Securely Solving Classical Network Flow Problems

Abdelrahaman Aly(✉) and Mathieu Van Vyve

Université catholique de Louvain, CORE, Voie du Roman Pays 34,
1348 Louvain-la-Neuve, Belgium
{abdelrahaman.aly,mathieu.vanvyve}@uclouvain.be

Abstract. We investigate how to solve several classical network flow problems using secure multi-party computation. We consider the shortest path problem, the minimum mean cycle problem and the minimum cost flow problem. To the best of our knowledge, this is the first time the two last problems have been addressed in a general multi-party computation setting. Furthermore, our study highlights the complexity gaps between traditional and secure implementations of the solutions, to later test its implementation. It also explores various trade-offs between performance and security. Additionally it provides protocols that can be used as building blocks to solve complex problems. Applications of our work can be found in: communication networks, routing data from rival company hubs; distribution problems, retailer/supplier selection in multi-level supply chains that want to share routes without disclosing sensible information; amongst others.

Keywords: Network Flows · Multi-party computation · Secure collaboration

1 Introduction

Secure Multi-party Computation (MPC), studies the problem where several players want to jointly compute a given function without disclosing their inputs; this problem was first addressed by Yao [1]. Different adversary models can be considered. A semi-honest setting, where corrupted players try to learn only what can be inferred from the information they have been provided with; or an active setting, where they manipulate the data in order to learn from any possible leakage caused. Several cryptographic primitives for secret sharing and homomorphic encryption, e.g. Shamir scheme [2] and Pailler encryption [3], have been proposed to address the problem.

Applications have emerged naturally in different fields, for instance, where all the secret information is sent by the players to a third trusted party who only reveals the final output. For example, in auctions, the auctioneer can be seen as a trusted third party. We study the scenario where no trusted third parties are allowed.

J. Lee and J. Kim (Eds.): ICISC 2014, LNCS 8949, pp. 205–221, 2015.
DOI: 10.1007/978-3-319-15943-0_13

Classical network flow problems arise in real life applications in several areas e.g. project planning, networking, supply chain management, production scheduling. Combinatorial optimization, dynamic programing and mathematical programming have yielded polynomial-time algorithms for many of these problems (a detailed treatment can be found in Ahuja et al. [4]).

Our central objects of study are the shortest path problem on weighted graphs, the Minimum Mean Cycle problem (MMC) and the the Minimum Cost Flow problem (MCF). We present algorithms that address privacy preserving constraints on these problems and solve them in polynomial time. We also empirically test the performance of our implementations. Finally, we show how to use our protocols as building blocks to solve more complex problems. For example, a WLAN network constructed by competing agents that want to securely compute, in a distributed fashion, their routing tables and the network flow configuration that supports its maximum traffic volume at the minimum cost possible. The routing algorithms could use our shortest path protocol to securely define the routing tables. Moreover, a combination of the max flow algorithm [5] with our minimum cost flow protocol could be used to obtain the desired flow distribution securely. Note that for these types of application the number of vertices e.g. routers, is not necessarily very large.

1.1 Our Contributions

We provide algorithmic solutions to three classical network flow problems in a secure, multi-party and distributed setting: the shortest path based on Dijkstra, the minimum mean cycle using Karp's solution and the minimum cost flow using the Minimum Mean Cycle Canceling (MMCC) algorithm. To the best of our knowledge, this is the first time the last two problems have been studied under MPC security constraints. We also introduce a novel technique to hide the vertex selected at each iteration of Dijkstra's algorithm, avoiding the overhead caused by the use of special data structures e.g. oblivious data structures. This is particularly relevant on dense graphs. We refer the reader to Sect. 1.2 for further analysis. Moreover, we show polynomial bounds for all three problems relying only on black box operations. In our configurations, the secret information can be distributed as pleased by the parties. Our work considers all input data to be secret except for a bound on the number of vertices of the graphs.

Security and Correctness. The security of our algorithms comes from the fact that we only use operations from the arithmetic black-box and prevent any information leakage. This implies that the protocols are as secure as the MPC primitives they are implemented over e.g. information-theoretic secure, (see also Sect. 2.1). Furthermore the correctness of our algorithms is essentially inherited from the correctness of the classical algorithms from which they are derived. More specifically, we modify the previously known and correct algorithms to avoid, in general, information leakage, while working on secret data, showing that these modifications do not alter their output.

Table 1. Asymptotic bounds of original and privacy-preserving algorithmic versions

	Advance Impl.	Simple Impl.	Complete Graphs	Privacy Preserving	Secure Comparisons
Dijkstra	$\|E\| + \|V\| \cdot log(\|V\|)$	$\|V\|^2$	$\|V\|^2$	$\|V\|^3$	$\|V\|^2$
MMC	$\|E\| \cdot \|V\|$	$\|E\| \cdot \|V\|$	$\|V\|^3$	$\|V\|^5$	$\|V\|^3$
MMCC	$\|V\|^2 \cdot \|E\|^3 log(\|V\|)$	$\|V\|^2 \cdot \|E\|^3 log(\|V\|)$	$\|V\|^8 \cdot log(\|V\|)$	$\|V\|^{10} \cdot log(\|V\|)$	$\|V\|^8 \cdot log(\|V\|)$

Complexity. We use atomic communication rounds as our main performance unit to determine the complexity (round complexity) of our protocols. Besides, because of the strong differences in performance between comparisons and multiplications we limit the use of comparisons in favor of more arithmetic operations. i.e. additions and multiplications. Table 1 presents the complexity bounds we obtain. In all cases, the number of comparisons matches the complexity of implementations on complete graphs. However, we need to introduce additional multiplications to hide the branchings involved in the algorithms.

