DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
41

MOHAMMAD ANAGREH

Privacy-Preserving Parallel Computations for Graph Problems

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

41

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
41

MOHAMMAD ANAGREH

Privacy-Preserving Parallel Computations for Graph Problems

UNIVERSITY OF TARTU
Press
1632

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in Computer Science on April 14, 2023 by the Council of the the Institute of Computer Science, University of Tartu.

Supervisor(s)

Prof. Dr. Eero Vainikko
University of Tartu
Tartu, Estonia

Dr. Peeter Laud
Cybernetica AS
Tartu, Estonia

Opponents

Prof. Dr. Shin'ichiro Matsuo
Georgetown University
Washington, D.C., United States of America

Prof. Mustafa A. Mustafa
University of Manchester
Manchester, United Kingdom

The public defense will take place on 29 May, 2023 at 14:15 in Narva mnt 18-1019.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

ISSN 2613-5906 (print) ISSN 2806-2345 (PDF)
ISBN 978-9916-27-217-6 (print) ISBN 978-9916-27-218-3 (PDF)

University of Tartu Press
http://www.tyk.ee/

ABSTRACT

The round complexity is one of the most critical challenges in the exciting Secure Multiparty Computation (SMC) protocols for general functionalities. SMC/MPC is a protocol that jointly computes a function privately over private inputs while each party knows his input and output and does not learn about others. Using these protocols in a real-world application computation with high round complexity is more crucial. This is a problem for secret-sharing based SMC protocol sets which require interaction between computation parties for each operation. Even though secret-sharing SMC protocols can be the most efficient protocol sets available, this efficiency only materializes for highly parallel tasks. The reduction of round complexity will reduce the network latency in the SMC protocol, which is a challenge to many researchers in the SMC platform. One of the most efficient strategies to reduce the round complexity in SMC is parallel calculation; this is the scope of this Thesis's research. The advancement of the privacy-preserving implementation of the shortest distances and minimum spanning tree algorithms has motivated researchers to make the privacy-preserving computation of the graph algorithms usable. Performing such a calculation for a big graph is too costly because of the network latency in the SMC platform. In this work, we perform privacy-preserving parallel computations for various graph algorithms that can be utilized in real-world applications relying on shortest path and minimum spanning tree algorithms.

In detail, we study secure multiparty computation protocol for Single-Source Shortest Distances (SSSD), All-Pairs Shortest Distances (APSD), Minimum Spanning Trees (MST), and sparse-linear systems with a semiring framework. The Single-Instruction-Multiple-Data (SIMD) approach is implemented to reduce the round complexity of the SMC protocol, which in turn reduces the network latency among the parties of the SMC platform. We present state-of-the-art privacy-preserving parallel computations of Bellman-Ford, Dijkstra, Breadth-First search, Radius-Stepping SSSD protocols, and Johnson APSD protocol benchmarked with algorithms, Floyd-Warshall and transitive closure of a graph. On the other hand, we present the state-of-the-art privacy-preserving parallel computation of Prim's minimum spanning tree for the dense graph. Moreover, this Thesis proposed the first secure multiparty parallel computation of Minimum Spanning Forest (MSF) protocol based on Prim's algorithm for dense graphs. The fourth part of our work presents the state-of-the-art privacy-preserving Algebraic path parallel computation protocol. The subroutines of the proposed protocol can be used to solve different problems in sparse-linear systems with a semiring framework.

We extensively benchmark our protocols and their related functions on graphs of different sizes in different network environments, allowing for a reasonable estimate of their performance, including for more extensive applications with special shortcuts. Some of our proposed parallel protocols exhibit a speed-up of thousands of times, making them faster than any protocol in previous works.

CONTENTS

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF TABLES

LIST OF ABBREVIATIONS

Acronyms

ABB Arithmetic Black Box. 22
APC Algebraic Path Computation. 23
APSD All-Pairs Shortest Distance. 23

BFS Breadth-First Search. 23
BGW Ben-Or, Goldwasser and Wigderson. 22

CPU Central Processing Unit. 44

EREW Exclusive Read Exclusive Write. 46

FPGA Field-programmable Gate Array. 44
FS Frontier Stack. 40

GPS Global Positioning System. 20
GPU Graphics Processing Unit. 44

HBHL High-Bandwidth High-Latency. 63
HBLL High-Bandwidth Low-Latency. 63
HSI Hyperspectral Imaging. 20

LBHL Low-Bandwidth High-Latency. 63

MPC Multi-party Computation. 21
MSF Minimum Spanning Forest. 24
MST Minimum Spanning Tree. 22

NS Next Frontier Stack. 40

ORAM Oblivious Random Access Machine/Oblivious RAM. 22

PRAM Parallel Random Access Machine. 23

QoS Quality of Service. 49

SCN Supply-Chain Network. 47
SIMD Single-Instruction-Multiple-Data. 22
SMC Secure Multiparty Computation. 20
SP Shortest Path. 22
SSSD Single-Source Shortest Distance. 23

ST Spanning Tree. 35

TSP Travelling Salesman Problem. 46

UB Universal Composability. 31
UBFS Unweighted Breadth-First Search. 74

VIFF Virtual Ideal Functionality Framework. 22

WBFS Weighted Breadth-First Search. 74

LIST OF NOTATIONS

$\mathbb{R}$	Set of real numbers
$\mathbb{N}$	Set of natural numbers
$\mathbb{A}$	Set of adversaries
$\mathbb{Z}$	Set of set of integers
$\mathcal{A}$	Adversary
$\mathcal{A}_s$	Semi-honest adversary / Passive adversary
$\mathcal{A}_m$	Malicious adversary / Active adversary
$\mathcal{A}'$	Corresponding adversary
$\mathcal{F}$	Ideal functionality
Π	Protocol
Π'	Prescribed protocol
Ξ	Another protocol
Env	Environment
Sim	Simulator
$[\![x]\!]$	Single private value
$[\![\vec{x}]\!]$	Private vector
$[\![\vec{x}']\!]$	Private permuted vector
$[\![\mathbf{X}]\!]$	Private adjacency matrix
$\vec{M}$	Public vector
$\mathcal{F}_{\text{ABB}}$	Ideal functionality for the ABB
x_i	Private input to ABB
y_i	Private output from ABB
op	Operation in ABB
$\mathcal{G}$	Ideal functionality
$\mathcal{T}_p$	Trusted party
G	Private graph
$\mathbf{A}$	Adjacency matrix
V	Set of vertices
E	Set of edges
W	Set of weights
u	Vertex
v	Starting vertex
e	Edge
w	Weight
n	Number of vertices
m	Number of edges
g	Number of sub-graphs / Number of components
t	Threshold
$\mathcal{P}_i$	Computation party

$\mathcal{M}_i$	Turing machine
$\mathcal{DV}_i$	data volume
s_i	Secret share
δ	Path
$\delta(\cdot)$	Shortest path
$[\![\sigma]\!]$	Private permutation
$[\![\mathbf{G}']\!]$	Permuted private adjacency matrix
u'	Permuted vertex
v'	Permuted another vertex
r	Radius
Δ	Delta
@	Concatenation of lists
$\otimes$	Addition (multiplicative operation in a tropical semiring)
$\oplus$	Minimum (additive operation in a tropical semiring)
X_h	Sub matrix
Y_h	Another sub matrix
Z_h	Another sub matrix
W_h	Another sub matrix
U_h	Another sub matrix
Y_h^T	Transpose of matrix Y_h
A^{-1}	Inverse matrix
$\langle\langle A \rangle\rangle$	A sparse representation of matrix A
A^*	Quasi-inverse of a matrix A
h	Level of the partitioning
d	Depth of the separator tree
F_k	Serial fraction
$\mathcal{T}_p$	Running time of parallel algorithm
$\mathcal{T}_s$	Running time of sequential algorithm
$\mathcal{T}$	Average of running time
p	Number of processors or threads
$\mathcal{DV}$	Average of data volume
T_1	Time to execute 1 graph
T_k	Time to execute k graphs
$\star$	All elements of a vector
k	Multiplication by 1000
M	Multiplication by 1000000

1. INTRODUCTION

1.1. Justification of research

A method has been proposed to implement privacy-preserving computation—secure multiparty computation, but this method is computationally costly [49, 153]. Privacy-preserving parallel algorithms are needed to expedite the processing of large private data sets for graph algorithms and meet high-end computational demands. In contrast, privacy-preserving parallel computation of graph algorithms has not been studied due to the novelty of the technology and the high computational costs associated with such issues, mainly when dealing with large data sets. Constructing real-world privacy applications based on secure multiparty computation is challenging due to the round complexity of the Secure Multiparty Computation (SMC) protocol [103]. The round complexity problem of SMC protocol can be solved using parallel computing [56]. This thesis is interested in providing solutions to various graph algorithms problems.

The combinatorial and algebraic graph algorithms can model various computer science applications, e.g., community detection (social media networks and contact tracing system of COVID-19), Global Positioning System (GPS)/Google maps, navigation systems, supply chain networks [141, 171], data mining, bioinformatics, numerical analysis and Hyperspectral Imaging (HSI) [52]. Moreover, parallel graph algorithms are required, for example, to perform efficient aggregation-based coarsening techniques in parallel preconditioners [149]. Statistical methods also require the solutions of large sparse systems of linear equations without knowing each party's full matrix. Therefore, graph algorithms are fundamental building blocks in computer science applications, highlighting the need for privacy-preserving parallel computation of graph algorithms to process large data sets with varying shapes, including sparse, dense, and planar graphs, with high computational requirements.

1.2. Research problems

The recent advances in privacy-preserving computation have opened up new possibilities for developing practical real-world applications [158]. Combinatorial graph algorithms play a critical role in solving various research problems across different fields. Most graph algorithms follow a greedy approach, consisting of several stages to construct the solution. However, the complexity of graph algorithms can become problematic, particularly when they require processing large datasets. Therefore, there is a pressing need to develop efficient algorithms for performing these computations while preserving privacy.

The usual approach to privacy-preserving combinatorial graphs algorithms is the implementation of classical algorithms [58] on top of some SMC protocol sets. At the same time, such algorithms contain a high number of iterations, with

the distribution of copies of input data among several parties handled through the standard means of this protocol set. However, such an approach for privacy-preserving combinatorial graphs algorithms and an alternative approach [66], i.e., privacy-preserving algebraic graph algorithms, is made more complex by several aspects of the problem itself. Moreover, the typical algorithms should be suitably adapted to run on top of an SMC protocol set. This approach to privacy-preserving combinatorial graphs has been reported in [5, 3]. The implementations are for Bellman-Ford and Dijkstra algorithms. The permutation operation is also used to mask the real identity of the vertices. This means an extra process will increase overhead, the method does not show scalability, and the graphs are too small.

Generally, the computations on top of the SMC protocol have a high amount of communication during running a task when the data input is distributed among the computation parties of the Multi-party Computation (MPC) platform, causing network latency overhead that is computationally intensive synchronous, and sequential. Consequently, the round complexity of a computation involving complex iterations in secure multiparty computation protocols is a worthy goal to be reduced. Only a little research has been effectively done on such problems, either over combinatorial or algebraic graph algorithms. Moreover, algorithms have been proven challenging to parallelize efficiently. The challenge will worsen using big graphs, which means high round complexity (High running time in tens of minutes, hours, or days). These problems make designing real-world privacy applications somewhat challenging.

1.3. Background of research

One of the most critical challenges in constructing privacy-preserving real applications is the round complexity of the SMC protocol. Secret-sharing based SMC protocol sets that require communication between computation parties are the focus of this research. Although secret-sharing SMC protocols can be among the most efficient protocol sets, their efficiency only manifests when highly parallel tasks are addressed. Some studies were conducted to improve the round complexity problem [103, 102, 44]. The problem of round complexity can be solved using parallel computation; some research improved this problem using parallel calculation [43, 56]. Nevertheless, the gap is still significant for building real-world applications that can operate large private data sets.

The research in this thesis focuses on studying and parallelizing different graph algorithms on top of the secret-sharing based SMC protocols [112]. Therefore, the work introduces novel methods in the privacy-preserving parallel computation of some combinatorial [74] or algebraic [66] graphs algorithms. The computation of graph algorithms using privacy-preserving techniques, whether combinatorial or algebraic, is complex when implemented on top of SMC protocols due to the mentioned round complexity. However, the round complexity can be reduced if an efficient privacy-preserving parallel computation of graph al-

gorithms can be constructed using the Single-Instruction-Multiple-Data (SIMD) approach. Single-Instruction-Multiple-Data is a parallel computation technique used to achieve data-level parallelism by executing the same instruction on multiple data points at the same time [72, 73]. SIMD framework is under SecreC high-level language [145]. SecreC is language-based security for secure multiparty computation applications. Generally, constructing efficient algorithms can be done by Parallel computing that will be used to reduce the number of iterating or by using alternative algorithms that solve the same task [95].

The secure multiparty computation protocols provide secure implementations for the Arithmetic Black Box (ABB) [61] abstraction, inside which the privacy-preserving operations are performed without leaking anything about the results of the main and intermediate computations. The functions of the ABB use its internal, private memory to store the data during the processing [112]. Each operation incurs significant latency due to the communication between the computation parties $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ of the SMC Sharemind [27] platform. Much research has already discussed various issues in privacy preservation topics, e.g., Wearable Technology in a review study [100], a survey on the Internet of Vehicles [77], and data mining [2, 118, 120, 127]. Recently, several methods and techniques have been developed for privacy-preserving computations in various fields over the Sharemind SMC platform, such as data mining problem [28], genotyping techniques [99, 134, 36], logic programming [139, 93], statistical data analysis [37, 39, 38, 33, 35, 98].

