Private Two-Party Random Minimum Spanning Forest Computation

by

Marian Dietz

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Finding the Minimum Spanning Tree, or more generally the Minimum Spanning Forest (MSF), of a weighted graph is a well-known algorithmic problem. While this problem itself can be directly applied to any kind of networks, it also has less obvious applications like the approximation of the Traveling Salesman Problem. However, there is only limited work on efficiently computing an MSF in a two-party computation setting, where the input graph is split between two parties, and the goal is to find the MSF on the combined graph without leaking any information about the parties' respective inputs. Any prior work on this problem either follows a generic approach that builds a circuit in order to run it through a general multi-party computation protocol, or requires a high number of communication rounds (usually at least linear in the graph size). In addition, all of these approaches assume that the edge weights are unique.

In this work, we are going to address these issues by first defining a lightweight and simple protocol with a low worst-case number of communication rounds, under the constraint that no two edges share the same weight. We then analyze the problems occurring after enabling the possibility of duplicated weights, and look at the more general problem of generating a Random Minimum Spanning Forest, which defines a distribution of the desired output in case the MSF is not unique. We carefully design a protocol for the semi-honest security model in such a way that as many values as possible can be published in the early stages. This reveals information about the graph structure that is then used to reduce the number of communication rounds. By doing this we get a protocol that performs especially well whenever the number of identical weights is low, while it stays secure and correct regardless of the graph structure and its edge weights.

Acknowledgements

I would like to thank Florian Kerschbaum for his supervision during the master's program. His guidance made this work possible and his outstanding support was always there when I needed it. Thank you to Mohammad Hajiabadi and Douglas Stebila for being part of the thesis committee and reading this work. I would also like to thank the CrySP lab for providing a great atmosphere to learn and study in.

I am grateful to my family for encouraging me to study in Waterloo. Many thanks to my partner Yueheng for going to the event at which we met, for always being there for me and making this time so much better, and for supporting me with all my decisions.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Finding a Minimum Spanning Forest is one of the fundamental and most basic problems in graph theory. Given a weighted and undirected graph, its Minimum Spanning Forest (MSF) is a subset of the graph's edges that fulfill the three requirements:

- *Minimum*: The sum of all weights of the edges in the MSF is as small as possible.
- *Spanning*: Every two vertices that are connected in the original graph are also connected using only the edges in the MSF.
- *Forest*: The MSF edges do not form a cycle.

On a connected graph (where each vertex can be reached from all other vertices), a Minimum Spanning Forest would also be called a Minimum Spanning Tree, because all vertices are reachable from each other using only the MSF edges, which thus form a tree.

The MSF problem has many immediate applications, including any kind of computer or communication networks, but also less obvious ones like speech recognition or clustering, and it can even be used to achieve a very simple 2-approximation of the NP-hard Traveling Salesman Problem [17].

There are three very common algorithms for solving the MSF problem:

- Kruskal's algorithm [23]: Add edges in the order of increasing weight to the MSF, under the condition that the two endpoints are not yet connected to each other using previous MSF edges.

- Prim's algorithm [30]: Start from an arbitrarily chosen vertex, and repeatedly add the smallest edge incident to already discovered vertices. If the given graph is not connected, this procedure needs to be repeated by starting at an undiscovered vertex until all vertices have been found.

- Borůvka's algorithm [8]: Maintain a set of components where for each of them the MSF is already known. In every iteration, select the best incident edge for each component, and add all of those to the MSF. This algorithm only works if all weights are unique, otherwise it might result in cycles.

In a graph where no edge weight occurs more than once, the MSF is always unique. In all other case, depending on the application, it might be sufficient to just compute the total weight of the MSF (which is unique independent of whether weights may occur multiple times), or an arbitrarily selected MSF. This can be achieved with all of the algorithms described above by making the edge weights unique, e.g. by adding a unique index to each edge, and comparing two edges by their indices if their weights are equal. However, sometimes this solution might not be sufficient: For example, edge indices could be heavily biased towards a specific graph structure. In such a case, it might be desired to assign random secondary weights to all edges, which lead 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 this is the main problem that we are going to study in this work.

This is close to the notion of Random Spanning Forests [13], where an unweighted graph is given, and the problem consists of finding an MSF on the graph when a uniformly random weight is assigned to each edge. However, in our scenario, we assume there already exists a weighted graph, and the randomly generated secondary weights are only used to compare two edges that have the same original weight. Note that our notion of a Random Minimum Spanning Forest also differs from generating a Uniform Spanning Forest, which selects any MSF from the set of all MSFs with uniform probability. This problem would be solvable for example with Wilson's algorithm [36].

In two-party computation, or more generally multi-party computation, the goal is for the involved parties to jointly compute a function that depends on private inputs by each of the parties, without learning anything about each other's input other than what is already implied by the final output [14]. There are two different notions of security that are usually considered: Semi-honest security, where all parties are assumed to follow the given protocol, while they might try to infer information from all information they received during the course of the protocol, and malicious security, where protection from misbehaving parties is desired (i.e., it needs to be possible to discover or correct such behavior). While

stronger, protocols secure against malicious adversaries have a significant overhead in time and communication amount [22].

Most generic protocols for multi-party computation require the problem to be written as a binary or arithmetic circuit, which can then be executed in a secure way. 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 [4], [15], and can handle either a binary or arithmetic domain for a varying number of parties [12]. There is a variety of different implementations and frameworks for multi-party computation [19]. Some frameworks switch between domains (e.g. binary or arithmetic) and the corresponding protocol during the execution in order to achieve an improved efficiency [11].

For the MSF problem we study in this work, we want to find two-party protocols in the semi-honest security model. The reason for this is that two-party protocols are usually easier to apply in practice as it allows two parties to compute their combined MSF without having to find an independent third party. Furthermore, the default way of running them in a generic framework requires more resources than a protocol for three or more parties based on an honest majority assumption [22], which makes it clear that protocols designed for one specific purpose are required to achieve a good performance for two parties.

Restricting ourselves to the semi-honest security model allows us to find much more optimized protocols. We are going to depend heavily on the ability to reveal certain parts of the output in the middle of the protocol. This is something that is difficult to make efficient in a maliciously secure protocol, as any adversary could simply adjust their own input whenever they receive new revealed information about the output. In addition, our protocol will be independent of the number of edges each party owns, so that no party ever knows how many edges the other party has. A maliciously secure protocol would require a commitment mechanism so that no party can change their own input after starting the protocol, and therefore any existing method would have to publicly fix the number of edges beforehand.

We also concentrate on a high-latency network, meaning that the two parties who want to compute their combined MSF are physically separated from each other by a large distance. A message sent from one party to the other one can take a long time, and therefore we are interested in primarily minimizing the number of communication rounds, while not significantly increasing the total amount of data sent. Yao's garbled circuits only use a constant number of rounds, but it has been shown that other lightweight protocols can be faster if the circuit depth (and therefore the number of communication rounds) is optimized [32]. Due to its simplicity, we use the GMW protocol [15], which maintains secret

bits by sharing them between the two parties (s.t. the two shares given to the parties add up to the secret bit). Any linear operations (negation, or XOR) on secret-shared bits are free and do not require any communication, while multiplications are typically evaluated using oblivious transfers. However, it is also possible to instead use pre-calculated multiplication triples (so-called Beaver-Triples) to speed up the online phase [3]. Additional optimizations like oblivious transfer extension can be used to efficiently compute a large amount of multiplication triples in an offline phase before starting the actual protocol [21].

The remainder of this thesis is organized as follows: In Chapter 2, we give an overview of previous work on computing MSFs in a secure setting. We state all the necessary definitions and notations in Chapter 3, and we define the problem that we want to solve in Chapter 4. In Chapter 5, we give a protocol based on Borůvka's algorithm for graphs with distinct edge weights, and we discuss the problems arising from edges with equal weights in Chapter 6. Chapter 7 gives a list of useful sub-protocol, which we use in Chapter 8 for a protocol for unweighted graphs, and in Chapter 9 for our full and general protocol. We analyze it and perform comparisons in Chapter 10. All of this is briefly summarized in the conclusion in Chapter 11.

Chapter 2

Related Work

There is only very limited previous work on multi-party protocols for computing Minimum Spanning Forests. However, there are still some publications that could be applied to this problem.

GraphSC is a general framework for graph algorithms in multi-party computation [29]. Its main feature is the ability of *scattering* data from vertices to all incident edges, and *gathering* data from all 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 result in a large number of iterations (each of them consisting of multiple communication rounds) that is at least linear in the number of vertices.

Similarly, Prim's algorithm has been implemented in the more general ObliVM framework [28], which can be applied to any secure computation problem. However, the number of iterations that require communication is also linear in the number of vertices combined with the number of edges. Therefore, this approach is not suitable for our goal of minimizing the number of communication rounds.

Blanton, Steele, and Aliasgari have studied oblivious algorithms targeted towards dense graphs [5]. The authors give a protocol solving the MSF problem by simulating Prim's algorithm on a graph given its adjacency matrix. However, it is infeasible to apply such protocols to large graphs with few edges, as the total communication complexity is at least as large as the square of the number of vertices.

Brickell and Shmatikov have studied a range of graph problems in the semi-honest security model, under the assumption that the output will be revealed at the end of the protocol [9]. While this restriction makes it impossible to use the graph protocol as just one

building block in a larger protocol, it allows for easier and much more efficient computation: In their protocols for all-pairs-shortest-distance and single-source-shortest-distance, Brickell and Shmatikov note that the current shortest distance can always be revealed, as it can be inferred from the final output in any case. This reveal step makes subsequent computation easier. The authors also give MSF protocols based on either Kruskal's or Prim's algorithm. However, both of these protocols would result in a number of iterations that is linear in the number of vertices, and all iterations need to be run sequentially as they require communication that depend on previous results. We address this problem by adapting this type of protocol to Borůvka's algorithm in Chapter 5. In addition, all of these protocols suffer from the fact that they cannot handle edges with the same weight. While Brickell and Shmatikov argue that all edge weights can be made unique in a canonical way, we are interested in designing a protocol that works for assigning a *random* ordering to edges with the same weight.

The most relevant work on MSFs has been done by Laud [25], as they are optimizing the number of communication rounds. They give an implementation of Awerbuch and Shiloach's adaption of Borůvka's algorithm [2] in a semi-honest 3-party computation setting in the Sharemind framework [6]. This protocol is slightly more general than our approach, as it does not need to reveal the MSF at the end of the computation in order to fulfill security. However, as we will see in Chapter 10, our protocol (Chapter 9) can achieve a much lower number of communication rounds in many cases. In addition, it does not depend on any complicated sorting or permutation procedures, and can be implemented easily in any two-party (or multi-party) computation framework with basic operations such and comparisons (see Section 3.3.3). As Laud's protocol is based on arithmetic sharing in a 3-party setting, any direct comparison of running times would not be fair. For this reason, we attempt to find a lower bound on the number of communication rounds required by this protocol in Section 10.2, and compare this with the number of rounds required by our own protocol.

In a subsequent work, Anagreh, Vainikko, and Laud attempt to use the primitives from Laud's protocol [25] for an optimized secure version of Prim's algorithm [1]. However, it is not suitable for our setting, as the use of Prim's algorithm means that it can be applied only to *connected* graphs (i.e., a minimum spanning tree instead of a forest needs to be computed). Furthermore, the presented protocol is based on adjacency matrices, and therefore the total communication cost is at least quadratic in the number of vertices.

Chapter 3

Preliminaries

In this chapter, we give all necessary definitions for graphs (Section 3.1) and minimum spanning forests (Section 3.2). Furthermore, we present the background on two-party computation and their protocols in Section 3.3.

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

3.1 Graphs

We define the universe of all possible *vertices* in a graph as $\mathcal{V}$, and the universe of all possible *edges* in a graph as $\mathcal{E}$. The function $\mathbf{r} : \mathcal{E} \to \{\{u, v\} \mid u, v \in \mathcal{V}, u \neq v\}$ maps each edge to an unordered set containing its two *endpoints*. $\mathbf{w} : \mathcal{E} \to \mathbb{N}$ assigns a *weight* to each edge. A *finite, weighted, undirected multigraph* consists of a pair (V, E) of two finite sets $V \subseteq \mathcal{V}$ and $E \subseteq \mathcal{E}$, s.t. for all edges $e \in E$, its two endpoints $\mathbf{r}(e) = \{u, v\}$ are in V. Since this is the only type of graphs we look at in this work, we may also just call them *graphs*.

For a finite set of edges $E \subseteq \mathcal{E}$, we define its *total weight* as $\mathbf{w}(E) := \sum_{e \in E} \mathbf{w}(e)$. We might also consider the subset $E_{=w} \subseteq E$ of *edges restricted to a certain weight* w, i.e., $E_{=w} := \{e \in E \mid \mathbf{w}(e) = w\}$. Similarly, sometimes we also consider $E_{<w}$ or $E_{\leq w}$, which are defined as the subset of E containing only those edges that have weight strictly less than w, or at most w, respectively.

Any $c \subseteq \mathcal{V}$ is called a *component* of $\mathcal{V}$. A *partitioning* C is a set of mutually disjoint components of $\mathcal{V}$. A partitioning C is a partitioning of a finite set of vertices $V \subseteq \mathcal{V}$, if

each $v \in V$ is contained in exactly one component $c \in C$. Given partitioning C of V, and a vertex $v \in V$, we use $C(v)$ to denote the unique component $c \in C$ with $v \in c$.

Given a set $V \subseteq \mathcal{V}$ of vertices, we define the corresponding *trivial partitioning* $\overline{V}$ as the partitioning $\{\{v\} \mid v \in V\}$, i.e., as the set consisting of all single-vertex components with vertices from V. When we have a partitioning C, we define $\overline{C}$ as the set of vertices that are contained in C, i.e., $\overline{C} := \bigcup_{c \in C} c$. Note that if C is a partitioning of V, then we have $\overline{C} = V$.

Given a set of edges $E \subseteq \mathcal{E}$, and given a component $c \subseteq \mathcal{V}$, or a partitioning C (s.t. every edge e in E has only endpoints in $\overline{C}$, i.e., $\mathbf{r}(e) \subseteq \overline{C}$), we define

$$E(c) := \{e \in E \mid \mathbf{r}(e) = \{u, v\}, u \in c, v \notin c\},$$
$$E(C) := \{e \in E \mid \mathbf{r}(e) = \{u, v\}, C(u) \neq C(v)\},$$

to be the set of edges having exactly one endpoint in c, or the set of edges with endpoints in two different components from C, respectively. For a set of vertices $V \subseteq \mathcal{V}$ (or a partitioning C with $\overline{C} = V$) and a set of edges $E \subseteq \mathcal{E}$, we define

$$E_C := E_V := \{e \in E \mid \mathbf{r}(e) \subseteq V\}$$

to be the set of all edges for which both endpoints are in V (or both endpoints are in arbitrary components of C). Note that $E_V = E(\overline{V})$. In addition, note that E_C contains all edges with endpoints anywhere in C, while $E(C)$ only contains edges where the two endpoints are in *different* components of C.

For a partitioning C and an edge $e \in \mathcal{E}$ (with endpoints in C), we define $C[e]$ to be the partitioning resulting from C by *merging* the components that are connected by e, i.e., if $\mathbf{r}(e) = \{u, v\}$, then

$$C[e] := \begin{cases} C & \text{if } C(u) = C(v) \\ (C \setminus \{C(u), C(v)\}) \cup (C(u) \cup C(v)) & \text{otherwise} \end{cases}.$$

Note that $C[e]$ is also a partitioning. For a finite set of edges $E \subseteq \mathcal{E}_C$, we let $C[E]$ be the graph partitioning resulting from C by merging the components that are connected through edges in E, i.e., if $E = \{e_1, \ldots, e_k\}$, then $C[E] = C[e_1]...[e_k]$. Note that the order of the edges in E does not matter, and that $C[E]$ is also a partitioning.

For a set $E \subseteq \mathcal{E}$ and a component $c \subseteq \mathcal{V}$, $\min\{\mathbf{w}(e) \mid e \in E(c)\}$ is the minimum weight of an edge from E leaving component c. If there is no such edge, this minimum is defined to be ∞.

3.2 Minimum Spanning Forests

For a graph $G = (V, E)$, 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*, if there are vertices $v_0, \ldots, v_k \in V$ with $\mathbf{r}(e_i) = \{v_{i-1}, v_i\}$ for every $1 \leq i \leq k$. In addition, this path is simultaneously a *cycle*, if $v_0 = v_k$. We say that two vertices $u, v \in V$ are *connected* w.r.t. E to each other if there is a path $(e_1, \ldots, e_k)$ with corresponding vertices $u, v_1, \ldots, v_{k-1}, v$.

A set of edges $F \subseteq E$ is a *spanning forest* of G, if both of the following conditions are satisfied:

(1) F is *spanning*: for every two vertices $u, v \in V$ that are connected on graph $G = (V, E)$, u and v are also connected on graph (V, F), and

(2) F is a *forest*: for any strict subset $F' \subsetneq F$, there are two vertices $u, v \in V$ that are connected on (V, F), but not connected on (V, F'). Equivalently, F does not contain a cycle.

A spanning forest F is a *minimum spanning forest* (MSF) of G, if the following additional condition is satisfied:

(3) F is *minimum*: there is no spanning forest F' of G with $\mathbf{w}(F') < \mathbf{w}(F)$.

For every graph G there is at least one minimum spanning forest. In addition, if $\mathbf{w}(e) \neq \mathbf{w}(e')$ for all $e, e' \in E$ with $e \neq e'$, it is known that there is a *unique* minimum spanning forest. If G has a unique minimum spanning forest F, we define $\mathsf{MSF}(V, E) := F$.

In order to be able 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. For $E' \subseteq E$, we can restrict a permutation $\pi \in E!$ to edges in E', which we denote as $\pi|_{E'}$. This $\pi|_{E'}$ is defined as the unique permutation in $E'!$ that fulfills $\pi|_{E'}(e) < \pi|_{E'}(e') \iff \pi(e) < \pi(e')$ for all $e, e' \in E'$, i.e., the ordering of edges E' in $\pi|_{E'}$ is the same as their ordering in π.

Now, a spanning forest F is a *minimum spanning forest* of G *with tie-breaker* π, if the following new minimality condition is satisfied (which replaces condition (3) above):

(3) F is *minimum*: there is no spanning forest F' of G 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 every graph $G = (V, E)$ and tie-breaker $\pi : E \to [|E|]$, there is exactly one minimum spanning forest, which we denote by $\mathsf{MSF}(V, E, \pi)$.

Usually, we are interested in studying *random tie-breakers*. This means that a uniformly random $\pi \in E!$ is chosen, and $\mathsf{MSF}(V, E, \pi)$ is returned. We denote by $\mathsf{MSF}(V, E)$ the result of this random process. Note that for graphs with a unique minimum spanning forest, the selected tie-breaker has no effect and the result is deterministic. Thus, this definition agrees with our previous definition of $\mathsf{MSF}(V, E)$ if $G = (V, E)$ has a unique MSF.

Our definitions of $\mathsf{MSF}(V, E)$ also work for unweighted graphs. We say that a graph $G = (V, E)$ is unweighted, if there is a fixed weight $w^* \in \mathbb{N}$ with $\mathbf{w}(e) = w^*$, i.e., every $e \in E$ has the same weight. In case of unweighted graphs, we may also just use the term *spanning forest* instead of *minimum spanning forest*, as there is no real notion of a minimum. A *random spanning forest* (i.e., $\mathsf{MSF}(V, E)$) is defined the same way as before, i.e., selecting a random $\pi \in E!$, and then returning the spanning tree that is minimum w.r.t. π.

Note that this definition $\mathsf{MSF}(V, E)$ of an MSF with a random tie-breaker is *not* the same as uniformly choosing a random MSF from the set of all minimum spanning forests. The probability of receiving an MSF F through $\mathsf{MSF}(V, E)$ may be different from the probability of receiving an MSF $F' \neq F$, depending on the structure of the graph G.

3.2.1 Minimum Spanning Forests for Graph Partitionings

Now we extend the definitions for MSFs from the previous section to the case where we have a partitioning C. The semantics are defined so that we can view each component $c \in C$ as a single vertex and each edge between two vertices (in the original sense) as an edge between their two components.

For a graph $G = (V, E)$, and a corresponding partitioning C of V (i.e., $V = \overline{C}$), we call $(e_1, \dots, e_k)$ ($k \geq 1$, $e_i \in E$, $e_i \neq e_j$ for $i \neq j$) a *path* on C, if there are components $c_0, \dots, c_k \in C$ with $\mathbf{r}(e_i) = \{u, v\}$ s.t. $C(u) = c_{i-1}$ and $C(v) = c_i$ for every $1 \leq i \leq k$. In addition, this path is simultaneously a *cycle*, if $c_0 = c_k$. We say that two components $c, c' \in C$ are *connected* w.r.t. C to each other if there is a path $(e_1, \dots, e_k)$ on C with corresponding components $c, c_1, \dots, c_{k-1}, c'$. We say that two vertices $u, v \in V$ are connected w.r.t. C, if $C(u)$ and $C(v)$ are connected to each other.

We can now define a *minimum spanning forest* (MSF) of G w.r.t. C and tie-breaker $\pi : E \to [|E|]$ as a set $F \subseteq E$ for which the following three conditions hold:

(1) for every two components $c, c' \in C$ that are connected on (V, E) w.r.t. C, c and c' are also connected on (V, F) w.r.t. C,

(2) for any strict subset $F' \subsetneq F$, there are two components $c, c' \in C$ that are connected on (V, F), but not connected on (V, F'), and

(3) 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 $\sum_{e \in F'} \pi(e) < \sum_{e \in F} \pi(e)$.

Informally, this definition just means that we are looking for the MSF on the graph where we merge all vertices in a component together, and adjust the edges accordingly. As such, for a fixed tie-breaker π, there is a unique minimum spanning forest, which we denote by $\mathsf{MSF}(C, E, \pi)$. As before, $\mathsf{MSF}(C, E)$ is the output of the random process that uniformly chooses a random $\pi \in E!$ and returns $\mathsf{MSF}(C, E, \pi)$.

Note that for a set of vertices $V \subseteq \mathcal{V}$ and edges $E \subseteq \mathcal{E}$, we have $\mathsf{MSF}(V, E, \pi) = \mathsf{MSF}(\overline{V}, E, \pi)$. This also implies that $\mathsf{MSF}(V, E)$ is a random process with the same output distribution as $\mathsf{MSF}(\overline{V}, E)$.

3.2.2 Connectivity

For a graph $G = (V, E)$, and a corresponding partitioning C of V, we define the connectivity $\mathrm{Conn}(C, E) := \{d_0, \ldots, d_{\ell-1}\}$ s.t. the following criteria are satisfied:

(1) $d_i \neq \emptyset$ for all $i \in [\ell]$ (no d_i is empty),

(2) $C = d_0 \sqcup \cdots \sqcup d_{\ell-1}$ is the union of the pairwise disjoint sets d_i, and

(3) two components $c, c' \in C$ are in the same d_i iff they are connected on G w.r.t. C.

Note that the set $\mathrm{Conn}(C, E)$ always exists and is unique.

Informally, $\mathrm{Conn}(C, E)$ partitions the components of C into equivalence classes, s.t. two components are in the same class iff they are connected to each other w.r.t. C.

3.3 Two-party computation

In two-party computation, we have two parties, each of them owning an input, who want to compute together the output of a predefined function on the given inputs, without gaining any information about the other party's input. The goal is to define an efficient protocol, specifying the actions taken by each party (i.e., communication with each other and local

computations), s.t. the function is computed while preserving privacy. A protocol is always run with security parameters λ and κ that it provides to both parties at the beginning of the execution. We require statistical correctness to be fulfilled w.r.t. λ, and that computational privacy holds w.r.t. κ.

A *functionality* f is a random function that specifies the desired output distribution given the two inputs. If x is the input owned by party 1 and y is the input owned by party 2, then $f(x, y)$ is a random variable (a, b), where a is the output that party 1 receives, and b is the output that party 2 receives. In many scenarios, we want the outputs a and b to be exactly the same. In this case, we may simply write $f(x, y) = a$ instead of $f(x, y) = (a, a)$, where a is the output for both party 1 and party 2.

For example, assume that f is the simple graph problem of testing whether two vertices u and v are connected by a path. Both parties know the complete set of vertices V and the two points u and v, but every party i has a secret set of edges $E^{(i)}$. They want to compute whether there is a path between u and v on the graph consisting of vertices V and edges $E^{(1)} \cup E^{(2)}$. We can model this as follows: party i's input is $(V, u, v, E^{(i)})$, and $f((V, u, v, E^{(1)}), (V, u, v, E^{(2)}))$ is 1 iff there is a path between u and v using edges $E^{(1)} \cup E^{(2)}$, and 0 otherwise. On the other hand, whenever the two parties have different opinions on how V, u, or v should look like, then this problem does not make much sense, and so we let $f((V^{(1)}, u^{(1)}, v^{(1)}, E^{(1)}), (V^{(2)}, u^{(2)}, v^{(2)}, E^{(2)}))$ be undefined if $V^{(1)} \neq V^{(2)}$, $u^{(1)} \neq u^{(2)}$, or $v^{(1)} \neq v^{(2)}$. In this way, we have an option of assuming that there is a "public input" known to both parties, and the protocol only has to output a useful result if both parties think that the public input is the same. In order to emphasize that certain parts of the input are public and known to both parties, we can just write $f_{V,u,v}(E^{(1)}, E^{(2)})$, i.e., we write the public part of the input in the functionality's subscript.

