A Parallel Privacy-Preserving Shortest Path Protocol from a Path Algebra Problem

Mohammad Anagreh[1] and Peeter Laud[2]

[1] Tartu University, Inst. of Computer Science, Narva mnt. 18, 51009 Tartu, Estonia
mohammad.anagreh@ut.ee
[2] Cybernetica AS, Narva mnt. 20, 51009 Tartu, Estonia
peeter.laud@cyber.ee

Abstract. In this paper, we present a secure multiparty computation (SMC) protocol for single-source shortest distances (SSSD) in undirected graphs, where the location of edges is public, but their length is private. The protocol works in the Arithmetic Black Box (ABB) model on top of the separator tree of the graph, achieving good time complexity if the subgraphs of the graph have small separators (which is the case for e.g. planar graphs); the achievable parallelism is significantly higher than that of classical SSSD algorithms implemented on top of an ABB.
We implement our protocol on top of the Sharemind MPC platform, and perform extensive benchmarking over different network environments. We compare our algorithm against the baseline picked from classical algorithms — privacy-preserving Bellman-Ford algorithm (with public edges).

Keywords: Secure multiparty computation · Path Algebra · Semiring framework · single-instruction-multiple-data · Bellman-Ford · Sharemind

1 Introduction

Graph algorithms are the foundation of many computer science applications such as navigation systems, community detection, supply chain networks [32, 38,39], hyperspectral imaging [35], and sparse linear solvers. Privacy-preserving parallel algorithms are needed to expedite the processing of large private data sets for graph algorithms and meet high-end computational demands. Constructing real-world privacy applications based on secure multiparty computation is challenging due to the round complexity of the computation parties of SMC protocol [11,23,24]. The round complexity problem of SMC protocol can be solved using parallel computing [10,14].

Single-Instruction-Multiple-Data (SIMD) a principle for parallel computations [19]. Recently, SIMD principles have been used to reduce the round complexities in many privacy-preserving graph algorithms, including minimum spanning tree [4,25] and shortest path [2,3,5]. These privacy-preserving graph protocols are constructed on top of SMC protocols, and they are capable to process sizeable private data sets, where both the location and length of edges are

private. The construction of these protocols follows the classical graph algorithms [15], storing the intermediate values privately, invoking SMC protocols for the computational steps in these algorithms, and attempting to parallelize the computations as much as possible.

In this paper we show that certain other SSSD algorithms may be even more suitable for conversion into SMC protocols. Considering the *Parallel RAM* (PRAM) model of execution, Pan and Reif [29,31] proposed a parallel algorithm for the Algebraic Path Computation (APC). Their algorithm assumes the availability of the *separator tree* of the graph and computes a recursive factorization of the graph's adjacency matrix on its basis [31], with the number of steps and the parallel complexity depending on the height of the tree and the size of separators. We also assume that the separator tree is among the public inputs for the SMC protocol, and show how to privately execute Pan and Reif's algorithm. Our empirical evaluation shows that for graphs with "good" separator trees, including planar graphs, our protocol may be up to two orders of magnitude faster than protocols based on classical SSSD algorithms.

The availability of the separator tree implies that the locations of the edges of the graph have to be public (this is accounted for in our empirical comparisons), only their lengths are private. Privately computing SSSD in this setting can still be relevant for a number of applications. E.g. private SSSD in city streets with public layouts has been tackled using either SMC [37] or differential privacy [34]. However, SMC protocols based on classical SSSD algorithms either do not benefit from the public end-points of edges at all [17], or benefit only slightly [6].

Our contributions. In this paper, we present the following:

- The first privacy-preserving parallel computation protocol of algebraic shortest path. The protocol uses the sparse representation of an (adjacency) matrix, where the locations of edges are public, while their lengths are private. We propose suitable data structures and normalizations for this task.
- Implementations (on top of the Sharemind MPC platform [7,8]) of the algebraic shortest path protocol, and an optimized privacy-preserving Bellman-Ford protocol, with public locations and private lengths of edges. Benchmarking results for both implementations for various sizes of graphs, and different network environments.

2 Preliminaries

2.1 Secure Multiparty Computation

Secure multiparty computation (SMC) is a cryptographic technique, allowing a number of parties each give input to a pre-agreed functionality F, and learn the input meant for this party, such that each party (or a *tolerable* coalition of parties) will learn nothing besides their own input and output. There exist a number of different approaches for constructing SMC protocols, including garbled circuits [40], homomorphic encryption [16,21], or secret sharing [12,20], and offering security either against passive or active adversaries. These approaches

typically include steps for entering a value into the computation in a privacy-preserving manner, for performing simple arithmetic operations (e.g. addition and multiplication in a finite field or ring) with private values present in the computation, and for opening a private value to a party upon the agreement of sufficiently many other parties. These steps, that constitute protocols by themselves, can be combined relatively freely. Hence, if the functionality F has been presented as an arithmetic circuit, then these protocols for input/output and arithmetic operations can be combined to yield a protocol for F.

Availability of such compositions leads to the typical abstraction of SMC in privacy-preserving applications — the *Arithmetic Black Box* (ABB) [16, 26]. An ABB is an ideal functionality in the *Universal Composability* [13] framework. This framework considers a set $\mathcal{T}$ of interacting Turing machines [22], executing a protocol Π. Beside the set of machines $\mathcal{T}$, there is also another Turing machine — the *adversary* that can interfere with Π by sending to machines in $\mathcal{T}$ certain commands that have been defined in the adversarial API's of these machines. The set of the machines also includes the *environment* that interacts with machines in $\mathcal{T}$ and the adversary over a well-defined API. Given two sets of machines $\mathcal{T}$ and $\mathcal{T}'$ implementing the same API towards the environment, we say that $\mathcal{T}$ is *at least as secure as* $\mathcal{T}'$, if for any possible adversary $\mathbf{A}$ targeting $\mathcal{T}$ (i.e. its adversarial API), there exists an adversary $\mathbf{S}$ targeting $\mathcal{T}'$, such that the environment cannot distinguish whether it is executing with $\mathcal{T}$ and $\mathbf{A}$, or with $\mathcal{T}'$ and $\mathbf{S}$. This notion is composable: if additionally $\mathcal{T} = \mathcal{T}_0 \cup \{\Xi\}$ for a Turing machine Ξ, and a set of machines $\mathcal{U}$ is at last as secure as $\{\Xi\}$, then $\mathcal{T}_0 \cup \mathcal{U}$ is at least as secure as $\mathcal{T}'$. Often, we say that Ξ is the ideal functionality for the corresponding real functionality $\mathcal{U}$ that implements it.