The classical graph algorithms we target in this work are the Shortest Path (SP) and Minimum Spanning Tree (MST) over different graphs' kinds, planar, sparse, and dense with various sizes. The gap in the round complexity problem will increase if we apply the protocols for computation with large graphs. Extensive benchmarking results for SMC of shortest path and minimum spanning tree algorithms have yet to be reported. Implementations of Dijkstra's and Bellman-Ford algorithms are documented [5], the implementations on top of Virtual Ideal Functionality Framework (VIFF) [78] with Ben-Or, Goldwasser and Wigderson (BGW) protocol [168] used for multiplication and Toft's protocol [160] for comparison. However, the implementations only on the dense representation of small graphs. Keller and Scholl [104] have implemented the operations of Oblivious Random Access Machine/Oblivious RAM (ORAM) on top of the SPDZ protocol set [63] and used them to implement a privacy-preserving version of Dijkstra's algorithm. The graph they used in the implementation is sparse and dense, and the implementation is suitable for sparse more than dense. Another implementation of Dijkstra's algorithm on small graphs is produced in [3], but no result was shown for big graphs. Using garbled circuits, an implementation for Dijkstra's algorithm is evaluated [50]. This parallel implementation handles 32 circuits simultaneously on a 32-core server. The result shows that the running time for 20-vertex graphs is 26 seconds, while for 15 minutes 100-vertex graphs, the parallel implementation is not scalable and unsuitable for running big graphs. Similarly, garbled

circuits are used to evaluate Dijkstra's algorithm on sparse graph [125]. To increase efficiency, oblivious priority queues are employed [165]. The estimate is that together with JustGarble [16]. Blanton et al. [25] provided data-oblivious algorithms for the shortest path for secure computation; they theoretically introduced their work without showing actual implementations. All previous works present protocols on single-source shortest distances by applying some combinatorial shortest path algorithms; no such works were implemented before for All-Pairs shortest distances. Another technique for finding the shortest distances is Algebraic Path Computation (APC) [70], which also has no such implementation in privacy-preserving.

In the case of preserving privacy in the minimum spanning trees, previous works have not efficiently provided solutions for various large graphs. Peeter Laud implemented a privacy-preserving computation for minimum spanning tree using Parallel Random Access Machine (PRAM) [114]. His work is efficient for a sparse graph where the number of edges reaches six times more than vertices, while for a dense graph, the algorithm is inefficient for big graphs. Rao and Singh [146] presented privacy-preserving MST sequentially by implementing two algorithms separately, Kruskal and Prim. There is no actual implementation in their work. The minimum spanning forest, which is a minimum spanning tree n times, has not been implemented in secure multiparty computation.

In detail, the thesis focuses on constructing a secret sharing based secure multiparty computation protocols Single-Source Shortest Distance (SSSD) based on some shortest paths algorithms separately, Radius-Stepping [26], Dijkstra [65], Bellman-Ford [17], Breadth-First Search (BFS) algorithms [15] in sparse and dense graphs. As well as privacy-preserving All-Pairs Shortest Distance (APSD) based on Johnson [96], Floyd-Warshall [71] and the transitive closure of the graph. Including the shortest path algorithms, this thesis also studies Prim's minimum spanning tree [143] and forest algorithms in privacy-preserving parallel computation. The last part of the study targeted the sparse-linear solver issues, i.e., algebraic graph computation and their related algorithms min, sum operations in semiring structure [81] and block diagonal matrix.

Accelerating the computations on top of SMC protocols using parallel computation is possible, but there can be significant variance between the sequential and parallel versions of the computation. Our hypothesis is summarized below.

Efficient protocols with low round complexity for graph algorithms on top of SMC protocols can be constructed using SIMD parallel computation. The shortest path algorithms used to build these efficient protocols on top of SMC include Radius-Stepping, Bellman-Ford, Dijkstra, BFS for SSSD algorithms, and Johnson's algorithms for APSD. Prim's algorithm also enables the construction of minimum spanning tree and forest protocols on top of SMC. Additionally, algebraic path computation and related algorithms can be parallelized and efficiently implemented on top of the SMC Sharemind platform.

1.4. Significance of research

The study investigated various parallel methods for graph algorithms, including minimum spanning trees, shortest distances, and algebraic paths. These algorithms are efficiently implemented on top of secret-sharing based SMC protocols, and many parts of this intensive work have already been completed with exciting results. This thesis outlines novel parallel methods, along with the results, speed-up, evaluations, and benchmarking. It also describes the state-of-the-art privacy-preserving implementations of the shortest distance and minimum spanning tree algorithms. Moreover, this is the first work to use such large datasets in graph computations compared to previous studies. This work investigated the first privacy-preserving computation of APSD algorithms, and the evaluation of APSD algorithms is not similar to any prior work.

Moreover, this work proposed the first privacy-preserving computation of Minimum Spanning Forest (MSF) protocol, which is efficient for dense and sparse graphs. The privacy-preserving MSF topic has never been addressed before due to the high computational cost of finding MSF. The methods we proposed and implemented in this work make some shortest path and minimum spanning tree algorithms efficiently work over dense or sparse graphs (no effectiveness in the number of edges). We evaluated and compared our privacy-preserving shortest path protocols with each other. The privacy-preserving BFS protocol is the most efficient protocol on sparse and dense graphs, with $\log n$ round complexity. The unweighted version of the BFS protocols has constant round complexity $O(1)$. Thus, the implementations of BFS can be used as subroutines for some algorithms on top of SMC, such as maximum flow algorithms. The speed-up in some of our new algorithms is thousands of times compared to the others' works. The parallel methods we presented are novel. The protocols obtained running times faster than almost all work done before. The algebraic path computation is the first such work that has never been done before on top of SMC protocols, thereby obtaining parallel methods for some related computations of the algebraic graphs that can be used to build up other implementations.

1.5. Research contributions

The purpose of the research presented in this thesis is to study how to reduce the computational cost of different computations in privacy-preserving, as well as to identify a suitable set of privacy-preserving subroutines and the manners of their combination, resulting in arguably as efficient as possible privacy-preserving implementations of common SSSD, APSD, and MST algorithms. The thesis presents the following research contributions:

1. Proposed state-of-the-art protocols in privacy-preserving parallel single shortest paths for dense and sparse graphs.
    - A new protocol for privacy-preserving parallel computation of a single-

source shortest distance based on the radius-stepping algorithm. The performance of this protocol is compared with the privacy-preserving implementation of the sequential radius-stepping and Δ-stepping algorithms [129]. The series of tests consider sparse and dense graphs, as well as graphs whose number of edges (for the given number of vertices) is similar to planar graphs, where the number of vertices and the number of edges is public. Still, the identities of vertices, edges, and weights are private.

- A privacy-preserving implementation of the Bellman-Ford SSSD algorithm for sparse graphs, where the number of vertices and edges is public. Still, the endpoints and lengths of edges are private. An implementation with this set of features was presented before by Keller [104], using heavyweight constructions for oblivious RAM (ORAM) on top of SMC protocols. Our implementation uses the parallel oblivious reading subroutine by Laud [114], which is an excellent fit for the Bellman-Ford algorithm.
- A novel method for a necessary privacy-preserving subroutine of the Bellman-Ford algorithm—computing the minima of several lists of private values, where the lengths of individual lists are private, and only their total length is public.
- A privacy-preserving implementation of Dijkstra's SSSD algorithm for dense graphs, where the number of vertices is public, but the lengths of edges are private. While implementation with this set of features has been given before [3], we make use of state-of-the-art subroutines for all parts of the algorithm, thereby learning its actual performance.
- Another version of the privacy-preserving Dijkstra's SSSD implementation is for simultaneously finding the shortest distances for multiple graphs.
- A state-of-the-art parallel privacy-preserving shortest path protocol for weighted and unweighted graphs. The two versions of the proposed protocol are based on a breadth-first search algorithm. The implementation of the proposed protocols was tested using different graphs, dense and sparse, represented as the adjacency matrix. The results show that this protocol is the fastest for finding SSSD compared to other proposed protocols.

2. A privacy-preserving implementation of the Johnson APSD algorithm, converting the graph from a sparse one to a dense one in the process. The two privacy-preserving protocols of the Dijkstra and Bellman-ford are combined to produce the Johnson APSD protocols. The protocol also uses the parallel oblivious reading and writing subroutine by Laud [114]. The protocol's performance is extensively compared with the privacy-preserving im-

plementation of the Floyd-Warshall APSD algorithm and with the private computation of the transitive closure of the graph.

3. Proposed state-of-the-art protocols in the privacy-preserving parallel minimum spanning tree and forest.
    - A state-of-the-art secret-sharing based secure multiparty computation (SMC) protocol for computing Prim's minimum spanning trees in dense graphs. The implementation uses the parallel oblivious reading subroutine by Laud [114].
    - Created the first privacy-preserving protocol of the minimum spanning forest for sparse and dense graphs. Two versions of the MSF protocol have been proposed, sequential and parallel. The MSF protocols use the optimized version of the privacy-preserving prims' MST protocol.
4. Proposed state-of-the-art protocol in privacy-preserving algebraic path parallel computation. 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.
    - An optimized version of the privacy-preserving SSSD Bellman-Ford protocol. The values of the edges are public, while the values of the weights are private. This protocol is benchmarked with algebraic path computation for different graphs.
5. An extensive benchmarking of the proposed protocols and their parts on graphs of different sizes is widely performed, thereby obtaining a reasonable estimate for their performance in larger applications, including those where specific shortcuts (e.g., not running the whole number of iterations) are justified. The proposed protocols in this work were implemented on top of secure multiparty computation protocols over different networks.

1.6. Thesis outline

The author has designed different protocols and related algorithms to solve various problems in graph algorithms and algebraic path computation on secure multiparty computation protocols. The author's primary strategy is to use single-instruction multiple data to efficiently reduce the high computational cost of such greedy algorithms on the SMC protocol set. This can be applied to different algorithms, not limited to those proposed in this work. Consequently, the proposed protocols in this work can serve as case studies or examples for implementing SIMD parallel computation in a privacy-preserving manner, efficiently reducing the number of rounds in SMC protocols. This thesis is organized into different sections or chapters and includes extensive benchmarking, experiments, and analysis showing the efficient privacy-preserving parallel computation of graph pro-

tocols. These protocols can be used in building real-world applications in various technology fields. This thesis is outlined as follows:

Chapter 2 gives an overview of secure multiparty computation and its related issues. This chapter also briefly summarises graph theory and some shortest path and minimum spanning tree algorithms. A summary is outlined for algebraic graph theory, specifically, Algebraic path computation and semiring framework. Furthermore, we describe the essential algorithms of our proposed privacy-preserving SSSD, APSD, MST, and MSF protocols and highlight the closer related works in the parallel and sequential calculation for graph algorithms in privacy-preserving and non-privacy-preserving.

Chapter 3 describes and discusses the methods and related issues in performing the research, tools, and materials. An overview of the Sharemind framework, the SecreC high-level programming language, and their parallel framework. The chapter has subsections for presenting the empirical research and methodology, research procedures, the methods of analyzing data, the abstraction, and the notations of the SMC protocol, as well as the experiments set up and the architecture of the SMC platform we use in the implementation.

Chapter 4 describes the proposed protocols and their subroutines for computing the single-source shortest distances for sparse, dense, and planar graphs. The chapter expressly represents the six protocols of privacy-preserving SSSD and their related algorithms. First, we present the privacy-preserving Dijkstra's protocol and the privacy-preserving Bellman-Ford versions of the protocol. These contributions are based on the previous publication of the author [8].

- Anagreh, M., Laud, P. and Vainikko, E.: 2021. Parallel Privacy-Preserving Shortest Path Algorithms. Cryptography, 5(4), p.27(2021).

The chapter also presents another single-source shortest distance protocol, the privacy-preserving radius-stepping published in [12].

- Anagreh, M., Vainikko, E. and Laud, P.: Parallel Privacy-Preserving Shortest Paths by Radius-Stepping. In: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE. 2021, pp. 276–280(2021).

Finally, the last protocol presented in this chapter is privacy-preserving BFS's shortest distances; the protocol has two different versions for dealing with weighted graphs and the second for an unweighted graph. This contribution is based on the previously published paper [10].

- Anagreh, M., Laud, P. and Vainikko, E.: 2022. Privacy-Preserving Parallel Computation of Shortest Path Algorithms with Low Round Complexity. In Proceedings of the 8th International Conference on Information Systems Security and Privacy - ICISSP, ISBN 978-989-758-553-1, pages 37-47.

As well as this chapter presents the round and communication complexities of the proposed protocols and comparisons among them. Furthermore, it shows the security and privacy of the proposed protocols. We discuss the security proof of Bellman-Ford's SSSD as an extending operation in ABB.

Chapter 5 presents how to use the privacy-preserving Dijkstra and Bellman-Ford protocols to build the parallel version of the privacy-preserving Johnson All-pairs shortest distances protocol. As well presents the privacy-preserving Floyd-Warshall, and transitive closure of the graph on top of SMC protocols to be used in benchmarking them with the Johnson APSD protocol. This contribution is based on the published paper [8].

- Anagreh, M., Laud, P. and Vainikko, E.: 2021. Parallel Privacy-Preserving Shortest Path Algorithms. Cryptography, 5(4), p.27(2021).

