33$; $r = 1$; $h = 3$) (b) DBLP ($|\Gamma| = 6$; $\lambda = 0.33$; $r = 1$; $h = 3$) (c) MovieLens ($|\Gamma| = 6$; $\lambda = 0.33$; $r = 1$; $h = 3$)

(d) Toronto ($|V| = 46073$; $\lambda = 0.33$; $r = 1$; $h = 3$) (e) DBLP ($|V| = 2497782$; $\lambda = 0.33$; $r = 1$) (f) MovieLens ($|V| = 62423$; $\lambda = 0.33$; $r = 1$)

(g) Toronto ($|V| = 46073$; $|\Gamma| = 8$; $r = 1$; $h = 3$) (h) DBLP ($|V| = 2497782$; $|\Gamma| = 6$; $r = 1$) (i) MovieLens ($|V| = 62423$; $|\Gamma| = 6$; $r = 1$)

(j) Toronto ($|V| = 46073$; $|\Gamma| = 8$; $\lambda = 0.33$; $h = 3$) (k) DBLP ($|V| = 2497782$; $|\Gamma| = 6$; $\lambda = 0.33$) (l) MovieLens ($|V| = 62423$; $|\Gamma| = 6$; $\lambda = 0.33$)

Figure 4: The main experiment results in which vertex groups are selected via the uniform approach.



(a) Toronto ($|\Gamma| = 8$; $\lambda = 0.33$; $r = 1$; $h = 3$) (b) DBLP ($|\Gamma| = 6$; $\lambda = 0.33$; $r = 1$; $h = 3$) (c) MovieLens ($|\Gamma| = 6$; $\lambda = 0.33$; $r = 1$; $h = 3$)

(d) Toronto ($|V| = 46073$; $\lambda = 0.33$; $r = 1$; $h = 3$) (e) DBLP ($|V| = 2497782$; $\lambda = 0.33$; $r = 1$) (f) MovieLens ($|V| = 62423$; $\lambda = 0.33$; $r = 1$)

(g) Toronto ($|V| = 46073$; $|\Gamma| = 8$; $r = 1$; $h = 3$) (h) DBLP ($|V| = 2497782$; $|\Gamma| = 6$; $r = 1$) (i) MovieLens ($|V| = 62423$; $|\Gamma| = 6$; $r = 1$)

(j) Toronto ($|V| = 46073$; $|\Gamma| = 8$; $\lambda = 0.33$; $h = 3$) (k) DBLP ($|V| = 2497782$; $|\Gamma| = 6$; $\lambda = 0.33$) (l) MovieLens ($|V| = 62423$; $|\Gamma| = 6$; $\lambda = 0.33$)

Figure 5: The main experiment results in which vertex groups are selected via the non-uniform approach.

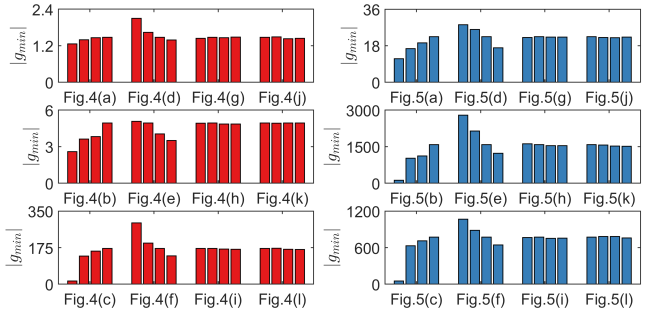


Figure 6: The average $|g_{min}|$ in the main experiment results.

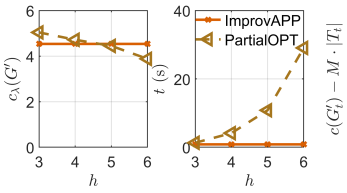


Figure 7: The experiment results of varying h.

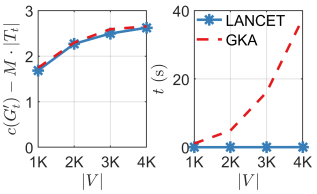


Figure 8: Finding vertex- and edge-weighted Steiner trees.

Notably, exENSteiner can produce high-quality solutions similar to those of ImprovAPP in some cases (*e.g.,* Figure 4b). However, exENSteiner has no approximation guarantee, and produces bad solutions in some other cases (*e.g.,* Figure 5b). In comparison, FastAPP and ImprovAPP have the approximation guarantee of $|\Gamma| - 1$. Note that, exIhlerA and IhlerA are more efficient than the other existing approximation algorithms for finding group Steiner trees (details in Section 6). Since FastAPP and ImprovAPP can achieve the guarantee of $|\Gamma| - 1$ when $|g_{min}|$ is large (while exIhlerA and IhlerA cannot), FastAPP and ImprovAPP advance the existing work on the efficiency of approximating group Steiner trees.

Furthermore, ImprovAPP dominates exENSteiner, exIhlerA and FastAPP on solution qualities. A solution refinement process is used in ImprovAPP, *i.e.,* Lines 17-29 in ImprovAPP. In the supplement [6], we show that the above dominance still holds after using this process to refine the solutions of exENSteiner, exIhlerA and FastAPP. Notably, ImprovAPP has a similar speed with FastAPP, although the time complexity of ImprovAPP is $O(|\Gamma| \cdot (|E| + |V| \log |V| + |g_{min}| \cdot (|V| + \log |\Gamma|)))$, while the time complexity of FastAPP is $O(|\Gamma| \cdot (|E| + |V| \log |V|))$. The reason is as follows. When enumerating vertex $i \in g_{min}$, ImprovAPP constructs a feasible solution tree Θ_i by greedily concatenating lowest weight paths. Suppose that the average number of vertices in Θ_i is d. Then, the time complexity of ImprovAPP can be seen as $O(|\Gamma| \cdot (|E| + |V| \log |V| + |g_{min}| \cdot (d + \log |\Gamma|)))$. In practice, d is often small, *i.e.,* the greedily constructed solution tree is often small. As a result, the cost of ImprovAPP is close to $O(|\Gamma| \cdot (|E| + |V| \log |V|))$ in practice.

Unlike $|g_{min}|$, we generally have a limited $|\Gamma|$ in practice. For example, in team formation scenarios, $|\Gamma|$ is the number of skills for performing a task (*e.g.,* [25, 30, 39]), and in region or keyword search scenarios, $|\Gamma|$ is the number of facility types or keywords that users enter (*e.g.,* [11, 14, 27, 28]). Therefore, it is reasonable to consider that ImprovAPP has a high efficiency in practice. We summarize the above experiment results and conclude that ImprovAPP combines superior efficiency and solution quality when it is too slow to find optimal solutions.

The trade-off in PartialOPT**.** PartialOPT can trade approximation guarantees with time complexities by varying h. We vary h in Figure 7, where the Toronto data is used, vertex groups are selected uniformly, $|V| = 46073$, $|\Gamma| = 6$, $\lambda = 0.33$. We observe that $c_\lambda(G')$ of PartialOPT decreases with h, since it connects h vertex groups optimally. Nevertheless, t of PartialOPT increases exponentially with h, since its time complexity grows exponentially with h. As a result, PartialOPT is mainly of theoretical interest, since, to our knowledge, it achieves the tightest polynomial-time approximation guarantee to date for solving the group Steiner tree problem in treewidth-unbounded graphs with both vertex and edge weights.

The efficiency of LANCET**.** GKA is a state-of-the-art algorithm for solving the vertex- and edge-weighted Steiner tree problem. We compare LANCET with GKA for solving this problem in the transformed graph G_t (see Theorem 1) in Figure 8, where the Toronto data is used, vertex groups are selected uniformly, $\lambda = 0.33$, $|\Gamma| = |T_t| = 6$, and $c(G'_t) - M \cdot |T_t|$ is the non-dummy part of the $c(G'_t)$ value of LANCET or GKA (since each solution contains $|T_t|$ dummy edges, and the weight of each dummy edge is M). We observe that GKA produces similar solutions with LANCET. However, GKA does not scale well to large graphs, while LANCET scales well to large graphs. The reason is that GKA requires finding the lowest weight paths between every pair of vertices, which induces a time complexity of $O(|V_t| \cdot (|E_t| + |V_t| \log |V_t|))$, while LANCET has a time complexity of $O(|T_t| \cdot (|E_t| + |V_t| \log |V_t|))$. Since the other existing algorithms have even weaker scalabilities than GKA (details in Section 6), LANCET advances the existing work on the efficiency of approximating vertex- and edge-weighted Steiner trees.

8 CONCLUSIONS AND FUTURE WORK

Few algorithms have been developed for finding vertex- and edge-weighted group Steiner trees. Here, we develop several algorithms to address this issue. First, we extend a heuristic algorithm and a $(|\Gamma| - 1)$-approximation algorithm from vertex-unweighted graphs to vertex- and edge-weighted graphs. Since the extended $(|\Gamma| - 1)$-approximation algorithm does not scale well to $|g_{min}|$, we develop two new $(|\Gamma|-1)$-approximation algorithms that scale well to $|g_{min}|$. We also propose a $(|\Gamma|-h+1)$-approximation algorithm. Experiments show that, while no algorithm is the best in all cases, our algorithms considerably outperform the state of the art in many scenarios.

Finding group Steiner trees helps retrieve information in relational databases (*e.g.,* [7, 11, 14, 27, 28]). In such applications, the databases are often modeled as directed graphs, and the task is to find a group Steiner tree with a root vertex (*i.e.,* there are directed paths from the root vertex to all the other vertices in this tree; *e.g.,* [7]). We can modify exIhlerA, FastAPP and ImprovAPP to obtain an approximation guarantee of $|\Gamma| - 1$ for finding this tree, since the methods of merging minimum-weight paths between vertices and vertex groups in these algorithms (*e.g.,* Line 5 in exIhlerA) suit directed graphs. We can also modify PartialOPT to obtain a guarantee of $|\Gamma| - h + 1$, since the incorporated DPBF suits directed graphs as well. Thus, it is possible to use the methods in this paper to retrieve information from various graphs (*e.g.,* [35, 40]).