1.2 Related Works

Graph Theory Problems. Different alternatives to solve some graph theory problems have been studied by Aly et al. [5], namely the shortest path and maximum flow problems. They provide bounds on the Bellman-Ford and Dijkstra algorithms. Our own bounds are slightly better with our version of Dijkstra's algorithm, using different approaches. Indeed, [5] uses a searching array technique, similar to the one proposed by Launchbury et al. [6], to keep track of a secret shared index. Our proposed Dijkstra implementation does not require the use of this technique, eliminating its overhead. Edmonds-Karp and push-relabel bounds are provided as well for the maximum flow problem. As in our case, their implementations are secure in the *information-theoretic* model relying on the same arithmetic black-box $\mathcal{F}_{ABB}$. Brickell and Shmatikov [7] have addressed the shortest path problem on a two-party case, limited to the honest by curious model. They succeed by revealing at each iteration the new edge of the shortest path added. Our approach attacks the problem in a different fashion by eliminating this requirement. We also address the problem in a multi-party setting and not limited to the two-party case. Moreover, Banton et al. [8] have proposed a data-oblivious alternative for the Breath-First-Search (BFS) algorithm, which is later used to solve the special case of the shortest path problem where all edges have the same weight e.g. all existing edges weight 1 and non-existing 0. We consider instead the more general case with weighted edges. Additionally, they use their BFS algorithm to provide bounds for the Max-Flow problem, where weighted edges with a positive residual capacity are mapped as 1 and its counterparts as 0, extending the definition of an existing edge.

Oblivious data structures over ORAM. Data structures are used to speed-up Dijkstra's algorithm and achieve its optimal complexity. ORAM has been viewed as a suitable mechanism to build oblivious distributed data structures with

the corresponding overhead and configuration e.g. The work of Wang et al. [9] designed to work on a client(s)-server configuration. Moreover, secure two-party computation protocols have been developed to take advantage of the recent advances on ORAM e.g. [10,11]. The two-party tool and algorithmic implementations of Liu et al. [12] securely address the shortest path and other combinatorial problems by using these kinds of data structures. More recently, Keller and Scholl [13] show how to use oblivious data structures on a multi-party setting, where none of the players have to fulfill the role of the server. Furthermore, they use their data structures to implement Dijkstra's algorithm. Their experimentation shows how some MPC solutions in the absence of ORAM can perform better for certain kinds of graphs than their proposed counterparts i.e. samples of smaller-to-medium sizes and complete graphs of any size. This is easily explained by the fact that the overhead coming from the ORAM exceeds the asymptotic advantage of the algorithms. Indeed, we address the problem differently, our Dijkstra algorithm is designed to work on plain vectors and matrices and does not require any secure data structure construction, slightly improving the bounds proposed by Aly et al. [5], who's work is later used in Keller and Scholl's analysis. This allows us to avoid any overhead caused by the use of ORAM or static secret sharing arrays. We refer to [13] for details.

1.3 Overview

Section 2 describes the notation we use, as well as the cryptographic primitives. It also serves to introduce "building blocks" i.e. small algorithmic procedures regularly used. In Sect. 3 we present a solution for Dijkstra. Section 4 introduces the minimum mean cycle problem. Section 5 then explains the implementation using MPC primitives. Section 6 gives an overview of the minimum flow problem and the minimum mean cycle-canceling algorithm. In the sections ahead, details on the algorithm are presented. In Sect. 7, we present and discuss our secure algorithmic solution. Section 8 shows the results of our computational experimentation. Lastly, Sect. 9 provides general conclusions.

2 Preliminaries

2.1 Security

We use the terms "securely" and "privacy preserving" indistinctly. We can succinctly formalize their notion as follows: parties $P_1, ..., P_n$ want to jointly and correctly compute the function $y = f(x_1, ..., x_n)$ where x_i is P_i secret input and only y is allowed to be revealed to all parties. In other words, the security constraint is such that each player P_i learns y and what can be inferred from y, but no more. In particular, any information given during the computation process should not allow him to infer information about other secret inputs.

Modulo arithmetic for some M or ring arithmetic allows to simulate secure integer arithmetic. Indeed, several multi-party computation solutions have been

designed to work on modulo arithmetic for an appropriate M e.g. a sufficiently big prime number (transforming the ring in a finite field over some M, $\mathbb{Z}_M$), such that no overflow occurs. This is true for secret sharing schemes the likes of Shamir [2] sharing or additive sharing, as well for homomorphic threshold public key encryption.

Primitives like addition between secret shared inputs on secret sharing, as well as additions and multiplications of these by public values, are linear operations and do not require any information transmission between players. When data is communicated between players it is called a communication round or just round. For complexity analysis purposes, we require constant-round protocols for multiplications. By extension, sharing and reconstruction are done in one round as well. There are still local operations involved with all the primitives, but the performance cost is mainly determined by the communication processes, as explained by Maurer [14]. We assume that the execution flavor i.e. sequential and parallel, does not compromise the security of the private data.

The concept of the arithmetic black-box $\mathcal{F}_{ABB}$ [15] embeds this behavior and makes the process transparent for the algorithm designer. It creates an abstraction layer between the protocol construction and functionality specificities, and at the same time it provides the security guarantees desired. Following [5,15] amongst others, we assume the following functionalities are available: storage and retrieval of ring $\mathbb{Z}_M$ elements, additions, multiplications, equality and inequality tests.

On a final note, all our protocols are designed under the *information-theoretic* model in the presence of passive or active adversaries over $\mathcal{F}_{ABB}$. This implies that as long as the parties do not have access to other private data but their own, unbounded computing power would not allow them to obtain any additional information. This means in practice that they will be as secure as the underlying MPC functionality and crypto-primitives they rely on.