The ABB functionality is represented by a Turing machine $\mathcal{F}_{\mathcal{ABB}}$ that allows the environment representing all parties of a multiparty application to perform private computations. If one of the parties sends the command (store, v) to the ABB, where v is a value from one of the rings that the ABB supports, then it creates a new *handle* h, stores the pair (h, v), and sends h back to all parties. If all (or sufficiently many) parties send the command $(\mathsf{perform}, op, h_1, \ldots, h_k)$ to the ABB, where op is one of the supported operations and $h_1, \ldots, h_k$ are existing handles, then the ABB looks up the stored pairs $(h_1, v_1), \ldots, (h_k, v_k)$, computes $v = op(v_1, \ldots, v_k)$, creates a new handle h, stores (h, v), and sends h back to all parties. If all (or sufficiently many) parties send the command $(\mathsf{declassify}, h)$, then ABB looks up (h, v) and sends v back to all parties. A secure application that makes use of the ABB remains secure if $\mathcal{F}_{\mathcal{ABB}}$ is replaced with a set of Turing machines that securely implement the ABB, i.e. run secure multiparty computation protocols. Note that if we want to compute a function F with the help of an ABB, and if the ABB only declassifies the end result of F, then the resulting protocol is trivially private [26].

In the following, a value v stored in the ABB and accessed through a handle is denoted by $[\![v]\!]$. Similarly, $[\![\vec{v}]\!]$ denotes a vector of values, and $[\![\mathbf{V}]\!]$ a matrix of values stored in the ABB. We use the notation $[\![u]\!] + [\![v]\!]$ [resp. $\mathsf{min}([\![u]\!], [\![v]\!])$] to denote that the addition [resp. minimum] operation is being invoked on the

values $[\![u]\!]$ and $[\![v]\!]$; the result of this operation is again stored in the ABB. We extend this notation pointwise to vectors and matrices.

The *cost* of the operations of the ABB depends on the implementation of $\mathcal{F}_{\mathcal{ABB}}$. If Sharemind has been used as the implementation, then the addition is a free operation (i.e. it requires no communication between parties), and minimum requires a constant amount of bits to be exchanged in a constant number of rounds. Hence the bandwidth cost of $\mathsf{min}([\![u]\!], [\![\vec{v}]\!])$ is linear in the length of $\vec{u}$ and $\vec{v}$, while the round complexity is logarithmic in their length. In the following descriptions of algorithms built on top of the ABB, we have to be explicit in stating, which operations can or cannot be performed in parallel. For loops, we write **forall** to denote that all iterations take place in parallel; we write **for** to state that the loop is sequential.

2.2 Graphs, separators, semirings, and algebra path problem

A *graph* $G = (V, E)$ is a mathematical structure consisting of a set V of *vertices* that are connected by *edges* from a set $E \subseteq V \times V$. The edges between vertices may have *lengths* assigned to them; these are given by a function $w : E \to \mathbb{Z}$. A graph may be directed or undirected; in the latter case, E is symmetric. If $V' \subseteq V$, then we let $G[V']$ denote the *induced subgraph* $(V', E \cap V' \times V')$.

A graph $G = (V, E)$ can be represented in computer memory in different ways. The *dense* representation of G is the *adjacency matrix* — a $|V| \times |V|$ matrix over $\mathbb{Z} \cup \{\infty\}$, where the entry at u-th row and v-th column is $w(u, v)$. On the other hand, the *adjacency list representation* gives for each vertex $u \in V$ the list of pairs $(v_1, w_1), \ldots, (v_k, w_k)$, where $(u, v_1), \ldots, (u, v_k)$ are all edges in G that start in u, and $w_i = w(u, v_i)$; we call such representations *sparse*. If $|E| \ll |V|^2$, then sparse representation takes up less space than dense representation and the algorithms working on sparse representation may be faster [9].

We call a graph (actually, an infinite family of graphs) *sparse* if its number of edges is *proportional* to its number of vertices, $|E| = O(|V|)$; otherwise we call it *dense*. A graph is *planar* if it can be drawn a plane without crossing the edges outside vertices. If G is planar, then $|E| \leq 3|V| - 6$ according to Euler's formula for the number of vertices, edges and faces of its drawing [36].

A *separation* of the graph $G = (V, E)$ is a partition of its vertices $V = V_1 \dot\cup S \dot\cup V_2$, such that any path from a vertex in V_1 to a vertex in V_2 must pass through a vertex in S (called a *separator*). It is known [27] that planar graphs have separations where $|S| = O(\sqrt{|V|})$ and $|V_1|, |V_2| \leq 2|V|/3$. A *separator tree* of G is either a single node containing $(\emptyset, V, \emptyset)$; or the root node (V_1, S, V_2) for some separation of G, and its two subtrees — separator trees for $G[V_1 \cup S]$ and $G[V_2 \cup S]$. Planar graphs thus have separator trees of height $O(\log |V|)$.

A *(commutative) semiring* is an algebraic structure S with two binary operations $\oplus$ and $\otimes$, where both are associative and commutative, have unit elements $\textcircled{0}$ and $\textcircled{1}$, where $\otimes$ distributes over $\oplus$, and where $a \otimes \textcircled{0} = \textcircled{0}$ for all $a \in S$. We overload $\oplus$ and $\otimes$ to also denote addition and multiplication of matrices with elements from S; the multiplication may also be denoted by juxtaposition. Given matrices $\mathbf{A} \in S^{m \times n}$ and $\mathbf{X} \in S^{m \times n}$, the *algebra path problem* is to find a matrix

$\mathbf{Y} \in S^{m \times n}$, such that $\mathbf{Y} = \mathbf{X} \oplus \mathbf{YA}$. Let $\mathbf{I} \in S^{n \times n}$ be the identity matrix. If $\mathbf{A}$ has a *quasi-inverse*, i.e. a matrix $\mathbf{A}^* \in S^{n \times n}$, such that $\mathbf{I} \oplus \mathbf{AA}^* = \mathbf{I} \oplus \mathbf{A}^*\mathbf{A} = \mathbf{A}^*$, then $\mathbf{Y} = \mathbf{XA}^*$ is a possible solution to the algebra path problem.

Algebra path problem generalizes a number of graph-theoretic tasks. Let $G = (V, E)$ be a graph and let $t \in V$, and let $S = \mathbb{N} \cup \{\infty\}$, $\oplus$ be the minimum, $\otimes$ be the addition, $\textcircled{0} = \infty$, $\textcircled{1} = 0$, $n = |V|$, $m = 1$, $\mathbf{A}$ be the adjacency matrix of G, and $\vec{x} = \mathbf{X} \in S^{1 \times n}$ be the t-th unit vector (i.e. $v_t = \textcircled{1} = 0$ and $v_j = \textcircled{0} = \infty$ for $j \neq t$). Then $\vec{y} = \mathbf{Y}$ is the vector of shortest distances from the t-th vertex [28]. Other instantiations of the semiring and $\mathbf{X}$ give solutions to other problems [30].