For a given protocol Π, we denote by

$$\mathsf{output}^{\Pi}(\lambda, \kappa, x, y) = (\mathsf{output}_1^{\Pi}(\lambda, \kappa, x, y), \mathsf{output}_2^{\Pi}(\lambda, \kappa, x, y))$$

a random variable that contains the output of running Π with security parameters λ and κ on party 1's input x, and party 2's input y. $\mathsf{output}_1^{\Pi}(\lambda, \kappa, x, y)$ is the random variable containing the output generated by party 1, and $\mathsf{output}_2^{\Pi}(\lambda, \kappa, x, y)$ is the random variable containing the output generated by party 2. We only consider semi-honest security, and therefore we assume that no party ever deviates from Π.

$\mathsf{view}_1^{\Pi}(\lambda, \kappa, x, y)$ is another random variable that contains the *view* that party 1 gets during the protocol execution, and similarly $\mathsf{view}_2^{\Pi}(\lambda, \kappa, x, y)$ is the view of party 2. A view of a party consists (1) of its input, (2) of all the randomness it has generated, and (3) of all the messages it has received. Thus, given a view, it is possible to reconstruct the party's internal state at every timestep.

The *correctness* of a protocol Π is generally handled in different ways by different authors. For example, [14] requires Π to have the exact same output distribution as the functionality f for every input pair (x, y). On the other hand, [12] incorporates the correctness condition into the security condition, implying that it suffices if the protocol's output distribution and the functionality are computationally indistinguishable. While one of our sub-protocols (generating random numbers from a secret-shared interval, see Section 7.3) has a small chance of error in which case the protocol aborts, we still want to have a strong notion of correctness that states that whenever the protocol does not abort, its output distribution is exactly the same as that of the functionality.

Definition 1. The protocol Π *correctly implements functionality* f, if for all x, y and security parameters λ and κ, the protocol fulfills the following two criteria:

- $\Pr[\Pi(\lambda, \kappa, x, y) \text{ aborts}] \leq \text{negl}(\lambda)$ and
- $\Pr[\Pi(\lambda, \kappa, x, y) = z \mid \Pi(\lambda, \kappa, x, y) \text{ does not abort}] = \Pr[f(x, y) = z]$ for any possible output z of the functionality.

Here, $\text{negl}(\lambda)$ means that the given probability grows smaller than the inverse of any polynomial in λ.

Our definition of semi-honest security is a standard one following e.g. that from [14] or [27]. While this definition is security parameter-based, it incorporates only one party's output into each of the equations (3.1) and (3.2).

Definition 2. For a set A of valid inputs, we say that the two probability ensembles $X = \{X(\lambda, \kappa, a)\}_{\lambda,\kappa\in\mathbb{N},a\in A}$ and $Y = \{Y(\lambda, \kappa, a)\}_{\lambda,\kappa\in\mathbb{N},a\in A}$ are computationally indistinguishable (denoted by $X \stackrel{c}{\equiv} Y$), if for every non-uniform polynomial-time algorithm D, there exists a negligible function $\text{negl}(\cdot)$, s.t. for every $a \in A$ and $\lambda, \kappa \in \mathbb{N}$:

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

The protocol Π *privately computes* f *in the semi-honest security model*, if there are polynomial-time algorithms S_1 and S_2 s.t.

$$\{(S_1(1^\kappa, x, f_1(x, y)), f_2(x, y))\}_{\lambda,\kappa,x,y} \stackrel{c}{\equiv} \{(\mathsf{view}_1^\Pi(\lambda, \kappa, x, y), \mathsf{output}_2^\Pi(\lambda, \kappa, x, y)\}_{\lambda,\kappa,x,y} \text{ and} \tag{3.1}$$

$$\{(S_2(1^\kappa, y, f_2(x, y)), f_1(x, y))\}_{\lambda,\kappa,x,y} \stackrel{c}{\equiv} \{(\mathsf{view}_2^\Pi(\lambda, \kappa, x, y), \mathsf{output}_1^\Pi(\lambda, \kappa, x, y)\}_{\lambda,\kappa,x,y}, \tag{3.2}$$

where x, y are inputs for which $f(x, y)$ is defined, and $\lambda, \kappa \in \mathbb{N}$.

Note that we require all algorithms to run in time polynomial in the security parameter κ. This means that we also need to incorporate this security parameter into the functionality f, so that the protocol is not required to work correctly in case of κ being too small to be able to read the complete input. However, in order to keep the notation clean, we just omit this.

For our case of semi-honest security, protocols can easily be composed. This means that it is possible to construct a protocol that *calls* another protocol as a subroutine. If the sub-protocol fulfills semi-honest security, and the main protocol fulfills semi-honest security when restricting the view s.t. it does not include the execution of the sub-protocol, then it fulfills semi-honest security when replacing the call to the sub-protocol by a real execution:

Lemma 1 (Following Theorem 2.2.3 from [14])**.** *Assume that there is a protocol Π^f for functionality f, which is allowed to make oracle queries to a functionality g. If Π^f fulfills semi-honest security (where the view of a party consists only of its input, its random choices, all received messages, and all outputs of calls to g), and there is a semi-honest protocol Π^g computing g, then the protocol Π computing f that results from Π^f by replacing the oracle calls by real executions of Π^g fulfills semi-honest security.*

Note that in the full protocol Π, for each execution of the sub-protocol Π^g, the security parameter κ may be the same as in the complete protocol. However, it might be necessary to increase λ for the call of Π^g so that the full protocol still fails correctness only with probability negligible in its own λ.

For example, if Π^g fails with probability $\leq \frac{1}{2^\lambda}$, and Π^f calls Π^g n times, then every execution of Π^g needs to be run with security parameter $\lambda + \log n$ (i.e., each execution fails with probability $\leq \frac{1}{2^{\lambda+\log n}}$), so that the overall failure probability is $\leq \frac{1}{2^\lambda}$.

3.3.1 Framework

There are many existing protocols that are able to compute any arbitrary functionality that can be expressed as a boolean or arithmetic circuit (see e.g. [12] for a overview of the most important fundamental protocols). However, random-access memory needs to be implemented in a static circuit by scanning through the whole memory to find the value corresponding to a secret index. One possible solution to this is oblivious RAM (ORAM) that attempts to emulate a memory in sublinear time per access (as first formalized by [16], and implementable using e.g. [34]). Since these operations are still very expensive, we avoid

building a large circuit by designing a protocol tailored specifically towards one application: finding a random MSF.

General-purpose protocols are still very useful as building blocks used by larger protocols (like our MSF protocols) if they do not require any RAM. For example, we will see that we need to privately compute the minimum of an integer known to party 1 and an integer known to party 2. This can be done easily by expressing the minimum operation using a boolean circuit, and then evaluating it securely.

Most general-purpose protocols are based on some variant of secret-sharing. This means that an integer or a single bit is split into two shares, s.t. each party knows only one of the two shares, and it looks completely random. When having both shares, it is easy to combine them in order to receive the actual value that they encode.

We base our protocols on *binary secret-sharing*, or more specifically the GMW protocol [15] by Goldreich, Micali, and Wigderson. A bit b is shared between two parties by selecting a random $r \in \{0, 1\}$, which is sent to party 1, and then sending $r \oplus b$ to party 2. To both parties, their shares look uniformly random. Some operations on secret shares do not require any communication. For example, to take the XOR of two secret-shared bit, both parties can simply compute the XOR of their respective shares locally. Multiplication on the other hand requires communication in this protocol, which could be based on a 1-out-of-4 oblivious transfer [14]. In order to optimize the online phase, we assume that Beaver-Triples are used, i.e., precalculated random multiplication triples that can be used to mask real multiplications when running the actual protocol [3].

In our protocols, we can now make use of secret-shared bits, and we can perform computation on them using the GMW protocol: each party's memory consists of their own plaintext data, and secret-shared bits. Secret-shared bits can be concatenated to receive secret-shared integers of a fixed bitlength. Everyone can work on their own local data without communicating, or perform shared computation together with the other party. Any communication has one of the following three forms:

- A party sends plaintext data to the other party. This adds more plaintext data to the second party's internal memory.
- The two parties reveal a secret-shared bit (by both parties sending their own share to the other one). This adds more plaintext data to both party's internal memory.
- Two secret-shared bits are multiplied with each other. This adds another secret-shared bit to both party's internal memory.

Now, any possible binary circuit can be implemented using only bit multiplication and XOR. Section 3.3.3 contains a list of low-level protocols (e.g. addition or comparison of two secret-shared integers) that we may use.

The bottleneck of a protocol can come either from the total amount of communication that is performed between the two parties, or from the network latency. If the latency is very low (e.g. the two parties are located in the same data center), the former might be more relevant. However, we are trying to find a protocol that performs well no matter where the two parties are located. For this reason, we try to minimize the number of communication rounds, so that the network latency is not a significant bottleneck of the protocol.

When using the GMW protocol, the number of required communication rounds is strongly correlated with the *multiplicative depth*. This is because for any two communication steps do not depend on each other, they can be sent together in one message, instead of sending the first one and then waiting for the result before sending the second one. However, whenever a secret-shared bit is computed as the multiplication of two other secret-shared bits, the protocol needs to wait until the shares of the two inputs bits have been calculated before running the multiplication itself. Therefore, whenever possible, any communication should be parallelized, which can be done in a layer-wise manner: all communication steps that do not have any yet unknown dependencies are performed in one batch. Under the assumption that any local computations take a negligible amount of time, and all communication in one layer can be executed in parallel, the total running time depends solely on the multiplicative depth and the time it takes for a message to be sent from one party to the other one.

Note that it would also be possible to run our protocols with a different underlying secret-sharing scheme, implemented not using the GMW protocol. This could include Yao's garbled circuits [37], or arithmetic circuits making use of Shamir's secret-sharing scheme [33] like the BGW protocol [4], or mixed protocols as used in the ABY framework [11]. However, we attempted to optimize our protocols towards binary secret-sharing, as for the purpose of minimum spanning forests, many comparisons or bit-decompositions need to be performed, which is simpler in a binary scheme than in an arithmetic domain.

3.3.2 Protocol Notation

In this work, instead of giving an explicit binary circuit, we always describe the corresponding program in pseudocode. A bit b that is secret-shared is denoted by $[\![b]\!]$, and for convenience we may also just write a secret-shared integer a composed of multiple bits as

	Protocol	Rounds	Total
Bit operations	$\neg[\![b]\!]$	0	0
	$[\![b]\!] \oplus [\![b']\!]$	0	0
	$[\![b]\!] \wedge [\![b']\!]$	1	$O(1)$
	$[\![b]\!] \vee [\![b']\!]$	1	$O(1)$
Arithmetic operations	$[\![a]\!] + [\![a']\!]$	$\lceil \log_2 B \rceil + 1$	$O(B \log B)$
	$[\![a]\!] - [\![a']\!]$	$\lceil \log_2 B \rceil + 2$	$O(B \log B)$
Comparisons	$[\![a]\!] = [\![a']\!]$	$\lceil \log_2 B \rceil$	$O(B)$
	$[\![a]\!] < [\![a']\!]$	$\lceil \log_2 B \rceil + 1$	$O(B)$
	$[\![a]\!] \leq [\![a']\!]$	$\lceil \log_2 B \rceil + 1$	$O(B)$
	$\min([\![a]\!], [\![a']\!])$	$\lceil \log_2 B \rceil + 2$	$O(B)$
Multiplexer	**if** $[\![b]\!]$ **then** $[\![a]\!]$ **else** $[\![a']\!]$	1	$O(B)$
Share and Reveal	$\textsc{Share}_p(a)$	1	$O(B)$
	$\textsc{Reveal}_p([\![a]\!])$	1	$O(B)$
	$\textsc{Reveal}([\![a]\!])$	1	$O(B)$

Table 3.1: The number of communication rounds required for the most common low-level protocols in the GMW protocol. These numbers are collected from [32], where the authors give a list of descriptions of low-level circuits with minimum multiplicative depth. We assume that $[\![b]\!]$ and $[\![b']\!]$ are shares of single bits, and $[\![a]\!]$ and $[\![a']\!]$ are shares of B-bit integers.

$[\![a]\!]$. If a value a is known to party i, we write $\textsc{Share}_i(a)$ to indicate that party i creates a sharing of a and sends one half of it to the other party. $\textsc{Reveal}_i([\![a]\!])$ means that the integer a, which is secret-shared, is revealed only to party i (i.e., the other party sends its own share of a to party i, who can then compute a itself). $\textsc{Reveal}([\![a]\!])$ indicates that a is revealed to both parties, i.e., everyone sends their own share to the other one. Values not written in the notation $[\![\cdot]\!]$ are either public, i.e., known to both parties, or known to one party, depending on the context.

Note that for a protocol doing only operations on secret shares, their security already follows from the security of the GMW protocol [15], because it can be fully described as a binary circuit evaluated within the GMW protocol. Only when we reveal values in the middle of a protocol, we need to provide a proof of security.

3.3.3 Low-level circuit communication cost

There is a variety of low-level protocols working on secret shares. This includes arithmetic operations like $[\![a]\!] + [\![a']\!]$ or $[\![a]\!] - [\![a']\!]$ (which return a secret-shared integer), comparisons like $[\![a]\!] < [\![a']\!]$ or $[\![a]\!] = [\![a']\!]$ (which return a secret-shared bit), bit operations like $[\![b]\!] \vee [\![b']\!]$ or $[\![b]\!] \wedge [\![b']\!]$, or a multiplexer **if** $[\![b]\!]$ **then** $[\![a]\!]$ **else** $[\![a']\!]$ (which returns a secret-shared integer if b is a single bit and a and a' are integers). Schneider and Zohner have summarized depth-optimized circuits for these kinds of operations on binary shares [32]. We recall the number of rounds and the total communication cost required for all low-level circuits used in this work in Table 3.1.

One round means that each party sends data to the other party and receives the data that the other party has sent in the same round. For example, in a network with a round-trip time of 50ms, one round might take 25ms.

When minimizing the number of communication rounds, it is important to keep the number of arithmetic operations and comparisons low, as all of them require a number of rounds that is roughly the logarithm of the integer bitlength. Other operations (like multiplexer, as well as sharing and revealing) all require only one round each.

Note that we give an explicit number of rounds, but only an asymptotic number of communicated bits. This is because the total communication depends on many factors resulting from the specific implementation, and for some of the low-level circuits (like addition) there is no simple explicit formula.

Chapter 4

Problem Definition

In this chapter we formally describe the general problem of computing MSFs using a two-party protocol. We assume that there is a public and finite set of vertices $V \subseteq \mathcal{V}$, and each party p has a private set of edges $E^{(p)}$. The two parties intent to jointly, and securely, compute $\mathsf{MSF}(V, E^{(1)} \cup E^{(2)})$, i.e., on the graph with vertices V and edges resulting from taking the union of the two private edge sets.

We assume that $\mathcal{E}^{(1)} = \binom{V}{2} \times \mathbb{N} \times \{1\}$ is the universe of potential edges that party 1 can choose from, i.e. any edge $e \in \mathcal{E}^{(1)}$ has the form $e = (\{u, v\}, w, 1)$, where $\mathbf{r}(e) = \{u, v\}$ and $\mathbf{w}(e) = w$. Similarly, $\mathcal{E}^{(2)} = \binom{V}{2} \times \mathbb{N} \times \{2\}$ is the universe of edges that party 2 can choose from, i.e., every $e \in \mathcal{E}^{(2)}$ has the form $e = (\{u, v\}, w, 2)$, where $\mathbf{r}(e) = \{u, v\}$ and $\mathbf{w}(e) = w$. As a result, the graph $(V, E^{(1)} \cup E^{(2)})$ may have multi-edges, but any edge e with two fixed endpoints $u, v \in V$ and weight $w \in \mathbb{N}$ can occur only twice: once in $E^{(1)}$ (in the form $(\{u, v\}, w, 1)$), and once in $E^{(2)}$ (in the form $(\{u, v\}, w, 2)$).

This desired behavior is formalized in Functionality 1. The universe of all edges is given as $\mathcal{E} = \mathcal{E}^{(1)} \sqcup \mathcal{E}^{(2)}$. It is guaranteed that $E^{(1)} \subseteq \mathcal{E}^{(1)}$ contains only edges from $\mathcal{E}^{(1)}$ and that $E^{(2)} \subseteq \mathcal{E}^{(2)}$ contains only edges from $\mathcal{E}^{(2)}$. This restriction makes sure that given any edge e in the MSF returned by any protocol, it is possible to determine whether it belongs to party 1 or to party 2.

Functionality 1 Random Minimum Spanning Forest

Public: Partitioning C of a finite set of vertices $V \subseteq \mathcal{V}$
functionality RandomMSF($E^{(1)}$, $E^{(2)}$)
  return $\mathsf{MSF}(C, E^{(1)} \cup E^{(2)})$
end functionality

Note that if $\mathsf{MSF}(V, E^{(1)} \cup E^{(2)})$ for a finite set of vertices V needs to be computed instead of $\mathsf{MSF}(C, E^{(1)} \cup E^{(2)})$ for a partitioning C, then the two parties can simply select $C := \overline{V}$ and run any protocol implementing this functionality. This results in the same output distribution because of $\mathsf{MSF}(V, E) = \mathsf{MSF}(\overline{V}, E)$ for any graph $G = (V, E)$.

In the following chapters, the goal will be to design efficient protocols implementing Functionality 1. We are going to see a protocol that works correctly and is secure only in the restricted setting where all the edges in $E^{(1)} \cup E^{(2)}$ have *distinct weights* (Section 5), a protocol for the case where all the edges in $E^{(1)} \cup E^{(2)}$ have the *same weight* (Section 8), and finally a protocol for the fully generic case where all kinds of weights are allowed (Section 9).

For convenience, whenever the set of edges is shared between two parties, we let $E := E^{(1)} \cup E^{(2)}$ to be the combined set of all edges given in the input.

Chapter 5

Protocol for graphs with distinct weights

For the semi-honest model, Brickell and Shmatikov have proposed a protocol for computing the minimum spanning forest [9] by following either Kruskal's algorithm [23] or Prim's algorithm [30]. Every party has the public set of edges that are already known to be part of the minimum spanning forest. In each iteration, every party chooses the next edge that it would add to the MSF if there were no other parties involved (i.e., the shortest edge that doesn't create a cycle in the MSF set in case of Kruskal's algorithm, or the shortest edge incident to previously discovered vertices in case of Prim's algorithm). This results in two candidates for the next MSF edge: one that is provided by party 1 and one that is provided by party 2. Using any secure multi-party computation protocol for computing the *minimum*, the parties compute together the shorter one of these two edges, and then they reveal it to everyone.

In case of a connected graph, this protocol (regardless of whether Kruskal's or Prim's algorithm is used) requires $\Theta(n)$ sequential applications of the secure protocol computing the minimum edge. Each such application is relatively cheap, because it has a constant number of rounds and total communication cost (i.e., the number of bits sent between the two parties). Thus, the bottleneck becomes the latency produced by the high number of iterations.

In order to address this problem, we give a new protocol for computing minimum spanning forests based on Borůvka's algorithm [8]. As this algorithm can be parallelized easily, it also provides a simple way of processing multiple MSF edges in a single iteration, reducing our number of rounds from linear to $O(\log n)$. The only downside is that, just like protocols based on Prim's or Kruskal's algorithm, this method only works for a graph

where no edge weight occurs more than once in the input (or if there is a publicly known tie-breaker deployed).

In Section 5.1, we describe how Borůvka's algorithm works in a one-party (i.e., non-secure) setting, and in Section 5.2, we translate the algorithm into a two-party computation protocol, whose security is proven in Section 5.3.

5.1 Borůvka's Algorithm

Borůvka's algorithm [8] makes use of the following fact that holds in a graph with unique edge weights: among all edges incident to a fixed vertex u, the best incident edge (i.e., the unique edge with smallest weight that has u as one of its endpoints) is always part of the MSF.

This fact can be used on all n vertices simultaneously, which yields n edges. However, these are not necessarily distinct, as two vertices that are directly connected may have chosen the same edge. This means that the set of selected edges (without duplicates) has size $\geq \lceil \frac{n}{2} \rceil$.

Now, all vertices that are connected by a chosen edge are merged into a single component and the next iteration starts: for each component, we look for the smallest incident edge connecting it to a different component. This process is repeated at most $\lfloor \log_2 n \rfloor$ times, as the number of components is reduced by half in each iteration.

This procedure is formalized in Algorithm 2. We assume that $V \subseteq \mathcal{V}$ is a finite set of vertices, and that $E \subseteq \mathcal{E}$ is a finite set of edges with unique weights (i.e., there are no two edges $e_1, e_2 \in E$, $e_1 \neq e_2$ with $\mathbf{w}(e_1) = \mathbf{w}(e_2)$). The algorithm's task is to return $\mathsf{MSF}(V, E)$, which is known to be unique (i.e., no randomization is required) due to the unique edge weights.

The set of components C can be maintained using a union-find data structure. In each of the $\lfloor \log_2 n \rfloor$ iterations, each component $c \in C$ is checked for its best incident edge e. If e exists (i.e., there is at least one edge leaving component c), it is added to M, a set that holds all edges selected in the current iteration. At the end of the iteration, vertices connected by edges in M are merged ($C \leftarrow C[M]$), and all selected edges are added into F ($F \leftarrow F \cup M$), which holds the final set of edges.

Note that the loop going over all components $c \in C$ in Algorithm 2 can be parallelized, as the selection of edges are independent of each other. This motivates our new protocol requiring less communication rounds than those relying on Kruskal's or Prim's algorithm.

Algorithm 2 Borůvka's MSF algorithm

```
1: function BorůvkaMSF(V, E)
2:     F ← ∅
3:     C ← V̄
4:     for i = 1, ..., ⌊log_2 |V|⌋ do
5:         M ← ∅
6:         for c ∈ C do
7:             e ← arg min_{e∈E(c)} w(e)
8:             if e ≠ ⊥ then
9:                 M ← M ∪ {e}
10:            end if
11:        end for
12:        C ← C[M]
13:        F ← F ∪ M
14:    end for
15:    return F
16: end function
```

5.2 Adaption as a two-party computation protocol

Now we transform the described algorithm into a protocol for two-party computation. After each of the $\lfloor \log_2 n \rfloor$ iterations, every party knows the current state of the partitioning C, and the set F of edges already selected for the MSF. The only remaining question is how to securely compute the best incident edge e for every component $c \in C$.

This problem can be easily solved as follows: For each $c \in C$, every party p can locally compute the best edge $e^{(p)} \in E^{(p)}$ going out of c. Then, in order to find the overall best edge in $E^{(1)} \cup E^{(2)}$, it suffices to compare the best edge $e^{(1)}$ from $E^{(1)}$ with the best edge $e^{(2)}$ from $E^{(2)}$. This can be done by secret-sharing the two values $\mathbf{w}(e^{(1)})$ and $\mathbf{w}(e^{(2)})$, and comparing them securely (i.e., testing which one of them is the smaller one). The result of this comparison can be revealed, and the party owning the better edge sends all information about this edge (its endpoints and its weight) to the other party. Both of them can then add this edge to M.

Protocol 3 (which implements Functionality 1) formalizes this procedure. It assumes that each party p is given its set of edges $E^{(p)} \subseteq \mathcal{E}^{(p)}$ s.t. no weight occurs in more than one edge in $E^{(1)} \cup E^{(2)}$, and that an MSF sampled from $\mathsf{MSF}(V, E^{(1)} \cup E^{(2)})$ needs to be returned. Note that we also assume that the initial partitioning is $C = \overline{V}$, i.e., each vertex is in its own partition. This is just to simplify the analysis of Protocol 3 and its comparison

with Algorithm 2, and the protocol would also work if the initial partition C implies that two or more vertices are already merged.

Protocol 3 Adaption of Borůvka's MSF algorithm as a two-party protocol

Public: Partitioning $C = \overline{V}$ of a finite set of vertices $V \subseteq \mathcal{V}$
Precondition: There are no two edges $e, e' \in E^{(1)} \cup E^{(2)}$, $e \neq e'$ with $\mathbf{w}(e) = \mathbf{w}(e')$
protocol BorůvkaMSF($E^{(1)}, E^{(2)}$)
    $F \leftarrow \emptyset$
    for $i = 1, \ldots, \lfloor \log_2 |V| \rfloor$ **do**
        $M \leftarrow \emptyset$
        for $c \in C$ **do**
            Party p: $e^{(p)} \leftarrow \arg\min_{e \in E^{(p)}(c)} \mathbf{w}(e)$ (for both $p = 1, 2$)
            $[\![v^{(1)}]\!] \leftarrow \text{Share}_1(\mathbf{w}(e^{(1)}))$ and $[\![v^{(2)}]\!] \leftarrow \text{Share}_2(\mathbf{w}(e^{(2)}))$
            if Reveal($[\![v^{(1)}]\!] < [\![v^{(2)}]\!]$) **then**
                Party 1 sends $e^{(1)}$ to **Party** 2
                $M \leftarrow M \cup \{e^{(1)}\}$
            else if $v^{(2)} < \infty$ **then**
                Party 2 sends $e^{(2)}$ to **Party** 1
                $M \leftarrow M \cup \{e^{(2)}\}$
            end if
        end for
        $C \leftarrow C[M]$
        $F \leftarrow F \cup M$
    end for
    return F
end protocol

Note that if a party p does not find any edge going out of c, we assume that it defines $e^{(p)} = \perp$, and $\mathbf{w}(e^{(p)}) = \infty$. Thus, if the other party still has an edge incident to c, then the other's weight will definitely be smaller than ∞. If none of the parties found any edge (i.e., $v^{(1)} = \infty = v^{(2)}$), then this information can be communicated and no party sends any edge to the other one.

