Private Shared Random Minimum Spanning Forests

Marian Dietz*
ETH Zurich
Zurich, Switzerland
marian.dietz@inf.ethz.ch

Florian Kerschbaum
University of Waterloo
Waterloo, Canada
florian.kerschbaum@uwaterloo.ca

Abstract

Finding the Minimum Spanning Tree or Forest (MSF) of a weighted graph is one of the most fundamental graph problems. It has many applications, and there are various algorithms to solve it in quasi-linear time. However, in a secure computation setting where the graph is *shared* between multiple parties, there are no fully satisfactory solutions. Any prior work on this problem either builds a circuit that is fed into a generic multi-party computation protocol, or is limited to graphs that have a *unique* MSF.

In this work, we first identify privacy and fairness issues that arise when the MSF is not necessarily unique, i.e., there exist duplicate edge weights. Subsequently, we consider the notion of a *Random Minimum Spanning Forest*, which defines a distribution of the desired output in the case where multiple MSFs exist. We carefully design a protocol for this problem in the semi-honest security model.

The main insight of our protocol is that we may reveal certain intermediate results over the entire course of the protocol execution (provably without impacting security), which are then used to make decisions that optimize efficiency. No party learns anything about the inputs of other parties except for the produced MSF, not even the number of input edges. Furthermore, the number of communication rounds is low for many typical graphs, which allows running the protocol even when the network latency is high. Our evaluation shows that, depending on the graph structure and its weight distribution, our protocol can outperform the previous baseline by Laud (PoPETs 2015) by up to 2-3 orders of magnitude in terms of running time.

From another perspective, this work exposes some disadvantages of using generic compilers to obtain MPC protocols, as their efficiency always equal that of the *worst-case* input. Our techniques show that even within the context of MPC, it is possible to obtain a secure protocol whose running time is not fixed a-priori, but instead determined by the output that is not known in advance. By carefully studying the desired functionality, this allows for significant efficiency improvements for any *realistic* inputs.

1 Introduction

The earliest use case for Minimum Spanning Forests (MSF) was the construction of an optimized electricity network with minimum cost [10]. Since then, the same problem has also been applied to computer/telecommunication/water networks, and even less obvious applications like speech recognition or clustering problems [18], all of which can be solved efficiently using common polynomial-time MSF algorithms (e.g. Kruskal's algorithm [22], Prim's algorithm [31], or Borůvka's algorithm [10]).

However, in a modern world where electricity networks already exist, a more difficult problem emerges: suppose that two (or more) companies are considering to *merge* their networks, in an attempt to minimize their combined cost. Each company has a set of existing connections with an associated cost function, and the goal is to find an MSF on the *combined* graph. Furthermore, the parties are not willing to share their entire network, as this might give them a disadvantage in case the merger turns out to be unsuccessful.

Hence, this problem is an example for a use case of *Multi-Party Computation* (MPC): the involved parties need a protocol that computes the MSF on the joint graph, with the guarantee that nobody learns any additional information about the other party's graph.

Protocol Setting. The goal of this paper is to develop a secure and efficient protocol in the MPC setting that solves the MSF problem. We are always going to assume that the graph consists of a *public set of vertices* V, while each involved party p has a *private set of edges* $E^{(p)}$. Jointly, all parties need to learn the MSF on the graph $G = (V, E)$ formed by the set of vertices V and the union $E = E^{(1)} \cup \cdots \cup E^{(k)}$ of the private edge sets.

Note that we could also allow the vertex set V to be private initially. In order to identify vertices from different parties with each other, each vertex would need have to have a corresponding label (from a possibly large domain). Then, since the MSF (including all vertices) needs to be published anyways, we can run a *set union* protocol, to jointly find all vertex labels that occur at least once in any party's input (see e.g. [11]) before starting our MSF protocol given the now public set of vertices.

Further applications. Minimum Spanning Forests are also commonly used to achieve a very simple 2-approximation of the Traveling Salesman Problem (TSP). This is useful especially in the context of multi-party computation, where there is no hope of solving the NP-hard problem of TSP optimally.

TSP itself serves as a problem that many others are easily reduced to, such as vehicle routing, warehouse order-picking, and scheduling problems [27]. Consider *Vehicle Routing*: in this optimization problem, routes have to be assigned to a fleet of vehicles [35]. Some variations of this problem (e.g., if there is one fixed depot with k vehicles) directly fit into the TSP framework, and can be approximated using MSF. Thus, a secure MSF protocol may benefit a set of small carriers who want to share their resources to save costs, while simultaneously revealing data only where it is indeed necessary.

Minimum Spanning Forests are also used to infer information about the *centrality* of the graph's vertices (see e.g. [4, 28]). Hence,

*Work done while at University of Waterloo and Paul G. Allen School of Computer Science & Engineering, University of Washington.

Proceedings on Privacy Enhancing Technologies 2025(2), 157–172

https://doi.org/10.56553/popets-2025-0055

our protocol can be used for privacy-preserving Social Network Analysis. For example, several law enforcement agencies may want to jointly find "central" subjects in a graph that represents relationships between entities. Every agency would have knowledge about a subset of connections, but privacy laws may prohibit them from sharing their entire data [21]. Using our framework, the agencies can compute the MSF on the combined graph, which yields the desired centrality properties.

1.1 The Issue of Duplicate Edge Weights

It is well-known that whenever a graph G contains two or more edges that have some weight w, then the MSF of G may not be uniquely defined. This is a subtle issue, and may arise in all of the applications mentioned above: costs for maintaining certain electricity lines may be identical, the distances between two pairs of physical nodes may be so close to each other that they should be rounded to the same numbers, and edge weights in a social network could simply be small integers that make collisions very likely.

Tie-Breakers. Therefore, when a graph may contain duplicate weights, it becomes necessary to assign *tie-breakers* to all edges. Whenever two edges have the same weight, their tie-breakers determine which one to prefer. In this way we get a total ordering of all edges, resulting in a unique MSF.

Relevant previous work [11] requires tie-breakers to be *publicly computable*. This means that any party needs to have the ability, given all information about a edge (their endpoints, their weights, and the index of the party it belongs to), to calculate its tie-breaker value. This can be done even for "hypothetical" edges that were not part of the input. We are going to argue that this leaks more information than necessary, and potentially impacts fairness.

In the simplest case, suppose that there are two vertices u and v and a small weight w. Then, there are two potential edges e_1 (belonging to party 1) and e_2 (belonging to party 2) with these endpoints and weight. Suppose that e_2 has the larger tie-breaker value $\pi(e_2) > \pi(e_1)$, which is public knowledge. Then, party 2 has an advantage (assuming that e_2 indeed exists): it will learn, *with absolute certainty*, whether party 1 did include e_1 in its input or not (because e_1 would always be preferred over e_2). Clearly, this reveals more information than necessary: in an "ideal" MSF protocol, party 2 should not know that e_1 did not exist just because e_1 is not included in the final MSF. For example, in the setting of two merging electricity networks, this would mean that a network operator with a given connection may infer that the other one is not maintaining an identical connection.

The leakage becomes even worse when considering a larger set of vertices, all reachable from each other using the same small weight w. Anyone can compute a list of edges (which would have low tie-breaker) guaranteed to not exist in the input graph, by only knowing that they did not appear in the MSF.

Furthermore, depending on the implementation of the tie-breaker, one party may have an inherent disadvantage (in the sense that it needs to share many of its edges merely due to the choice of the tie-breaker). Consider two electricity networks, each having one large "hub" with many outgoing edges of the same weight. If the tie-breaker is implemented in such a way that it prefers edges whose endpoints have lower indices, then the MSF will include more edges incident to the lower-index hub.

Random Minimum Spanning Forests. The most natural way of addressing the fairness issue is to assign a *uniformly random* tie-breaker value to every edge. In other words, for every weight w, we pick a uniformly random *permutation* of all edges with this weight, which allows us to decide which edges to prefer. This leads to a more balanced selection of an MSF among the set of all MSFs.

We call an MSF selected by this procedure a *Random Minimum Spanning Forest*, and computing it for a shared graph is the main problem that we are going to study in this work.

Randomness needs to stay hidden! It may be tempting to just let everyone sample random tie-breaker values for their own edges, and then run any standard MSF protocol on the resulting graph (which now has a unique MSF). Unfortunately, this is still provable insecure: Intuitively, when the MSF output contains an edge of weight w which we know had a high tie-breaker value, then the probability of the input containing another weight-w edge connecting the same vertices (e.g. owned by the other party) is very low. (See Appendix A for a formal proof.) This shows why we need to very carefully construct a protocol that does not reveal *any* tie-breaker values, not even for those edges that belong to the MSF output.

Alternatives. Note that our definition of Random MSFs is not completely new: There also exists the notion of Random Spanning Forests [14], a problem in which you are given an unweighted graph, and need to find an MSF on the graph after independently assigning random weights to all edges. However, in our scenario, all edges already have primary weights, and the randomly generated tie-breaker values are only used to compare two edges that have the same weight. Our definition also differs from generating a Uniform Spanning Forest, which selects any MSF from the set of all MSFs with uniform probability (consider a diamond graph as a minimal example on which these two notions differ). However, algorithms generating Uniform Spanning Trees are already expensive in a non-MPC setting and require more advanced techniques like loop-erased random walks (e.g. Wilson's algorithm [36]), and therefore we cannot expect efficient MPC protocols for this problem.

1.2 Related Work

There is a vast amount of literature on various techniques for generic multi-party protocols. Usually they require an algorithm to be written as a binary or arithmetic circuit, which can then be executed in a secure way [15]. This way, the involved parties can jointly compute the output of the algorithm without learning anything about each other's input other than what is already implied by the final output.

One of the most common ways to do this is Yao's garbled circuit method [37] based on binary circuits for two parties in the semi-honest security model. This protocol requires only a constant number of communication rounds, independent of the structure of the circuit. Other protocols depend on the circuit depth [6, 16], and can handle either a binary or arithmetic domain for a varying number of parties [13]. There is a variety of different implementations and frameworks for multi-party computation [19]. Some frameworks also switch between domains (e.g. binary or arithmetic)

and the corresponding protocol during the execution in order to achieve an improved efficiency [12].

Solving Random MSF with Circuits. To solve the MSF problem, any of the standard MSF algorithms (Kruskal, Prim, Borůvka [10, 22, 31]) can easily be turned into a circuit and then transformed into a protocol as decsribed above. However, a huge disadvantage of such an approach is that static circuits do not support random-access memory. Solving this issue naively would require a linear scan per memory access. An alternative solution is oblivious RAM (ORAM) that attempts to emulate a memory in sublinear time per access (as first formalized by [17], and implementable using e.g. [34]). Unfortunately, these type of operations are still a theoretical construct with costs too high to be used in practice. See for example the ObliVM framework [26], in which the authors have also implemented Prim's MSF algorithm.

Secure Graph Computation Frameworks. GraphSC is a general framework for graph algorithms in multi-party computation [30]. Its main feature is the ability of *scattering* data from vertices to incident edges, and *gathering* data from incident edges into a vertex. While these operations can be parallelized for the whole graph, each such operation distributes information only locally. Thus, applying GraphSC to MSF problems would also result in a *linear* number of iterations, and at least *quadratic* communication complexity.

There are multiple other works that focus specifically on dense graphs, but all of them will always have a communication complexity of at least $O(|V|^2)$, which is infeasible for large but sparse graphs. One example is a paper by Blanton et al. [7]. The authors give a protocol solving the MSF problem by simulating Prim's algorithm on a graph given its adjacency matrix. Another example is the work by Anagreh et al. [1], which only targets the MSF problem specifically.

Laud's MSF Protocol. Most relevantly, Laud has designed and implemented a protocol specifically for computing MSFs [24]. Unlike all approaches above (except for the *concretely* highly inefficient ORAM-based version), it has sublinear round complexity and running time that is quasi-linear in $|V| + |E|$. This protocol is based on Awerbuch's MSF algorithm [3], which itself is an adaption of Borůvka's algorithm. The implementation utilizes the Sharemind framework [8]. It fulfills semi-honest security, and works for exactly 3 parties.

The approach by Brickell and Shmatikov. In [11], the authors have a very different approach towards solving graph problems. It is based on the following observation: some values may be revealed as cleartext without waiting for the entire execution to finish, which results in significant efficiency savings later on. For example, consider the single-source-shortest-distance problem. Brickell and Shmatikov note that it is possible to iteratively find *and reveal* the next-closest vertex to all previously discovered ones (there can also be multiple such vertices). Finding them is simple, because both parties can *locally* find the best option among their own edges, and then the best of the two options can be found using a single secure comparison.

The authors also give lightweight MSF protocols based on Prim's and Kruskal's algorithms. Both variants would have a linear number of communication rounds, but we note that it would be possible to utilize the same observations to get a logarithmic-round protocol based on Borůvka's algorithm (see Section 3.1).

Unfortunately, while these protocols are extremely lightweight, they do not have the capability of computing a *Random* MSF.

Table 1: Comparison of MSF protocols, in terms of total communication complexity and the number of communication rounds. We ignore security parameters and other factors that we may treat as constants (such as the bitlength of edge weights). Note that [11] (including its Borůvka variant) does not allow duplicate edge weights.