2.2 Notation

We use the traditional square brackets e.g. [x], to denote secret shared or encrypted values contained in the $\mathcal{F}_{ABB}$. This notation is commonly used by secure applications e.g. [5]. Sometimes $[\infty]$ is used on our algorithms. Given that the $\mathcal{F}_{ABB}$ is limited by the size of M, this value, has to be understood as a sufficiently large constant smaller than M but much bigger than the values of the inputs. On the size of M, it has to be noted that some comparison protocols require a security parameter on the size of M that has to be taken into account when defining its size. We also assume all values analyzed by our protocols, including intermediate data, to be integers bounded to M to avoid overflows. Moreover, secure operations are described using the infix operation e.g. $[z] \leftarrow [x] + [y]$ for secure addition into the $\mathcal{F}_{ABB}$ and $[z] \leftarrow [x] \cdot [y]$ for secure multiplication. The secret result of any secure operation primitive is stored in $[z]$ and onto the $\mathcal{F}_{ABB}$. This notation covers all operations performed with secret values, including those performed between public scalars and secret values. These operations are provided by the $\mathcal{F}_{ABB}$.

We define two repeatedly used subroutines to improve readability and simplify expressions. They only use the primitives available in the $\mathcal{F}_{ABB}$ and work under the same general assumptions.

conditional assignment: Overloaded functionality of the assignment operator represented by $[z] \leftarrow_{[c]} [x] : [y]$. Much like in [5,13], the behavior of the assignment is tied to a secretly shared binary condition $[c]$. If $[c]$ is one, $[x]$ is assigned to $[z]$ or $[y]$ otherwise. The operation can be characterized as follows: $[z] \leftarrow [y] + [c] \cdot ([x] - [y])$. The subroutine can be extended for other mathematical structures i.e. vectors, matrices.

conditional exchange: We define the operator $condexch([c], i, j, [v])$. It exchanges the values held in position i and j of secretly shared vector $[v]$ if a secretly shared binary condition $[c]$ is 1 and leaves the vector unchanged otherwise. We describe the algorithm as Protocol 1. We also extend this operator to work with matrices. In that case both i^{th} and j^{th} rows and columns are swapped.

Protocol 1. condexch: Exchanges the values of 2 different vector positions

Input: Any vector $[v]$. Indexes i, j
Output: The vector $[v]$ with values i,j swapped if $[c]$ true.

1 $[a] \leftarrow [c] \cdot ([v]_j - [v]_i)$;
2 $[v]_i \leftarrow [v]_i + [a]$;
3 $[v]_j \leftarrow [v]_j - [a]$;

2.3 On Network Flows and Matrix Representation

The number of vertices in the graph or at least an upper bound on them are assumed to be publicly known with no restrictions on how the information is distributed amongst the players. Following [5,8,13] our protocols assume complete graph representation for their inputs, as a tool to hide the graph structure. That is why an adjacency matrix representation of the graph, using the bound as its size, is preferred. Capacities and/or costs of the edges are represented as elements in matrices. This allows the algorithm designer to decouple the graph representation from its topology. The application designer has to define how information of the topology is actually distributed and what is hidden. For instance, if its known that each player owns at most a single vertex, then, each player has to secretly share a row of a capacity adjacency matrix where he places a [0] at each unconnected vertex position or $[\infty]$ if its a cost matrix. We briefly describe some general definitions on graph theory that are often used during the following sections. Ahuja et al. [4] provides more formal notions.

Residual Graph: Is the associated network defined by all edges with positive residual capacities.

Walk: Is a sequence of contiguous edges $(v_1, w_1), ..., (v_k, w_k)$ such that $w_i = v_{i+1}$ for all $1 \leq i \leq n-1$ and $(v_i, w_i) \in E$ for all $1 \leq i \leq n$.

Path: A path is a walk of G where no vertex is visited more than once. Every path, by definition is a walk, but not all paths are walks.

Cycle: A cycle is a special path $(v_1, w_1)...(v_d, w_d) \in E$ where $v_1 = w_d$. Every cycle by definition is a path, but not all paths are cycles.

3 Dijkstra's Algorithm

The algorithm provides a greedy way to find the shortest path from a source vertex s in a directed connected graph with non-negative capacities. Basically, it selects the vertex with the smallest accumulated distance and then propagates the path forward until all vertices have been explored. This ensures to get the shortest path from a source vertex to all other vertices in the graph. To find the shortest path to a single vertex is also possible. Our secure implementation can be adapted to detect at each iteration whether the target vertex has been reached to stop the algorithm.

Adapting Dijkstra to MPC. The input data in our case is a weighted adjacency matrix $[U]$ where non existing edges are represented by $[\infty]$. Dijkstra's algorithm treats the vertices of the graph in an order that depends on the capacities of the edges. The main challenge is to hide this order. Earlier work [5] has proposed to hide the position of the vertex accessed by using a secretly shared unary vector $[0, 0, ..., 0, 1, 0, ..., 0]$. We introduce a different technique. The basic idea is to exploit the symmetry in the data structure. More precisely, the numbering of the vertices or equivalently, the position of a vertex in the data structure is indifferent for the algorithm. We exploit this by positioning at iteration i, the vertex with the lowest distance in position i. That way we align the vertex exploration of our protocol with the secret data stored in all the structures. This enables us to gain in the number of operations performed because we can avoid considering edges pointing to vertices already explored. The algorithm is detailed as Protocol 2.

Correctness. Because the algorithm constantly reshuffles the positions of the vertices in all matrices and vectors used, we need to (secretly) track the position of the vertices. This is the role of the vector π. Throughout the algorithm π_j holds the node number that is currently in position j.

