Received: 26 June 2020 | Revised: 20 April 2021 | Accepted: 28 April 2021

DOI: 10.1002/net.22046

RESEARCH ARTICLE

WILEY

Approximation algorithms for Steiner forest: An experimental study

Laleh Ghalami | **Daniel Grosu**

Department of Computer Science, Wayne State University, Detroit, Michigan, USA

Correspondence
Daniel Grosu, Department of Computer Science, Wayne State University, Detroit, MI 48202, USA.
Email: dgrosu@wayne.edu

Abstract

In the Steiner forest problem, we are given a set of terminal pairs and need to find the minimum cost subgraph that connects each of the terminal pairs together. Motivated by the recent work on greedy approximation algorithms for the Steiner forest, we provide efficient implementations of existing approximation algorithms and conduct a thorough experimental study to characterize their performance. We consider several approximation algorithms: the influential primal-dual 2-approximation algorithm due to Agrawal, Klein, and Ravi, the greedy algorithm due to Gupta and Kumar, and a randomized algorithm based on probabilistic approximation by tree metrics. We also consider the simplest heuristic greedy algorithm for the problem, which picks the closest unconnected pair of terminals and connects it using the shortest path between the terminals in the current graph. To characterize the performance of the algorithms, we created a new library with more than one thousand Steiner forest problem instances and conducted an extensive experimental analysis on those instances. Our analysis reveals that for the majority of instances the primal-dual algorithm is the fastest among all the algorithms considered here, and obtains solutions that are very close to the optimal solutions obtained by solving the integer program formulation of the problem.

KEYWORDS

approximation algorithms, performance evaluation, Steiner forest

1 | INTRODUCTION

The *Steiner tree problem* is one of the most fundamental problems in combinatorial optimization having a large number of applications ranging from network design to computational biology. In the Steiner tree problem, we are given an undirected graph with nonnegative weights for all edges and a subset of vertices, called terminals, and the goal is to find a tree of minimum cost spanning all terminal nodes, where the cost is the sum of the weights of the edges in the tree. This is one of the problems in Karp's original list of NP-complete problems [25]. A natural generalization of the Steiner tree problem is the *Steiner forest problem*, in which instead of a subset of vertices (terminals), we are given k pairs of vertices, and the goal is to find a minimum cost subset of edges such that every pair is connected in the subset of selected edges. The Steiner forest problem is defined as follows:

> *The Steiner forest problem*: Given an undirected graph $G=(V,E)$, with nonnegative weights $w_e \geq 0$ for all edges $e \in E$, and k *terminal pairs*, (s_i, t_i), $s_i, t_i \in V$, find a minimum cost subset of edges $F \subseteq E$ such that every pair (s_i, t_i) is connected in the subgraph (V, F).

The Steiner forest problem is a generalization of the Steiner tree problem and therefore it is APX-hard [6] and NP-hard to approximate within 96/95 [9]. The Steiner forest admits a PTAS if the graph is planar, or more generally, if the graph has

a bounded genus [4]. Several approximation algorithms have been proposed for the Steiner forest problem. The first such approximation algorithm is the 2-approximation primal-dual algorithm proposed by Agrawal, Klein, and Ravi [1] (henceforth called PD-SF [primal dual-Steiner forest]), which was later revisited and extended to several network design problems by Goemans and Williamson [17]. PD-SF is based on the undirected cut relaxation (UCR) for the Steiner forest problem. This has been the only known constant-factor approximation algorithm for the Steiner forest problem for a long time. Jain [22] proposed a 2-approximation algorithm for the *survival network design problem* based on an iterative-rounding approach. The *survival network design problem*, where the goal is to design low-cost networks that can survive failures in the edges of the network, is a generalization of the Steiner forest problem and therefore, the iterative-rounding algorithm is a 2-approximation algorithm for the Steiner forest problem.

Könemann et al. [27] designed a primal-dual algorithm for a game-theoretical variant of the Steiner forest problem. They introduced a new LP relaxation for the Steiner forest problem, called the lifted cut relaxation (LCR). Although the authors showed that the new relaxation provides a better solution than PD-SF in some cases, its approximation ratio is still 2, and the solution computed is usually costlier than that obtained by PD-SF. Recently, Çivril [11] proposed a different LP relaxation for the problem obtaining an algorithm with the same approximation ratio of 2. The design idea was inspired by the bidirected cut relaxation (BCR) for the Steiner tree problem in which each edge is replaced by two arcs in both directions. Using the new relaxation, the author designed an algorithm that is similar to the standard primal-dual algorithm, which grows the dual variables in a synchronized fashion. The algorithm constructs the solution in two directions starting from both terminal end points. Similar to [27], Çivril [11] provided some tight examples for which the new algorithm finds a near-optimal solution.

A simple greedy algorithm for the problem is the *paired greedy algorithm* that iteratively connects terminal pairs which have the minimum distance. Chan, Roughgarden, and Valiant [8] showed that the solution obtained by the paired greedy algorithm has cost $\Omega(\log n)$ times that of the optimal solution, where n is the number of vertices in the graph. Until recently, there were no known constant-factor approximation algorithms based on purely combinatorial techniques. An algorithm based on purely combinatorial techniques is the *gluttonous algorithm* which is similar to the paired greedy algorithm, except that it repeatedly connects the two closest terminals regardless of whether they are pairs. Gupta and Kumar [19] proved that gluttonous is a 96-approximation algorithm for the Steiner forest problem. A randomized $O(\log n)$-approximation algorithm for the Steiner forest problem can be obtained using probabilistic approximation of metrics by tree metrics [15]. Recently, Groß et al. [18] proposed a local-search-based constant-factor approximation algorithm for the problem. The authors introduced a set of local moves that are generalizations of simple edge swaps that are applied at each local step. At each step, the algorithm maintains a feasible forest and performs one of the local moves.

1.1 | Related work

The closest works to ours consist of experimental studies [7, 12, 23, 32–34, 39], software packages implementing various algorithms (GeoSteiner [41], SCIP-Jack [33]), and benchmark suites (SteinLib [26], PACE2018 [31], Vienna [29]) for the Steiner tree problem and its variants. To the best of our knowledge there are no previous experimental studies on approximation algorithms for Steiner forest. Also, there are no publicly available benchmark suites for the Steiner forest problem.

1.2 | Our contributions

In this article, we consider the Steiner forest problem and explore from a practical point of view the design space of approximation algorithms for solving it. In particular, we are interested in determining which of the existing approximation algorithms perform well in practice. To achieve this, we implement and engineer several approximation algorithms for Steiner forest and perform an extensive experimental analysis that allows us to characterize their performance with respect to both running time and quality of the solution. More specifically, we provide efficient C++ implementations for the following algorithms: (i) the 2-approximation primal-dual algorithm proposed by Agrawal, Klein, and Ravi [1] (denoted here by PD-SF); (ii) the 96-approximation algorithm (gluttonous) studied by Gupta and Kumar [19] (denoted here by GL-SF); (iii) a randomized $O(\log n)$-approximation algorithm obtained using probabilistic approximation of metrics by tree metrics [15] (denoted here by TM-SF); and (iv) a heuristic greedy algorithm called paired greedy (denoted here by PG-SF) studied by Chan, Roughgarden, and Valiant [8]. These algorithms are representative for three of the techniques used in the design of approximation algorithms for the Steiner forest, primal-dual, greedy, and probabilistic approximation by tree metrics. For the primal-dual technique, we selected the standard primal-dual algorithm (PD-SF) by Agrawal, Klein, and Ravi [1] because all the LP-based algorithms (the iterative-rounding approach [22], the LCR-based primal-dual algorithm [27], and the BCR-based primal-dual algorithm [11]) have the same approximation ratio of 2, and the PD-SF algorithm is expected to be more practical than the other three algorithms. The algorithm based on iterative rounding [22] first needs to obtain the optimal solution for the LP relaxation, which is computationally demanding. The LCR-based primal-dual algorithm [27] obtains a costlier solution than the standard primal-dual

algorithm (PD-SF) in most of the cases, as already pointed out by the authors. The BCR-based primal-dual algorithm [11] basically executes the standard primal-dual algorithm twice, which makes it costlier than PD-SF. Although Groß et al. [18] showed that their algorithm runs in polynomial time, its implementation is quite complex and not practical, therefore, we decided not to implement the local search algorithm for our experimental analysis. The source code implementing the four algorithms considered for our experiments was released as part of the SFlib, a project available on Github [16].

Since we could not find any publicly available benchmark suites for the Steiner forest, we created a new library, called SteinForestLib, consisting of approximately 1200 Steiner forest problem instances, and made it publicly available [36]. To create the library, we used some of the test sets from SteinLib [37], a library for the Steiner tree, and converted them to Steiner forest instances. As most of the instances from SteinLib are sparse graphs, we also created some denser graph instances to have a more comprehensive set of instances including both sparse and dense graphs. We give more details of each test set in Section 3.1. We used the newly created benchmark instances for Steiner forest to drive our experiments. In order to validate the solutions obtained by the algorithms, we find the optimal solutions for a subset of small-size instances by solving the flow-based integer program formulations corresponding to those instances. We compare the costs of the optimal solutions with those of the solutions obtained by the algorithms for the subset of small-size instances.

The main findings from our experimental analysis are as follows. For the vast majority of instances considered in the experiments, the primal-dual algorithm, PD-SF, is substantially faster than the other algorithms while providing solutions of comparable quality. We also observed that the algorithms based on more refined techniques do not necessarily lead to better results. For example, PG-SF, a very simple algorithm with no theoretical approximation guarantees, obtained very good solutions which are on par with those obtained by PD-SF. On the other hand, TM-SF which has a theoretical approximation guarantee of $O(\log n)$ is not competitive with PG-SF in terms of the quality of solutions. Our results also show that the number of terminals has a stronger effect on PG-SF and GL-SF than on TM-SF and PD-SF. Therefore, for practical purposes, we recommend using PD-SF.

1.3 | Organization

The rest of the article is organized as follows. In Section 2, we briefly present the algorithms we consider for our study and describe their implementations. In Section 3, we present our experimental analysis of the approximation algorithms. In Section 4, we conclude the article and present possible directions for future research.

2 | ALGORITHMS

In this section, we describe the four algorithms for Steiner forest that we implement and investigate in our experimental study.

2.1 | The primal-dual algorithm for Steiner forest (PD-SF)

The 2-approximation algorithm for Steiner forest proposed by Agrawal, Klein, and Ravi [1] is based on a primal-dual schema in which the dual variables are increased in a synchronized manner. In order to describe the algorithm we employ the notation from [42]. We denote by $\mathcal{S}_i$ the subsets of vertices separating terminals s_i and t_i, that is, $\mathcal{S}_i = \{S \subseteq V : |S \cap \{s_i, t_i\}| = 1\}$, and by $\delta(S)$ the set of edges crossing the cut $(S, \overline{S})$ in G. The cut-based integer program formulation of the problem is as follows:

$$\text{minimize} \quad \sum_{e \in E} w_e x_e \tag{1}$$

subject to:

$$\sum_{e \in \delta(S)} x_e \geq 1, \quad \forall S \subseteq V : S \in \mathcal{S}_i \text{ for some } i \tag{2}$$

$$x_e \in \{0, 1\}, \quad \forall e \in E. \tag{3}$$

The first constraints guarantee that for any cut $(S, \overline{S})$ with $s_i \in S$ and $t_i \in \overline{S}$ or vice versa, one edge is selected from $\delta(S)$. The dual of the linear program relaxation of the above integer program is

$$\text{maximize} \quad \sum_{S \subseteq V : \exists i, S \in \mathcal{S}_i} y_S \tag{4}$$

subject to:

$$\sum_{S : e \in \delta(S)} y_S \leq w_e, \forall e \in E \tag{5}$$

$$y_S \geq 0, \exists i : S \in \mathcal{S}_i. \tag{6}$$

Algorithm 1. PD-SF [Agrawal, Klein, and Ravi [1]]

Initialization: $y \leftarrow 0$; $F \leftarrow \emptyset$;
while not all (s_i, t_i) pairs are connected in (V, F) **do**
  Increase y_C for all C in $\mathcal{C}$ uniformly until for some $e \in \delta(C')$, $C' \in \mathcal{C}$, $w_e = \sum_{S: e \in \delta(S)} y_S$
  $F \leftarrow F \cup \{e\}$
end while
$F' \leftarrow \text{Prune}(F)$
return F'

A natural interpretation of dual variable y_S is to consider it as the cost of the subset S that needs to be paid if subset S is picked. Dual variables can also be viewed as *moats* of width y_S surrounding subset S. Selecting any edges from $\delta(S)$ to connect a terminal pair means crossing the moat corresponding to y_S and paying at least y_S. Since the moats cannot overlap, an edge e cannot cross moats having the sum of the widths greater than w_e. This corresponds to the first constraint in the dual.

We now describe the primal-dual algorithm for Steiner forest (given in Algorithm 1). We refer to this algorithm as PD-SF. PD-SF starts with dual feasible solution $y = 0$ and primal infeasible solution $F = \emptyset$. We denote by $\mathcal{C}$ the set of all connected components C of (V, F) such that $|C \cap \{s_i, t_i\}|=1$. PD-SF increases the dual variables y_C uniformly for all the connected components $C \in \mathcal{C}$ until a dual inequality corresponding to some edge $e \in \delta(C)$ becomes tight. Then, it adds e to the solution and continues until all pairs are connected. The solution F may contain edges that are redundant, that is, edges that can be pruned from the solution F without affecting its feasibility. PD-SF prunes those redundant edges in the last step. For this, PD-SF repeatedly removes an edge from the solution F and checks whether all pairs are still connected, that is, the solution is still feasible. If the removal of the edge leads to an infeasible solution the edge is added to the final solution F', otherwise it is not added to F'. PD-SF is a 2-approximation algorithm for Steiner forest as proved by Agrawal, Klein, and Ravi [1]. Next, we describe our implementation of PD-SF.

PD-SF **implementation.** There are two main issues that need to be addressed in order to efficiently implement PD-SF: (i) maintaining the connected components; and (ii) maintaining the costs of the edges so that finding the minimum weighted edge can be performed fast.

To maintain the connected components efficiently, we construct a trivial infeasible solution that places each vertex in its own singleton component (moat) and marks it as active or nonactive, based on being a terminal or not. Furthermore, each component has a label associated with it, which is the lowest index vertex in the component. For the trivial infeasible solution, each component's label is its own vertex's index. This information is kept in each vertex, for example, each vertex knows if it is in an active or nonactive component and knows its component label. Furthermore, each vertex has an associated growth rate. If a vertex is included in an active moat, its growth rate will be increased by the global growth rate which is the rate that is used to increase all moats. We use this to update the reduced cost of the remaining edges. In order to implement it efficiently, we define three functions. These functions map each vertex to its label, status, and growth, respectively. This let us to find the connected components in constant time, and merge them in $O(n)$.