Having the semiring instantiated as in the previous paragraph, the quasi-inverse of $\mathbf{A} \in S^{n \times n}$ is defined; it is equal to $(\mathbf{I} \oplus \mathbf{A})^n$. If $\mathbf{A}$ is the adjacency matrix of some graph, then $\mathbf{A}^*$ is the matrix of shortest distances between the vertices of the same graph. Hence any all-pairs shortest distance (APSD) algorithm is suitable for computing $\mathbf{A}^*$; but it would be inefficient to use for the SSSD problem, particularly when $\mathbf{A}$ is a sparse(ly represented) matrix.

Given $\vec{x}$ and symmetric $\mathbf{A}$, Pan and Reif [30], proposed the following algorithm for computing $\vec{x} \otimes \mathbf{A}^*$ without ever materializing $\mathbf{A}^*$. Let $d \in \mathbb{N}$ and pick numbers $n = n_0 > n_1 > \cdots > n_d > 0$. Let $\mathbf{P} \in S^{n \times n}$ be a permutation matrix. Define matrix $\mathbf{A}_0 = \mathbf{PAP}^T$ (i.e. we permute the rows and columns of $\mathbf{A}$ in the same manner; this corresponds to reordering the vertices of G) and define the matrices $\mathbf{X}_h, \mathbf{Y}_h, \mathbf{Z}_h, \mathbf{A}_{h+1}$ (for $h \in \{0, \ldots, d-1\}$) by

$$\begin{bmatrix} \mathbf{X}_h & \mathbf{Y}_h^T \\ \mathbf{Y}_h & \mathbf{Z}_h \end{bmatrix} := \mathbf{A}_h \qquad \mathbf{A}_{h+1} := \mathbf{Z}_h \oplus \mathbf{Y}_h \mathbf{X}_h^* \mathbf{Y}_h^T \tag{1}$$

where $\mathbf{Z}_h, \mathbf{A}_{h+1} \in S^{n_{h+1} \times n_{h+1}}$; this also defines the sizes of $\mathbf{X}_h$ and $\mathbf{Y}_h$. Letting $\mathbf{I}$ and $\mathbf{O}$ denote identity and zero matrices of appropriate sizes, one can verify that the following identity holds:

$$\mathbf{A}_h^* = \begin{bmatrix} \mathbf{I} & \mathbf{X}_h^* \otimes \mathbf{Y}_h^T \\ \mathbf{O} & \mathbf{I} \end{bmatrix} \otimes \begin{bmatrix} \mathbf{X}_h^* & \mathbf{O} \\ \mathbf{O} & \mathbf{A}_{h+1}^* \end{bmatrix} \otimes \begin{bmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{Y}_h \otimes \mathbf{X}_h^* & \mathbf{I} \end{bmatrix} \quad . \tag{2}$$

We thus have an algorithm to compute $\vec{y} = \vec{x} \otimes \mathbf{A}^*$. Let $\vec{x}_\mathrm{P} = \vec{x} \otimes \mathbf{P}^T$. Let $h = 0$. Extract $\mathbf{X}_h, \mathbf{Y}_h, \mathbf{Z}_h$ from $\mathbf{A}_h$ and compute $\mathbf{X}_h^*$ (using any APSD algorithm), $\mathbf{Q}_h = \mathbf{Y}_h \otimes \mathbf{X}_h^*$ and $\mathbf{A}_{h+1}$. Note that $\mathbf{X}_h^* \otimes \mathbf{Y}_h^T = \mathbf{Q}_h^T$. Multiply $\vec{x}_\mathrm{P}$ with the first matrix in (2), then the result with the second matrix, and then the result with the third matrix, thus defining $\vec{y}_\mathrm{P} = \vec{x}_\mathrm{P} \otimes \mathbf{A}_0^*$. Finally remove the permutation, computing $\vec{y} = \vec{y}_\mathrm{P} \otimes (\mathbf{P}^T)^{-1}$. All computations are done with sparse matrices. Importantly, multiplication with the second matrix in (2) splits the current vector into two parts, where the left part is multiplied with $\mathbf{X}_h^*$, and the right part with $\mathbf{A}_{h+1}^*$ through a *recursive call*. The recursion stops by computing $\mathbf{A}_d^*$ directly (using any APSD algorithm).

Pan and Reif [30] show that if the choice of $\mathbf{P}$ and $n_0, \ldots, n_d$ is informed by a separator tree $\mathbf{T}$ of G with height $d = O(\log n)$ and separators of size $O(\sqrt{n})$, then, depending on how $\mathbf{X}_h^*$ and $\mathbf{A}_d^*$ are computed, the described algorithm requires either $O(\log^3 n)$ parallel time and $O(n^{3/2} \log n)$ work, or $O(\sqrt{n} \log n)$

parallel time and $O(n^{3/2})$ work. The time estimate follows directly from the parallel time complexity of the matrix operations, multiplied by the depth of the recursion. The work estimate follows from careful counting of elements in the sparse representations of matrices [31].

Pan and Reif [31, Sec. 7] describe, what kind of information is extracted from the separator tree $\mathbf{T}$. We refer to them for details, but let us describe the result. The main outcome is a list $\vec{\mathcal{L}} = (\mathcal{L}_0, \ldots, \mathcal{L}_d)$ of lists of lists of vertices of G, such that each $i \in V$ occurs in $\vec{\mathcal{L}}$ exactly once. The permutation matrix $\mathbf{P}$ must reorder the vertices so, that they appear in the same order as in flattened $\vec{\mathcal{L}}$. For $h \in \{0, \ldots, d\}$, the number n_h is equal to the number of vertices in $(\mathcal{L}_h, \mathcal{L}_{h+1}, \ldots, \mathcal{L}_d)$. Let $\vec{L} = (L_0, \ldots, L_d)$, where each L_h is the list of lengths of elements of $\mathcal{L}_h$ (note that elements of $\mathcal{L}_h$ are lists of vertices). In (1), $\mathbf{X}_h$ is going to be a block-diagonal matrix with the block sizes listed in L_h; this is used in the computation of $\mathbf{X}_h^*$.

3 Privacy-Preserving Algebraic Shortest Path Protocol

This section presents the privacy-preserving version of the algorithm decsribed above. We present the used data structures, the auxiliary functionalities, and the main computation.

Data structures. We mostly use the sparse representation of matrices. The representation $\langle\langle \mathbf{A} \rangle\rangle$ for a matrix $\mathbf{A} \in S^{m \times n}$ where we do not hide the position of non-$\textcircled{0}$ *cells*, but we hide the contents of these cells, is a triple $\langle\langle \mathbf{A} \rangle\rangle = \langle m, n, \mathcal{C} \rangle$, where $\mathcal{C}$ is the list of cells of the matrix that may contain an element different from $\textcircled{0}$. Each cell is again a triple $(i, j, [\![v]\!])$, where $i \in \{0, \ldots, m-1\}$ and $j \in \{0, \ldots, n-1\}$ are the coordinates of that cell, and $v \in S = \mathbb{N} \cup \{\infty\}$ is the value in it. The value v is stored privately in the ABB. In our implementation on top of Sharemind, we represent elements of S as 64-bit integers (representing ∞ as a large number). In the following, we use the standard list constructors, destructors, and combinators — $\mathsf{NIL}, \mathsf{cons}, \mathsf{length}, \mathsf{head}, \mathsf{tail}, ++$ (concatenation) — to express algorithms working with lists. We write $\mathcal{C}[k]$ for the k-th element of the list (starting with 0).