Chapter 6 describes the privacy-preserving parallel computation for Prim's minimum spanning tree and forest protocols. In detail, the chapter presents a secret-sharing based secure multiparty computation (SMC) protocol for computing, Prim's MST. This contribution is based on published publications for the author [11] in the following:

- Anagreh, M., Vainikko, E. and Laud, P.: 2021. Parallel Privacy-preserving Computation of Minimum Spanning Trees. In Proceedings of the 7th International Conference on Information Systems Security and Privacy - ICISSP, ISBN 978-989-758-491-6; ISSN 2184-4356, pages 181-190.

This chapter also presents the privacy-preserving parallel computation of the minimum spanning forest protocol, which has two versions, sequential and parallel. The creation of the MSF protocol is based on the optimized version of MST protocol, which is also presented in this chapter. These contributions are based on the published paper [9] in the following:

- Anagreh, M., Laud, P. and Vainikko, E.: 2022. Privacy-Preserving Parallel Computation of Minimum Spanning Forest. SN Computer Science Journal, 3(6), pp.1-19.

Chapter 7 describes Algebraic path computation and their related algorithms, which are all presented for sparsely represented graphs. The main calculation of the algebraic path, the min and sum semiring structure operations, and the block diagonal matrix, which is presented sparsely, are all illustrated. Moreover, this chapter presents the third version of the Bellman-Ford protocol for finding SSSD with public edges for undirected graphs. The contributions are based on the published paper [7] entitled:

- Anagreh, M. and Laud, P.: 2023. A Parallel Privacy-Preserving Shortest Path Protocol from a Path Algebra Problem. In Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2022 International Workshops, DPM 2022 and CBT 2022, Copenhagen, Denmark, September 26–30, 2022, Revised Selected Papers (pp. 120-135). Cham: Springer International Publishing.

Chapter 8 describes the experiments, the data input, result testing, and analysis for the proposed privacy-preserving SSSD protocols. The experiments are based on the protocols' running time and data volume. In detail, it presents the benchmarking result in previous work, the investigations of the SSSD protocols separately with their related algorithms, and compares all protocols. As well as present the experiments of Johnson APSD protocols with Floyd-Warshall and transitive closure of the graph in privacy preservation. The last section presents the evaluation of the APSD protocols depending on running time, data volume, and over different network environments.

Chapter 9 presents the benchmarking result in previous works, privacy preservation versions experiments of minimum spanning tree protocols and minimum spanning forest protocols, sequential and parallel. The experiments used various graph sizes in different network environments. As well as the evaluation of the privacy-preserving MST prim's protocols with previous work is presented.

Chapter 10 describes the benchmarking result for the privacy-preserving Algebraic path computation with its related algorithms that are represented sparsely in parallel. As well as describe the benchmarking result for the privacy-preserving Bellman-Ford Version-3, which has public edges. Moreover, it presents the evaluation of both protocols using various graphs sizes in different network environments.

Chapter 11 describes the work's conclusions, future work, and recommendations.

2. BACKGROUND AND RELATED WORKS

2.1. Secure Multiparty Computation

Secure multiparty computation (SMC/MPC) is a cryptographic protocol that allows n parties to compute a function $(y_1,\ldots,y_n) = \mathcal{F}(x_1,\ldots,x_n)$, with the party $\mathcal{P}_i$ submitting x_i and learning y_i [60]. Moreover, the i-th party does not learn anything beyond x_i and y_i, and whatever can be deduced from them, considering the public description of the function $\mathcal{F}$. More generally, there is a downwards closed set of subsets of $\{1,\ldots,n\}$, such that for any element I of this set, the coalition of parties $\{\mathcal{P}_i\}_{i\in I}$ learns nothing beyond their inputs $(x_i)_{i\in I}$ and their outputs $(y_i)_{i\in I}$ during the protocol computing $\mathcal{F}$. Generic protocols for secure multiparty computation translate a computation represented as a boolean or arithmetic circuit into a cryptographic protocol [172, 53, 82]. There exist several different approaches to executing the circuit or program of the function $\mathcal{F}$ in a privacy-preserving manner, including garbled circuits [172], homomorphic encryption [61, 89], or secret sharing [79, 46, 62], and offering security either against passive or active adversaries [138]. The secret sharing approach can accept an adversary who is rolling up t of the total n computation parties $\{\mathcal{P}_n\}$, where t varies depending on the protocol Π' of an adversary. The secret sharing approach is secure when the number of computation parties $t < n/2$ in the case of a passive adversary, while it is secure when $t < n/3$ for an active adversary.

A passive adversary, which is also called *Semi-honest*, is the one who corrupts computation parties $\{\mathcal{P}_i\}$ but executes the steps of the protocols honestly, which the aim is to learn more information about the messages of other computation parties. In the semi-honest model, we can say that a protocol Π is secure enough if the whole computations in $(y_1,\ldots,y_n) = \mathcal{F}(x_1,\ldots,x_n)$ that is computed by a party $\mathcal{P}_i$ in the protocol Π can be only obtained from its input x_i and output y_i.

A malicious adversary is an adversary who may convince corrupted parties $\{\mathcal{P}_i\}$ to deviate arbitrarily from the prescribed protocol Π' to violate the security of the protocol Π. The malicious adversary has features and facilities of the semi-honest adversary in learning some information about the messages of the computation parties $\{\mathcal{P}_i\}$ and that he can make some actions that will threaten the security in sachem. In other words, the malicious has more powerful technical aspects than the semi-honest ones, changing the messages and manipulating and arbitrarily creating messages for the computation parties $\{\mathcal{P}_i\}$. The general-purpose model for the analysis of cryptographic protocols with strong security properties is called the framework of universal composability of the SMC [47, 48].

2.1.1. Secret-sharing based SMC protocols

In this work, we use this approach to execute the program of the function $\mathcal{F}$ in a privacy-preserving manner. Secret-shared is a cryptography technique for taking

a secret s and splitting it into multiple shares, s_1, s_2, ..., s_n, and distributing the shares among the parties $\mathcal{P}_i$. For reconstructing the secret s, parties bring their respective shares together. The secure multiparty computation uses the concept of secret sharing, which was independently produced by Shamir [152] and Blakley [24]. In detail, the handler of a secret called *dealer*, besides creating the n shares of a secret, as well dealer creates the threshold t, which is the number of secret shares $\{s_i, \ldots, s_j\}$ that require to reconstruct the secret $t \leqslant n$.

The secret-shared data can be processed via secure multiparty computation protocols. These protocols take the t secret-shared as input and output a n secret-shared result during the computation. If the privacy-preserving manner has three computation parities $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$, that means 3-out-of-3 secret sharing. It supports several different SMC schemes called *protection domain*.

2.1.2. Universal composability

The framework of Universal Composability (UB) considers a set of interacting Turing machines [92] running in parallel and sending data among them, including the adversarial Turing machines. The framework states that when a set of Turing machines *securely implements* the other set—this happens when any behavior observed by an environment *Env* machine while interacting with the first set can be matched by the second set. The value of the framework is in the composition theorem. A protocol Π may securely implement an ideal functionality $\mathcal{G}$ in the $\mathcal{F}$-hybrid model, i.e., the Turing machines constituting Π may access some ideal functionality $\mathcal{F}$. Suppose Ξ is another protocol that securely implements the functionality $\mathcal{F}$. In that case, the composition of Π and Ξ (i.e., for each protocol party, we compose this party's Turing machine in Π with this party's Turing machine in Ξ) securely implements $\mathcal{G}$.

In the ideal world, the computation parties $\{\mathcal{P}_i\}$ privately send their private inputs $\{x_i\}$ to a trusted party $\mathcal{T}_p$ to compute a function $\mathcal{F}$, refereed to as functionality $\mathcal{F}_{\mathcal{ABB}}$. Each party has its own private input $x_i \in \{x_1, x_2, \ldots, x_n\}$, which is sent to trusted party $\mathcal{T}_p$ who securely compute the function $\mathcal{F}(x_1, x_2, \ldots, x_n)$. Then, result $(y_1, y_2, \ldots, y_n)$ will be returned to the parties $\{\mathcal{P}_i\}$. The function $\mathcal{F}$ is known to every party $(y_1, \ldots, y_n) = \mathcal{F}(x_1, \ldots, x_n)$. Each party knows his own x_i, learns his own output y_i, and does not learn other parties x_j and y_j. When an adversary $\mathcal{A}$ is trying to attack the ideal world, the adversary $\mathcal{A}$ has control only in any computation parties $\mathcal{P}_i$, but not against the trusted party $\mathcal{T}_p$.

In the real world, the trusted party does not exist—ideal world with its trusted party $\mathcal{T}_p$ is used only for benchmarking to evaluate an actual protocol's security. Rather than the trusted part $\mathcal{T}_p$, the real world uses a protocol Π for parties communicating with each other while performing the secure computation in function $\mathcal{F}$. The computation parties run a protocol Π and send their private inputs $\{x_1, x_2, \ldots, x_n\}$ to the function $\mathcal{F}$, then getting their private output $\{y_1, y_2, \ldots, y_n\}$. An adversary $\mathcal{A}$ can corrupt the computation parties $\mathcal{P}_i$ may either deviate arbi-

trarily in their behavior or follow the protocol Π. The protocol Π is secure if an adversary $\mathcal{A}$ can achieve any effect in the real world that can also be achieved in the ideal world by a corresponding adversary $\mathcal{A}'$.

2.1.3. Arithmetic Black Box

The Arithmetic Black Box (ABB) is an ideal functionality $\mathcal{F}_{ABB}$ that performs calculations with private data. It allows parties to store the private data handed over to it, performs operations based on users' instructions, and sends certain values back to users if a sufficient number of them request it. Let us suppose a party sends a command **store**(v) to the ideal functionality to do some calculation, where v is some value. The functionality $\mathcal{F}_{ABB}$ receives and stores the value v, then assigns a fresh *handle* h to that value by storing the pair (h, v). Finally, the ideal functionality sends h to all parties. To perform the computations without revealing any knowledge about the intermediate result, the ABB waits for a command $(\mathsf{perform}, op, h_1, \ldots, h_k)$ from all (or sufficiently many) computing parties. It looks up the values $v_1, \ldots, v_k$ stored with the handles $h_1, \ldots, h_k$, applies the operation op on them, obtaining a value v, stores it under a new handle h, and returns h to all parties. To learn a value stored under the handle h, all (or sufficiently many) parties send the command $(\mathsf{declassify}, h)$ to the ABB, which then looks up the pair (h, v) and responds with v [112].

The ideal functionality $\mathcal{F}_{ABB}$ forward to the adversary the commands they receive from the parties $\mathcal{P}_1$, ..., $\mathcal{P}_n$, except for the sensitive input values that are part of a command. Also, the adversary receives the fresh handles that the ABB creates and the results of declassification.

2.1.4. Secret-sharing based parallel SMC

In the secure multiparty computation, secret shares are the parts of a secret s, which are elements of values. Usually, the secrets' amount n is based on the number of private values which sequentially arrived from computation on top of SMC. In contrast, in the secure multiparty parallel computation, the computation parties simultaneously perform operations on the multiple data using a single instruction, which means the SIMD parallel computation on top of SMC produces vectors transmitted by secret shares to the computation parties $\{\mathcal{P}_i\}$. This technique will reduce the number of round complexity; hence it will increase the communication for each round. Sequentially, the transmitted bits among computation parties $\{\mathcal{P}_i\}$ is a single value that will be sent one by one. In parallel, the transmitted bits among computation parties $\{\mathcal{P}_i\}$ are located in a vector that will be sent once.

The operating system will divide the vector into groups if the transmitted vector's size is bigger than the bandwidth. Later, data groups will be sent in separate round complexity—an extra round complexity but still less than in sequential computation of SMC protocols. The parallel computation reduces the amount of the secret share since the interaction required among computation parties will

be reduced. We can summarize that the parallel computation reduces the round complexity of SMC protocols, reducing the number of secret shares among the computation parties. This will positively affect the speed-up of the computation; at the same time, it will not affect privacy.

2.2. Combinatorial graph problems

2.2.1. Graph

The graph is a mathematical structure consisting of points called vertices $|V|$ connected subset of them (possibly empty) by lines called edges $|E|$. The edges between vertices may have values that describe the distance of the edges called weights $|W|$. The graph can be directed, which means that its edges have a particular direction between vertices and can be undirected (both sides). Let $G = (V,E)$ be a directed weighted graph with the set of vertices $V = \{0,1,2,\ldots,n-1\}$, and the set of the directed weighted edges $E \subseteq V \times V$. Each edge $e \in E$ has a *weight* $w(e) \in \mathbb{R}$.

A graph $G = (V,E)$ can be represented in computer memory differently. The *adjacency matrix* of G is a $|V| \times |V|$ matrix over $\mathbb{Z} \cup \{\infty\}$, where the entry at u-th row and v-th column is $w(u,v)$. Such representation has $|V|^2$ entries, and we call it the *dense* representation. 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 $w_i = w(u,v_i)$. Such representation has $O(|E|)$ entries, and we call it the *sparse* representation. If edges $|E|$ are significantly smaller than $|V|^2$, then sparse representation takes up less space than dense representation, and the algorithms working on sparse representation may be faster [167].

A graph (actually, an infinite family of graphs) is *sparse* if its number of edges is "proportional" to its number of vertices, $|E| = O(|V|)$. A graph is *dense* if $|E| = \omega(|V|)$. A graph is *planar* if it can be drawn as a plane without crossing the edges outside vertices. If G is planar, then $|E| \leq 3|V| - 6$ according to Euler's formula relating the numbers of a planar graph's edges, vertices, and faces of its drawing [41].