In order to compute the reduced costs of edges in an efficient way, we use the functions described above. For each edge (i, j) we have constant time access to its endpoints' growth, status, and label, which let us compute the reduced cost of the edge in constant time. Therefore, we can find the minimum edge to add to the solution in $O(m)$ by going through all uncovered edges that have at least one active endpoint. After we found the minimum weight to growth moats, we increase the growth rate for all active vertices. Note that the running time of this procedure is decreasing as the list of uncovered edges reduces by adding edges to the solution. This yields a time bound of $O(m)$ per iteration, and $O(mn)$ for the entire loop.

2.2 | The paired greedy algorithm for Steiner forest (PG-SF)

We introduce the first greedy algorithm for the Steiner forest problem, called the *paired greedy* algorithm. In the rest of the article we refer to this algorithm as PG-SF. This is a simple greedy algorithm that repeatedly picks the terminal pair (s_i, t_i) which has the minimum distance between the terminals and connects them until all pairs are connected. The terminals of the selected pair are connected using the shortest path between them. The weights of the edges on the shortest path between the terminals of the selected pair are set to zero so that in the next iterations they can be considered in the computation of the shortest paths for the remaining pairs. As in the case of PG-SF, the solution F may contain edges that are redundant, that is, edges that can be pruned from the solution F without affecting its feasibility. PG-SF prunes those redundant edges in the last step using the same

Algorithm 2. PG-SF

1: $F \leftarrow \emptyset$
2: **while** not all terminals are connected **do**
3: Find the terminal pair (s_i, t_i) with minimum distance
4: Add to F the edges on the shortest path between s_i and t_i
5: Set the weights of the edges on the shortest path between s_i and t_i to zero
6: **end while**
7: $F' \leftarrow \mathsf{Prune}(F)$
8: **return** F'

procedure as in PD-SF. The PG-SF algorithm is presented in Algorithm 2. Chan, Roughgarden, and Valiant [8] showed that the solution obtained by PG-SF has cost $\Omega(\log n)$ times that of the optimal solution, where n is the number of vertices in the graph.

PG-SF **implementation.** We now describe the details of our implementation of the PG-SF algorithm for the Steiner forest problem. In each iteration, the algorithm connects the terminal nodes of the terminal pair that has the minimum distance between them. For this, we need to compute the shortest distance between each terminal pairs. The number of terminals is much smaller than n, and the number of unconnected terminal pairs, k, is decreasing as the algorithm connects at least one terminal pair in each iteration. Therefore, we do not use an all-shortest paths algorithm in each iteration to find the closet terminal pair, but instead, we employ a single source shortest paths algorithm for each terminal pair to find the terminal pair with the minimum distance. We choose Dijkstra's shortest paths algorithm and implemented it using a priority queue. As the graph is represented using adjacency lists, the time complexity of Dijkstra's algorithm is $O(m \log n)$. Since in each iteration Dijkstra's algorithm is executed at most k times this leads to a time complexity of $O(km \log n)$ for each iteration. The total number of iterations of the main loop is at most k. Therefore, the overall complexity of our PG-SF implementation is $O(k^2 m \log n)$.

2.3 | The gluttonous algorithm for Steiner forest (GL-SF)

The gluttonous algorithm is a greedy constant approximation algorithm proposed by Gupta and Kumar [19]. In the rest of the article, we refer to this algorithm as GL-SF. We first introduce the notation and terminology from Gupta and Kumar [19] and then describe the algorithm. Given a Steiner forest instance, a *supernode* S_i is a subset of terminals, and a *clustering*, $\mathscr{C} = \{S_1, S_2, \ldots, S_q\}$ is a partition of the set of terminals into supernodes. We call a terminal *active* if it belongs to a supernode that does not contain its mate. Consequently, a supernode is *active* if it contains an active terminal. A clustering is *trivial* if each of the terminals is a supernode of the clustering. All supernodes and terminals in a trivial clustering are active.

Given a clustering $\mathscr{C} = \{S_1, S_2, \ldots, S_q\}$, the $\mathscr{C}$-punctured distance is defined as follows. We consider the complete graph $G_{\mathscr{C}}$ on the set of vertices V and set the length $d(u, v)$ of edge (u, v) in this graph to the shortest path in the original graph G, if u and v lie in different supernodes of $\mathscr{C}$, and to zero, if they lie in the same supernode. The $\mathscr{C}$-punctured distance denoted by $d_{\mathscr{C}}$, is defined as the shortest path in $G_{\mathscr{C}}$. The distance between two supernodes S_1 and S_2 can now be defined as

$$d_{\mathscr{C}}(S_1, S_2) = \min_{u \in S_1, v \in S_2} d_{\mathscr{C}}(u, v).$$

We present the GL-SF algorithm in Algorithm 3. GL-SF starts with the trivial clustering $\mathscr{C}$, and with $F = \emptyset$. It iteratively, tries to connect the terminals by connecting the supernodes. In each iteration it chooses the closest supernodes S_1 and S_2, that is, those with the minimum $\mathscr{C}$-punctured distance, and merges them. Then, it updates the clustering $\mathscr{C}$ with the new merged supernode $S_1 \cup S_2$ and removes supernodes S_1 and S_2 from it. Finally, it adds the edges corresponding to the intersupernode edges on the shortest path between S_1 and S_2 in the graph $G_{\mathscr{C}}$. GL-SF terminates when there are no more active terminals. As in the case of the previous two algorithms, the solution F may contain edges that are redundant. GL-SF prunes those redundant edges in the last step using the same procedure as in the case of PD-SF. Gupta and Kumar [19] showed that GL-SF is a 96-approximation algorithm for Steiner forest. Next, we describe our implementation of GL-SF.

GL-SF **implementation.** There are three main issues that we need to address when implementing GL-SF, maintaining the supernodes, finding the minimum distance between the supernodes, and merging them. To address these issues, we design data structures for terminals, supernodes, and clusters, that lead to an efficient implementation of the algorithm. The algorithm needs to determine the status of a terminal by checking if its mate belongs to the terminal's supernode. Thus, it is beneficial to consider each terminal as an instance of a class, called terminal, that has as data members a triple composed of the index of the terminal, the index of the terminal's mate, and the status of the terminal.

Algorithm 3. GL-SF [Gupta and Kumar [19]]

$\mathscr{C} \leftarrow$ a trivial clustering
$F \leftarrow \emptyset$
while there exist active supernodes in $\mathscr{C}$ **do**
    Find active supernodes S_1, S_2 in $\mathscr{C}$ with minimum $\mathscr{C}$-punctured distance.
    $\mathscr{C} \leftarrow (\mathscr{C} \setminus \{S_1, S_2\}) \cup \{S_1 \cup S_2\}$
    Add to F the edges corresponding to the inter-supernode edges on the shortest path between S_1, S_2 in the graph $G_{\mathscr{C}}$
end while
$F \leftarrow$ Prune(F)
return F

As mentioned above, a supernode is a subset of terminals that is active or not. Similarly, each supernode is an instance of a supernode class that contains a list of terminals and its status (active or nonactive). Since GL-SF needs to traverse the list of terminals of a supernode several times in an iteration, we implement the list as an associative container that contains key-value pairs with unique keys (i.e., unordered_map container from the C++ STL library). Therefore, each supernode is a map of terminal indices along with their mates' indices and its status. Similarly, clustering is a set of supernodes, and therefore, it is implemented as an associative container. These structures make the procedure of merging the supernodes and updating the clusters efficient.

Following these ideas, the main loop of the algorithm is implemented as follows. It starts by constructing the complete graph $G_{\mathscr{C}}$, as mentioned in the description of the algorithm. For that it finds all shortest paths on the vertex set V by using the Floyd–Warshall algorithm (running time, $O(n^3)$). Then it finds two supernodes with the minimum distance, adds the edges on the minimum shortest path between them to the solution, and zeros them out. After this is completed it merges the two supernodes into a single one. The last step is to update the status of each terminal and supernode. Since each terminal knows its mate index and they are listed as a map in each supernode, it is easy to check if each terminal mates are in the same supernode or not in $O(n^2)$, since there are at most n supernodes, each containing at most n terminals. It is worth pointing out that in each iteration the algorithm connects at least two supernodes and thus, the number of supernodes decreases as the algorithm executes. The algorithm terminates when there is no active supernode. The total number of iterations of the main loop is at most n. Therefore, the overall complexity of our GL-SF implementation is $O(n^4)$.

2.4 | The randomized tree metric-based algorithm for Steiner forest (TM-SF)

In this section, we use the method of probabilistic approximation of metrics by tree metrics proposed by Fakcharoenphol, Rao, and Talwar [15] to obtain a randomized $O(\log n)$-approximation algorithm for the Steiner forest problem. Since the Steiner forest is a special case of the buy-at-bulk network design problem, we adapt and engineer the algorithm for buy-at-bulk network design proposed by Awerbuch and Azar [3] and make it suitable for the Steiner forest. In the rest of the article we refer to this algorithm as TM-SF.

First, we review some key concepts and results about tree metrics. Let d be a given distance metric on a set of vertices V, that is, d_{uv} is the distance between u and v, where $u, v \in V$. We would like to find a tree metric that approximates d. A *tree metric* (W, T) for a set of vertices V is a tree T defined on some set of vertices $W \supseteq V$, where each edge of T has a length. The distance between the vertices $u \in W$ and $v \in W$, denoted by T_{uv}, is given by the length of the unique shortest path in T between the two vertices. The goal is to obtain a tree metric (W, T) that approximates d on the original set of vertices V such that $d_{uv} \leq T_{uv} \leq \alpha \cdot d_{uv}$ for all $u, v \in V$, where α is the *distortion of the embedding* of d into the tree metric (W, T). The factor α characterizes the loss in performance guarantee when we reduce a problem originally defined on general metric spaces to a corresponding problem on tree metrics. The main reason of reducing a problem to another problem in tree metrics is that it is much easier to find an algorithm for solving the problem in tree metrics than in general metrics. Unfortunately, it is not possible to find a deterministic algorithm with low distortion, instead, we rely on a randomized algorithm. Fakcharoenphol, Rao, and Talwar [15] proposed such a randomized algorithm and proved that given a distance metric (V, d) the algorithm produces a tree metric (W, T) such that for all $u, v \in V$, $d_{uv} \leq T_{uv}$ and $E[T_{uv}] \leq \alpha \cdot d_{uv}$, where $\alpha = O(\log n)$. The randomized algorithm that produces the tree metric constructs the tree via a hierarchical cut decomposition of the metric d. The resulting tree has $\log_2 \Delta + 1$ levels, where Δ is the smallest power of two greater than $2\max_{u,v} d_{uv}$.

Algorithm 4. HTD: Hierarchical tree decomposition

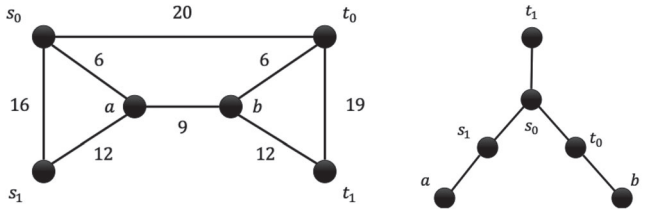
1: $\sigma \leftarrow$ random permutation of vertices of G
2: $\Delta \leftarrow$ the smallest power of 2 greater than $2 \max_{u,v} d_{uv}$
3: $r_0 \leftarrow$ random number from [1/2, 1)
4: **for** $i = 1$ to $\log_2 \Delta$ **do**
5: $\quad r_i \leftarrow 2^i r_0$
6: **end for**
7: $\mathcal{L}[\log_2 \Delta] = V$, where $\mathcal{L}[i]$ is the set of nodes at level i
8: Create the root node of the tree corresponding to all vertices, V
9: **for** $i = \log_2 \Delta$ down to 1 **do**
10: $\quad \mathcal{L}[i-1] \leftarrow \emptyset$
11: $\quad$ **for all** $L \in \mathcal{L}[i]$ **do**
12: $\quad\quad C \leftarrow L$
13: $\quad\quad$ **for** $j = 1$ to n **do**
14: $\quad\quad\quad$ **if** $(B(\sigma(j), r_{i-1}) \cap C \neq \emptyset)$ **then**
15: $\quad\quad\quad\quad \mathcal{L}[i-1] \leftarrow \mathcal{L}[i-1] \cup \{B(\sigma(j), r_{i-1}) \cap C\}$
16: $\quad\quad\quad\quad C \leftarrow C \setminus \{B(\sigma(j), r_{i-1}) \cap C\}$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad$ **end for**
19: $\quad\quad$ Create a node for each set in $\mathcal{L}[i-1]$ that is a subset of L
20: $\quad\quad$ Join each of these nodes to the node corresponding to (L) byan edge of length 2^i
21: $\quad$ **end for**
22: **end for**

To describe the randomized algorithm we follow Williamson and Shmoys [42]. The algorithm, called HTD, is given in Algorithm 4. HTD begins by generating a random permutation of the vertices list and builds the tree level by level. Each node in the tree corresponds to a ball centered on one vertex with radius less than 2^i and at least 2^{i-1} at level i. Nodes at level zero are leaves with a single vertex, since each ball has radius $2^0 = 1$ centered on each vertex. The children at level $i-1$ are joined to their parent at level i by an edge of length 2^i.

The algorithm constructs the tree as follows. It starts by generating a random permutation σ of vertices V (Line 1). Let $B(u, r)$ be the set of vertices at distance no greater than r from vertex u, that is, $B(u, r) = \{v \in V : d_{uv} \leq r\}$. Then the root node, at level $l = \log_2 \Delta$ is a ball of radius less than $r_l = 2^l = \Delta$; therefore, it corresponds to V itself. The nodes at each level of the tree correspond to some partitioning of the vertex set V. We denote by $\mathcal{L}[i]$ the set of nodes at level i. Starting with the root node, the algorithm creates the tree level by level (Lines 9–22). At each level i, a given node corresponds to some subset $C \subseteq V$ whose vertices are the vertices in the ball of radius less than 2^i and at least 2^{i-1} centered on some vertex. Therefore at level zero, each leaf is in a ball of radius less than $2^0 = 1$ centered on some vertex. Thus, each leaf corresponds to a single vertex of V. The algorithm joins the children at level $i-1$ to their parent at level i with an edge of length 2^i.

In order to illustrate how the hierarchical cut decomposition is performed, we apply the algorithm on a small Steiner forest instance taken from [40]. The graph has six vertices, $V = \{s_0, t_0, s_1, t_1, a, b\}$ with two terminal pairs (s_0, t_0) and (s_1, t_1). The graph is shown in Figure 1(A). Suppose that the random permutation is $\sigma = \langle s_1, t_0, a, s_0, t_1, b\rangle$, and $r_0 = \frac{1}{2}$. Since $\log_2 \Delta$ is 7, following the algorithm, the set of nodes at level 7 is V. On the next level, the ball centered on $\sigma[1] = t_0$ contains $\{s_0, t_0, a, b, s_1\}$ since their distance to t_0 is less than $r_6 = 2^5$, and the ball centered on $\sigma[6] = t_1$ contains only $\{t_1\}$. Now, at the next level the algorithm partitions the subsets $\{s_0, t_0, a, b, s_1\}$ and $\{t_1\}$. This procedure is repeated until level zero, when all leaves are single vertices from V. The hierarchical cut decomposition of the graph is shown in Figure 2.