Acknowledgments. We sincerely thank Dr. Bolin Ding [14] and Dr. Ronghua Li [28] for providing the codes of their algorithms. This work is funded by ********** and MOE2016-T2-2-022 from the Singapore Ministry of Education.

REFERENCES

[1] 2021. AMiner. https://www.aminer.org/.
[2] 2021. DBLP: computer science bibliography. https://dblp.uni-trier.de/.
[3] 2021. GroupLens. https://grouplens.org/.
[4] 2021. MovieLens. https://movielens.org/.
[5] 2021. The City of Toronto's Open Data Portal. https://open.toronto.ca/.
[6] 2021. The supplement. https://github.com/YahuiSun/GroupSteinerTree/blob/main/Supplement.pdf.
[7] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. 2002. Keyword searching and browsing in databases using BANKS. In *IEEE International Conference on Data Engineering*. IEEE, 431–440.
[8] Parinya Chalermsook, Syamantak Das, Bundit Laekhanukit, and Daniel Vaz. 2017. Beyond metric embedding: Approximating group steiner trees on bounded treewidth graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 737–751.
[9] Moses Charikar, Chandra Chekuri, To-yat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. 1999. Approximation algorithms for directed Steiner problems. *Journal of Algorithms* 33, 1 (1999), 73–91.
[10] Chandra Chekuri and Martin Pal. 2005. A recursive greedy algorithm for walks in directed graphs. In *46th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 245–253.
[11] Joel Coffman and Alfred C Weaver. 2014. An empirical performance evaluation of relational keyword search techniques. *IEEE Transactions on Knowledge and Data Engineering* 26, 1 (2014), 30–42.
[12] Erik D Demaine, Mohammad Taghi Hajiaghayi, and Philip N Klein. 2014. Node-weighted Steiner tree and group Steiner tree in planar graphs. *ACM Transactions on Algorithms* 10, 3 (2014), 13.
[13] Edsger W Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.
[14] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. 2007. Finding top-k min-cost connected trees in databases. In *IEEE International Conference on Data Engineering*. IEEE, 836–845.
[15] Stuart E Dreyfus and Robert A Wagner. 1971. The Steiner problem in graphs. *Networks* 1, 3 (1971), 195–207.
[16] CW Duin, A Volgenant, and Stefan Voß. 2004. Solving group Steiner problems as Steiner problems. *European Journal of Operational Research* 154, 1 (2004), 323–329.
[17] Karoline Faust, Pierre Dupont, Jérôme Callut, and Jacques Van Helden. 2010. Pathway discovery in metabolic networks by subgraph extraction. *Bioinformatics* 26, 9 (2010), 1211–1218.
[18] Michael L Fredman and Robert Endre Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* 34, 3 (1987), 596–615.
[19] Naveen Garg, Goran Konjevod, and R Ravi. 2000. A polylogarithmic approximation algorithm for the group Steiner tree problem. *Journal of Algorithms* 37, 1 (2000), 66–84.
[20] Sudipto Guha and Samir Khuller. 1999. Improved methods for approximating node weighted Steiner trees and connected dominating sets. *Information and computation* 150, 1 (1999), 57–74.
[21] Shuo Han, Lei Zou, Jeffery Xu Yu, and Dongyan Zhao. 2017. Keyword search on RDF graphs-a query graph assembly approach. In *Proceedings of the ACM Conference on Information and Knowledge Management*. ACM, 227–236.
[22] Christopher S Helvig, Gabriel Robins, and Alexander Zelikovsky. 2001. An improved approximation scheme for the group Steiner problem. *Networks* 37, 1 (2001), 8–20.
[23] Edmund Ihler. 1990. Bounds on the quality of approximate solutions to the group Steiner problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science*. Springer, 109–118.
[24] Philip Klein and R Ravi. 1995. A nearly best-possible approximation algorithm for node-weighted Steiner trees. *Journal of Algorithms* 19, 1 (1995), 104–115.
[25] Theodoros Lappas, Kun Liu, and Evimaria Terzi. 2009. Finding a team of experts in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 467–476.
[26] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. 2008. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 903–914.
[27] Guoliang Li, Xiaofang Zhou, Jianhua Feng, and Jianyong Wang. 2009. Progressive keyword search in relational databases. In *IEEE International Conference on Data Engineering*. IEEE, 1183–1186.
[28] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2016. Efficient and progressive group Steiner tree search. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 91–106.
[29] Xiang Li, Yan Zhao, Xiaofang Zhou, and Kai Zheng. 2020. Consensus-Based Group Task Assignment with Social Impact in Spatial Crowdsourcing. *Data Science and Engineering* 5, 4 (2020), 375–390.
[30] Anirban Majumder, Samik Datta, and KVM Naidu. 2012. Capacitated team formation problem on social networks. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1005–1013.
[31] Silviu Maniu, Pierre Senellart, and Suraj Jog. 2019. An experimental study of the treewidth of real-world graph data. In *22nd International Conference on Database Theory*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[32] Robert Clay Prim. 1957. Shortest connection networks and some generalizations. *Bell system technical journal* 36, 6 (1957), 1389–1401.
[33] Gabriele Reich and Peter Widmayer. 1989. Beyond Steiner's problem: A VLSI oriented generalization. In *International Workshop on Graph-theoretic Concepts in Computer Science*. Springer, 196–210.
[34] G Reich and P Widmayer. 1991. *Approximate minimum spanning trees for vertex classes*. Technical Report. Technical Report, Inst. fur Informatik, Freiburg Univ.
[35] Xiaohan Shan, Wei Chen, Qiang Li, Xiaoming Sun, and Jialin Zhang. 2019. Cumulative activation in social networks. *Science China Information Sciences* 62, 5 (2019), 1–21.
[36] Yahui Sun, Jun Luo, Theodoros Lappas, Xiaokui Xiao, and Bin Cui. 2020. Hunting multiple bumps in graphs. *Proceedings of the VLDB Endowment* 13, 5 (2020), 656–669.
[37] Yahui Sun, Daniel Rehfeldt, Marcus Brazil, Doreen Thomas, and Saman Halgamuge. 2020. A Physarum-inspired algorithm for minimum-cost relay node placement in wireless sensor networks. *IEEE/ACM Transactions on Networking* 28, 2 (2020), 681–694.
[38] Hiromitsu Takahashi and Akira Matsuyama. 1980. An approximate solution for the Steiner problem in graphs. *Math. Japonica* 24, 6 (1980), 573–577.
[39] Xinyu Wang, Zhou Zhao, and Wilfred Ng. 2016. Ustf: A unified system of team formation. *IEEE Transactions on Big Data* 2, 1 (2016), 70–84.
[40] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. 2019. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering* 4, 4 (2019), 352–366.
[41] Jing Zhang and Jie Tang. 2020. Name disambiguation in AMiner. *Science China Information Sciences* 64, 4 (2020), 144101.
[42] Feng Zou, Xianyue Li, Suogang Gao, and Weili Wu. 2009. Node-weighted Steiner tree approximation in unit disk graphs. *Journal of Combinatorial Optimization* 18, 4 (2009), 342.

Finding Group Steiner Trees in Graphs with both Vertex and Edge Weights: Some Supplemental Materials

Road map. In Section 1, we prove the transformation from group Steiner trees to Steiner trees. In Section 2, we prove the approximation guarantee of LANCET. In Section 3, we explain the time complexity of LANCET. In Section 4, we explain the time complexity of exENSteiner. In Section 5, we prove the approximation guarantee of exIhlerA. In Section 6, we prove the approximation guarantee of FastAPP. In Section 7, we prove the approximation guarantee of ImprovAPP. In Section 8, we prove the approximation guarantee of PartialOPT. In Section 9, we explain the time complexity of PartialOPT. In Section 10, we show the usefulness of the refinement process (*i.e.,* Lines 16-29) in ImprovAPP. In Section 11, we show that PrunedDP and PrunedDP++ in [2] rely on techniques that do not hold in graphs with vertex weights. In Section 12, we show the memory consumption results. In Section 13, we refine the solutions of exENSteiner, exIhlerA and FastAPP.

1 THE TRANSFORMATION

Theorem 1. *Let $G(V, E, w, c)$ be a connected undirected graph, and Γ be a set of vertex groups. Let $G_t(V_t, E_t, w_t, c_t)$ be a connected undirected graph, and $T_t \subseteq V_t$ be a set of compulsory vertices. Based on G and Γ, we construct G_t and T_t in the following way:*

(1) Initialize $V_t = V$, $E_t = E$, $T_t = \emptyset$, $w_t = (1-\lambda)w$, and $c_t = \lambda c$.

(2) For each vertex group $g \in \Gamma$, (i) add a dummy vertex v_g into T_t and V_t, such that $w_t(v_g) = 0$, and (ii) add dummy edges (v_g, j) for all $j \in g$ into E_t, such that $c_t(v_g, j) = M$, and M is a constant satisfying

$$M > (1-\lambda)\sum_{v\in V} w(v) + \lambda \sum_{e\in E_{MST}} c(e), \tag{1}$$

and E_{MST} is the set of edges in a Minimum Spanning Tree of G.

Let Θ_{G_t} be an optimal solution to the vertex- and edge-weighted Steiner tree problem in G_t, and $\Theta^{non}_{G_t}$ be the non-dummy part of Θ_{G_t}. Then, there is an optimal solution to the vertex- and edge-weighted group Steiner tree problem in G, namely, Θ_G, that has the same sets of vertices and edges with $\Theta^{non}_{G_t}$.