We allow the same coordinates (i, j) to occur several times in $\mathcal{C}$. We define that the triple $\langle m, n, \mathcal{C} \rangle$ represents a $m \times n$ matrix, where the cell at coordinates (i, j) contains the value $\min\{v \mid (i, j, [\![v]\!]) \in \mathcal{C}\}$.

We also make use of the dense representation $[\![\mathbf{V}]\!]$ of (small) matrices. It is simply a matrix of elements of S stored in the ABB.

Auxiliary functions. We have a relatively large set of helper functions for decomposing and combining matrices, as well as normalizing and converting between different representations. We list them below and shortly describe how they work.

$\mathsf{getMin}([\![\vec{v}]\!], \vec{i})$ takes a private vector of values, and an equal-length public vector of *indicators*. The indicator vector consists of segments of equal values. If

there are k such segments of length $l_1, \ldots, l_k$ (with $|\vec{v}| = \sum_j l_j$), then the output of getMin is a private vector of k values, where the j-th element is the minimum among the elements of $\vec{v}$ at the positions corresponding to the j-th segment of equal values in $\vec{i}$. The implementation of getMin is straightforward, we can divide $[\![\vec{v}]\!]$ into k segments according to the values in $\vec{i}$, and then call min from the ABB for all segments in parallel. Sharemind does not directly support such parallel invocation for segments of different length, but it is still possible to design getMin to run in SIMD fashion, doing $O(|\vec{v}|)$ work and requiring $O(\log \max_j l_j)$ rounds.

$\mathsf{norm}_1(\langle m, n, \mathcal{C}\rangle)$ takes a sparsely represented matrix. It returns the same matrix, having sorted elements $(i, j, [\![v]\!])$ of $\mathcal{C}$ by (i, j). It does not invoke any MPC protocols.

$\mathsf{norm}_2(\langle m, n, \mathcal{C}\rangle)$ first invokes norm_1 on its input, and then removes the duplicate occurrences of the same cell from $\mathcal{C}$. It does the latter by invoking getMin.

$\mathsf{getSlice}(\langle m, n, \mathcal{C}\rangle, u, l, m', n')$ returns the $m' \times n'$-sized submatrix of $\langle m, n, \mathcal{C}\rangle$, whose upper corner is in the cell (u, l) of the input matrix. Its output is $\langle m', n', \mathcal{C}'\rangle$, where $\mathcal{C}'$ is the list of elements $(i-u, j-l, [\![v]\!])$, where $(i, j, [\![v]\!]) \in \mathcal{C}$, $u \leq i < u + m'$, and $l \leq j < l + n'$.

$\mathsf{overlay}(\langle m, n, \mathcal{C}\rangle, u, l, m', n')$, where $m' \geq m+u$ and $n' \geq n+l$, outputs $\langle m', n', \mathcal{C}'\rangle$, where $\mathcal{C}'$ is the list of elements $(i + u, j + l, [\![v]\!])$, where $(i, j, [\![v]\!]) \in \mathcal{C}$. I.e. overlay creates a $m' \times n'$-sized supermatrix of the original matrix, where the upper left corner of the original matrix is at position (u, l), and the rest of the matrix is filled with $\textcircled{0} = \infty$.

$\mathsf{overlap}(\langle m, n, \mathcal{C}_1\rangle, \ldots, \langle m, n, \mathcal{C}_k\rangle)$ returns $\langle m, n, \mathcal{C}_1 +\!\!+ \cdots +\!\!+ \mathcal{C}_k\rangle$.

$\mathsf{transpose}(\langle m, n, \mathcal{C}\rangle)$ returns $\langle n, m, \mathcal{C}'\rangle$, where the elements of $\mathcal{C}'$ are the elements of $\mathcal{C}$ with their first two components swapped.

$\mathsf{identity}(n)$ returns the $n \times n$ identity matrix, represented sparsely.

$\mathsf{sparse\text{-}to\text{-}dense}(\langle m, n, \mathcal{C}\rangle)$ returns the dense representation of its argument (which has to be normalized). It initializes a $m \times n$ array of values $[\![\infty]\!]$, and copies the elements of $\mathcal{C}$ to their places.

$\mathsf{dense\text{-}to\text{-}sparse}([\![\mathbf{V}]\!])$ returns the sparse representation of its argument. It returns $\langle m, n, \mathcal{C}\rangle$, where m and n are dimensions of $\mathbf{V}$, and $\mathcal{C}$ is a list of length mn, containing one element for each cell of $[\![\mathbf{V}]\!]$.

Major functions. These include the addition and multiplication of matrices, and the computation of quasi-inverses of block-diagonal matrices. The first of them — pointwise minimum — is simple: if $\langle\!\langle \mathbf{M} \rangle\!\rangle$ and $\langle\!\langle \mathbf{N} \rangle\!\rangle$ have the same dimensions, then $\langle\!\langle \mathbf{M} \rangle\!\rangle \oplus \langle\!\langle \mathbf{N} \rangle\!\rangle = \mathsf{norm}_2(\mathsf{overlap}(\langle\!\langle \mathbf{M} \rangle\!\rangle, \langle\!\langle \mathbf{N} \rangle\!\rangle))$.

The multiplication protocol for sparse matrices, given in Alg. 1, is also unsurprising. An interesting detail is the transposition (and normalization) of the first matrix before the actual multiplication. In this way, the values of both x_1 and x_2 are non-decreasing during the loop. In our implementation we optimize the inner loop by running only through the segment of $\mathcal{D}$, where $x_2 = x_1$.