We observe that if we had a tree metric (W, T) with possibly $W = V$, it would have been easier to solve the problem on a tree metric rather than on a general metric space. Unfortunately, in the obtained tree metric (W, T), we have $V \subseteq W$ and we need to translate the result of the tree metric (W, T) back into the original graph. Konjevod, Ravi, and Salman [28] showed that given any tree metric (V', T) with $V \subseteq V'$ defined by a hierarchical cut decomposition, with the vertices in V as the leaves of T, it is possible to find in polynomial time another tree metric (V, T'), on the same set of vertices as in the original graph, such that $T_{uv} \leq T'_{uv} \leq 4T_{uv}$. We call such an algorithm, the *tree shrinking* (TS) algorithm. TS creates a new node by merging each vertex $v \in V$ with its parent w, if $w \in V' - V$ and identifies it as v. The algorithm terminates when no vertex in the tree is in $V' - V$. Considering the tree in Figure 2, and applying TS, we obtain the tree shown in Figure 1(B).



(A) Steiner Forest instance. (B) The tree obtained by TS.

FIGURE 1 A Steiner forest instance and the resulting tree after applying HTD and TS

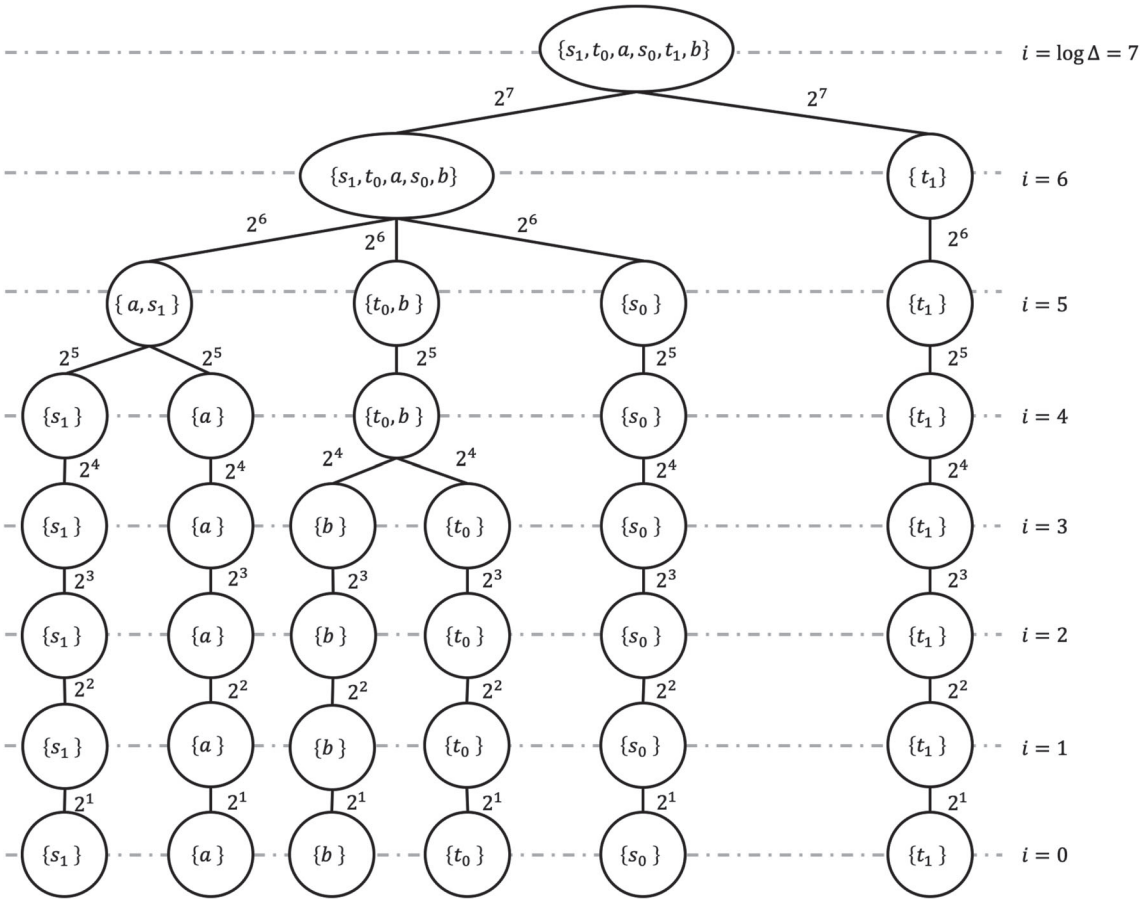


FIGURE 2 An example of a hierarchical tree decomposition

The TM-SF algorithm is given in Algorithm 5. The algorithm starts by using HTD to obtain a hierarchical tree decomposition of the graph, that is, the tree metric (W, T) with $V \in W$, followed by TS to convert the tree metric (W, T) into the tree (V, T') (Lines 2 and 3). Now, that we have an approximately equivalent tree, solving the Steiner forest problem for it is much easier than for the original graph. Next, TM-SF converts the solution back to the original graph (Lines 4–12). This is achieved by finding the unique path P'_i for each terminal pair (s_i, t_i) in T' (Line 6) and then, converting this unique path to a path in the original graph G. That is, for each edge (u, v) in P'_i, TM-SF finds the shortest path P_{uv} between u and v, and adds it to the solution F.

TM-SF **implementation.** We now give more details on our implementation of TM-SF. The implementation consists of two main procedures. The first one builds the hierarchical tree decomposition and corresponds to HTD. The implementation of this procedure is based on the breadth first search (BFS) algorithm. In order to form the balls, the procedure considers the shortest path matrix, the random permutation of the vertices σ, and the set of radius r. It creates the tree level by level and returns a tree metric with the set of vertices W. The second main procedure, corresponding to TS, transforms the tree metric with vertices W obtained by the hierarchical cut decomposition into another tree metric with vertices V. This procedure is also based on the BFS algorithm. It starts with the root node and iteratively merges the subtree at node v with its parent w if $w \in W - V$. This is repeated until all vertices in the updated tree metric form the initial set of vertices V. After these procedures are completed, the algorithm solves the problem in the new metric. For this we implemented a procedure that finds the unique $s_i - t_i$ path in T'

Algorithm 5. TM-SF

```
Initialization: F ← ∅;
(W, T) ← HTD(G);
(V, T′) ← TS(W, T);
for i = 0 to k do
    P_i ← ∅;
    P′_i ← the unique path for each terminal pair (s_i, t_i) in T′
    for all edges (u, v) ∈ P′_i do
        P_uv ← the shortest path u − v in G
        P_i ← P_i ∪ P_uv
    end for
    F ← F ∪ P_i
end for
return F
```

for all i and then determines the output path from s_i to t_i in the original metric as the concatenation of the paths P_{xy} for all edges $(x, y) \in T'$ on the path from s_i to t_i in the tree metric T'.

3 | EXPERIMENTAL ANALYSIS

In this section, we investigate the performance of the four algorithms by performing extensive experiments on a multicore system. We implement PD-SF, PG-SF, GL-SF, and TM-SF and compare their performance in terms of running time and cost of the solution. We also compare the cost of the solutions obtained by the algorithms against the optimal solutions obtained by solving the flow-based IP formulation of the problem for a set of small-size instances.

3.1 | Experimental setup

We used C++ without any external graph library to implement the algorithms and compiled the codes with the GNU g++ compiler version 4.8.5. The experiments were conducted on a 64-bit Linux (Ubuntu) machine with the following configuration: 2.4 GHz AMD Opteron(tm) Processor 6378, 512 GB of RAM, 6144 KB L3 cache, and private 2048 KB L2 cache per core. We measured the running time using the gettimeofday function. The code implementing the algorithms was executed on a single core without using any parallelization. The machine on which we run our experiments has a large amount of RAM, but the code implementing the algorithms used at most one percent of the machine's total memory (approximately 5 GB).

To evaluate the algorithms, we created a library of Steiner forest instances, called SteinForestLib, and made it publicly available [36]. We created the Steiner forest instances from the Steiner Tree problem instances available in the SteinLib library [37]. In addition to the instances created from SteinLib, which are mainly sparse graphs, we created several instances consisting of dense graphs and added them to the SteinForestLib. This allows us to experiment with a wide range of instances and perform a more systematic study on the performance of the algorithms. The SteinForestLib instances and their characteristics are given in Table 1.

We first introduce the notation that we use to characterize the instances and then describe the instances in the library in more details. Besides the key characteristics of the problem instances such as number of vertices, n, number of edges, m, and number of terminals, k, we also consider two metrics, graph density and terminal coverage, to categorize the instances and analyze the results. We define the *density* d of a graph with n vertices, as the ratio of the number of edges in the graph and the maximum possible number of edges for the graph, that is, $d = m/\binom{n}{2}$. We also define the *terminal coverage*, c, as the ratio of the total number of terminals and the number of vertices, that is, $c = k/n$.

In the following, we describe the types of instances we created and included in the new library of Steiner forest instances, SteinForestLib. The first six groups of instances are derived from the SteinLib library [37]. They are obtained by randomly selecting a list of disjoint terminal pairs from the original terminal sets for each instance. For the instances having an odd number of terminals, we randomly added one more terminal node.

(i) Sparse graphs. This group contains sparse graphs with a maximum density of 20%. The instances are randomly generated graphs with different edge weight types such as random, incidence, and Euclidian. Test sets B, C, D, and E were introduced in [5] and are generated based on the scheme outlined in [2]. They are randomly generated sparse graphs with edge weights

TABLE 1 Characteristics of SteinForestLib instances (n is the number of vertices, m is the number of edges, k is the number of terminals, d is the graph density, and c is the terminal coverage)

Group	Weights	Test set	# of instances	n	m	k	d (%)	c (%)
Sparse	Random	B	18	50–100	63–200	10–50	3–8	18–52
		C	20	500	625–12 500	6–250	0.5–1	1–50
		D	20	1000	1250–25 000	6–500	0.3–5	1–50
		E	20	2500	3125–62 500	6–1250	0.1–2	0.2–50
		P6Z	15	100–200	180–370	6–100	1.9–3.6	5–50
		MC	3	400	760	170–214	1	42–53
	Incidence	I080	100	80	120–3160	6–20	3.8–20	8–25
		I160	100	160	240–12 720	8–40	1.9–20	4.4–25
		I320	100	320	480–51 040	8–80	0.9–20	2.5–25
		I640	100	640	960–204 480	10–160	0.5–20	1.4–25
	Euclidian	P6E	15	100–200	180–370	6-100	1.9–3.6	5–50
Complete	Random	MC	3	97–150	4656–11 175	46–80	100	47–53
		P4Z	10	100	4950	6–50	100	50
	Euclidian	X	3	52–666	1326–222 446	16–174	100	26–43
		P4E	11	100–200	4950–19 900	6–100	100	5–50
Constructed	Unit	SP	8	6–3997	9–10 278	4–2284	0.1–71.4	31–57
Grid/VLSI	Fixed	ALUE	15	940–34 479	1474–55 494	16-2344	0.01–0.3	0.2–6.8
		ALUT	9	387–36 711	626–68 117	34–879	0.01–0.8	0.6–8.8
		DIW	21	212–11 821	381–22 516	10–50	0.01–1.7	0.3–5.2
		LIN	37	53–38 418	80–71 657	4–172	0.01–5.8	0.1–14.0
Rectilinear	L_1	ES10FST	15	12–24	11–32	10	11.6–16.7	42–83
		ES20FST	15	27–57	26–83	20	5.2–7.4	35–74
		ES30FST	15	43–118	44–188	30	2.7–4.9	25–70
		ES40FST	15	55–121	55–180	40	2.5–3.7	33–73
		ES50FST	15	83–143	96–211	50	2.1–2.8	35–60
		ES60FST	15	109–188	133–280	60	1.6–2.3	32–55
		ES70FST	15	142–209	181–314	70	1.4–1.8	34–49
		ES80FST	15	147–236	180–343	80	1.2–1.7	34–54
		ES90FST	15	175–284	221–430	90	1.1–1.5	32–51
		ES100FST	15	188–339	233–522	100	0.9–1.4	30–53
		ES250FST	15	542–713	719–1053	250	0.4–0.5	35–46
		ES500FST	15	1172–1477	1627–2204	500	0.2	34–43
		ES1000FST	15	2532–2984	3615–4484	1000	0.1	34–40
Wire routing	Real	WRP3	64	84–3168	149–6220	12-100	0.1–4.3	2.6–24.9
		WRP4	62	110–1898	188–3616	12-76	0.2–3.1	3.8–16.0
Erdős–Rényi	Random	SSRG	6	50–300	375–13 404	10–60	30	20
		SMRG	6	400–650	23 823–63 334	80–130	30	20
		SLRG	6	750–1000	84 260–149 593	150–200	30	20
		MSRG	6	50–300	729–26 851	10-60	60	20
		MMRG	6	400–650	47 601–126 258	80–130	60	20
		MLRG	6	750–1000	168 267–299 605	150–200	60	20
		DSRG	6	50–300	1089-40 358	10–60	90	20
		DMRG	6	400–650	71 834–190 009	80–130	90	20
		DLRG	6	750–1000	252 894–449 880	150–200	90	20

between 1 and 10. The number of edges and the number of terminals are derived from the number of vertices, n, as follows $m \in \{1.25n, 2n, 5n, 25n\}$, and $t \in \{\frac{1}{100}n, \frac{1}{50}n, \frac{1}{6}n, \frac{1}{4}n, \frac{1}{2}n\}$. The number of vertices ranges from 50 to 2500. Test sets P6Z and MC are also sparse graphs with random weights that were first introduced in [10]. Test sets I080, I160, I320, and I640 are random sparse graphs with incidence weight edges that were originally generated by Duin [13] with the intention to be difficult problems for preprocessing of the instances by removing the unnecessary vertices and edges from the graph. Each test set contains 20 different instances with various number of edges and number of terminals. They are categorized based on the number of vertices, given as a number in each test set name. Test set P6E consists of sparse graphs with Euclidian weights. The Euclidian weight of an edge is the Euclidian distance between its end vertices that are placed at random on the plane.

(ii) Complete graphs. Test sets P4E and P4Z, are complete graphs with Euclidian and random weights that were first introduced in [10]. Test set MC consists of complete graphs with random edge weights, while test set X consists of complete graphs with Euclidian weights. More details of each test set are presented in Table 1.

(iii) Constructed graphs. Test set SP consists of artificially constructed graphs that have unit weights. Each instance is a combination of either a fixed number of wheels and cycles, or both.

(iv) Grid/VLSI graphs. This group consists of 82 instances classified into four different test sets, ALUE, ALUT, DIW, and LIN. These instances are very sparse grid graphs derived from placement of rectangular blocks in a 2D plane, thus, all line segments are either horizontal or vertical. They are actual instances from a VLSI application [24].

(v) Rectilinear graphs. This group consists of 195 instances classified into 13 different test sets. The instances consist of planar graphs obtained by converting the random terminal points in the plane into rectilinear graphs with rectilinear (L1) edge weights, by building the Hanan grid [20]. The edge weights are rectilinear distances, where the *rectilinear distance* between two points p_1 and p_2 is defined as $|x_1 - x_2| + |y_1 - y_2|$, where (x_i, y_i) are the coordinates of p_i. The instances are categorized into 13 test sets based on the number of terminals from 10 to 1000 terminals. They have been used in the literature to evaluate algorithms for planar Steiner tree [38].