Protocol	Communication	Rounds
MSF stays secret-shared		
Blanton et al. [7]	$O(\|V\|^2)$	$O(\|V\|^2)$
Anagreh et al. [1]	$O(\|V\|^2)$	$O(\|V\| \log \|V\|)$
ObliVM [26]	$O(\|E\| \log^2 \|V\|)$	$O(1)$
Laud [24]	$O(\|E\| \log^2 \|V\|)$	$O(\log^2 \|V\|)$
MSF will be revealed		
Brickell / Shmatikov [11]	$O(\|V\|)$	$O(\|V\|)$
+ Borůvka (Section 3.1)	$O(\|V\|)$	$O(\log \|V\|)$
This work (Section 3.2)	variable	variable

Comparison. We compare all aforementioned approaches in Table 1. We distinguish between two settings: in the first one, the MSF protocol takes a secret-shared graph as input and outputs the MSF in secret-shared form (this allows using the MSF protocol as a building block of larger protocols). All of these protocols can also handle duplicate edge weights with random tie-breakers. On the other hand, for the protocol by Brickell and Shmatikov [11], it is necessary that each party initially holds a subset of the input graph, and the output will always be an MSF *in the clear*. Our protocol will be in the same category, but unlike [11] it allows duplicate edge weights.

Concrete costs. Note that both ObliVM [26] and Laud's protocol [24] have quasi-linear asymptotic cost. However, ObliVM is very far from practical, as their evaluation shows that computing the MSF of a graph with 1000 vertices and 3000 edges takes over 10 hours [26]. This is several orders of magnitude worse than Laud's protocol [24] (for that reason, we only compare against Laud's protocol in Section 5). While [11] is the concretely most efficient protocol with only one secure comparison per vertex, it has the major disadvantage of requiring distinct edge weights.

1.3 Our Contributions

We present a novel protocol for the Random MSF problem, which can be seen as a tradeoff between Laud's protocol [24] and the approach by Brickell and Shmatikov [11]: on one hand, our protocol can handle edges with duplicate weights by computing Random MSF. On the other hand, its running time is (for many input graphs) more similar to the lightweight variants based on the ideas by Brickell and Shmatikov.

We achieve this speedup by avoiding to secret-share the entire graph. Instead, we analyze the Random MSF problem in detail to find out what information we are allowed to reveal without

affecting security, and how this helps with saving unnecessary secure operations. This has the additional advantage that none of the parties is able to learn anything about the *number of edges* in the other party's input. This is impossible in MPC frameworks which inherently leak the input size. Furthermore, our protocol is round-optimized, so that it may be used with reasonable overhead in a high-latency network.

While all our techniques would work for any number of parties, in order to simplify the presentation we will only describe and evaluate the two-party version. There is more necessity for two-party protocols in practice, because generic MPC baselines are already much more efficient for three or more parties based on an honest majority assumption [20].

Data-dependent running time. An interesting property of our protocol is that its running time (i.e., communication and round complexity) is influenced by the structure and weights of the MSF, which are unknown in the beginning. While it may seem counterintuitive at first, we will show that semi-honest security holds despite this observation, because all leaked information (including the running time) may be deduced from the MSF itself.

In Section 5, we provide an empirical evaluation of our protocol's efficiency. More edges of identical weight will typically lead to increased running time, but we show that for randomly generated graphs with a large range of parameters, and for graphs taken from TSPLIB [32], our protocol performs better than Laud's MSF protocol (we use Laud's method as a baseline because it is the most efficient existing MSF protocol that supports computing a *Random* MSF).

We also believe that our techniques achieving "data-dependent" running time may be applicable to problems other than finding MSFs. For generic MPC compilers or frameworks, if they have a fixed running time, their efficiency is always restricted by the *worst-case* input. However, this work shows that it is possible to create protocols breaking this barrier, by offering improved running time for any *typical* inputs (e.g., graphs with a limited number of duplicate edge weights), while being secure even in the worst-case.

1.4 Paper Organization

We describe all required notation, some background on two-party computation, and the precise definition of the problem we are solving in Section 2. We then present our protocol in Section 3, and describe how to realize its building blocks in Section 4. Section 5 contains details on our implementation and its evaluation, including a comparison with the baseline.

2 Preliminaries

Note that throughout this work, we let $[n] := \{0, 1, \ldots, n-1\}$ for any non-negative integer $n \in \mathbb{N}$.

2.1 Graphs & MSFs

A *graph* $G = (V, E, \mathbf{r}, \mathbf{w})$ is a tuple consisting of a finite set of vertices V, a finite set of edges E, an *endpoint* function $\mathbf{r}$ that maps each edge $e \in E$ to its endpoints $\mathbf{r}(e) = \{u, v\} \subseteq V$ (with $u \neq v$), and a weight function $\mathbf{w} : E \to \mathbb{N}$. (Note that all graphs we consider in this work are weighted and undirected as defined above. They may also contain multiple edges between the same pair of vertices, but self-loops are not allowed.)

For a set of edges $F \subseteq E$, we define its *total weight* as $\mathbf{w}(F) := \sum_{e \in F} \mathbf{w}(e)$. We use $F_{=w}$ to denote the set of *edges restricted to a certain weight* w, i.e., $F_{=w} := \{e \in F \mid \mathbf{w}(e) = w\}$. The notation $\mathbf{r}|_F$ and $\mathbf{w}|_F$ denotes the functions $\mathbf{r}$ and $\mathbf{w}$ restricted to edges F (this will be useful when considering the graph obtained by removing some edges).

Given a set of edges $F \subseteq E$ and any vertex $v \in V$, we define its *boundary*

$$\delta_F(v) := \{e \in F \mid v \in \mathbf{r}(e)\}$$

to be the subset of edges in F that are incident to v.

Sometimes we write $\min\{\mathbf{w}(e) \mid e \in \delta_F(v)\}$ to be the weight of the minimum edge incident to v. If there is no such edge, this minimum is defined to be ∞.

Minimum Spanning Forests. For a graph $G = (V, E, \mathbf{r}, \mathbf{w})$, we call $(e_1, \ldots, e_k)$ ($k \geq 1, e_i \in E, e_i \neq e_j$ for $i \neq j$) a *path* on G, if there exist vertices $v_1, \ldots, v_{k+1}$ with $\mathbf{r}(e_i) = \{v_i, v_{i+1}\}$ for $i \in [k]$. In addition, this path is simultaneously a *cycle*, if $v_1 = v_{k+1}$. We say that two vertices $v, v' \in V$ are *connected* if there exists a path $(e_1, \ldots, e_k)$ with $v = v_1$ and $v' = v_{k+1}$.

To select a single MSF if G has multiple MSFs, we need the notion of a tie-breaker. A *tie-breaker* is a permutation π of the edges E, which assigns a distinct number from $[|E|]$ to every $e \in E$, i.e., $\pi : E \to [|E|]$ is a bijective function. Given a fixed edge set E, there are exactly $|E|!$ of these permutations, and by $E!$ we denote the set consisting of all $|E|!$ permutations.

We can now define a *minimum spanning forest* (MSF) of $G = (V, E, \mathbf{r}, \mathbf{w})$ with tie-breaker $\pi : E \to [|E|]$ as a set $F \subseteq E$ for which the following three conditions hold:

(1) F is *spanning*: every two vertices $v, v' \in V$ that are connected on the graph $(V, E, \mathbf{r}, \mathbf{w})$, are also connected on the graph $(V, F, \mathbf{r}|_F, \mathbf{w}|_F)$,
(2) F is a *forest*: there does not exist any cycle on the graph $(V, F, \mathbf{r}|_F, \mathbf{w}|_F)$, and
(3) F is *minimum*: there is no spanning forest F' fulfilling (1) and (2) with $\mathbf{w}(F') < \mathbf{w}(F)$ or $\mathbf{w}(F') = \mathbf{w}(F)$ and $\sum_{e \in F'} \pi(e) < \sum_{e \in F} \pi(e)$.

For a fixed tie-breaker π, there is a unique minimum spanning forest, which we denote by $\mathsf{MSF}(G, \pi)$.

The *Random MSF* $\mathsf{MSF}(G)$ is a non-deterministic function that samples a uniformly random $\pi \in E!$ and returns $\mathsf{MSF}(G, \pi)$. It is well-known that for a graph with unique edge weights, there is a unique MSF, and therefore the chosen tie-breaker π does not have any effect in such a case (i.e., $\mathsf{MSF}(G)$ would be deterministic).

Merging vertices. For a graph $G = (V, E, \mathbf{r}, \mathbf{w})$, and a set of vertices $c \subseteq V$, we can *merge* the vertices in c. The resulting graph $(V', E', \mathbf{r}', \mathbf{w}') \leftarrow \mathrm{merge}_c(G)$ is naturally defined as follows.

- The new vertex set $V' = (V \setminus c) \cup \{c\}$ contains a single vertex c in place of $|c|$ individual vertices.
- The new edge set $E' = E \setminus \tilde{E}$ is as before, except that we remove any introduced self-loops $\tilde{E} = \{e \in E \mid \mathbf{r}(e) \subseteq c\}$.
- For any edge e that previously connected $\mathbf{r}(e) = \{u, v\}$ for some $v \in c$, we modify its endpoints to be $\mathbf{r}'(e) = \{u, c\}$. All other endpoints are unmodified.
- No weights are modified: $\mathbf{w}' = \mathbf{w}|_{E'}$.

2.2 Two-Party Computation

In two-party computation, a *functionality* f is a random function that specifies the desired output distribution given the two inputs. Note that we require $f(x, y)$ to be a random variable, since in our case of random MSF's, the output may involve randomness. Public input is achievable by letting it be contained of both inputs x and y. For a protocol Π correctly implementing the functionality f, the notion of semi-honest security is given in Definition 2.1, which follows the definitions in prior work [15, 25].

Definition 2.1. For a set A of valid inputs, we say that the two ensembles $X = \{X(\lambda, a)\}_{\lambda\in\mathbb{N}, a\in A}$ and $Y = \{Y(\lambda, a)\}_{\lambda\in\mathbb{N}, a\in A}$ are computationally indistinguishable (denoted by $X \overset{c}{\equiv} Y$), if for every non-uniform polynomial-time algorithm D, there exists a negligible function $\mathrm{negl}(\cdot)$, s.t. for every $a \in A$ and $\lambda \in \mathbb{N}$:

$$|\Pr[D(X(\lambda, a)) = 1] - \Pr[D(Y(\lambda, a)) = 1]| \leq \mathrm{negl}(\lambda)$$

A protocol Π *privately computes* f *in the semi-honest security model*, if there exists a polynomial-time algorithm S_1 s.t.

$$\begin{aligned}&\{(S_1(1^\lambda, x, f(x, y)), f(x, y))\}_{\lambda,x,y}\\ &\overset{c}{\equiv} \{(\mathsf{view}_1^\Pi(\lambda, x, y), \mathsf{output}_2^\Pi(\lambda, x, y))\}_{\lambda,x,y}\,,\end{aligned} \tag{1}$$

where $\lambda \in \mathbb{N}$, and x, y are inputs on which $f(x, y)$ is defined, and the same indistinguishability holds with parties 1 and 2 swapped.

For our case of semi-honest security, protocols can easily be composed, in the sense that a protocol can call other semi-honest secure protocols as a subroutine [15].

2.3 Definition of Random MSF Functionality

We consider the problem of sampling a Random MSF inside a two-party protocol, where the set of vertices V is public, and each party $p \in \{1, 2\}$ has a private set of edges $E^{(p)}$ (s.t. $E^{(1)}$ and $E^{(2)}$ are disjoint) with corresponding endpoints $\mathbf{r}^{(p)}$ and weights $\mathbf{w}^{(p)}$. The protocol should be computing a Random MSF of the graph $(V, E^{(1)} \cup E^{(2)}, \mathbf{r}^{(1)} \cup \mathbf{r}^{(2)}, \mathbf{w}^{(1)} \cup \mathbf{w}^{(2)})$, i.e., on the graph with vertices V and edges resulting from taking the union of the two private edge sets.

Functionality 1 Random Minimum Spanning Forest

Public: Set of vertices V
functionality RandomMSF($E^{(1)}, E^{(2)}$)
  return $\mathsf{MSF}(V, E^{(1)} \cup E^{(2)}, \mathbf{r}, \mathbf{w})$
end functionality

This desired behavior is formalized in Functionality 1. For simplicity, we are going to assume that $E^{(1)} \subseteq \binom{V}{2} \times \mathbb{N} \times \{1\}$, i.e., each edge $e \in E^{(1)}$ belonging to party 1 can be written as $e = (\{u, v\}, w, 1)$, s.t. $\mathbf{r}(e) = \{u, v\}$ and $\mathbf{w}(e) = w$. Similarly, we also assume that $E^{(2)} = \binom{V}{2} \times \mathbb{N} \times \{2\}$, i.e., every $e \in E^{(2)}$ has the form $e = (\{u, v\}, w, 2)$, where $\mathbf{r}(e) = \{u, v\}$ and $\mathbf{w}(e) = w$. As a result, there can be at most two different edges between the same endpoints $u, v \in V$ and with the same weight $w \in \mathbb{N}$: one belonging to party 1, and one belonging to party 2.