The only non-local operation in Alg. 1 is the final norm_2. The addition in line 8 is performed locally by the parties running the protocols implementing

Algorithm 1: Matrix multiplication over the semiring $\mathbb{N} \cup \{\infty\}$

Data: Matrices $\langle\langle \mathbf{M} \rangle\rangle = \langle m, n, \mathcal{C} \rangle$ and $\langle\langle \mathbf{N} \rangle\rangle = \langle n, k, \mathcal{D} \rangle$
Result: Matrix $\langle\langle \mathbf{M} \rangle\rangle \otimes \langle\langle \mathbf{N} \rangle\rangle$

```
1 begin
2     ⟨n, m, C'⟩ ← norm_1(transpose(⟨⟨M⟩⟩))
3     E ← NIL
4     for i ← 0 to length(C') − 1 do
5         (x_1, y_1, [[v_1]]) ← C'[i]
6         for j ← 0 to length(D) − 1 do
7             (x_2, y_2, [[v_2]]) ← D[j]
8             if x_1 = x_2 then E ← cons((y_1, y_2, [[v_1]] + [[v_2]]), E)
9     return norm_2(⟨m, k, E⟩)
```

Algorithm 2: Quasi-inverse of a block-diagonal matrix

Data: Matrix $\langle\langle \mathbf{M} \rangle\rangle = \langle n, n, \mathcal{C} \rangle$, list of block-sizes $\vec{B}$
Requires: $\mathcal{C}$ contains no cells outside the blocks defined by $\vec{B}$
Result: Matrix $\langle\langle \mathbf{M} \rangle\rangle^*$

```
1 begin
2     forall i ∈ {0, ..., length(B⃗) − 1} do
3         ⟨⟨A_i⟩⟩ ← getSlice(⟨⟨M⟩⟩, Σ_{j=0}^{i−1} B[j], Σ_{j=0}^{i−1} B[j], B[i], B[i])
4         ⟨⟨B_i⟩⟩ ← dense-to-sparse(FloydWarshall(sparse-to-dense(⟨⟨A_i⟩⟩)))
5         ⟨⟨C_i⟩⟩ ← overlay(⟨⟨B_i⟩⟩, Σ_{j=0}^{i−1} B[j], Σ_{j=0}^{i−1} B[j], n, n)
6     return overlap(⟨⟨C_0⟩⟩, ..., ⟨⟨C_{length(B⃗)−1}⟩⟩)
```

the ABB. Both the round complexity and the number of non-free operations of Alg. 1 depend on the cells included in $\mathcal{C}$ and $\mathcal{D}$.

Alg. 2 for quasi-inverse of $\langle\langle \mathbf{M} \rangle\rangle$ finds the quasi-inverse of each block of $\mathbf{M}$, and then combines the blocks. The input to Alg. 2 is a list of sizes of the blocks on the main diagonal of $\mathbf{M}$; the sum of elements of $\vec{B}$ has to be n. We use the Floyd-Warshall APSD algorithm [18] for computing the quasi-inverse of a single block. We have adapted our privacy-preserving implementation [2, Alg. 8] to compute the APSD for several (adjacency) matrices at the same time, such that the round complexity of Alg. 2 is $O(\max \vec{B})$, while the number of non-free operations is $O(\sum_i (B[i])^3)$. Our experiments [2] show that despite greater round complexity, Floyd-Warshall is faster than repeated squaring.

Main computation. The computation corresponding to the multiplication $\vec{x} \otimes \mathbf{A}^*$ according to (2) is given in Alg. 3. It takes as inputs the sparse matrix representations of both $\mathbf{A}$ and $\vec{x}$, where we think of the latter as a matrix with a single row. The multiplication operation also takes as input a list $\vec{L}$ of lists of block-sizes; it is formed on the basis of the separator tree of the graph having the adjacency matrix $\mathbf{A}$ (described at the end of Sec. 2.2), its length is $d + 1$.

Algorithm 3: Main loop of the algebra path computation

Data: Symmetric matrix $\langle\langle \mathbf{A} \rangle\rangle$ of size $n \times n$, non-empty list of lists of lengths $\vec{L}$, vector $\langle\langle \vec{x} \rangle\rangle$ of length n
Result: Vector $\langle\langle \vec{y} \rangle\rangle = \langle\langle \vec{x} \rangle\rangle \otimes \langle\langle \mathbf{A} \rangle\rangle^*$

1 **Function** *Algebraic-paths*$(n, \langle\langle \mathbf{A} \rangle\rangle, \vec{L}, \langle\langle \vec{x} \rangle\rangle)$ **is**
2 $\quad \vec{B} \leftarrow \mathsf{head}(\vec{L})$; $\vec{L}' \leftarrow \mathsf{tail}(\vec{L})$; $s \leftarrow \sum \vec{B}$
3 $\quad$ **if** $\vec{L}' = \mathsf{NIL}$ **then**
4 $\quad\quad$ **return** $\langle\langle \vec{x} \rangle\rangle \otimes \mathsf{quasi\text{-}inverse}(\langle\langle \mathbf{A} \rangle\rangle, \vec{B})$
5 $\quad \langle\langle \mathbf{X}^{\mathrm{ast}} \rangle\rangle \leftarrow \mathsf{quasi\text{-}inverse}(\mathsf{getSlice}(\langle\langle \mathbf{A} \rangle\rangle, 0, 0, s, s), \vec{B})$
6 $\quad \langle\langle \mathbf{Y} \rangle\rangle \leftarrow \mathsf{getSlice}(\langle\langle \mathbf{A} \rangle\rangle, s, 0, n-s, s)$
7 $\quad \langle\langle \mathbf{Z} \rangle\rangle \leftarrow \mathsf{getSlice}(\langle\langle \mathbf{A} \rangle\rangle, s, s, n-s, n-s)$
8 $\quad \langle\langle \mathbf{Q} \rangle\rangle \leftarrow \langle\langle \mathbf{Y} \rangle\rangle \otimes \langle\langle \mathbf{X}^{\mathrm{ast}} \rangle\rangle$
9 $\quad \langle\langle \mathbf{A}' \rangle\rangle \leftarrow \langle\langle \mathbf{Z} \rangle\rangle \oplus \langle\langle \mathbf{Q} \rangle\rangle \otimes \mathsf{transpose}(\langle\langle \mathbf{Y} \rangle\rangle)$
10 $\quad \langle\langle \vec{z} \rangle\rangle \leftarrow \langle\langle \vec{x} \rangle\rangle \otimes \mathsf{overlap}(\mathsf{identity}(n), \mathsf{overlay}(\mathsf{transpose}(\langle\langle \mathbf{Q} \rangle\rangle), 0, s, n, n))$
11 $\quad \langle\langle \vec{z}_{\mathrm{L}} \rangle\rangle \leftarrow \mathsf{getSlice}(\langle\langle \vec{z} \rangle\rangle, 0, 0, 1, s)$
12 $\quad \langle\langle \vec{z}_{\mathrm{R}} \rangle\rangle \leftarrow \mathsf{getSlice}(\langle\langle \vec{z} \rangle\rangle, 0, s, 1, n-s)$
13 $\quad \langle\langle \vec{w}_{\mathrm{L}} \rangle\rangle \leftarrow \langle\langle \vec{z}_{\mathrm{L}} \rangle\rangle \otimes \langle\langle \mathbf{X}^{\mathrm{ast}} \rangle\rangle$
14 $\quad \langle\langle \vec{w}_{\mathrm{R}} \rangle\rangle \leftarrow \mathsf{Algebraic\text{-}paths}(n-s, \langle\langle \mathbf{A}' \rangle\rangle, \vec{L}', \langle\langle \vec{z}_{\mathrm{R}} \rangle\rangle)$
15 $\quad \langle\langle \vec{w} \rangle\rangle \leftarrow \mathsf{overlap}(\mathsf{overlay}(\langle\langle \vec{w}_{\mathrm{L}} \rangle\rangle, 0, 0, 1, n), \mathsf{overlay}(\langle\langle \vec{w}_{\mathrm{R}} \rangle\rangle, 0, s, 1, n))$
16 $\quad$ **return** $\langle\langle \vec{w} \rangle\rangle \otimes \mathsf{overlap}(\mathsf{identity}(n), \mathsf{overlay}(\langle\langle \mathbf{Q} \rangle\rangle, s, 0, n, n))$