(vi) Wire routing. This is the last group of instances derived from SteinLib. It consists of 126 instances of VLSI wire-routing problems from IBM chips (test sets WRP3 and WRP4).

(vii) Erdős–Rényi graphs. As we mentioned above, the instances derived from the SteinLib library do not cover the whole range of graph densities. They do not contain any graphs with density between 20% and 100%. Furthermore, most of them are not organized well to be able to perform a systematic analysis of the effect of changing the instance parameters on the performance of the algorithms. Therefore, for the sake of completeness, we constructed several new instances for the Steiner forest problem and added them to the new SteinForestLib library. We generated the instances based on the Erdős–Rényi model [14]. We considered different number of vertices $n = 50, 100, \ldots, 1000$, and for each number of vertices, we randomly generate edges with probability $p = 0.1, \ldots, 0.9$. The generated random graphs range from very sparse graphs with $p = 0.1$, to very dense graphs with $p = 0.9$. For each random graph, we selected 20% of the vertices of each connected component as terminals. We then randomly paired these selected terminals to form disjoint terminal pairs. The edge weights were drawn from the uniform distribution on the interval $[1, 100]$. We constructed a total of 180 instances. For reasons of space, we picked a subset of these instances to present our analysis of the results, and eliminated those that exhibit similar behavior. We categorize these instances into three groups: small ($50 \leq n \leq 300$), medium ($400 \leq n \leq 650$), and large ($750 \leq n \leq 1000$). For all instances, we generate random graphs with $p = 0.3, 0.6$, and 0.9 corresponding to sparse, medium sparse, and dense graphs, respectively. The characteristics of the instances are presented in Table 1.

3.2 | Analysis of results

To characterize the performance of the algorithms, we perform four types of experiments. We first analyze the algorithms based on the type of the underlying graph as presented in Table 1, then we analyze in more detail the performance of the algorithms by changing various graph parameters. In the next two types of experiments, we investigate the effect of the number of edges and the number of terminals on the running times and the costs of solutions obtained by the algorithms. We also analyze the correlation between the graph parameters and the running time of the algorithms for each algorithm. In the fourth type of experiments, we consider a set of small-size instances and compare the cost of solutions obtained by the four algorithms against the cost of the optimal solutions obtained by solving the flow-based integer program formulations corresponding to those instances.

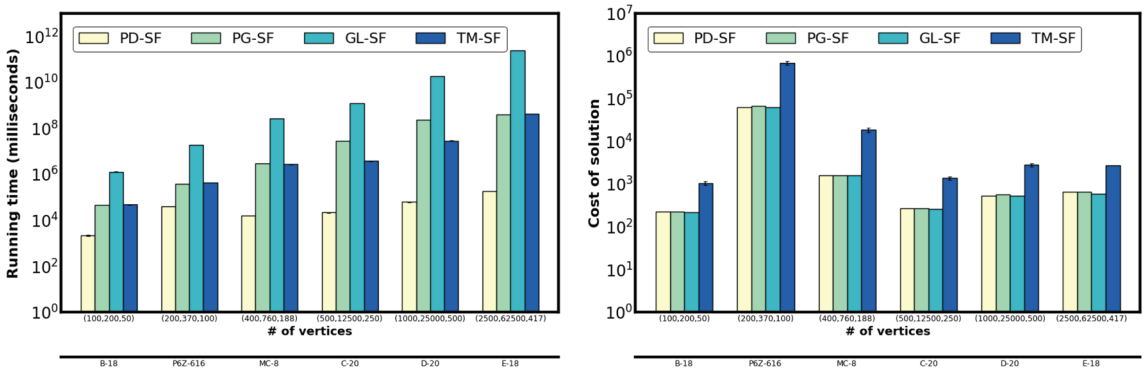
We conducted five independent runs for each instance. The results presented on the plots are averaged over the five independent runs. We also calculated the standard deviation and represented it on the plots. The standard deviation is very small in the case of PD-SF, PG-SF, and GL-SF, and reasonably small in the case of TM-SF. Being very small, the standard deviation is not visible on some of the plots.

3.2.1 | Performance with respect to the graph type

For this set of experiments we consider the seven types of instances from the newly created library of Steiner forest instances, SteinForestLib. The characteristics of these types of instances are given in Table 1. In the following we analyze the results for each group of instances, separately.

Sparse graphs

For the sparse graphs group we consider three categories of instances with random, incidence, and Euclidian weights, respectively. All these instances have a maximum graph density of 20%. Their characteristics are presented in Table 1. In the following we analyze the performance of the algorithms on those instances.



(A) Running time (B) Cost of solution

FIGURE 3 Sparse graphs with random weights (selected instances): Running time and cost of solution (The main horizontal axis shows the key parameters of each instance as a triple (n, m, k), while the secondary horizontal axis shows the corresponding test set. The instances are listed in nonincreasing order of the number of vertices from left to right.) [Color figure can be viewed at wileyonlinelibrary.com]

Sparse graphs with random weights. We analyze the performance of the algorithms for the selected sparse graph instances with random weights. The instances are selected from test sets B, C, D, E, P6Z, and MC. We selected the largest instance from each set and presented the results in Figure 3. The main horizontal axis shows the key parameters of each instance as a triple (n, m, k), while the secondary horizontal axis shows the corresponding test set.

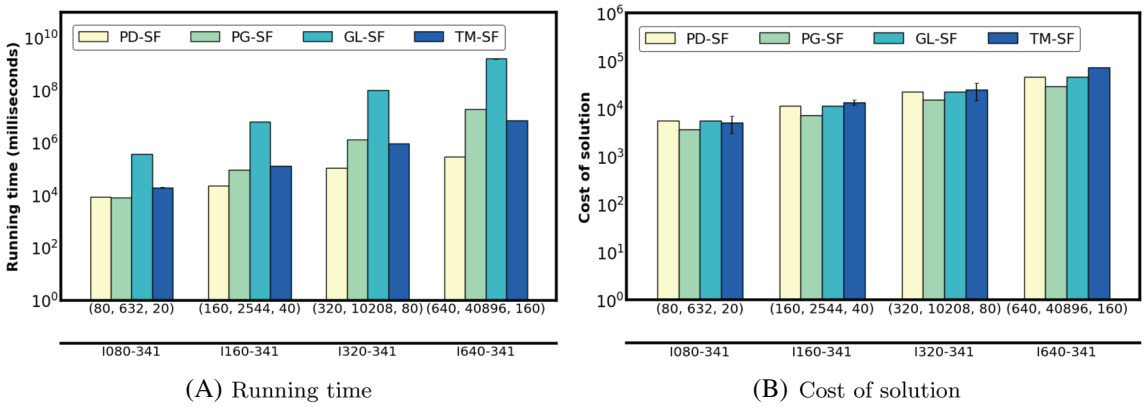
Figure 3(A) shows the running times for the selected instances. For all instances considered in this experiment, GL-SF is the slowest algorithm, while PD-SF is the fastest. The running time of PD-SF is about one to three orders of magnitude less than the second fastest algorithm, PG-SF. We observe that for some instances, such as C-20 and D-20, the running time of PG-SF is greater than that of TM-SF. This is mainly due to the fact that these specific instances have the largest number of terminals among all instances, and the running time of PG-SF is highly dependent on the number of terminals because it checks each terminal pair for inclusion in the forest one at the time. This is in contrast to TM-SF, which builds the tree metric based only on the shortest paths between vertices, and thus, its running time is not highly dependent on the number of terminal pairs. This is true for PD-SF as well, which considers all the terminals as a whole and its running time is not highly dependent on the number of terminal pairs. In Section 3.2.3, we provide a more detailed analysis of the effect of the number of terminals on the performance of the algorithms. By considering the running times for instances MC-8 and C-20, we observe that although the number of edges in C-20 is much greater than the number of edges in MC-8, we do not see any significant increase in the running time of TM-SF. Figure 3(B) shows the cost of the solution obtained by the algorithms on the selected instances. PD-SF and the two greedy algorithms, GL-SF and PG-SF, obtain solutions with almost similar costs, that are much lower than the cost of the solutions obtained by TM-SF.

Considering the quality of the solutions together with the running time of the algorithms for sparse graphs with random weights, we can say that PD-SF performs the best for instances that consists of sparse graphs with random weights.

Sparse graphs with incidence weights. We investigate the performance of the algorithms on sparse graphs with incidence weights. The instances are selected from test sets I080, I160, I320, and I640, that were originally designed to be difficult for preprocessing. We selected the largest instance from each set and presented the results in Figure 4. The behavior of the algorithms in terms of the running time (Figure 4(A)) is very similar to what we observed in the case of sparse graphs with random weights. Again, PD-SF is the fastest algorithm, while GL-SF is the slowest. By increasing the number of terminals, the running time of PG-SF surpasses that of TM-SF. On the other hand, we see a different behavior of the algorithms in terms of the quality of solution (Figure 4(B)). We observe that for this type of graphs, all algorithms obtain solutions having almost the same cost. PG-SF obtains slightly better solutions than the other algorithms. We also observe a huge improvement in the quality of solutions obtained by TM-SF compared with the case of sparse graphs with random weights. Considering the quality of the solution for sparse graphs with incidence weights, we can conclude that PG-SF performs the best and it is very suitable for use in practice when one is willing to accept a reasonable increase in the running time in order to obtain a solution of lower cost. On the other hand, PD-SF is the choice in cases where the running time is of importance.

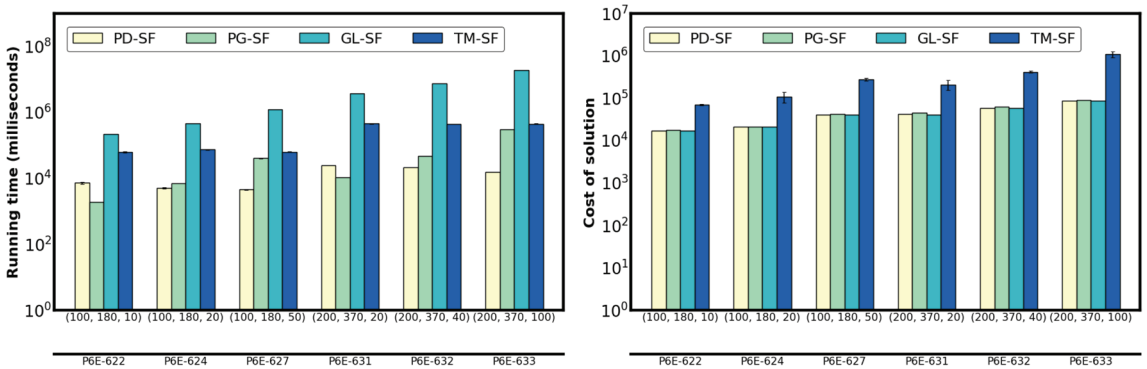
Sparse graphs with Euclidian weights. Figure 5(A,B) shows the running time and the cost of the solution for the sparse graphs with Euclidian weights from test set P6E. This set consists of instances with six combinations of number of vertices, number of edges, and number of terminals. We picked one instance from each of these combinations.

We observe almost the same behavior as in the case of sparse graphs with random weights in terms of both the running time and the cost of the solution. PD-SF is the fastest algorithm for most of the instances, except for instances with small terminal coverage (P6E-622 and P6E-631) in which PG-SF is faster than PD-SF. Again, GL-SF is the slowest algorithm, and



(A) Running time (B) Cost of solution

FIGURE 4 Sparse graphs with incidence weights (selected instances): Running time and cost of solution (The main horizontal axis shows the key parameters of each instance as a triple (n, m, k), while the secondary horizontal axis shows the corresponding test set. The instances are listed in nonincreasing order of the number of vertices from left to right.) [Color figure can be viewed at wileyonlinelibrary.com]



(A) Running time (B) Cost of solution

FIGURE 5 Sparse graphs with Euclidian weights (selected instances): Running time and cost of solution (The main horizontal axis shows the key parameters of each instance as a triple (n, m, k), while the secondary horizontal axis shows the corresponding test set. The instances are listed in nonincreasing order of the number of vertices from left to right.) [Color figure can be viewed at wileyonlinelibrary.com]

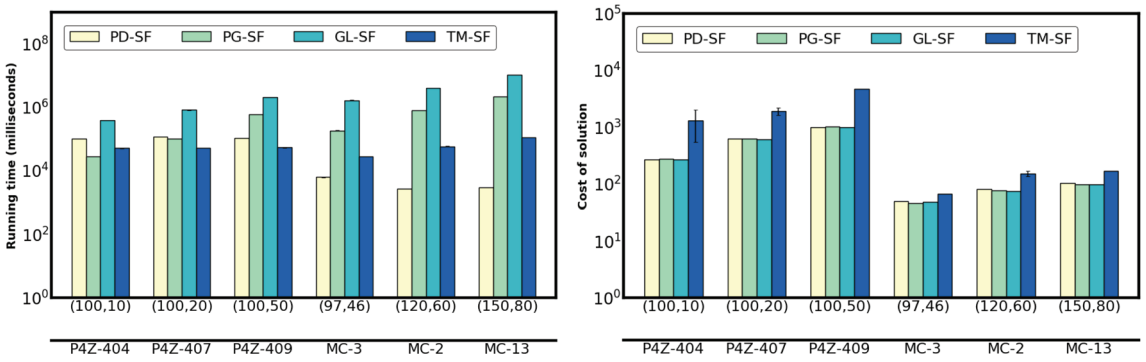
the running time of TM-SF does not increase with the number of terminals. We can see this clearly by considering the first three instances and the last three instances in the plot. Also, we observe that the running time of PG-SF is very dependent on the number of terminals. PD-SF and the two greedy algorithms, GL-SF and PG-SF, obtain solutions with almost similar costs, much lower than the cost of the solutions obtained by TM-SF. Overall, we do not see a significant difference on the behavior of the algorithms compared with the case of sparse graphs with random weights.

Complete graphs

In this section, we analyze the performance of the algorithms on complete graphs with both random and Euclidian edge weights. Also, since the graph density is fixed for all instances, we can analyze the effect of changing the number of vertices and terminals on the algorithms' performance. The instances are selected from test sets MC, P4Z, P4E, and X.

Complete graphs with random weights. We consider instances consisting of complete graphs with random weights from two test sets, MC and P4Z [10]. There are four combinations of instances in P4Z with 100 vertices, 4950 edges, and 5, 10, 20, and 50 terminals. Test set MC contains three complete graphs with different number of vertices and terminals. Figure 6(A,B) show the running time and the cost of the solution for the selected instances.