The loop on lines 5–8 determines the untreated vertex with current minimum distance. This vertex is brought to position i in all data structures. Loop on lines 9–14 scans all edges leaving node in position i to all other untreated vertices (positioned after i). If the edge improves the current best path (Line 11), the current best distances and predecessors are updated (Lines 12–13). The predecessor of node i is recorded as P_j. If the path needs to be kept secret and subsequently used in a parent protocol, then it would be more suitable to record this information in a matrix with $P_{i,j} = 1$ indicating that the predecessor of i is j (and 0 otherwise). It is easy to adapt the algorithm for this case.

Security. Following the correctness analysis, (i) it is easy to check that no intermediate value is revealed. (ii) The execution flow only depends on publicly known

Protocol 2. Shortest Path Protocol based on Dijkstra's algorithm

Input: A matrix of shared weights $[U]_{i,j}$ for $i, j \in \{1, ..., |V|\}$ and a unit vector $[S]$ encoding the source vertex.
Output: The vector of predecessors $[P]$ and/or the vector of distances $[d]_i$.

1 **for** $i \leftarrow 1$ **to** $|V|$ **do**
2 | $[\pi]_i \leftarrow i$; $[d]_i \leftarrow_{[S_i]} [0] : [\infty]$; $[P]_i \leftarrow i[S]_i$;
3 **end**
4 **for** $i \leftarrow 1$ **to** $|V|$ **do**
5 | **for** $j \leftarrow |V|$ **to** $i + 1$ **do**
6 | | $[c] \leftarrow [d]_j < [d]_{j-1}$;
7 | | $([\pi], [P], [d], [U]) \leftarrow condexch([c], j, j-1, [\pi], [P], [d], [U])$;
8 | **end**
9 | **for** $j \leftarrow i + 1$ **to** $|V|$ **do**
10 | | $[a] \leftarrow [d]_i + [U]_{i,j}$;
11 | | $[c] \leftarrow [a] < [d]_j$;
12 | | $[d]_j \leftarrow_{[c]} [a] : [d]_j$;
13 | | $[P]_j \leftarrow_{[c]} [\pi]_i : [P]_j$;
14 | **end**
15 **end**

values (the same follows for the execution time memory usage) and (*iii*) all operators on private data is provided by the $\mathcal{F}_{ABB}$.

Complexity. The algorithm performs $|V|^2 + \mathcal{O}(|V|)$ comparisons (at Lines 6 and 11) and $\frac{4 \cdot |V|^3}{3} + \mathcal{O}(|V|^2)$ multiplications, dominated by Line 7 (the 4/3 factor is 4 times the sum of the square of the integers 1 to $|V|$). This distinction is important for small graph instances where the comparison complexity dominates over round complexity. The performance of our privacy preserving version of Dijkstra has an extra factor of $|V|$ when compared with a vanilla implementation. Moreover, it can also be extended to obtain the shortest path between any pair of vertices $(v, w) \in V$.

4 Minimum Mean Cycle Problem

The Minimum Mean Cycle problem (MMC) is to determine on a directed graph $G = (V, E)$ with edge costs C, the cycle W with the minimum averaged cost (total cost divided by the number of edges in W).

Our interest on the MMC problem comes from the fact that it is used as a subroutine to solve the minimum cost flow problem by the minimum mean cycle canceling algorithm [16]. It is also used by other algorithms of the same nature. More details like applications, proofs and algorithms can be found in [4]. The following analysis assumes strong connectivity on G. In case a graph instance does not provide enough edges to fulfill this requirement, edges with a very large cost can be added to the graph.

The solution we study was proposed by Karp [17] and can be divided in two steps: First, we arbitrarily define a vertex s to be the origin of all paths to all

vertices in V. Let $d^k(i)$ be the smallest weighted walk from s to the vertex i that contains exactly k edges. The walk obtained might contain one or several cycles. Then, we calculate $d^k(v)\ \forall v \in V$ with k from 1 to $|V|$. The following shows how to compute this recursively:

$$d^k(j) = \min_{\{i:(i,j)\in E\}} \{d^{k-1}(i) + c_{ij}\}, \tag{1}$$

where $d^0(s) = 0$ and $d^0(v) = \infty\ \forall v \in \{V - s\}$. Second, we calculate the cost of the minimum mean cycle as follows:

$$\mu^* = \min_{j\in V} \max_{0\le k\le |V|-1} \left[\frac{d^{|V|}(j) - d^k(j)}{|V| - k}\right] \tag{2}$$

This expression can be intuitively explained as follows. Let j^* and k^* the indexes achieving μ^*. Then $d^{|V|}(j^*)$ is the cost of a walk containing the cycle W and $d^{k^*}(j^*)$ is the cost of the same walk with the cycle removed e.g. it is a path. The difference between the two yields the cycle cost. Proofs can be found in [17]. A strictly positive or negative μ^* means that at least a positive/negative cycle is present with μ^* as its mean. A case where the answer is 0 might also mean no cycle was found in the graph. The algorithm can be extended to find the cycle W as part of the answer. Overall algorithmic complexity is $\mathcal{O}(|V||E|)$.

5 Privacy-Preserving Minimum Mean Cycle Solution

The privacy-preserving solution we introduce follows the steps provided by the previous section. Moreover, each step and the whole protocol are designed to be used as sub-routines. As usual, our approach assumes all input data is in secret form, including the adjacency matrix of costs $[C]$ (where non-existing edges are represented by $[\infty]$) except the upper bound on the number of vertices. The final goal of the protocol is to obtain not only the mean cost of the minimum cycle, but the cycle itself as well. We use the function *getmincycle* to refer to the protocol.