The protocol should return an MSF according to the distribution $\mathsf{MSF}(V, E^{(1)} \cup E^{(2)}, \mathbf{r}, \mathbf{w})$, where $\mathbf{r}$ and $\mathbf{w}$ are defined by $\mathbf{r}(e) = \{u, v\}$ and $\mathbf{w}(e) = w$ for any edge $e = (\{u, v\}, w, p) \in \binom{V}{2} \times \mathbb{N} \times \{1, 2\}$. As V is public, we also write an invocation as $\textsc{RandomMSF}_V(E^{(1)}, E^{(2)})$. Note that there is no need to take endpoints $\mathbf{r}$ or weights $\mathbf{w}$ as an argument, this information can be inferred from the edges themselves, which have the form $(\{u, v\}, w, p)$. Furthermore, they will also gain knowledge about *who* this edge belongs to (party p). This is important whenever two parties both have an edge with identical endpoints and weight. They will know whether their own or the opposite party's edge was selected for the MSF. For convenience, we let $E := E^{(1)} \cup E^{(2)}$ denote the combined set of all edges given in the input.

2.4 Framework

As for any type of MPC, our protocols will be working on secret values, which can be manipulated whenever two parties agree to perform some operation on it. Thus, there are two different kinds of values used in the pseudocode of a protocol: plaintext values known to either of the two or to both parties, and secret-shared values not known to anyone. We denote a secret shared value by $[\![a]\!]$, and if party p constructs secret shares of a value known to it, we write $\textsc{Share}_p(a)$. $\textsc{Reveal}_p([\![a]\!])$ reveals secret-share $[\![a]\!]$ to party p (i.e., the other party sends its share of $[\![a]\!]$ to party p), and $\textsc{Reveal}([\![a]\!])$ reveals it to both parties (i.e., both parties send their respective share to the other party).

We will use the GMW protocol by Goldreich, Micali, and Wigderson [16] as a black-box to implement the underlying secret-shares. This decision is primarily for efficiency reasons: most of the operations required by our protocol are cheaper in binary form than in arithmetic form.

Linear operations on secret-shared bits (e.g. XOR and negation) are free and do not require any communication. Multiplications can be constructed from 1-out-of-4 oblivious transfer [15]. However, in order to optimize the online phase, we use Beaver triples, i.e., precalculated random multiplication triples (which are independent of the circuit) that can be used to mask real multiplications when running the actual protocol [5]. Multiplications allow us to perform bit-and, bit-or, and multiplexers on secret-shared bits.

In contrast to constant-round protocols (such as Yao's garbled circuits), we have an additional overhead resulting from the number of communication rounds, which is equal to the *multiplicative depth* of the circuit that is run. Therefore, we need to ensure that, besides from minimizing the number of bit multiplications, we also keep the multiplicative depth reasonably low.

Secret-shared integers of bitlength B can be implemented as B separate secret-shared bits. For the most common low-level operations, we get the following efficiencies by utilizing depth-optimized circuits [33]. Arithmetic operations (addition and subtraction) require about $\log B$ communication rounds, and consume $O(B \log B)$ multiplication triples. Comparisons (equality, greater than, ...) require about $\log B$ communication rounds, and consume $O(B)$ multiplication triples.

3 Our Random MSF Protocol

In this section we present our main protocol. In Section 3.1, we start by describing a simpler protocol based on Borůvka's MSF algorithm for graphs with *unique edge weights* that utilizes ideas

similar to those used by Brickell and Shmatikov [11]. Then, Section 3.2 gives an overview of the full protocol that allows duplicate weights. Section 3.3 contains a formalized version, and then we describe optimizations in Section 3.4. We show semi-honest security in Appendix C.

In Appendix A, we additionally argue why the naive idea, which would let each party assign a random tie-breaker to their local edges and then run e.g. the protocol in Section 3.1, would not be secure.

3.1 Warm-up: Finding the MSF of a Graph with Unique Weights

In their paper on MPC protocols for graph algorithms, Brickell and Shmatikov describe protocols for computing an MSF (when the graph has unique weights) based on either Prim's or Kruskal's algorithm [11]. However, both implementations would result in linear round complexity. We now describe an adaptation of this protocol, based on Borůvka's MSF algorithm [10], which would have logarithmic round complexity.

Borůvka's MSF algorithm. Borůvka's algorithm for finding the unique MSF F on a graph $G = (V, E, \mathbf{r}, \mathbf{w})$ with unique edge weights, repeats the following two steps after initializing $F \leftarrow \emptyset$.

- For all vertices $v \in V$ with at least one incident edge, we find the *best* incident edge $e \in \delta_E(v)$ (i.e., $\mathbf{w}(e) < \mathbf{w}(e')$ for all other edges $e' \in \delta_E(v)$, $e' \neq e$). We add e to the MSF F.
- For all edges e found in the previous step, we merge the two endpoints $\mathbf{r}(e) = \{u, v\}$ together: $G \leftarrow \text{merge}_{\{u,v\}}(G)$.

These steps are repeated until no edges are found anymore (i.e., $E = \emptyset$). The set F then forms the MSF.

Note that an edge e may either be selected by only one of its endpoints, or by both of them in the same iteration (in which case the merge-step for e is only run once). In contrast to Prim's and Kruskal's algorithms, each iteration can be parallelized (by computing the best incident edges simultaneously for all vertices), and adds at least $\lceil \frac{|V|}{2} \rceil$ new edges to the MSF. Therefore, there will be at most $\log_2 |V|$ iterations.

Correctness follows from the fact that each selected edge must be contained in the MSF. More specifically,

$$MSF(G) = \{e\} \cup \mathsf{MSF}(\text{merge}_{\{u,v\}}(G))$$

for any edge $e \in \delta_E(v)$ (with endpoints $\mathbf{r}(e) = \{u, v\}$) that is the best incident edge for some vertex $v \in V$. This observation is similar to those required for correctness of Prim's and Kruskal's algorithms.

Transformation into a 2-party protocol. Applying ideas from [11] to the algorithm described above, we get a protocol for computing the MSF on a graph with unique weights, which repeats the following steps after initializing $F \leftarrow \emptyset$.

- Each party p *locally* finds for each vertex $v \in V$ the best edge $e_v^{(p)} \in E^{(p)}$ of smallest weight $\mathbf{w}(e)$ incident to v.
- For each $v \in V$, the two parties compare the two edges $e_v^{(1)}$ and $e_v^{(2)}$ to check one has the smaller weight. They do so by utilizing any MPC protocol for integer comparison. The edge weights are not revealed, but only the comparison outcome.
- The party p whose edge was better sends all information about $e_v^{(p)}$ (i.e., endpoints and weight) to the other party.

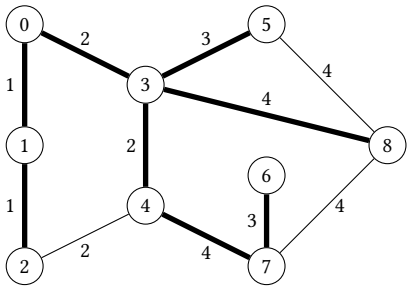

Figure 1: This is an example for a graph in which weights occur more than once. There are exactly 15 different MSF's, each of them would contain all edges of weight 1 and 3, any two edges of weight 2, and two additional edges of weight 4 that do not result in a cycle. One possible MSF is indicated by the thick edges.

- The endpoints $\mathbf{r}(e) = \{u, v\}$ of all edges e discovered in the previous step are merged together.

Assuming that there is a protocol for comparing two edge weights that runs in k communication rounds, then the description above yields an MSF protocol of $\leq k \cdot \log |V|$ rounds. Also note that the protocol is very lightweight in terms of total communication complexity, because $|V| \cdot \log |V|$ secure comparisons are all that is needed.

3.2 Overview of Our MSF Protocol

The simple protocol above does not work anymore as soon as the graph contains more than one edge with some particular weight (in fact, not even Borůvka's algorithm would work anymore by itself). The issue is that for a vertex $v \in V$, there may be multiple edges (say, e_v and e'_v) incident v, while all of them have the same best weight $\mathbf{w}(e_v) = \mathbf{w}(e'_v)$. Then, to avoid cycles in the MSF, one of these edges needs to be picked through employing a tie-breaker. Our protocol is going to achieve this for a uniformly random tie-breaker π, without leaking anything about π.

Isolatable subgraphs. In our protocol, we will rely on what we call an *isolatable subgraph*:

Definition 3.1. Given a graph $G = (V, E, \mathbf{r}, \mathbf{w})$, an *isolatable subgraph of weight w* is a set $c \subseteq V$ of vertices of size ≥ 2, s.t.

- all vertices in c are pairwise reachable from each other using only edges of weight *exactly* w, while none of them is incident to any edge of weight $< w$, and
- all edges $e \in E$ that *leave* c (i.e., e has exactly one endpoint in c) have weight $\mathbf{w}(e) > w$.

Consider the graph in Figure 1, which will be our running example. The two vertex sets $\{0, 1, 2\}$ and $\{6, 7\}$ are both isolatable subgraphs of weights 1 and 3, respectively (in fact, these are the *only* two isolatable subgraphs).

An isolatable subgraph c has the following interesting property, which we show in Appendix B:

Lemma 3.2. *Given a graph $G = (V, E, \mathbf{r}, \mathbf{w})$ and an isolatable subgraph $c \subseteq V$ of weight w, the distribution of MSF(G) is identical*

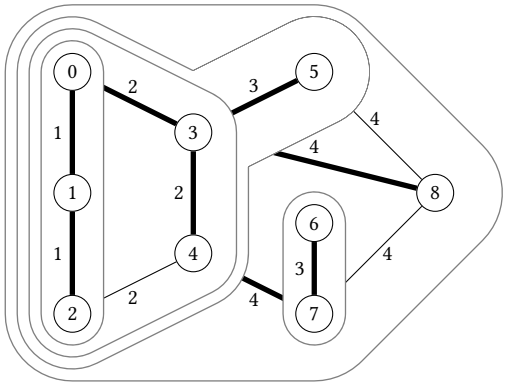

Figure 2: For the example graph given in Figure 1, this figure depicts the isolatable subgraphs and their merging process. In the beginning, $\{0,1,2\}$ and $\{6,7\}$ are isolatable subgraphs of weights 1 and 3, respectively. After merging $\{0,1,2\}$, we see that $\{\{0,1,2\},3,4\}$ is a new isolatable subgraph of weight 2. After merging those vertices, $\{\{\{0,1,2\},3,4\},5\}$ is a new isolatable subgraph of weight 3. Finally, the three vertices $\{\{\{\{0,1,2\},3,4\},5\},\{6,7\},8\}$ form an isolatable subgraph of weight 4. After merging these, only one vertex remains. Generating and combining Random MSF's for all isolatable subgraphs encountered during this process may result in the thick edges, which together form an MSF.

to the distribution of

$$MSF(c, \tilde{E}, \mathbf{r}|_{\tilde{E}}, \mathbf{w}|_{\tilde{E}}) \cup MSF(merge_c(G)) \ ,$$

which samples the two MSFs on c and $merge_c(G)$ independently of each other, and then takes their union. The set $\tilde{E} = \{e \in E_{=w} \mid \mathbf{r}(e) \subseteq c\}$ denotes the set of all edges between two vertices in c.

This lemma tells us that we can regard the tie-breakers of the edges in c and the tie-breakers of the remaining edges in $\text{merge}_c(G)$ completely independent of each other! In other words, we do not need to sample the tie-breaker permutation for the entire graph G, but instead we can simply compute Random MSFs on c and on $\text{merge}_c(G)$ independently of each other and then merge the results. This process will result in exactly the same output distribution as directly computing a Random MSF on the entire graph G.

The above can also be seen as a generalization of the observation behind Kruskal's algorithm for unique-weight graphs: this algorithm would repeatedly choose the globally lowest-weight edge into the MSF, and merge its two endpoints. In our case, we choose a *set* of vertices connected through low weights, add their MSF, and merge all vertices in it together.

With this in mind, our protocol will naturally follow the following strategy: find any isolatable subgraph c, compute a Random MSF for it, and then continue on the remaining graph $\text{merge}_c(G)$. This will be repeated until no edges remain (note that whenever there exists at least one edge, there must also be at least one isolated component). See Figure 2 for an example.

The first question is: Even if we are able to find isolatable subgraphs, *would it be secure reveal them to both parties?* The answer to this is yes – irrespectively of the exact output of $\text{MSF}(c, \tilde{E}, \mathbf{r}|_{\tilde{E}}, \mathbf{w}|_{\tilde{E}})$ for isolatable subgraphs c, the merging sequence will be identical. Given only the final MSF (i.e., the protocol's output), it will be possible to "retrace" isolatable subgraphs. For example, in Figure 2, if we were to remove all non-MSF edges, we can see that $\{0,1,2\}$ and $\{6,7\}$ are still the only isolatable subgraphs, and similarly after performing the merges.