For the MC test set, PD-SF is the fastest algorithm, followed by TM-SF. The two greedy algorithms have the highest running times, and their running times increase as the number of terminals increases. The running time of TM-SF increases slightly by increasing the number of vertices. Later on, when we perform correlation analysis on the effect of the number of terminals on the algorithms' performance, we find out that TM-SF performance has a higher positive correlation with the number of vertices rather than with the number of terminals. Considering instances from test set P4Z, we do not see any significant increase in the running time of PD-SF as we expect from PD-SF's time complexity, which depends on the number of vertices. We see a slightly different behavior considering instances from the MC test set. By increasing the number of vertices, the running time of the



(A) Running time (B) Cost of solution

FIGURE 6 Complete graphs with random weights (selected instances): Running time and cost of solution (The main horizontal axis shows the key parameters of each instance as a pair (n, k), while the secondary horizontal axis shows the corresponding test set.) [Color figure can be viewed at wileyonlinelibrary.com]

PD-SF algorithm decreases (MC-2, MC-3, and MC-13). This is mainly due to the number of terminals. In a complete graph, a larger number of terminals can help the algorithm connect the pairs faster. Also, this is very dependent on the structure of the terminals in the graph. For example, in a complete graph, if all terminals are adjacent, the PD-SF algorithm terminates quickly. By considering the cost of the solutions of the algorithms for test set MC in Figure 6(B), we observe a good performance of TM-SF. The costs of solutions obtained by TM-SF for complete graphs with random weights instances are greater than the costs of those obtained by the other algorithms by at most one order of magnitude.

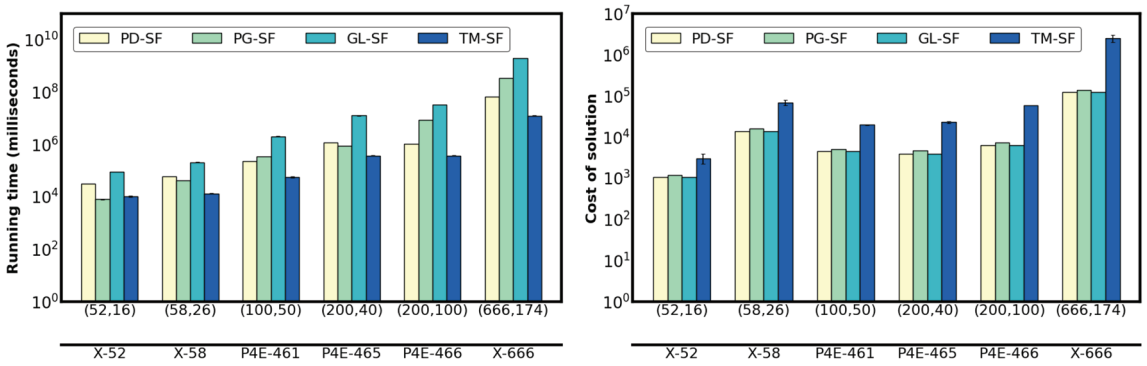
While the performance of the two greedy algorithms, PG-SF and GL-SF, along with TM-SF in terms of the running time for P4Z instances is very similar to what we observed for MC instances, we see a slightly different behavior of PD-SF in terms of the running times. The running time of PD-SF is higher than that of both PG-SF and TM-SF for instances P4Z-404 and P4Z-407, and higher than that of TM-SF for instance P4Z-409. This can be due to the terminal locations in the graph. Similarly to the case of sparse random graphs, the cost of the solution of TM-SF is about one order of magnitude higher than those of the other algorithms. Overall, we observe that although the graph characteristics affect the performance of the algorithms, the graph structure is another important factor that affects the performance of the algorithms.

Complete graphs with Euclidian weights. Test set P4E contains 11 complete graphs with Euclidian weights. It consists of instances with eight different combinations of number of vertices, number of edges, and number of terminals. Test set X contains three complete instances with Euclidian weights, which are originally converted from traveling salesman problems by selecting real-world sights and cites as terminals. X-52 has as vertices 52 sights from Berlin, X-58 has as vertices 58 cities from Brazil with the provincial capitals as terminals, and X-666 has as vertices 666 cities from around the world with capitals as the terminals.

Figure 7(A,B) shows the running time and the cost of the solution for complete graphs with Euclidian weights. The costs of the solution are similar to what we observed for complete graphs with random weights, PD-SF and the two greedy algorithms, PG-SF and GL-SF, obtain solutions with similar costs, while TM-SF obtains solutions with the highest cost, which is expected based on its approximation guarantee. By analyzing the running times presented in Figure 7(A), we observe that TM-SF is the fastest algorithm and that GL-SF is the slowest. We can explain the behavior of TM-SF by considering the algorithm structure in which the tree is built based on the shortest paths between nodes. Since the graph is complete and has Euclidian weights, TM-SF can find the solution faster. We see that the running time of PG-SF increases with the number of terminals. This is due to fact that the running time of the greedy algorithms is highly dependent on the number of terminals. This is applicable to PD-SF too. Overall, TM-SF obtains the best running time but produces solutions with greater costs than those obtained by the other algorithms.

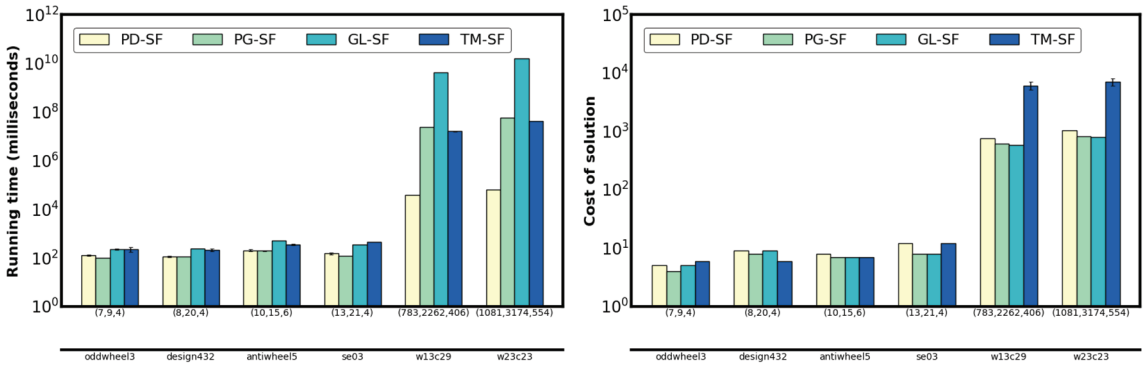
Constructed graphs

In this section, we present the computational results of the algorithms on test set SP, which contains artificially constructed instances. Each instance is a combination of either a fixed number of wheels and cycles, or both. For this test we select four small-size instances and two large-size instances. Figure 8(A,B) shows the running time and the cost of solution on these instances. For the four small instances that are shown on the left in each plot, the algorithms perform almost the same. This is because the instances are very small and the actual difference in performance is not significant. We can see the difference on the next two instances that are larger. PD-SF is the fastest algorithm, but it obtains slightly worse solutions than the other algorithms.



(A) Running time (B) Cost of solution

FIGURE 7 Complete graphs with Euclidian weights (selected instances): Running time and cost of solution (The main horizontal axis shows the key parameters of each instance as a pair (n, k), while the secondary horizontal axis shows the corresponding test set. The instances are listed in nonincreasing order of the number of vertices from left to right.) [Color figure can be viewed at wileyonlinelibrary.com]



(A) Running time (B) Cost of solution

FIGURE 8 Constructed graphs (selected instances): Running time and cost of solution (The main horizontal axis shows the key parameters of each instance as a triple (n, m, k), while the secondary horizontal axis shows the corresponding test set. The instances are listed in nonincreasing order of the number of vertices from left to right.) [Color figure can be viewed at wileyonlinelibrary.com]

VLSI-derived grid graphs

In this section, we present the result on the VLSI-derived grid graphs group which consists of 82 instances classified into four different test sets. All test sets in this group, except LIN, consists of instances that are grid graphs. They come from a VLSI application [24]. Instances from test set LIN are very sparsified grid graphs that are derived from a placement of rectangular blocks in a 2D plane, thus, all line segments are either horizontal or vertical. Due to the similarity of results, we selected only the relevant instances from each test set and present them in one plot to be able to analyze the behavior of the algorithms on these test sets.

Figure 9(A,B) shows the running time and the cost of the solution for the selected instances from four test sets in the VLSI-derived grid graphs group. The costs of the solutions obtained by PD-SF, PG-SF, and GL-SF are similar, while TM-SF obtains solutions with the highest cost. Although we observe similar results for the cost of the solution, the algorithms perform differently in terms of the running time. PG-SF is the fastest for test sets LIN and DIW, but not for test sets ALUE and ALUT, for which PD-SF is the fastest. Although the GL-SF algorithm is the slowest algorithm, its running time is very close to that of TM-SF. This is due to the fact that the instances of this group are very sparse with a maximum graph density of 5%, therefore GL-SF spends less effort finding all pairs shortest paths.

Rectilinear graphs

In this section, we present the result on the rectilinear graphs group, which consists of 195 instances classified into 13 different test sets. The instances are preprocessed grid instances derived from random terminal points in the plane. The edge weights are the rectilinear distances of the points that are placed at random on the plane.

Figure 10(A,B) shows the running time and the cost of the solution for the selected instances from the group of rectilinear graphs. We selected the largest instances from each set. For all instances considered in this experiment, GL-SF is the slowest algorithm, while PD-SF is the fastest. The running time of PD-SF is about one order of magnitude less than the second

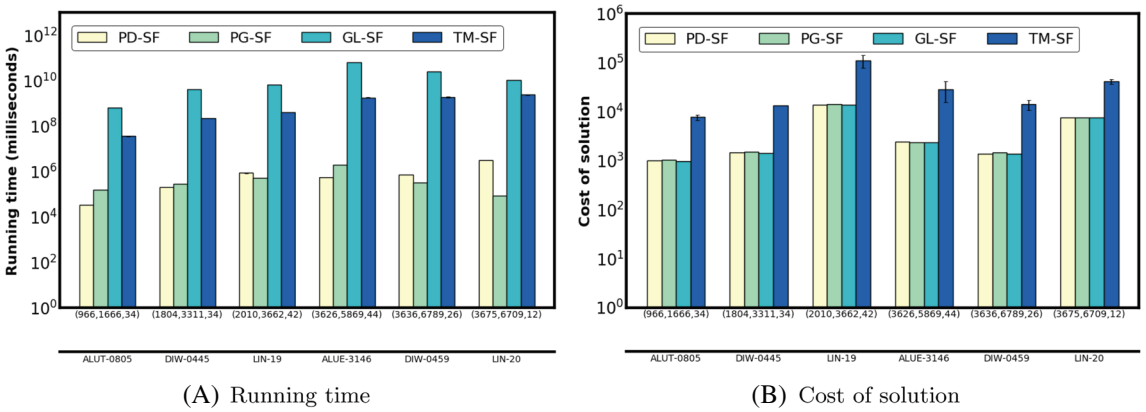


FIGURE 9 VLSI-derived grid graphs (selected instances): Running time and cost of solution (The main horizontal axis shows the key parameters of each instance as a triple (n, m, k), while the secondary horizontal axis shows the corresponding test set. The instances are listed in nonincreasing order of the number of vertices from left to right.) [Color figure can be viewed at wileyonlinelibrary.com]

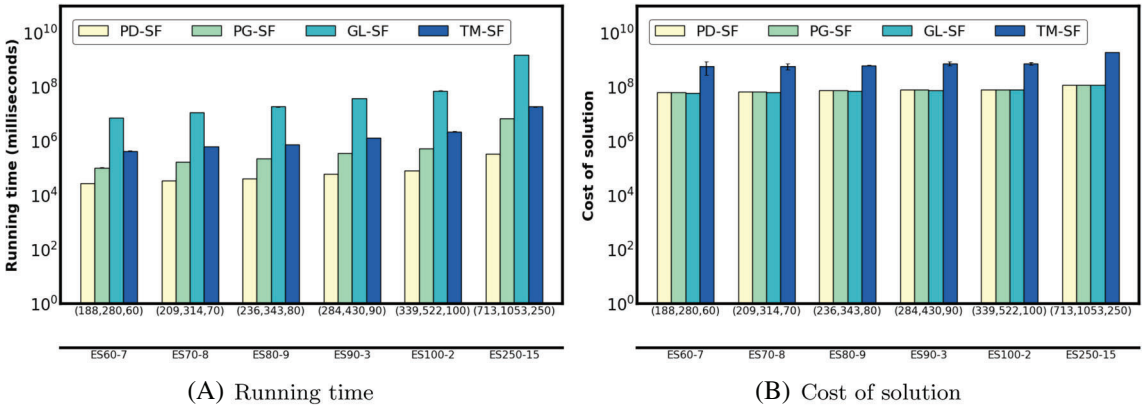


FIGURE 10 Rectilinear graphs (selected instances): Running time and cost of solution (The main horizontal axis shows the key parameters of each instance as a triple (n, m, k), while the secondary horizontal axis shows the corresponding test set. The instances are listed in nonincreasing order of the number of vertices from left to right.) [Color figure can be viewed at wileyonlinelibrary.com]

fastest algorithm, PG-SF. The running time of PG-SF increases by increasing the number of vertices. This is true for all other algorithms as well. Similarly to the case of VLSI-derived grid graphs, PG-SF is faster than TM-SF. The costs of the solutions obtained by PD-SF, PG-SF, and GL-SF are similar, while TM-SF obtains solutions with the highest cost. Considering the quality of the solutions together with the running time of the algorithms for rectilinear graphs, we can say that PD-SF performs the best.

Wire-routing graphs

Figure 11(A,B) shows the running time and the cost of the solution for the selected instances from four test sets of the wire-routing graphs group. The group contains 126 instances of VLSI wire-routing problems from IBM chips, called WRP3 and WRP4. We selected some of the instances from both test sets and analyzed the performance of the algorithms.

Although we observe a similar behavior of the algorithms in terms of running time to that observed in the case of VLSI-derived graphs, the algorithms perform differently in terms of the quality of the solution. In Figure 11(B) we can see that all algorithms obtain solutions with almost the same cost. This is an improvement for TM-SF.

Erdős–Rényi random graphs

In this section, we analyze the performance of the algorithms on random graphs generated using the Erdős–Rényi model [14]. As we mentioned earlier, we generated these instances in order to analyze the performance of the algorithms on a wider range of graph types that were not available from the original SteinLib library. We randomly generated 180 instances using the Erdős–Rényi model. The density distribution of these instances ranges from 10% to 90%, while the terminal coverage of all instances is 20%. We categorize the instances into nine test sets based on the number of vertices and the graph density. The details of each of these instances are given in Table 1. For reasons of space and because of the similarity of results, we do not present the results for instances from sets MSRG, MMRG, and MDRG.