2.2.2. Shortest path problem

A path $\delta(u,v)$ from source vertex u to target vertex v in the graph G is a finite sequence of vertices $u = v_0, v_1, ..., v_n = v$, where $(v_{i-1}, v_i) \in E$ for all $i \in \{1,\ldots,n\}$. The shortest path between two vertices $\delta(u,v)$ in a graph G is a path has the minimum sum of the weights $\sum_{i=1}^{n} w(v_{i-1}, v_i)$ of the whole vertices among the source vertex u to target vertex v, shortest distance from u to v is [59]:

$$w(u,v) = \begin{cases} min\{w(p) : u \longrightarrow v\}, & \text{if a path } u \longrightarrow v \\ \infty, & \text{otherwise.} \end{cases}$$

The shortest path problem is finding a path between two vertices in a graph $G = (V, E)$ such that the sum of edges weight $w(e) \in \mathbb{R}$ from vertex u to vertex v is the minimum. It is one of the most critical problems in combinatorial and algebraic graph theories. The tradeoffs between computation and communication cost significantly make some algorithms optimal in different circumstances.

Finding the single-source shortest distances in a small or medium-sized graph can be done by implementing one of the classical sequential SSSD algorithms such as Dijkstra, Bellman-Ford, or Thorup [159]. Dijkstra works with non-negative weights for the edges, while Bellman-Ford works over some negative weights. The Bellman-Ford algorithm works for general graphs with time complexity $O(nm)$ where n is the number of vertices, and m is the number of the weighted edges in the graph G. The time complexity of the Dijkstra algorithm is $O(n^2)$ in the adjacency matrix representation of G, and $O(m \log n)$ in the adjacency list representation (using a binary heap).

Some sequential algorithms' poor worst-case bounds may perform well for such graphs, while processing large graphs might need parallel algorithms. The parallel single-source shortest path algorithms for solving this problem are Δ-Stepping and Radius-Stepping algorithms. The Δ-stepping algorithm works by correcting the tentative distances of the vertex several times during edge relaxations until all tentative distances are settled; it is called a label-correcting algorithm. The Radius-stepping algorithm corrects the shortcoming of the Δ-stepping algorithm: it has a provable upper bound on the number of steps. The average time complexity of the Δ-stepping algorithm is $O(n \log n + m)$, while it is $O(m \log n)$ for the Radius-stepping algorithm.

The shortest paths also can be calculated by applying non-classical shortest path algorithms such as a search algorithm, i.e., Breadth-first search, and different techniques like Algebraic path computation. The algebraic path problem has a different approach based on a particular algebraic structure called a *closed semiring*. Moreover, the general setting of the problem is for solving various graph problems, not only algebraic paths [70].

The second version of the shortest path problem is the all-pairs shortest distance which involves computing the shortest path among all vertices in a graph G. A naive approach to the computation of all-pairs shortest distance would have a computational complexity that is n times the computational complexity of SSSD, where n is the number of the graph's vertices. Different algorithms for all-pairs shortest distance, such as Johnson, Floyd-Warshall, and transitive closure. Johnson's algorithm with $O(nm + n^2 \log n)$ time complexity is more efficient than the Floyd-Warshall algorithm, the time complexity of the Floyd-Warshall is $O(n^3)$. Johnson uses Bellman-Ford and Dijkstra algorithms; optimizing Bellman-Ford, or Dijkstra will positively affect Johnson's algorithm.

2.2.3. Minimum spanning tree and forest

A Spanning Tree (ST) of a connected graph G is a subgraph that is a tree that includes all of the vertices $|V|$ of G, and every edge $|E|$ in the tree T belongs to the graph G. The minimum spanning tree for the graph G is a spanning tree whose edge weights are as minimum as possible for weighted graphs. The oldest algorithm for finding a minimum spanning tree or forest for a not connected graph is Borůvka's Algorithm [42]. The most well-known MST algorithms are Prim's minimum spanning tree algorithm for a graph and Kruskal algorithm [110], given graph must be connected, undirected, and weighted. The general idea in MST is to create two sets of vertices. The first set includes the vertices already in MST, while the second contains those not included yet. Prim's algorithm creates the MST from any vertex in the graph G. In contrast, Kruskal builds the MST from the vertex with the smallest weight in the graph [167].

Both Prim and Kruskal's algorithms are a greedy approach to the MST problem, with few significant differences. Prim's algorithm is the best option to run over dense graphs, while Kruskal is more efficient over the sparse graph. Moreover, Prim's algorithm traverses one vertex more than once in minimum getting, while the traversing in Kruskal for one vertex is once. The time complexity of the prim's algorithms is $O(n^2)$, and in the Fibonacci heap's structure, it can be improved up to $O(m \log n)$. In contrast, the time complexity in Kruskal's algorithm is $O(m \log n)$ [67].

The computation of the minimum spanning forest is considered the computational squaring of the minimum spanning tree that would increase the time complexity n times. Indeed, the time complexity of the minimum spanning forest is n times prim or Kruskal's algorithms. The high time complexity of the minimum spanning forest can be reduced by parallelization of the unconnected components of the graph. A parallel algorithm for finding the minimum spanning forest for an undirected graph and minimum spanning tree for the connected components are presented in [13]. The Boruvka algorithm for both finding MST and MSF might be easily parallelized. The naive parallel approach for finding MSF, regardless of the algorithm, simultaneously runs each component of the unconnected graph on a processor, collecting the results from each component to establish the forest.

2.3. Algebraic graph theory

The approach of which algebraic methods are applied to various graph problems, such as minimum spanning tree and shortest path called *Algebraic graph theory*. It has three main branches: linear algebra, group theory, and graph invariants [20]. The algebraic properties of matrices are related to the combinatorial properties of the graph and their applications. Various graph problems can be solved combinatorially. Thus, it can also be solved by algebra.

2.3.1. Algebraic path computation

Let $G = (V,E)$ be a weighted graph with a set of the vertices $V = \{1,2,\ldots,n\}$, and a set of the weighted edges $E \subseteq V \times V$ and a weight function $W : E \to S$, where S is a *semiring*. A semiring (or dioid) is an algebraic structure with two binary operations, $\oplus$ and $\otimes$. A path in G among any two non-neighbor vertices is a sequence of vertices $P = \{v_1, v_2, \ldots, v_n\}$ and the weight of a path which is defined in the semiring as $W(p) = w(v_1,v_2) \otimes w(v_2,v_3) \otimes, \ldots, \otimes w(v_{n-1},v_n)$. Let an $n \times n$ matrix $A = [a_{ij}]$ with graph $G = G(A)$, where $a_{ij} = \infty$ if there is no edge between the vertices, i be associated j in graph G. The two versions of the shortest distance problems over the algebraic structure are given as follows [86]:

- For single-source shortest distance, finding the vector $\bar{X} = [x(i)]$ of distances $x(i)$ from vertex 1 to all vertices i in the graph G. Finding the shortest path is iteratively given by **sum** $+$, and **min** operations as follows :

  $x(1) = \min(\min(x(j) + a_{j1}), 0)$,
  $x(i) = \min(\min(x(j) + a_{ji}), \infty)$, where $i = 2, \ldots, n$

  The operations, **min** will be substituted by $\oplus$ and **sum** $+$ will substituted by $\otimes$. The distances in the graph using semiring structure satisfy the following:

  $x(1) = \oplus\ (\ \oplus\ (\ x(j) \otimes a_{j1}\ ), 0\ )$,
  $x(i) = \oplus\ (\ \oplus\ (\ x(j) \otimes a_{ji}\ ), \infty\ )$, where $i = 2, \ldots, n$, and denoting $\bar{I}^{(1)} = [0, \infty, \ldots, \infty]$,

$$\bar{X} = \bar{X} \otimes A \oplus \bar{I}^{(1)} \tag{2.1}$$

- For all-pairs shortest distance, finding the matrix $\mathbf{X} = [x(i,j)]$ of distances between all pairs of the vertices in the graph $\mathbf{G}$, and denoting $\mathbf{I} = [\delta_{ii}]$, $\delta_{ii} = 0$, $\delta_{ij} = \infty$ if $i \neq j$.

$$X = X \otimes A \oplus I \tag{2.2}$$

The systems (2.1) for vector and (2.2) for matrix can be used to solve various path problems classes, existence, enumeration, counting, and optimization. i.e., paths of maximum capacity, paths with a minimum number of arcs, paths of maximum reliability, reliability of a network, longest paths, and shortest paths.

2.3.2. Algebraic framework for minimum spanning tree

Besides solving the shortest path problem using the semiring structure, it can also be used to solve other graph problems, such as the minimum spanning tree. Another structure can be used to solve MST problem called c-semiring—a constraint-

based semiring. We can define c-semiring as a semiring whose addition operation is disabled and described over arbitrary sets [22, 21]. This general algebraic framework for solving minimum spanning trees is similar to the one proposed for shortest path problems in [130].

2.4. Overview of essential algorithms

This section presents the essential algorithms of the single-source shortest distance, all-pairs shortest distance, and minimum spanning tree. Our proposed protocols are constructed by following the general structure of that classical SSSD, APSD, MST, and MSF algorithms.

2.4.1. Overview algorithms for single-source shortest distance

Dijkstra Algorithm

The Dijkstra algorithm is a greedy approach to solve the single-source shortest path problem in various ways, such as the Dijkstra algorithm with a list [117, 142], with binary heap [96] and with Fibonacci heap [76]. The solution to the SSSD problem in the Dijkstra algorithm is for both directed and undirected graphs; all edges E must have *non-negative* weights, and the graph must be connected.

Algorithm 1: Dijkstra

```
   Data: Adjacency matrix G[u,v], number of vertices n, source vertex s
   Result: Distances d⃗
 1 Function minDistance(d⃗, M⃗) is
 2     m = ∞
 3     for v = 0 to n − 1 do
 4         if M[v] == false && d[v] ≤ m then
 5             m = d[v], u = v
 6     return u
 7 begin
 8     for i = 0 to n − 1 do
 9         d[i] = ∞, M[i] = false
10     d[s] = 0
11     for j = 0 to n − 1 do
12         u = minDistance(d⃗, M⃗), M[u] = true
13         for v = 0 to n − 1 do
14             if [u,v] > 0 && M[v] == false && d[u] + g[u,v] < d[v] then
15                 d[v] = d[u] + g[u,v]
16     return d⃗
```

The bounds of the time complexity of Dijkstra's algorithm on a graph G with edges E and vertices V depend on the data structures used by the algorithm. The time complexity of Dijkstra's algorithm with a list is $O(n^2)$, where n is the number of vertices and m is the number of edges. The time complexity with the binary heap is $O((m+n)\log n)$. Algorithm 1 is the Dijkstra algorithm with an adjacency matrix; the time complexity is $O(n^2)$. It also can perform better for finding SSSD on a dense graph. Once the algorithm starts to iterate from source vertex s to target vertices $v \in V$, the algorithm finds the minimum distance from vertex u to vertex v until the last vertex in the graph G, and it is a blind search that will consume much time.

Bellman-Ford Algorithm

The Bellman-Ford algorithm finds the shortest paths from a single source vertex s to all other vertices $\{v_0, v_1, ..., v_{n-1}\} \in V$ in a weighted undirected graph G. Bellman-Ford is more straightforward than the Dijkstra algorithm and suits well for distributed systems, so it is highly recommended to be used to construct privacy-preserving protocols. The shortcoming in the Bellman-Ford Algorithm is that the time complexity is $O(nm)$, where n is a number of vertices, and m is a number of edges, which is more complex than Dijkstra.

Algorithm 2: Bellman-Ford Algorithm

```
   Data: G = (V, E)
   Data: Source vertex s
   Result: Distances d⃗
1  begin
2      foreach v ∈ |V| do
3          d[v] = ∞
4          d[s] = 0
5      foreach v ∈ |V| do
6          foreach e ∈ |E| do
7              if (d[u] + w(e) < d[v]) then
8                  d[v] = d[u] + w(e)
9      foreach e ∈ |E| do
10         if (d[u] + w(e) < d[v]) then
11             error : Graph contains a negative − weight cycle
12     return d⃗
```

In the case of the negative cycle, there is no shortest path because any path can be made cheaper by one more walk around the negative cycle. Several graphs have a negative cycle, so the Dijkstra algorithm is not the best option for all kinds

of such graphs. Bellman-Ford Algorithm can detect and report where there is a negative cycle in the graph; see Algorithm 2.

The main feature of the Bellman-Ford is relaxation, which is the approximations to the shortest distance are replaced by shorter ones until they eventually reach the correct distance (shortest path). Bellman-Ford algorithm relaxes all the edges E compared to the Dijkstra algorithm, which uses the priority queue. This Bellman-Ford feature allows an application to have a larger range of inputs than the Dijkstra algorithm.

Radius-Stepping Algorithm

Radius-stepping algorithm is a trade-off algorithm between the work and depth bounds for the single-source shortest paths. The algorithm finds the single-source shortest paths for the weighted undirected graph G, where the vertices V and edges E are represented in an adjacency matrix as input to the algorithm. As well as the source vertex s, a target radius value for every vertex is given as a function r: $V \rightarrow \mathbb{R}+$.

The Radius-Stepping algorithm is a hybrid algorithm of the Dijkstra and Bellman-Ford algorithms. The steps of the Bellman-Ford algorithm are internal operations in the Dijkstra algorithm. The Dijkstra part visit vertices in increasing distance from the source s and settling each vertex $v \in V$ in the undirected graph G (Determining the correct distance $d(s,v)$).