Finding isolatable subgraphs. Now we discuss how our protocol finds all isolatable subgraphs in a secure way.

Step 1: Similar to the Borůvka-based algorithm (Section 3.1), we first compute and reveal for each vertex $v \in V$ the value

$$w_v = \min\{\mathbf{w}(e) \mid e \in \delta_E(v)\} \ ,$$

i.e., the weight of its best incident edge. Doing so is simple, assuming the existence of an integer comparison protocol: each party p locally computes $w_v^{(p)} := \min\{\mathbf{w}(e) \mid e \in \delta_{E^{(p)}}(v)\}$, and secret-shares this value. Then, they securely compute $\min([\![w_v^{(1)}]\!], [\![w_v^{(2)}]\!])$, and reveal the outcome.

Revealing these values is secure, because they can be simulated given *any* MSF (see Lemma C.1 for details). In addition, the above can be executed for all vertices in parallel, yielding a protocol with a constant number of communication rounds!

Step 2: We can now (locally, without communication) group the vertices into sets $S_w = \{v \in V \mid w_v = w\}$ of vertices with the same best weight w_v. For example, in Figure 2, we would get $S_1 = \{0,1,2\}$, $S_2 = \{3,4\}$, $S_3 = \{5,6,7\}$, and $S_4 = \{8\}$.

All of these sets contain *candidates* in the following sense: each isolatable subgraph (i.e., $\{0,1,2\}$ and $\{6,7\}$) needs to be a subset of one of the sets S_w. But how can we "extract" isolatable subgraphs? We will do so with a sub-protocol that implements Functionality 2.

Functionality 2 Connectivity

Public: Set of vertices V, and a subset $S \subseteq V$
functionality $\textsc{Connectivity}((E^{(1)}, \mathbf{r}^{(1)}), (E^{(2)}, \mathbf{r}^{(2)}))$
Remove all vertices from S that may reach any vertex in $V \setminus S$ through edges in $E^{(1)} \cup E^{(2)}$
Partition S into disjoint sets $c_1 \sqcup \cdots \sqcup c_\ell = S$, s.t. any two vertices $u, v \in S$ are in the same c_i iff they are reachable from each other through edges in $E^{(1)} \cup E^{(2)}$
return $c_1, \ldots, c_\ell$
end functionality

We will abuse notation by writing $\textsc{Connectivity}_{V,S}(E, \mathbf{r})$ for the combined set of edges $E = E^{(1)} \cup E^{(2)}$. This is defined as running Connectivity on the input $(E^{(1)}, \mathbf{r}^{(1)}), (E^{(2)}, \mathbf{r}^{(2)})$.

Now we can see that $\textsc{Connectivity}_{V,S_w}(E_{=w}, \mathbf{r}|_{E_{=w}})$ directly gives us all isolatable subgraphs of weight w (because each c_i is internally connected using edges of weight w, and no edge of weight w leaves c_i). For example, with the graph in Figure 2, we would reveal the following values to both parties:

$$\textsc{Connectivity}_{V,\{0,1,2\}}(E_{=1}, \mathbf{r}|_{E_{=1}}) \to \{0,1,2\}$$
$$\textsc{Connectivity}_{V,\{3,4\}}(E_{=2}, \mathbf{r}|_{E_{=2}}) \to \bot$$
$$\textsc{Connectivity}_{V,\{5,6,7\}}(E_{=3}, \mathbf{r}|_{E_{=3}}) \to \{6,7\}$$
$$\textsc{Connectivity}_{V,\{8\}}(E_{=4}, \mathbf{r}|_{E_{=4}}) \to \bot$$

As for step 1, we need to argue about why revealing these outputs to both parties is secure. The reason is again that this information may be inferred from any MSF (i.e., running Connectivity as above using only edges from any given MSF will yield the exact same output). We formally show this in Lemma C.2.

Functionality 2 can be realized in different ways, which we discuss in Section 4.1.

Repeat: After completing step 1 and step 2, we merge all isolatable subgraphs that have been found. We repeat until convergence (i.e., no edges remain).

Combining MSFs: The final MSF will be computed as the union of the individual Random MSFs found for each isolatable subgraph. We discuss a protocol IsolatedMSF to compute a Random MSF for an isolatable subgraph c in Section 4.2.

3.3 Formalized MSF Protocol

Protocol 3 formally describes our Protocol. The initial state of endpoints $\mathbf{r}^{(1)}$, $\mathbf{r}^{(2)}$, and weights $\mathbf{w}^{(1)}$, $\mathbf{w}^{(2)}$ are inferred directly from the edges, which have the form $e = (\{u, v\}, w, p)$.

Protocol 3 Random Minimum Spanning Forest

1: **Public:** V
2: **protocol** RandomMSF($E^{(1)}$, $E^{(2)}$)
3: $C \leftarrow \emptyset$ ▷ set of isolatable subgraphs, handled at the end
4: **while true do**
5: $W \leftarrow \emptyset$ ▷ set of weights seen during this iteration
6: **for** $v \in V$ (in parallel) **do**
7: **Party** p: $w_v^{(p)} \leftarrow \min\{\mathbf{w}(e) \mid e \in \delta_{E^{(p)}}(v)\}$
8: $w_v \leftarrow \text{Reveal}(\min(\text{Share}_1(w_v^{(1)}), \text{Share}_2(w_v^{(2)})))$
9: **if** $w_v \neq \infty$ **then** $W \leftarrow W \cup \{w_v\}$ **end if**
10: **end for**
11: **if** $W = \emptyset$ **then break end if**
12: **for** $w \in W$ (in parallel) **do**
13: $S_w \leftarrow \{v \in V \mid w_v = w\}$
14: $c_1, \ldots, c_\ell \leftarrow \text{Connectivity}_{V,S_w}((E^{(1)}_{=w}, \mathbf{r}^{(1)}|_{E^{(1)}_{=w}}), (E^{(2)}_{=w}, \mathbf{r}^{(2)}|_{E^{(2)}_{=w}}))$
15: **for** $i = 1, \ldots, \ell$ **do**
16: **Party** p: $\tilde{E}^{(p)} \leftarrow \{e \in E^{(p)}_{=w} \mid \mathbf{r}^{(p)}(e) \subseteq c_i\}$
17: $C \leftarrow C \cup \{(c_i, (\tilde{E}^{(1)}, \mathbf{r}^{(1)}|_{\tilde{E}^{(1)}}), (\tilde{E}^{(2)}, \mathbf{r}^{(2)}|_{\tilde{E}^{(2)}}))\}$
18: **Party** p: $(V, \tilde{E}^{(p)}, \tilde{\mathbf{r}}^{(p)}, \tilde{\mathbf{w}}^{(p)}) \leftarrow \text{merge}_{c_i}(V, E^{(p)}, \mathbf{r}^{(p)}, \mathbf{w}^{(p)})$
19: **end for**
20: **end for**
21: **end while**
22: $F \leftarrow \emptyset$ ▷ the MSF
23: **for** $(c, (\tilde{E}^{(1)}, \tilde{\mathbf{r}}^{(1)}), (\tilde{E}^{(2)}, \tilde{\mathbf{r}}^{(2)})) \in C$ (in parallel) **do**
24: $F \leftarrow F \cup \text{IsolatedMSF}_c((\tilde{E}^{(1)}, \tilde{\mathbf{r}}^{(1)}), (\tilde{E}^{(2)}, \tilde{\mathbf{r}}^{(2)}))$
25: **end for**
26: **return** F
27: **end protocol**

The loop in lines 6 to 10 resembles step 1 as described in Section 3.2, i.e., it computes the best incident weight w_v for each vertex $v \in V$. The loop in lines 12 to 20 resembles step 2: for each group $S_w \subseteq V$ of vertices that have the same best weight w, we utilize Connectivity to find all subsets $c_1, \ldots, c_\ell$ of S_w that form isolatable subgraphs. All isolatable subgraphs are then merged. Finally, in lines 22 to 25, we then compute individual Random MSFs separately for every isolatable subgraph c.

Correctness. By definition of w_v and Connectivity, it is clear that each c_i will always be an isolatable subgraph of the current state of the graph. Therefore, correctness follows directly from Lemma 3.2 (which is proven in Appendix B).

Security. In Appendix C, we prove that Protocol 3 fulfills semi-honest security.

3.4 Optimizations

It is not guaranteed that every vertex $v \in V$ will always be contained in some isolatable subgraph. For any given weight w, this affects exactly all vertices in $S_w \setminus c_1 \setminus \cdots \setminus c_\ell$. These vertices will stay untouched in the current iteration, which potentially leads to more iterations and hence more communication rounds.

However, in many cases the following optimization is useful: Suppose $W = \{w_1, \ldots, w_k\}$ is the set of all weights that have been discovered in the current iteration, sorted in increasing order. Furthermore, suppose that there was only *one* isolatable subgraph c of the smallest weight w_1.

Now, let $d := S_{w_2} \setminus c_1 \setminus \cdots \setminus c_\ell$ be the set of vertices with second-best weight not covered by any isolatable subgraph. Then, after the current iteration, we know for sure that $\{c\} \cup d$ forms an isolatable subgraph. Therefore, we may take care of this isolatable subgraph immediately, by adding it to C and merging its vertices.

To see why $\{c\} \cup d$ is an isolatable subgraph, recall that all vertices in d must be reachable by some vertex in $V \setminus d$ through edges of weight w_2. However, the only candidates for vertices connected to d through edges of weight w_2 are in c (because all other vertices in $S_{w_3}, \ldots, S_{w_k}$ do not have any incident edges of weight w_2).

For example, revisit the graph in Figure 2. The smallest weight is $w_1 = 1$, and the second-smallest weight is $w_2 = 2$. While $c = \{0, 1, 2\}$ is an isolatable subgraph, the two vertices in $d = \{3, 4\}$ are not part of any isolatable subgraph. However, after merging $\{0, 1, 2\}$, the set $\{\{0, 1, 2\}, 3, 4\}$ must now form an isolatable subgraph of weight 2.

The optimization described above may be generalized: after merging $\{c\} \cup d$, if there was no other isolatable subgraph of weight w_2, then the resulting vertex will form an isolatable subgraph together with all untouched vertices in S_{w_3}, and so on. Thus, in the example graph in Figure 1, this optimization would allow not only for merging $\{\{0, 1, 2\}, 3, 4\}$, but also for merging $\{\{\{0, 1, 2\}, 3, 4\}, 5\}$ afterwards.

The described optimization does not require any communication, because it can be done *locally* by each party. On the other hand, note that the described optimization is *heuristic* in the sense that it may not be useful for all input graph. However, our evaluation (see Section 5) showed that for "normal" graphs (e.g., randomly generated graphs), it may significantly reduce the number of rounds.

Other practical considerations. Orthogonally to the optimization above, note that it can happen that the protocol attempts to compute $w_v = \min\{\mathbf{w}(e) \mid e \in \delta_E(v)\}$ multiple times for the same vertex $v \in V$. However, since it is guaranteed that the value of w_c does not change, this recomputation does not need to take place explicitly.

The values from the previous iteration can be cached and re-used. The same can be done for any calls to Connectivity.

4 Building Blocks

Our Random MSF protocol described in Section 3 makes use of two building blocks – Connectivity and IsolatedMSF. Both of these subprotocols work on subgraphs of the original graph (S_w in the case of Connectivity, and c in the case of IsolatedMSF).

In both subprotocols, all input edges will have the same weight. Unfortunately, this means that there are no tricks (unlike in our main protocol) to reveal any intermediate values to speed up the protocol without leaking any information. Furthermore, there is another difficulty: Because Connectivity and IsolatedMSF are only called for subgraphs, they even need to ensure that the input size (i.e., the number of edges in those subgraphs) stays hidden. Otherwise, an adversary may gain too much information – even if the number of edges in the *original* graph is allowed to be leaked.

As a result, the two protocols described in this section are expensive: the number of bit multiplications is in the order of k^3, where k is the number of vertices in the subgraph. This is why our main protocol has its best performance when there is a low number of edges with the same weight. Adding more edges with equal weight requires running these two subprotocols on larger instances.

4.1 Implementing Connectivity

We first describe ways of implementing Functionality 2. Publicly given is the set of vertices V, and a subset $S \subseteq V$ of size $|S| = k$. The private input for party p is a set of edges $E^{(p)}$.

Because the size $|E^{(p)}|$ needs to stay hidden, we work on adjacency *matrices* instead of lists: The first step is to secret-share the (symmetric) adjacency matrix $M \in \{0,1\}^{(k+1)\times(k+1)}$, where $M[i,j]$ denotes whether there exists at least one edge between the i-th vertex in S and the j-th vertex in S. Note that we consider $V \setminus S$ as one single vertex that corresponds to be the $(k+1)$-th row and column in M. The secret-sharing of M can be easily obtained by having each party compute $M^{(p)}$ locally (i.e., the adjacency matrix when restricted to edges in $E^{(p)}$), and then taking the bit-wise OR.