Correctness. First, we have to replicate the result of Eq. (1). We select node 1 as the source node s. Implementing the recursion is fairly straightforward as the order in which the edges are scanned does not depend on the input. The more difficult task is to encode the walks. To that end, we define the 4-dimensional matrix $[walk]$ where $[walk]_{i,j,k,l}$ is the number of times the edge (i, j) is traversed by the shortest walk of length k from s to l. Also, because of the specific way we want to use our secure version of the MCC algorithm as a sub-routine, we define an additional argument $[b]$ to the protocol. Specifically, $[b]_{i,j} = 1$ indicates that the edge (i, j) is forbidden, i.e. cannot be part of the solution. The algorithm is detailed as Protocol 3.

Loop 5–8 checks whether edge (i, j) improves the walk of length k from s to j. This is done by comparing the best one found so far with cost $[A]_{jk}$ to $[A]_{ik-1}$

Protocol 3. First step of: MMC protocol based on Karp's algorithm

Input: A matrix of shared costs $[C]_{i,j}$ for $i, j \in \{1, ..., |V|\}$, a binary matrix on viable edges $[b]_{i,j}$ for $i, j \in \{1, ..., |V|\}$.

Output: A matrix of walk costs $[A]_{i,k}$ for $i \in \{1, ..., |V|\}$ and $k \in \{0, ..., |V|\}$, a walk matrix $walks_{ij}$ for $i, j \in \{1, ..., |V|\}$ encoding these walks.

1 $[A] \leftarrow [\infty]$; $[A]_{00} \leftarrow [0]$; $[C] \leftarrow [C] + [\infty](1 - [b])$;
2 **for** $k \leftarrow 1$ **to** $|V| + 1$ **do**
3 **for** $j \leftarrow 1$ **to** $|V|$ **do**
4 **for** $i \leftarrow 1$ **to** $|V|$ **do**
5 $[c] \leftarrow [A]_{ik-1} + [C]_{ij} < [A]_{jk}$;
6 $[A]_{jk} \leftarrow_{[c]} [A]_{ik-1} + [C]_{ij} : [A]_{jk}$;
7 $[walks]_{..kj} \leftarrow_{[c]} [walks]_{..k-1i} : [walks]_{..kj}$;
8 $[walks]_{ijkj} \leftarrow_{[c]} [walks]_{ijkj} + 1 : [walks]_{ijkj}$;
9 **end**
10 **end**
11 **end**

plus the cost of edge (i, j). Depending on the result, the best costs and walks are updated.

Second, we adapt (2) to obtain the value of the minimum mean cycle, as well as the encoding of the cycle. We achieve it by iterating over the matrices $[A]$ and $[walks]$ generated in the *first* step. The only difficulty is to workaround the non-integer division. In place of any costly procedure, we keep track of the numerators and the denominators separately, and compare the cross multiplication instead. The minimum mean cost cycle is encoded as a $|V| \times |V|$ matrix $[min - cycle]$ where $[min - cycle]_{ij} = 1$ if the edge (i, j) is part of the minimum mean cycle. The rest of the algorithm is a straightforward implementation of (2). The details are provided as Protocol 4.

Security. Like with our Dijkstra implementation, no intermediate data is released and the operations are provided by the $\mathcal{F}_{ABB}$, following our definition of security.

Complexity. In total (Protocols 4 and 5), our implementation of MMC requires $\mathcal{O}(|V|^3)$ (Line 5 of Protocol 4) and $\mathcal{O}(|V|^5)$ multiplications or communication rounds (from the conditional assignments of Lines 7 and 8 of Protocol 4). One might ask whether this could not be brought down to $\mathcal{O}(|V|^4)$ by encoding the walks in Protocol 4 as a 3-dimensional matrix holding the predecessor node of each node. However, reconstructing the walks for the operation performed at Line 9 of Protocol 5 would then need $\mathcal{O}(|V|^5)$ conditional assignments instead of the currently $\mathcal{O}(|V|^4)$. So we prefer to stick with our simple and as efficient approach.

6 Minimum Cost Flow Problem

The Minimum-Cost Flow problem (MCF) is of finding a feasible flow in a capacitated directed graph $G = (E, V)$ that minimizes the costs (proportional to the

Protocol 4. Second step of: MMC protocol based on Karp's algorithm

Input: A matrix of walk costs $[A]_{i,k}$ for $i \in \{1, ..., |V|\}$ and $k \in \{0, ..., |V|\}$, a walk matrix $walks_{ij}$ for $i, j \in \{1, ..., |V|\}$ encoding these walks.

Output: The cost of the minimum mean cycle $[min - cost]$. A matrix with the minimum mean cycle $[\text{min-cycle}]_{i,j}$ for $i, j \in \{1, ..., |V|\}$.

for $j \leftarrow 1$ **to** $|V|$ **do**
  $[\text{max-cycle}], [\text{max-cost}] \leftarrow \emptyset$;
  for $k \leftarrow |V|$ **to** 1 **do**
    $[\text{a-num}] \leftarrow [A]_{j(|V|+1)} - [A]_{jk}$;
    $[\text{a-den}] \leftarrow |V| - k$;
    $[c] \leftarrow [\text{k-num}] \cdot [\text{k-den}] < [\text{a-num}] \cdot [\text{k-den}]$;
    $[\text{k-num}] \leftarrow_{[c]} [\text{a-num}] : [\text{k-num}]$;
    $[\text{k-den}] \leftarrow_{[c]} [\text{a-den}] : [\text{k-den}]$;
    $[\text{max-cycle}] \leftarrow_{[c]} [walks]_{..|V|j} - [walks]_{..kj} : [\text{max-cycle}]$;
    $[\text{max-cost}] \leftarrow_{[c]} [A]_{jk} : [\text{max-cost}]$
  end
  $[c] \leftarrow [\text{j-num}] \cdot [\text{k-den}] > [\text{k-num}] \cdot [\text{j-den}]$;
  $[\text{j-num}] \leftarrow_{[c]} [\text{k-num}] : [\text{j-num}]$;
  $[\text{j-den}] \leftarrow_{[c]} [\text{k-den}] : [\text{j-den}]$;
  $[\text{min-cycle}] \leftarrow_{[c]} [\text{max-cycle}] : [\text{min-cycle}]$;
  $[\text{min-cost}] \leftarrow_{[c]} [\text{max-cost}] : [\text{min-cost}]$
end

magnitude of the flows). The problem can be modeled as a linear program but there exists more efficient and well known strongly polynomial time combinatorial algorithms, see [4] for details. The more traditional minimum capacitated cost flow problem can be shown to be equivalent to the transshipment and the minimum-cost circulation (MCC) problem.