Proof. Since dummy vertices only connect non-dummy vertices, there are at least $|\Gamma|$ dummy edges in Θ_{G_t}. If $c_\lambda(\Theta_G) < c(\Theta^{non}_{G_t})$, then there is a feasible solution to the vertex- and edge-weighted Steiner tree problem in G_t: Θ'_{G_t} such that

$$c(\Theta'_{G_t}) = c_\lambda(\Theta_G) + M|\Gamma| < c(\Theta_{G_t}), \tag{2}$$

which is not possible. Thus, we have $c_\lambda(\Theta_G) \geq c(\Theta^{non}_{G_t})$. Let Θ''_{G_t} be a tree in G_t such that (i) every dummy vertex v_g is a leaf of Θ''_{G_t}; and (ii) the non-dummy part of Θ''_{G_t}, namely, $\Theta^{non''}_{G_t}$, is in a Minimum Spanning Tree of G. Suppose that there is a dummy vertex v_g in Θ_{G_t} that is not a leaf. Since $c(\Theta^{non''}_{G_t}) < M$, we have

$$c(\Theta_{G_t}) \geq c(\Theta^{non}_{G_t}) + M(|\Gamma|+1) > c(\Theta''_{G_t}) = c(\Theta^{non''}_{G_t}) + M|\Gamma|, \tag{3}$$

which is not possible. Thus, every dummy compulsory vertex v_g is a leaf of Θ_{G_t}. As a result, $\Theta^{non}_{G_t}$ is connected and shares the same sets of vertices and edges with a feasible solution to the vertex- and edge-weighted group Steiner tree problem in G, which means that $c_\lambda(\Theta_G) \leq c(\Theta^{non}_{G_t})$. Therefore, $c_\lambda(\Theta_G) = c(\Theta^{non}_{G_t})$. Hence, this theorem holds. □

2 THE APPROXIMATION GUARANTEE OF LANCET

LANCET can be regarded as the vertex- and edge-weighted version of the algorithm in [3], which achieves an approximation guarantee of $2(1-1/|T_t|)$ for solving the vertex-unweighted Steiner tree problem. This approximation guarantee relies on the following deduction (*i.e.,* Lemma 1 in [3]): since a pre-order traversal of a tree traverses every edge in this tree exactly twice (see Figure 1 in [3]), in a graph with only edge weights, if we perform a pre-order traversal of an optimal solution tree and sum up every weight that we encounter (including duplicates), then the result is exactly twice the weight of an optimal solution tree. However, in a graph with both vertex and edge weights, summing up the weights that we encounter during this traversal does not always result in twice the weight of an optimal solution tree, since (i) an optimal solution tree may contain non-compulsory vertices with positive weights; and (ii) a pre-order traversal of an optimal solution tree may visit such a vertex more than twice (specifically, the number of times that a pre-order traversal of an optimal solution tree visits such a vertex equals the degree of this vertex in this optimal solution tree). Thus, the above approximation guarantee of $2(1-1/|T_t|)$ does not hold for LANCET. In what follows, we establish the approximation guarantee of LANCET.

Theorem 2. LANCET *has a sharp approximation guarantee of $|T_t|-1$ for solving the vertex- and edge-weighted Steiner tree problem.*

Proof. LANCET merges $|T_t|-1$ LWPs to connect all compulsory vertices together. Suppose that the highest-weight one of these LWPs is LWP', and Θ_{opt} is an optimal solution. Since $c(LWP')$ is smaller than or equal to the weight of the LWP between a pair of compulsory

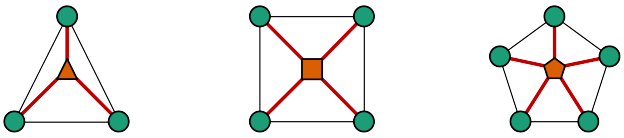


Figure 1: Touching the approximation guarantee of $|T_t| - 1$.

vertices, we have

$$c(\Theta_{opt}) \geq c(LWP'). \tag{4}$$

Since there are $|T_t| - 1$ LWPs that have been merged, we have

$$(|T_t| - 1)c(\Theta_{opt}) \geq (|T_t| - 1)c(LWP') \geq c(\Theta). \tag{5}$$

Therefore, LANCET has an approximation guarantee of $|T_t| - 1$. We further show that $|T_t| - 1$ is the sharp approximation guarantee of LANCET. Consider a regular polygon composed of $|T_t|$ compulsory vertices, and a non-compulsory vertex that is in the middle of this polygon and connects $|T_t|$ compulsory vertices (see Figure 1). The weight of each edge between compulsory vertices is x, the weight of each edge between a compulsory vertex and the middle non-compulsory vertex is 0, the weight of each compulsory vertex is 0, and the weight of the middle non-compulsory vertex is z. Suppose that $x = z - \delta$, where δ is a tiny positive value; and $z < (|T_t| - 1)x$. Since $x < z$, Θ contains $|T_t| - 1$ edges between compulsory vertices, and $c(\Theta) = (|T_t| - 1)x$. Since $z < (|T_t| - 1)x$, Θ_{opt} contains all the edges between compulsory vertices and the middle non-compulsory vertex, and $c(\Theta_{opt}) = z$. We have

$$\lim_{\delta \to 0} \frac{c(\Theta)}{c(\Theta_{opt})} = \frac{(|T_t| - 1)(z - \delta)}{z} = |T_t| - 1. \tag{6}$$

Hence, $|T_t| - 1$ is the sharp approximation guarantee of LANCET. □

3 THE TIME COMPLEXITY OF LANCET

Time complexity of LANCET:

$$O\Big(|T_t| \cdot (|E_t| + |V_t| \log |V_t|)\Big).$$

The details are as follows. The overhead of the initialization (Lines 1-2) is $O(|T_t|)$. The LWPs from a vertex to the other vertices can be found using Dijkstra's algorithm in $O(|E_t| + |V_t| \log |V_t|)$ time. Thus, the cost of finding the LWPs from every vertex in V_2 to the other vertices (Line 3) is $O(|T_t|(|E_t| + |V_t| \log |V_t|))$. It takes $O(|T_t|)$ time to push the LWPs from every vertex in V_2 to i_{rand} into Q (Line 4). It concatenates the LWPs between unconnected compulsory vertices and connected vertices using a while loop with $O(|T_t|)$ iterations as follows. Since popping out the top element in the Fibonacci heap takes $O(\log |T_t|)$ time, it pops out $LWP_{min}(V_{min}, E_{min})$ in $O(|T_t| \log |T_t|)$ time throughout the loop (Line 6). We use adjacency lists based on hashes to store graphs. Since adding a vertex or an edge into such an adjacency list takes $O(1)$ time, LANCET merges $LWP_{min}(V_{min}, E_{min})$ into Θ (Line 7) in $O(|V_t|)$ time in all iterations combined. We use hashes to store V_1 and V_2. As a result, updating V_1 and V_2 (Lines 8-9) takes $O(|V_t|)$ time throughout the loop. Since the time complexity of decreasing the key of an element in a min Fibonacci heap is $O(1)$; and LANCET checks and updates the minimum-weight LWPs from every vertex in V_2 to V_1 only when a new vertex is added into V_1, updating the LWPs (Line 10) takes $O(|T_t||V_t|)$ time throughout the loop.

4 THE TIME COMPLEXITY OF exENSteiner

Time complexity of exENSteiner:

$$O\Big(|\Gamma| \cdot (|E| + |V| \log |V| + |\Gamma| \log |\Gamma| + |\Gamma||V|)\Big).$$

The details are as follows. exENSteiner first transforms the input graph in $O(|E| + |\Gamma||V|)$ time (Line 1). This is because (i) this transformation requires examining all vertices and edges to decide the value of M, which takes $O(|V|+|E|)$ time, and (ii) it adds $|\Gamma|$ dummy vertices and $\sum_{g \in \Gamma} |g|$ dummy edges, which incurs $O(|\Gamma||V|)$ overhead. Then, it employs LANCET to find a Steiner tree in $O(|\Gamma|(|E|+|V| \log |V|+|\Gamma| \log |\Gamma|+|\Gamma||V|))$ time (Line 2), since LANCET takes $O(|T_t|(|E_t| + |V_t| \log |V_t|))$ time; and $|T_t| = |\Gamma|$, $|V_t| = |V| + |\Gamma|$ and $|E_t| = |E| + \sum_{g \in \Gamma} |g|$. After that, it removes $|\Gamma|$ dummy vertices and $|\Gamma|$ dummy edges (Line 3), which takes $O(|\Gamma|)$ time. It identifies the MST (Line 4) in $O(|E| + |V| \log |V|)$ time. Notably, since $|\Gamma|$ is often limited in practice, the cost of exENSteiner is close to $O(|\Gamma| \cdot (|E| + |V| \log |V|))$ in practice.

5 THE APPROXIMATION GUARANTEE OF exIhlerA

Theorem 3. exIhlerA *has a sharp approximation guarantee of* $|\Gamma| - 1$ *for solving the vertex- and edge-weighted group Steiner tree problem.*

Proof. Suppose that $\Theta_{OPT}(V_{OPT}, E_{OPT})$ is an optimal solution. Let $\Gamma = \{g_1, \ldots, g_{|\Gamma|}\}$. There is a tuple $(v_1, \ldots, v_{|\Gamma|})$ such that $v_i \in V_{OPT} \cap g_i$ for all $i \in \{1, \ldots, |\Gamma|\}$. Without loss of generality, assume that $g_{min} = g_1$. For every $i \in \{2, \ldots, |\Gamma|\}$, there is exactly one simple path between v_1 and v_i in Θ_{OPT}, which we refer to as $P_{v_1 v_i}$. We have

$$c_\lambda(P_{v_1 v_i}) \leq c_\lambda(\Theta_{OPT}), \tag{7}$$

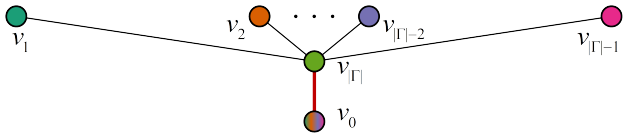


Figure 2: Touching the approximation guarantee of $|\Gamma|-1$.

$$\sum_{g\in\Gamma\setminus g_1} c_\lambda(LWP_{\lambda v_1 g}) \le \sum_{i\in\{2,\ldots,|\Gamma|\}} c_\lambda(P_{v_1 v_i}). \tag{8}$$