The only difference between the two-party Protocol 3 and the one-party Algorithm 2 is the way in which edges are selected. However, the protocol guarantees that we always choose the best edge from $(E^{(1)} \cup E^{(2)})(c)$. Therefore, this protocol imitates Algorithm 2 on edges $E^{(1)} \cup E^{(2)}$, and inherits its correctness.

5.3 Security

Despite revealing the edges and the comparison results during the course of the protocol, Protocol 3 is secure, as we show in Theorem 1. This is because given the full MSF returned by the protocol, it is possible to infer all comparison results and edges sent in each step.

Theorem 1. *Protocol 3 privately computes Functionality 1 in the semi-honest security model if no edge weight occurs more than once in the combined input.*

Proof. A view view_1 in this protocol consists of $(E^{(1)}, q_1, \ldots, q_{\lfloor \log_2 |V| \rfloor})$, where q_i is the view corresponding to one of the iterations. Thus, $q_i = (s_c, e_c)_{c \in C_i}$ is comprised of all comparison results and edges sent in iteration i, with C_i being the current state of the partitioning in iteration i. Note that we did not add secret shares themselves to the view, for the simple reason that they are always completely random-looking (and independent of each other). Thus, a simulator can just insert a random value for each secret-share in the view, and this always has the correct distribution.

A simple simulator $S(E^{(1)}, F)$ (for party 1, a simulator for party 2 works in an analogous way) that only requires input $E^{(1)}$ and the MSF F returned by Protocol 3 in its input can be defined in the following way: S simulates the Protocol 3 pretending that party p's input was $F^{(p)} := \{(\{u, v\}, w^*, p^*) \in F \mid p^* = p\}$. Informally, $F^{(p)}$ contains the set of MSF edges that belong to party p, and we are going to show that it makes no difference whether the input to the protocol was $(E^{(1)}, E^{(2)})$ or $(F^{(1)}, F^{(2)})$.

We need to show that the output $(E^{(1)}, q'_1, \ldots, q'_{\lfloor \log_2 |V| \rfloor})$ of S is exactly the same as view_1, where $q'_i = (s'_c, e'_c)_{c \in C'_i}$, with C'_i being the current partitioning state within the simulator in iteration i. In order to do this, we need to show $q_i = q'_i$ for any iteration i. Because the real partitioning state C_1, and the simulated partitioning state C'_1 are both equal to $\overline{V}$ in the first iteration, they are also equal to each other ($C_{i+1} = C'_{i+1}$) in iteration $i + 1$ if we prove $q_i = q'_i$ for every i under the assumption $C_i = C'_i$.

Therefore, it suffices to show that for every i, if C_i is the current partitioning state, then the comparison result s'_c is the same as s_c, and that $e'_c = e_c$ for every $c \in C_i$. So assume that the protocol is trying to find the best edge going out of $c \in C_i$. Let $e^{(p)} \leftarrow \arg\min_{e \in E^{(p)}(c)} \mathbf{w}(e)$ be the edge selected by party p in the real execution. Let $e'^{(p)} \leftarrow \arg\min_{e' \in F^{(p)}(c)} \mathbf{w}(e')$ be the best edge selected by party p in the simulation of S.

Assume that in the real world, the comparison is true, i.e., $s_c = 1$, and we had $\mathbf{w}(e^{(1)}) < \mathbf{w}(e^{(2)})$. Then, $e^{(1)}$ was sent to party 2 and it must have been added to $F^{(1)}$. Thus, the edge $e'^{(1)}$ selected by the simulator must be at least as good as $e^{(1)}$: $\mathbf{w}(e'^{(1)}) \leq \mathbf{w}(e^{(1)})$. In addition, we can't have the strict inequality $\mathbf{w}(e'^{(1)}) < \mathbf{w}(e^{(1)})$ (because $e'^{(1)} \in F^{(1)} \subseteq E^{(1)}$

would have a smaller weight than $e^{(1)}$, a contradiction to the minimality of $e^{(1)}$). Similarly, we know that $\mathbf{w}(e'^{(2)}) \geq \mathbf{w}(e^{(2)})$, and combining these relations yields

$$\mathbf{w}(e'^{(1)}) = \mathbf{w}(e^{(1)}) < \mathbf{w}(e^{(2)}) \leq \mathbf{w}(e'^{(2)}).$$

This means that the simulator determined the correct result $s'_c = 1$ of the comparison between the edges selected by the two parties for component c. Because the weights of $e'^{(1)}$ and $e^{(1)}$ are equal, by uniqueness of edge weights, we also have $e'^{(1)} = e^{(1)}$. This means that the edge $e'^{(1)}$ added to the view by the simulator is exactly the same as the real edge $e^{(1)}$ that party 1 sent to party 2.

If $\mathbf{w}(e^{(1)}) \geq \mathbf{w}(e^{(2)})$ (i.e., $s_c = 0$), then for an analogous reason, we also have $\mathbf{w}(e'^{(1)}) \geq \mathbf{w}(e'^{(2)})$, and $e'^{(2)} = e^{(2)}$, thus showing that the simulator always produces the correct comparison result, and the correct edge sent by one of the parties to the other. □

Communication cost

Let B be the bitlength used for edge weights. This imposes the restriction of any weight being in the range $[0, 2^B)$. A typical implementation could reserve weight $2^B - 1$ for encoding the value ∞ to indicate that no such edge exists at all.

There are $\lfloor \log_2 |V| \rfloor$ iterations of the outer loop. Within each such iteration, all components $c \in C$ can be processed simultaneously because they do not depend on each other's results. For any $c \in C$, the protocol first needs to share $v^{(1)}$ and $v^{(2)}$ in one round, then run $[\![v^{(1)}]\!] < [\![v^{(2)}]\!]$, which takes $\lceil \log_2 B \rceil + 1$ rounds, and revealing its result requires another round. Afterwards, one of the parties might need to send an edge to the other party, which can be done in plaintext and does not require any multiplication, but still adds one communication round to the cost.

The total communication cost is dominated by the comparisons $[\![v^{(1)}]\!] < [\![v^{(2)}]\!]$. For every component $c \in C$, such a comparison only needs to be performed once (if nobody has any incident edge, i.e., $v^{(1)} = \infty = v^{(2)}$, in iteration i, then nobody will have any incident edge in iteration $i + 1$ either, therefore making it unnecessary to re-run the comparison in the next iteration if no edge was added). Over the course of the whole protocol, there will be less than $2|V|$ different components in C: Initially, there are $|V|$ components, and each of the $< |V|$ merges can create only one new component. Every comparison requires $O(B)$ multiplications, and therefore the total communication cost is bounded by $O(|V| \cdot B)$.

Number of rounds: $\leq \lfloor \log_2 |V| \rfloor \cdot (\lceil \log_2 B \rceil + 4)$

Total communication: $O(|V| \cdot B)$

Chapter 6

Issues arising from edges with the same weight

Borůvka's algorithm itself (even in a local one-party computation as in Algorithm 2) does not necessarily work correctly if the minimum spanning forest is not unique (i.e., there must be at least two edges with the same weight). The problem is that in a single iteration, there might be a cycle of edges added to the set of edges forming the minimum spanning forest. For example, look at the fully connected graph with $n = 3$ vertices and equal weights on all edges. In the first (and only) iteration, the algorithm may choose for every vertex any of the two incident edges, as all of them have the same weight. Thus, it may choose all three edges simultaneously, which produces a cycle and clearly contradicts the definition of a minimum spanning forest.

For the one-party algorithm, this problem can be addressed by using any arbitrary *tie-breaker*. This could for example be the index of the edge in the array of all edges, or a random secondary weight s.t. no two edges share the same value. Now, whenever two edges have the same weight, they can still be compared by looking at the tie-breaker which implicitly defines a total ordering on the set of all edges.

Doing this guarantees that Borůvka's algorithm finds a minimum spanning forest. Which of the MSFs is returned (in case of non-uniqueness of the MSF) depends on the tie-breaker used. For example, using a deterministic tie-breaker like the index of the edge results in a deterministic MSF. In some cases, one might want to find a random minimum spanning forest by choosing random secondary weights, as in our randomized definition of $\mathsf{MSF}(V, E)$.

6.1 Leaky tie-breaker for two-party protocols

A similar solution works for finding a minimum spanning forest in a graph with non-unique edge weights with a two-party protocol. Protocol 3 can be adapted in a straightforward way to add tie-breakers to the comparison of two edges. However, this may result in some undesired leakage of information. This is because the tie-breaker value itself needs to be part of the output of the protocol. Only if this is done, will the simulator be able to correctly reconstruct the view of a party given the output and this party's input.

If this leakage is acceptable, tie-breakers π can be computed in the following way: In the beginning, every party assigns the value of π to each of its edges (for example, an index, or a unique random value). Now, whenever two edges need to be compared (which can happen locally when a party needs to find the best incident edge, or securely when the parties compare their respective edges), it needs to be possible to do this using only the two weights and the tie-breaker π. There must be a secure two-party computation protocol for computing the result of such a comparison. When the owner of an edge e needs to share it in plaintext (because it turns out that it is part of the desired minimum spanning forest), both the endpoints $\mathbf{r}(e)$ and the weight $\mathbf{w}(e)$, and the tie-breaker $\pi(e)$ need to be published.

This tie-breaking process preserves security of the two-party protocol: Given the additional data π that is sufficient for inferring the total ordering of the edges, the simulator can correctly determine the order in which the edges where added (like in Theorem 1 for the case of unique edge weights).

Examples for possible tie-breakers computable in this fashion include:

- Every party i randomly permutes its edges and then uses their new indices as the tie-breaker π. Now, two edges are compared (1) by weight, (2) by permutation index, and (3) by the index of the edge's owner (i.e., 1 or 2, depending on which party the edge belongs to).

  The disadvantage of this method is that it may be slightly biased towards edges belonging to one of the two parties, and that the permutation indices of the edges in the final MSF that belong to each of the parties are revealed. This might reveal information about the number of edges that each party owns.

- It is also possible to assign long random tie-breakers π to all edges. Note that π will not necessarily need to be a permutation $E \to [|E|]$, but it suffices if $\pi(e) \neq \pi(e')$ for all distinct edges $e, e' \in E$, $e \neq e'$. With long random values of $\pi(e)$ for all $e \in E$, if

the bitlength of these values is large enough, the probability of two edges having the same value can be made negligibly small.

The corresponding leakage reveals less information about the number of edges owned by the other party than the previous method. A completely random tie-breaker π might even intuitively sound like it doesn't give anything away. However, this is not true: for example, if a party has a lot of edges of a specific weight w^* that are added to the MSF, it is likely that many of these edges have a low tie-breaker value $\pi(\cdot)$. If there are not many edges of w^*, but they are still added to the minimum spanning forest (e.g. because w^* is small), then their values of π can have a "more uniform" distribution, i.e., they have larger tie-breaker values. Thus, the additional leakage can reveal something about the edges owned by the other party, and this additional information gained is hidden behind the seemingly random secondary weights.

- One approach that doesn't require any randomness is the following: Use the indices of the two endpoints of an edge, or secondarily the number of the owning party (i.e., 1 or 2) as its tie-breaker. That is, if two edges e and e' with $\mathbf{r}(e) = \{u, v\}$ and $\mathbf{r}(e') = \{u', v'\}$ have equal weight $\mathbf{w}(e) = \mathbf{w}(e')$, then they are compared by u and u'. If $u = u'$, then they are compared by v and v'. If these indices are still equal, then the edge that belongs to party 1 is defined to be the smaller one.

  This approach does not have any additional leakage (because all information by which two edges are compared, including its weight, endpoints, and owning party, are always given in the output, even without a tie-breaker). However, in many scenarios it might be undesirable to use such a deterministic tie-breaker, as it is heavily biased towards vertices with smaller indices if there many duplicated edge weights.

6.2 Omitting tie-breaker from the output

Since the goal is to only compute the minimum spanning forest, leaking any additional graph information might not be acceptable for certain use cases. This is the reason why we are looking for a protocol that *only* outputs the minimum spanning forest without any other information. One very apparent solution that might seem correct is to just leave away the additional tie-breaker information from the output. For example, assume that each party assigns random secondary weight $\pi(\cdot)$ to all its edges, and when sharing an edge e in plaintext with the other party, they only transmit the endpoints $\mathbf{r}(e)$ and the original weight $\mathbf{w}(e)$, not the random tie-breaker $\pi(e)$.

Now, since the tie-breaker values were selected in a uniformly random way, a potential simulator (that only sees the minimum spanning forest without the tie-breaker π) could

Figure 6.1: Two different graphs that can be distinguished when the tie-breaker π is leaked. Party 1's input $E^{(1)} = \{e_1, e_2, e_3\}$ consists of the thick edges, while party 2's input consists of either $E^{(2)} = \{e_4\}$ (in the left graph) or $\tilde{E}^{(2)} = \{\tilde{e}_4\}$ (in the right graph). To show that we can distinguish between the two graphs, we calculate the probabilities of the MSF consisting of exactly the thick edges (i.e., $E^{(1)}$) and simultaneously satisfying $\pi(e_1) < \pi(e_2) < \pi(e_3)$.

assign random values for π on its own and compute the views accordingly. While for a fixed input, the distribution of such a simulated view sounds like it should be the same as the real distribution of all views producing this minimum spanning forest, this is actually not the case for all possible inputs. In fact, it can be proven that there are different input pairs which have a different distribution of views, even with the same overall protocol output. Thus, given any simulator, it is possible to computationally distinguish between its simulation and the real distribution of views on at least one of these inputs. As a result, no correct simulator can exist.

Theorem 2. *Assume that there is a protocol* Π *implementing Functionality 1. This means that the protocol returns* $\mathsf{MSF}(V, E, \pi)$ *for a uniformly random* $\pi : E \to [|E|]$*. If one of the parties* p *is able to determine the relative ordering of the edges in* $E^{(p)}$ *that were selected for the MSF (i.e.,* $\mathsf{MSF}(V, E, \pi) \cap E^{(p)}$*) w.r.t.* π*, 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 that were selected for the MSF. We show that there is no simulator for party 1 satisfying the necessary conditions for semi-honest security (Definition 2).

We construct two graphs that we will be able to distinguish, even when the same MSF is returned by protocol Π. They are shown in Figure 6.1. 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 a $\pi : E^{(1)} \cup E^{(2)} \to [4]$, let $F(\pi) := \mathsf{MSF}(V, E^{(1)} \cup E^{(2)}, \pi)$ be the MSF generated in the first graph on tie-breaker π. For a $\pi : E^{(1)} \cup \tilde{E}^{(2)} \to [4]$, let $\tilde{F}(\pi) := \mathsf{MSF}(V, E^{(1)} \cup$

$E^{(2)}, \pi)$ be the MSF generated in the second graph on tie-breaker π.

For the purpose of a contradiction, assume that there is a simulator S for party 1 fulfilling semi-honest security, and that the protocol's output will be $E^{(1)}$. We know that it is possible (by assumption) to efficiently compute $\pi|_{E^{(1)}}$ given view_1^{Π}. Thus, we may assume that simulator S computes $\pi|_{E^{(1)}}$ instead of the complete view. Then, by equation (3.1),

$$|\Pr[D(S(1^\kappa, E^{(1)}, F(\pi)), F(\pi)) = 1] - \Pr[D(\pi|_{E^{(1)}}, F(\pi)) = 1]| \leq \mathrm{negl}(\kappa) \tag{6.1}$$

$$|\Pr[D(S(1^\kappa, E^{(1)}, \tilde{F}(\pi)), \tilde{F}(\pi)) = 1] - \Pr[D(\pi|_{E^{(1)}}, \tilde{F}(\pi)) = 1]| \leq \mathrm{negl}(\kappa) \tag{6.2}$$

where $\pi : E^{(1)} \cup E^{(2)} \to [4]$ (or $\pi : E^{(1)} \cup \tilde{E}^{(2)} \to [4]$ in case of the second graph) is the uniformly random permutation of all edges, and $\pi|_{E^{(1)}} : E^{(1)} \to [3]$ is the relative ordering restricted to edges $E^{(1)}$.

We construct a distinguisher D as follows: $D(\pi|_{E^{(1)}}, M)$ returns 1 iff $M = E^{(1)}$ and $\pi(e_1) < \pi(e_2) < \pi(e_3)$. That is, D returns 1 iff the MSF returned by the protocol consists exactly of edges $E^{(1)}$, and they are ordered by the tie-breaker π as $\pi(e_1) < \pi(e_2) < \pi(e_3)$. Note that $\pi(e_1) < \pi(e_2) < \pi(e_3)$ can be efficiently checked by D as it is equivalent to $\pi|_{E^{(1)}}(e_1) < \pi|_{E^{(1)}}(e_2) < \pi|_{E^{(1)}}(e_3)$, and D has knowledge of $\pi|_{E^{(1)}}$.

Note that the generated MSF for the first graph is equal to $E^{(1)}$ (i.e., $\mathsf{MSF}(E^{(1)} \cup E^{(2)}, \pi) = E^{(1)}$) iff $\pi(e_1) < \pi(e_4)$ and $\pi(e_2) < \pi(e_4)$. The probability for this happening (over the random choice of π) is exactly $\frac{1}{3}$. Similarly, the probability of $\mathsf{MSF}(E^{(1)} \cup \tilde{E}^{(2)}, \pi) = E^{(1)}$ over the random choice of π is exactly $\frac{1}{3}$. In other words, no matter whether party 2's input is $E^{(2)}$ or $\tilde{E}^{(2)}$, the probability for the MSF containing exactly the edges $E^{(1)}$, but not the edge provided by party 2, is exactly $\frac{1}{3}$.

Thus, the probability of the distinguisher returning 1 must be the same, regardless of whether party 2 was providing e_4 or $\tilde{e}_4$:

$$\begin{aligned}
&\Pr[D(S(1^\kappa, E^{(1)}, F(\pi)), F(\pi)) = 1] \\
=\ &\Pr[F(\pi) = E^{(1)} \text{ and } D(S(1^\kappa, E^{(1)}, E^{(1)}), E^{(1)}) = 1] \\
=\ &\frac{1}{3} \cdot \Pr[D(1^\kappa, S(E^{(1)}, E^{(1)}), E^{(1)}) = 1] \\
=\ &\Pr[\tilde{F}(\pi) = E^{(1)} \text{ and } D(S(1^\kappa, E^{(1)}, E^{(1)}), E^{(1)}) = 1] \\
=\ &\Pr[D(S(1^\kappa, E^{(1)}, \tilde{F}(\pi)), \tilde{F}(\pi)) = 1]
\end{aligned}$$

Combined with the two inequalities (6.1) and (6.2), we may conclude that D is unable distinguish between the two scenarios (if Π were secure):

$$|\Pr[D(\pi|_{E^{(1)}}, F(\pi)) = 1] - \Pr[D(\pi|_{E^{(1)}}, \tilde{F}(\pi)) = 1]| \leq \mathrm{negl}(\kappa). \tag{6.3}$$

However, this does not hold:

- $D(\pi|_{E^{(1)}}, F(\pi))$ returns 1 for exactly two permutations of π: $\pi(e_1) < \pi(e_2) < \pi(e_3) < \pi(e_4)$ and $\pi(e_1) < \pi(e_2) < \pi(e_4) < \pi(e_3)$.
- $D(\pi|_{E^{(1)}}, \tilde{F}(\pi))$ returns 1 for exactly one permutation of π: $\pi(e_1) < \pi(e_2) < \pi(e_3) < \pi(\tilde{e}_4)$.

Thus, the difference between the two probabilities in equation (6.3) is $\frac{1}{4!}$, which is a constant and therefore not negligible.

In conclusion, the assumed simulator cannot exist, and protocol Π is not secure. □

Theorem 2 shows that if the tie-breaker π is supposed to stay secret, but (partial) information about it is revealed through a party's view, then this party can be capable of inferring information about the other party's input (that it would not have known without the leakage through the simulator). In particular, even if all values of π are kept secret-shared, and the protocol is designed in such a way that a party can only infer information about its own edge ordering, the protocol cannot be secure. This also means that if a protocol assigns random tie-breaker values (in public) to all edges, but they are not part of the functionality that is supposed to be computed (e.g. Functionality 1 in this work), then this protocol is insecure.

Note that in the proof we assumed that Π is perfect, and produces the correct output with probability 1. However, even if a security parameter λ is used and the protocol may fail with negligible probability, D can still distinguish between the two graphs with non-negligible probability.

Chapter 7

Building blocks

There is a variety of smaller sub-protocols we will use for computing a random MSF of an unweighted graph (Protocol 12) and for our fully general MSF protocol (Protocol 15). We define them here and show their correctness and security (when non-trivial), in order to keep the later presentations simple.

7.1 Find first

This building block takes a secret-shared list $L = [[\![L_0]\!], [\![L_1]\!], \ldots, [\![L_{|L|-1}]\!]]$ of public length, where each element in L is of the form $([\![d]\!], [\![x]\!])$. $[\![d]\!]$ is an arbitrary secret-shared value (or multiple values, depending on the application), while $[\![x]\!]$ is a secret-shared bit. The output is the first element for which $x = 1$. If there is no such element (i.e., x is 0 for each entry in L), then the last element ($[\![L_{|L|-1}]\!]$) is returned. This is formally described by Functionality 4.

Functionality 4 Find first

functionality FINDFIRST(L)
  if $x = 0$ for all $([\![d]\!], [\![x]\!]) \in L$ **then**
    return $[\![L_{|L|-1}]\!]$
  else
    return $[\![L_i]\!]$ for the smallest i with $[\![L_i]\!] = (\_, [\![1]\!])$
  end if
end functionality

Note that FindFirst(L) returns two secret-shares $[\![d]\!]$ and $[\![x]\!]$. Since secret-shares always look completely random, the returned share $([\![d]\!], [\![x]\!])$ is independent of the shares $[\![L_i]\!] = ([\![d]\!], [\![x]\!])$ of selected index i, even though they represent the same values. This makes it impossible to determine which i is the smallest for which the corresponding $[\![x]\!]$ is 1.

Protocol 5 implements this behavior. It depends on the ability to "merge" two consecutive elements in L. Such a merge of two entries $([\![d]\!], [\![x]\!])$ and $([\![d']\!], [\![x']\!])$ results in $([\![d]\!], [\![x]\!])$ if $x = 1$ or $([\![d']\!], [\![x']\!])$ otherwise. When doing this repeatedly until L consists only of one entry, this remaining entry is the overall result.

Protocol 5 Find first

protocol FindFirst(L)
  while $|L| > 1$ **do**
    $L' \leftarrow []$
    for $i = 0, 2, \ldots$ with $i + 1 < |L|$ **do**
      $([\![d]\!], [\![x]\!]) \leftarrow [\![L_i]\!]$, $([\![d']\!], [\![x']\!]) \leftarrow [\![L_{i+1}]\!]$
      $[\![d^*]\!] \leftarrow$ **if** $[\![x]\!]$ **then** $[\![d]\!]$ **else** $[\![d']\!]$
      $[\![x^*]\!] \leftarrow [\![x]\!] \vee [\![x']\!]$
      Append $([\![d^*]\!], [\![x^*]\!])$ to L'
    end for
    Append $[\![L_{|L|-1}]\!]$ to L' **if** $|L|$ is odd
    $L \leftarrow L'$
  end while
  return L_0
end protocol

Theorem 3. *Protocol 5 correctly implements Functionality 4.*

Proof. We show correctness by induction over increasing $|L|$.

Assume $|L| = 1$. In that case $[\![L_0]\!]$ is returned, which obviously fulfills the functionality regardless of the value of x.

Assume $|L| > 1$. If $x = 0$ for all $([\![d]\!], [\![x]\!]) \in L$, then L' will also only contain entries $([\![d]\!], [\![x]\!])$ with $x = 0$. Thus, by induction, the protocol will output $[\![L'_{|L'|-1}]\!]$, which is equal to $[\![L_{|L|-1}]\!]$ (whether $|L|$ is odd or not). Otherwise, let i be the smallest index with $[\![L_i]\!] = (\_, [\![1]\!])$. Then, $[\![L'_{\lfloor \frac{i}{2} \rfloor}]\!] = [\![L_i]\!]$ and $[\![L'_j]\!] = (\_, [\![0]\!])$ for all $j < \lfloor \frac{i}{2} \rfloor$. Thus, by induction, the protocol will output $[\![L_i]\!]$. □

Since this protocol is working solely on secret-shares, semi-honest security follows directly from the security of the secret-sharing scheme, i.e., the GMW protocol in our case.

Communication cost

The cost of communication depend on the size $|L|$ of the given list, and the bitlength B of a single element in L.

There are exactly $\lceil \log_2 |L| \rceil$ iterations of the while-loop, as the number of elements remaining decreases from n to $\lceil \frac{n}{2} \rceil$ in every iteration. Note that within every iteration, all i's can be processed simultaneously, as they do not depend on each other's results. In addition, $[\![d^*]\!]$ and $[\![x^*]\!]$ can be computed simultaneously. Both the multiplexer and the bit-or take exactly one round, and therefore each iteration of the while-loop requires only one round. There are $|L| - 1$ merge step in total, and each of them uses $B + 1$ multiplications (B for the multiplexer, and 1 for the bit-or).

This approach of merging two consecutive elements is much better than a linear scan: The number of rounds grows only logarithmically with the length of L, while for a linear scan it would grow linearly.