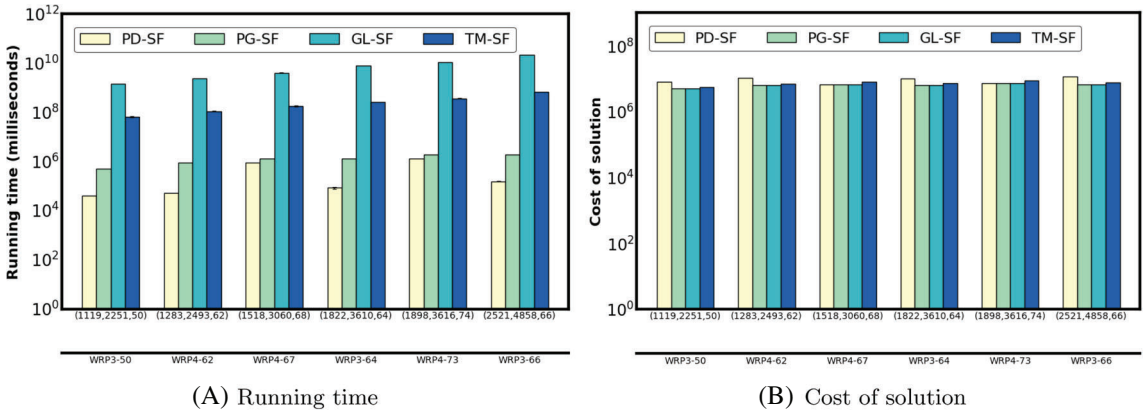


FIGURE 11 Wire-routing graphs (selected instances): Running time and cost of solution (The main horizontal axis shows the key parameters of each instance as a triple (n, m, k), while the secondary horizontal axis shows the corresponding test set. The instances are listed in nonincreasing order of the number of vertices from left to right.) [Color figure can be viewed at wileyonlinelibrary.com]

Figures 12 and 13 present the running times and the cost of the solutions for the six test sets considered in the experiment. Plots in these figures are presented based on the order given in Table 1. The first row of plots corresponds to instances with the same graph density ($d = 30\%$) and different number of vertices, n. Similarly, the second and third rows correspond to instances with $d = 60\%$ and $d = 90\%$, respectively. Since all selected instances in each plot have the same density and terminal coverage, we eliminated the number of edges and the number of terminals in the plots presentation and the horizontal axis shows only the number of vertices.

Presenting all plots together in one figure gives us a high-level picture of the algorithms' performance, which allows us to investigate the effect of the density and the terminal coverage. Figure 12 shows that the performance of the algorithms on different rows is very similar. Therefore, as long as we keep the terminal coverage fixed, the graph density does not have a significant effect on the algorithms running times. In each row, we see a slow increase in the algorithms' running times with the increase in the number of vertices. Hence, considering all plots together we can say that the running times of the algorithms have a small positive correlation with the number of vertices and a very small positive correlation with the graph density. This is also confirmed by the correlation analysis presented in Section 3.2.4. Figure 13 shows similar behavior as in the case of sparse graphs with random weights. The cost of solutions obtained by PD-SF, PG-SF, and GL-SF is almost the same. For this type of graphs, PD-SF performs the best in terms of both the running time and the quality of solution. The worst performance in terms of running time is obtained by GL-SF, while the worst performance in terms of quality of solution is obtained by TM-SF.

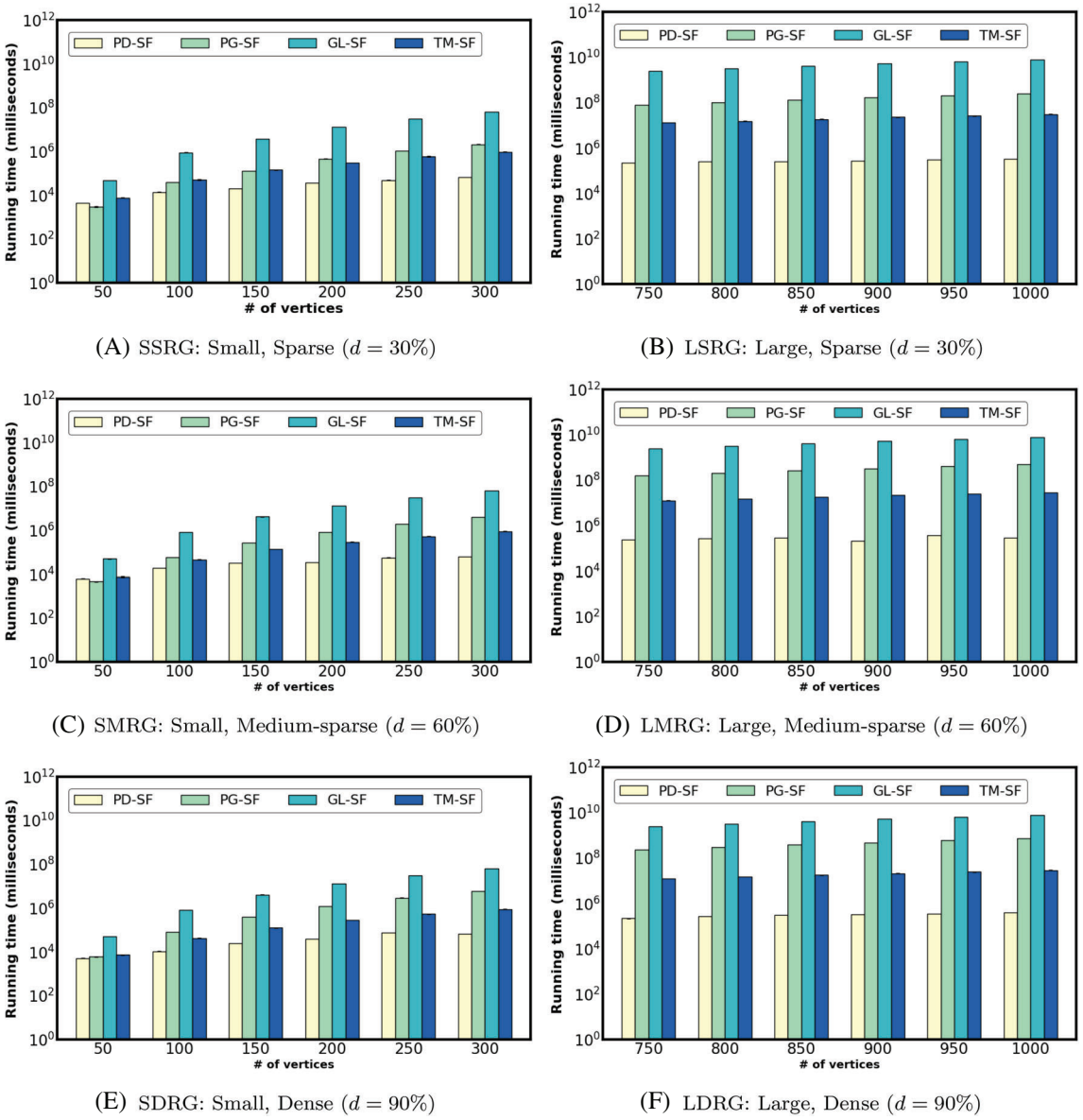
3.2.2 | Performance with respect to the number of edges

In this section, we analyze the performance of the algorithms with respect to the number of edges, m. In order to accurately investigate the effect of increasing the number of edges, m, on the performance of the algorithms, we fixed the number of vertices and terminals and varied the number of edges. For this experiment, we considered 16 instances from test D. For reasons of space and because of the similarity of results, we do not present the results for instances from test sets B and C.

Figure 14(A,B) shows the running time of the algorithms versus the number of edges for instances from test set D, with 1000 vertices and the number of terminals ranging from 10 to 500. Each plot shows the running time of the algorithms versus the number of edges for instances with the same number of vertices and number of terminals. For example, Figure 14(A) shows the results corresponding to instances with 1000 vertices and 10 terminals.

For all instances considered in this experiment, GL-SF is the slowest algorithm, followed by TM-SF. The running time of GL-SF is about an order of magnitude greater than that of TM-SF. This difference becomes more visible, to about two orders of magnitude, for the instances with a larger number of terminals ($t = n/6, n/4$, and $n/2$), which are presented in Figure 14(B). PD-SF is the fastest algorithm for all instances except the instances with very small number of terminals for which PG-SF is slightly faster, as shown in Figure 14(A). This is mainly due to the fact that the running time of PG-SF is highly dependent on the number of terminals, that is, PG-SF checks each terminal pair for inclusion in the forest one at the time. This is in contrast to PD-SF which considers all the terminals as a whole and its running time is not highly dependent on the number of terminal pairs.

The running time of PD-SF is less than 10 s for almost all instances. The running times of PG-SF, GL-SF, and TM-SF are about four to six orders of magnitude greater than those of PD-SF. Looking at each plot in Figure 14 separately, uncovers another interesting fact, increasing the number edges does not affect the running time of any of the algorithms. For example, in



(A) SSRG: Small, Sparse ($d = 30\%$)

(B) LSRG: Large, Sparse ($d = 30\%$)

(C) SMRG: Small, Medium-sparse ($d = 60\%$)

(D) LMRG: Large, Medium-sparse ($d = 60\%$)

(E) SDRG: Small, Dense ($d = 90\%$)

(F) LDRG: Large, Dense ($d = 90\%$)

FIGURE 12 Erdős–Rényi random graphs (selected instances): Running time versus number of vertices. The terminal coverage for all instances is 20% [Color figure can be viewed at wileyonlinelibrary.com]

Figure 14(B) the running time of GL-SF algorithm is almost the same for all instances with different number of edges. This is true for all other algorithms.

Figure 15(A,B) shows the cost of the solutions obtained by the algorithms versus the number of edges. The setup and the order of the plots in Figures 15 is similar to Figure 14. PD-SF and the two greedy algorithms, GL-SF and PG-SF obtain solutions with almost similar costs, much lower than the cost of the solutions obtained by TM-SF, the difference being about an order of magnitude,

Considering the quality of the solution together with the running time of the algorithms we can draw three conclusions. First, increasing the number of edges does not increase the running time of the algorithms. Second, PD-SF performs the best in terms of both the solution quality and its running time and is very suitable for use in practice. Third, even though there is no known approximation guarantee for the paired greedy algorithm, its actual performance in terms of both the solution quality and running time makes it suitable for instances with small number of terminals.

3.2.3 | Performance with respect to the number of terminals

In this section, we investigate the effect of the number of terminals on the performance of the algorithms. For this experiment, we consider instances from test set D. Similarly to the previous experiments, we keep two parameters fixed (n and m) and vary the third one, the number of terminals, t, for each instance type.

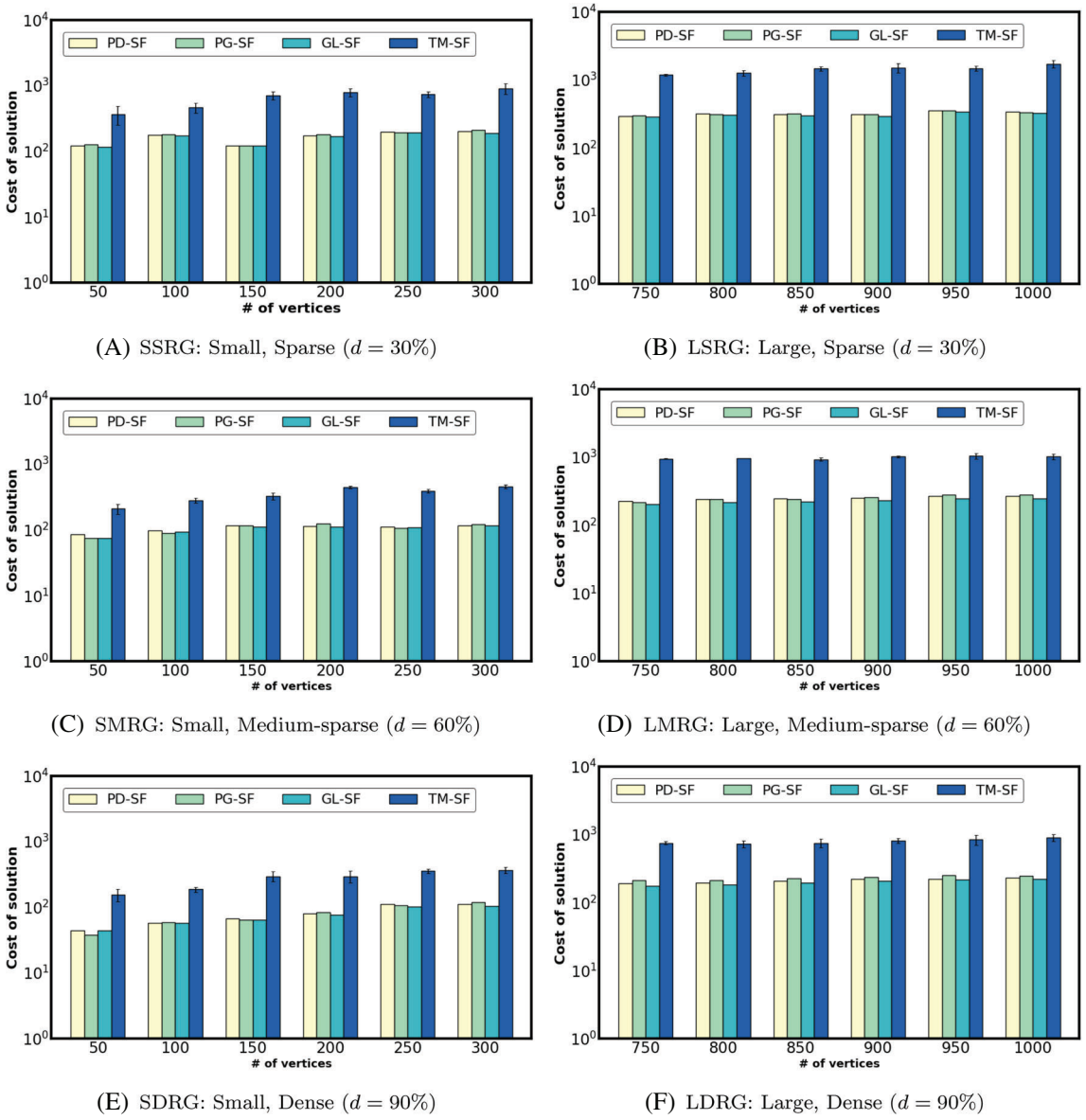
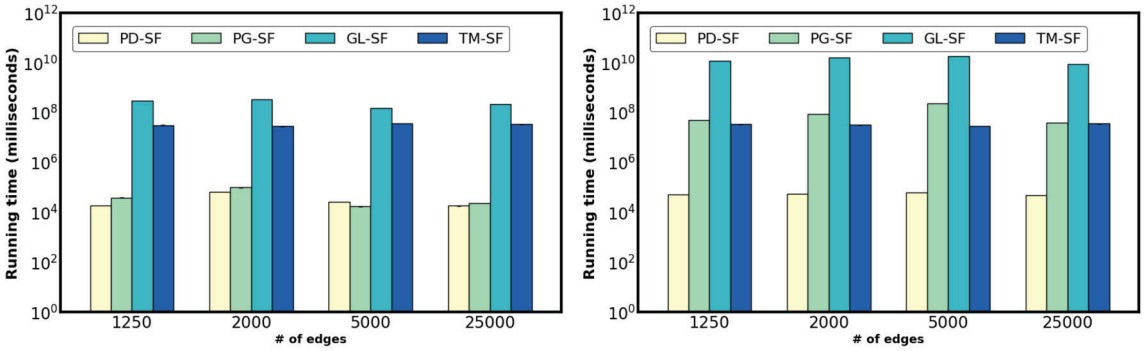


FIGURE 13 Erdős–Rényi random graphs (selected instances): Cost of solution versus number of vertices. The terminal coverage for all instances is 20% [Color figure can be viewed at wileyonlinelibrary.com]