The goal is to jointly compute $[\![M^k]\!]$, because $M^k[i,j]$ denotes whether there exists a path of length $\leq k$ between the i-th and the j-th vertex. Because no simple path can be longer than k edges, the entry $M^k[i,j]$ denotes whether i and j are reachable from each other using *any* path. Thus, given the revealed matrix M^k, any party can compute the sets $c_1, \dots, c_\ell$.

We propose two different ways of computing $[\![M^k]\!]$.

First option. We use *repeated squaring* on matrices. That is, we compute $\lfloor \log_2 k \rfloor$ matrices

$$[\![M]\!], [\![M^2]\!], [\![M^4]\!], [\![M^8]\!], \dots ,$$

where $[\![M^{2\ell}]\!] \leftarrow [\![M^\ell]\!] \cdot [\![M^\ell]\!]$. Then, $[\![M^k]\!]$ can be computed as the product of a subset of these matrices.

Each matrix multiplication takes $O(k^3)$ multiplications over $O(\log k)$ rounds. As a result, the overall communication cost is $O(k^3 \log k)$, and the number of rounds is $O(\log^2 k)$ (because all $[\![M^{2^\ell}]\!]$'s need to be computed *sequentially*).

Second option. Here, we iteratively compute

$$[\![A_\ell]\!] := [\![(M[:\ell+1,:\ell+1])^\ell]\!]$$

for increasing $\ell \in [k]$, where $M[:\ell+1,:\ell+1]$ denotes the first $\ell+1$ rows and columns of M. Thus, each matrix A_ℓ contains information about which pairs among the first $\ell+1$ vertices in S are connected to each other (using only edges among these $\ell+1$ vertices). Repeating this for k times, we will get the desired matrix $[\![A_k]\!] = [\![M^k]\!]$.

For a fixed ℓ, the corresponding matrix A_ℓ is computed as follows: first, we calculate

$$[\![A_\ell[i,\ell+1]]\!] \leftarrow \bigvee_{j\in[\ell]} [\![A_{\ell-1}[i,j]]\!] \wedge [\![M[j,\ell+1]]\!]$$

for each $i \in [\ell]$, which denotes whether the "new" vertex $\ell+1$ is reachable from the i-th vertex. Then, we use this to update

$$[\![A_\ell[i,j]]\!] \leftarrow [\![A_{\ell-1}[i,j]]\!] \vee (A_\ell[i,\ell+1] \wedge A_\ell[j,\ell+1])$$

for all $i,j \in [\ell]$, to take paths through the $(\ell+1)$-th vertex into consideration for the reachability between i and j.

Each iteration requires $O(i^2)$ multiplications, and $\log i$ rounds. Thus, the overall communication is $O(k^3)$ multiplications, and the number of rounds is $O(k \log k)$.

Summary. The first approach has lower round complexity, while the second approach has lower communication complexity. In our implementation we utilize the second option, because the protocol is only feasibly for small k anyways, so the round complexity of $O(k \log k)$ is not a significant issue.

4.2 Implementing IsolatedMSF

Note that we can think of IsolatedMSF as a protocol that computes a Random MSF on an unweighted graph. Hence, it crucially depends on tie-breaker values. Unfortunately, we cannot simply sample tie-breaker values for all edges, because the running time of IsolatedMSF needs to be independent of the number of edges!

Our implementation loosely follows Kruskal's algorithm [22] for computing an MSF. Interestingly, Kruskal does not need to know all edge weights (or, in our case, tie-breakers) at the same time—it suffices to know *which edge currently has the smallest weight.* Its endpoints are then merged together. Hence, in our protocol, we just pick the "best" edge *on-the-fly*: we sample a uniformly random edge (whose endpoints are not merged yet) and pretend that its tie-breaker is smaller than all others. This eliminates the need to assign tie-breakers to all edges.

A formal version is described in Protocol 4. It uses the following secret-shared variables:

- u_i (for $i \in [n]$) denotes the union-find data structure. Two vertices i and j are reachable from each other using previously selected edges iff $u_i = u_j$.
- $a_{d(i,j,p)}$ denotes the number of edges in $E_{ij}^{(p)}$, which is the set of edges in $E^{(p)}$ that connect vertices i and j. However, whenever vertices i and j become reachable from each other through previously selected edges, we update $a_{d(i,j,p)} \leftarrow 0$.
- $s_{d(i,j,p)}$ denotes the index of the edge within $E_{ij}^{(p)}$ that has been selected to be part of the MSF (or $\perp$, if no edge from $E_{ij}^{(p)}$ has been selected).

Here, $d(i,j,p) = (p-1)\cdot\frac{n(n-1)}{2} + \frac{j\cdot(j-1)}{2} + i$ is a bijection, which for every pair $0 \le i < j < n$ and every $p = 1, 2$, assigns a different index between 0 and $N-1$, where $N := n\cdot(n-1)$. This allows us to write a and s as *flattened* arrays of lengths N.

Protocol 4 Random Spanning Forest for an unweighted graph

1: **Public:** Set of vertices V
2: **protocol** IsolatedMSF($(E^{(1)}, \mathbf{r}^{(1)})$, $(E^{(2)}, \mathbf{r}^{(2)})$)
3: $[\![u_i]\!] \leftarrow [\![i]\!] \quad \forall i \in [n]$
4: $[\![a_{d(i,j,p)}]\!] \leftarrow \text{Share}_p(|E_{ij}^{(p)}|)$, $[\![s_{d(i,j,p)}]\!] \leftarrow [\![\bot]\!]$ } $\forall 0 \le i < j < n, p = 1, 2$
5: **for** $n-1$ times **do**
6: $[[\![A_0]\!], \ldots, [\![A_{N-1}]\!]] \leftarrow \text{PrefixSum}([[\![a_0]\!], \ldots, [\![a_{N-1}]\!]])$
7: $[\![r]\!] \leftarrow \text{RandInt}([\![A_{N-1}]\!])$
8: $[\![g_z]\!] \leftarrow [\![A_z]\!] \stackrel{?}{>} [\![r]\!] \quad \forall z \in [N]$
9: $(([\![\tilde{u}]\!], [\![\tilde{v}]\!]), [\![\tilde{g}]\!]) \leftarrow \text{FindFirst}([([([\![u_i]\!], [\![u_j]\!]), [\![g_z]\!])_{z\in[N]}])$
10: $[\![u_i]\!] \leftarrow$ **if** $[\![u_i]\!] \stackrel{?}{=} [\![\tilde{v}]\!]$ **then** $[\![\tilde{u}]\!]$ **else** $[\![u_i]\!] \quad \forall 0 \le i < n$
11: $[\![a_z]\!] \leftarrow$ **if** $[\![u_i]\!] \stackrel{?}{=} [\![u_j]\!]$ **then** 0 **else** $[\![a_z]\!]$; $[\![s_z]\!] \leftarrow$ **if** $[\![g_z]\!] \wedge \neg[\![g_{z-1}]\!]$ **then** $[\![r]\!] - [\![A_{z-1}]\!]$ **else** $[\![s_z]\!]$ } $\forall z = d(i,j,p) \in [N]$
12: **end for**
13: $F \leftarrow \emptyset$
14: **for** $0 \le i < j < n$ and $p = 1, 2$ **do**
15: Send $\text{Reveal}_p([\![s_{d(i,j,p)}]\!])$ to party p
16: **if** party p determines that $s_{d(i,j,p)} \neq \bot$ **then**
17: **Party** p: $e \leftarrow s_{d(i,j,p)}$-th element in $E_{ij}^{(p)}$
18: **Party** p sends e to the other party
19: $F \leftarrow F \cup \{e\}$
20: **end if**
21: **end for**
22: **return** F
23: **end protocol**

In each of the $n-1$ iterations, this protocol chooses a uniformly random edge and updates all variables in the following way:

- First, we compute the prefix-sums $A_0, \ldots, A_{N-1}$ of the array $a_0, \ldots, a_{N-1}$. Note that A_{N-1} is the number of remaining edges that could be selected without introducing a cycle.
- Then, we generate a uniformly random integer r between 0 and A_{N-1} (exclusive). Note that r corresponds to the integer z for which $A_z > r \ge A_{z-1}$. This $z = d(i,j,p)$ indicates that an edge between vertices i and j, belonging to party p has been selected.
- In order to update the union-find data structure that we obtain by merging i and j, we need the secret-shares $[\![u_i]\!]$ and $[\![u_j]\!]$, where i and j correspond to the lowest $z = d(i,j,p)$ for which $g_z := A_z \stackrel{?}{>} r$ is true. We obtain these secret-shares through $\text{FindFirst}([([\![u_i]\!], [\![u_j]\!]), [\![g_z]\!]_{z\in[N]})$.
- Finally, we update the union-find data structure, set values of a_z to 0 where necessary, and update s_z (because we picked the $(r - A_{z-1})$-th edge of $E_{ij}^{(p)}$ into our MSF).

Then, the protocol reveals all values of $s_{d(i,j,p)}$ to party p, who sends the $s_{d(i,j,p)}$-th edge within $E_{ij}^{(p)}$ to the other party. All edges communicated in this way will together form the MSF.

Note that IsolatedMSF makes use of the following three subprotocols (where B is the bitlength of the secret-shares of $a_{d(i,j,p)}$ and $s_{d(i,j,p)}$, i.e., 2^B is an upper bound on the number of edges in the input):

- $\text{PrefixSum}([\![a_0]\!], \ldots, [\![a_{N-1}]\!])$ is trivially implemented using $N-1$ secure additions. Using the Ladner-Fischer adder [23], this requires in total $O(N \cdot B \log B)$ bit multiplications and $O(B \log B)$ communication rounds.
- $\text{RandInt}([\![M]\!])$ computes a uniformly random number in the range $[0, M)$, where M itself is secret-shared. We realize this using [9, Algorithm 7]. Note that this protocol has an error probability of $\le \frac{1}{2^\kappa}$ if a pool of κ generated numbers is used. The number of multiplications is $O(B \cdot \kappa)$, and the number of communication rounds is $O(\log B + \log \kappa)$.
- $\text{FindFirst}(L)$ takes a list L of secret-shares of the form $[\![L_i]\!] = ([\![x_i]\!], [\![g_i]\!])$, where g_i is a single bit. It returns the first element $[\![L_i]\!]$ for which $g_i = 1$. This procedure can be realized through $O(\log |L|)$ iterations, each of them merging any two neighboring elements together. The total number of multiplications is $O(|L| \cdot \log n)$ (with $\log n$ being the bitlength of x_i), and the number of rounds is $O(\log |L|)$.

In total, IsolatedMSF requires $O(n \cdot B \cdot \kappa + n^3 \cdot (\log n + B \log B))$ multiplications, and $O(n \cdot (B \log B + \log n + \log \kappa))$ rounds.

5 Evaluation

Our protocol's efficiency depends on the graph structure and its weights. In the best case, it will have $O(|V|)$ communication and $O(\log |V|)$ rounds, but in the worst case (when all edges have the same weight) it may require $O(|V|^3 \log |V|)$ communication and $O(|V| \log |V|)$ rounds due to running IsolatedMSF on a size-$|V|$ subgraph. More generally, in a graph with m isolatable subgraphs of size n (with distinct weights), the overall communication will be $O(m \cdot n^3 \log n)$, with $O(\log |V| \cdot n \log n)$ rounds. Due to the large variance, we will give an empirical evaluation in the following.

We have implemented our RandomMSF protocol in C++, based on the semi-honest two-party computation framework ABY [12]. For every "stage" of our protocol that reveals some intermediate result (i.e., weight comparisons in line 8, Connectivity in line 14, and IsolatedMSF in line 24), we create a new circuit, evaluate it using ABY, and then proceed with the remaining protocol whose execution path depends on the output values of the previous circuit evaluation. To save memory and improve the running time, our circuits utilize SIMD gates whenever the same instructions are executed for several inputs in parallel (e.g., any two calls to Connectivity on the same number of vertices would be run simultaneously in SIMD copies of the same gates).

Hardware setup. We evaluate our implementation on pairs of AWS instances with AMD EPYC processors with 8 cores and 16 GiB memory. We consider two settings:

- *LAN*: Two servers in the same availability zone. The round-trip time is 0.27 ms, and the bandwidth is about 5 Gbps.

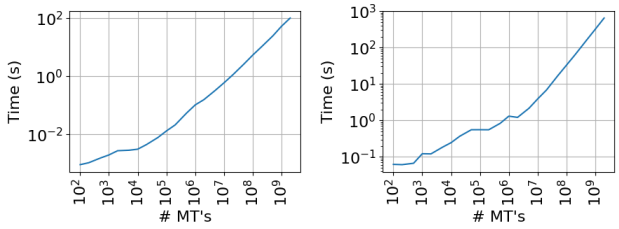

Figure 3: Time required for generating multiplication triples in the offline phase. Left: *LAN* setup, right: *WAN* setup.

- *WAN*: One server on the US west coast, and one server on the US east coast. The round-trip time is 60 ms, and the bandwidth is about 390 Mbps.

Parameters. In all cases, we choose the statistical security parameter $\kappa = 40$ for calls to RandInt. We select the bitlength $B = 32$ for any secret-shared values that represent numbers of edges (e.g. $[\![a_z]\!]$'s in IsolatedMSF, Protocol 4). This means that no party is allowed to input more than 2^{32} edges of any given weight.