Algorithm 3: Radius-Stepping

Data: $G = (V,E)$, vertex radii $r(\cdot)$, source vertex s
Result: The graph distance $\vec{\delta}$ from s

$\delta(\cdot) \leftarrow +\infty$, $\delta(s) \leftarrow 0$
foreach $v \in N(s)$ **do**
    $\delta(v) \leftarrow w(s,v)$
    $S_0 \leftarrow \{s\}$
    $i \leftarrow 1$
while $|S_{i-1}| < |V|$ **do**
    $d_i \leftarrow \mathsf{min}_{v \in V \setminus S_{i-1}}\{\delta(v) + r(v)\}$
    repeat
        foreach $u \in V \setminus S_{i-1}$ *s.t* $\delta(u) \leq d_i$ **do**
            foreach $v \in N(u) \setminus S_{i-1}$ **do**
                $\delta(v) \leftarrow \mathsf{min}\{\delta(v), \delta(u) + w(u,v)\}$
    until *no* $\delta(v) \leq d_i$ *was updated*
    $S_i = \{v \mid \delta(v) \leq d_i\}$
    $i = i + 1$
return $\vec{\delta}$

Instead of visiting one vertex v at a time, Radius-Stepping visits the vertices in steps (inside the while-loop, from repeat to the end in Algorithm 3). The algorithm increases the radius centered at vertex s from d_{i-1} to d_i, and all vertices $v \in V$ in the annulus $d_{i-1} < d(s,v) < d_i$ will be settled. The Bellman-Ford parts (Repeat-Until Loop) settle the vertices in sub-steps. In the Δ-stepping algorithm, the radius will be increased by a fixed amount in each step in the round distance $d_i = d_{i-1} + \Delta$. In the worst case, this increase can require $\theta(n)$ sub-steps. As well as in some cases, it required $\theta(nm)$ work because each sub-step will perform the computation on the same set of vertices and their edges. The Radius-Stepping algorithm solves this problem efficiently by proposing a new round distance d_i in each round.

The objective is bounding the number of sub-steps to decrease the time complexity of processing the same set of vertices and their edges in each sub-step. In general, the Radius-Stepping algorithm has efficiently optimized the shortcoming in the Δ-Stepping algorithm, proving a bound on the number of steps. The contribution of radius-stepping algorithms is optimizing the algorithm by using SIMD to run efficiently, reducing time complexity.

Breadth-First Search Algorithm

The serial version of the Breadth-First Search algorithm is presented in Algorithm 4. The main feature of this algorithm is traversing over all vertices in the graph using two stacks, Frontier Stack (FS) and Next Frontier Stack (NS). The stacks are to store the visited vertices; the stack FS is to store the visited vertices, while the stack NS is to store the next frontier of the visited vertices.

Algorithm 4: Serial breadth-first search

```
Data: Adjacency matrix G ∈ ℤ^{n×n}, source vertex s
Result: The graph distance δ⃗ from s
1  begin
2      forall v ∈ V do δ(v) ← +∞
3      δ[s] ← 0, level ← 0 FS ← {}, NS ← {}
4      push(s, FS)
5      while FS ≠ {} do
6          foreach u ∈ FS do
7              foreach neighbour v of u do
8                  if δ[v] ← ∞ then
9                      push(v, NS) δ[v] ← level
10         FS := NS
11         NS := {}
12         level := level + 1
13     return δ⃗
```

The frontier has some distances for vertices from the source vertex s; distances are called level. The neighbors of this vertex have to be explored, some of which are not explored yet. The BFS algorithm discovers these vertices and stores them in the next frontier, and so on, until all vertices in the graph are already stored in the stack FS. In the outer **while**-loop, the number of the iterations (Iter) is bounded by the distances of the longest shortest distance from source vertex s to any vertex in the whole graph G. The two **for**-loops can be executed in parallel, either in shared or distributed memory, using some parallel framework.

This work uses the same algorithmic structure of the breadth-first search algorithm to perform the parallel calculation of the BFS using Single-instruction-multiple-data. The elements of the two sets of vertices $u, v \in V$, will be vectorized in two private vectors $[\![\vec{v}]\!]$ and $[\![\vec{u}]\!]$. Then, using the SIMD instructions, the algorithm can traverse over all vertices in the graph simultaneously, then update the distances, eventually finding the shortest distances in the given graph G.

2.4.2. Overview algorithms for the all-pairs shortest distance

Johnson's algorithm

Johnson's algorithm is a weighted directed graph G algorithm that finds the shortest paths between all pairs of vertices V, as illustrated in Algorithm 5.

Algorithm 5: Johnson Algorithm

Data: Source vertex s, Adjacency matrix $G \in \mathbb{Z}^{n \times n}$
Result: All-pairs shortest distances [D]

begin
  Create $[G']$, where
    $G'.\mathsf{V} = G.\mathsf{V} \cup \{s\}$, $G'.\mathsf{E} = G.\mathsf{E} \cup \{(s,v) : v \in G.\mathsf{V}\}$, and
    $w(s,v) = 0$ for all $v \in G.\mathsf{V}$
  if Bellman-Ford(G', w, s) = false **then**
    return *Negative cycle*
  else
    foreach *vertex* $v \in G'.\mathsf{V}$ **do**
      $h(v) = \delta[s,v]$ `// computed by Bellman-Ford`
    foreach *edge* $e(u,v) \in \mathbf{G}'.\mathsf{E}$ **do**
      $w'(u,v) = w(u,v) + h(u) - h(v)$
  foreach *vertex* $u \in G.\mathsf{V}$ **do**
    $\mathsf{D}'[u,v]$ = Dijkstra(G, w', u)
    foreach *vertex* $v \in G.\mathsf{V}$ **do**
      $\mathsf{D}[u,v] = \mathsf{D}'[u,v] + h(v) - h(u)$
  return [D]

This algorithm does not allow for a negative weight cycle but for some negative-weight edges. Dijkstra and Bellman-Ford algorithms are subroutines in the Johnson algorithm. The Bellman-Ford algorithm is used to re-weigh all negative edges and make them all positive, and the Dijkstra algorithm is then used to compute the shortest path among any two vertices in the graph G. The Bellman-Ford subroutine calculates the minimum weight $h(v)$ for each vertex v starting with a new vertex s. The algorithm is ended if the iteration finds a negative cycle.

The values computed by the Bellman-Ford subroutine will be used to re-weigh the edges in the graph G. The edge with weight $w(u,v)$ from vertex u to vertex v will have a new length of $w(u,v)+h(u)-h(v)$. The Dijkstra subroutine is then used to find the shortest path to the remaining vertices V in the graph G. Although a Fibonacci heap is used to implement Dijkstra's algorithm, it is the most efficient structure for the algorithm. As a practical matter, it is unsuitable for SIMD implementation. Rather than Fibonacci heap or queue, we employ the adjacency matrix in Dijkstra's algorithm. The Johnson algorithm is best for sparse graph G, while Floyd-Warshall is best for large dense graph G.

2.4.3. Overview algorithms for minimum spanning tree

Prim's algorithms

Prim's algorithm is a greedy algorithm that uses the hops to build a solution step by step. The procedure starts with an empty spanning tree that should be completed during processing, resulting in a minimum of the edges' weights. The goal is to keep two sets of vertices separate. The first set contains vertices that have already been included in the MST, whereas the second set includes vertices that have not yet been formed. It evaluates all the edges that connect the two sets at each step and selects the minimum weighted edge from among them. After choosing the edge, the algorithm moves the edge's opposite endpoint to the MST set as a return value. Note that spanning-tree means that all vertices in a given graph **G** must be connected into one component. The vertices must be connected with the edges with minimum weights, called the minimum spanning tree. If multiple components exist (a group of trees or a disconnected graph), it is a spanning forest. Furthermore, if the minimum weight edges for all vertices are in the various components, it is a minimum spanning forest. Prim's minimum spanning tree algorithm is presented in Algorithm 6. It is also called Jarnik's algorithm [151] and Prim-Dijkstra algorithm [54]. Different versions of Prim's algorithm exist, such as the adjacency matrix and priority queue. The priority queue must include the vertex and the weight of the edge that led us to it. It must also order the vertex inside it according to the passed weight. Due to their complex control flow, priority queues are challenging to implement efficiently in parallel privacy-preserving implementations using SIMD.

The advantage of Prim's algorithm is that it has lower complexity than similar algorithms, making it superior to Kruskal's method. Prim's algorithm helps to

deal with dense graphs with many edges. However, if numerous edges with the same weight occur, Prim's algorithm needs to provide us more control over the selected edges—this shortcoming may be solved in our proposed version with the SIMD approach.

Minimum spanning forest may be determined by applying some algorithms such as Borůvka's and Kruskal's for an unconnected graph. Prim's algorithm, in its most basic version of the algorithm, only finds an MST in connected graphs. However, the MSF may also be found by executing Prim's algorithm separately for each connected graph component [105]—this is the feature of our proposed privacy-preserving MSF protocols.

Algorithm 6: Prim

Data: Adjacency matrix $\mathbf{G}$, number of vertices n, source vertex s
Result: Minimum spanning tree $\vec{P}$

Function $\mathsf{minKey}(\vec{K},\vec{M})$ **is**
  $m = \infty$
  for $v = 0$ **to** $n-1$ **do**
    if $M[v] == \mathsf{false}$ & $K[v] \leq m$ **then**
      $m = K[v]$
      $idx = v$
  return idx
begin
  for $i = 0$ **to** $n-1$ **do**
    $K[i] = \infty$
    $M[i] = \mathsf{false}$
  $K[s] = 0$
  $P[s] = -1$
  for $j = 0$ **to** $n-1$ **do**
    $u = \mathsf{minKey}(\vec{K},\vec{M})$
    $M[u] = \mathsf{true}$
    for $v = 0$ **to** $n-1$ **do**
      if $\mathbf{G}[u,v]$ & $M[v] == \mathsf{false}$ & $\mathbf{G}[u,v] < K[v]$ **then**
        $P[v] = u$
        $K[v] = \mathbf{G}[u,v]$
  return $\vec{P}$

2.5. Related work

In this section, we highlight the previous studies related to the work in this thesis. Some of these are not directly related to privacy preservation for some combina-

torial and algebraic graph algorithms. However, we are interested in studying the parallelization methods for such problems, even in non-privacy-preserving. These studies give us some features about the proper techniques for parallelization that can help us generate some parallel algorithms sine on top of SMC protocols.

2.5.1. Parallel shortest distances

We study the implementations of shortest path algorithms on parallel architectures besides existing works on privacy-preserving shortest path computation; such implementations tackle similar issues in reducing the dependencies among algorithm's parts.

A new implementation of the Dijkstra Algorithm on a directed graph using a STAR-machine is proposed by Nepomniaschaya et al. [133]. The algorithm simultaneously finds the shortest path and the distance between the source vertex and all other vertices in a graph. Another implementation of Dijkstra's algorithm with a high degree of parallelism while reducing memory usage is proposed [157]. An Field-programmable Gate Array (FPGA)-based accelerator with SIMD architecture is used; such proposed architecture is suitable for sparse graphs only. It does not fit with a dense graph.

Parallel SSSD algorithms have lower time complexity than any sequential algorithms [109, 128, 161]. However, the creation and joining of threads in these algorithms depend again on private data. The emulation of pools of private threads, possibly similar to garbled RAM [80], will likely introduce its overheads, which overwhelm any gains in efficiency obtained from the more complex algorithm.

There exists some proposed interesting work for the APSD problem. A vectorized version of the Floyd-Warshall Algorithm for finding the shortest path to improve performance is proposed by Han et al. [87]. There is no optimal exploitation for the SIMD framework. The SIMD is supposed to give more speed-up than what is implemented in their work. The result shows that the speed-up reaches between 2.3 to 5.2 times. Matsumoto et al. [126] used a hybrid Central Processing Unit (CPU)- Graphics Processing Unit (GPU) system to run a blocked united algorithm for the APSD problem. This algorithm simultaneously computes the shortest-path distance matrix and the shortest-path construction matrix. To reduce data communication between CPU and GPU, Floyd-Warshall Algorithm is used. The method is efficient but unsuitable for the Sharemind SMC platform that we employ in this work.

In [132], two different CPU platforms (2x Intel Xeon and Intel Core) are used with various components (GPUs) to implement Dijkstra algorithms and Floyd-Warshall. Three different versions are implemented in the CPU platform for the Dijkstra algorithm—serial, parallel, and Parallel with Queue. The fastest one is the Parallel with Queue version for both variants of Intel Hardware. In the case of the Floyd-Warshall Algorithm, the result shows that GPU CUDA is the fastest one in both hardware with different sizes of vertices in the graph. The SIMD version

of the algorithm is faster than a standard case, but the CUDA version is still faster.

2.5.2. Shortest distances for planar graph

We have been looking for existing SSSD algorithms for planar graphs, of which there exist a few, to be adapted to privacy-preserving computations. The linear-time algorithm for single-source shortest paths in planar graphs with a non-negative edge is proposed by Henzinger et al. [90] with time complexity $O(n)$. Lipton et al. [122] proposed an algorithm for finding the shortest path for a planar graph with arbitrary real-valued edge weights; the time complexity of this algorithm is $O(n^{3/2})$. The shortest path algorithm for n-vertex planar graph with real-valued weights [68] is proposed with time complexity $O(n\log^3 n)$. A linear space recursive algorithm exists with time complexity $O(n\log^2 n)$ [108]. Later, the liner space recursive algorithm proposed by Klein et al. was improved by Mozes et al. [131], they presented a method to compute single-source shortest path distances in the graph in $O(n\log^2 n/\log\log n)$ time.
The Δ-stepping algorithm [129] is a single-source shortest path algorithm for arbitrary directed graphs with non-negative edges. The algorithm can be implemented efficiently in large graphs' sequential and parallel counterparts. The average time complexity of the Δ-Stepping algorithm is $O(n\log n+m)$. The shortcoming in the Δ-Stepping algorithm is that the algorithm has no known theoretical bounds on general graphs; bounds can increase n-times in each iteration. The algorithm performs a sequence of steps, each increasing the *radius*. Each step treats the vertices in the current annulus, but the time complexity of each step may include that of n substeps. The Radius-Stepping algorithm is proposed in [26] to solve the bounds' problem in the Δ-Stepping algorithm. It has the best tradeoff between work and depth bounds using a variable instead of fixed-size radius increases. There are proven bounds on the number of steps. The time complexity of the Radius-Stepping algorithm is $O(m\log n)$.