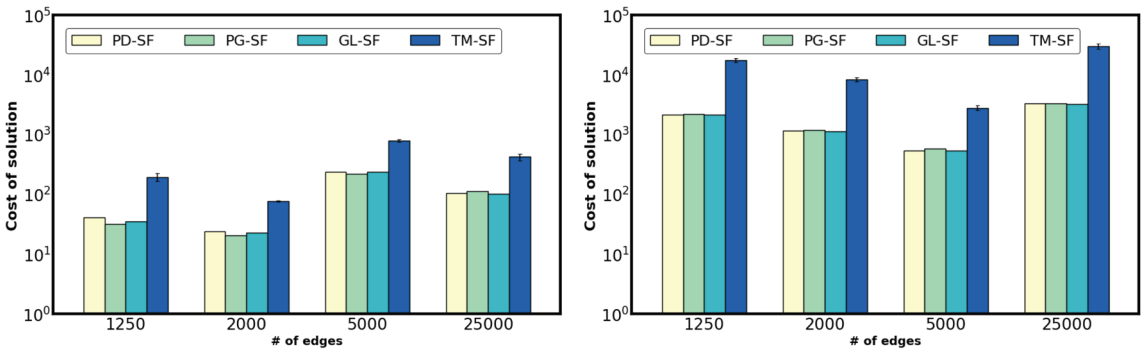
Figure 16(A,B) shows the running time of the algorithms versus the number of terminals for the 10 instances from test set D. For all types of instances considered in this experiment, PD-SF is the fastest except for instances with a small number of the terminals ($t = 6$). Similarly, to the previous experiments, GL-SF is the slowest algorithm, followed by TM-SF. For instances with a lower number of terminals ($t = 6, 10$), this difference is not significant, and their running times are very close to each other. But, by increasing the number of terminals this difference becomes more visible. The running time of GL-SF is about two orders of magnitude greater than that of TM-SF for instances with the number of terminals larger than 168. This is true if we compare the running times of PD-SF and PG-SF, that have a significant difference for the instances with larger number of terminals. This leads us to another observation, the two algorithms, PD-SF and TM-SF, have running times that do not increase by increasing the number of terminals. This is because the number of operations executed by TM-SF is independent of the number of terminals and the algorithm builds the tree metric only based on the shortest paths between vertices. Thus, the running time of TM-SF is highly dependent on the number of vertices, but not on the number of terminals. And similarly for PD-SF, its running time is not dependent on the number of terminals as it considers all the terminals as a whole. By contrast, we see that the running times of both greedy algorithms, PG-SF and GL-SF, are highly dependent on the number of terminals because they are both based on finding the shortest path between two terminal nodes, and increasing the number of terminals adds to the workload.

Figure 17(A,B) shows the cost of the solutions obtained by the algorithms versus the number of terminals. The set up and order of the plots in Figure 17 is similar to Figure 16. Again, PD-SF and the two greedy algorithms, PG-SF and GL-SF, produce



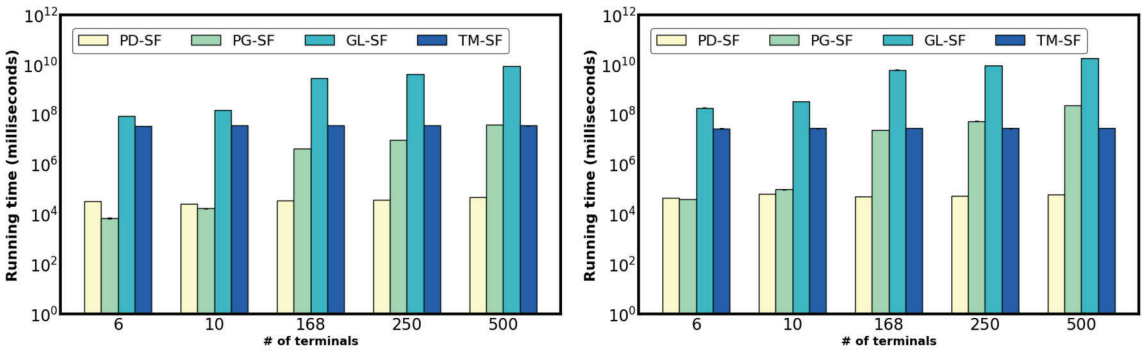
(A) Set D: $n = 1000, t = 10$. (B) Set D: $n = 1000, t = 500$.

FIGURE 14 Running time versus number of edges (sparse graphs with random weights) [Color figure can be viewed at wileyonlinelibrary.com]



(A) Set D: $n = 1000, t = 10$. (B) Set D: $n = 1000, t = 500$.

FIGURE 15 Cost of solution versus number of edges (sparse graphs with random weights) [Color figure can be viewed at wileyonlinelibrary.com]



(A) Set D: $n = 1000, m = 1250$. (B) Set D: $n = 1000, m = 25000$.

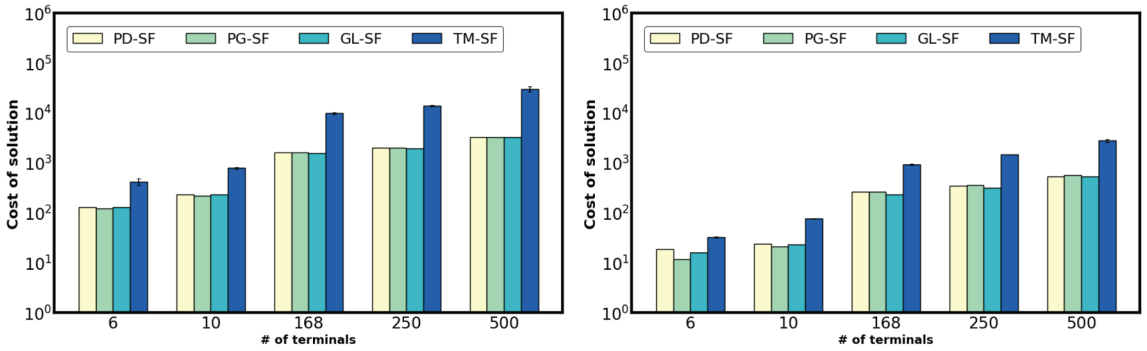
FIGURE 16 Running time versus number of terminals (sparse graphs with random weights) [Color figure can be viewed at wileyonlinelibrary.com]

the best solution and they have significant differences compared with TM-SF's solution. The quality of the solution obtained by PG-SF improves slightly by increasing the number of terminals. Overall, the number of terminals has a stronger effect on PG-SF and GL-SF than on the other two approximation algorithms.

3.2.4 | Overall analysis

In this section, we analyze the strength and direction of the relationship between the running time of each algorithm and graph parameters, n, m, k, d, and c, and give an overall analysis of the performance of each algorithm. To do so, we compute the correlation coefficients for each algorithm's running time to measure its dependency on the graph parameters. We applied the Pearson's method to obtain the correlation coefficients. Table 2 presents the correlation coefficients.

We recall that the value of the correlation coefficient is between $+1$ and -1, where $+1$ indicates a perfect positive relationship and -1 indicates a perfect negative relationship. Table 2 shows that the maximum positive correlation coefficient of PD-SF is 0.34 with respect to the number of vertices, followed by those with respect to m, k, and d, that are very small. This means that



(A) Set D: $n = 1000, m = 1250$. (B) Set D: $n = 1000, m = 25000$.

FIGURE 17 Cost of the solution versus number of terminals (sparse graphs with random weights) [Color figure can be viewed at wileyonlinelibrary.com]

TABLE 2 Correlation coefficients (n is the number of vertices, m is the number of edges, k is the number of terminals, d is the density, and c the is coverage)

Algorithm	n	m	k	d	c
PD-SF	0.34	0.15	0.19	0.10	−0.04
PG-SF	0.31	0.42	0.68	0.12	0.05
GL-SF	0.47	0.05	0.43	−0.06	−0.06
TM-SF	0.72	−0.03	0.12	−0.11	−0.16

the running time of PD-SF is not very dependent on the number of edges (i.e., density) and the number of terminals, and it has a very small positive correlation with the number of vertices. Figures 14 and 17 that present the effect of changing the number of terminals and the number of edges on the performance of the algorithms, confirm the same findings. We observed that by increasing the number of edges and terminals, the running time of PD-SF does not increase significantly. On the other hand, we observed in Figures 3 and 4 that whenever the number of vertices increases, the running time of PD-SF increases. It does not have a strong relationship but its increase is as we expect from a correlation coefficient value of 0.34. The correlation coefficient of PD-SF with respect to the terminal coverage, c, is −0.04. It is too small to show any relationship with the terminal coverage.

We observe higher correlation coefficients for other algorithms. PG-SF has a high correlation coefficient with respect to the number of terminals. This means that the algorithm's running time has a strong dependency on the number of terminals. Again, we can observe the same behavior in Figure 17, when we analyze the effect of increasing the number of terminals on the algorithm's performance. Figure 14 shows that by keeping the number of terminals fixed, the running time of the PG-SF does not increase significantly and it is almost nonincreasing. This is due to the strong dependency of the algorithm's performance on the number of terminals which dominates the other parameters. The highest correlation coefficients of GL-SF are with respect to the number of vertices and the number of terminals, respectively. This shows GL-SF's dependency on n and k, which is not that strong to cause a huge increase in its running time. We can confirm it in Figure 16, and in almost all other figures that present the results with respect to the number of vertices. Finally, the highest correlation coefficient, 0.72, for TM-SF is with respect to the number of vertices, which is clearly visible from all previous analysis and figures confirming the fact that increasing the number of edges and the number terminals does not affect TM-SF's performance in terms of running time. On the other hand, the running time of TM-SF increases a lot by increasing the number of vertices. We see negative values for its correlation with respect to the density and the terminal coverage.

3.2.5 | Comparison with the optimal solution for small-size instances

In this section, we compare the solutions obtained by the algorithms with the solution obtained by solving the *flow-based* IP formulation of the Steiner forest problem [30, 35]. The goal is to determine how far from the optimal are the solutions obtained by the four algorithms considered here. Because the cut-based formulation presented in Equations (1) to (3) has an exponential number of constraints and is not feasible to solve, for our experiments, we choose the flow-based formulation.

The IP flow-based formulation [30, 35] is given in Equations (7) to (11). The basic idea behind this formulation is to send one unit of flow from a terminal node to its mate. To do so, one node from each terminal pair is selected as a fixed root of the terminal pair. For simplicity of notation, we assume that s_i is the root of each terminal pair (s_i, t_i). Then, for each edge $(i, j) \in E$ and each nonroot terminal t we introduce two flow variables f_{ij}^t and f_{ji}^t, and for each edge $(i, j) \in E$, a selection variable x_{ij}. The

TABLE 3 Cost of solutions obtained by the algorithms versus cost of optimal solution (small-size instances)

Name	*n*	*m*	*k*	OPT cost	PD-SF cost	PD-SF ratio	PG-SF cost	PG-SF ratio	GL-SF cost	GL-SF ratio	TM-SF cost	TM-SF ratio
b01	50	63	9	80	94	1.175	80	1.000	90	1.125	258	3.225
b02	50	63	13	83	84	1.012	84	1.012	83	1.000	556	6.699
b03	50	63	25	142	144	1.014	151	1.063	142	1.000	528	3.718
b04	50	100	9	61	61	1.000	68	1.115	64	1.049	198	3.246
b05	50	100	13	53	61	1.151	53	1.000	56	1.057	232	4.377
b06	50	100	25	122	127	1.041	124	1.016	125	1.025	440	3.607
b07	75	94	13	112	112	1.000	113	1.009	112	1.000	656	5.857
b08	75	94	19	106	108	1.019	108	1.019	108	1.019	716	6.755
b09	75	94	38	220	223	1.014	229	1.041	220	1.000	1559	7.086
b10	75	150	13	81	96	1.185	88	1.086	86	1.062	320	3.951
b11	75	150	19	88	93	1.057	91	1.034	90	1.023	307	3.489
b13	100	125	17	165	178	1.079	167	1.012	179	1.085	991	6.006
b14	100	125	25	226	244	1.080	228	1.009	240	1.062	928	4.106
b15	100	125	50	315	320	1.016	335	1.063	316	1.003	1816	5.765
b16	100	200	17	131	138	1.053	139	1.061	138	1.053	545	4.160
b17	100	200	25	128	134	1.047	128	1.000	129	1.008	437	3.414
b18	100	200	50	222	224	1.009	225	1.014	222	1.000	1057	4.761
c01	500	625	6	80	98	1.225	80	1.000	89	1.113	364	4.550
c02	500	625	10	137	144	1.051	144	1.051	144	1.051	616	4.496
c06	500	1000	6	63	74	1.175	63	1.000	71	1.127	272	4.317
c07	500	1000	10	98	114	1.163	98	1.000	102	1.041	355	3.622
c11	500	2500	6	26	33	1.269	26	1.000	33	1.269	144	5.538
c12	500	2500	10	44	47	1.068	45	1.023	46	1.045	147	3.341

TABLE 4 The running time by the algorithms versus the running time of solving the IP by CPLEX (small-size instances)

Name	*n*	*m*	*k*	OPT time	PD-SF time	PG-SF time	GL-SF time	TM-SF time
b01	50	63	9	0.126	0.001	0.001	0.026	0.008
b02	50	63	13	0.117	0.001	0.002	0.034	0.009
b03	50	63	25	0.097	0.001	0.006	0.076	0.008
b04	50	100	9	0.139	0.001	0.001	0.037	0.008
b05	50	100	13	0.137	0.001	0.002	0.049	0.008
b06	50	100	25	0.255	0.001	0.007	0.093	0.008
b07	75	94	13	0.169	0.001	0.002	0.116	0.024
b08	75	94	19	0.047	0.001	0.005	0.158	0.024
b09	75	94	38	0.109	0.001	0.016	0.277	0.024
b10	75	150	13	0.015	0.002	0.003	0.144	0.021
b11	75	150	19	0.180	0.001	0.006	0.220	0.020
b13	100	125	17	0.116	0.002	0.005	0.305	0.050
b14	100	125	25	0.115	0.002	0.009	0.440	0.049
b15	100	125	50	0.319	0.002	0.034	1.056	0.055
b16	100	200	17	0.225	0.003	0.006	0.468	0.049
b17	100	200	25	0.230	0.002	0.013	0.706	0.046
b18	100	200	50	0.340	0.002	0.046	1.301	0.048
c01	500	625	6	0.052	0.008	0.004	11.376	4.598
c02	500	625	10	0.356	0.01	0.008	18.831	4.780
c06	500	1000	6	0.251	0.008	0.005	15.240	4.514
c07	500	1000	10	0.310	0.011	0.011	28.326	4.380
c11	500	2500	6	0.110	0.015	0.008	21.307	4.174
c12	500	2500	10	1.193	0.016	0.018	37.834	4.195