Number of rounds: $\lceil \log_2 |L| \rceil$

Total communication: $O(|L| \cdot B)$

7.2 Secret prefix operations

Assume that we have a list $L = [[\![L_0]\!], [\![L_1]\!], \ldots, [\![L_{N-1}]\!]]$ of secret-shares of length N. These values can be single bits, integers, or data with an arbitrary amount of bits, as long as they all have the same length. In addition, assume that $\odot$ is an arbitrary known associative binary operation on two of those secret-shared values, for which we have a secure semi-honest protocol (e.g. the logical operator $\vee$ for bits, or $+$ for integers) which requires $r_\odot$ communication rounds and $c_\odot$ total communicated bits per invocation. Now we want to compute shares of $[\![L_0]\!] \odot [\![L_1]\!] \odot \cdots \odot [\![L_i]\!]$ for every $i \in [N]$ quickly. If $\odot$ is an operation that does not require any communication, this is a very simple task. Otherwise, we might want to minimize the number of communication rounds, or the total amount of communication.

In order to do this, we can make use of existing algorithms for computing prefix sums (or other associative operations) in parallel. For example, [20] presents an algorithm for computing the prefix array with any associative operation in $\log_2 N$ depth and $O(N \log N)$

applications of $\odot$ (if N is a power of 2). As shown in Protocol 6, this can be trivially turned into a semi-honest protocol for two parties to compute the prefix-$\odot$ of a list of length N, using just $r_\odot \cdot \log_2 N$ rounds of communication and $O(c_\odot \cdot N \log N)$ total communication.

Protocol 6 Prefix-$\odot$

1: **protocol** PREFIXARRAY$_\odot(L)$
2: **for** $i = 0, 1, \ldots$ with $2^i < |L|$ **do**
3: **for** $j = |L-1|, \ldots, 0$ with $j - 2^i \geq 0$ **do**
4: $[\![L_j]\!] \leftarrow [\![L_j]\!] \odot [\![L_{j-2^i}]\!]$
5: **end for**
6: **end for**
7: **return** L
8: **end protocol**

If the total communication is a bottleneck that needs to be minimized, other parallel prefix array algorithms can be used, which exchange more rounds for fewer operations. For example, [24] presents an algorithm using about $2 \log_2 N$ depth, but on the other hand only $O(N)$ operations in total.

Communication cost

The cost of communication depends on the length $|L|$ of the given list, and on the cost for one operation of $\odot$.

There are exactly $\lceil \log_2 |L| \rceil$ iterations of the outer while loop. All j for one such iteration can be processed simultaneously, because they do not depend on each other's results. Therefore, one outer iteration takes $r_\odot$ rounds. The total communication can be bounded by the number of rounds, multiplied with $|L|$, which is a bound for the number of j's that the protocol iterates over.

Note that a linear scan (i.e., first computing $[\![L_0]\!] \oplus [\![L_1]\!]$, then $[\![L_0]\!] \oplus [\![L_1]\!] \oplus [\![L_2]\!]$, and so on) would result in a linear number of communication rounds, while for the approach presented here, this number grows only logarithmically.

Number of rounds: $\lceil \log_2 |L| \rceil \cdot r_\odot$

Total communication: $O(|L| \cdot \log |L| \cdot c_\odot)$

7.3 Joint random number generation

It will be necessary to generate a secret random number that is within a range $[0, N)$, where $N \geq 0$ is also a non-negative secret number not known to any party (i.e., it is secret-shared). If $N = 0$, it is supposed to return 0. This is shown in Functionality 7.

Functionality 7 Random number generation

functionality RANDINT($[\![N]\!]$)
  return a secret-shared integer generated uniformly from $[0, \max(1, N))$
end functionality

Our approach, shown in Protocol 8, is the same one as [7, Algorithm 7], which works as follows: Both parties prepare λ uniformly random numbers $[0, 2^B)$, where B is the bitlength. They combine them into λ secret-shared random numbers c_i. In our secret-sharing scheme, this comes for free without any communication: each party can take its random number as its own share, and the XOR of these two shares the secret-shared uniformly random number c_i.

Then, the two parties jointly compute a secret bitmask M from N that has the lowest ℓ bits set to one, where ℓ is the highest bit set in N. For every c_i, they take the bit-wise AND of c_i and M. If the result is smaller than N, then it is a uniformly random number in $[0, N)$, otherwise it is rejected. The bit-share $[\![d_i]\!]$ denotes whether c_i was accepted or not.

It suffices if one of the k random numbers is accepted. Finding any of these numbers is done by making use of the FINDFIRST-protocol (Protocol 5).

Note that this protocol will abort if $(c_i \wedge M) \geq N$ for all $i \in [\lambda]$, while $N > 0$ (which is equivalent to $M_0 = 1$). However, M is always smaller than $2N$, and therefore the probability of $c_i \wedge M$ being not in $[0, N)$ is at most $\frac{1}{2}$. Repeating this process λ times ensures that this only happens with probability at most $\frac{1}{2^\lambda}$, i.e., negligible in λ. For a more detailed correctness and security analysis, refer to [7].

Communication cost

Let B be the bitlength of the bound N, and simultaneously the bitlength of the generated number. Let λ be the number of repetitions made.

The first step is to compute the prefix-or of the B bits of N, which takes $\lceil \log_2 B \rceil$ rounds and $O(B \log B)$ total communication. The following for-loop can be executed simultaneously for all i, contributing another $\lceil \log_2 B \rceil + 2$ rounds (1 for the bit-and, and

Protocol 8 Random number generation, following [7, Algorithm 7]

protocol RandInt($[\![N]\!] = [\![N_{B-1}]\!], \ldots, [\![N_0]\!]$)
  Party 1: $a_i \leftarrow \{0,1\}^B$ for $i \in [\lambda]$
  Party 2: $b_i \leftarrow \{0,1\}^B$ for $i \in [\lambda]$
  $[\![c_i]\!] \leftarrow a_i \oplus b_i$
  $[[\![M_{B-1}]\!], \ldots, [\![M_0]\!]] \leftarrow \text{PrefixArray}_\vee([[\![N_{B-1}]\!], \ldots, [\![N_0]\!]])$
  $R \leftarrow []$
  for $i \in [\lambda]$ **do**
    $[\![c_i]\!] \leftarrow [\![c_i]\!] \wedge [\![M]\!]$
    $[\![d_i]\!] \leftarrow [\![c_i]\!] < [\![N]\!]$
    Append $([\![c_i]\!], [\![d_i]\!])$ to R
  end for
  $([\![c]\!], [\![d]\!]) \leftarrow \text{FindFirst}(R)$
  if $[\![d]\!] = [\![0]\!]$ **and** $[\![M_0]\!]$ **then**
    abort
  end if
  return $[\![c]\!]$
end protocol

$\lceil \log_2 B \rceil + 1$ for the comparison) and $O(\lambda \cdot B)$ communication. The call to FindFirst then adds $\lceil \log_2 \lambda \rceil$ rounds and $O(\lambda \cdot B)$ communication.

We assume that the condition $[\![d]\!] = [\![0]\!]$ and $[\![M_0]\!]$ is not checked explicitly. Instead of aborting, we can also let the protocol produce wrong outputs. Alternatively, if this protocol is used as a building block within a larger protocol, it would also be possible to first return $[\![c]\!]$, and then check in the background whether the condition is true. This would not add any extra rounds.

Number of rounds: $2\lceil \log_2 B \rceil + \lceil \log_2 \lambda \rceil + 2$

Total communication: $O(B \cdot (\lambda + \log_2 B))$

7.4 Boolean Matrix Multiplication

In order to test the connectivity of a graph in a low number of communication rounds (see Section 7.5), we need to be able to compute the product of two $n \times n$-matrices consisting of shares of single bits.

Functionality 9 Boolean Matrix Multiplication of two shared matrices of size $n \times n$ each

functionality BMM($[\![A]\!], [\![B]\!]$)
  return $[\![A \cdot B]\!]$
end functionality

The corresponding protocol is defined in a straightforward way: Assume that the shares of matrices A and B are given as $[\![A]\!] = ([\![a_{ij}]\!])_{i,j\in[n]}$ and $[\![B]\!] = ([\![b_{ij}]\!])_{i,j\in[n]}$. Then, the returned matrix $[\![C]\!] = ([\![c_{ij}]\!])_{i,j\in[n]}$ is computed as $[\![c_{ij}]\!] = \bigvee_{k\in[n]} a_{ik} \wedge b_{kj}$.

Communication cost

After computing $a_{ik} \wedge b_{kj}$ simultaneously for all $i, j, k \in [n]$, the protocol needs to combine them into $\bigvee_{k\in[n]} a_{ik} \wedge b_{kj}$. This can be done for all $i, j \in [n]$ simultaneously in $\lceil \log_2 n \rceil$ rounds with exactly $n - 1$ applications of $\wedge$ for all i, j in a tree-like manner by combining $\frac{n}{2}$ pairs of bits in the first round, $\frac{n}{4}$ bits in the second round, and so on.

Number of rounds: $\lceil \log_2 n \rceil + 1$

Total communication: $O(n^3)$

7.5 Connectivity

Using Boolean Matrix Multiplication (BMM, see Section 7.4), we can compute the connectivity $\mathrm{Conn}(C, E)$ for a public partitioning C of vertices $V \subseteq \mathcal{V}$ and a set of edges E split between the two parties in a low number of rounds, in exchange for relatively high communication costs. Functionality 10 contains the desired behavior. Note that as for MSF computation, we require $E^{(1)} \subseteq \mathcal{E}^{(1)}$ and $E^{(2)} \subseteq \mathcal{E}^{(2)}$. In words, a protocol for this functionality is required to split the components from C into sets in such a way that two components are in the same set iff they are reachable from each other on the graph (V, E) w.r.t. C.

Functionality 10 Connectivity between a subset of components

Public: Partitioning C of a finite set of vertices $V \subseteq \mathcal{V}$
functionality CONNECTIVITY($E^{(1)}$, $E^{(2)}$)
  return $\mathrm{Conn}(C, E^{(1)} \cup E^{(2)})$
end functionality

Protocol 11 contains our solution to this problem. The components in C are numbered from 0 to $k-1$, and we denote them by $c_0, \ldots, c_{k-1}$. Then, a secret-shared adjacency matrix $A = (a_{ij})_{i,j\in[k]}$ is created. a_{ij} is 1 iff there is an edge between c_i and c_j (or if $i = j$). Since A^ℓ_{ij} is 1 iff components c_i and c_j are reachable from each other using a path with at most ℓ edges, and two reachable components need to have a path of length $< k$ between each other, we only need to compute A^{k-1}. The resulting matrix tells us for each pair of vertices whether they are reachable from each other.

Protocol 11 Connectivity between a subset of components

Public: Partitioning $C = \{c_0, \ldots, c_{k-1}\}$ of a finite set of vertices $V \subseteq \mathcal{V}$
protocol CONNECTIVITY($E^{(1)}$, $E^{(2)}$)
  for $i, j \in [k]$ with $i \neq j$ **do**
    Party p: $a^{(p)}_{ij} \leftarrow E^{(p)}(c_i, c_j) \neq \emptyset$ (for both $p = 1, 2$)
    $[\![a_{ij}]\!] \leftarrow \text{SHARE}_1(a^{(1)}_{ij}) \vee \text{SHARE}_2(a^{(2)}_{ij})$
  end for
  $[\![a_{ii}]\!] \leftarrow [\![1]\!]$ for all $i \in [k]$
  $[\![A]\!] \leftarrow ([\![a_{ij}]\!])_{i,j\in[k]}$
  for $\lceil \log_2(k-1) \rceil$ times **do**
    $[\![A]\!] \leftarrow \text{BMM}([\![A]\!], [\![A]\!])$
  end for
  $A \leftarrow \text{REVEAL}([\![A]\!])$
  return $\{\{c_j \mid a_{ij} = 1\} \mid i \in [k]\}$ (after removing duplicates)
end protocol

Theorem 4. *Protocol 11 correctly implements Functionality 10.*

Proof. By definition of $[\![A]\!]$, we have $[\![a_{ij}]\!] = [\![1]\!]$ iff there is a direct edge between c_i and c_j, or if $i = j$. This means that $[\![a_{ij}]\!] = [\![1]\!]$ iff there is a path of length ≤ 1 between c_i and c_j. For any $\ell \geq 1$, by using induction and the definition of Boolean Matrix Multiplication, A^ℓ contains a 1 in row i and column j iff there is a path of length $\leq \ell$ between c_i and c_j.

By the repeated squaring method, after the second for-loop, we have $[\![a_{ij}]\!] = [\![1]\!]$ iff there is a path of length $2^{\lceil \log_2(k-1) \rceil} \geq k-1$ between c_i and c_j, which happens iff there is any path at all between c_i and c_j.

Thus, A divides all components into equivalence classes of components reachable from each other. The return value of Protocol 11 contains for every c_i the set $\{c_j \mid a_{ij} = 1\}$ of components connected to c_i. We remove any duplicate sets from $\{\{c_j \mid a_{ij} = 1\} \mid i \in [k]\}$

(each equivalence class would be added once for every component in it), and therefore this is by definition of $\mathrm{Conn}(C, E^{(1)} \cup E^{(2)})$ the required output. □

Theorem 5. *Protocol 11 privately computes Functionality 10 in the semi-honest security model.*

Proof. Everything up to the $\textsc{Reveal}(\cdot)$-step is computed on secret-shared bits and therefore secure. Thus, it suffices to show that given the output $\{d_0, \ldots, d_{\ell-1}\}$ of the protocol, we can reconstruct the revealed matrix A.

As shown in Theorem 4, after revealing, matrix $A = (a_{ij})_{i,j\in[k]}$ has $a_{ij} = 1$ iff there is a path on E w.r.t. C between c_i and c_j. By definition of $\mathrm{Conn}(C, E)$, we know that c_i and c_j are in the same set d_r iff they are connected to each other, i.e. there is a path w.r.t. C between them. Thus, a simulator can perfectly reconstruct A by setting $a_{ij} = 1$ iff c_i and c_j are in the same set d_r. □

Communication cost

The protocol consists of one share step, one step in which a bit-or is computed for all $i, j \in [n]$ in order to find a_{ij}, $\lceil \log_2(k-1) \rceil$ applications of BMM, and one reveal step.

Number of rounds: $\lceil \log_2(k-1) \rceil \cdot (\lceil \log_2 k \rceil + 1) + 3$

Total communication: $O(k^3 \log k)$

Chapter 8

Protocol for unweighted graphs

The protocol presented in this chapter computes a random spanning forest (following Functionality 1) on an unweighted graph (or, alternatively, a graph where every edge has the same weight).

The main purpose of this protocol is to use it as a building block for the full protocol (see Chapter 9) on very small subgraphs (more specifically, whenever the computed MSF contains components connected by edges of the same weight). However, the number of edges in such a subgraph can be arbitrarily high, and therefore the protocol in this chapter is designed in such a way that it does not depend on this number (as long as it fits into the integers of a previously fixed bitlength). The number of edges that a party owns should not be leaked to the other party. This is the reason that for every pair of vertices and the index of a party, we only keep track of the number of edges between this pair of vertices that belong to the given party. From the indices of its chosen edges, each party is able to select the corresponding edge to add it to the output.

We loosely follow Kruskal's algorithm [23] for computing a random spanning forest. As a reminder, this algorithm proceeds as follows: It iterates through all edges in order of increasing weight, and adds one of them to the minimum spanning forest if the two endpoints are not already connected by edges that were previously added. Checking this condition is usually done with a union-find data structure, by merging vertices already connected together to one component. Such a union-find data structure can be implemented by choosing a root vertex for each component, and letting every vertex in the component point to the root vertex. The difference between Kruskal's algorithm and our protocol is that in our case, all edges have the same weight, and so we simply select a random edge that we add to the spanning forest and repeat this process for $n - 1$ times.

Protocol 12 shows a formal description of our procedure. We assume that the components in the initial partitioning $C = \{c_0, \ldots, c_{n-1}\}$ are numbered from 0 to $n-1$, and that there is a weight w^* s.t. $\mathbf{w}(e) = w^*$ for every $e \in E^{(1)} \cup E^{(2)}$ in the input. The goal is to compute $\mathsf{MSF}(C, E^{(1)} \cup E^{(2)})$.

Over the course of the protocol, components will be merged together if an edge between them has been selected and added to the MSF. For this reason, we store a union-find data structure: for every $i \in [n]$, we define u_i to be the index of the root component that c_i belongs to. That is, in the beginning, we have $u_i = i$ for all $i \in [n]$, and when we merge along an edge connecting c_i and c_j, we set the new value of u_k to u_i for every $k \in [n]$ with $u_k = u_j$.

For every pair of components (c_i, c_j) $(0 \leq i < j < n)$ and for every party $p = 1, 2$, we want to store two secret-shares: (1) the number of edges $|E^{(p)}(c_i, c_j)|$ that party p has that connect a vertex from c_i with a vertex from c_j, and (2) the index of the edge within $E^{(p)}(c_i, c_j)$ that has been added to the MSF (if there is any at all). To simplify the protocol, we "flatten" these arrays by defining $d(i, j, p) = (p-1) \cdot \frac{n(n-1)}{2} + \frac{j \cdot (j-1)}{2} + i$. Note that $d(i, j, p)$ is a bijection, i.e., for every pair $0 \leq i < j < n$ and every $p = 1, 2$ it assigns a different index between 0 and $N - 1$, where $N := n \cdot (n-1)$ is the total length of the flattened array (in other words, N is the number of tuples (i, j, p) denoting two different components $0 \leq i < j < n$ and a party $p = 1, 2$).

We want $a_{d(i,j,p)}$ to be the the number of edges $|E^{(p)}(c_i, c_j)|$ that party p has that connect a vertex from c_i with a vertex from c_j. Thus, at the beginning of the protocol, we secret-share $a_{d(i,j,p)} := |E^{(p)}(c_i, c_j)|$. As soon as c_i and c_j are connected through edges selected for the MSF (i.e., c_i and c_j are merged together, meaning that $u_i = u_j$), we set $a_{d(i,j,p)} \leftarrow 0$, in order to make sure that no edges between c_i and c_j can be selected anymore.

We let $s_{d(i,j,p)}$ be the index of the edge within $E^{(p)}(c_i, c_j)$ that has been added to the MSF. In the beginning, we choose $s_{d(i,j,p)} := \perp$ to indicate that no edge from $E^{(p)}(c_i, c_j)$ has been added to the MSF yet.

As in Kruskal's algorithm, we add an edge to our spanning forest in the *main loop* (line 9) at most $n-1$ times. To do this, we compute the prefix sum $A_z = a_0 + \cdots + a_z$ of the a_z's. Then, $A_{N-1} = a_0 + \cdots + a_{N-1}$ contains the total number of edges for which the two endpoints have not been merged yet into the same component. We choose a random number $r \in [A_{N-1}]$ that corresponds to the edge that we are selecting in this iteration. This can be done using the RandInt protocol (line 11). Then, we want to find the edge that this r corresponds to, i.e., the $\tilde{z} = (\tilde{i}, \tilde{j}, \tilde{p})$ for which $A_{\tilde{z}-1} \leq r < A_{\tilde{z}}$. This is equivalent to the first $\tilde{z}$ fulfilling $r < A_{\tilde{z}}$. Thus, we can call the sub-protocol FindFirst to find the first z for which $g_z := A_z > r$ is true (line 13).

Protocol 12 Random Spanning Forest for an unweighted graph

Public: Partitioning $C = \{c_0, \ldots, c_{n-1}\}$ of a finite set of vertices $V \subseteq \mathcal{V}$
Precondition: There is a $w^* \in \mathbb{N}$ with $\mathbf{w}(e) = w^*$ for every $e \in E^{(1)} \cup E^{(2)}$
protocol UNWEIGHTEDSF($E^{(1)}$, $E^{(2)}$)
    $[\![u_i]\!] \leftarrow i \quad \forall i \in [n]$
    for $0 \le i < j < n$ and $p = 1, 2$ **do**
        $[\![a_{d(i,j,p)}]\!] \leftarrow \text{SHARE}_p(|E^{(p)}(c_i, c_j)|)$
        $[\![s_{d(i,j,p)}]\!] \leftarrow \perp$
    end for
    for $n - 1$ times **do**
        $[[\![A_0]\!], \ldots, [\![A_{N-1}]\!]] \leftarrow \text{PREFIXARRAY}_+([[\![a_0]\!], \ldots, [\![a_{N-1}]\!]])$
        $[\![r]\!] \leftarrow \text{RANDINT}([\![A_{N-1}]\!])$
        $[\![g_z]\!] \leftarrow [\![A_z]\!] > [\![r]\!] \quad \forall\, z \in [N]$
        $(([\![\tilde{u}]\!], [\![\tilde{v}]\!]), [\![\tilde{g}]\!]) \leftarrow \text{FINDFIRST}([([\![u_i]\!], [\![u_j]\!]), [\![g_z]\!]]_{d(i,j,p)=z \in [N]})$
        if $\text{REVEAL}(\tilde{g})$ **is false then break end if**
        for $0 \le i < n$ **do**
            $[\![u_i]\!] \leftarrow$ **if** $[\![u_i]\!] = [\![\tilde{v}]\!]$ **then** $[\![\tilde{u}]\!]$ **else** $[\![u_i]\!]$
        end for
        for $d(i, j, p) = z \in [N]$ **do**
            $[\![a_z]\!] \leftarrow$ **if** $[\![u_i]\!] = [\![u_j]\!]$ **then** 0 **else** $[\![a_z]\!]$
            $[\![s_z]\!] \leftarrow$ **if** $[\![g_z]\!] \wedge \neg[\![g_{z-1}]\!]$ **then** $[\![r]\!] - [\![A_{z-1}]\!]$ **else** $[\![s_z]\!]$ $\quad \triangleright\ g_{-1} = 0,\ A_{-1} = 0$
        end for
    end for
    $F \leftarrow \emptyset$
    for $0 \le i < j < n$ and $p = 1, 2$ **do**
        Send $\text{REVEAL}_p(s_{d(i,j,p)})$ to party p
        if party p determines that $s_{d(i,j,p)} \neq \perp$ **then**
            Party p: $e \leftarrow s_{d(i,j,p)}$-th element from $E_{ij}^{(p)}$
            Party p sends e to the other party
            $F \leftarrow F \cup \{e\}$
        end if
    end for
    return F
end protocol

Next, all u_i with $u_i = u_{\tilde{j}}$ are updated to $u_{\tilde{i}}$ (line 16). This makes sure that for all i, j where c_i are merged together, we have $u_i = u_j$. In addition, for all merged components c_i, c_j, we set $a_{d(i,j,1)}$ and $a_{d(i,j,2)}$ to 0 (line 19). Finally, $s_{\tilde{z}}$ is updated to $r - A_{\tilde{z}-1}$ (line 20). This is the value that corresponds to the index of the selected edge selected from $E^{(\tilde{p})}(c_{\tilde{i}}, c_{\tilde{j}})$.

At the end of the protocol (line 24), $s_{d(i,j,p)}$ is revealed to party p for all i and j. If p notices that one of their edges between c_i and c_j was selected ($s_{d(i,j,p)} \neq \perp$), they send the corresponding edge to the other party. The final output is the set of all selected edges.

8.1 Correctness and Security

The correctness is based on the following lemma, which states that a random MSF can be found as done in Protocol 12: selecting a uniformly random edge between two unmerged components, and then continuing to find an MSF on the partitioning where the two endpoints have been merged.

Lemma 2. *Let $G = (V, E)$ be an unweighted graph (i.e., there is a weight $w^* \in \mathbb{N}$ s.t. $\mathbf{w}(e) = w^*$ for all $e \in E$) and C a partitioning of V s.t. $E(C) \neq \emptyset$. Then, choosing a uniformly random edge e from $E(C)$ and returning $\{e\} \cup \mathsf{MSF}(C[e], E)$ has the same output distribution as $\mathsf{MSF}(C, E)$.*

Proof. Note that the described procedure is equivalent to choosing a uniformly random $e \in E(C)$, a uniformly random $\pi \in E!$, and returning $\{e\} \cup \mathsf{MSF}(C[e], E, \pi)$. In addition, the return value of $\mathsf{MSF}(C, E)$ is equivalent to selecting a uniformly random $\tau \in E!$ and returning $\mathsf{MSF}(C, E, \tau)$.

Note that there are $|E(C)| \cdot |E|!$ possible ways for choosing a pair (e, π), and there are $|E|!$ possible ways for choosing a permutation τ. Thus, in order to prove this lemma, it suffices to assign $|E(C)|$ different pairs (e, π) to every fixed permutation τ (with each (e, π) being assigned to only one τ) s.t. the two outputs $\{e\} \cup \mathsf{MSF}(C[e], E, \pi)$ and $\mathsf{MSF}(C, E, \tau)$ are equal.

For a fixed pair $(e, \pi) \in E(C) \times E!$, we can construct a permutation $\tau \in E!$ s.t. e becomes the best edge among all $E(C)$ according to τ, and the positions of the other edges from $E(C)$ are changed in such a way that they retain their relative ordering:

(a) e is the best edge from $E(C)$ according to τ: $e = \arg\min_{e' \in E(C)} \tau(e')$,

(b) all other edges in $E(C)$ maintain their ordering: $\pi(e') < \pi(e'') \Leftrightarrow \tau(e') < \tau(e'')$ for all $e', e'' \in E(C) \setminus \{e\}$, and

(c) all edges that are not in $E(C)$ maintain their assigned index in the permutation: $\pi(e') = \tau(e')$ for all $e' \notin E(C)$.