Alg. 3 closely follows (1)–(2). The current length of $\vec{L}$ describes the current depth of the recursion; length 1 (checked in line 3) indicates the base. Otherwise, $\langle\langle \mathbf{A} \rangle\rangle$ is the reprensetation of one of the matrices $\mathbf{A}_h$. We start by decomposing $\mathbf{A}_h$ into $\mathbf{X}_h$ (and find its quasi-inverse, using the list of lengths in the first element of $\vec{L}$), $\mathbf{Y}_h$ and $\mathbf{Z}_h$, compute $\mathbf{Q}_h$ and $\mathbf{A}_{h+1}$, multiply $\langle\langle \vec{x} \rangle\rangle$ with the first matrix in (2). We will then split the resulting vector $\vec{z}$ into two parts of lengths s and $n-s$, and multiply the left half with $\mathbf{X}_h^*$. We now recursively call Alg. 3 with the right half of $\vec{z}$, with $\mathbf{A}_{h+1}$, and with the list of lists of lengths missing the first element. We complete the computation by concatenating the two vectors, and multiplying it with the third matrix in (2). The round complexity, and the number of invoked ABB operations follow directly from Pan and Reif's analysis [31].

In order to compute the distances from a vertex t of an undirected graph $G = (V, E)$ with public locations, but private lengths of edges, we have to perform more steps before and after invoking Alg. 3, but all these steps are public. Starting from the sparsely represented adjacency matrix $\langle\langle \mathbf{A} \rangle\rangle$ of G, we have to find the separator tree of G, permute the vertices of G (giving us the matrix $\langle\langle \mathbf{A}_0 \rangle\rangle$), and create the list $\vec{L}$. We have to create the vector $\langle\langle \vec{x} \rangle\rangle$ as a unit vector, where we have the value $\textcircled{1} = 0$ only at the position corresponding to the location of vertex t after the permutation. After calling Alg. 3 with $\langle\langle \mathbf{A}_0 \rangle\rangle$, $\vec{L}$ and $\langle\langle \vec{x} \rangle\rangle$, we have to apply the inverse permutation to the resulting vector $\langle\langle \vec{y} \rangle\rangle$.

4 Security and privacy of protocols

The privacy-preserving APC protocol is built on top of a universally composable ABB. It receives its private inputs through the handles to values stored in the ABB, and returns its private outputs in the same fashion. The protocol contains no declassify-operations. Hence, as discussed in Sec. 2.1, it inherits the same security properties against various adversaries as the underlying secure computation protocol set. In particular, if the ABB is implemented by the Sharemind MPC platform, then the resulting APC protocol is a three-party protocol, working with public locations but secret-shared lengths of edges, and provides information-theoretic security against an adversary passively corrupting at most one of the parties.

5 Empirical evaluation

5.1 Privacy-preserving Bellman-Ford with public edges

We want to compare the APC protocol with protocols based on classical SSSD algorithms, where the locations of edges are public, but their lengths are private. We see that Dijkstra's algorithm cannot benefit from public location of edges, because the order in which it relaxes the vertices depends on the lengths of the edges, thus the random permutation of vertices that could hide that order [1] would make the locations of edges private again. Hence we think that it is fair to compare the new protocol against a protocol based on the Bellman-Ford (BF) algorithm.

Such privacy-preserving algorithm is given in Alg. 4. We see that at each iteration of the main loop, it defines $[\![\vec{a}]\!]$ as the current distance of the start vertex of each edge from s. Vector $[\![\vec{b}]\!]$ will then record the current distance of the end vertex of each edge, when the last step is made over this edge. The same getMin operation as in Sec. 3 is used to find the minimum distance for each vertex. We see that the number of non-free operations executed by Alg. 4 is $O(mn)$, while its round complexity is $O(n \log D)$, where D is the maximum in-degree of a vertex.

5.2 Setup of benchmarking

We have implemented the APC and BF algorithm on the Sharemind MPC platform, using the SecreC language [33] offered by this platform. The benchmarking took place on three servers with 12-core 3 GHz CPUs with Hyper-Threading running Linux, and 48 GB of RAM, connected by an Ethernet 1 Gbps LAN. The local computations in Sharemind MPC are single-threaded, and there is no support for performing computations and network operations at the same time.

We want to measure the performance in different network environments, corresponding to LAN and WAN deployments. We throttle the connections between

Algorithm 4: Bellman-Ford based SSSD algorithm with public edge locations

Data: Number of vertices and edges n and m
Data: Vectors (of length m) of starting and ending vertices, and lengths of edges: $\vec{S}$, $\vec{T}$, and $[\![\vec{W}]\!]$
Data: starting vertex $s \in \{0, \ldots, n-1\}$
Requires: $\vec{T}$ is sorted
Requires: There is a loop edge with length 0 at each vertex
Result: Vector of distances $[\![\vec{D}]\!]$ from vertex s

```
1 begin
2     [[D]] ← ∞; [[D[s]]] ← 0
3     for i ← 0 to n − 1 do
4         forall j ∈ {0, ..., m − 1} do [[a[j]]] ← [[D[S[j]]]];
5         [[b]] ← [[a]] + [[W]]
6         [[D]] ← getMin([[b]], T)
7     return [[D]]
```

the servers in order to simulate these environments. In our experiments, we consider "HBLL", "HBHL" and "LBHL" settings. Here HB (high-bandwidth) means 1 Gbps and LB (low-bandwidth) 100 Mbps link speed between servers. Also, LL (low-latency) means no added delay for the messages sent between the servers, while HL (high-latency) means additional 40 ms delay.

The performance of the APC algorithm is highly dependent on the locations of edges. As we are most interested in the performance of the algorithms on planar graphs, and as we want to focus on optimizing the privacy-preserving computations, not the computation of the separator tree, we have selected *grid graphs* as the family of graphs on which we have performed benchmarking. The $R \times C$ grid graph has RC vertices that can be thought as being placed in a $R \times C$ grid. Each vertex is connected with 4 of its closest neighbours (less for vertices at the edges of the grid); the number of (undirected) edges is $(2RC - R - C)$. Grid graphs have easy-to-compute separators of size $\min(R, C)$ that split their set of vertices into two roughly equal parts; the height of the resulting separator tree is $\approx \log R + \log C$. In the following we let $G(N)$ denote the $N \times N$ grid graph.