Thus,

$$c_\lambda(\Theta) \le c_\lambda(G_{v_1}) \le \sum_{g\in\Gamma\setminus g_1} c_\lambda(LWP_{\lambda v_1 g}) \le \sum_{i\in\{2,\ldots,|\Gamma|\}} c_\lambda(P_{v_1 v_i}) \le (|\Gamma|-1)c_\lambda(\Theta_{OPT}). \tag{9}$$

Hence, exIhlerA has an approximation guarantee of $|\Gamma|-1$. We note that this guarantee is sharp. To explain, consider the graph $G(V,E,w,c)$ in Figure 2, where $V=\{v_0,v_1,\ldots,v_{|\Gamma|}\}$, $E=\{(v_{|\Gamma|},v_0),(v_{|\Gamma|},v_1),\ldots,(v_{|\Gamma|},v_{|\Gamma|-1})\}$, $w(i)=0$ for all $i\in V$, $c(v_{|\Gamma|},v_1)=\ldots=c(v_{|\Gamma|},v_{|\Gamma|-1})=1$, and $c(v_{|\Gamma|},v_0)=1+\delta$, where δ is a tiny positive value. In addition, $\Gamma=\{v_0,v_1\}\cup\ldots\cup\{v_0,v_{|\Gamma|-1}\}\cup\{v_{|\Gamma|}\}$. Let $\lambda=1$. Since $g_{min}=\{v_{|\Gamma|}\}$, exIhlerA produces the solution $\Theta=\{(v_{|\Gamma|},v_1),\ldots,(v_{|\Gamma|},v_{|\Gamma|-1})\}$, and $c_\lambda(\Theta)=|\Gamma|-1$. When $|\Gamma|=2$, Θ is the optimal solution, *i.e.*, the approximation ratio is $|\Gamma|-1=1$. When $|\Gamma|>2$, we have $\Theta_{OPT}=\{(v_{|\Gamma|},v_0)\}$, and

$$\lim_{\delta\to 0}\frac{c_\lambda(\Theta)}{c_\lambda(\Theta_{OPT})}=\frac{|\Gamma|-1}{1+\delta}=|\Gamma|-1. \tag{10}$$

Hence, the best possible approximation guarantee of exIhlerA is $|\Gamma|-1$. □

6 THE APPROXIMATION GUARANTEE OF FastAPP

Theorem 4. FastAPP *has a sharp approximation guarantee of* $|\Gamma|-1$ *for solving the vertex- and edge-weighted group Steiner tree problem.*

Proof. Let $\Theta_{OPT}(V_{OPT},E_{OPT})$ be an optimal solution, and $\Gamma=\{g_1,\ldots,g_{|\Gamma|}\}$. There is a tuple $(v_1,\ldots,v_{|\Gamma|})$ such that $v_i\in V_{OPT}\cap g_i$ for all $i\in\{1,\ldots,|\Gamma|\}$. Without loss of generality, suppose that $g_{min}=g_1$. Let $g_x\in\Gamma\setminus g_1$ be such a vertex group that

$$c_\lambda(LWP_{\lambda v_1 g_x})=\max\{c_\lambda(LWP_{\lambda v_1 g})\mid\forall g\in\Gamma\setminus g_1\}. \tag{11}$$

Since $LWP_{\lambda v_1 g_x}$ links fewer groups to v_1 than Θ_{OPT}, we have

$$c_\lambda(LWP_{\lambda v_1 g_x})\le c_\lambda(\Theta_{OPT}). \tag{12}$$

Lines 5-8 in FastAPP guarantee that

$$\max\{c_\lambda(LWP_{\lambda i_{ming}})\mid\forall g\in\Gamma\setminus g_1\}\le c_\lambda(LWP_{\lambda v_1 g_x}). \tag{13}$$

By Lines 10-11 in FastAPP, we have

$$c_\lambda(\Theta)\le(|\Gamma|-1)\cdot\max\{c_\lambda(LWP_{\lambda i_{ming}})\mid\forall g\in\Gamma\setminus g_1\}. \tag{14}$$

Thus,

$$c_\lambda(\Theta)\le(|\Gamma|-1)c_\lambda(LWP_{\lambda v_1 g_x})\le(|\Gamma|-1)c_\lambda(\Theta_{OPT}). \tag{15}$$

Hence, FastAPP has an approximation guarantee of $|\Gamma|-1$. The sharpness of this guarantee can be seen from the example in Section 5, *i.e.,* Figure 2. Hence, this theorem holds. □

7 THE APPROXIMATION GUARANTEE OF ImprovAPP

Theorem 5. ImprovAPP *has a sharp approximation guarantee of* $|\Gamma|-1$ *for solving the vertex- and edge-weighted group Steiner tree problem.*

Proof. Let $\Theta_{OPT}(V_{OPT},E_{OPT})$ be an optimal solution. Let $\Gamma=\{g_1,\ldots,g_{|\Gamma|}\}$. There is a tuple $(v_1,\ldots,v_{|\Gamma|})$ such that $v_i\in V_{OPT}\cap g_i$ for all $i\in\{1,\ldots,|\Gamma|\}$. Without loss of generality, suppose that $g_{min}=g_1$. When ImprovAPP processes v_1 in the for loop (Lines 5-14), it concatenates $|\Gamma|-1$ lowest weight paths that link $\{g_2,\ldots,g_{|\Gamma|}\}$, respectively, to build Θ_{v_1}. Let $LWP_{\lambda v_x g_y}$ be one of these paths that has the largest regulated weight, and links $g_y\in\{g_2,\ldots,g_{|\Gamma|}\}$. Then,

$$c_\lambda(\Theta_{v_1})\le(|\Gamma|-1)c_\lambda(LWP_{\lambda v_x g_y}). \tag{16}$$

Let $LWP_{\lambda v_1 g_y}$ be the lowest weight path between v_1 and g_y. Since $LWP_{\lambda v_1 g_y}$ has been pushed into Q initially (Line 6) and has (possibly) been updated to $LWP_{\lambda v_x g_y}$ (Line 12), we have

$$c_\lambda(LWP_{\lambda v_x g_y})\le c_\lambda(LWP_{\lambda v_1 g_y}). \tag{17}$$

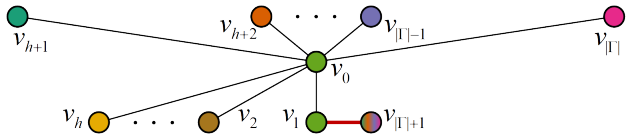


Figure 3: Touching the approximation guarantee of $|\Gamma| - h + 1$.

Since $LWP_{\lambda v_1 g_y}$ links fewer groups to v_1 than Θ_{OPT}, we have

$$c_\lambda(LWP_{\lambda v_1 g_y}) \leq c_\lambda(\Theta_{OPT}). \tag{18}$$

Thus,

$$c_\lambda(\Theta) \leq c_\lambda(\Theta_{v_1}) \leq (|\Gamma| - 1)c_\lambda(LWP_{\lambda v_x g_y}) \leq (|\Gamma| - 1)c_\lambda(LWP_{\lambda v_1 g_y}) \leq (|\Gamma| - 1)c_\lambda(\Theta_{OPT}). \tag{19}$$

Therefore, ImprovAPP has an approximation guarantee of $|\Gamma| - 1$. The sharpness of this guarantee can be seen from the example in Section 5, *i.e.,* Figure 2. Thus, this theorem holds. □

8 THE APPROXIMATION GUARANTEE OF PartialOPT

Theorem 6. PartialOPT *has a sharp approximation guarantee of* $|\Gamma| - h + 1$ *for solving the vertex- and edge-weighted group Steiner tree problem.*

Proof. Suppose that $\Theta_{OPT}(V_{OPT}, E_{OPT})$ is an optimal solution, and $\Gamma = \{g_1, \ldots, g_{|\Gamma|}\}$. There is a tuple $(v_1, \ldots, v_{|\Gamma|})$ such that $v_i \in V_{OPT} \cap g_i$ for all $i \in \{1, \ldots, |\Gamma|\}$. Without loss of generality, let $g_{min} = g_1$. For $v_1 \in g_{min}$, $\Theta^h_{v_1}$ is optimal for $\Gamma_1 = \{\{v_1\}, g_2, \ldots, g_h\}$. Since $\Theta^h_{v_1}$ connects fewer vertex groups to v_1 than Θ_{OPT}, we have

$$c_\lambda(\Theta^h_{v_1}) \leq c_\lambda(\Theta_{OPT}). \tag{20}$$

If $\Gamma_2 = \{\{v_1\}\}$ (*i.e.,* $h = |\Gamma|$), then $\Theta^{|\Gamma|}_{v_1} = \{v_1\}$, and $c_\lambda(\Theta) = c_\lambda(\Theta_{OPT})$. Otherwise (*i.e.,* $h < |\Gamma|$), we implement exIhlerA to produce $\Theta^{|\Gamma|}_{v_1}$ for $\Gamma_2 = \{\{v_1\}, g_{h+1}, \ldots, g_{|\Gamma|}\}$. Suppose that $\Theta^{|\Gamma|}_{OPT}$ is an optimal solution for Γ_2. The proof of Theorem 3 shows that

$$c_\lambda(\Theta^{|\Gamma|}_{v_1}) \leq (|\Gamma| - h)c_\lambda(\Theta^{|\Gamma|}_{OPT}). \tag{21}$$

Since $\Theta^{|\Gamma|}_{OPT}$ connects fewer vertex groups to v_1 than Θ_{OPT}, we have

$$c_\lambda(\Theta^{|\Gamma|}_{OPT}) \leq c_\lambda(\Theta_{OPT}). \tag{22}$$

Thus,

$$c_\lambda(\Theta) \leq c_\lambda(G_{v_1}) = c_\lambda(\Theta^h_{v_1} \cup \Theta^{|\Gamma|}_{v_1}) \leq c_\lambda(\Theta^h_{v_1}) + c_\lambda(\Theta^{|\Gamma|}_{v_1}) \leq (|\Gamma| - h + 1)c_\lambda(\Theta_{OPT}). \tag{23}$$