Note that this defines a unique τ for any given pair (e, π), and every τ is constructed from exactly $|E(C)|$ different pairs. Thus, it suffices to show that for a pair $(e, \pi) \in E(C) \times E!$ and corresponding $\tau \in E!$, the following two MSFs are equal:

$$M := \mathsf{MSF}(C[e], E, \pi) \cup \{e\} = \mathsf{MSF}(C, E, \tau) =: F.$$

Note that M is a spanning forest of G on partitioning C.

Assume that this equality does not hold, and let e' be the best edge (w.r.t. τ) that is contained in one of F and M, but not both. There are two cases:

(1) $e' \in F$, but $e' \notin M$. We can add e' to M in order to obtain M'. If this does not create a cycle on C, M was not a spanning forest. If it does create a cycle, all edges in it are better than e' w.r.t. π (otherwise M was not minimum).

We are going to show that any edge $e'' \neq e'$ on the cycle is also contained in F, which proves that F also contains a cycle on C, a contradiction to the fact that F is a spanning forest. To do this, it suffices to show that $\tau(e'') < \tau(e')$. Then, because e' is the best edge w.r.t. τ on which F and M differ, e'' must also be contained in F.

Note that e'' must be in $E(C)$, otherwise it could be removed from M, and the result would still be spanning. In addition, e' must be in $E(C)$, otherwise it could be removed from F, and the result would still be spanning.

We also know that $e' \neq e$ (because of $e' \notin M$). There are two cases:

- $e'' = e$. Then, by condition (a), we have $\tau(e'') < \tau(e')$.
- $e'' \neq e$. Then, by condition (b), $\tau(e'') < \tau(e')$ follows from $\pi(e'') < \pi(e')$ (since by assumption, all edges in the cycle, including e'', are better than e' w.r.t. π).

(2) $e' \in M$, but $e' \notin F$. We can add e' to F in order to obtain F'. If this does not create a cycle on C, F was not a spanning forest. If it does create a cycle, all edges in it must better than e' w.r.t. τ (otherwise F was not minimum). Thus, since e' is the best edge w.r.t. τ on which F and M differ, the whole cycle is contained in M as well, which contradicts the fact that M is a spanning tree.

□

Using this lemma, we can now prove the correctness of Protocol 12.

Theorem 6. *Protocol 12 correctly implements Functionality 1 if all edges in the input have the same weight.*

Proof. Note that before and after each iteration of the main loop, there is a current partitioning D of V, which is at least as coarse as C (i.e., for every $c \in C$ there is a $d \in D$ with $c \subseteq d$) fulfilling the following requirements:

(1) For every $d \in D$, there is a "root" component $c_i \subseteq d$, s.t. $u_j = i$ for all $c_j \subseteq d$.

(2) For any $0 \leq i < j < n$ with $u_i = u_j$ (i.e., c_i and c_j have been merged) we have $a_{d(i,j,1)} = a_{d(i,j,2)} = 0$. If $u_i \neq u_j$, then we have $a_{d(i,j,p)} = |E^{(p)}(c_i, c_j)|$ for $p = 1, 2$. In other words, $a_{d(i,j,p)}$ is initially the number of edges that party p owns between components c_i and c_j, but it is set to 0 as soon as components c_i and c_j are merged together.

These requirements trivially hold before the first iteration of the main loop by choosing $D := C$. The root of a component c_i is $u_i = i$ itself. During an iteration, if the protocol selects an edge e, then the new partitioning will be $D' := D[e]$.

Now we are going to show the correctness by induction in the opposite direction as how the protocol proceeds. We show that the following invariant holds after the last iteration of the for-loop, and that it must also hold if we go back one iteration: the protocol is going to output all edges $E^{(p)}(c_i, c_j)[s_{d(i,j,p)}]$ (i.e., the $s_{d(i,j,p)}$-th edge in $E^{(p)}(c_i, c_j)$ according to an arbitrary ordering) for which $s_{d(i,j,p)} \neq \perp$, combined with $\mathsf{MSF}(D, E)$, a random MSF on the remainder of the graph. In other words, the output of the protocol will be

$$\{E^{(p)}(c_i, c_j)[s_{d(i,j,p)}] \mid 0 \leq i < j < n, p = 1, 2 \text{ with } s_{d(i,j,p)} \neq \perp\} \cup \mathsf{MSF}(D, E).$$

This trivially holds after the last iteration of the main loop, because either the loop terminated due to $\tilde{g} = 0$ (which means that there are no edges remaining in $E(D)$), or $n - 1$ iterations have taken place (as exactly one edge has been selected in each iteration, all components must have been merged already). Thus, $\mathsf{MSF}(D, E) = \emptyset$.

Now, for the induction step, let $s_{d(i,j,p)}$ and D be the selected indices and the partitioning *before* the current iteration, and let $s'_{d(i,j,p)}$ and D' be the selected indices and the new partitioning *after* the current iteration. By induction hypothesis, we know that after the current iteration, the actual output will be

$$\{E^{(p)}(c_i, c_j)[s'_{d(i,j,p)}] \mid 0 \leq i < j < n, p = 1, 2 \text{ with } s'_{d(i,j,p)} \neq \perp\} \cup \mathsf{MSF}(D', E),$$

and we need to show that including the current iteration, the protocol will output

$$\{E^{(p)}(c_i, c_j)[s_{d(i,j,p)}] \mid 0 \leq i < j < n, p = 1, 2 \text{ with } s_{d(i,j,p)} \neq \perp\} \cup \mathsf{MSF}(D, E).$$

As the protocol chooses a uniformly random $r \in [a_0 + \cdots + a_{N-1}]$ that corresponds to an edge $e \in E^{(\tilde{p})}(c_{\tilde{i}}, c_{\tilde{j}})$, we may use Lemma 2 on partitioning D to see that $\mathsf{MSF}(D, E) = \{e\} \cup \mathsf{MSF}(D', E)$ (i.e., we can first sample the randomly selected edge e, and then continue to calculate an MSF on the graph after we merged along e). In addition, $s'_{d(\tilde{i},\tilde{j},\tilde{p})}$ is set to the index in $E^{(\tilde{p})}(c_{\tilde{i}}, c_{\tilde{j}})$ corresponding to e, and $s'_{d(i,j,p)}$ stays the same as before (i.e., $s'_{d(i,j,p)} = s_{d(i,j,p)}$) for all $(i, j, p) \neq (\tilde{i}, \tilde{j}, \tilde{p})$. In conclusion,

$$\begin{aligned}
&\{E^{(p)}(c_i, c_j)[s'_{d(i,j,p)}] \mid 0 \leq i < j < n, p = 1, 2 \text{ with } s'_{d(i,j,p)} \neq \perp\} \cup \mathsf{MSF}(D', E)) \\
&= \{E^{(p)}(c_i, c_j)[s_{d(i,j,p)}] \mid 0 \leq i < j < n, p = 1, 2 \text{ with } s_{d(i,j,p)} \neq \perp\} \cup \{e\} \cup \mathsf{MSF}(D', E) \\
&= \{E^{(p)}(c_i, c_j)[s_{d(i,j,p)}] \mid 0 \leq i < j < n, p = 1, 2 \text{ with } s_{d(i,j,p)} \neq \perp\} \cup \mathsf{MSF}(D, E).
\end{aligned}$$

By induction, before running the first iteration of the main loop, we know that the output will simply be $\mathsf{MSF}(C, E)$. This is because $s_{d(i,j,p)} = \perp$ for all $0 \leq i < j < n$ and $p = 1, 2$, and initially we have $D = C$. □

Theorem 7. *Protocol 12 privately computes Functionality 1 in the semi-honest security model if all edges in the input have the same weight.*

Proof. As most computations happen on secret-shared values, the security of this protocol is very simple. It suffices to prove that there is a simulator that is able to efficiently reconstruct all values that are revealed during the protocol.

The only values revealed are $\tilde{g}$ to check whether the main loop can be aborted, the value of $s_{d(i,j,p)}$ (which is only revealed to party p), and all the edges that are sent and added to F.

- Given the final protocol output F, every party knows that exactly $|F|$ full iterations have taken place. This is because every iteration (that is not aborted due to $\tilde{g} = 0$) adds exactly one edge to the MSF. In all iterations where an edge has been selected we have $\tilde{g} = 1$, and in an aborted iteration we have $\tilde{g} = 0$.
- Let $0 \leq i < j < n$ and $p = 1, 2$. Then, the simulator of party p can check whether there is a an edge $e \in F$ belonging to party p that connects components c_i and c_j. If this is not the case, then $s_{d(i,j,p)}$ must have been $\perp$. Otherwise, the simulator can simply search for the unique edge $e' \in E^{(p)}(c_i, c_j)$ with $\mathbf{r}(e') = \mathbf{r}(e)$ and output its index w.r.t. $E^{(p)}(c_i, c_j)$.

- Any edges that are sent and added to F are already contained in the output F themselves. Thus, there is no need for the simulator to recompute them.

□

Communication cost

The dependencies of the secret-shared values in Protocol 12 on each other are more difficult to analyze than in the previous protocols. However, it is still possible to find an exact formula for the number of rounds in a binary-secret-shared setting. Let $n := |C|$ be the number of components in the input, $N := n \cdot (n-1)$ the number of triples (i, j, p), B the bitlength of values containing numbers of edges (i.e., the total number of edges needs to be bounded by 2^B), and λ the number of repetitions made in every call to RandInt. Note that the bitlength of the u_i's can be set to $\lceil \log_2 n \rceil$, as this is enough to hold all possible indices from 0 to $n-1$, i.e., indices for the n initial components.

In the beginning, 1 round is required for sharing the a_z's. We assume that all $n-1$ iterations of the main loop are executed, i.e., no early termination occurs. Then, in every iteration, the following secret-shares need to be computed in sequential order, because each of them depends on the previous one:

(1) $[\![A_i]\!]$ for $i \in [N]$. This is computed using PrefixArray$_+$, which takes $\lceil \log_2 N \rceil \cdot (\lceil \log_2 B \rceil + 1)$ rounds.

(2) $[\![r]\!]$, which requires a call to RandInt with $2\lceil \log_2 B \rceil + \lceil \log_2 \lambda \rceil + 2$ rounds.

(3) g_z for all $z \in [N]$. This can be done in parallel for all z, so all together it takes only $\lceil \log_2 B \rceil + 1$ rounds.

(4) $(([\![\tilde{u}]\!], [\![\tilde{v}]\!]), [\![\tilde{g}]\!])$ is computed using FindFirst, taking $\lceil \log_2 N \rceil$ rounds.

(5) For every $i \in [n]$, the protocol computes whether $[\![u_i]\!] = [\![\tilde{v}]\!]$, which takes $\lceil \log_2 \lceil \log_2 n \rceil \rceil$ rounds.

(6) Then, for every $i \in [n]$, a multiplexer is used to compute the new value of $[\![u_i]\!]$, taking one round.

(7) For every $d(i, j, p) = z \in [N]$, the protocol computes (in parallel) whether $[\![u_i]\!] = [\![u_j]\!]$. This takes $\lceil \log_2 \lceil \log_2 n \rceil \rceil$ rounds.

(8) Then, for every $d(i,j,p) = z \in [N]$, the new value of $[\![a_z]\!]$ is constructed by using a multiplexer, which contributes another round.

Note that we ignored the rounds required for computing s_z here. This is because it can already be started as soon as step (3) is finished. s_z requires 1 round for computing $[\![g_z]\!] \wedge \neg[\![g_{z-1}]\!]$, which can be done in parallel to $[\![r]\!] - [\![A_{z-1}]\!]$, which takes $\lceil \log_2 B \rceil + 1$ rounds. Then, the multiplexer for assigning the new value of s_z adds another round for a total of $\lceil \log_2 B \rceil + 2$. This is less than one full computation of steps (1)-(8).

The protocol can also be optimized using the following observation: in the last iteration (the $(n-1)$-th one), it is not necessary to actually perform steps (4)-(8), as afterwards u_i and a_z are not used anymore. However, the steps for computing s_z (and its dependencies (1)-(3)) still need to be performed in the last iteration. This means that in the main loop, in total we need to run $n-1$ times steps (1)-(3), $n-2$ times steps (4)-(8), and one additional time $\lceil \log_2 B \rceil + 2$ for computing s_z in the last iteration. Before the main loop, we need one round for sharing the values of a_z, and after the main loop, we need to add two more rounds for revealing the s_z's, and then sending the corresponding edges in plaintext.

The total number of multiplications is dominated by $\textsc{PrefixArray}_+$ (with $O(N \cdot \log N \cdot B \cdot \log B)$ multiplications per iteration) and $\textsc{RandInt}$ (with $O(B \cdot (\lambda + \log B))$ multiplications per iteration). In our upper bound, we replace N by n^2.

Number of rounds:

$$\begin{aligned}
&(3n-2)\lceil \log_2 B \rceil \\
+&(2n-3)\lceil \log_2 N \rceil \\
+&(2n-4)\lceil \log_2 \lceil \log_2 n \rceil \rceil \\
&+(n-1)\lceil \log_2 N \rceil \cdot \lceil \log_2 B \rceil \\
&+(n-1)\lceil \log_2 \lambda \rceil \\
&+5n-2
\end{aligned}$$

Total communication: $O(n \cdot B \cdot (\lambda + \log B \cdot n^2 \log n))$

8.2 Optimization for the case $|C| = 2$

If C consists of only two components $C = \{c, c'\}$, then it is unnecessary to run the full Protocol 12, as it consists of many steps that are redundant in this case. This is because the two parties only need to select a random edge from $(E^{(1)} \cup E^{(2)})(C)$. We can do this

with the following more general functionality RandomSelection (Functionality 13) that selects a uniformly random element from a list L_1 owned by party 1, and a list L_2 owned by party 2, where $|L_1| + |L_2|$ is guaranteed to be greater than 0.

Functionality 13 Random selection

functionality RandomSelection(L_1, L_2)
  r is selected uniformly at random from $[|L_1| + |L_2|]$
  if $r < |L_1|$ **then**
    return $(L_1[r], 1)$
  else
    return $(L_2[r - |L_1|], 2)$
  end if
end functionality

In this scenario, each party has full knowledge of its own list, but no information about the other party's list. In addition, we assume that none of the two lists L_1 and L_2 contains any duplicate elements (note that this is fulfilled for the use case described above, as no party is allowed to have two edges with the exact same endpoints and the same weight). The randomly selected element, together with its owner (i.e., whether it belongs to L_1 or L_2) is revealed to both parties. Note that this functionality is not the same as choosing a uniformly random $i \in \{1, 2\}$, and then a uniformly random element from L_i.

We implement this behavior by using RandInt (Protocol 8) to select a random number $r \in [|L_1| + |L_2|]$ (this bound $|L_1| + |L_2|$ needs to be secret, since otherwise a party would learn the length of the other party's list). Whether r corresponds to L_1 (i.e., $r < |L_1|$) or L_2 (i.e., $r \geq |L_1|$) can be revealed to everyone, as it will be part of the output in any case. Then, the corresponding party receives the index in its own list in plaintext, and sends the element to the other party. This is shown in Protocol 14.

The correctness of Protocol 14 is immediate, as it follows the steps in Functionality 13. However, some values are revealed in the middle of the protocol execution, making it necessary to prove the semi-honest security:

Theorem 8. *Protocol 14 privately computes Functionality 13 in the semi-honest security model.*

Proof. There are two paths that the protocol can take, depending on whether $r < a$ or $r \geq a$. Which of these two paths is taken can be inferred easily from the protocol output, as it reveals whether the given element was provided by party 1 or by party 2.

Protocol 14 Random selection

protocol RANDOMSELECTION(L_1, L_2)
  $[\![a]\!] \leftarrow \text{SHARE}_1(|L_1|), [\![b]\!] \leftarrow \text{SHARE}_2(|L_2|)$
  $[\![r]\!] \leftarrow \text{RANDINT}([\![a]\!] + [\![b]\!])$
  if $\text{REVEAL}([\![r]\!] < [\![a]\!])$ **then**
    $\text{REVEAL}_1([\![r]\!])$ is sent to **party 1**
    Party 1 sends $L_1[r]$ to **party 2**
    Both parties: return $(L_1[r], 1)$
  else
    $\text{REVEAL}_2([\![r]\!] - [\![a]\!])$ is sent to **party 2**; denote its value by r'
    Party 2 sends $L_2[r']$ to **party 1**
    Both parties: return $(L_2[r'], 2)$
  end if
end protocol

For simplicity, we only look at the first path, as the second one is analogous. The simulator for party 1 receive the input L_1 and the protocol output $(d, 1)$, where d is an element from L_1. Now, the simulator knows exactly what the value of the revealed index r is: the unique $i \in [|L_1|]$ with $d = L_1[i]$.

The simulator for party 2 is even simpler, as the only values revealed to party 2 are the fact that the first path was taken, and the output $(d, 1)$ itself. □

Communication cost

Let B be the bitlength used for the shares $[\![a]\!]$ and $[\![b]\!]$. It defines an upper bound (of 2^B) for the size of $|L_1| + |L_2|$.

In the case of $r < a$, the number of rounds consists of 1 for sharing $|L_1|$ and $|L_2|$, $2\lceil \log_2 B \rceil + \lceil \log_2 \lambda \rceil + 2$ for the RANDINT, $\lceil \log_2 B \rceil + 1$ for the comparison $[\![r]\!] < [\![a]\!]$, 1 for revealing r to party 1, and 1 for party 1 to send the result to party 2. In the case of $r \geq a$, we would additionally take $\lceil \log_2 B \rceil + 2$ rounds for the computation of $[\![r]\!] - [\![a]\!]$. However, this value can already be computed while performing the comparison $[\![r]\!] < [\![a]\!]$. Then, $[\![r]\!] - [\![a]\!]$ is simply discarded if it turns out that $r < a$, in which case some unnecessary communication took place, without increasing the number of rounds. Still, computing $[\![r]\!] - [\![a]\!]$ takes one round more than testing $[\![r]\!] < [\![a]\!]$.

The total communication is dominated by RANDINT, which requires $O(B \cdot (\lambda + \log_2 B))$ multiplications. Note that it might be possible that sending one element (e.g. $L_1[r]$) requires

sending an even larger amount of bits than RandInt. However, this is not the case for any of our use cases, and furthermore no costly multiplication triples would be required as the value $L_1[r]$ can be sent in plaintext.

Number of rounds: $3\lceil \log_2 B \rceil + \lceil \log_2 \lambda \rceil + 7$

Total communication: $O(B \cdot (\lambda + \log_2 B))$

Now, if RandomSelection (Protocol 14) is used instead of Protocol 12 for the case of only two components $C = \{c, c'\}$, then we save $2\lceil \log_2 B \rceil + 2$ communication rounds. The full Protocol 12 would require $5\lceil \log_2 B \rceil + \lceil \log_2 \lambda \rceil + 9$ rounds.

Chapter 9

General Protocol

We have seen that a plain implementation of Borůvka's algorithm as a multi-party protocol may fail to work correctly if there are edges with the same weight while no tie-breaker is deployed. In such a case there may be multiple minimum spanning forests, and the parties need to commit to a strategy for choosing which MSF to output before running the protocol. One natural and useful tie-breaker would be to choose a *random* minimum spanning forest, in the sense that (1) an ordering of all edges is selected uniformly at random, and (2) ties between edges of the same weight are broken by considering this ordering. This is exactly our definition of the randomized function $\mathsf{MSF}(C, E)$.

However, if the parties are able to gain knowledge about this ordering through their view of the protocol, this can result in leaked information (Chapter 6). This is the reason that Protocol 3 is not secure when the two parties use a public random tie-breaker. An alternative would be to use a fully secure protocol without any intermediate steps of revealing edges, as it has been done in most prior work, e.g. by Laud [25]. As this usually results in significantly higher communication costs, we present an optimized protocol in this chapter which reveals information whenever it is possible, while falling back to a fully secure MSF protocol when necessary. The protocol is carefully crafted in such a way that it still fulfills semi-honest security. It works the fastest if there is a very low number of edges with the same weight. Thus, if the parties are unsure whether they share edges of the same weight, or if they know that there are only a few of them, this protocol can be much better than protocols without any intermediate steps of revealing information, as shown in Chapter 10.

This chapter is organized as follows: Section 9.1 first gives some Lemmas that are required to see why the protocol works. Section 9.2 then gives an overview and intuition

about the full protocol. Section 9.3 contains the formal pseudocode (Protocol 15). Its correctness and security are proven in Sections 9.4 and 9.5, respectively.

9.1 Crucial observations

First, we are going to prove a simple and helpful lemma that allows us to disregard any irrelevant edges (Lemma 3). Then, our main observation (Corollary 2) will be a combination of Lemma 4 and Lemma 5. In addition, we also obtain Corollary 1, which we will use to prove a termination condition of Protocol 15.

We start with proving that whenever $\mathsf{MSF}(C, E)$ on a partitioning C is computed, any edge connecting two vertices within the same component can be disregarded. In other words, the random process of $\mathsf{MSF}(C, E)$ is the same as $\mathsf{MSF}(C, E(C))$ where we restrict ourselves to the edge set $E(C)$.

Lemma 3. *Let $G = (V, E)$ be a graph and $C = \{c_0, \dots, c_{k-1}\}$ a partitioning of V. Then, every MSF of (V, E) on components C only contains edges from $E(C)$ (i.e., edges that connect two different components). More specifically, the following two random processes are equal.*

$$\mathit{MSF}(C, E) = \mathit{MSF}(C, E(C)).$$

Proof. For any permutation $\pi : E \to [|E|]$ of E, we have a corresponding permutation $\pi|_{E(C)}$ of $E(C)$. Since every permutation of $E(C)$ has exactly $\frac{|E|!}{|E(C)|!}$ corresponding permutations of E, it suffices if we prove

$$\mathsf{MSF}(C, E, \pi) = \mathsf{MSF}(C, E(C), \pi|_{E(C)})$$

for every permutation $\pi : E \to [|E|]$.

Fix any such π and assume that this equality does not hold. Abbreviate $\mathsf{MSF}(C, E, \pi)$ by F and $\mathsf{MSF}(C, E(C), \pi|_{E(C)})$ by M. Let $e \in E$ be the best edge (w.r.t. $\mathbf{w}$, or secondarily π) that is in either F or M but not in the other one. There are two cases:

- $e \in F$ and $e \notin M$.

  If $e \notin E(C)$, we could just remove e from F to receive $F' := F \setminus \{e\}$. Note that all components $c, c' \in C$ are still connected w.r.t. F' if they were connected w.r.t. F, since both endpoints of e are in the same component. Thus, F is not a forest, which is a contradiction.

If $e \in E(C)$, then we can add it to M to receive $M' := M \cup \{e\}$. If this doesn't result in a cycle w.r.t. C, M was not a minimum *spanning* forest, a contradiction. If it does, then all of the other edges in the cycle are better than e (otherwise, adding e and removing an edge on the cycle that is worse than e would result in an MSF that is better than M) and therefore also contained in F (as e is the best edge on which F and M differ). Thus, the MSF F contains a cycle, a contradiction.

- $e \in M$ and $e \notin F$.

  We can add e to F to obtain $F' := F \cup \{e\}$. This must result in a cycle w.r.t. C, as otherwise F was not a spanning forest. All of the other edges in the cycle are better than e, and therefore also contained in M. Thus, the MSF M contains a cycle, a contradiction.

□

The following Lemma shows that if a graph can be split into components, s.t. each edge between two different components has a larger weight than any edge required for an MSF within a single component, then the full MSF can be split into MSF's for the components.

Lemma 4. *Let $G = (V, E)$ be a graph and $C = \{c_0, \dots, c_{k-1}\}$ a partitioning of V. Assume that there is a weight $w^* \in \mathbb{N}$ s.t. the following conditions are satisfied for every $i \in [k]$:*

(i) all vertices within c_i are connected on the graph $(V, E_{<w^*})$ (i.e., they connected to each other even when considering only edges of weight $< w^*$), and

(ii) $\min\{\mathbf{w}(e) \mid e \in E(c_i)\} \geq w^*$ (i.e., the best outgoing edge of c_i has weight $\geq w^*$).

Then, sampling F_i as $\mathsf{MSF}(c_i, E_{c_i})$ for all $i \in [k]$, sampling F as $\mathsf{MSF}(C, E)$ (all of these happening independently of each other), and returning $F_0 \cup \dots \cup F_{k-1} \cup F$ has the same output distribution as $\mathsf{MSF}(V, E)$.

Proof. By Lemma 3, F is sampled according to $\mathsf{MSF}(C, E(C))$. Also note that for any MSF $M := \mathsf{MSF}(V, E)$ and $i \in [k]$, the restricted set of edges M_{c_i} that only contains edges connecting vertices from c_i is still a spanning forest on c_i. If this were not true, then we could add an arbitrary edge e of weight $< w^*$ between two vertices in c_i that are not connected using M_{c_i} to M to obtain $M' := M \cup \{e\}$. This must result in a cycle on V (otherwise M was not spanning), and it must contain at least one edge e' connecting two different components in C, which has weight $\geq w^*$ by assumption (ii).

Thus, $M'' := M' \cup \{e'\}$ is also a spanning forest, but its total weight $\mathbf{w}(M'') < \mathbf{w}(M)$ is better than that of M, a contradiction to M being minimal.