2.5.3. Privacy-Preserving shortest distances

The study of privacy-preserving shortest distance algorithms started with Brickell and Shmatikov [45], who proposed protocols for privacy-preserving computation of APSD and SSSD. Their protocols are built on top of protocols for privacy-preserving set unions, which they also proposed. Their SSSD algorithm requires $O(|V|^2\log|V|)$ oblivious transfers, where V is the set of vertices of the graph. They presented their algorithm sequentially; it can be parallelized to a certain extent.

We have already mentioned the use of the techniques of Oblivious RAM [83] in implementing the Bellman-Ford algorithm on top of an SMC protocol set [104]. Oblivious RAM has also been used for privacy-preserving implementation of Dijkstra's algorithm [124], achieving good communication usage but having $O(|E|)$ round complexity, where E is the set of edges of the graph. Aly and Cleemput [3]

give another implementation of Dijkstra's algorithm, this time for dense graphs and with $O(|V|)$ round complexity. This implementation has been improved in [4], with no more use for Oblivious RAM in this protocol.

An efficient protocol for privacy-preserving shortest paths computation for navigation is proposed in [169]. They formulated the problem of compressing the next-hop matrices for road networks. This compressing method they developed enabled an efficient cryptographic protocol for fully private shortest-path computation in real-time navigation on city streets. The work uses sparse graphs as the input data modeling road network (where a node has four outgoing edges). The work does not follow the general paradigm of implementing graph algorithms on top of general-purpose SMC protocol sets. It is specific for this application, as real-time navigation apps deal with a limited range of graph sparsity. It does not claim to offer general-purpose privacy-preserving shortest path algorithms.

Blanton et al. [25] presented several data-oblivious algorithms, breadth-first search, single-source shortest path, minimum spanning tree, and maximum flow. Theoretically, they introduced the data-oblivious algorithms without actual implementation shown.

Ramezanian et al. [144] proposed the Extended Floyd Warshall Algorithm and used it in a novel protocol that enables privacy-preserving path queries on directed graphs. Extending the Floyd-Warshall algorithm aims to generate a matrix that holds the penultimate vertices of the shortest paths between each pair of vertices. This matrix is then queried using the techniques of private information retrieval. Again, the method is heavily adapted for the application.

2.5.4. Parallel minimum spanning tree and forest

There have also been works on optimizing the calculation of finding a minimum spanning tree to reduce the time complexity of the algorithms. Chung et al. [55] proposed a parallel method for finding a minimum spanning tree based on the sequential version of Borůvka's algorithm. Their implementation used four different kinds of graphs—random, geometric, structured, and Travelling Salesman Problem (TSP) on asynchronous, distributed-memory machines. The distributed memory cannot run the method in privacy-preserving architecture.

A simple parallel algorithm for finding a minimum spanning tree for undirected weighted graph $G = (V,E)$ on Exclusive Read Exclusive Write (EREW) PRAM is proposed by [97]. The time complexity of their algorithm is $O(\log^{3/2} n)$ using $n+m$ processors, where $n = |V|$ is the number of the vertices and $m = |E|$ is the number of the edges. This algorithm's significant innovation is extracting necessary information about elements without explicitly shrinking components. The algorithm is faster by a factor of $\sqrt{\log n}$ than any deterministic algorithm. The algorithm is designed to be run over the EREW PRAM machine, which is not suitable for massive graphs that occupy much memory, significantly if the algorithm is modified to be compatible with the SIMD approach as our protocols.

Vineet et al. [162] presented a minimum spanning tree algorithm on Nvidia GPUs under CUDA. The proposed algorithm is a recursive formulation of Borůvka's algorithm for a huge undirected graph. The graphs they use in the implementation reach 5 million vertices and 30 million edges. The result shows that the speed-up of the algorithm is 50 times over the CPU and around nine times over the best GPU implementation for finding the MST. An efficient algorithm gives the result within 1 second for a huge graph. Another minimum spanning tree algorithm on Nvidia GPU under CUDA is invented [164]. This algorithm is based on Prim's Algorithm using the newly developed GPU-based Min-Reduction data-parallel primitives. The result shows that the speed-up is two times on GPU over CPU implementation and three times on non-primitive performance. Both proposed algorithms over GPU are not fit to be run in our Sharemind SMC platform.

Boldon et al. [40] proposed a minimum-weight degree-constraint spanning tree algorithm. They used a massively-parallel SIMD machine, MasPar MP-1, to implement the four heuristics for approximate solutions to the d-MST problem. The parallel implementation method is designing a suitable PRAM algorithm and then implementing it directly in the MasPar MP-1. The result shows that the graph with 5000 vertices and 12.5 million edges can be processed in less than 10 seconds.

In [156], a parallel algorithm for finding a minimum spanning tree for a weighted undirected graph is proposed; the time complexity is $O(\log m)$. The parallel algorithm in this paper is based on the modification of the sequential algorithm in [155] and Klein's algorithm [107]. The implementation using $O(m+n)$ processors is presented for the SIMD machine where m and n are the edges and vertices, respectively. The optimization in this work achieves a speed-up of $O((m \log\log n)/\log m)$.

Significantly, the computation of minimum spanning forest is considered the computational squaring of minimum spanning tree that would increase the computational cost n times. The applications that use minimum spanning forest, e.g., hyperspectral image [51], designing Supply-Chain Network (SCN)., e.g., political districting [170], and designing communication networks [171, 141]. Furthermore, the need to perform such high computational costs in privacy preservation motivated the researcher to study and develop an efficient protocol. In [69], a segmentation method based on Kruskal's MST algorithm is proposed. The algorithm output is a group of MSTs, i.e., a forest; each tree corresponds to a segment of the given hyperspectral image. Another sequential algorithm based on Prim's minimum spanning tree is with a cutting criterion for image segmentation [19, 148]. Wassenberg et al. [166] proposed a parallel method that computes multiple independent minimum spanning trees and then connects them to a forest. As well as many strategies for dealing with different properties over the minimum spanning forest-based hyperspectral image are proposed [88, 154, 140].

2.5.5. Privacy-preserving minimum spanning tree

In our work, we are interested in optimizing the calculation of the minimum spanning tree using the SIMD approach in a privacy-preserving manner. The motivation is that the privacy-preserving minimum spanning tree has not been studied yet. In [146], two privacy-preserving minimum spanning tree algorithms in the semi-honest model are proposed. One of them is a privacy-preserving MST algorithm based on Prim's algorithm, and the other is a privacy-preserving MST algorithm based on Kruskal's algorithm. Both proposed privacy-preserving MST algorithms implemented on top of Yao's garbled circuit protocols [172, 64, 125]. The graph's structure is public, but the weights of the edges are private. It would be more secure if the graph's whole structure were private, which is one of the most significant issues in our work.

The privacy-preserving minimum spanning tree algorithm based on the Awerbuch and Shiloach algorithm [13] is proposed by Laud [114]. He proposed privacy-preserving protocols to perform in parallel many reads or writes of the elements from the private vectors, according to private addresses. Implementing the privacy-preserving minimum spanning tree algorithm with private read or write protocols had not been investigated before. In our paper, we use the same protocols for private reading or writing to find the privacy-preserving minimum spanning tree based on Prim's algorithm using the SIMD approach in the Sharemind SMC platform efficiently.

2.5.6. Semiring framework for graph algorithms

The graph algorithm problems, i.e., shortest path and minimum spanning tree and forest, also can be solved by semiring framework, not only by following the classical graph algorithms [57, 85].

In [130], a general algebraic framework for single-source shortest distances based on the semiring framework is proposed. The algorithm finds the shortest distance for a weighted directed graph and the k-shortest distances in a directed graph. The proposed general algebraic framework reduces the gap between theoretical computer science and actual implementation.

Pan and reif [137] proposed a parallel algorithm for the path algebra computation in an n-vertex graph. More specifically, it proposed a general stream contraction technique for speed-up of parallel algorithms through their systolic rearrangement and showed its power by accelerating a parallel algorithm of [136]. They presented two algorithms, the first is generalizing the algorithm (for computing path algebra) based on [136], and the second is the acceleration version of the algorithm; hence both algorithms are based on a semiring framework. The proposed work is theoretically presented, and no actual implementation has been done.

In [135], presented a solution of linear system $Ax = b$ in parallel with a sparse $n \times n$ symmetric matrix A. The adjacency matrix is associated with graph $G(A)$,

with n vertices and edge m for each nonzero entry in matrix A. The algorithm uses a tree-separator approach to split the graph into s(n)-separator, then computes a special recursive factorization of A. This approach can be used with a semiring framework for sparsely computing the algebraic path of matrix A.

Stefano and Francesco [23] proposed a general algebraic framework based on the c-semiring framework for solving minimum spanning tree problems. The available algorithms can solve MST problems by following different cost criteria. The classical algorithms for MST. Prim's and Kruskal's algorithms are blocked keys for constructing the MST algorithms based on c-semiring in this work. The Quality of Service (QoS) [163] in the multicast communication networks requires minimizing the cost of tree construction for several clients simultaneously. The proposed algorithms, Prim's and Kruskal's algorithms, built based on the c-semiring framework, reduced the time complexities to logarithmic time.

Besides different problems such as shortest distance and minimum spanning tree problems are effectively solved by semiring framework, parallel minimum spanning forest algorithms using linear algebra primitives are recently proposed in [14]. The proposed MSF algorithm is based on the Awerbuch-Shiloach algorithm. A multi-linear kernel operates on an adjacency matrix, and two vectors are introduced. This proposed kernel updates graph vertices by combining information from adjacent edges and vertices.

3. PRIVACY-PRESERVING PARALLEL COMPUTATION METHODOLOGY

3.1. Introduction

This chapter presents the main features of the empirical research and methodology in constructing the graph protocols and their related algorithm on top of SMC protocols. This thesis introduces the actual implementation of the proposed protocols on top of ABB. Therefore, this chapter presents the main points in research procedures and designing the parallel algorithms on top of the SMC protocols and the implementation on the SMC Sharemind platform. This chapter also discusses the notation and abstraction of the SMC protocol and the SecreC programming used to develop secure multiparty computation protocols. SIMD parallel is also discussed, showing how the round complexity can be reduced.

3.2. Tools and material

This section is essential to discuss some related protocols of privacy-preserving computation and abstractions and notations of the SMC protocol under ScereC.

3.2.1. Sharemind protocols

This work assumes that the ABB functionality is implemented through the Sharemind protocol set [34]. We assume that the ABB's operations include the logical operations on private booleans, arithmetic operations on private n-bit integers for various values of n, conversions between them, assignment of the private values, and classifications and declassifications between public and private values. These operations can be performed in the SIMD manner. Similar operations are available in other protocol implementations, e.g., the SPDZ protocol set with security against active adversaries [63].

Since designing applications on top of ABB, we must consider the cost of operations in implementing the ABB via cryptographic protocols. The addition of private integers and the multiplication of a private integer with a public one are assumed to have zero cost because these require no communication between the computation parties. All other operations over private elements have significant latency; hence operating in a SIMD manner is highly desired.

One of the efficient protocols of the Sharemind protocol set [1] is privately *permuting* vectors of private values [116]; we let the ABB give us safe access to these protocols. There is an efficient protocol for generating a private permutation of n elements; these protocols have zero cost round complexity. However, applying this permutation's protocols or inverse to a private vector has $O(1)$ rounds and $O(kn)$ communication, where k is the bit-length of vector elements. SPDZ protocol set [113] gives the same a similar cost.

Laud protocol

One can implement proper subroutines, e.g., sorting a vector of private values [31] on top of the ABB. Later, some applications can call this subroutine, effectively making it a part of the ABB [112]. Another useful subroutine is reading (or writing) from a vector by a private index. Straightforward memory access by a private address cannot be done; hence more complex protocols, often with significant overhead, are necessary. Laud [114] has proposed the subroutines prepareRead and performRead, such that if $[\![\vec{v}]\!]$ is a vector of length n, and $[\![\vec{z}]\!]$ is a vector of integers of length m, all elements of which are between 0 and $(n-1)$, then $[\![\vec{w}]\!] = \mathsf{performRead}([\![\vec{v}]\!], \mathsf{prepareRead}(n, [\![\vec{z}]\!]))$ is a vector of m elements satisfying $w_i = v_{z_i}$ for each i. The subroutine prepareRead requires $O((m+n)\log(m+n))$ communication and $O(\log(m+n))$ rounds, while performRead only requires $O(m+n)$ communication in $O(1)$ round rounds. There exist similar subroutines for writing [114], with $\mathsf{performWrite}([\![\vec{v}]\!], [\![\vec{w}]\!], \mathsf{prepareWrite}(n, [\![\vec{z}]\!]))$ writing the element w_i into the z_i-th position of $\vec{v}$. Also, the communication and round complexities of the two writing routines are the same as the corresponding reading routines.