Formally, the MCC problem is of finding a capacitated flow in a symmetric graph $G = (E, V)$ of minimum cost. The problem can be modeled as follows:

$$\text{min} \qquad \frac{1}{2} \sum_{v,w \in E} C_{v,w} f_{v,w} \tag{3}$$

$$\text{subject to} \qquad f_{v,w} \leq U_{v,w} \qquad \forall (v,w) \in E \tag{4}$$

$$f_{v,w} = -f_{w,v} \qquad \forall (v,w) \in E \tag{5}$$

$$\sum_{v \in \mathbf{E}(w)} f_{v,w} = 0 \qquad \forall w \in V \tag{6}$$

Here the graph is assumed to be symmetric, i.e. for every $(v, w) \in E$ there is an edge $(w, v) \in E$. Each edge (v, w) has a maximal capacity $U_{v,w}$ and a cost $C_{v,w}$ per unit of flow. Additionally, all costs are antisymmetric, i.e. $c(v, w) = -c(w, v)$ $\forall (v, w) \in E$. The variable f represents the amount of flow passing through an edge. Using this notation, the residual capacity can be formally defined as $r_{v,w} = U_{v,w} - f_{v,w}$.

Constraints (4) are the capacity constraints. Constraints (5) are the flow antisymmetry constraints. Constraints (6) are the flow conservation constraints at each node. This characterization of the problem is the same used by Goldberg and Tarjan [16] for their description of the MCC problem using the Minimum Mean Cycle-Canceling algorithm (MMCC). It can be seen as a variant of the non-polynomial cycle-canceling algorithm proposed by Klein in [18], but where the next cycle to be canceled is chosen by finding the minimum mean cost cycle. The change makes the algorithm strongly polynomial, i.e. its complexity only depends on $|V|$ and $|E|$ and no other parameter.

The algorithm is based on the finding of Busacker and Saaty [19], which asserts that a circulation with no residual negative cost cycles is of minimal cost. Moreover, the algorithm can be characterized as follows:

1. Initialize the feasible circulation of as 0.
2. Obtain the minimum mean cycle W in the associated residual graph.
3. Set $\delta \leftarrow min\{(v, w) \in Wr_{v,w}\}$.
4. Augment the flow by δ along the cycle W.
5. If there are still negative cycles *goto* 2.

Basically, we compute the cycle with the minimum negative average cost W in the associated residual graph. Then, we augment the flow along this cycle until an edge reaches its capacity. This process is repeated until no negative cycle is found. Its complexity is $O(|V|^2 \cdot |E|^3 \cdot \log |V|)$.

7 Privacy-Preserving Minimum-Cost Flow Problem

The input data are the capacity and cost adjacency matrices $[U]$ and $[C]$, where non-existing edges are represented by [0] on the capacity matrix and by $[\infty]$ on the cost matrix. As usual, all input data is secretly shared, except the bound on the number of vertices.The solution is to be provided as the flow matrix $[F]$ and total cost $[totcost]$. The final composition of $[F]$ might leak some details on the graph's topology depending on the answer. The protocol can be used as a sub-routine for more complex applications in case the final output is kept private. Once the MMF problem is modeled as a MCC problem, it is sufficient to securely solve the minimum circulation problem using a privacy-preserving implementation of the MMCC algorithm to obtain a flow of minimum cost.

Adapting the MMCC algorithm. If one wants to avoid any leakage of information, an important difference between a standard implementation and a secure one is that the augmenting flow process has to be repeated as many times as the worst case analysis guarantees, instead of stopping it as soon as no negative cycle is detected. We call each flow augmentation along the cycle a phase/iteration. We use the bound provided by Goldberg and Tarjan on [16]: $|V||E|^2 \log |V| + |V| \cdot |E|$ flow augmentations at most. Note that this is not an asymptotic bound. Given that we also hide the graph structure, $|E|$ has to be replaced by $|V|^2$ in our capacities estimates. Our secure protocol requires to perform that many

iterations to guarantee correctness with no leakage. Possible stopping conditions to reduce the number of iterations are considered later in this section. Protocol 5 shows our privacy-preserving solution for the MMCC algorithm, which is a straightforward translation of the algorithm outlined above.

Protocol 5. Privacy-preserving MMCC

Input: $|V| \times |V|$ matrices of shared capacities $[U]_{i,j}$ and shared costs $[C]_{i,j}$.
Output: The $|V| \times |V|$ matrix of flows $[F]$ and the associated total cost $[totcost]$.