Note that $E = E_{c_0} \sqcup \cdots \sqcup E_{c_{k-1}} \sqcup E(C)$, i.e., the edge set E can be split into edges E_{c_i} within components c_i and into edges $E(C)$ connecting different components. Thus, every permutation $\tau : E \to [|E|]$ on E can be split into $\tau|_{E_{c_i}}$ corresponding to the components c_i (for all $i \in [k]$) and into $\tau|_{E(C)}$ corresponding to $E(C)$. Every tuple $(\tau_{E_{c_0}}, \ldots, \tau_{E_{c_{k-1}}}, \tau_{E(C)})$ can be constructed from exactly $\frac{|E|!}{|E(C)|! \cdot \prod_{i \in [k]} |E_{c_i}|!}$ different τ's. As a result, in order to prove this lemma, it suffices if we show the following for every permutation τ of E:

$$\mathsf{MSF}(c_0, E_{c_0}, \tau|_{E_{c_0}}) \cup \cdots \cup \mathsf{MSF}(c_{k-1}, E_{c_{k-1}}, \tau|_{E_{c_{k-1}}}) \cup \mathsf{MSF}(C, E(C), \tau|_{E(C)}) = \mathsf{MSF}(V, E, \tau)$$

Fix any such τ and assume that the equality does not hold. Let $F_i := \mathsf{MSF}(c_i, E_{c_i}, \tau|_{E_{c_i}})$, $F := \mathsf{MSF}(C, E(C), \tau|_{E(C)})$, and $M := \mathsf{MSF}(V, E, \tau)$. Let $e \in E$ be the best edge (w.r.t. $\mathbf{w}$, or secondarily τ) that is in either $F_0 \cup \cdots \cup F_{k-1} \cup F$ or M but not in the other one. There are a few cases:

- Assume $e \in M$, but $e \notin F_0 \cup \cdots \cup F_{k-1} \cup F$.

  If $e \in E(C)$, then the two endpoints of e are in different components in C, and therefore we have $\mathbf{w}(e) \geq w^*$ (by assumption (ii)). We can add e to F to obtain $F' := F \cup \{e\}$. This must result in a cycle w.r.t. C, as otherwise F was not spanning. All edges in the cycle must be better than e (otherwise F was not minimum), and because e is the best edge on which the two sets differ, all of them are also contained in M. Thus, M contains a cycle w.r.t. C. Since every two vertices in a component $c \in C$ are connected by edges in M, M even has a cycle w.r.t. V, a contradiction.

  If $e \in E_{c_i}$, then the two endpoints of e are in the same component c_i. We can add e to F_i to obtain $F_i' := F_i \cup \{e\}$, which must result in a cycle on c_i. All edges in the cycle are better than e (otherwise F_i was not minimum), so M contains a cycle on c_i, a contradiction.

- Assume $e \in F$, but $e \notin M$.

  We can add e to M to obtain $M' := M \cup \{e\}$, which must contain a cycle on C. All of the edges in the cycle are better than e, and therefore also contained in F. Thus, F contains a cycle, a contradiction.

- Assume $e \in F_i$, but $e \notin M$.

  We can add e to M to obtain $M' := M \cup \{e\}$, which must contain a cycle on c_i. All of the edges in the cycle are better than e, and therefore also contained in F_i. Thus, F_i contains a cycle, a contradiction.

□

The following corollary can be used to show that when no edges between the components in C exist, an MSF for the whole graph can be constructed as a combination of MSFs for the components itself.

Corollary 1. *Let $G = (V, E)$ be a graph and $C = \{c_0, \dots, c_{k-1}\}$ a partitioning of V. Assume that $E(C) = \emptyset$, i.e., there are no edges remaining that connect two different components of C. Then, sampling F_i as $\mathsf{MSF}(c_i, E_{c_i})$ for all $i \in [k]$ (all of them happening independently of each other) and returning $F_0 \cup \dots \cup F_{k-1}$ has the same output distribution as $\mathsf{MSF}(V, E)$.*

Proof. Let $w^* := \max\{\mathbf{w}(e) \mid e \in E\} + 1$. Using this w^*, both prerequisites for Lemma 4 are satisfied. In addition, the return value of $\mathsf{MSF}(C, E)$ is always $\emptyset$ because of $E(C) = \emptyset$. Thus, according to Lemma 4, $\mathsf{MSF}(V, E)$ is equivalent to $F_0 \cup \dots \cup F_{k-1}$ for F_i sampled from $\mathsf{MSF}(c_i, E)$. □

Now we show that if all remaining components in a graph are connected using a specific weight w^*, then all other edges can be disregarded:

Lemma 5. *Let $G = (V, E)$ be a graph and C a partitioning of V. Assume that there is a weight $w^* \in \mathbb{N}$ s.t. the following conditions are satisfied:*

(i) all components of C are connected to each other on the graph $(V, E_{=w^*})$ w.r.t. C (i.e., they are connected to each other even when considering only edges of weight $= w^*$), and

(ii) $\min\{\mathbf{w}(e) \mid e \in E(c_i)\} = w^*$ for every $i \in [k]$ (i.e., there are no edges of weight $< w^*$ that could be added to an MSF).

Then, $\mathsf{MSF}(C, E)$ has the same output distribution as $\mathsf{MSF}(C, E_{=w^*})$.

Proof. For every permutation $\pi : E \to [|E|]$, we define $\tau = \pi|_{E_{=w^*}}$ as the corresponding permutation restricted to $E_{=w^*}$. Since every $\tau : E_{=w^*} \to [|E_{=w^*}|]$ has exactly $\frac{|E|!}{|E_{=w^*}|!}$ corresponding π's, it suffices if we prove that

$$\mathsf{MSF}(C, E, \pi) = \mathsf{MSF}(C, E_{=w^*}, \pi|_{E_{=w^*}})$$

for every permutation $\pi : E \to [|E|]$.

Fix any such π and assume that the equality does not hold. Abbreviate $\mathsf{MSF}(C, E, \pi)$ by F and $\mathsf{MSF}(C, E_{=w^*}, \pi|_{E_{=w^*}})$ by M. Let $e \in E$ be the best edge (w.r.t. $\mathbf{w}$, or secondarily π) that is in either F or M but not in the other one. There are two cases:

- $e \in F$ but $e \notin M$. According to the assumptions, and because M is spanning, every two components $c, c' \in C$ are reachable on (V, M) w.r.t. C. Thus, if we add e to M to obtain $M' := M \cup \{e\}$, M' must contain a cycle on C. All edges besides from e on the cycle have weight w^*. Because of $\mathbf{w}(e) \geq w^*$, all edges on the cycle must be better than e (w.r.t. $\mathbf{w}$, or secondarily π), and therefore they are all contained in F. Thus, F also contains a cycle on C, a contradiction.
- $e \in M$ but $e \notin F$. If we add e to F to obtain $F' = F \cup \{e\}$, F' must contain a cycle. Since all edges on the cycle (besides from e itself) are better than e, they must have weight w^*. This means that they are all contained in M, and therefore M contains a cycle, which is a contradiction.

□

By combining Lemma 5 and Lemma 4, we now obtain the following corollary. It states the most important observation that we will make use of in Protocol 15: MSFs on components that require only edges of weight $< w^*$ can be combined into an MSF for the whole graph if the components are connected to each other using edges of weight w^*.

Corollary 2. *Let $G = (V, E)$ be a graph and $C = \{c_0, \ldots, c_{k-1}\}$ a partitioning of V. Assume that there is a weight $w^* \in \mathbb{N}$ s.t. the following conditions are satisfied:*

(i) all vertices within c_i are connected on the graph $(V, E_{<w^*})$ for all $i \in [k]$, and

(ii) all components of C are connected to each other on the graph $(V, E_{=w^*})$ w.r.t. C, and

(iii) $\min\{\mathbf{w}(e) \mid e \in E(c_i)\} = w^*$ for every $i \in [k]$.

Then, sampling F_i as $\mathsf{MSF}(c_i, E_{c_i})$ for all $i \in [k]$, sampling F as $\mathsf{MSF}(C, E_{=w^*})$ (all of these happening independently of each other), and returning $F_0 \cup \cdots \cup F_{k-1} \cup F$ has the same output distribution as $\mathsf{MSF}(V, E)$.

Proof. First, apply Lemma 5 to show that $\mathsf{MSF}(C, E)$ has the same output distribution as $\mathsf{MSF}(C, E_{=w^*})$. Then this corollary follows from Lemma 4. □

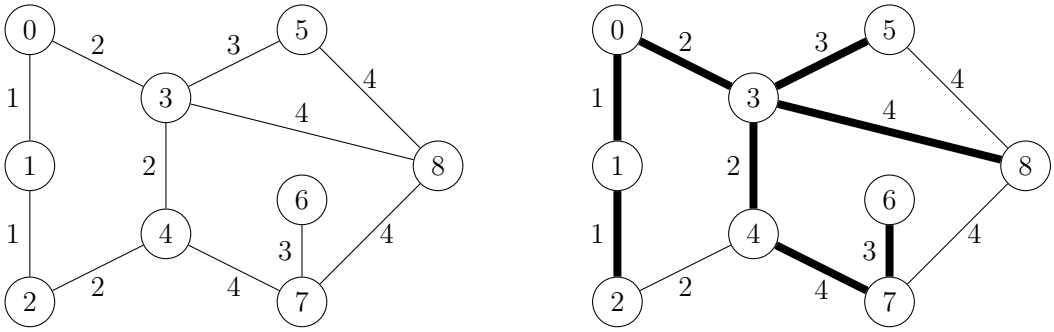


Figure 9.1: The graph on the left is an example where each weight may occur more than once. Any MSF (e.g. the thick edges on the right) would consist of all edges of weight 1 and 3, any two edges of weight 2, and two edges of weight 4 that do not result in a cycle.

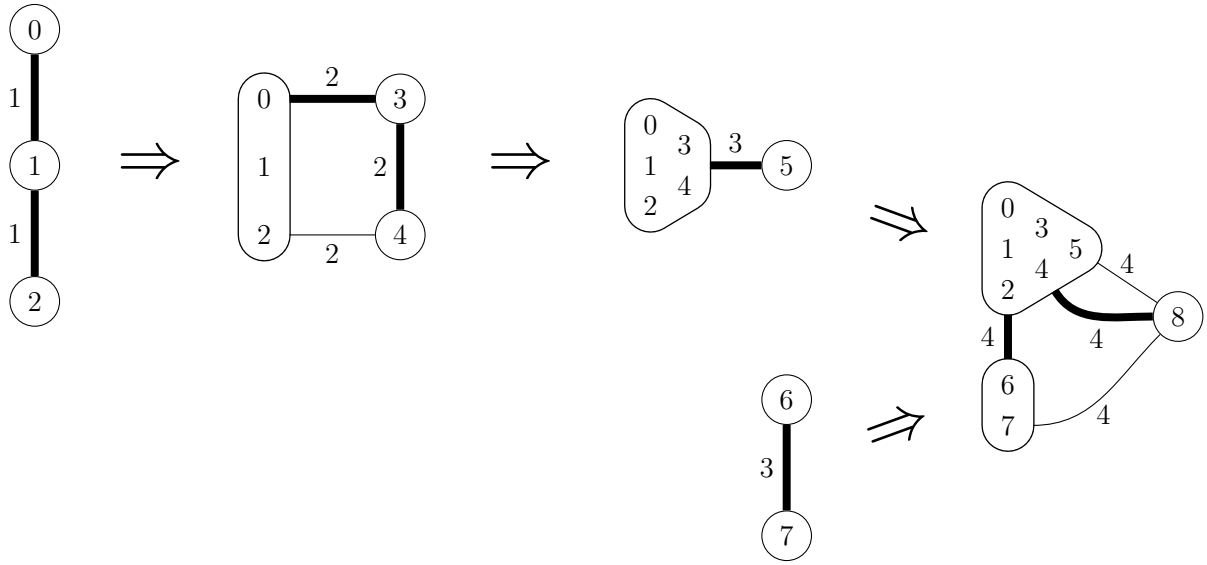


Figure 9.2: This graphic shows for the graph given in Figure 9.1 all subgraphs for which the MSF can be computed separately in order to combine them into one MSF for the whole graph. A component of vertices that are already merged together is depicted in this figure as multiple indices surrounded by a single line. $\mathsf{MSF}(\{\{0\},\{1\},\{2\}\}, E_{=1})$ results in the merged component $\{0,1,2\}$. Then, $\mathsf{MSF}(\{\{0,1,2\},\{3\},\{4\}\}, E_{=2})$ leads to the component $\{0,1,2,3,4\}$, $\mathsf{MSF}(\{\{0,1,2,3,4\},\{5\}\}, E_{=3})$ to $\{0,1,2,3,4,5\}$, $\mathsf{MSF}(\{\{6\},\{7\}\}, E_{=3})$ to $\{6,7\}$, and finally $\mathsf{MSF}(\{\{0,1,2,3,4,5\},\{6,7\},\{8\}\}, E_{=4})$ results in all vertices being merged together into one component $\{0,1,2,3,4,5,6,7,8\}$. Combining these subgraph MSF's results in the MSF shown on the right-hand side in Figure 9.1.

9.2 Informal description

Figure 9.1 contains an example on which we will describe how the protocol works.

The protocol maintains a partitioning C of the vertices $V \subseteq \mathcal{V}$ such that the MSF for each component $c \in C$ can be found separately, because all edges in the MSF of c have strictly smaller weight than any edge leading out of component c. The goal is to use UnweightedSF (Protocol 12) on subgraphs that are as small as possible, and then merge the corresponding components together. We use Corollary 2 to do this in the following way: Whenever there is a subset $S = \{c_0, \ldots, c_{k-1}\}$ of components, all of them are connected to each other on the graph $(V, E_{=w^*})$ w.r.t. C, and the combination $c^* = c_0 \cup \cdots \cup c_{k-1}$ of all these components does not have any outgoing edge of weight $\leq w^*$, then the MSF for c^* can be computed as $\mathsf{MSF}(c_0, E_{c_0}) \cup \cdots \cup \mathsf{MSF}(c_{k-1}, E_{c_{k-1}}) \cup \mathsf{MSF}(S, (E_{c^*})_{=w^*})$. In the following description, we call such a set $S \subseteq C$ of the partitioning C simply a *subgraph*.

As an example, Figure 9.2 shows how the graph in Figure 9.1 can be broken down into minimal subgraphs for which their MSF can be computed separately and then combined into the complete MSF. The combined MSF is shown on the right-hand side in Figure 9.1.

It is important to note the following: the edges returned by UnweightedSF are not needed before the end of the MSF protocol, and therefore it is not necessary to wait for the result of a call to this sub-protocol. Instead, our protocol can compute all UnweightedSF's together in the end, and then output the union of all returned sets of edges.

9.2.1 Finding subgraphs

It remains to answer the question of how to efficiently and securely find those subgraphs for which the MSF's can be computed separately. For a single party who only knows their own edges, it is impossible to determine these subgraphs on their own. Thus, a few important observations regarding publishable information (without any leakage) are required. These are shown in Section 9.5. For example, whenever we have a component $c \subseteq V$, then the minimum weight $\min\{\mathbf{w}(e) \mid e \in E(c)\}$ of an edge leading out of the component c can be revealed without any leakage (Lemma 6). In addition, given a weight w, it is allowed to reveal the set of components resulting from merging only edges of weight $\leq w$ (Lemma 7).

One option for finding subgraphs with separately computable subgraph MSF's would be to find $\mathrm{Conn}(V, E_{\leq w})$ simultaneously for every possible weight $w \in \mathbb{N}$. Then, if $c_0, \ldots, c_{k-1}$ are components found by $\mathrm{Conn}(V, E_{\leq w-1})$, but all vertices in $c_0 \cup \cdots \cup c_{k-1}$ are connected

to each other by $\mathrm{Conn}(V, E_{\leq w})$, then $\mathsf{MSF}(\{c_0, \ldots, c_{k-1}\}, w)$ is one of the desired subgraph MSF's. However, this method has a huge communication overhead: Every call to Connectivity would require a total amount of $O(|V|^3)$ communicated bits, and if the space of possible weights (which could be 32-bit integers in a typical implementation) is not extremely small, this method is infeasible.

For this reason, we attempt to compute Connectivity only whenever it is necessary. Our protocol applies some optimizations to use as much of the public information as possible in order to determine subgraphs which can be merged and for which the MSF can be computed separately.

Assume that for every remaining component $c \in C$, we already know the weight $\min\{\mathbf{w}(e) \mid e \in E(c)\}$ of the best incident edge. Then, let w be such a weight that occurred as the best possible weight leading out of at least one component $c \in C$, and let $S \subseteq C$ be the subset of components with $\min\{\mathbf{w}(e) \mid e \in E(c)\} = w$ for all $c \in S$. Using Connectivity, we can compute which of the components in S are connected to each other using edges of weight w. If $s \subseteq S$ is a set of components connected to each other using edges of weight w, and none of them has another edge of weight w going to a component that is not part of s, then s is one of the subgraphs for which the MSF can be computed separately, and its components can be merged.

9.2.2 Detailed description

The protocol alternates (until no new edges can be discovered) between two essential steps that require communication with the other party. Both of these steps can be performed for all remaining components in parallel. After performing both steps, the protocol discovers subgraphs for which the MSF can be computed separately, and merges the components accordingly. After performing all of these steps, the protocol begins the next iteration, starting again with the first step.

First step The first step is to compute $w_c = \min\{\mathbf{w}(e) \mid e \in E(c)\}$ for each component $c \in C$, which represents the weight of the best edge leaving c. This value is revealed to both parties. Computing w_c is very simple: party p can find $v_p := \min\{\mathbf{w}(e) \mid e \in E^{(p)}(c)\}$ locally, and then the result is simply $\min(v_1, v_2)$, computable by secret-sharing v_1 and v_2, testing whether $[\![v_1]\!] < [\![v_2]\!]$, and then using a multiplexer to choose and reveal the smaller one of $[\![v_1]\!]$ or $[\![v_2]\!]$.

Second step Locally, every party can compute $S_w = \{c \in C \mid w_c = w\}$ for every weight $w \in \mathbb{N}$ that occurred in the previous step, which contains all components $c \in C$ where w is the best weight of an incident edge (i.e., $w_c = w$). Assume that $S_w = \{c_0, \ldots, c_{k-1}\}$. Now, the protocol starts to test which of these components are connected to each other (using edges of weight w), and which of them are connected to components in $C \setminus \{c_0, \ldots, c_{k-1}\}$. This information can be used to decide which of the components to merge, and for which of them we need to wait for more information about the rest of the graph. To do this, we compute the connectivity of components $\{c_0, \ldots, c_{k-1}, c_w^*\}$ (where $c_w^* = V \setminus \bigcup_{i \in [k]} c_i$ consists of all vertices that are in none of the c_i's) on edges $E_{=w}$. This is done as described in Functionality 10 (which implements the function $\mathrm{Conn}(\cdot, \cdot)$), and results in $d_0, \ldots, d_{\ell-1}$, groups of c_i's.

For example, consider the graph in Figure 9.1, and look at all vertices whose best incident weight is 3. These are exactly the vertices 5, 6, and 7 (i.e., $S_3 = \{\{5\}, \{6\}, \{7\}\}$). Then $c_3^* = \{0, 1, 2, 3, 4, 8\}$ is the set of vertices not in S_3. Now the protocol would compute $\mathrm{Conn}(\{\{5\}, \{6\}, \{7\}, \{0, 1, 2, 3, 4, 8\}\}, E_{=3})$ in order to find out which of the vertices are connected to each other using edges of weight 3. This computation would yield the two groups $\{\{6\}, \{7\}\}$ and $\{\{5\}, \{0, 1, 2, 3, 4, 8\}\}$, which means that 6 and 7 are connected to each other with edges of weight 3, while vertex 5 is connected with edges of weight 3 to components that are not in S_3.

Discovering subgraphs Using the information revealed in the first two steps, each party can now locally (without any communication with the other party) decide which subgraphs can be merged. Assume that $(d_0, \ldots, d_{\ell-1}) = \mathrm{Conn}(\{c_0, \ldots, c_{k-1}, c_w^*\}, E)$ was the result of a of the connectivity test in the previous step. Then we can process this information as follows:

(1) Let $i \in [\ell]$, where $c_w^* \notin d_i$. This means that all components in d_i are connected to each other using edges of weight w, and none of them has an outgoing edge of weight w to a component that is not in d_i. Thus, d_i is a set of components for which the MSF can be computed separately: We merge all components in d_i, i.e., define $g_i := \bigcup_{c \in d_i} c$, and we set $C \leftarrow (C \setminus d_i) \cup \{g_i\}$. By Corollary 2, the MSF for g_i can be generated as the union of the MSF's for all $c \in d_i$, combined with the MSF generated by Protocol 12 on the components from d_i.

Note that $\min\{\mathbf{w}(e) \mid e \in E(g_i)\} > w$, i.e., the weight of the best edge leaving the new component g_i is larger than w. The new value of $\min\{\mathbf{w}(e) \mid e \in E(g_i)\}$ will be computed in the next iteration of the protocol.

(2) For the special set d_{i^*} that contains c_w^* (i.e., $c_w^* \in d_{i^*}$), we can't do anything yet: There must be other components in $C \setminus \{c_0, \ldots, c_{k-1}\}$ that $c \in d_i$ are connected to using edges of weight w, but we do not know which ones they are. For this reason, the components in d_{i^*} will wait until in some iteration, there is another component that has w as its best incident weight.

In our example with $w = 3$, case (1) occurs for $\{\{6\}, \{7\}\}$, and case (2) occurs for $\{\{5\}, \{0, 1, 2, 3, 4, 8\}\}$. This means that the protocol can merge $\{\{6\}, \{7\}\}$ into a single component $\{6, 7\}$, and compute the corresponding MSF (consisting only of a single edge) separately. The only other subgraph MSF that can be computed immediately in the first iteration is $\mathsf{MSF}(\{\{0\}, \{1\}, \{2\}\}, E_{=1})$, because the vertices 0, 1, and 2 are connected to each other using edges of weight 1, and all of their edges going to other components have weight > 1. This is shown in Figure 9.3(a).

This protocol terminates after a finite amount of iterations for the following simple reasons: Given the current partitioning C, let w^* be the smallest weight among all $\min\{\mathbf{w}(e) \mid e \in E(c)\}$ for any $c \in C$, and let $c_0, \ldots, c_{k-1} \in C$ be those components which fulfill $\min\{\mathbf{w}(e) \mid e \in E(c_i)\} = w^*$. Then, every c_i is connected to at least one other c_j ($j \neq i$), thus resulting in at least one merge happening in this iteration. There can be at most $|V| - 1$ merge steps that take place before the whole graph has been merged into connected components. At that point of time, we will have $w_c = \infty$ for all $c \in C$.

9.2.3 Optimization

There is a simple, but very effective, optimization that can be applied in order to minimize the amount of unnecessary computations.

At the end of an iteration (after some separately computable subgraphs have already been determined), let w be the smallest incident weight for a set of components s_w that have *not* been merged (i.e., case (2) occurred). If there is only one component $c \in C$ (that is not contained in s_w) which can potentially have outgoing edges of weight w, then the set $s_w \cup \{c\}$ is another separately computable subgraph. That is, the components in $s_w \cup \{c\}$ can be merged together using edges of weight w, and its MSF needs to be added to the final output. This optimization can be repeated for as long as possible in a single iteration. Whenever there is more than one component $c \in C$ that might have outgoing edges of weight w, it is impossible to determine which of these components the s_w are connected to, and therefore we cannot do anything further.

It remains to answer the question of how to check whether a component $c \in C$ may have an outgoing edge of weight w. If c resulted from the merge in the current iteration by

edges of weight w', then we know that any outgoing edge of c has weight $> w'$. This yields a very simple way (that doesn't require any computation shared between parties) to restrict the weights of outgoing edges of a component c, and therefore this is the mechanism that we deploy in our protocol for checking the condition needed for applying the optimization.

More specifically, if w is the second-to-smallest weight used for creating any component in C as the combination of multiple smaller components in the current iteration, then for any $w' \leq w$, we can merge all components in $s_{w'}$ with the unique (already merged) component c for which the corresponding weight was smaller than w'. As shown in Figure 9.3, this optimization would yield to two additional merge steps in the first iteration for our example graph.

9.3 Formalized protocol

Protocol 15 formally describes this process. Note that we assume that the initial partitioning C is given as $C = \overline{V}$, i.e. every vertex is in its own partition. Thus, Protocol 15 is going to find a random MSF from $\mathsf{MSF}(V, E)$, which slightly differs from the more general Functionality 1, which requires to find a random MSF from $\mathsf{MSF}(C, E)$ for a partitioning C of V. Even though Protocol 15 also works for an arbitrary partitioning C, the correctness and security proofs become simpler if we restrict ourselves to $C = \overline{V}$.

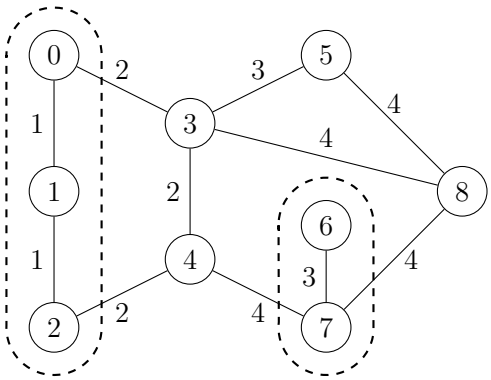
In this protocol, the current partitioning C is modified whenever two or more components are merged together. Additionally, a function $F : C \to 2^{\mathcal{E}}$ is maintained, which maps each component $c \in C$ to a set of edges that together form the MSF that was generated on c. The protocol uses a function $\textsc{Merge}(S = \{c_0, \ldots, c_{k-1}\}, w)$ (line 4) to merge together components of a subgraph $S = \{c_0, \ldots, c_{k-1}\}$ that fulfill the following conditions (which are exactly the preconditions of Corollary 2):

(i) all vertices within c_i are connected on the graph $(V, E_{<w})$ for all $i \in [k]$, and

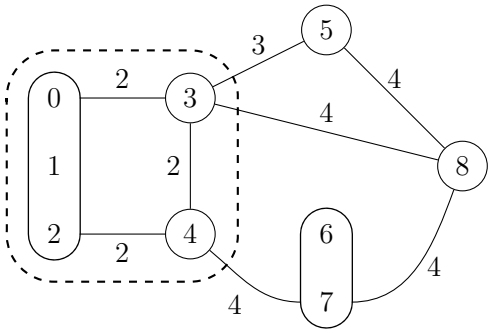
(ii) all components of C are connected to each other on the graph $(V, E_{=w})$ w.r.t. C, and

(iii) $\min\{\mathbf{w}(e) \mid e \in E(c_i)\} = w$ for every $i \in [k]$.