3.2.2. SecreC programming language and parallel framework

SecreC is a domain-specific language designed for implementing secure multiparty computation applications [29]. The main goals of the language are to strengthen the protection of applications on a high level and efficiently implement an application for those non-familiar with cryptography. The language and its compiler are not embedded in the Sharemind platform. The compiler converts the secret code to a byte code executed over the Sharemind platform.

The SecreC integrated with a Sharemind instance running on the three parties $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$. In Sharemind parties, three copies of the secret code will be manually taken into the three parties of Sharemind, and each party gets a copy of the code. The SecreC compiler compiles the codes xxx.sc to a byte-code xxx.sb. Thereby, Sharemind has to check where the three byte codes are identical, and there is no error before execution starts. Note that the compiler of SecreC is not only for standard functionality, as error checking; it also checks that three codes (one for each computation party) are identical.

The private data type is defined on module pd_shared3p protection domain functions over three parties of Sharemind. It can be module pd_shared2p over two parties only, such as in secure two-party computation [172, 75, 121, 119]. During computation using pd_shared3p, the SMC protocols check the private values of pd_shared3p into three parties to ensure that there are no corrupted values by a passively corrupted party, where the messages are handled by secret-sharing. The challenge will be more if this computation is sequentially performed inside an iteration; often, the challenge is by producing more communication rounds.

The most critical challenge in secure multiparty computations applications is

communication rounds, and more certain specific is a round complexity. SecreC supports a parallel framework for multiple processing elements that simultaneously perform a single instruction on multiple data elements (SIMD approach). In general, the operations that will be executed on the SIMD approach supposed that the multiple data should be vectorized into vectors; a single instruction performs multiple data. Consequently, there is no need to iterate over arrays (or any sequence data structure) to access the elements serially to execute them n time where n is the number of data elements. Let us look at the Listing 3.1, which is SecreC code written sequentially. The procedure is for iterating over vectors to perform multiplication, since multiply element A_i to the corresponding B_i and store the result in X_i. Such a simple computation on top of SMC protocols written sequentially has n round complexity. This function has a high computational cost, i.e., compared with the one that can be written using the SIMD approach for a similar problem because of the network latency of the SMC. The running time will be highly increased when the number of iterations increases. Therefore, using such a technique to build an efficient algorithm for real-world applications is not beneficial.

To solve this problem, the parallel approach in SecreC can be efficiently used to reduce the round complexity of the SMC protocols. The SIMD framework allows multiple data to be processed by single instruction and direct access to the memory of the whole elements without iteration, see Listing 3.2.

Listing 3.1: Sequential multiplication in SecreC

```
template <domain D>
D uint [[1]] Sequential_mult(D uint [[1]] A, D uint [[1]] B) {
    D uint [[1]] X = 0;
    for(uint i = 0; i < size(A); i++)
            X[i] = A[i] * B[i];
    return X;
}
```

This feature of the SIMD framework omitted the **for**-loop in the sequential version of the code. The single round iteration for multiplication on multiple data means a single round complexity on the SMC Sharemind platform. Accordingly, the gap between sequential and parallel versions is noticeably different. SIMD parallel framework efficiently reduces the round complexity, and such a technique can efficiently be used to construct real-world privacy applications.

Listing 3.2: Parallel multiplication in SecreC

```
template <domain D>
D uint [[1]] Parallel_mult(D uint [[1]] A, D uint [[1]] B) {
    D uint [[1]] X = 0;
    X = A * B;
    return X;
}
```

3.2.3. Abstractions and notations for SMC

Abstractions are needed to build upon complex cryptographic protocols like SMC and deduce the result's functional and non-functional properties. A good abstraction of secure multiparty computation is the *arithmetic black box (ABB)* [112], which allows more complex privacy-preserving computations to be described without delving into the details of protocols for primitive operations with private data. The ABB is an ideal functionality in the sense of Universal Composability [47], and its corresponding real functionality consists of the implementations of the protocols for the single operations supported on private data [115]. Its internal state consists of private values entered into it or computed by it; these values cannot be accessed directly by the protocol parties. Instead, the protocol parties may instruct to take some values stored in it (pointing to them through *handles*), perform an operation with them, and return a new *handle* pointing to the result—the handles of the ABB returns are always new. The set of available operations depends on the SMC protocol set used to implement the ABB. The operations may be randomized, e.g., there may be an operation to generate a new, random private value. The ABB also supports instructions for an input party to give an input value (making it accessible to the computing parties through a handle), for a value to be given to a result party (referred by the handle known to the computing parties; the result party obtains the actual value), and for a value to be *declassified* to the computing parties (they learn the true value behind the handle). The ABB only acts if it receives the same instruction from all computing parties.

In order to show that an application built on top of ABB preserves privacy, it is sufficient to show that the declassified values do not give any novel information to the computing parties [112]. We show this by constructing a simulator that samples from the distribution of these declassified values while using only public information to implement protocols for the shortest distances. The kind of adversary against which protection is obtained is the same as for the underlying SMC protocol set. Note that if an application built on top of an ABB never calls the declassification operation, it is trivially privacy preservation.

We present our protocols as algorithms making use of the ABB. The notation $[\![v]\!]$ denotes that the value v is stored in the ABB and accessed by the rest of the algorithm only through a *handle*. E.g., $[\![\vec{v}]\!]$ is a vector of private values; the data type of the values shall be clear from the context. This notation resembles some programming languages [29] used to express privacy-preserving computations; these have the information-flow types `public` and `private` to denote that a value is known to the computation parties resp, that a value is stored inside the ABB. Also, the notation $[\![\mathbf{V}]\!]$ denotes the private adjacency matrix, which is also stored in the ABB.

A value stored in the ABB can be made available to the rest of the algorithm as the outcome of the operation $\mathsf{declassify}([\![v]\!])$, which corresponds to the invocation

of a declassification protocol. We denote the invocations of other primitive protocols working on values stored in the ABB by overloading the notations for the operations that these protocols implement—writing $[\![u]\!] + [\![v]\!]$, or $c \cdot [\![v]\!]$, or $[\![u]\!] \cdot [\![v]\!]$ denotes calls to the addition, or constant multiplication, or multiplication protocol, respectively. Sharemind's protocol set also gives us access to comparison protocols (equality and less-than) and two protocols for operations with Boolean values stored in the ABB. Combining them, we get the protocol for the operation $\mathsf{choose}([\![b]\!], [\![x]\!], [\![y]\!])$. The result of this operation is $[\![x]\!]$, if $[\![b]\!]$ contains true, and $[\![y]\!]$, if $[\![b]\!]$ contains false. The $[\![b]\!]$ value is not leaked by the choose-operation.

One of the most important and usable operations in Sharemind protocols set in our algorithms is min-operation, which has the same functionality but with different shapes based on given arguments (polymorphic function). For $\mathsf{min}([\![\mathsf{x}]\!], [\![\mathsf{y}]\!])$, the result is minimum value in corresponding value either in $[\![x]\!]$ or $[\![y]\!]$. If the given argument is a vector in operation $\mathsf{min}([\![\vec{x}]\!])$, the result is a single minimum value among all vectors' elements. The last shape of the operation $\mathsf{min}([\![\vec{x}]\!], n)$, where the vector $[\![\vec{x}]\!]$ is a $n \times n$ size, operation returns a minimum value for each block, vector $[\![\vec{x}]\!]$ is an n-block, and each block has n-element.

All operations above can be applied to lists (and matrices) of values stored in the ABB. The (private) arguments to the operation must have the same length, resulting in a vector (or matrix) of equal length. As mentioned before, performing many operations in parallel is essential for reducing the round complexity of our algorithms. Besides the SIMD notation, we also use **forall**-loops in our algorithms to denote that all iterations of that loop may be performed in parallel. This contrasts with the **for**-loops, which have to be performed sequentially.

Beside integers and booleans, our algorithms also use *permutations* $[\![\sigma]\!]$ as primitive private values. Our protocols support the operation $\mathsf{randPerm}(n)$ of generating a random private permutation with public length n, the application of the permutation of length n to a vector of private values of length n, resulting in a new vector, where the elements have been permuted according to the permutation, and the application of the *inverse* of the permutation to a vector of private values (again, with the same length). The last two operations are denoted apply, and unApply. They may also be applied to public vectors, but the result is still private.

As data structures, we present our algorithms using pairs, lists, and matrices as input data. A pair of x and y is denoted (x, y), functions first and second take the respective components of a pair. A list/vector is denoted by $\vec{v}$, and its i-th element is denoted either by v_i or $v[i]$. The construction of a vector from the elements $x_1, \ldots, x_n$ is denoted by $[x_1, \ldots, x_n]$. An empty list or vector is denoted by *NIL*, and the concatenation of two lists by @. The notation $\vec{v}[i : j]$ denotes the slice $[v_i, v_{i+1}, \ldots, v_j]$ of the vector $\vec{v}$. Matrices are denoted by boldface letters; the element at the i-th row and j-th column of a matrix $\mathbf{G}$ is denoted by $\mathbf{G}[i, j]$, and the entire row and column vectors are denoted by $\mathbf{G}[i, \star]$ and $\mathbf{G}[\star, j]$, respectively.

Finally, the descriptions of the protocols/algorithms in this thesis deviate somewhat from their encoding in SecreC high-level language (C-based language) be-

cause SecreC does not support certain kinds of expressions and abstractions, e.g., parallel threads executing somewhat different code. In other words, SecreC has no special directives for coding, particularly in parallel frameworks such as OpenMP, MPI, or OpenCL. Hence, if the thesis presents larger pieces of code as running in parallel to each other, then in SecreC it may mean that these pieces of code have been taken apart and combined across the threads to turn them into sequences of SIMD operations.

3.3. Research procedures

The research begins with the research proposal, including the main features of the study and the reasons for engaging in the research project, the goal, novelty, and expected results. This research is conducted in several stages, including studying different cases in graph algorithms and intensive experiments based on various graphs. Compile and classify the results from previous related works. The results show that the gap still has a considerable variance between what is already done and the expected results of research, which will be considered in constructing real-world applications. We reviewed most of the research papers available in the secure multiparty computation of graphs algorithms and many papers on non-privacy-preserving parallel graphs. Research procedures are described as follows.

3.3.1. Data structure

The data structure of standard arrays/matrices is only sometimes suited for SIMD instructions. For example, a vectorized adjacency matrix is fitter for SIMD than a standard. This is why this section has been provided to illustrate re-organizing data input into vectors. Re-organizing data in a way that can be used effectively and compatible with the proposed SMC of parallel graph algorithms. The data structure can play a significant role in constructing the proposed privacy-preserving parallel graph algorithms. There are three main structures for input data used in the proposed protocols.

First, re-shaping the given private adjacency matrix $[\![\mathbf{A}]\!]$ for given graph G into three private vectors (or lists), source vertices $[\![\vec{S}]\!]$, target vertices $[\![\vec{T}]\!]$, and the weights' edges $[\![\vec{W}]\!]$, see Figure 1. All vectors' sizes are the same; this data model is used in privacy-preserving Bellman-Ford protocols in both versions 1 and 2. For version 3, the privacy-preserving Bellman-Ford protocol uses the same structure with one main difference, that the vertices' vectors $\vec{S}$, $\vec{T}$ are public, while $[\![\vec{W}]\!]$ is only private.

Second, this data model vectorizes the adjacency matrix $[\![\mathbf{A}]\!]$ into three vectors/lists, rows $\vec{R}$, columns $\vec{C}$ and the private vector for weights' edges $[\![\vec{W}]\!]$, see Figure 2. The number of rows and columns of the adjacency matrix should be given, which are denoted numR and numC whose arguments will be used on some related functions of the main program. This structure is indicated by a function that transfers grid graph coordinates into a private adjacency matrix. The vectors for both

rows $\vec{R}$ and columns $\vec{C}$ are public. This data structure is used in algebraic path computation protocol.

Third, this model is an ordinary adjacency matrix, which is a square $n \times n$ matrix $[\![\mathbf{A}]\!]$, wherever A_{ij} is non-infinite when there is an edge between two vertices in the graph G, vertex u_i to vertex v_j. We used this data model in Dijkstra, Prim, Johnson, BFS, and Radius-stepping algorithms. The Johnson algorithm uses two data structures together, three vectors for the Bellman-Ford part, and an adjacency matrix for Dijkstra's part since switching between both structures using private writing in laud protocol.

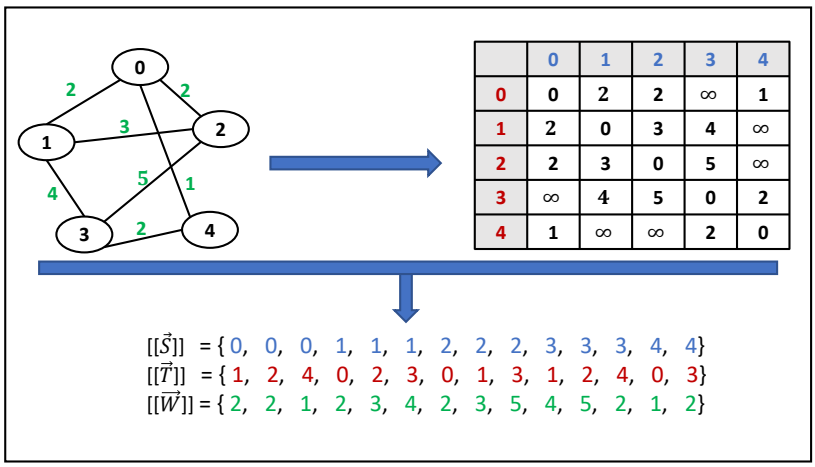


Figure 1: Three vectors representation of the graph.