Therefore, PartialOPT has an approximation guarantee of $|\Gamma| - h + 1$. We show that this guarantee is sharp. Consider the graph $G(V, E, w, c)$ in Figure 3, where $V = \{v_0, v_1, \ldots, v_{|\Gamma|+1}\}$, $E = \{(v_0, v_1), (v_0, v_2), \ldots, (v_0, v_{|\Gamma|}), (v_1, v_{|\Gamma|+1})\}$, $w(i) = 0$ for every $i \in \{v_0, \ldots, v_{h-1}\}$, $w(i) = 1$ for every $i \in \{v_h, \ldots, v_{|\Gamma|+1}\}$, $c(v_0, v_1) = \delta_1$, $c(v_1, v_{|\Gamma|+1}) = \delta_2$, where δ_1 and δ_2 are two tiny positive values, and $\delta_1 < \delta_2$, and all the other edge weights are zero. In addition, $\Gamma = \{g_1, \ldots, g_{|\Gamma|}\} = \{v_0, v_1\} \cup \{v_2, v_{|\Gamma|+1}\} \cup \ldots \cup \{v_{|\Gamma|}, v_{|\Gamma|+1}\}$. Let $\lambda = 0.5$, *i.e.,* vertex and edge weights are regulated equally. PartialOPT enumerates two vertices in g_{min}: v_0 and v_1. For v_0, PartialOPT produces $\Theta^h_{v_0} = \{(v_0, v_2), \ldots, (v_0, v_h)\}$, and $\Theta^{|\Gamma|}_{v_0} = \{(v_0, v_{h+1}), \ldots, (v_0, v_{|\Gamma|})\}$. Thus, $\Theta_{v_0} = \{(v_0, v_2), \ldots, (v_0, v_{|\Gamma|})\}$. Similarly, for v_1, since $\delta_1 < \delta_2$, PartialOPT produces $\Theta_{v_1} = \{(v_0, v_1), \ldots, (v_0, v_{|\Gamma|})\}$. We have $\Theta = \Theta_{v_0}$. When $|\Gamma| = h$, Θ is the optimal solution, *i.e.,* the approximation ratio is $|\Gamma| - h + 1 = 1$. When $|\Gamma| > h$, we have $\Theta_{OPT} = \{(v_1, v_{|\Gamma|+1})\}$, and

$$\lim_{\delta_2 \to 0} \frac{c_\lambda(\Theta)}{c_\lambda(\Theta_{OPT})} = \frac{|\Gamma| - h + 1}{1 + \delta_2} = |\Gamma| - h + 1. \tag{24}$$

Hence, $|\Gamma| - h + 1$ is the best possible approximation guarantee of PartialOPT. This theorem holds. □

9 THE TIME COMPLEXITY OF PartialOPT

Time complexity of PartialOPT:

$$O\Big(|g_{min}| \cdot \Big(|\Gamma||V| + 3^h|V| + 2^h(|E| + h|V| + |V|\log|V|)\Big)\Big).$$

It initializes Θ and finds g_{min} in $O(|\Gamma|)$ time. For each vertex in g_{min}, it builds Γ_1 and Γ_2 in $O(|\Gamma|)$ time (Line 4). It applies DPBF to produce Θ^h_i in $O(3^h|V|+2^h(|E|+h|V|+|V|\log|V|))$ time (Line 5; details in [1]), and may invoke exIhlerA to produce $\Theta^{|\Gamma|}_i$ in $O((|\Gamma|-h)|V|+|E|+|V|\log|V|)$ time (Line 9; here, $|g_{min}|$ in the time complexity of exIhlerA is 1, since Γ_2 contains $\{i\}$). After that, PartialOPT merges Θ^h_i and $\Theta^{|\Gamma|}_i$ (Line 11)

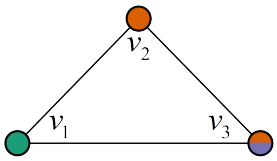


Figure 4: An example for showing the usefulness of the refinement process in ImprovAPP.

in $O(|V|)$ time, and computes an MST as Θ_i (Line 12) in $O(|E| + |V| \log |V|)$ time. It refines Θ_i in $O(|\Gamma||V| + |V| \log |V|)$ time (Line 13), and updates Θ in $O(|V|)$ time (Line 14).

10 AN EXAMPLE FOR ImprovAPP

Here, we show the usefulness of the refinement process (*i.e.,* Lines 16-29) in ImprovAPP via a triangular graph in Figure 4. Let this triangular graph be the input graph G. There are three vertex groups: $g_1 = \{v_1\}$, $g_2 = \{v_3\}$ and $g_3 = \{v_2, v_3\}$. Let the vertex weights be $w(v_1) = 2$, $w(v_2) = 6$ and $w(v_3) = 6$. Let the edge weights be $c(v_1, v_2) = 2$, $c(v_1, v_3) = 8$ and $c(v_2, v_3) = 6$. Let $\lambda = 0.5$. Suppose that ImprovAPP selects g_1 as g_{min} at Line 1. When it processes $v_1 \in g_1$ at Line 4, it pushes $LWP_{\lambda v_1 g_2} = \{(v_1, v_3)\}$ and $LWP_{\lambda v_1 g_3} = \{(v_1, v_2)\}$ into Q at Line 6. The regulated weights of these two paths are 8 and 5, respectively. It first merges $LWP_{\lambda v_1 g_3} = \{(v_1, v_2)\}$ into Θ_{v_1} at Line 9. Since the regulated weight of path $\{(v_1, v_3)\}$ is smaller than the regulated weight of path $\{(v_2, v_3)\}$ (*i.e.,* 8 is smaller than 9), it then merges $\{(v_1, v_3)\}$ into Θ_{v_1} at Line 9. Thus, ImprovAPP builds $\Theta_{min} = \Theta_{v_1} = \{(v_1, v_2), (v_1, v_3)\}$. However, since the weight of edge (v_2, v_3) is smaller than the weight of edge (v_1, v_3) (*i.e.,* 6 is smaller than 8), the MST that spans the vertices in Θ_{min} is $\{(v_1, v_2), (v_2, v_3)\}$. Therefore, after the loop at Lines 4-15, Θ_{min} may not be an MST that spans the vertices in Θ_{min}, which means that finding an MST at Line 16 is useful.

If the weight of edge (v_2, v_3) is 9, then ImprovAPP builds $\Theta_{min} = \Theta_{v_1} = \{(v_1, v_2), (v_1, v_3)\}$, and the MST that spans the vertices in Θ_{min} is still Θ_{min}. However, v_2 is not a unique-group leaf of this MST, and can be removed. Thus, after Line 16, it is not guaranteed that all leaves of Θ_{min} are unique-group leaves, which means that implementing Lines 17-29 to remove non-unique-group leaves is also useful.

11 THE RECENT WORK ON ENHANCING DPBF

The PrunedDP and PrunedDP++ algorithms in [2] enhance the DPBF algorithm in [1] for finding optimal vertex-unweighted group Steiner trees. The main idea of this enhancement is to incorporate pruning techniques into the process of DPBF. In this section, we show that PrunedDP and PrunedDP++ rely on pruning techniques that do not hold in graphs with vertex weights. For the sake of simplicity, we do not use λ to regulate vertex and edge weights in the examples in this section, *i.e.,* we sum vertex and edge weights directly when calculating the weight of a tree. We use $T(v, \Gamma)$ to signify the minimum-weight tree that roots at vertex v and covers all vertex groups in Γ.

Theorem 2 in [2] does not hold in graphs with vertex weights. Theorem 2 in [2] is the core pruning technique in PrunedDP, and is also an important pruning technique in PrunedDP++. This theorem does not hold in graphs with vertex weights. To explain, we first briefly describe the dynamic programming process of DPBF through an example in Figure 5. Understanding this process is necessary for understanding the reason why Theorem 2 in [2] does not hold in graphs with vertex weights.

In Figure 5, there are three vertex groups $g_1 = \{v_1\}$, $g_2 = \{v_2\}$ and $g_3 = \{v_3\}$. The weight of u is 1, and each of the other vertex and edge weights is δ, and δ is a tiny positive value. The optimal solution tree is the whole graph, and the weight of this tree is $1 + 6\delta$ (*i.e.,* the sum of vertex and edge weights). To find this tree, DPBF first initializes $T(v_1, \{g_1\})$ as vertex v_1; $T(v_2, \{g_2\})$ as vertex v_2; and $T(v_3, \{g_3\})$ as vertex v_3. Then, DPBF grows $T(v_1, \{g_1\})$, $T(v_2, \{g_2\})$ and $T(v_3, \{g_3\})$ to vertex u, and produces $T(u, \{g_1\})$ as edge (u, v_1); $T(u, \{g_2\})$ as edge (u, v_2); and $T(u, \{g_3\})$ as edge (u, v_3). Subsequently, it merges $T(u, \{g_1\})$ and $T(u, \{g_2\})$ as $T(u, \{g_1, g_2\}) = \{(u, v_1), (u, v_2)\}$ (similarly, it also merges $T(u, \{g_1\})$ and $T(u, \{g_3\})$ as $T(u, \{g_1, g_3\})$, and merges $T(u, \{g_2\})$ and $T(u, \{g_3\})$ as $T(u, \{g_2, g_3\})$). At last, it produces the optimal solution tree by merging $T(u, \{g_3\})$ and $T(u, \{g_1, g_2\})$ (or $T(u, \{g_1\})$ and $T(u, \{g_2, g_3\})$, or $T(u, \{g_2\})$ and $T(u, \{g_1, g_3\})$).