Separating offline and online phase. In an input-independent offline phase we can preprocess multiplication triples (MT's; also called *beaver triples* [5]), which saves time during the online phase that is run once both parties know their inputs. An MT can be *consumed* when multiplying two secret-shared bits with each other. An MT-based multiplication requires each party to send one bit to the other party (this is essentially the only source of communicated data in our MSF protocol). Until it is consumed, an MT requires 3 bits of storage by each party.

ABY uses OT-extension to generate multiplication triples [2, 29]. Figure 3 shows how much time this requires on our hardware setup. We can see that our *LAN* setup can generate almost $20 \cdot 10^6$ MT's per second, and our *WAN* setup can generate about $3.1 \cdot 10^6$ MT's per second. Each MT requires about 16 byte of communication in both directions.

Baseline. We use Laud's protocol [24] as our baseline, because it is the most optimized existing protocol for computing MSFs that can be adapted towards our notion of a *Random* MSF.

We directly take the numbers from Laud's paper [24], but also note that it is expected that Laud's protocol will slow down further when computing a *Random* MSF (due to additional secret-shared values). Furthermore, it is implemented in the information-theoretic 3-party framework Sharemind [8], and therefore likely to be slower in a two-party setting. Its evaluation is based on hardware with 12 cores and 48 GiB of memory. Laud only considers the *LAN* setting with a bandwidth of only 1 Gbps.

5.1 Random Graphs

We generated graphs with $|E| = 3|V|$ edges that have uniformly random (but distinct) endpoints. Those edges are split among the two parties, s.t. each party's input contains $\frac{|E|}{2}$ edges.

Because our protocol's running time depends on the edge weights and graph structure (instead of merely the input size), we need another parameter to determine how "difficult" a graph is for our protocol to process. We do so by choosing all edge weights uniformly at random between 0 and $w \cdot |E|$, for some fixed $w \in \mathbb{R}$. For example, when $w = 0.05$, then every possible weight will be used by 20 different edges on average. When decreasing w, we can expect our protocol to become slower, because more edges will share the same weight.

Figure 4 shows the required time and amount of communication by our protocol (which is proportional to the number of bit multiplications), for different values of the parameter w. For example, for a graph with $2 \cdot 10^5$ vertices and parameter $w = 0.05$, our protocol requires 107 seconds in the *LAN* setup, and about 804 seconds (13.4 minutes) in the *WAN* setup. The amount of communicated data has a size of roughly 925 MiB (due to roughly $3.7 \cdot 10^9$ secret bit multiplications).

Comparison with baseline. We compare the results for our lowest choice of w ($w = 0.05$) with Laud's protocol in Figure 5. For a fair comparison, because Laud does not require any offline phase, in the same figure we also state the combined time required for online and offline phase. This is estimated using the numbers from Figure 3.

We can see that our protocol outperforms Laud's protocol by a factor of almost 100×. When considering the online phase only, the difference is even higher (about 200×). Furthermore, for fixed w, while the time for Laud's protocol increases with the number of edges, the performance of our protocol is essentially independent of $|E|$ (see the right-hand side of Figure 5).

We note that higher choices of w (as depicted in Figure 4, e.g. $w = 0.2$ or even completely unique weights) would lead to even larger discrepancies between our protocol and that by Laud. Conversely, when choosing w too small (the most extreme case would be to have a graph with edges of identical weights only), it is expected that Laud's protocol becomes the better choice at some point.

5.2 TSPLIB Graphs

To further demonstrate that our protocol is useful in practice when used as a 2-approximation of the Traveling Salesman Problem, we consider several randomly selected graphs from TSPLIB collection [32]. All of these graphs are complete, meaning that there exists an edge between each pair of vertices. All of them contain many edges with identical weights.

Figure 6 contains the results for graphs with up to 1500 vertices. We do note that for several inputs from TSPLIB with less than 1500 vertices (not depicted in Figure 6), our protocol's required resources blow up so quickly (due to too many edges of the same weight) that we were unable to run the entire protocol. However, on other inputs, our protocol significantly outperforms Laud's protocol, because the baseline requires the entire input to be secret-shared.

5.3 In-depth Runtime Analysis

We now provide some more insight into what is the most time-consuming part of our protocol. Note that our protocol RandomMSF can be split into two phases:

- Phase 1 (lines 4 to 21) iteratively computes the best incident weight for each vertex (which is very cheap) and runs Connectivity on vertices with the same weight.
- Phase 2 (lines 22 to 25) simultaneously computes IsolatedMSF for all isolatable subgraphs found before.

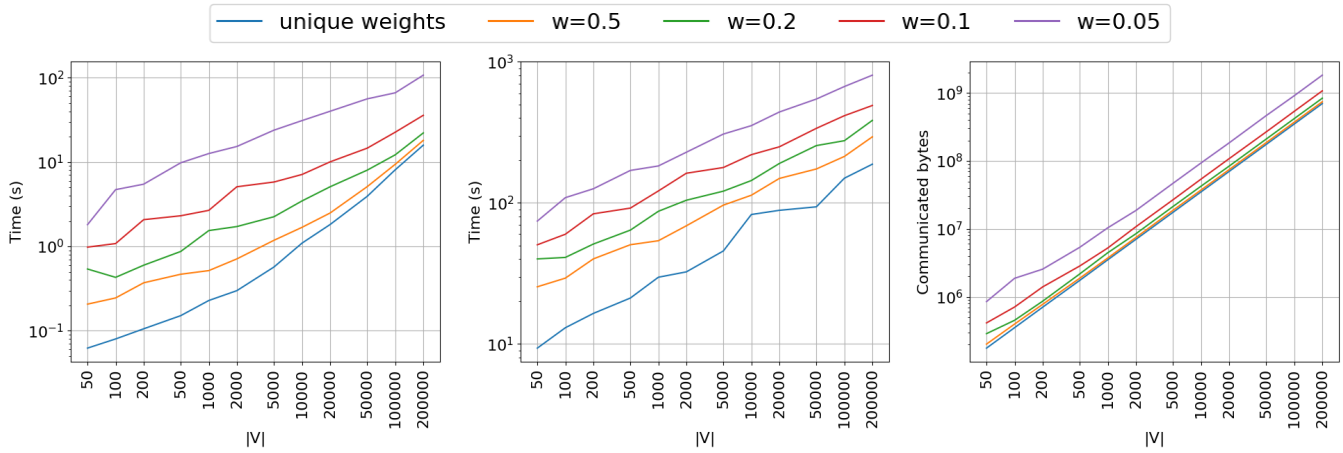

Figure 4: The amount of time in the online phase, and the number of bit multiplications required by our protocol on random input graphs with $|E| = 3|V|$. Left: *LAN* setting, middle: *WAN* setting, right: amount of communication (note that every byte of communication means that 4 MTs have been consumed). Edge weights are either chosen uniquely (blue line), or uniformly at random in the range $[0, w \cdot |E|)$. All numbers are averaged over 3 runs with independently generated graphs.

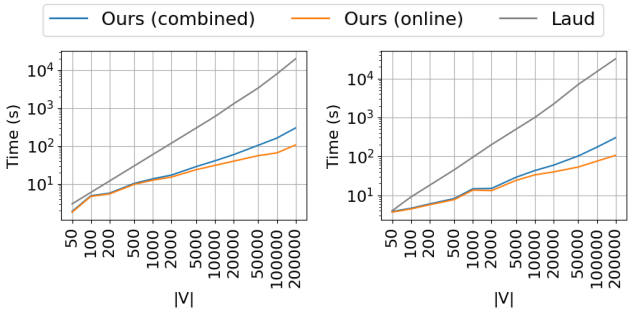

Figure 5: Comparison of our protocol with that by Laud in the *LAN* setting, for a random graph with weight parameter $w = 0.05$. Left: $|E| = 3|V|$, right: $|E| = 6|V|$. For our protocol, we display both the online running time, and the combined running time that includes the time for generating MT's.

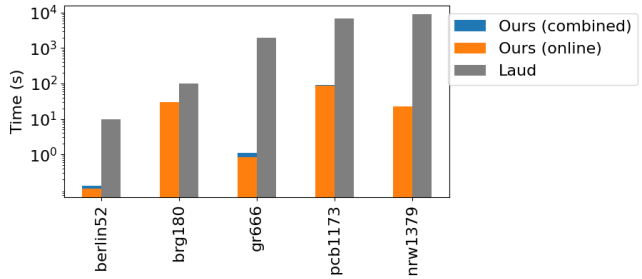

Figure 6: The time required by our protocol (compared with Laud's protocol) for several TSPLIB graphs in the *LAN* setting. We display both the online running time, and the combined running time that also includes the offline time for generating MT's.

Both Connectivity and IsolatedMSF require communication that grows cubic in the number of vertices of the provided subgraph. These two subprotocols are responsible for the vast majority of the running time.

Figure 7 contains the results for a random graph with of size $|V| = 2 \cdot 10^5$ and varying weight parameter w. It shows that, despite phase 1 having many iterations (typically between 100 and 200 for $|V| = 2 \cdot 10^5$ vertices), RandomMSF mostly outweighs the rest of the protocol due to its high costs. However, this discrepancy becomes smaller when considering the *WAN* setting where the number of communication rounds plays a bigger role.

For the same graphs, the right-hand side of Figure 7 shows the distribution of sizes of isolatable subgraphs on which IsolatedMSF needs to be run. Naturally, lower w means that isolatable subgraphs will be larger (up to almost 25 when $w = 0.05$), hence resulting in more expensive calls to RandomMSF in phase 2.

5.4 Individual Performance of Connectivity and IsolatedMSF

We also benchmark the two subprotocols by themselves, in order to test which subgraph sizes still feasible for our protocol to process.

Figures 8 and 9 contain the results for Connectivity and IsolatedMSF, respectively. We can see that testing connectivity for up to 150 vertices, and computing a Random MSF of an isolatable subgraph with up to 60 vertices is doable within reasonable time. Figure 10 depicts the number of multiplications for a single call to either of the subprotocols.

Running k SIMD instances simultaneously (as shown in the figures for $k = 100$ and $k = 1000$), significantly decreases the per-instance time. For example, in the *LAN* setting, 1000 instances of Connectivity for 150 vertices is merely 2.5 as expensive as a single instance. The main reason is that most of the time is spent on constructing the binary circuit that is then passed to ABY and reused for all k instances.

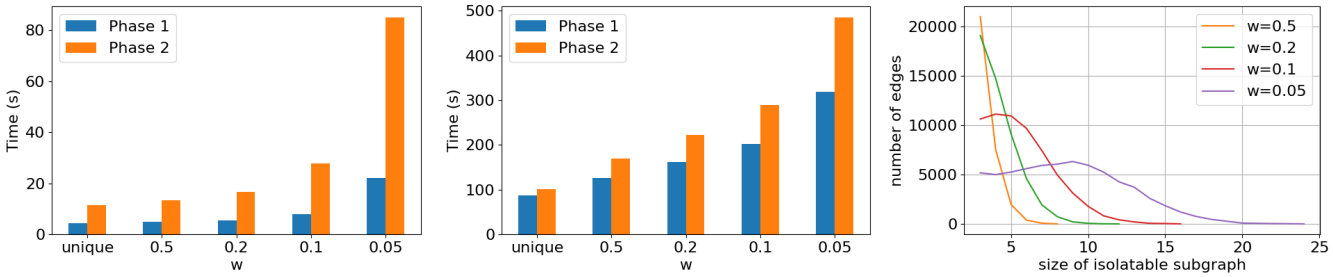

Figure 7: The two figures on the left (*LAN* setting and *WAN* setting, respectively) show the time required for a random graph with $|V| = 2 \cdot 10^5$, $|E| = 3|V|$ and weight parameter $w = 0.05$, split into the two phases. The figure on the right-hand side displays the number of edges discovered as part of an isolatable subgraph of a particular input size (≥ 3). All remaining edges of the MSF (e.g., about 133 000 for $w = 0.05$) not depicted are found inside an isolatable subgraph of exactly 2 vertices.

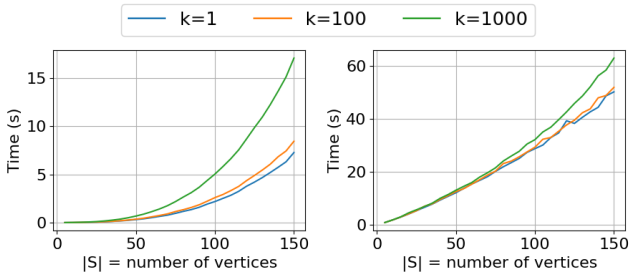

Figure 8: Time required by k simultaneous runs of Connectivity in dependency of the input size (i.e., the number of vertices $|S|$). Left: *LAN* setting, right: *WAN* setting.