1 $[F], [b], [totcost] \leftarrow 0$;
2 **for** $k \leftarrow 1$ **to** $|V|^5 \log |V| + |V|^3$ **do**
3 $\quad [cost], [cycle] \leftarrow getmincycle([C], [b])$;
4 $\quad \delta \leftarrow [\infty]$;
5 $\quad$ **for** $(i, j) \in [U]$ **do**
6 $\quad\quad [r] \leftarrow [U]_{ij} - [F]_{ij}$;
7 $\quad\quad [c] \leftarrow [min - cycle]_{ij} \cdot ([\delta] > [r])$;
8 $\quad\quad [\delta] \leftarrow_c [r] : [\delta]$;
9 $\quad$ **end**
10 $\quad [\delta] \leftarrow [\delta] \cdot ([cost] < 0)$;
11 $\quad [totcost] \leftarrow [totcost] + [\delta] \cdot [cost]$;
12 $\quad$ **for** $(i, j) \in [F]$ **do**
13 $\quad\quad [c] \leftarrow [cycle]_{ij}$;
14 $\quad\quad [F]_{ij} \leftarrow_c [F]_{ij} + [\delta] : [F]_{ij}$;
15 $\quad\quad [F]_{ji} \leftarrow_c [F]_{ji} - [\delta] : [F]_{ji}$;
16 $\quad\quad [b]_{ij} \leftarrow [U]_{ij} - [F]_{ij} > 0$;
17 $\quad$ **end**
18 **end**

Correctness. The initial solution is set to zero at Line 1. The body of the main loop is one flow augmentation phase. It starts by calling our secure implementation of the Min Mean Cycle problem, leaving out saturated edges. Loop 5-9 computes the maximum augmentation possible along the cycle identified. If the cycle has non-negative cost, this augmentation is set to zero at Line 10, before updating the cost of the solution. Then, the flow itself is augmented at Loop 12-17.

Security. Following the previous protocols, the current solution does not leak intermediate values and uses $\mathcal{F}_{ABB}$ operations to calculate secret data, respecting our definition of security.

Complexity. The most costly operation during one augmentation phase is the call to *getmincycle* with $\mathcal{O}(|V|^3)$ comparisons and $\mathcal{O}(|V|^5)$ communication rounds. The overall complexity is $\mathcal{O}(|V|^8 \log |V|)$ comparisons and $\mathcal{O}(|V|^{10} \log |V|)$ communicational rounds. As mentioned above, one main difference between our secure MCF algorithm described above and a standard implementation is that, to guarantee no leakage of information, we have to execute as many iterations as in the theoretical worst case. This makes the practical performance of the

algorithm much worse than a standard implementation because, in most practical applications, it is expected that the number of iterations needed to find the optimal solution is much smaller than the theoretical upper bound. Of course, one could easily publicly reveal the outcome of the test performed at Line 10 of Protocol 5 and stop the algorithm if the cost of the cycle is non-negative. But some information would be leaked. To limit it, several strategies are possible. One is to open the test every K iterations, with K being a publicly known integer. Another solution is to multiply the result of the test by a random bit (for $= 1$ with probability p) to statistically hide the result. These two would also be combined. In both cases, the parameters (K and/or p) would control the trade-off between performance and information leaked.

8 Computational Experiments

The theoretical bounds only give a rate of increase on the size of the instance. They do not say anything about the actual computing time. Our interest is to determine what is the size of the instances that can be solved in a "reasonable" amount of time. Moreover, we want to determine the impact that the number of players and the size of the graph instances have on CPU time performance. We chose the Virtual Ideal Functionality Framework (VIFF) to run, given its availability (open source) and easy coupling with larger applications, bearing in mind its scalability is an additional concern.

VIFF benefits from passive security under the information theoretic model on the multi-party case. VIFF provides access to Shamir secret sharing and basic arithmetic secure functionality [20]. For comparisons we use the most recent Toft comparison method implemented [21]. Additionally, for our experiments we use randomly generated complete graphs. All results presented are averaged over 20 instances of the same size with 3 and 4 players.

Table 2. CPU time of protocol 2

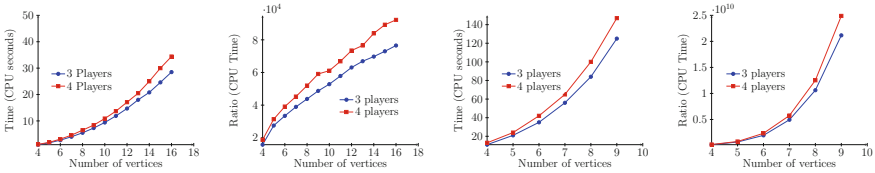
Number of vertices		4	8	12	16	20
Execution times (in seconds)	3 Players	0.9	5	14	28	48
	4 Players	1	7	17	34	57

All trials used the same workstation, an Intel Xeon CPUs X5550 (2.67GHz) and 42 GB of memory, running Mac OS X 10.7. Additionally, every single process had the same amount of CPU power and memory available.

8.1 Shortest Path Problem

Table 2 shows the results obtained by our shortest path prototype: Additionally, we could run 64-vertex instances, using adjacency matrices, with a total of 4032 edges/matrix entries, taking around 18 minutes. The spike in computing time

while working with these big instances follows the fact of the difficulty to manage the memory for large graph instances. Additional experimentation (where we assume the performance cost added by the secure functionalities of our $\mathcal{F}_{ABB}$ to be 0 and implemented on nothing but python) showed that roughly an extra factor of $1.4|V|$ is needed when executing crypto-primitives have 0 cost. Figure 1 also shows the CPU time and the ratios calculated by comparing our Dijkstra prototype against a vanilla implementation of the algorithm:



(a) CPU Time Dijkstra (b) Ratio Dijkstra (c) CPU Time MMCC (d) Ratio MMCC

Fig. 1. Dijkstra CPU times and ratio analysis

From our experimentation we can conclude the following: We can solve securely, in reasonable time, shortest path problems on complete graphs of sizes up to 64 vertices over VIFF. As expected, the number of players, have little incidence on the general behavior, given that in VIFF performance cost increases linearly in the number of players [20]. Compared to the standard implementation, roughly a factor of $5000|V|$ is needed to securely solve the Dijkstra algorithm on VIFF. Combining the previous remark and the results obtained by our experimentation, we conclude that out of the $5000|V|$ overhead of our SMC implementation, the factor $|V|$ is explained by algorithmic design, a factor 1.4 is due to non-crypto related VIFF implementation, and the rest (a factor of a few thousands) is due to the crypto-related VIFF implementation.