Theorem 2 in [2] is that: in DPBF, we can merge two subtrees $T(u, \Gamma')$ and $T(u, \Gamma'')$ for $\Gamma'' \subset \Gamma \setminus \Gamma'$ only when the total weight of these two subtrees is not larger than $\frac{2}{3}$ of the weight of an optimal solution tree. This theorem is true when vertex weights are omitted. For example, if vertex weights are omitted in the above instance, then the weight of the optimal solution tree is 3δ. When we merge $T(u, \{g_1\})$ and $T(u, \{g_2\})$ as $T(u, \{g_1, g_2\})$ in the above process, the total weight of $T(u, \{g_1\})$ and $T(u, \{g_2\})$ is 2δ, which is not larger than $\frac{2}{3}$ of the weight of an optimal solution tree. By Theorem 2 in [2], merging these two subtrees may help produce the optimal solution tree. If the total weight of $T(u, \{g_1\})$ and $T(u, \{g_2\})$ is larger than $\frac{2}{3}$ of the weight of an optimal solution tree, then merging these two subtrees does not help produce the optimal solution tree, and thus this merge can be avoided. However, this is not true when vertex weights are considered. For example, if we consider the vertex weights in the above instance, then the total weight of $T(u, \{g_1\})$ and $T(u, \{g_2\})$ is $2 + 4\delta$ (since the weight of each of these two trees is $1 + 2\delta$), which is larger than $\frac{2}{3}$ of the weight of an optimal solution tree: $1 + 6\delta$ (notably, even the weight of $T(u, \{g_1, g_2\}) = \{(u, v_1), (u, v_2)\}$, which is $1 + 4\delta$, is larger than $\frac{2}{3}$ of the weight of an optimal solution tree). As a result, if we use Theorem 2 in [2] in the above instance with vertex weights, then the merge of $T(u, \{g_1\})$ and $T(u, \{g_2\})$ is not carried out (similarly, the merge of $T(u, \{g_1\})$ and $T(u, \{g_3\})$, or the merge of $T(u, \{g_2\})$ and $T(u, \{g_3\})$, is not carried out). Consequently, the optimal solution tree will never be found. That is to say, Theorem 2 in [2] does not hold in graphs with vertex weights.

We point out the specific statement in the proof of Theorem 2 in [2] that does not hold in graphs with vertex weights as follows. In the beginning of the proof of Theorem 2 in [2], an optimal solution is assumed to be a tree rooted at vertex u with k subtrees, $T_1, \ldots, T_k$. Each

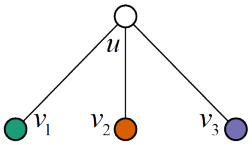


Figure 5: An example for showing Theorem 2 in [2].

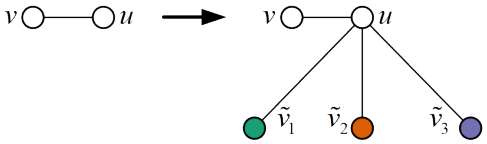


Figure 6: An example for showing Lemmas 2 and 3 in [2].

subtree T_i roots at vertex v_i, and the weight of each subtree is smaller than half of the weight of an optimal solution tree (*e.g.,* in Figure 5, T_i is the single vertex v_i). Let $\overline{T_i}$ be the the edge-grown subtree that is grown by T_i with an edge (v_i, u) (*e.g.,* in Figure 5, $\overline{T_i}$ is the edge (v_i, u)). The proof of Theorem 2 in [2] states that there are three different cases: (1) the weight of each $\overline{T_i}$ is smaller than half of the weight of an optimal solution tree; (2) there is only one edge-grown subtree $\overline{T_i}$ that has a weight no smaller than half of the weight of an optimal solution tree; and (3) there are two edge-grown subtrees and the weight of each one is half of the weight of an optimal solution tree. This statement is not true in vertex-weighted scenarios, where there is a fourth case: there are more than two edge-grown subtrees such that the weight of each one is large than half of the weight of an optimal solution tree. For example, in Figure 5, if we consider vertex weights, then the weight of $\overline{T_1}$, $\overline{T_2}$ or $\overline{T_3}$ is $1 + 2\delta$, which is larger than half of the weight of an optimal solution tree.

Lemmas 2 and 3 in [2] do not hold in graphs with vertex weights. Except Theorem 2 in [2], another important pruning technique in PrunedDP++ is the tour-based lower bounds construction method for A^*-search. There are two types of tour-based lower bounds, which are based on Lemmas 2 and 3 in [2], respectively. We show that these two lemmas do not hold in vertex-weighted scenarios as follows.

First, we introduce the label-enhanced graph in [2], which is constructed by adding dummy vertices and edges into the graph as follows. For each group $g_i \in \Gamma$, we add a dummy vertex $\widetilde{v_i}$, and also add a dummy edge $(\widetilde{v_i}, u)$ with zero weight for every $u \in g_i$. For example, in Figure 6, the graph contains two vertices v and u, and one edge (v, u), and there are three vertex groups $g_1 = g_2 = g_3 = \{u\}$. We add dummy vertices $\widetilde{v_1}, \ldots, \widetilde{v_3}$ and dummy edges $(\widetilde{v_1}, u), \ldots, (\widetilde{v_3}, u)$ for creating the label-enhanced graph. [2] uses $W(\widetilde{v_i}, \widetilde{v_j}, \Gamma')$ to refer to the weight of the minimum-weight route that starts from $\widetilde{v_i}$, ends at $\widetilde{v_j}$, and passes through all dummy vertices that correspond to vertex groups in Γ'. Moreover, [2] uses $d(v, \widetilde{v_i})$ to refer to the weight of the minimum-weight path between non-dummy vertex v and dummy vertex $\widetilde{v_i}$.

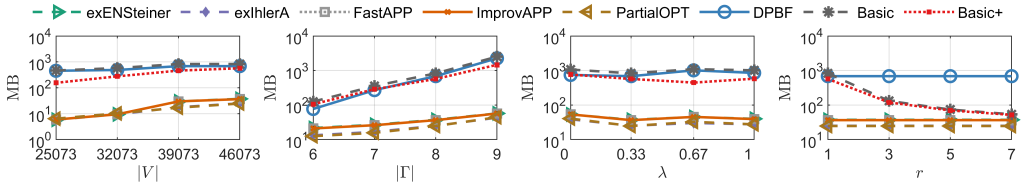
Lemma 2 in [2] is that: for every pair of vertex $v \in V$ and a subset of vertex groups $\Gamma' \subseteq \Gamma$, the weight of $T(v, \Gamma')$ is larger than or equal to $lb_1 = \frac{\min_{g_i, g_j \in \Gamma'} \{d(v, \widetilde{v_i}) + W(\widetilde{v_i}, \widetilde{v_j}, \Gamma') + d(\widetilde{v_j}, v)\}}{2}$. This lemma is true when all vertex weights are zero. For example, in Figure 6, let the weight of edge (v, u) be δ, which is a tiny positive value, and all vertex weights be zero, and $\Gamma' = \{g_1, g_2\}$. Then, $d(v, \widetilde{v_1}) = d(\widetilde{v_2}, v) = \delta$, $W(\widetilde{v_1}, \widetilde{v_2}, \Gamma') = 0$, and $T(v, \Gamma')$ is the edge (v, u). As a result, $lb_1 = \delta$, which equals the weight of $T(v, \Gamma')$. Thus, Lemma 2 in [2] holds. This lemma is proven in [2] by first doubling every edge in the label-enhanced $T(v, \Gamma')$ to obtain an Euler tour that starts from v, ends at v, and passes through all dummy vertices that correspond to vertex groups in Γ'; and then employing the fact that the total edge weight (including duplicates) that we encounter in this Euler tour is twice the total edge weight in $T(v, \Gamma')$. Nevertheless, Lemma 2 in [2] does not hold in vertex-weighted scenarios. For example, in the above instance, let the weights of v and u be 0 and 1, respectively. Then, $d(v, \widetilde{v_1}) = d(\widetilde{v_2}, v) = 1 + \delta$, and $W(\widetilde{v_1}, \widetilde{v_2}, \Gamma') = 1$. As a result, $lb_1 = \frac{3+2\delta}{2}$, which is larger than the weight of $T(v, \Gamma')$: $1 + \delta$. Thus, Lemma 2 in [2] does not hold any more. The reason is that the above Euler tour encounters u three times, and as a result the weight of u is counted three times in lb_1. Generally speaking, the total vertex and edge weight that we encounter in the Euler tour in the proof of Lemma 2 in [2] may be more than twice the weight of $T(v, \Gamma')$, since this tour may visit a vertex in $T(v, \Gamma')$ more than twice. As shown in Section 2 in this supplement, for a similar reason, LANCET does not have an approximation guarantee of 2.

Also for a similar reason, Lemma 3 in [2] does not hold in graphs with vertex weights. The details are as follows. [2] uses $W(\widetilde{v_i}, \Gamma')$ to refer to the weight of the minimum-weight route that starts from $\widetilde{v_i}$, and passes through all dummy vertices that correspond to vertex groups in Γ'. Lemma 3 in [2] is that: the weight of $T(v, \Gamma')$ is larger than or equal to $lb_2 = \frac{\max_{g_i \in \Gamma'} \{d(v, \widetilde{v_i}) + W(\widetilde{v_i}, \Gamma') + \min_{g_j \in \Gamma'} \{d(\widetilde{v_j}, v)\}\}}{2}$. This lemma is true when all vertex weights are zero. Consider the above instance. If all vertex weights are zero, then $d(v, \widetilde{v_1}) = d(v, \widetilde{v_2}) = \delta$, $W(\widetilde{v_1}, \Gamma') = W(\widetilde{v_2}, \Gamma') = 0$, $\min_{g_j \in \Gamma'} \{d(\widetilde{v_j}, v)\} = \delta$, and the weight of $T(v, \Gamma')$ is δ. Consequently, $lb_2 = \delta$, which equals the weight of $T(v, \Gamma')$. Thus, Lemma 3 in [2] holds. Like Lemma 2, Lemma 3 is proven in [2] by doubling every edge in the label-enhanced $T(v, \Gamma')$ to obtain an Euler tour. Also like Lemma 2, since this tour may visit a vertex in $T(v, \Gamma')$ more than twice, Lemma 3 in [2] does not hold in graphs with vertex weights. For example, suppose that, in Figure 6, the weights of v and u are 0 and 1, respectively. Then, $d(v, \widetilde{v_1}) = d(v, \widetilde{v_2}) = 1 + \delta$, $W(\widetilde{v_1}, \Gamma') = W(\widetilde{v_2}, \Gamma') = 1$, $\min_{g_j \in \Gamma'} \{d(\widetilde{v_j}, v)\} = 1 + \delta$, and the weight of $T(v, \Gamma')$ is $1 + \delta$. As a result, $lb_2 = \frac{3+2\delta}{2}$, which is larger than the weight of $T(v, \Gamma')$. Thus, Lemma 3 in [2] does not hold any more.