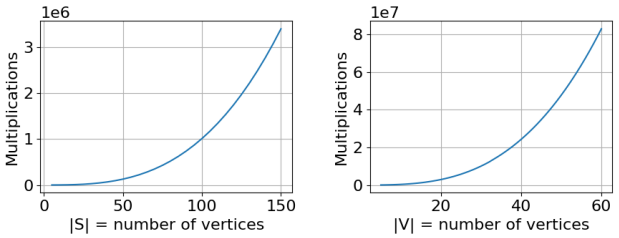

Figure 10: Number of multiplications for a single run of Connectivity (left) or IsolatedMSF (right) in dependency of the input size (i.e., the number of vertices $|S|$ or $|V|$). Recall that each multiplication consumes one MT, and requires communicating 2 bits (1 bit in each direction).

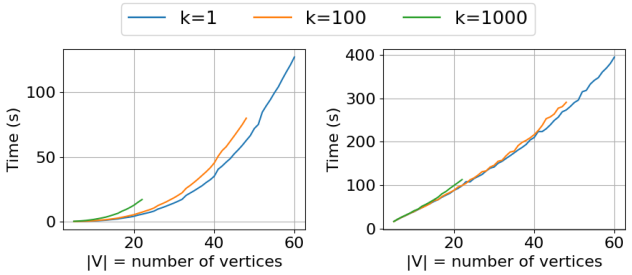

Figure 9: Time required by k simultaneous runs of IsolatedMSF in dependency of the input size (i.e., the number of vertices $|V|$). Left: *LAN* setting, right: *WAN* setting. Technical restrictions do not allow us to test configuration in which the number of bit multiplications becomes too large.

These numbers may be used as a guideline to estimate what graphs (not randomly generated as in Section 5.1) are still within scope of our protocol. For example, a graph with up to 100 isolatable subgraphs of size 50 each may still be doable in under two minutes in the *LAN* setting.

6 Conclusion

In this work, we have presented a novel protocol for computing a Minimum Spanning Forest in a semi-honest, two-party computation setting. Importantly, we are able to handle non-unique edge weights in a fairness-preserving way.

The performance highly depends on the graph structure and its weights, but our evaluation shows that we gain large improvements over the baseline for many settings. Our protocol also has a low round-complexity, making it useful even for a WAN setting, and it is independent of the number of edges.

There is still much potential for improvements, e.g. in the subprotocols Connectivity and IsolatedMSF whose running time quickly grow if there are many edges with the same weights. It may be useful to fall back to alternative solutions (such as ORAM) to counter cubic growth in these cases.

Independently of the MSF problem, our work also sheds light on the importance of analyzing a problem, instead of simply applying general-purpose MPC compilers. Our work constructs a protocol that yields significant improvements for "typical" inputs. These techniques, i.e., revealing information (in a secure way) that is expected to speed up the protocol on any *realistic* inputs, and therefore breaks the "worst-case barrier" of general MPC frameworks, might also be useful for applications other than MSFs.

Acknowledgments

We gratefully acknowledge the support of NSERC for grants RGPIN-2023-03244, IRC-537591, the Government of Ontario and the Royal Bank of Canada for funding this research.

References

[1] Mohammad Anagreh, Eero Vainikko, and Peeter Laud. 2021. Parallel Privacy-preserving Computation of Minimum Spanning Trees. In *Proceedings of the 7th International Conference on Information Systems Security and Privacy, ICISSP 2021, Online Streaming, February 11-13, 2021*, Paolo Mori, Gabriele Lenzini, and Steven Furnell (Eds.). SCITEPRESS, 181–190. https://doi.org/10.5220/0010255701810190

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More efficient oblivious transfer and extensions for faster secure computation. In *ACM CCS 2013: 20th Conference on Computer and Communications Security*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 535–548. https://doi.org/10.1145/2508859.2516738

[3] Baruch Awerbuch and Yossi Shiloach. 1987. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Trans. Comput.* 36, 10 (1987), 1258–1263.

[4] Marc Barthélemy. 2011. Spatial networks. *Physics reports* 499, 1-3 (2011), 1–101.

[5] Donald Beaver. 1992. Efficient Multiparty Protocols Using Circuit Randomization. In *Advances in Cryptology – CRYPTO'91 (Lecture Notes in Computer Science, Vol. 576)*, Joan Feigenbaum (Ed.). Springer, Heidelberg, Germany, Santa Barbara, CA, USA, 420–432. https://doi.org/10.1007/3-540-46766-1_34

[6] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, Janos Simon (Ed.). ACM, 1–10. https://doi.org/10.1145/62212.62213

[7] Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. 2013. Data-oblivious graph algorithms for secure computation and outsourcing. In *ASIACCS 13: 8th ACM Symposium on Information, Computer and Communications Security*, Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng (Eds.). ACM Press, Hangzhou, China, 207–218.

[8] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS 2008: 13th European Symposium on Research in Computer Security (Lecture Notes in Computer Science, Vol. 5283)*, Sushil Jajodia and Javier López (Eds.). Springer, Heidelberg, Germany, Málaga, Spain, 192–206. https://doi.org/10.1007/978-3-540-88313-5_13

[9] Jonas Böhler and Florian Kerschbaum. 2020. Secure Sublinear Time Differentially Private Median Computation. In *ISOC Network and Distributed System Security Symposium – NDSS 2020*. The Internet Society, San Diego, CA, USA.

[10] Otakar Borůvka. 1926. O jistém problému minimálním. In *Práce Moravské přírodovědecké společnosti*, Vol. 3. 37–58.

[11] Justin Brickell and Vitaly Shmatikov. 2005. Privacy-Preserving Graph Algorithms in the Semi-honest Model. In *Advances in Cryptology – ASIACRYPT 2005 (Lecture Notes in Computer Science, Vol. 3788)*, Bimal K. Roy (Ed.). Springer, Heidelberg, Germany, Chennai, India, 236–252. https://doi.org/10.1007/11593447_13

[12] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *ISOC Network and Distributed System Security Symposium – NDSS 2015*. The Internet Society, San Diego, CA, USA.

[13] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Found. Trends Priv. Secur.* 2, 2-3 (2018), 70–246. https://doi.org/10.1561/3300000019

[14] Alan M. Frieze. 1985. On the value of a random minimum spanning tree problem. *Discret. Appl. Math.* 10, 1 (1985), 47–56. https://doi.org/10.1016/0166-218X(85)90058-7

[15] Oded Goldreich. 1998. Secure multi-party computation. *Manuscript. Preliminary version* 78 (1998), 110.

[16] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, Alfred V. Aho (Ed.). ACM, 218–229. https://doi.org/10.1145/28395.28420

[17] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473. https://doi.org/10.1145/233551.233553

[18] Ronald L Graham and Pavol Hell. 1985. On the history of the minimum spanning tree problem. *Annals of the History of Computing* 7, 1 (1985), 43–57.

[19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. 2019. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Francisco, CA, USA, 1220–1237. https://doi.org/10.1109/SP.2019.00028

[20] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, Virtual Event, USA, 1575–1590. https://doi.org/10.1145/3372297.3417872

[21] Florian Kerschbaum and Andreas Schaad. 2008. Privacy-preserving social network analysis for criminal investigations. In *Proceedings of the 7th ACM workshop on Privacy in the electronic society*. 9–14.

[22] Joseph B Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society* 7, 1 (1956), 48–50.

[23] Richard E. Ladner and Michael J. Fischer. 1980. Parallel Prefix Computation. *J. ACM* 27, 4 (1980), 831–838. https://doi.org/10.1145/322217.322232

[24] Peeter Laud. 2015. Parallel Oblivious Array Access for Secure Multiparty Computation and Privacy-Preserving Minimum Spanning Trees. *Proceedings on Privacy Enhancing Technologies* 2015, 2 (April 2015), 188–205. https://doi.org/10.1515/popets-2015-0011

[25] Yehuda Lindell. 2017. How to Simulate It - A Tutorial on the Simulation Proof Technique. In *Tutorials on the Foundations of Cryptography*, Yehuda Lindell (Ed.). Springer International Publishing, 277–346. https://doi.org/10.1007/978-3-319-57048-8_6

[26] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 359–376. https://doi.org/10.1109/SP.2015.29

[27] Rajesh Matai, Surya Prakash Singh, and Murari Lal Mittal. 2010. Traveling salesman problem: an overview of applications, formulations, and solution approaches. *Traveling salesman problem, theory and applications* 1 (2010).

[28] Charalampos Mavroforakis, Richard Garcia-Lebron, Ioannis Koutis, and Evimaria Terzi. 2015. Spanning edge centrality: Large-scale computation and applications. In *Proceedings of the 24th international conference on world wide web*. 732–742.

[29] Moni Naor and Benny Pinkas. 2001. Efficient Oblivious Transfer Protocols. In *12th Annual ACM-SIAM Symposium on Discrete Algorithms*, S. Rao Kosaraju (Ed.). ACM-SIAM, Washington, DC, USA, 448–457.

[30] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, San Jose, CA, USA, 377–394. https://doi.org/10.1109/SP.2015.30

[31] Robert Clay Prim. 1957. Shortest connection networks and some generalizations. *The Bell System Technical Journal* 36, 6 (1957), 1389–1401.

[32] Gerhard Reinelt. 1991. TSPLIB - A Traveling Salesman Problem Library. *INFORMS J. Comput.* 3, 4 (1991), 376–384. https://doi.org/10.1287/ijoc.3.4.376

[33] Thomas Schneider and Michael Zohner. 2013. GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits. In *FC 2013: 17th International Conference on Financial Cryptography and Data Security (Lecture Notes in Computer Science, Vol. 7859)*, Ahmad-Reza Sadeghi (Ed.). Springer, Heidelberg, Germany, Okinawa, Japan, 275–292. https://doi.org/10.1007/978-3-642-39884-1_23

[34] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM CCS 2013: 20th Conference on Computer and Communications Security*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, Berlin, Germany, 299–310. https://doi.org/10.1145/2508859.2516660

[35] Paolo Toth and Daniele Vigo. 2002. *The vehicle routing problem*. SIAM.

[36] David Bruce Wilson. 1996. Generating Random Spanning Trees More Quickly than the Cover Time. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, Gary L. Miller (Ed.). ACM, 296–303. https://doi.org/10.1145/237814.237880

[37] Andrew Chi-Chih Yao. 1982. Protocols for Secure Computations (Extended Abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*. IEEE Computer Society, 160–164. https://doi.org/10.1109/SFCS.1982.38

A Security issues in the naive solution

Looking closely at our Random MSF protocol in Section 3, it never *explicitly* generates the tie-breaker π that is used to sample the MSF $\mathrm{MSF}(G, \pi)$. Instead, we perform a decomposition into isolatable subgraphs.

It turns out that it is actually *necessary* to never explicitly generate the tie-breaker (unless it is kept secret-shared for the entire duration of the protocol execution). The following theorem shows that any party who has knowledge of the tie-breaker values (even if it is just the tie-breaker values of their own edges) can use it to gain information about the other party's input, thereby breaking security.

Therefore, Theorem A.1 will serve as a justification for our protocol, and why it needs to have some expensive steps such as the subprotocols Connectivity and IsolatedMSF which require cubic communication in the size of the subgraphs they are working on.

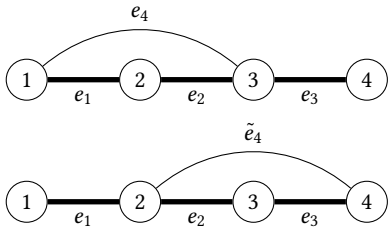

Figure 11: Two different graphs that can be distinguished when the tie-breaker π is leaked.

Theorem A.1. *Suppose Π is a protocol implementing Functionality 1, i.e., it outputs MSF(G, π) for a uniformly random $\pi : E \to [|E|]$. If one of the parties p is able to determine (given its view of the protocol execution) the relative ordering of their own edges $E^{(p)}$ w.r.t. π (i.e., for all edges $e, e' \in E^{(p)}$ they know whether $\pi(e) < \pi(e')$ or $\pi(e) > \pi(e')$), then this protocol is not secure.*

Proof. W.l.o.g. assume that party 1 is able to infer the relative ordering of their own edges $E^{(1)}$ w.r.t. π. We show that there is no simulator for party 1 satisfying the necessary conditions for semi-honest security (Definition 2.1).

We construct two graphs that party 1 will be able to distinguish with non-negligible probability, even when the same MSF is returned by protocol Π for both input graphs. They are shown in Figure 11. In both graphs, we have four vertices $V = \{1, 2, 3, 4\}$, and the input of party $E^{(1)}$ is always defined as $E^{(1)} = \{e_1, e_2, e_3\}$, where $\mathbf{r}(e_1) = \{1, 2\}$, $\mathbf{r}(e_2) = \{2, 3\}$, and $\mathbf{r}(e_3) = \{3, 4\}$. The difference is that, in the first graph, party 2 has the single edge $E^{(2)} = \{e_4\}$ with $\mathbf{r}(e_4) = \{1, 3\}$, while it has the single edge $\tilde{E}^{(2)} = \{\tilde{e}_4\}$ with $\mathbf{r}(\tilde{e}_4) = \{2, 4\}$ in the second graph. All weights $\mathbf{w}(e) = 0$ (for $e = e_1, e_2, e_3, e_4, \tilde{e}_4$) are equal. This is why the generated MSF crucially depends on the chosen permutation π.