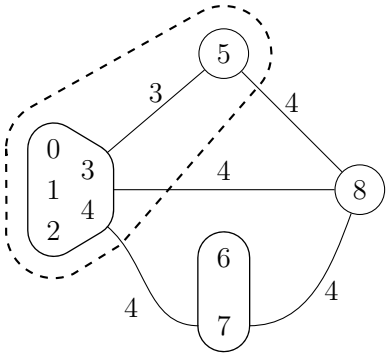
Thus, using Corollary 2, the MSF $F(c^*)$ on the new component $c^* = \bigcup_{i \in [k]} c_i$ can be found by computing $\mathsf{MSF}(S, E_{=w})$, and combining this with $F(c_i)$ for all $i \in [k]$. As $E_{=w}$ contains only edges of a single weight, $\mathsf{MSF}(S, E_{=w})$ can be found securely with a call to UnweightedSF. However, this does not need to happen immediately. Instead, as the overall output of Protocol 15 is simply the union of the return values of all calls to



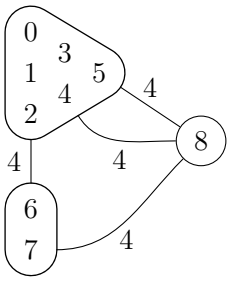
(a) First iteration (without optimization)



(b) First application of optimization



(c) Second application of optimization



(d) State after the complete first iteration

Figure 9.3: This example illustrates all the subgraphs that can be discovered in the first iteration of the protocol. Each component contains only a single vertex, so in the beginning of this first iteration of the protocol, a vertex is equivalent to a component. After computing $\min\{\mathbf{w}(e) \mid e \in E(c)\}$ for every $c \in C$, and testing the connectivity for all weights that occurred in the first step (1, 2, 3, and 4), the components $\{0\}$, $\{1\}$, and $\{2\}$ could be merged (because $\mathsf{MSF}(\{\{0\},\{1\},\{2\}\}, E_{=1})$ can be computed separately), and the components $\{6\}$ and $\{7\}$ could be merged (and $\mathsf{MSF}(\{\{6\},\{7\}\}, E_{=3})$ can be computed separately). This is shown by the dashed lines in (a). The optimization then notices that vertices 3 and 4 must both be connected to the new component $\{0,1,2\}$ using edges of weight 2 (b). Thus, $\mathsf{MSF}(\{\{0,1,2\},\{3\},\{4\}\}, E_{=2})$ can be computed separately, and the components are merged together into $\{0,1,2,3,4\}$. Then, according to the optimization, $\{5\}$ must be connected to the new component $\{0,1,2,3,4\}$, so these two components are merged together, and $\mathsf{MSF}(\{\{0,1,2,3,4\},\{5\}\}, E_{=3})$ is added to the global MSF (c). In the final state (d), it is not possible to apply the optimization again, because for the component $\{8\}$ it is unknown whether it has an edge to $\{0,1,2,3,4,5\}$ or to $\{6,7\}$ (or both).

UnweightedSF that were made, it is possible to run all UnweightedSF's in parallel at the end of the protocol in order to save communication rounds.

The while-loop in line 10 alternates between the two described steps until no more edges can be added to the MSF (line 17). The for-loop in line 12 finds the best incident weight w_c for each remaining component $c \in C$. W contains the set of all weights w_c that were found in this first step. The for-loop in line 18 tests which of the components in S_w (which contains all components $c \in C$ with $w_c = w$) are connected to each other using edges of weight w, and which of them are still connected to components outside of S_w using edges of weight w. t_w contains the result of the call to Connectivity.

The rest of the protocol consists only of local computations (and calls to Merge which are also purely local if all UnweightedSF's are called simultaneously in the end). The set merged (line 23) will contain pairs (c, w) of a component $c \in C$ and a weight $w \in \mathbb{N}$, s.t. it is known that the best outgoing edge of component c has weight $> w$ (i.e., $\min\{\mathbf{w}(e) \mid e \in E(c)\} > w$). In line 29, a set d_i of components is merged together if it is known that all of them are reachable from each other using edges of weight w, and if no component from d_i is connected to any component outside of d_i using an edge of weight at most w. Thus, in line 30, (c, w) is added to merged with $c = \bigcup_{c' \in d_i} c'$ being the new component. On the other hand, for every $w \in W$, s_w (line 27) is defined to contain those components $c \in C$ with $\min\{\mathbf{w}(e) \mid e \in E(c)\}$ that are not yet merged because they are connected using edges of weight w to components outside of S_w.

The while-loop in line 33 performs the optimization described in Section 9.2.3. Let w be the smallest weight (if any exists) s.t. $s_w \neq \emptyset$. If there is only one $(c, w') \in$ merged with $w' < w$ (i.e., only c can be connected to the components in s_w using edges of weight w), then the components in $s_w \cup \{c\}$ are merged together. (c, w') needs to be replaced by (c', w) in merged (where c' is the new merged component), as it is known that any edge leaving c' needs to have weight $> w$ (line 35).

9.3.1 Practical considerations

As described, whenever Merge is called on a subgraph S for weight w, UnweightedSF is used in order to find $\mathsf{MSF}(S, E_{=w})$. However, the actual output of UnweightedSF is not used until Protocol 15 terminates, at which point the union of all outputs of UnweightedSF is returned. Therefore, we can defer the execution of all these calls until the rest of the protocol has terminated in order to run them in parallel and save communication rounds.

In addition, it can happen that the protocol attempts to compute $w_c = \min\{\mathbf{w}(e) \mid e \in E(c)\}$ multiple times for a single component $c \in C$. For example, this happens when w_c is

Protocol 15 Random Minimum Spanning Forest

Public: Partitioning $C = \overline{V}$ of a finite set of vertices $V \subseteq \mathcal{V}$
protocol RandomMSF($E^{(1)}$, $E^{(2)}$)
  State $F : C \to 2^{\mathcal{E}}$, i.e. the MSF's discovered for the already merged components
  function Merge($S = \{c_0, \ldots, c_{k-1}\}$, w)
    $c^* \leftarrow \bigcup_{i \in [k]} c_i$
    $C \leftarrow (C \setminus S) \cup \{c^*\}$
    $F(c^*) \leftarrow F(c_0) \cup \cdots \cup F(c_{k-1}) \cup \text{UnweightedSF}_S((E^{(1)}_{c^*})_{=w}, (E^{(2)}_{c^*})_{=w})$
    return c^*
  end function
  while true do
    $W \leftarrow \emptyset$
    for $c \in C$ (in parallel) **do**
      Party p computes $v_p \leftarrow \min\{\mathbf{w}(e) \mid e \in E^{(p)}(c)\}$
      $w_c \leftarrow \text{Minimum}(v_1, v_2)$
      if $w_c \neq \infty$ **then** $W \leftarrow W \cup \{w_c\}$ **end if**
    end for
    if $W = \emptyset$ **then break end if**
    for $w \in W$ (in parallel) **do**
      $S_w \leftarrow \{c \in C \mid w_c = w\}$
      $c^*_w \leftarrow V \setminus \bigcup_{c \in S_w} c$
      $t_w \leftarrow \text{Connectivity}_{S_w \cup \{c^*_w\}}(E^{(1)}_{=w}, E^{(2)}_{=w})$
    end for
    $\mathsf{merged} \leftarrow \emptyset$
    for $w \in W$ **do**
      $(\{d_0, \ldots, d_{\ell-1}\}) \leftarrow t_w$
      $i^* \leftarrow$ the unique index $i \in [\ell]$ with $c^*_w \in d_{i^*}$
      $s_w \leftarrow d_{i^*} \setminus \{c^*_w\}$
      for $i \in [\ell] \setminus \{i^*\}$ **do**
        $c \leftarrow \text{Merge}(d_i, w)$
        $\mathsf{merged} \leftarrow \mathsf{merged} \cup \{(c, w)\}$
      end for
    end for
    while $\exists$ smallest $w \in W$ with $s_w \neq \emptyset$ s.t. there is only one $(c, w') \in \mathsf{merged}$ with $w' < w$ **do**
      $c' \leftarrow \text{Merge}(s_w \cup \{c\}, w)$
      $\mathsf{merged} \leftarrow (\mathsf{merged} \setminus \{(c, w')\}) \cup \{(c', w)\}$
      $s_w \leftarrow \emptyset$
    end while
  end while
  return $\bigcup_{c \in C} F(c)$
end protocol

computed in the first iteration, but c is not merged together with any other component yet. Then, in the second iteration, w_c would have to be computed again. 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: whenever this sub-protocol is called with the same parameters for a second time, a cached value from the first call can be used instead.

9.4 Correctness

Theorem 9. *Protocol 15 correctly implements Functionality 1.*

Proof. We only need to show that at any point of time the following two invariants hold for any $c \in C$:

(1) $F(c)$ is an MSF sampled according to the distribution $\mathsf{MSF}(c, E_c)$, and

(2) all vertices in c are connected to each other on the graph $(V, E_{<w^*})$, where $w^* = \min\{\mathbf{w}(e) \mid e \in E(c)\}$ (i.e., any edge leaving c is larger than any edge in the MSF of c).

If this holds, the algorithm terminates with $\bigcup_{c \in C} F(c)$ when $W = \emptyset$, i.e., there is no edge $e \in E(c)$ for any $c \in C$. Thus, $E(C) = \emptyset$ and by Corollary 1, the output is $\mathsf{MSF}(V, E)$, which is exactly the desired behavior.

Now we show that (1) and (2), which trivially hold in the beginning, are maintained throughout Protocol 15. Whenever MERGE$(S = \{c_0, \ldots, c_{k-1}\}, w)$, $k \geq 2$ is called with components $c_i \in C$ s.t.

(i) all pairs of c_i, c_j are connected to each other on the graph $(V, E_{=w})$ w.r.t. C,

(ii) $\min\{\mathbf{w}(e) \mid e \in E(c_i)\} = w$ for all $i \in [k]$, and

(iii) $\min\{\mathbf{w}(e) \mid e \in E(c^*)\} > w$ for the new component $c^* = \bigcup_{i \in [k]} c_i$,

then both invariants (1) (according to Corollary 2) and (2) (which follows from (i) and (iii)) are maintained. Thus, it suffices to prove that (i)-(iii) hold whenever MERGE is called.

Note that up to line 23, MERGE is not called at all, but the protocol computes $w_c = \min\{\mathbf{w}(e) \mid e \in E(c)\}$ for each $c \in C$, and for every weight $w \in W = \{w_c \mid c \in C\}$, the following values are found:

- $S_w = \{c \in C \mid w_c = w\}$ (the set of components with w being the best incident weight),
- $c_w^* = V \setminus \bigcup_{c \in S_w} c$ (the set of vertices not contained in S_w), and
- t_w (indicating which components in $S_w \cup \{c_w^*\}$ are connected to each other using only edges of weight w).

In the subsequent for-loop (line 24), let $w \in W$ and $i \in [\ell] \setminus \{i^*\}$ as defined in Protocol 15, with $d_i = \{c_0, \ldots, c_{k-1}\}$, for which MERGE$(d_i, w)$ will be called. Because of $w_{c_i} = w$ for all c_i, we know that condition (ii) holds. Furthermore, by definition of t_w, all $c \in d_i$ are connected to each other on the graph $(V, E_{=w})$ w.r.t. C (condition (i)). If $c^* = \bigcup_{c \in d_i} c$ is the new combined component, then we will also have $\min\{\mathbf{w}(e) \mid e \in E(c^*)\} > w$ (condition (iii)), as otherwise there would be an edge e between $c \in d_i$ and $c' \notin d_i$ with $\mathbf{w}(e) = w$, a contradiction to the definition of t_w.

For the following while-loop that performs the optimization (line 33), we require the following additional invariants that hold before and after each of its iterations:

(a) For every $c \in C$ with $E(c) \neq \emptyset$: there is a w with $(c, w) \in \mathsf{merged}$ or $c \in s_w$ (i.e., each remaining component $c \in C$ has either been merged in this iteration, or it is still in the set of unmerged components s_w).

(b) $\forall (c, w) \in \mathsf{merged}$: $c \in C$ with $\min\{\mathbf{w}(e) \mid e \in E(c)\} > w$ (i.e., for all merged components, the best incident weight is higher than that of all previously seen edges).

(c) $\forall w \in W, c \in s_w$: $c \in C$ with $\min\{\mathbf{w}(e) \mid e \in E(c)\} = w$, and c is connected to c_w^* on the graph $(V, E_{=w})$ w.r.t. C.

These invariants hold before the first iteration, as for any $w \in W$, there are just two changes that could have happened in the previous for-loop:

- For i^* with $c_w^* \in d_{i^*}$, any component $c \in d_{i^*} \setminus \{c_w^*\}$, c is added to s_w. By construction, we have $\min\{\mathbf{w}(e) \mid e \in E(c)\} = w$, and c is connected to c_w^* using edges from $E_{=w}$ (because CONNECTIVITY put c and c_w^* into the same set d_{i^*}). Thus, (c) is fulfilled for this c.
- For every other $i \neq i^*$, the components in d_i are merged together into one component c. Then (c, w) is inserted into merged. (b) is fulfilled, because otherwise there would be an edge $e \in E(c)$ with $\mathbf{w}(e) = w$. However, then one of the components in d_i would have had an edge of weight w to a component in d_j for $j \neq i$, a contradiction to the correctness of CONNECTIVITY.

Invariant (a) is satisfied as for every $c \in C$ with $E(c) \neq \emptyset$, the iteration for the corresponding $w \in W$ either put c into merged, or into s_w.

Now we show that (i)-(iii) hold for every MERGE in the while-loop in line 34, and that invariants (a)-(c) are maintained. Assume that w is the smallest $w \in W$ with $s_w \neq \emptyset$ and that there is only one $(c, w') \in \mathsf{merged}$ with $w' < w$. Then, MERGE$(s_w \cup \{c\}, w)$ will be called, which creates a new component c'. By (c), every component in s_w is connected to c^*_w using edges of weight w. In addition, by assumption (w being minimal and there is only one $(c, w') \in \mathsf{merged}$ with $w' < w$), and (a)-(c), every component $c'' \in C$ with $c'' \notin s_w \cup \{c\}$ has only incident edges of weight $> w$. This means that every component in s_w is connected to c itself using edges of weight w, which implies (i). Furthermore, every edge leaving c' must have weight $> w$, which proves (iii). (ii) follows directly from (c).

It remains to show that the invariants (a)-(c) continue to hold for every iteration of this while-loop in line 33. Invariant (a) stays true (since for the new component $c' \in C$, we have $(c', w) \in \mathsf{merged}$), invariant (b) stays true (since the best incident edge of c' has weight $> w$), and invariant (c) stays true (since we set s_w to $\emptyset$).

In conclusion, we have shown correctness of the protocol under the condition that it terminates. For a single iteration, let w be the weight of the smallest edge remaining, i.e., $\min\{\mathbf{w}(e) \mid e \in E(C)\}$. At least two components $c, c' \in C$ must be directly connected by an edge of weight w, and they are in the same d_i with $c^*_w \notin d_i$ because the best outgoing edge of any component that is part of c^*_w must be larger than w. As a result, c and c' will be merged together in this iteration. The protocol always terminates, because in every iteration of the outer while-loop, there will be at least one merge happening, and there can be at most $|V| - 1$ merges in total. □

9.5 Security

The security of Protocol 15 is based on the following two lemmas.

Lemma 6. *Let $G = (V, E)$ be a graph. If $F \subseteq E$ is a minimum spanning forest (i.e., it is a possible output of $\mathsf{MSF}(V, E)$), then the following equation holds for any component $c \subseteq V$:*

$$\min\{\mathbf{w}(e) \mid e \in E(c)\} = \min\{\mathbf{w}(e) \mid e \in F(c)\}$$

Informally, this lemma states that the smallest weight of an edge leading out of component c stays the same if we restrict the edges we are looking at to a minimum spanning forest.

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

$$\min\{\mathbf{w}(e) \mid e \in E(c)\} < \min\{\mathbf{w}(e) \mid e \in F(c)\}.$$

That is, there is an edge $e \in E(c)$ with $\mathbf{w}(e) < \min\{\mathbf{w}(e) \mid e \in F(c)\}$. We can define $F' := F \cup \{e\}$. If this does not result in a cycle, then F was not a minimum *spanning* forest. If it does, remove any edge $e' \neq e$ on the cycle to obtain $F'' := F' \setminus \{e'\}$. Since we know that $\mathbf{w}(e) < \mathbf{w}(e')$, and breaking up the cycle does not make any pair of vertices unreachable if they were previously reachable from each other, F'' would be a spanning forest with $\mathbf{w}(F'') < \mathbf{w}(F)$, a contradiction to F being minimal. □

Lemma 7. *Let $G = (V, E)$ be a graph, C a partitioning of V, and $w^* \in \mathbb{N}$ an arbitrary weight. If $F \subseteq E$ is a minimum spanning forest of G, then the following equation holds:*

$$\mathrm{Conn}(C, E_{\leq w^*}) = \mathrm{Conn}(C, F_{\leq w^*})$$

Proof. Any two components $c_1, c_2 \in C$ that are connected on C using only edges from $F_{\leq w^*}$ are trivially also connected on C using only edges from $E_{\leq w^*}$ (because of $F_{\leq w^*} \subseteq E_{\leq w^*}$).

Now assume that c_1 and c_2 are connected using $E_{\leq w^*}$, but not using $F_{\leq w^*}$. Then at least one of the edges $e \in E_{\leq w^*}$ with $\mathbf{w}(e) \leq w^*$ on the path between c_1 and c_2 can be added to $F_{\leq w^*}$ without creating a cycle. However, if we add e to F and obtain $F' := F \cup \{e\}$, then F' must contain a cycle (otherwise F was not spanning). There is at least one edge e' on the cycle with weight $> w^*$ (because by assumption, the two endpoints of e are not connected using $F_{\leq w^*}$). Now, $F'' := F' \setminus \{e'\}$ no longer contains a cycle and therefore F'' is also a spanning forest. However, we have $\mathbf{w}(F'') < \mathbf{w}(F)$ (because of $\mathbf{w}(e) < \mathbf{w}(e')$), which is a contradiction to F being minimum.

We have shown that any two components $c_1, c_2 \in C$ are connected to each other with $E_{\leq w^*}$ iff they are connected to each other with $F_{\leq w^*}$. In conclusion, the two definitions of $\mathrm{Conn}(C, E_{\leq w^*})$ and $\mathrm{Conn}(C, F_{\leq w^*})$ are equivalent (see the definition in Section 3.2.2), which proves this lemma. □

Theorem 10. *Protocol 15 privately computes Functionality 1 in the semi-honest security model.*

Proof. A simple simulator S (that works for both parties in the same way) that only requires the MSF F returned by Protocol 15 in its input can be defined in the following way: S simulates the Protocol 15 pretending that party p's input was $F^{(p)} := \{(\{u, v\}, w^*, p^*) \in F \mid p^* = p\}$ (i.e., no edges other than those in the selected MSF exist).

In order to show security, we need to prove that the output of every call to a sub-protocol is the same in the real execution on edges E as in the simulator's execution on edges F. We may assume that all previous outputs of sub-protocols have already been the same, and we only need to show that the next one will be the same in both worlds.

Note that Protocol 15 only calls sub-protocols at three different places:

(1) $\textsc{Minimum}(v_1, v_2)$ in line 14. In the real world, this is

$$\textsc{Minimum}(\min\{\mathbf{w}(e) \mid e \in E^{(1)}(c)\}, \min\{\mathbf{w}(e) \mid e \in E^{(2)}(c)\}) = \min\{\mathbf{w}(e) \mid e \in E(c)\}.$$

Similarly, the simulator calls

$$\textsc{Minimum}(\min\{\mathbf{w}(e) \mid e \in F^{(1)}(c)\}, \min\{\mathbf{w}(e) \mid e \in F^{(2)}(c)\}) = \min\{\mathbf{w}(e) \mid e \in F(c)\}.$$

By Lemma 6 and due to the fact that F is an MSF of (V, E), we have $\min\{\mathbf{w}(e) \mid e \in E(c)\} = \min\{\mathbf{w}(e) \mid e \in F(c)\}$. Thus, the simulator's output is equal to the real output.

(2) $\textsc{Connectivity}_{S_w \cup \{c_w^*\}}(E^{(1)}_{=w}, E^{(2)}_{=w})$ in line 21. By correctness of $\textsc{Connectivity}$, the return value would be $\mathrm{Conn}(S_w \cup \{c_w^*\}, E_{=w})$. On the other hand, the simulator would call $\textsc{Connectivity}_{S_w \cup \{c_w^*\}}(F^{(1)}_{=w}, F^{(2)}_{=w})$, which returns $\mathrm{Conn}(S_w \cup \{c_w^*\}, F_{=w})$. Note that none of the components in S_w or c_w^* can have an outgoing edge of weight $< w^*$, so the outputs are the same as $\mathrm{Conn}(S_w \cup \{c_w^*\}, E_{\leq w^*})$ and $\mathrm{Conn}(S_w \cup \{c_w^*\}, F_{\leq w^*})$ in the real and the simulator's world, respectively. Thus, according to Lemma 7, the simulator's output is equal to the real output.

(3) Whenever $\textsc{UnweightedSF}_S$ is called on edges $(E_{c^*})_{=w}$ (with $c^* = \bigcup_{i \in [k]} c_i$) in the real execution, its output M is added to the final return value of Protocol 15. This means that $M \subseteq F$.

Let the simulator's output of this call to $\textsc{UnweightedSF}$ be M'. We have $M' \subseteq M$, as no edge $e \in F \setminus M$ can connect two different components from $\{c_0, \ldots, c_{k-1}\}$ (since $\{c_0, \ldots, c_{k-1}\}$ are already pairwise connected by M, and any other edge would introduce a cycle). In addition, any edge in M must also be in M', otherwise M' is not spanning.

This shows that the simulator's output M' is equal to the real output M.

□

Chapter 10

Performance Analysis

The performance (in terms of efficiency, especially the number of communication rounds) of Protocol 15 highly depends on the structure of the graph. In this chapter, we give an analysis to find out in which cases the protocol performs better than its alternatives.

In Section 10.1, we calculate the number of communication rounds per protocol iteration and give examples which minimize or maximize the number of iterations. In order to compare our protocol with Laud's MSF protocol [25], Section 10.2 presents a brief analysis of the number of communication rounds required by their protocol. In Section 10.3, we then use a random graph generation algorithm to show that as long as the input graph is not explicitly constructed for the worst case of Protocol 15 and the number of duplicate edge weights is not extremely high, our protocol can achieve very good results.

10.1 Theoretical Analysis

We assume that Protocol 15 is split into two phases: (1) the merge phase where all the separately computable subtrees are found, and (2) the final step where all calls to UnweightedSF are run at the same time.

Each iteration in the first phase consists of two steps that require communication:

(1) For each $c \in C$, computing the Minimum weight of an edge incident to c. In order to do this, first the locally best values of each party need to be shared, then the actual Minimum protocol is executed, and finally the result needs to be revealed. In total, this takes $\lceil \log_2 B \rceil + 4$ rounds (which can be run in parallel for all $c \in C$), where B

is the bitlength of the edge weights. For example, when $B = 32$ is used, this step would take 9 communication rounds.

(2) For each $w \in W$, computing Connectivity on the partitioning $S_w \cup \{c_w^*\}$. The performance depends on the number of components k in $S_w \cup \{c_w^*\}$ (which can be at most one more than the number of components with w being their best incident weight). The required number of rounds is $\lceil \log_2(k-1) \rceil \cdot (\lceil \log_2 k \rceil + 1) + 3$, so e.g. 6 for $k = 3$, 11 for $k = 5$, or 45 for $k = 50$.

While all calls to Connectivity can be run simultaneously, they could take different numbers of rounds. Thus, the total number of rounds depends on the maximum k over all parallel runs.

The second protocol phase only consists of parallel calls to UnweightedSF. The number of communication rounds required for this phase depends solely on the size of the largest subgraph for which UnweightedSF is called.

10.1.1 Extreme Cases

The number of required communication rounds itself can be easily maximized by letting all edges in a connected graph have the same weight. While there would be only one full iteration of the first protocol phase, UnweightedSF will be called on the whole graph, therefore taking a linear number of rounds (in the number of vertices) with a relatively high constant factor.

It is more interesting to look at graphs with a low number of duplicate weights. In particular, in this section we present two examples that minimize and maximize the number of required rounds when no two edges have the same weight. Note that in the second phase, UnweightedSF will only be called on subgraphs of size 2, thus requiring a constant number of communication rounds that is independent of the number of vertices.

Figure 10.1 gives an example of a graph where the number of iterations is minimized: Only one full iteration is required in the first phase, because the optimization can be applied on the whole graph s.t. afterwards only one component remains.

Figure 10.2 on the other hand gives an example where the edge weights are still unique, however Protocol 15 requires a linear number of iterations because the optimization is not applicable at all.