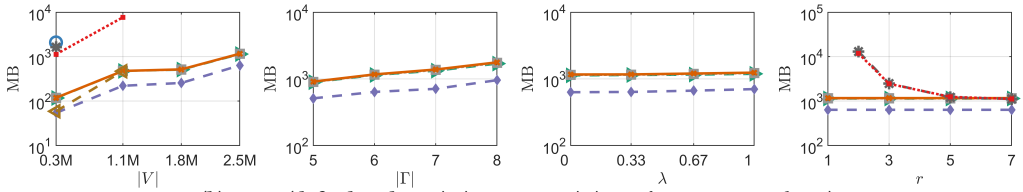
Recall that (i) Theorem 2 in [2] is the core pruning technique in PrunedDP, and is also an important pruning technique in PrunedDP++; and (ii) another important pruning technique in PrunedDP++ is the tour-based lower bounds construction method for A^*-search, and there are two types of tour-based lower bounds, which are based on Lemmas 2 and 3 in [2], respectively. Since Theorem 2 and Lemmas 2 and 3 in [2] do not hold in graphs with vertex weights, we do not implement PrunedDP and PrunedDP++ in our paper.

12 MEMORY CONSUMPTION RESULTS

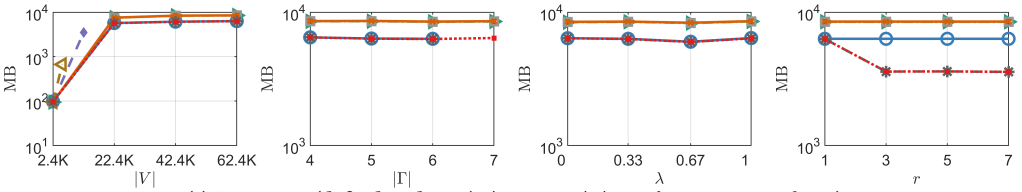
Here, we evaluate the memory consumption of algorithms. The reported memory consumption of each algorithm contains the memory consumed by each input of this algorithm (*e.g.,* G and Γ) as well as any other memory consumed in the process of this algorithm. We use



(a) Toronto **(default values:** $|V| = 46073$; $|\Gamma| = 8$; $\lambda = 0.33$; $r = 1$; $h = 3$**)**

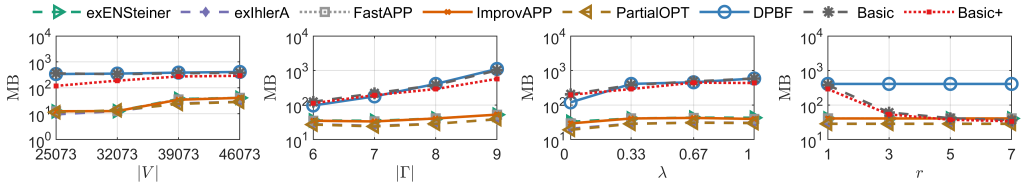


(b) DBLP **(default values:** $|V| = 2497782$; $|\Gamma| = 6$; $\lambda = 0.33$; $r = 1$; $h = 3$**)**

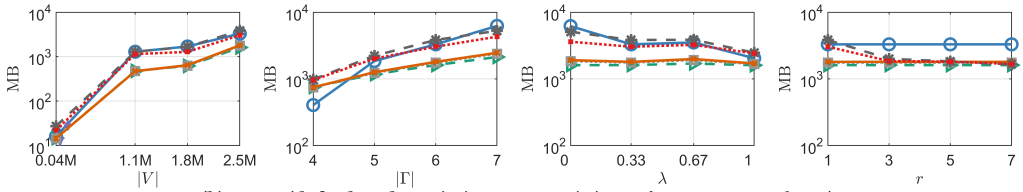


(c) MovieLens **(default values:** $|V| = 62423$; $|\Gamma| = 6$; $\lambda = 0.33$; $r = 1$; $h = 3$**)**

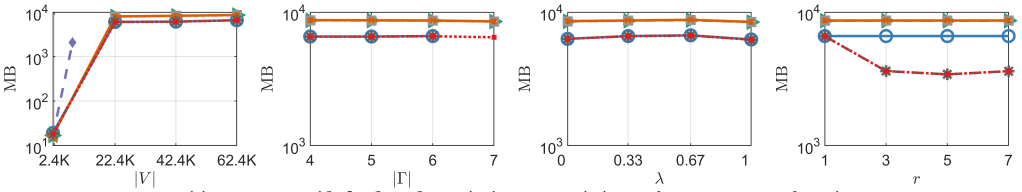
Figure 7: The memory consumption in the main experiments where vertex groups are selected uniformly.



(a) Toronto **(default values:** $|V| = 46073$; $|\Gamma| = 8$; $\lambda = 0.33$; $r = 1$; $h = 3$**)**



(b) DBLP **(default values:** $|V| = 2497782$; $|\Gamma| = 6$; $\lambda = 0.33$; $r = 1$; $h = 3$**)**



(c) MovieLens **(default values:** $|V| = 62423$; $|\Gamma| = 6$; $\lambda = 0.33$; $r = 1$; $h = 3$**)**

Figure 8: The memory consumption in the main experiments where vertex groups are selected non-uniformly.

adjacency lists based on hashes to store graphs. Hashes consume more memories than arrays. Our purpose of using adjacency lists based on hashes is to fully optimize the time complexities of algorithms.

The main experiments. We report the memory consumption results in the main experiments in our paper in Figures 7 and 8. We observe that DPBF, Basic and Basic+ often consume more memory than the other non-exact algorithms (*e.g.*, when varying $|V|$ in Figure 7a), and the memory consumption of DPBF, Basic and Basic+ often increases quickly with $|\Gamma|$ (*e.g.*, when varying $|\Gamma|$ in Figure 7a). The reason is that, except the $O(|V| + |E|)$ memory consumed by the input graph, DPBF, Basic and Basic+ additionally consume $O(2^{|\Gamma|}|V|)$ memory in the dynamic programming process, while the other non-exact algorithms do not consume such an exponential amount of memory. Notably, the memory consumption of DPBF, Basic and Basic+ does not grow much with $|\Gamma|$ for MovieLens (*e.g.*, when varying $|\Gamma|$ in Figure 7c). The reason is that the MovieLens graph is dense, and as a result the memory consumed by the MovieLens graph dominates the increase of the $O(2^{|\Gamma|}|V|)$ memory. Further note that, the non-exact algorithms may consume more memory than DPBF, Basic and Basic+ in some cases

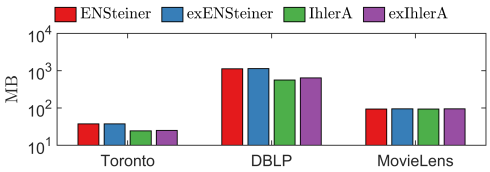


Figure 9: Our extensions.

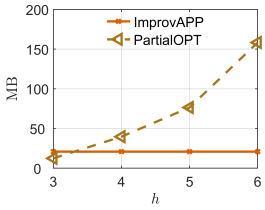


Figure 10: Varying h.

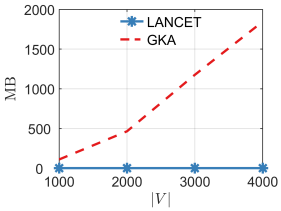


Figure 11: LANCET versus GKA.

(*e.g.,* Figure 7c). The reason is that these non-exact algorithms build an additional graph that has (i) the same set of edges with the input graph G and (ii) newly defined edge weights for finding lowest weight paths in G (see Section 3.2 in our paper on how to find lowest weight paths), while DPBF, Basic and Basic+ do not build such an additional graph.

Since the dynamic programming process of Basic and Basic+ terminates earlier when r is larger, the memory consumption of Basic and Basic+ decreases with r (*e.g.,* when varying r in Figure 7a). Notably, by incorporating pruning techniques, Basic and Basic+ enumerate fewer trees than DPBF in the dynamic programming process. As a result, Basic and Basic+ may consume a smaller amount of memory than DPBF (*e.g.,* when $|V| = 0.3M$ in Figure 7b). However, DPBF may consume a smaller amount of memory than Basic and Basic+ in some cases (*e.g.,* when $|\Gamma| = 6$ in Figure 7a). There are two reasons. First, Basic and Basic+ store the lowest weight paths between vertices and vertex groups, while DPBF does not store these paths. Second, all these three algorithms iteratively pop trees out of a min priority queue. Basic and Basic+ record trees that have been popped out (details in [2]), while DPBF does not record these trees.

Our extensions. We compare the memory consumption of ENSteiner, IhlerA, exENSteiner and exIhlerA in Figure 9, where vertex groups are selected via the uniform approach, and the parameter settings are: for Toronto, $|V| = 46073$, $|\Gamma| = 8$, $\lambda = 0.33$; for DBLP, $|V| = 2497782$, $|\Gamma| = 6$, $\lambda = 0.33$; for MovieLens, $|V| = 2423$, $|\Gamma| = 6$, $\lambda = 0.33$ (this corresponds to the experiments in Figure 3 in our paper). We observe that exENSteiner and exIhlerA may consume slightly more memory than ENSteiner and IhlerA, respectively. The reason is that exENSteiner and exIhlerA build an additional graph for finding lowest weight paths (as discussed above). In comparison, ENSteiner and IhlerA do not build such an additional graph for finding shortest paths.