For any tie-breaker $\pi : E^{(1)} \cup E^{(2)} \to [4]$ of the first graph, let $F(\pi) := \mathsf{MSF}(V, E^{(1)} \cup E^{(2)}, \pi)$ be the MSF generated for π. For any tie-breaker $\pi : E^{(1)} \cup \tilde{E}^{(2)} \to [4]$ of the second graph, let $\tilde{F}(\pi) := \mathsf{MSF}(V, E^{(1)} \cup \tilde{E}^{(2)}, \pi)$ be the MSF generated in the second graph on tie-breaker π.

By semi-honest security (Equation (1)), there must be a simulator S simulating the view of party 1. This theorem's assumption additionally states that the tie-breaker restricted to the edges of party 1 (let us denote this by the unique $\pi' : E^{(1)} \to [3]$ for which $\pi'(e) < \pi'(e')$ iff $\pi(e) < \pi(e')$) can be extracted from the view, and therefore we may assume that S itself outputs this tie-breaker. Thus, the following distributions are indistinguishable:

$$\{(S(E^{(1)}, F(\pi)), F(\pi))\} \overset{c}{\equiv} \{(\pi', F(\pi))\} \tag{2}$$

$$\{(S(E^{(1)}, \tilde{F}(\pi)), \tilde{F}(\pi))\} \overset{c}{\equiv} \{(\pi', \tilde{F}(\pi))\} \tag{3}$$

We now analyze the case in which the protocol outputs $E^{(1)}$ as its final Random MSF (i.e., $F(\pi) = E^{(1)}$ or $\tilde{F}(\pi) = E^{(1)}$, respectively). Note that in both graphs, the probability of this happening is exactly $\frac{1}{3}$ (in the first graph, $F(\pi) = E^{(1)}$ happens iff $\pi(e_1), \pi(e_2) < \pi(e_4)$, and in the second graph, $\tilde{F}(\pi) = E^{(1)}$ happens iff $\pi(e_2), \pi(e_3) < \pi(\tilde{e}_4)$).

Thus, by combining (2) and (3), we know that, conditioning on the output being equal to $E^{(1)}$, it will be impossible to distinguish between the distribution of π' on graph 1 and graph 2:

$$\{\pi' \mid F(\pi) = E^{(1)}\} \overset{c}{\equiv} \{\pi' \mid \tilde{F}(\pi) = E^{(1)}\}$$

However, this does not hold. By checking the condition $\pi(e_1) < \pi(e_2) < \pi(e_3)$ (testing this merely requires knowledge π'), it is easy to distinguish between these two distributions:

- In the first graph, this conditions holds for exactly two tie-breakers π: when $\pi(e_1) < \pi(e_2) < \pi(e_3) < \pi(e_4)$ and when $\pi(e_1) < \pi(e_2) < \pi(e_4) < \pi(e_3)$.
- In the second graph, this condition holds for only one tie-breaker π: $\pi(e_1) < \pi(e_2) < \pi(e_3) < \pi(\tilde{e}_4)$.

□

B Correctness

Proof of Lemma 3.2. We are given a graph $(V, E, \mathbf{r}, \mathbf{w})$, and an isolatable subgraph $c \subseteq V$ of weight w. Let $\tilde{E} := \{e \in E_{=w} \mid \mathbf{r}(e) \subseteq c\}$ be the set of all edges that connect two vertices in c and have weight w, and let $E' := E \setminus \{e \in E \mid \mathbf{r}(e) \subseteq c\}$ be the set of all edges that do *not* connect two vertices in c.

We need to prove that

$$\mathsf{MSF}((V, E, \mathbf{r}, \mathbf{w}), \pi) \quad \text{with } \pi \leftarrow E!$$

has the same distribution as

$$\mathsf{MSF}((c, \tilde{E}, \mathbf{r}|_{\tilde{E}}, \mathbf{w}|_{\tilde{E}}), \tilde{\pi}) \cup \mathsf{MSF}(\mathrm{merge}_c(G), \pi') \quad \text{with } \tilde{\pi} \leftarrow \tilde{E}! \text{ and } \pi' \leftarrow E'!\,,$$

where π, $\tilde{\pi}$, and π' are uniformly sampled.

Fix any tie-breaker $\pi \in E!$ on edges E. We define $\tilde{\pi} \in \tilde{E}!$ as the corresponding permutation restricted to $\tilde{E}$ (i.e., s.t. $\tilde{\pi}(e) < \tilde{\pi}(e')$ holds iff $\pi(e) < \pi(e')$ for all $e, e' \in \tilde{E}$), and we define $\pi' \in E'!$ as the corresponding permutation restricted to E'. Then, it suffices to show that the two spanning trees

$$\mathsf{MSF}((V, E, \mathbf{r}, \mathbf{w}), \pi)$$

$$= \mathsf{MSF}((c, \tilde{E}, \mathbf{r}|_{\tilde{E}}, \mathbf{w}|_{\tilde{E}}), \tilde{\pi}) \cup \mathsf{MSF}(\mathrm{merge}_c(G), \pi')$$

are equal, because any pair $(\tilde{\pi}, \pi') \in \tilde{E}! \times E'!$ is generated by exactly the same amount of π's.

Since both sides of the equation will have the same size, we only need to prove that every edge in $\tilde{F} := \mathsf{MSF}((c, \tilde{E}, \mathbf{r}|_{\tilde{E}}, \mathbf{w}|_{\tilde{E}}), \tilde{\pi})$ is also in $F := \mathsf{MSF}((V, E, \mathbf{r}, \mathbf{w}), \pi)$, and every edge that is in the set $F' := \mathsf{MSF}(\mathrm{merge}_c(G), \pi')$ is also in F.

- Let $e \in \tilde{F}$ be some edge in the MSF of the isolatable subgraph. For the sake of contradiction, assume $e \notin F$. Adding e to F would result in a cycle (otherwise F was not a forest), and furthermore all edges $e' \in F$ on the cycle need to be "better" than e, i.e., either $\mathbf{w}(e') < \mathbf{w}(e)$, or $\mathbf{w}(e') = \mathbf{w}(e)$ and $\pi(e') < \pi(e)$ (otherwise F was not minimum).
  As a result, no edge on the cycle may leave the isolatable subgraph c (because all edges leaving c have weight $> w = \mathbf{w}(e)$). Hence, every such e' is in $\tilde{E}$. Furthermore, at least one of the edges of the cycle must cross the cut of c induced by e, and therefore replacing e by e' in $\tilde{F}$ would result in another

spanning forest $\tilde{F} \setminus \{e\} \cup \{e'\} \subseteq \tilde{E}$ on c, which is better than $\tilde{F}$ itself. This contradicts the fact that $\tilde{F}$ is minimum.

- Let $e \in F'$ be some edge in the MSF of the graph $\text{merge}_c(G)$. For the sake of contradiction, assume $e \notin F$. As before, adding e to F would result in a cycle, and all edges e' on the cycle are better than e.
  Consider the cut (S, T) (with $S, T \subseteq V'$) on $\text{merge}_c(G)$ induced by e. W.l.o.g. assume that the merged vertex c is in S. Then, $(S \setminus \{c\} \cup c, T)$ forms a cut on G crossed by e, and the cycle must contain another edge e' that crosses this cut. This means that at most one endpoint of e' is in c, and therefore $e' \in E'$. As a result, we could replace e by e' in F', and receive another spanning forest $F \setminus \{e\} \cup \{e'\} \subseteq E'$ on $\text{merge}_c(G)$, which is better than F itself. This contradicts the fact that F' is minimum.

□

C Security

Security of Protocol 3 (RandomMSF) is based on the following two lemmas. Intuitively, they show that all intermediate values revealed in our protocol (i.e., values w_v in line 8 and the output of Connectivity in line 14) may be simulated given an arbitrary MSF, i.e., given any output of RandomMSF.

Lemma C.1. *Let $G = (V, E, \mathbf{r}, \mathbf{w})$ be a graph, and let $F \subseteq E$ be an MSF of G. Then, for any vertex $v \in G$, the weight of its best outgoing edge is the same for both E and F:*

$$\min\{\mathbf{w}(e) \mid e \in \delta_E(v)\} = \min\{\mathbf{w}(e) \mid e \in \delta_F(v)\}$$

Proof. Because of $F \subseteq E$, it is obvious that the right-hand side cannot be strictly smaller than the left-hand side. So for the purpose of a contradiction, assume that

$$\min\{\mathbf{w}(e) \mid e \in \delta_E(v)\} < \min\{\mathbf{w}(e) \mid e \in \delta_F(v)\}.$$

That is, there is an edge $e \in \delta_E(v)$ with $\mathbf{w}(e) < \min\{\mathbf{w}(e) \mid e \in \delta_F(v)\}$. We can define a new set of edges $F' := F \cup \{e\}$ by adding e to the MSF F. The graph $(V, F', \mathbf{r}|_{F'}, \mathbf{w}|_{F'})$ must contain a cycle, because otherwise F was not *spanning*.

Now remove any edge $e' \in \delta_F(c)$ on the cycle to obtain $F'' := F' \setminus \{e'\}$. Then, F'' is still a spanning forest (because removing an edge e' from a cycle does not impact reachability, and because there exists no cycle anymore). Furthermore, due to $\mathbf{w}(e) < \mathbf{w}(e')$, F'' would be a spanning forest with $\mathbf{w}(F'') < \mathbf{w}(F)$, a contradiction to F being minimum. □

Lemma C.2. *Let $G = (V, E, \mathbf{r}, \mathbf{w})$ be a graph, and let $F \subseteq E$ be an MSF of G. Furthermore, let c be an isolatable subgraph of weight w in G. Then, the following equation holds:*

$$\begin{aligned} &\textsc{Connectivity}_{V,c}((E_{=w}, \mathbf{r}|_{E_{=w}})) \\ &= \textsc{Connectivity}_{V,c}((F_{=w}, \mathbf{r}|_{F_{=w}})) \end{aligned}$$

Proof. Any two vertices $u, v \in c$ that are connected through edges in $F_{=w}$ are trivially also connected through edges in $E_{=w}$ (because of $F_{\leq w} \subseteq E_{\leq w}$). Similarly, any vertex $v \in c$ that is connected to some vertex in $V \setminus c$ through edges in $F_{=w}$ is also connected to some vertex in $V \setminus c$ through edges in $E_{=w}$.

Now, for the sake of contradiction, suppose that $u, v \in c$ are reachable from each other using only edges in $E_{=w}$, but not using edges in $F_{=w}$. Then at least one of the edges $e \in E_{=w}$ on the path between u and v can be added to $F_{=w}$ without creating a cycle. Denote its endpoints by $\mathbf{r}(e) = \{u', v'\}$. If we add e to F to obtain $F' := F \cup \{e\}$, then F' must contain a cycle (otherwise F was not a *spanning* forest).

Assume that all edges in the cycle have weight $\leq w$. Then, in fact, none of those edges on the cycle can weight have *strictly* less than w, because that would mean that c is not an isolatable subgraph (since there would be some path $u \rightsquigarrow u' \rightsquigarrow v' \rightsquigarrow v$ using only edges of weight $\leq w$, and at least one of them would have weight strictly less than w). But this contradicts the assumption that u' and v' are not reachable using only edges in $F_{=w}$.

In conclusion, there exists some edge e' on the cycle in F' with weight $\mathbf{w}(e') > w$. Then, $F'' := F' \setminus \{e'\}$ no longer contains a cycle and hence F'' is a spanning forest. However, we have $\mathbf{w}(F'') < \mathbf{w}(F)$, which is a contradiction to F being minimum.

Analogous reasoning shows that whenever a vertex $v \in c$ can reach any vertex in $V \setminus c$ using edges in $E_{=w}$, then it can also reach some vertex in $V \setminus c$ using edges in $F_{=w}$. Therefore, the two outputs of Connectivity are identical. □

Theorem C.3. *Protocol 3 privately computes Functionality 1 in the semi-honest security model.*

Proof. We construct the following simulator S, which takes the protocol output (i.e., an MSF F) as input, and returns the view of any fixed party. This simulator simply follows the execution of Protocol 3 (on edges F), and simulates all communicated values as follows:

- When encountering line 8 that computes best incident weight w_v, simulate w_v as $\min\{\mathbf{w}(e) \mid e \in \delta_F(v)\}$. This is equal to the *actual* value in the real execution, due to Lemma C.1. Therefore, all communication in this step may be simulated using semi-honest security of the underlying protocol that computes w_v.
- When encountering line 14 that runs Connectivity, simulate its output as $\textsc{Connectivity}_{V,S_w}((F_{=w}, \mathbf{r}|_{F_{=w}}))$. This is equal to the *actual* value in the real execution, due to Lemma C.2. Therefore, all communication in this step may be simulated using semi-honest security of the Connectivity protocol.
- When encountering line 24 that runs IsolatedMSF, simulate its output as the set $\tilde{F}$, with $\tilde{F} \subseteq F$ being the set of MSF edges that connect vertices in c. All communication in this step may be simulated using semi-honest security of the IsolatedMSF protocol.

□