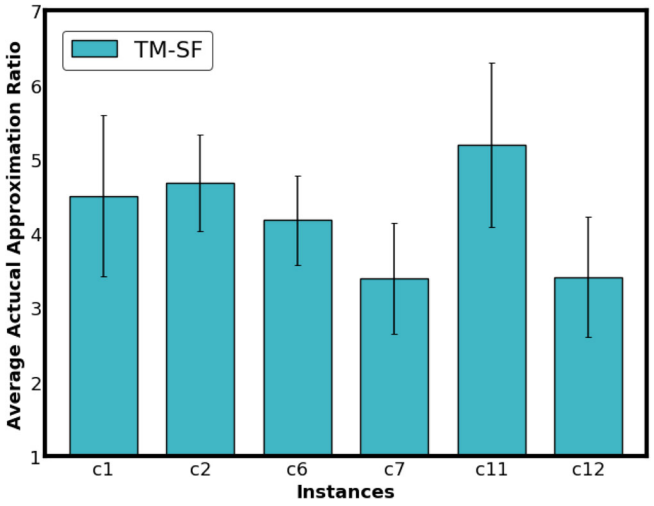


FIGURE 18 TM-SF: Actual approximation ratio analysis [Color figure can be viewed at wileyonlinelibrary.com]

idea behind this formulation is to send one unit of flow from each nonroot terminal t_i to its terminal root s_i, therefore the variable x_{ij} induces the selection of the edge (i,j), if it allows the flow.

$$\text{minimize} \quad \sum_{\{i,j\}\in E} w_{ij}x_{ij} \tag{7}$$

subject to:

$$\sum_{j\in V} f_{ji}^t - \sum_{l\in V} f_{il}^t = \begin{cases} 1, & \text{if} \quad i = s \\ -1, & \text{if} \quad i = t, \quad \forall i \in V \text{ and all } t. \\ 0, & \text{otherwise} \end{cases} \tag{8}$$

$$f_{ij}^t + f_{ji}^t \leq x_{ij} \qquad \forall (i,j) \in E \text{ and all nonroot terminal } t. \tag{9}$$

$$f_{ij}^t, f_{ji}^t \in \{0,1\} \qquad \forall (i,j) \in E \text{ and all nonroot terminal } t. \tag{10}$$

$$x_{ij} \in \{0,1\} \qquad \forall (i,j) \in E. \tag{11}$$

We solved the IP formulation of the problem by using the CPLEX solver provided by IBM ILOG CPLEX Optimization Studio for Academics Initiative [21]. We selected small instances from test set B, for which the IP is solvable using the CPLEX solver in a reasonable amount of time. The number of vertices is between 50 and 500, the number of edges is between 63 and 2500, and the number of terminals is between 9 and 84. To compare the quality of the solutions obtained by the four algorithms considered here, we define the *actual approximation ratio* for an algorithm A as the ratio of the cost of the solution obtained by algorithm A and the cost of the optimal solution. The actual approximation ratios and the cost of the solutions for each of the algorithms and for each of the instances considered in our experiments are given in Table 3. In the table, the cost of the optimal solution for each instance is given in the fifth column (OPT cost), while the cost of the solutions and the actual approximation ratios for each algorithm are given in the rest of the columns (e.g., column "PD-SF cost" gives the cost of the solutions obtained by PD-SF, and column "PD-SF ratio" gives the actual approximation ratio for PD-SF).

The actual approximation ratios of the two greedy algorithms PG-SF and GL-SF for almost all instances are very close to 1, that is, their solutions are very close to the optimal solution. The worst actual approximation ratios for PG-SF and GL-SF are 1.115 and 1.127. The actual approximation ratio of PD-SF is very close to 1 for almost all instances except few instances for which it reaches 1.185. On the other hand, TM-SF has the highest ratio among all algorithms, which is expected from its theoretical guarantees. Its worst actual approximation ratio is 7.086. The results show that the greedy algorithms PG-SF and GL-SF, despite having a worse theoretical guarantees, perform very well on the instances considered here.

Since TM-SF is a randomized algorithm, we performed an additional analysis to provide more detailed statistics on the quality of the solution. We considered instances c1, c2, c6, c7, c11, and c12, and ran the algorithm on each of these instances 10 times, independently. We obtained the actual approximation ratio for each run and computed the average and the standard deviation over the 10 independent runs. Figure 18 shows the average of actual approximation ratio and the standard deviation for the instances considered in this analysis. We observed that the actual approximation ratio is less than 7, which is within the expected bound, as TM-SF is a $O(\log n)$-approximation algorithm and the number of vertices for the instances considered here is $n = 500$.

Table 4 presents the actual running times of the CPLEX solver and the other algorithms. Although GL-SF is the slowest when compared with the other algorithms, as we discussed in the previous sections, its running time is way less than that required to solve the IP using the CPLEX solver. When we consider the quality of the solutions together with the running time of the optimal algorithm, we can conclude that we can rely on the PD-SF algorithm for obtaining near-optimal solutions in a significantly smaller running time than the CPLEX solver.

4 | CONCLUSION

We provided efficient implementations of existing approximation algorithms for Steiner forest and conducted a thorough experimental study to characterize their performance. The algorithms were implemented in C++ without any external graph libraries and the code was released as part of the SFlib, a project available on Github [16]. For our experiments, we created a library, called SteinForestLib, of about 1200 instances for the Steiner forest and made it publicly available [36]. We also compared experimentally the cost of the solutions obtained by the algorithms with the cost of the optimal solution obtained by solving a flow-based IP formulation of the Steiner forest problem. As expected, for the instances considered in our experiments, the approximation algorithms obtained actual approximation ratios that are much better than their theoretical approximation ratios. We also observed that the algorithms based on more refined techniques do not necessarily lead to better results. For example, although PG-SF has no theoretical guarantee with respect to the quality of solutions and at best it is a $O(\log n)$-approximation, obtained very good solutions which are on par with those obtained by PD-SF. On the other hand, TM-SF which has a theoretical approximation guarantee of $O(\log n)$ is not competitive with PG-SF in terms of the quality of solutions. Overall, PD-SF and the two greedy algorithms, GL-SF and PG-SF, obtain solutions with almost similar costs, much lower than the cost of the solutions obtained by TM-SF. This is true for most of the instances except for the wire-routing instances and the sparse graphs with incidence weights for which we see a different behavior due to the structure of the graphs. PD-SF is much faster than all the other algorithms. The implementation of PD-SF does not use a priority queue to update the reduced costs of the edges; therefore, for dense graphs, we expect even better performance if such a priority queue is used. PG-SF is fast only when the number of terminals is small. This is due to the fact that it only computes the shortest path between the terminal pairs, and it takes less effort to compute the solution. We can see this behavior in the case of VLSI-derived grid graphs. As the number of terminals, and the number of edges increases, the running time of TM-SF does not change. This is because the number of operations executed by TM-SF is independent of the number of terminals and the algorithm builds the tree metric only based on the shortest paths between vertices. The running time of TM-SF is much smaller than that of PG-SF for instances with a large number of terminals. The running times of PG-SF and GL-SF increase rapidly with the number of terminals. Therefore, for practical purposes, we recommend using PD-SF. We also obtained the correlation coefficients for the running time of the algorithms with respect to the graph parameters. The correlation analysis reveals that the running time of TM-SF has a strong positive correlation with the number of vertices, PG-SF is highly dependent on the number of terminals, GL-SF has roughly similar dependency on the number of vertices and terminals, and PD-SF has a small dependency on the number of vertices. For future work, we plan to investigate the performance of approximation algorithms for Steiner forest that are specifically designed for planar graphs [4], and of approximation algorithms for other network design problems.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in Github at https://github.com/lalehghalami/SteinForestProblem.

ORCID

Daniel Grosu https://orcid.org/0000-0003-2340-5433

REFERENCES

[1] A. Agrawal, P. N. Klein, and R. Ravi, *When trees collide: An approximation algorithm for the generalized Steiner problem on networks*, SIAM J Comput **24** (1995), 440–456.
[2] Y. Aneja, *An integer linear programming approach to the Steiner problem in graphs*, Networks **10** (1980), 167–178.
[3] B. Awerbuch and Y. Azar, *Buy-at-bulk network design*, *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, Miami Beach, Florida, USA, 1997, pp. 542–547.
[4] M. Bateni, M. Hajiaghayi, and D. Marx, *Approximation schemes for Steiner forest on planar graphs and graphs of bounded treewidth*, J. ACM **58**(5) (2011), 21:1–21:37.
[5] J. Beasley, *An algorithm for the Steiner problem in graphs*, Networks **14** (1984), 147–159.

[6] M. Bern and P. Plassmann, *The Steiner problem with edge lengths 1 and 2*, Inf. Process. Lett. **32**(4) (1989), 171–176.
[7] S. Beyer and M. Chimani, *Strong Steiner tree approximations in practice*, ACM J. Exp. Algorithmics **24**(1) (2019), 1.7:1–1.7:33.
[8] H.-L. Chen, T. Roughgarden, and G. Valiant, *Designing network protocols for good equilibria*, SIAM J. Comput. **39**(5) (2010), 1799–1832.
[9] M. Chlebik and J. Chlebikova, *The Steiner tree problem on graphs: Inapproximability results*, Theor. Comput. Sci. **406**(3) (2008), 207–214.
[10] S. Chopra, E. Gorres, and M. Rao, *Solving the Steiner tree problem on a graph using branch and cut*, ORSA J. Comput. **4** (1992), 320–335.
[11] A. Çivril, *Approximation of steiner forest via the bidirected cut relaxation*, J. Comb. Optim. **38**(4) (2019), 1196–1212.
[12] M. P. de Aragão and R. F. Werneck, *On the implementation of mst-based heuristics for the Steiner problem in graphs*, in *Algorithm Engineering and Experiments (ALENEX)*, D. M. Mount and C. Stein, Eds., Springer, Berlin/Heidelberg, Germany, 2002, 1–15.
[13] C. Duin, Steiner problems in graphs. Ph.D. Thesis, Univ. Amsterdam, 1993.
[14] P. Erdős and A. Rényi, *On random graphs I*, Publ. Math. **6**(3) (1959), 290–297.
[15] J. Fakcharoenphol, S. Rao, and K. Talwar, *A tight bound on approximating arbitrary metrics by tree metrics*, J. Comput. Syst. Sci. **69**(3) (2004), 485–497.
[16] L. Ghalami, *SFlib: A library of approximation algorithms for Steiner forest*, 2020, available at https://github.com/lalehghalami/A-Library-of-Approximation-Algorithms-f%or-Steiner-Forest.
[17] M. X. Goemans and D. P. Williamson, *A general approximation technique for constrained forest problems*, SIAM J. Comput. **24**(2) (1995), 296–317.
[18] M. Groß, A. Gupta, A. Kumar, J. Matuschke, D. R. Schmidt, M. Schmidt, and J. Verschae, *A local-search algorithm for Steiner forest*, in *Proceedings of the 9th Innovations in Theoretical Computer Science Conference (ITCS 2018), volume 94 of Leibniz International Proceedings in Informatics (LIPIcs)*, A. R. Karlin, Ed., Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2018, 31:1–31:17.
[19] A. Gupta and A. Kumar, *Greedy algorithms for Steiner forest*, Proceedings of the 47th Annual ACM Symposium on Theory of Computing, Portland, OR, USA 2015, pp. 871–878.
[20] M. Hanan, *On Steiner's problem with rectilinear distance*, J. SIAM Appl. Math. **14**(2) (1966), 255–265.
[21] IBM *IBM ILOG CPLEX optimization studio for academics initiative*, 2018, available at https://ibm.onthehub.com/WebStore/ProductSearchOfferingList.aspx?srch=i%logcplex.
[22] K. Jain, *A factor 2 approximation algorithm for the generalized steiner network problem*, Proceedings 39th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA 1998, pp. 448–457.
[23] D. Juhl, D. M. Warme, P. Winter, and M. Zachariasen, *The GeoSteiner software package for computing steiner trees in the plane: An updated computational study*, Math. Program. Comput. **10** (2018), 487–532.
[24] M. Jünger, A. Martin, G. Reinelt, and R. Weismantel, *Quadratic 0/1 optimization and a decomposition approach for the placement of electronic circuits*, Math. Program. **63** (1994), 257–279.
[25] R. M. Karp, *Reducibility among combinatorial problems*, in *Complexity of computer computations*, R. Miller and J. Thatcher, Eds., Plenum Press, New York, NY, 1972, 85–104.
[26] T. Koch, A. Martin, and S. Voß, *SteinLib: An updated library on Steiner tree problems in graphs. Technical report ZIB-Report 00-37*, Vol **7**, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustr, Berlin, Germany, 2000.
[27] J. Könemann, S. Leonardi, G. Schäfer, and S. van Zwam, *A group-strategyproof cost sharing mechanism for the steiner forest game*, SIAM J. Comput. **37**(5) (2008), 1319–1341.
[28] G. Konjevod, R. Ravi, and F. Salman, *On approximating planar metrics by tree metrics*, Inf. Process. Lett. **80**(4) (2001), 213–219.
[29] M. Leitner, I. Ljubic, M. Luipersbeck, M. Prossegger, and M. Resch, *New real-world instances for the Steiner tree problem in graphs. Technical report*, ISOR Technical Report, Uni Wien, Wien, 2014.
[30] T. L. Magnanti and S. Raghavan, *Strong formulations for network design problems with connectivity requirements*, Networks **45**(2) (2005), 61–79.
[31] PACE *PACE: The parameterized algorithms and computational experiments challenge*, 2018, available at https://pacechallenge.wordpress.com/pace-2018/.
[32] T. Polzin and S. V. Daneshmand, *Improved algorithms for the Steiner problem in networks*, Discret. Appl. Math. **112**(1) (2001), 263–300.
[33] D. Rehfeldt, A generic approach to solving the Steiner tree problem and variants, Master's thesis, Technische Universitat Berlin, 2015.
[34] A. Sadeghi and H. Frohlich, *Steiner tree methods for optimal sub-network identification: An empirical study*, BMC Bioinform **14** (2013), 144.
[35] D. R. Schmidt, B. Zey, and F. Margot, *MIP formulations for the Steiner forest problem*, CoRR (2017), abs/1709.01124.
[36] SteinForestLib *A collection of Steiner forest problems in graphs*, 2020, available at https://github.com/lalehghalami/SteinForestProblem.
[37] SteinLib *A collection of Steiner tree problems in graphs and variants*, 2015, available at http://steinlib.zib.de/steinlib.php.
[38] S. Tazari and M. Müller-Hannemann, *Dealing with large hidden constants: Engineering a planar steiner tree ptas*, ACM J. Exp. Algorithmics **16** (2011).
[39] E. Uchoa and R. F. Werneck, *Fast local search for steiner trees in graphs*, Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX), Austin, Texas, USA, 2010, pp. 1-10.
[40] V. V. Vazirani, *Approximation algorithms*, Springer-Verlag, Berlin/Heidelberg, Germany, 2001.
[41] D. Warme, P. Winter, and M. Zachariasen, *GeoSteiner: Software for computing steiner trees*, 2018, available at http://www3.cs.stonybrook.edu/~algorith/implement/geosteiner/implement%shtml.
[42] D. P. Williamson and D. B. Shmoys, *The design of approximation algorithms*, Cambridge University Press, Cambridge, MA, 2011.

How to cite this article: Ghalami L, Grosu D. Approximation algorithms for Steiner forest: An experimental study. *Networks.* 2021;1–25. https://doi.org/10.1002/net.22046