5.3 Measuring the performance of algebraic path computation

We report the running times and the bandwidth consumption (per computing server) for grid graphs $G(N)$ for different values of N, on Sharemind cluster for the HBLL network environment in Table 1. The times correspond to the execution of Alg. 3; we have not measured the time it takes to construct the separator tree, the lists $\vec{\mathcal{L}}$ and $\vec{L}$, or to permute the matrices and vectors.

The largest grid graph that we ran our implementation on, was $G(600)$. This graph has 360k vertices and $\approx$ 1.4M (directed) edges. We are not aware of any

Table 1. Running time (in seconds) and bandwidth consumption of privacy-preserving algebraic path computation protocol for graphs $G(N)$

N	Bandwidth	Time
5	0.16 MB	0.1
9	0.30 MB	0.3
17	2.31 MB	1.2
33	27.3 MB	8.2
50	90.3 MB	30.1

N	Bandwidth	Time
65	366 MB	66.4
100	874 MB	244
129	1972 MB	522
150	3136 MB	838

N	Bandwidth	Time
200	7792 MB	2029
257	16.4 GB	4280
513	138.3 GB	35341
600	224.6 GB	58082

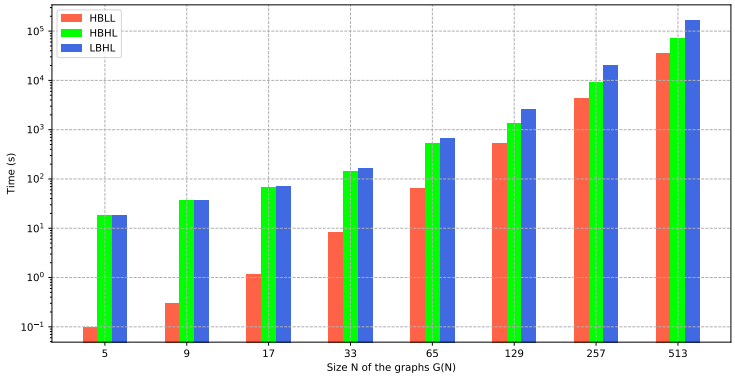

Fig. 1. Performance of algebraic path computation protocol on graphs with given numbers of vertices in different network environments

previous executions of privacy-preserving SSSD on graphs of similar size, no matter if the locations of edges are private or not, or what the actual shape of the graph is. We see that the running time for such a graph was a bit over 16 hours, which may be practical for certain settings.

In Figure 1, we compare the running time of privacy-preserving Algebraic path computation protocol on graphs of different sizes in different network environments. We see that for small graphs, the performance only depends on the latency of the network. Only for graphs with 1000 or more vertices ($N = 33$) does the available bandwidth start having an effect.

5.4 Comparison of APC and BF protocols

The running times of both privacy-preserving SSSD protocols that use public edges — Bellman-Ford and Algebraic path computation — for the sparse representation of the graphs are illustrated in Table 2. The experiments also show average bandwidths in different network environments. The running times of all graphs in different network environments for Algebraic path computation are

Table 2. Benchmarking results (bandwidth for a single computing server) for Bellman-Ford and Algebraic path protocol in different network environments, for grid graphs $G(N)$

	Bellman-Ford				**Algebraic Path Computation**				**Speed-up**		
	Band-	**Running time (s)**			**Band-**	**Running time (s)**			**BF** vs. **APC**		
N	**width**	HBLL	HBHL	LBHL	**width**	HBLL	HBHL	LBHL	HBLL	HBHL	LBHL
5	0.4 MB	0.33	33.3	33.3	0.09 MB	0.1	18.2	18.2	3.3x	1.8x	1.8X
9	2.64 MB	2.74	108	108	0.28 MB	0.3	38.0	38.0	9.1x	2.8x	2.8x
17	22.3 MB	18.4	388	399	2.33 MB	1.2	69.4	71.4	15.3x	5.6x	5.6x
33	324 MB	214	1509	1684	24.1 MB	8.2	146	165	26.1x	10.3x	10.2
65	4.4 GB	819	6542	9205	273 MB	66.4	522	670	12.3x	12.5x	13.7x
129	173 GB	13395	36835	81346	2005 MB	522	1355	2669	25.6x	27.1x	30.5x
257	2.86 TB	203428	521491	1154261	17.2 GB	4280	9182	20276	47.5x	56.8x	56.9x
513	37.3 TB	3092314	7147049	17883699	144 GB	35341	73215	166643	87.4x	97.6x	107.3x

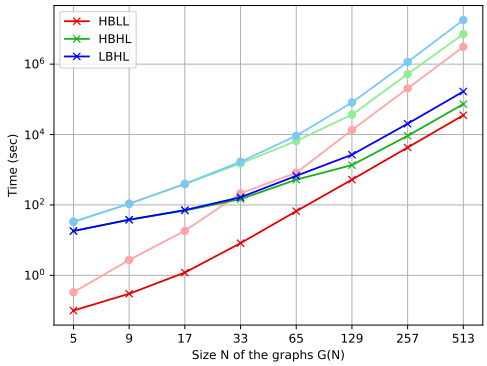

Fig. 2. Performance (time in seconds) of Bellman-Ford Version 3 and Algebraic path computation protocols on graphs of different sizes in different network environments (red: HBLL, green: HBHL, blue: LBHL, light: Bellman-Ford, dark: Algebraic path computation)

lower than the running times of the Bellman-Ford protocol. Similarly, the bandwidth consumption in Algebraic path computation is smaller than bandwidth in Bellman-Ford protocol.

In Table 2, the execution times of the Bellman-Ford protocol on larger graphs have been estimated: we benchmarked the larger examples by running only a few iterations of the main loop in Alg. 4, measured the running time of a single iteration, and then multiplied with the total number of iterations.

We depict the running times also in Fig. 2, presenting the comparison of Algebraic path computation and Bellman-Ford protocol for different network environments. We see that despite the simple structure of Bellman-Ford, Algebraic path computation is still faster also in high-latency environments.

6 Conclusion and future work

We have shown that designers of privacy-preserving applications working with data in graph form and needing to find the distances between vertices should look beyond the classical SSSD algorithms when selecting the protocol for shortest paths' computation on top of a SMC framework. Even though many of the Parallel RAM algorithms proposed for SSSD have components that are not easily converted into parallel privacy-preserving protocols (e.g. the spawning and scheduling of tasks based on private data), there may be algorithms that process data sufficiently uniformly in order to serve as basis of SMC protocols.

We have shown how APC may be used to compute SSSD in privacy-preserving manner. It gives us efficient protocols, compared to classical SSSD algorithms. The same semiring framework may be instantiated in different ways, and be used for solving other graph problems, e.g. finding the minimum spanning trees or solving the all-pairs shortest distance problem. These algorithms may be converted into SMC protocols exactly as we have done here, with the only possible slight difference arising from the scalar $\otimes$-operation no longer being free.