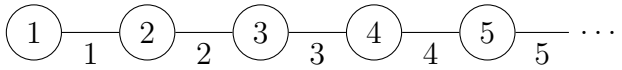


Figure 10.1: An example for a graph on which Protocol 15 takes a minimum number of rounds. In the first iteration, the best incident edge weight is computed for every vertex. This would be $w_{\{1\}} = w_{\{2\}} = 1$, and $w_{\{u\}} = u - 1$ for all $u \geq 3$. Then, after testing the connectivity, vertices 1 and 2 are merged together (and $\mathsf{MSF}(\{\{1\},\{2\}\}, E_{=1})$ is added to the global MSF). Afterwards, the optimization can infer that 3 must be connected to the new component $\{1,2\}$ using an edge of weight 2. Thus, $\mathsf{MSF}(\{\{1,2\},\{3\}\}, E_{=2})$ is computed and we get the new component $\{1,2,3\}$. This continues until all vertices are merged in the first iteration of the protocol. The total number of communication rounds consists of $\lceil \log_2 B \rceil + 4$ for computing the minimum weights, 6 for the connectivity tests, plus the number of rounds required for UnweightedSF on two components as described in Section 8.2.

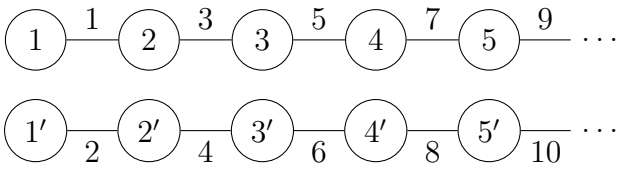


Figure 10.2: An example for a graph with unique edge weights on which Protocol 15 takes a maximum number of rounds. In the first iteration, the best incident edge weight is computed for every vertex. This would be $w_{\{1\}} = w_{\{2\}} = 1$, $w_{\{1'\}} = w_{\{2'\}} = 2$, and $w_{\{u\}} = 2u - 3$ and $w_{\{u'\}} = 2u - 2$ for all $u \geq 3$. Then, after testing the connectivity, vertices 1 and 2 are merged together (and $\mathsf{MSF}(\{\{1\},\{2\}\}, E_{=1})$ is added to the global MSF), and vertices $1'$ and $2'$ are merged together (and $\mathsf{MSF}(\{\{1'\},\{2'\}\}, E_{=2})$ is added to the global MSF). However, the optimization cannot be applied in this case: For any of the other components (i.e., $\{u\}$ or $\{u'\}$ for $u \geq 3$), it is unknown whether it is connected to $\{1,2\}$ or $\{1',2'\}$. Thus, the protocol needs to start the next iteration, in which again just two merges of two components each can take place. In summary, if there are $2n$ vertices in the beginning, then $n-1$ full iterations need to be executed, each of them using $\lceil \log_2 B \rceil + 4$ rounds for the first step, and 6 rounds for testing the connectivity. The number of full protocol iterations is maximized: the termination condition at the end of the proof of Theorem 9 shows that at least one merge for the smallest overall weight w' happens in each iteration. If there is no second merge in the same iteration, then the optimization could be applied to the smallest w with with $s_w \neq \emptyset$, which results in another merge step for a total of at least two merges per iteration.

10.2 Analysis of Laud's protocol

Laud has described a way to securely compute a minimum spanning forest in the semi-honest model for 3-party computation [25], and provided an implementation in the Sharemind framework [6]. It is based on Awerbuch and Shiloach's MSF algorithm [2], which modifies Borůvka's algorithm [8] in such a way that no union-find operations with unpredictable running time need to be performed. The algorithm is highly parallelizable and requires only $O(\log_2 |V|)$ iterations in total when using enough processors.

Laud's protocol does not perform any reveal steps during the execution (other than values required for sorting, but they do not yield any information about the input). Instead, it takes secret-shares of the complete graph (i.e., a list of secret-shared edges), and it returns a list of secret-shared edges. This problem is stronger than the one we are looking at (Functionality 1): revealing any values correlated to the input is not allowed, as a simulator is not able to see the produced MSF, but only its secret-shares.

In addition, Laud's protocol assumes that all edge weights are unique. However, due to the fact that no reveal steps take place, it is still possible to use it in order to simulate Functionality 1 in a simple way: A secret-shared tie-breaker permutation π for the set of edges is generated either explicitly (i.e., take an array with numbers $0, \ldots, |E| - 1$ and shuffle it), or in a more efficient way with small probability of error (i.e., generate a random secondary weight for each edge, s.t. the probability of generating two times the same weight is negligible). Supplementing each edge weight by this tie-breaker yields the option to compare two edges by their new weight for the cost of just one secret integer comparison (provided that its bitlength is large enough to hold both the original weight and the additional tie-breaker).

The fact that this protocol is based on arithmetic secret sharing and its framework has only been implemented for 3-party computation makes it impossible to compare it experimentally with our 2-party protocol based on binary secret sharing. However, in this section we attempt to give a loose lower bound on the number of communication rounds required by Laud's protocol [25]. This will show that, even though Laud's protocol has a poly-logarithmic running time independent of the structure of the graph, the involved constant factors are so large that it is often worth it to use Protocol 15 instead.

10.2.1 Reading and Writing

Laud's protocol makes use of parallel writing and reading, i.e., it stores a shared memory that can be efficiently read from and written to for a batch of requests [25]. The reading

and writing procedures are based on the ability to securely *shuffle* an array and to securely *sort* an array.

There are many ways to shuffle an array in a 3-party computation setting [26]. Even though there are constant-round shuffle protocols, they usually require large total communication costs (e.g. $O(n^2)$ when using permutation matrices). Laud instead suggests applying Waksman networks [35]. Each party could generate their own random permutation, and then propagate the given array through a Waksman network in order to apply the permutation. If this is done (sequentially) for all parties, a uniformly random permutation was applied, and no party has any knowledge about how it looks like. A plain Waksman network would require roughly $2\log_2 n$ rounds, and repeating this for all involved parties makes the constant factor even higher. However, for our comparison, we simply make the conservative assumption that shuffling a length-n array costs $\lfloor \log_2 n \rfloor$ rounds, as a variety of optimizations could be applied. We assume that shuffling, i.e., applying a uniformly random permutation, has the same costs as applying a secret-shared permutation (e.g., a secret permutation that is split into the composition of exactly one permutation known by each party) to a secret-shared array.

Sorting an array of length n can be done by randomly permuting it, and then applying a common sorting algorithm (e.g. QuickSort) [18]. Any comparisons in this sorting algorithm can be revealed to all parties as the shuffle step guarantees that the array ordering looks completely random. If r is the number of rounds it takes to compare two secret-shared numbers, then the sort step would take at least $\lfloor \log_2 n \rfloor (r+1)$ rounds: Every iteration in QuickSort (including the subsequent reveal step) requires $(r+1)$ rounds, and there are at least $\lfloor \log_2 n \rfloor$ such iterations. If one comparison of two secret-shared numbers takes 3 rounds [10], then combining all of this means that one sort operation takes $5\lfloor \log_2 n \rfloor$ rounds.

Assume that there is a secret-shared array of length m, and the protocol needs to perform n read accesses, given by a length-n array of secret-shared indices in the range $[0, m-1)$. Laud's protocol splits this process into two steps: The first one creates a secret permutation based on the access indices. In the second step, this permutation is applied to the original array containing the data. This procedure implies that the permutation created in the first step can be re-used if the protocol requires reading values from the same indices in a different array. Our lower bound yields $5\lfloor \log_2 n \rfloor$ rounds for the preparation step, and $2\lfloor \log_2 n \rfloor$ rounds (because two permutations need to be applied) for the second step. Note that for simplicity, we just use n instead of $n+m$ in these round complexities.

The write operation requires the ability to sort a binary array (that contains only 0s and 1s) which is stable w.r.t. the 0s, and which may reveal the number of 0s. This is done by Laud with one shuffle and some additional operations that we may ignore (because they

require a constant amount of rounds), thus $\lfloor \log_2 n \rfloor$ is a lower bound for this operation.

The write operation, similarly to the read operation, can be split into a preparation and an application step. However, the preparation step of a write operation requires more work: besides from sorting one array, one binary array sort and one permutation step need to be applied. Thus, at least $7\lfloor \log_2 n \rfloor$ rounds are required for the first step. Applying the actual write operation requires applying only two permutations, which takes a total of $2\lfloor \log_2 n \rfloor$ rounds.

10.2.2 The complete MSF protocol

Laud's MSF protocol runs for $\lfloor \log_{3/2} n \rfloor$ iterations, each of them containing many steps that need to be run sequentially.

Checking whether the current union-find data structure contains a star for a given vertex requires (in total) one read preparation step, two read perform steps, one write preparation step, and one write perform step, in addition to some low-level computations with constant round complexity. Thus, at least $18\lfloor \log_2 n \rfloor$ rounds are needed.

One iteration of the full protocol requires checking for stars as described, and additionally performing two read preparation steps, three read perform steps, one write prepare step, and one write perform step. This sums up to a lower bound of $43\lfloor \log_2 n \rfloor$ rounds.

In total, $\lfloor \log_{3/2} n \rfloor$ of these iterations need to be performed. This gives us a conservative lower bound of $\approx 1.7 \cdot 43 \cdot \lfloor \log_2 n \rfloor^2 > 73 \cdot \lfloor \log_2 n \rfloor^2$ communication rounds required by Laud's algorithm.

10.3 Comparison

We implemented a program in such a way that for any given graph, we can calculate the number of communication rounds that the real protocol would take. This program simply simulates all the steps in Protocol 15 and computes the number of rounds required for each call to a sub-protocol (which depends on the number of rounds calculated for each sub-protocol in the previous sections of this work). Note that for our protocol, it does not matter which party each edge belongs to, because the number of communication rounds and the total communication cost does not depend on this.

We run this on a variety of graphs and compare the results with Laud's algorithm. In order to show where the bottleneck of our protocol lies, we report other values in addition

to the number of communication rounds (split into phase one and two): For phase 1 of the protocol, we calculate the number of full iterations and the maximum number of components on which Protocol 11 (CONNECTIVITY) has been called. For the second phase, we report the maximum subgraph size on which UNWEIGHTEDSF is called.

We perform comparisons of two types of graphs: (1) randomly generated graphs, and (2) Traveling Salesman Problem graphs [31].

Randomly generated graphs We generate random graphs by sampling two uniformly random and distinct endpoints for each edge. Weights are either generated as a uniformly random integer weight in the range $[0, w)$ for a fixed upper bound w, or we make the edge weights completely unique by assigning indices $[0, |E|)$ to the edges and then shuffling them.

We generate graphs for $|V| = 10, 10^2, 10^3, 10^4, 10^5$, and always set $|E| := 10|V|$. This high number of edges ensures that most vertices in the graph are connected to each other. We repeat each configuration 5 times and report the mean values and standard deviations in Table 10.1. We attempt to choose values for w such that we can see in which cases our protocol achieves a better number of rounds than our lower bound for Laud's protocol.

Traveling Salesman Problem In order to show that our protocol also works well on "real" graphs, we also take a few randomly selected TSP graphs from the collection TSPLIB [31]. This is because one important application of MSFs is to approximate the Traveling Salesman Problem. We select the graph in such a way that the number of vertices ranges between 52 and 1748. As all of the graphs are complete, the number of edges is roughly the square of the number of vertices, and therefore higher number of vertices would result in rather infeasible problem instances. The results are presented in Table 10.2.

10.3.1 Results

All of our experiments assume bitlength $B = 32$, and $\lambda = 32$ repetitions for securely generating random numbers with Protocol 8.

We can clearly see from Table 10.1 that the number of rounds increases when w, the upper bound on edge weights, is small. It is minimized when the edge weights are fully unique. The reason for this is that a small bound of w means that many edges may have the same weight, and therefore UNWEIGHTEDSF needs to be run on very large subgraphs in the second phase. For example, on 10^5 vertices and 10^6 edges, our protocol requires

$\|V\|$	$\|E\|$	w	Phase 1 Iterations	Phase 1 Protocol 11	Phase 1 **Rounds**	Phase 2 Protocol 12	Phase 2 **Rounds**	[25]
10	10^2	5	1	11	32	10	699	657
		10	1.4 ± 0.5	9.8 ± 0.4	39 ± 9.7	7.4 ± 2.3	469.8 ± 213.9	
		100	2.6 ± 0.9	5.2 ± 1.5	45.8 ± 16.0	3.2 ± 0.4	112.6 ± 30.4	
		unique	2 ± 0.7	3	30 ± 10.6	2	27	
10^2	10^3	20	2.6 ± 0.5	63.2 ± 8.0	131 ± 20.7	34.4 ± 4.6	3562 ± 567.0	2628
		50	4 ± 0.7	34.2 ± 5.3	160 ± 21.6	15.2 ± 2.6	1261.2 ± 288.8	
		100	5.6 ± 1.3	21.6 ± 0.5	175 ± 35.4	7.6 ± 0.9	477.6 ± 85.9	
		1000	7.4 ± 1.1	7 ± 0.7	142 ± 22.6	3.8 ± 0.8	157.4 ± 64.3	
		unique	8.6 ± 1.1	3	129 ± 17.1	2	27	
10^3	10^4	100	7.4 ± 0.5	185.4 ± 20.4	479.6 ± 41.9	62.4 ± 7.2	7151.6 ± 1024.1	5913
		200	10.2 ± 1.3	96.4 ± 7.2	527.6 ± 58.2	32.4 ± 2.4	3258.2 ± 371.8	
		500	15.6 ± 1.1	48 ± 2.5	563.6 ± 39.3	15.4 ± 1.3	1248.2 ± 163.0	
		1000	17.8 ± 3.7	29 ± 2.3	536.4 ± 87.2	11.8 ± 1.3	872.6 ± 144.0	
		10000	20.2 ± 2.8	10 ± 1.7	386.6 ± 43.5	4.2 ± 0.4	184.6 ± 39.4	
		unique	22 ± 4.8	3	330 ± 72.7	2	27	
10^4	10^5	500	18.4 ± 0.5	414.2 ± 4.1	1345.4 ± 29.6	116.8 ± 6.5	14926.2 ± 833.0	12337
		1000	25.2 ± 2.0	209.4 ± 16.9	1486.2 ± 101.9	63.6 ± 4.0	7374.8 ± 699.2	
		5000	35.4 ± 1.7	58.4 ± 4.2	1202.2 ± 38.2	20 ± 1.9	1779 ± 175.9	
		10000	37.8 ± 2.9	35.8 ± 3.1	1076 ± 61.8	12.4 ± 0.5	965 ± 46.6	
		100000	39.4 ± 5.7	11.6 ± 0.9	771.6 ± 99.9	5.2 ± 0.4	267.8 ± 28.6	
		unique	48 ± 10.3	3	720 ± 155.2	2	27	
10^5	10^6	5000	53.2 ± 4.7	439.4 ± 9.8	3549.8 ± 334.2	120.4 ± 2.1	15390.6 ± 267.5	18688
		20000	91.2 ± 8.7	130.2 ± 3.8	3560.6 ± 271.2	42.6 ± 4.2	4550.8 ± 583.1	
		50000	97.8 ± 15.9	64.2 ± 2.4	2928.6 ± 419.5	22.6 ± 2.1	2088.8 ± 274.6	
		100000	101.6 ± 22.3	41.4 ± 2.6	2612.2 ± 453.8	16.2 ± 1.6	1374.4 ± 216.3	
		1000000	97.8 ± 9.9	14.2 ± 0.8	1827.6 ± 145.9	6.2 ± 0.4	340 ± 47.0	
		unique	91.8 ± 15.1	3	1377 ± 226.8	2	27	

Table 10.1: This table displays the results produced by Protocol 15 on random graphs. The leftmost columns denote the number of vertices, the number of edges, and the upper bound for the randomly generated weights. For phase 1 of the protocol, we report the number of full iterations, the maximum number of components on which Protocol 11 (Connectivity) was called, and the resulting number of communication rounds. For the second phase, we report the maximum subgraph size on which UnweightedSF is called and the corresponding number of communication rounds. In the rightmost column we state our lower bound of the number of rounds that Laud's protocol would take on the given number of vertices. As we generate 5 independent graphs in each setting, we report both the mean values and the standard deviations in this table.

			Phase 1			Phase 2		
$\|V\|$	$\|E\|$	Graph	Iterations	Protocol 11	**Rounds**	Protocol 12	**Rounds**	[25]
52	1326	berlin52	7	6	117	3	99	1825
180	16110	brg180	3	181	184	15	1186	3577
666	221445	gr666	28	9	490	3	99	5913
1379	950131	nrw1379	19	76	734	13	1016	7300
1655	1368685	d1655	10	1297	557	64	7235	7300
1748	1526878	vm1748	32	583	943	14	1101	7300

Table 10.2: This table shows the results produced by Protocol 15 on TSP graphs. The reported numbers follow the same format as in Table 10.1, with the only difference that we do not have a fixed upper bound on the weight, but we just take the graph with name stated in the third column from TSPLIB [31].

about 8000 rounds if $w = 20\,000$, but nearly 20 000 rounds if $w = 5000$. Laud's protocol has a constant performance independent of the edge weights, and our lower bound indicates that it would require at least 18 000 rounds.

Note that in contrast to the second phase, the number of iterations in the first phase increases for larger w. However, Connectivity needs to be run for a smaller number of components, which in turn results in a decrease of the number of communication rounds, as shown in Table 10.1. Still, its effect on the overall number of rounds is much smaller than the additional number of rounds required for larger subgraphs in the second phase.

Table 10.2 shows the results on the selected TSP graphs. Note that, in terms of rounds, our protocol performs better on all of them, except for d1655. The reason for this is that on d1655, UnweightedSF needs to be called on a very large subgraph consisting of 64 vertices, which requires a large number of communication rounds.

Taking a closer look, on vm1748 and brg180, Connectivity is called for a rather high number of components (583 and 181, respectively). Because the total communication cost of this sub-protocol is cubic, these instances might also be infeasible for our protocol, depending on the available bandwidth between the two parties. For all other tested TSP graphs (berlin52, gr666, and nrw1379), the number of duplicate weights are relatively small, and accordingly the subgraph size is also small (at most 13 on nrw1379), and therefore our protocol performs much better than Laud's protocol.

Chapter 11

Conclusion

In this work, we have presented several novel protocols for computing a Minimum Spanning Forests for the semi-honest security model in a two-party computation setting. Our objective was to minimize the number of communication rounds, so that the protocol is usable in any network setting, even when the latency is high (for example due to a large geographical distance).

First, we have seen a very simple protocol based on Borůvka's algorithm that only has a logarithmic worst-case round complexity. However, it is not secure anymore if a random tie-breaker for edges with equal weights needs to be used, and this tie-breaker is published or the parties can infer any information about it.

As a result, we proposed a protocol that can compute a Random Minimum Spanning Forest without any kind of leakage. However, the required number of communication rounds highly depends on the graph structure and its weights. Whenever the number of duplicate weights is rather small, and the graph does not follow an artificially constructed worst-case, then the number of iterations is very low and our protocol performs better than alternatives.

In future work, it would be interesting to further optimize this process for graphs with structures that lead to a high number of rounds in our current protocol. One main point of improvement that the protocol would benefit from is reducing the total amount of communication required for testing connectivity on a subgraph with many components. Additionally, constructing a new MSF protocol with a sublinear number of communication rounds for general unweighted graphs (e.g. by adapting prior fully secure protocols without any reveal step) is a promising direction for further research. This could lead to better performance of the second phase of our protocol, which is currently one of the main bottlenecks for graphs with many duplicate weights.

References

[1] M. Anagreh, E. Vainikko, and P. Laud, "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*, P. Mori, G. Lenzini, and S. Furnell, Eds., SCITEPRESS, 2021, pp. 181–190.

[2] B. Awerbuch and Y. Shiloach, "New connectivity and msf algorithms for shuffle-exchange network and pram", *IEEE Transactions on Computers*, vol. 36, no. 10, pp. 1258–1263, 1987.

[3] D. Beaver, "Efficient multiparty protocols using circuit randomization", in *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, J. Feigenbaum, Ed., ser. Lecture Notes in Computer Science, vol. 576, Springer, 1991, pp. 420–432.

[4] M. Ben-Or, S. Goldwasser, and A. Wigderson, "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*, J. Simon, Ed., ACM, 1988, pp. 1–10.

[5] M. Blanton, A. Steele, and M. Aliasgari, "Data-oblivious graph algorithms for secure computation and outsourcing", in *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, K. Chen, Q. Xie, W. Qiu, N. Li, and W. Tzeng, Eds., ACM, 2013, pp. 207–218.

[6] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations", in *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, S. Jajodia and J. López, Eds., ser. Lecture Notes in Computer Science, vol. 5283, Springer, 2008, pp. 192–206.

[7] J. Böhler and F. Kerschbaum, "Secure sublinear time differentially private median computation", in *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*, The Internet Society, 2020.

[8] O. Borůvka, "O jistém problému minimálním", in *Práce Moravské přírodovědecké společnosti*, vol. 3, 1926, pp. 37–58.

[9] J. Brickell and V. Shmatikov, "Privacy-preserving graph algorithms in the semi-honest model", in *Advances in Cryptology - ASIACRYPT 2005, 11th International Conference on the Theory and Application of Cryptology and Information Security, Chennai, India, December 4-8, 2005, Proceedings*, B. K. Roy, Ed., ser. Lecture Notes in Computer Science, vol. 3788, Springer, 2005, pp. 236–252.

[10] O. Catrina and S. d. Hoogh, "Improved primitives for secure multiparty integer computation", in *International Conference on Security and Cryptography for Networks*, Springer, 2010, pp. 182–199.

[11] D. Demmler, T. Schneider, and M. Zohner, "ABY - A framework for efficient mixed-protocol secure two-party computation", in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, The Internet Society, 2015.

[12] D. Evans, V. Kolesnikov, and M. Rosulek, "A pragmatic introduction to secure multi-party computation", *Found. Trends Priv. Secur.*, vol. 2, no. 2-3, pp. 70–246, 2018.

[13] A. M. Frieze, "On the value of a random minimum spanning tree problem", *Discret. Appl. Math.*, vol. 10, no. 1, pp. 47–56, 1985.

[14] O. Goldreich, "Secure multi-party computation", *Manuscript. Preliminary version*, vol. 78, p. 110, 1998.

[15] O. Goldreich, S. Micali, and A. Wigderson, "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*, A. V. Aho, Ed., ACM, 1987, pp. 218–229.

[16] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams", *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[17] R. L. Graham and P. Hell, "On the history of the minimum spanning tree problem", *Annals of the History of Computing*, vol. 7, no. 1, pp. 43–57, 1985.

[18] K. Hamada, R. Kikuchi, D. Ikarashi, K. Chida, and K. Takahashi, "Practically efficient multi-party sorting protocols from comparison sort algorithms", in *International Conference on Information Security and Cryptology*, Springer, 2012, pp. 202–216.

[19] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic, "Sok: General purpose compilers for secure multi-party computation", in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, IEEE, 2019, pp. 1220–1237.

[20] W. D. Hillis and G. L. Steele Jr, "Data parallel algorithms", *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, 1986.

[21] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank, "Extending oblivious transfers efficiently", in *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, D. Boneh, Ed., ser. Lecture Notes in Computer Science, vol. 2729, Springer, 2003, pp. 145–161.

[22] M. Keller, "MP-SPDZ: A versatile framework for multi-party computation", in *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, J. Ligatti, X. Ou, J. Katz, and G. Vigna, Eds., ACM, 2020, pp. 1575–1590.

[23] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem", *Proceedings of the American Mathematical society*, vol. 7, no. 1, pp. 48–50, 1956.

[24] R. E. Ladner and M. J. Fischer, "Parallel prefix computation", *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 831–838, 1980.

[25] P. Laud, "Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees", *Proc. Priv. Enhancing Technol.*, vol. 2015, no. 2, pp. 188–205, 2015.

[26] S. Laur, J. Willemson, and B. Zhang, "Round-efficient oblivious database manipulation", in *International Conference on Information Security*, Springer, 2011, pp. 262–277.

[27] Y. Lindell, "How to simulate it - A tutorial on the simulation proof technique", in *Tutorials on the Foundations of Cryptography*, Y. Lindell, Ed., Springer International Publishing, 2017, pp. 277–346.

[28] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation", in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, 2015, pp. 359–376.

[29] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi, "Graphsc: Parallel secure computation made easy", in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, IEEE Computer Society, 2015, pp. 377–394.

[30] R. C. Prim, "Shortest connection networks and some generalizations", *The Bell System Technical Journal*, vol. 36, no. 6, pp. 1389–1401, 1957.

[31] G. Reinelt, "TSPLIB - A traveling salesman problem library", *INFORMS J. Comput.*, vol. 3, no. 4, pp. 376–384, 1991.

[32] T. Schneider and M. Zohner, "GMW vs. yao? efficient secure two-party computation with low depth circuits", in *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers*, A. Sadeghi, Ed., ser. Lecture Notes in Computer Science, vol. 7859, Springer, 2013, pp. 275–292.

[33] A. Shamir, "How to share a secret", *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[34] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol", in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, A. Sadeghi, V. D. Gligor, and M. Yung, Eds., ACM, 2013, pp. 299–310.

[35] A. Waksman, "A permutation network", *Journal of the ACM (JACM)*, vol. 15, no. 1, pp. 159–163, 1968.

[36] D. B. Wilson, "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*, G. L. Miller, Ed., ACM, 1996, pp. 296–303.

[37] A. C. Yao, "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, 1982, pp. 160–164.