Varying h in PartialOPT. We report the memory consumption of PartialOPT with respect to h in Figure 10, where the Toronto data is used, vertex groups are selected via the uniform approach, $|V| = 46073$, $|\Gamma| = 6$, $\lambda = 0.33$ (this corresponds to the experiments in Figure 7 in our paper). We observe that the memory consumed by PartialOPT grows quickly with h. The reason is that PartialOPT employs DPBF to connect h vertex groups optimally, and the space complexity of this process is $O(2^h|V|)$.

Comparing LANCET with GKA. We compare the memory consumption of LANCET and GKA in Figure 11, where the Toronto data is used, vertex groups are selected via the uniform approach, $\lambda = 0.33$, $|\Gamma| = |T_t| = 6$ (this corresponds to the experiments in Figure 8 in our paper). We observe that the memory consumption of GKA increases quickly with $|V|$. The reason is that GKA stores the lowest weight paths between all pairs of vertices, which has a space complexity of $O(|V_t|^2)$, where $|V_t| = |V| + |\Gamma|$. In comparison, LANCET only stores the lowest weight paths from compulsory vertices to the other vertices, which has a space complexity of $O(|T_t||V_t|)$ (see Line 3 of LANCET).

13 REFINING THE SOLUTIONS OF exENSteiner, exIhlerA AND FastAPP

There is a solution refinement process in ImprovAPP, *i.e.,* Lines 17-29 in ImprovAPP. This process refines a sub-optimal solution by removing non-unique-group leaves from this solution. Here, we use this process to refine the solutions of exENSteiner, exIhlerA and FastAPP. Notably, this process has already been incorporated into PartialOPT (*i.e.,* Line 13 in PartialOPT). Thus, we do not refine the solutions of PartialOPT here. We report the refinement results in Figures 12 and 13, where exENSteiner+R, exIhlerA+R and FastAPP+R are the refinements of exENSteiner, exIhlerA and FastAPP, respectively.

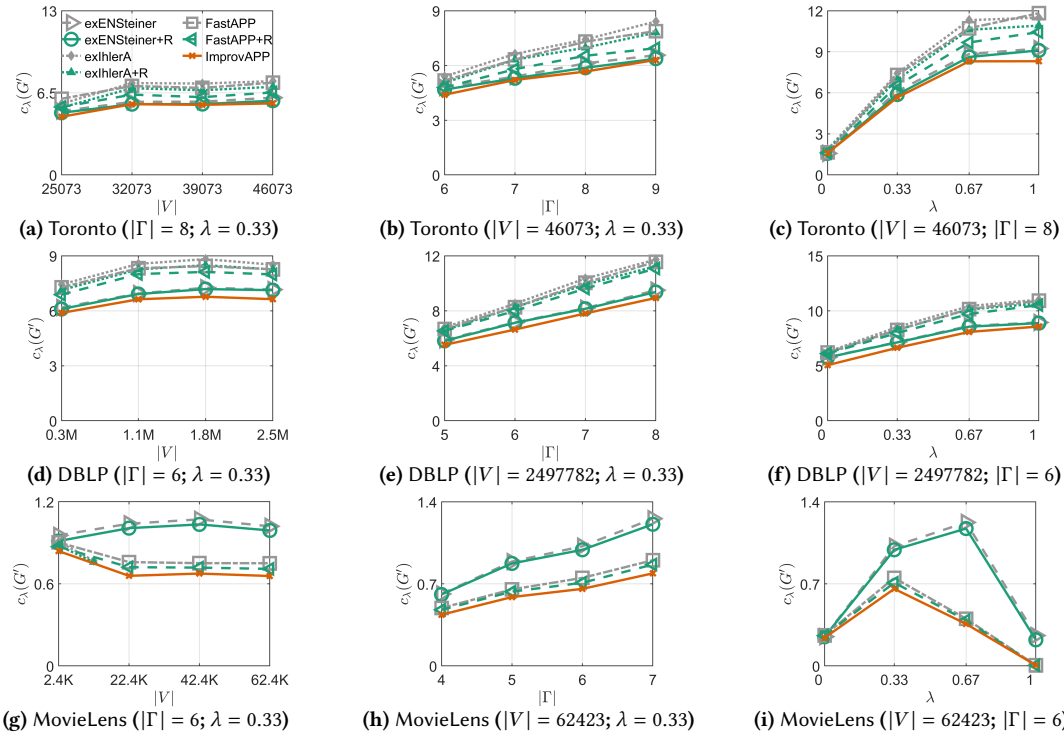
Suppose that there is a feasible solution tree $\Theta(V_\Theta, E_\Theta)$. Then, the time complexity of refining this solution is $O(|\Gamma||V_\Theta| + |V_\Theta| \log |V_\Theta|)$ (details in Section 4.3 in our paper). Since we generally have $|V_\Theta| \ll |V|$ in practice, the running times of refinement are negligible when comparing to the running times of our algorithms. For example, each of our algorithms takes around 100s to produce a feasible solution in the full DBLP graph, while it only takes around 2ms to refine this solution. Thus, we only evaluate the solution qualities in Figures 12 and 13, and do not evaluate the running times of refinement. We observe that ImprovAPP dominates exENSteiner+R, exIhlerA+R and FastAPP+R on solution qualities. We also observe that the refinement is often more effective when vertex groups are selected non-uniformly. For example, the refinement is more effective in Figure 13d than in Figure 12d. The reason is that, when vertex groups are selected non-uniformly, the sizes of the selected vertex groups are often larger, and as a result the leaves in the feasible solutions produced by exENSteiner, exIhlerA and FastAPP are more likely to be non-unique-group leaves.

REFERENCES

[1] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. 2007. Finding top-k min-cost connected trees in databases. In *IEEE International Conference on Data Engineering*. IEEE, 836–845.

[2] Rong-Hua Li, Lu Qin, Jeffrey Xu Yu, and Rui Mao. 2016. Efficient and progressive group Steiner tree search. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 91–106.

[3] Hiromitsu Takahashi and Akira Matsuyama. 1980. An approximate solution for the Steiner problem in graphs. *Math. Japonica* 24, 6 (1980), 573–577.

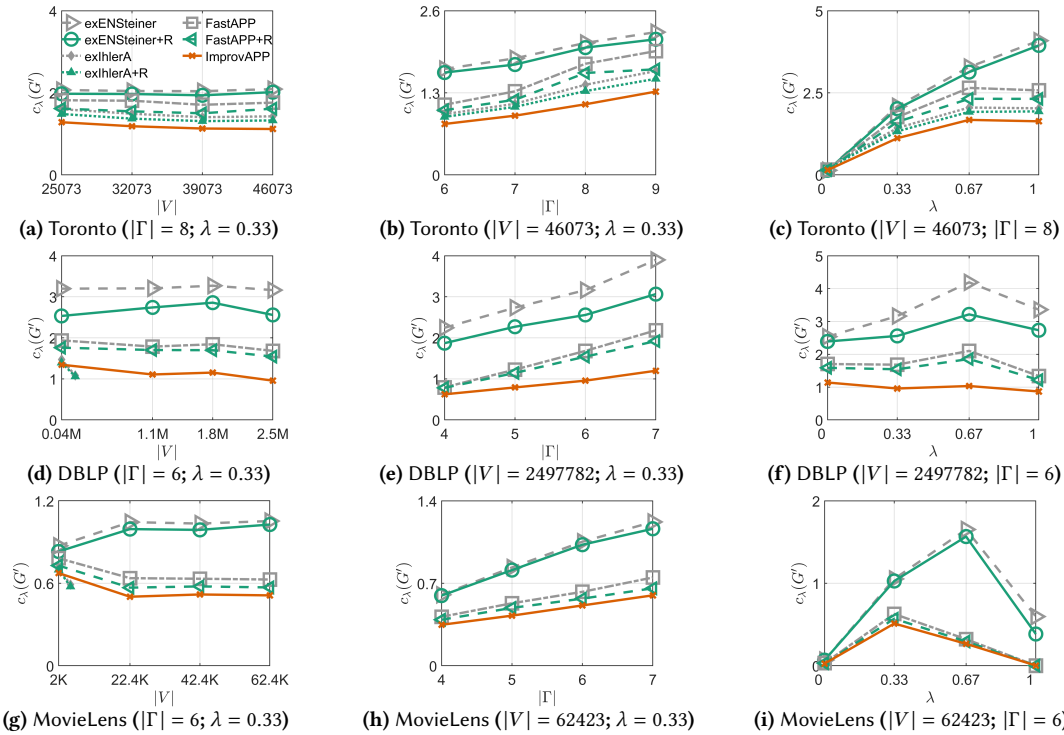


(a) Toronto ($|\Gamma| = 8$; $\lambda = 0.33$) **(b)** Toronto ($|V| = 46073$; $\lambda = 0.33$) **(c)** Toronto ($|V| = 46073$; $|\Gamma| = 8$)

(d) DBLP ($|\Gamma| = 6$; $\lambda = 0.33$) **(e)** DBLP ($|V| = 2497782$; $\lambda = 0.33$) **(f)** DBLP ($|V| = 2497782$; $|\Gamma| = 6$)

(g) MovieLens ($|\Gamma| = 6$; $\lambda = 0.33$) **(h)** MovieLens ($|V| = 62423$; $\lambda = 0.33$) **(i)** MovieLens ($|V| = 62423$; $|\Gamma| = 6$)

Figure 12: The refinement results in the main experiments where vertex groups are selected uniformly.



(a) Toronto ($|\Gamma| = 8$; $\lambda = 0.33$) **(b)** Toronto ($|V| = 46073$; $\lambda = 0.33$) **(c)** Toronto ($|V| = 46073$; $|\Gamma| = 8$)

(d) DBLP ($|\Gamma| = 6$; $\lambda = 0.33$) **(e)** DBLP ($|V| = 2497782$; $\lambda = 0.33$) **(f)** DBLP ($|V| = 2497782$; $|\Gamma| = 6$)

(g) MovieLens ($|\Gamma| = 6$; $\lambda = 0.33$) **(h)** MovieLens ($|V| = 62423$; $\lambda = 0.33$) **(i)** MovieLens ($|V| = 62423$; $|\Gamma| = 6$)

Figure 13: The refinement results in the main experiments where vertex groups are selected non-uniformly.