In this paper, we have presented a protocol for undirected graphs. The APC algorithm is equally well applicable to directed graphs [31, Remark 6.1], and this change can also be implemented on top of an ABB.

In this paper, we have required the locations of edges to be public. We believe that a protocol with private locations is possible. This would not significantly change the subroutines. Still the matrix multiplication may become more expensive due to the need to run through both loops in Alg. 1, and quasi-inverse will become more expensive due to the need to consider a central stripe of diagonals, instead of just the blocks on the main diagonal. There may be more changes to the main computation, as we no longer know the sizes of matrices; hence padding may be necessary. Also, the main computation would receive the list of lists of block-sizes as a private parameter, too.

Computing that list of lists of block-sizes privately is likely an even more complex problem. We are not aware of efficient parallel RAM algorithms for computing the separator tree, that could be easily converted to run on top of a SMC framework.

Acknowledgements. This research received funding from the European Regional Development Fund through the Estonian Centre of Excellence in ICT Research—EXCITE.

References

1. Aly, A., Cleemput, S.: An improved protocol for securely solving the shortest path problem and its application to combinatorial auctions. Cryptology ePrint Archive, Paper 2017/971 (2017)
2. Anagreh, M., Laud, P., Vainikko, E.: Parallel privacy-preserving shortest path algorithms. Cryptography **5**(4), 27 (2021)
3. Anagreh, M., Laud, P., Vainikko, E.: Privacy-preserving parallel computation of shortest path algorithms with low round complexity. In: Mori, P., Lenzini, G.,

Furnell, S. (eds.) Proceedings of the 8th International Conference on Information Systems Security and Privacy, ICISSP 2022, Online Streaming, February 9-11, 2022. pp. 37–47. SCITEPRESS (2022)
4. Anagreh, M., Vainikko, E., Laud, P.: Parallel privacy-preserving computation of minimum spanning trees. In: Mori, P., Lenzini, G., Furnell, S. (eds.) Proceedings of the 7th International Conference on Information Systems Security and Privacy, ICISSP 2021, Online Streaming, February 11-13, 2021. pp. 181–190. SCITEPRESS (2021)
5. Anagreh, M., Vainikko, E., Laud, P.: Parallel privacy-preserving shortest paths by radius-stepping. In: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). pp. 276–280. IEEE (2021)
6. Bellman, R.: On a routing problem. Quarterly of applied mathematics **16**(1), 87–90 (1958)
7. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: European Symposium on Research in Computer Security. pp. 192–206. Springer (2008)
8. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multiparty computation for data mining applications. International Journal of Information Security **11**(6), 403–418 (2012)
9. Bollobás, B.: Modern graph theory, Graduate Texts in Mathematics, vol. 184. Springer Science & Business Media (1998)
10. Boyle, E., Chung, K.M., Pass, R.: Large-scale secure computation: Multi-party computation for (parallel) ram programs. In: Annual Cryptology Conference. pp. 742–762. Springer (2015)
11. Boyle, E., Jain, A., Prabhakaran, M., Yu, C.H.: The bottleneck complexity of secure multiparty computation. In: 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)
12. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In: 19th USENIX Security Symposium (USENIX Security 10) (2010)
13. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001)
14. Cohen, R., Coretti, S., Garay, J., Zikas, V.: Round-preserving parallel composition of probabilistic-termination cryptographic protocols. Journal of Cryptology **34**(2), 1–57 (2021)
15. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2022)
16. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Annual International Cryptology Conference. pp. 247–264. Springer (2003)
17. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik **1**(1), 269–271 (1959)
18. Floyd, R.W.: Algorithm 97: shortest path. Communications of the ACM **5**(6), 345 (1962)
19. Flynn, M.J.: Very high-speed computing systems. Proceedings of the IEEE **54**(12), 1901–1909 (1966)
20. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In: Proceedings of the

seventeenth annual ACM symposium on Principles of distributed computing. pp. 101–111 (1998)
21. Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: Tasty: tool for automating secure two-party computations. In: Proceedings of the 17th ACM conference on Computer and communications security. pp. 451–462 (2010)
22. Hodges, A.: Alan Turing: the enigma. Princeton University Press (2014)
23. Katz, J., Koo, C.Y.: Round-efficient secure computation in point-to-point networks. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 311–328. Springer (2007)
24. Katz, J., Ostrovsky, R., Smith, A.: Round efficiency of multi-party computation with a dishonest majority. In: International Conference on the Theory and Applications of Cryptographic Techniques. pp. 578–595. Springer (2003)
25. Laud, P.: Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees. Proceedings on Privacy Enhancing Technologies **2015**(2), 188–205 (2015)
26. Laud, P.: Stateful abstractions of secure multiparty computation. Applications of Secure Multiparty Computation. Cryptology and Information Security **13**, 26–42 (2015)
27. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. SIAM Journal on Applied Mathematics **36**(2), 177–189 (1979)
28. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics **7**(3), 321–350 (2002)
29. Pan, V., Reif, J.: The parallel computation of minimum cost paths in graphs by stream contraction. Information Processing Letters **40**(2), 79–83 (1991)
30. Pan, V., Reif, J.: Fast and efficient solution of path algebra problems. Journal of Computer and System Sciences **38**(3), 494–510 (1989)
31. Pan, V., Reif, J.: Fast and efficient parallel solution of sparse linear systems. SIAM Journal on Computing **22**(6), 1227–1250 (1993)
32. Pinto, A., Carloni, L.P., Sangiovanni-Vincentelli, A.L.: Efficient synthesis of networks on chip. In: Proceedings 21st International Conference on Computer Design. pp. 146–150. IEEE (2003)
33. Randmets, J.: Programming Languages for Secure Multi-party Computation Application Development. Ph.D. thesis, Tartu University (2017)
34. Sealfon, A.: Shortest paths and distances with differential privacy. In: Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. pp. 29–41 (2016)
35. Tarabalka, Y., Chanussot, J., Benediktsson, J.A.: Segmentation and classification of hyperspectral images using watershed transformation. Pattern Recognition **43**(7), 2367–2379 (2010)
36. West, D.B., et al.: Introduction to graph theory, vol. 2. Prentice hall Upper Saddle River (2001)
37. Wu, D.J., Zimmerman, J., Planul, J., Mitchell, J.C.: Privacy-preserving shortest path computation. arXiv preprint arXiv:1601.02281 (2016)
38. Yamada, T.: A mini–max spanning forest approach to the political districting problem. International Journal of Systems Science **40**(5), 471–477 (2009)
39. Yamada, T., Takahashi, H., Kataoka, S.: A heuristic algorithm for the mini-max spanning forest problem. European Journal of Operational Research **91**(3), 565–572 (1996)
40. Yao, A.C.: Protocols for secure computations. In: 23rd annual symposium on foundations of computer science (sfcs 1982). pp. 160–164. IEEE (1982)