8.2 Minimum Flow Problem

For the minimum flow problem, we measure the time a single phase (one iteration of Protocol 5) takes to be executed, that is because stopping conditions with some leakage can substantially reduce the number of phases needed e.g. A graph with a single cycle would only take one phase to be completed. To estimate the execution time of the full algorithm, it suffices to multiply this by the known number of phases needed. Our analysis includes the ratio between the time it takes a vanilla implementation to find an answer and the privacy preserving versions full execution time to guarantee correctness with no leakage. The results of these experiments can be found in Table 3 and Fig. 1.

From these we can conclude the following: The fully secure version of our implementation is highly costly in terms of performance even for very small instances. This highlights the necessity of using termination conditions. Once again, the influence of the extra player has little incidence on the overall performance time. The overhead of our secure implementation versus a standard one is

Table 3. Execution times per phase MMCC Algorithm for a complete graph.

Number of vertices		4	5	6	7	8	9
Execution times (in seconds)	MMCC Phase - 3 Players	11	21	35	56	84	125
	MMCC Phase - 4 Players	13	24	42	65	100	147

of the order of $2.5 \cdot 10^8 |V|^2$. Note that both algorithms have different complexity functions and vanilla versions of the algorithm typically converge towards an answer before reaching its worst case complexity. Again, one can observe that the multiplications absorb a larger fraction of the computing time as the size of the instances increases.

9 Conclusions and Future Work

Strongly polynomial-time algorithms are appealing for MPC implementations because, as the worst-case complexity is polynomial, it is possible to obtain fully secure (i.e. no leakage) and theoretically efficient algorithms and implementations. We have demonstrated this for three classical network problems: Shortest Path, Minimum Mean Cycle and Minimum Cost Flow. However, our computational experiments demonstrate that the price to pay for such security is very high for the simplest problem (shortest path) and extremely penalizing for the more complicated ones.

This research raises several questions for further research. A first one is whether theoretically more efficient algorithms can be obtained for these problems. Another one is related to the development of more efficient MPC platforms compared to the one we used for our computational experiments. Also one could consider other classical optimization problems.

Acknowledgements. This research was supported by the WIST Walloon Region project CAMUS and the Belgian IAP Program P7/36 initiated by the Belgian State, Prime Minister's Office, Science Policy Programming. The scientific responsibility is assumed by the authors. The authors are grateful to Edouard Cuvelier, Sophie Mawet, Olivier Pereira and the anonymous reviewers for their feedback.

References

1. Yao, A.C.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science, pp. 160–164. IEEE (1982)
2. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
3. Paillier, P.: Public-Key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, p. 223. Springer, Heidelberg (1999)
4. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice-Hall Inc., Upper Saddle River (1993)

5. Aly, A., Cuvelier, E., Mawet, S., Pereira, O., Van Vyve, M.: Securely solving simple combinatorial graph problems. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 239–257. Springer, Heidelberg (2013)
6. Launchbury, J., Diatchki, I.S., DuBuisson, T., Adams-Moran, A.:Efficient lookup-table protocol in secure multiparty computation. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, pp. 189–200. ACM, New York (2012)
7. Brickell, J., Shmatikov, V.: Privacy-preserving graph algorithms in the semi-honest model. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 236–252. Springer, Heidelberg (2005)
8. Blanton, M., Steele, A., Alisagari, M.: Data-oblivious graph algorithms for secure computation and outsourcing. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, ASIA CCS 2013, pp. 207–218. ACM, New York (2013)
9. Wang, X., Nayak, K., Liu, C., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. Cryptology ePrint Archive, Report 2014/185 (2014). http://eprint.iacr.org/
10. Lu, S., Ostrovsky, R.: Distributed oblivious ram for secure two-party computation. Cryptology ePrint Archive, Report 2011/384 (2011). http://eprint.iacr.org/
11. Gordon, S.D., Katz, J., Kolesnikov, V., Krell, F., Malkin, T., Raykova, M., Vahlis, Y.: Secure two-party computation in sublinear (amortized) time. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012, pp. 513–524. ACM, New York (2012)
12. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.: Automating efficient ram-model secure computation. In: 35th IEEE Symposium on Security and Privacy (2014)
13. Keller, M., Scholl, P.: Efficient, oblivious data structures for mpc. IACR Cryptology ePrint Archive, 137 (2014)
14. Maurer, U.: Secure multi-party computation made simple. Discrete Appl. Math. **154**(2), 370–381 (2006). Coding and Cryptography
15. Damgård, I.B., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 247–264. Springer, Heidelberg (2003)
16. Goldberg, A.V., Tarjan, R.E.: Finding minimum-cost circulations by canceling negative cycles. J. ACM **4**, 873–886 (1989)
17. Karp, R.M.: A characterization of the minimum cycle mean in a digraph. Discrete Math. **3**, 309–311 (1978)
18. Klein, M.: A primal method for minimal cost flows with applications to the assignment and transportation problems. Manag. Sci. **14**(3), 205–220 (1967)
19. Busacker, R., Saaty, T.: Finite Graphs and Networks: An Introduction with Applications. International Series in Pure and Applied Mathematics. McGraw-Hill, New York (1965)
20. Geisler, M.: Cryptographic protocols: theory and implementation. Ph.D. thesis, Aarhus University Denmark, Department of Computer Science (2010)
21. Toft, T.: Primitives and applications for multi-party computation. Ph.D. thesis, Department of Computer Science, Aarhus University (2007)