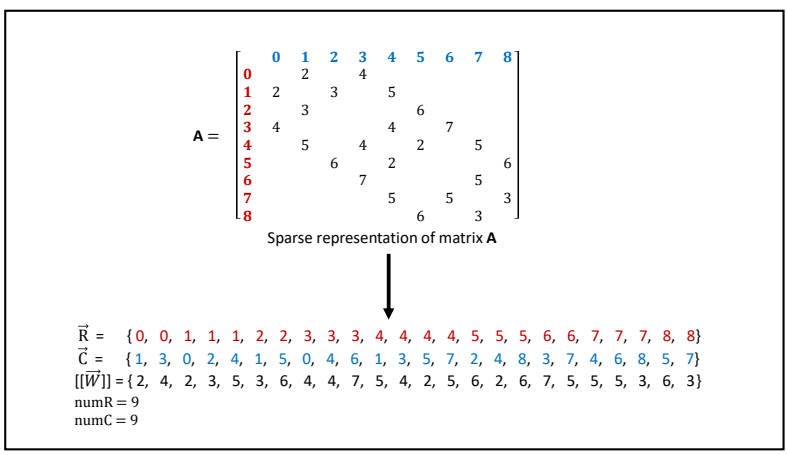


Figure 2: Sparse representation of the graph.

Fourth, this structure is used during computation inside the algorithm, not as data input. It consists of x and y elements stated in vectors and denoted as (x, y). The related algorithm uses this data structure; the algorithms are a minimal first component pair in Prim and Dijkstra algorithms.

3.3.2. Vectorization

The critical point in SIMD implementation is matrix vectorization, a linear transformation of a $m \times n$ matrix into vectors. The vectorization is based on a column vector or transpose of a row vector. The notation $vec(A)$ denotes a vectorized matrix, is $mn \times 1$ column vector expressed by constructing the columns of a $m \times n$ on top of each other, one by one:

$$vec(A) = [a_{1,1}, \ldots, a_{m,1}, a_{1,2}, \ldots, a_{m,2}, \ldots, a_{1,n}, \ldots, a_{m,n}]^T$$

The conversion between column-major and row-major vectorization is based on *commutation matrix*. This means the vectorized shape of a $m \times n$ matrix $vec(A)$ transformation to its vector transpose $vec(A^T)$, as in the following example:

Let A denote the following 3×3 matrix:

$$\mathrm{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

Both column-major and row-major vectorization for matrix A represented as follows, respectively:

$$vec(A) = [A_{col}] = \begin{bmatrix} 1 \\ 4 \\ 7 \\ 2 \\ 5 \\ 8 \\ 3 \\ 6 \\ 9 \end{bmatrix}, \qquad vec(A^T) = [A_{row}] = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{bmatrix}$$

Re-shaping them horizontally is as follows:

$$vec(A) = [A_{col}] = \begin{bmatrix} 1 & 4 & 7 & 2 & 5 & 8 & 3 & 6 & 9 \end{bmatrix},$$
$$vec(A^{T)} = [A_{row}] = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

The input data have been carefully vectorized to support their implementation on a secret-sharing based SMC protocol set. Besides single vectorization for a matrix, our algorithm also performs multiple vectorizations for n matrices into a single large vector, *e.g.*, Dijkstra, and minimum spanning forest protocols. The vectorization should be based on public indices of the given matrices.

Example: Finding matrix multiplication in SIMD parallel, if A is a 3×3 matrix and B is a 3×3 matrix, find $C = A \times B$:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{pmatrix}$$

The matrix multiplication is given by $C = A \times B$

$$C = \begin{pmatrix} a_{11}b_{11}+a_{12}b_{21}+a_{13}b_{31} & a_{11}b_{12}+a_{12}b_{22}+a_{13}b_{32} & a_{11}b_{13}+a_{12}b_{23}+a_{13}b_{33} \\ a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32} & a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32} & a_{21}b_{12}+a_{22}b_{22}+a_{23}b_{32} \\ a_{31}b_{13}+a_{32}b_{23}+a_{33}b_{33} & a_{31}b_{13}+a_{32}b_{23}+a_{33}b_{33} & a_{31}b_{13}+a_{32}b_{23}+a_{33}b_{33} \end{pmatrix}$$

The indices of the vectors, $vec(A)$ and $vec(B)$, are obtained by calling list. 3.3.

Listing 3.3: Vectorize matrices A and B for matrix multiplication

```
uint [[1]] indices_vec(A)(n * n * n) = 0;
uint [[1]] indices_vec(B)(n * n * n) = 0;
uint itr = 0, con = 0;
for(uint k = 0; k < n; k++){
   for(uint i = 0; i < n; i++)
      for(uint j = 0; j < n; j++){
         indices_vec(A)[(i * n + j) + itr] = j + con;
         indices_vec(B)[(i * n + j) + itr] = j * n + i;}
   itr = itr + (n * n);
   con = con + n;}
```

$$indices\_vec(A) = [\, a_{11}, a_{12}, a_{13}, a_{11}, a_{12}, a_{13}, a_{11}, a_{12}, a_{13}, a_{21}, a_{22}, a_{23}, a_{21}, a_{22}, a_{23}, a_{21}, a_{22}, a_{23}, a_{31}, a_{32}, a_{33}, a_{31}, a_{32}, a_{33}, a_{31}, a_{32}, a_{33} \,]$$

$$indices\_vec(B) = [\, b_{11}, b_{21}, b_{31}, b_{12}, b_{22}, b_{32}, b_{13}, b_{23}, b_{33}, b_{12}, b_{22}, b_{32}, b_{12}, b_{22}, b_{32}, b_{12}, b_{22}, b_{32}, b_{13}, b_{23}, b_{33}, b_{13}, b_{23}, b_{33}, b_{13}, b_{23}, b_{33} \,]$$

Thus, matrix multiplication will be computed using single instruction handled by vectors as illustrated in list. 3.4; the result is as follows:

Listing 3.4: Parallel SIMD Matrix multiplication

```
D uint [[1]] vec(A)(n * n * n) = 0;
D uint [[1]] vec(B)(n * n * n) = 0;
D uint [[1]] vec(X)(n * n * n) = 0;
D uint [[1]] vec(C)(n * n * n) = 0;
vec(A) = A[indices_vec(A)];
vec(B) = B[indices_vec(B)];
vec(X) = vec(A) * vec(B);
vec(C) = sum(vec(X),(n*n));
```

The matrix multiplication performance in SIMD is deeply based on the vector size. We first compute the multiplication by multiplying each element in vector $vec(A)$ with its corresponding element in vector $vect(B)$; the result is also stored in the corresponding index in $vec(X)$. Later, compute summation for each $n \cdot n$ value together. The two main operations in matrix multiplication are multiplication and summation; both operations will be computed in the SIMD manner using only single instruction for all vectorized multiple data.

3.3.3. Parallel algorithms design

Constructing the privacy-preserving parallel graphs algorithms is based on essential combinatorial and algebraic graph algorithms. A group of constrictions must be considered in designing the parallel algorithms, SecreC language and its parallel framework, Sharemind protocol set, data structure, and vectorization. Moreover, the SIMD parallel codes of the proposed privacy-preserving graphs algorithms in SecreC language also consider the computational cost of implementing the ABB via cryptographic protocols. SIMD instructions set must target the arithmetic operations of private integers, and Boolean operations are also considered; even assigning operations should be performed in SIMD. All those operations and constrictions will be collectively integrated to construct efficient parallel methods for the graphs on top of SMC protocols. SecreC has specific parallel directives, notations, and protocols for functions (for more details, see Sec 3.2.3).

Topically, the SIMD framework avoids an iteration over operations that operates the private data, and such an iteration yields communication complexity among parties of the SMC platform. The computations occur in all parties $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ of the SMC, and computation processes are the same. In designing parallel algorithms on top of SMC that will be securely processed through ABB, we focus on setting up the parallel instructions on private vectors, which has communication rounds among the computation parties. In contrast, no SIMD instructions are implemented on public vectors because such computations require no communication round.

We assume that privacy-preserving parallel protocols and their subroutines perform computations on top of ABB without leaking data, and their input and output data are private. The protocols have been carefully designed without declassification. Hence, parallel directives and the essential algorithm re-writing do not generally affect declassification or privacy. However, declassification has been used in some algorithms, i.e., Dijkstra and Prim algorithms, to get the real identities of the vertices. The permutation has been used to mask the real identities of the vertices before performing on ABB. Note that in Bellman-Ford protocol version 3, public edges are used to design the algorithms while still privacy-preserving. The aim of permutation usage is that an adversary will get permuted data (encrypted data) even when declassification has occurred, which is mandatory. The permutation is an extra operation since it is necessary for privacy keeping.

In algorithm designing, simplifying algorithms (producing a vectorized version of an algorithm) as much as possible is mainly desired to reduce the round complexity. Due to simplifying the protocols, proposed BFS protocols for finding the shortest path are divided into two versions, weighted and unweighted graphs. The weighted version of BFS has lower private operations than the unweighted (the justification for proposing the two versions of BFS).

Replacing some subroutines or creating a new one is necessary (to fit with SIMD) to use a function more efficiently, e.g., prefixMin2 in Bellman-Ford and getMin in algebraic path computation. Moreover, using some functions that have not existed in essential algorithms is needed because these require keeping the computation without leakage, such as permutation as we mentioned above. The first normalization of the data is also used for sorting elements, while the second normalization is for reducing the size of the data elements, which will let the application perform long data vectors. Both normalizations are used in Bellman-Ford version-3 and algebraic path computation.

Laud's protocol has been used in designing different protocols, not just for the essential functionality of the protocol, which is performing private reads and writes. The second aim of laud's protocol is represented in Johnson's protocols for re-weighting.

3.4. Theoretical framework of parallelism on top of SMC

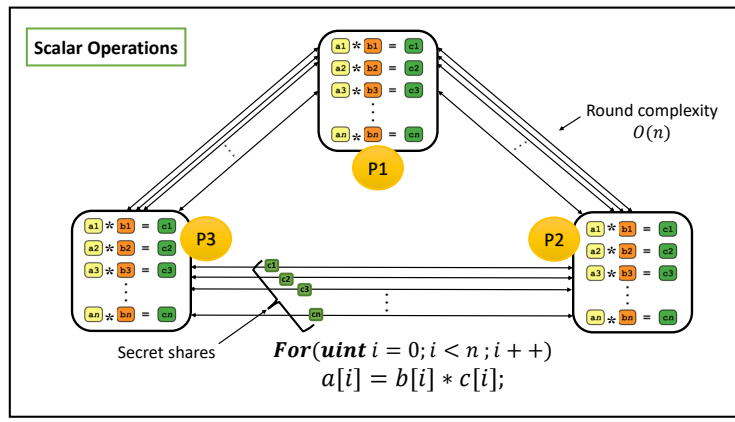
We design the parallel graph algorithms in privacy-preserving based on the combinatorial and algebraic graph algorithms and Sharemind protocols set, including SIMD parallel framework with vectorization. As well as we replaced some related algorithms with other algorithms assumed to be more efficient and more compatible with parallel designing. The main goal of the parallelism presented in this thesis is to identify an efficient set of privacy-preserving protocols with their subroutines, resulting in efficient privacy-preserving protocols of shortest distances and minimum spanning tree and forest.

The efficiency of the proposed privacy-preserving parallel computation protocols is based on keeping the computations in the ABB via cryptographic protocols without declassification; compatibility of the protocols subroutines and their combination with SMC protocols and the round complexity reduction is the most important point.

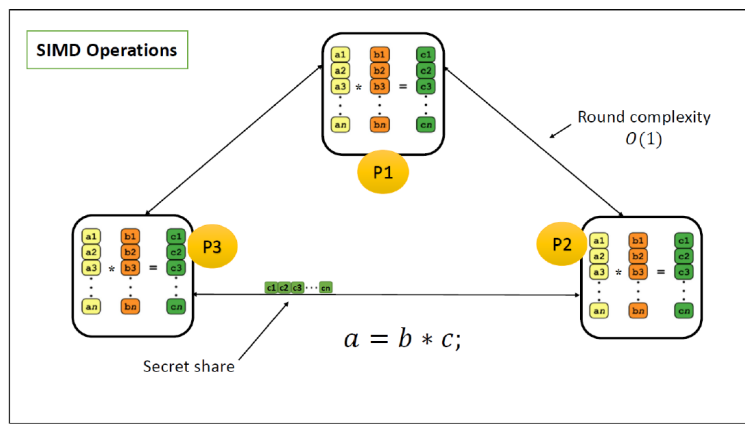
SIMD framework significantly contributes to round complexity reduction for all proposed algorithms. We focus on computation that requires single, repetitive calculations on top of ABB of massive private data sets in parallel algorithm designing. Scalar operations are the conventional sequential approach to processing each data element using one instruction, which iterates n times, one by one, to the end, often with a significant overhead that will produce high computational costs. The difference between the SIMD and scalar operations at the top of ABB on the Sharemind platform is illustrated in Figure 3. It shows the round complexities of a simple multiplication implemented on the Sharemind platform for both operations. With scalar operations, n multiplication operations must be sequentially executed one by one to obtain the result, as shown in Figure 3 (a). Scalar operations process data on the principle of single data = single data $*$ single data. This means that n multiplication instructions must be executed in sequence, producing communications among the computation parties of the Sharemind handled by secret shares. Getting the data indices is done by traversing using **for**-loop. In every single operation, the three parties $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ of Sharemind send messages to each other using secret shares; the estimated round complexity during computation is $O(n)$.

Nevertheless, the same computation result can be efficiently achieved using SIMD operations, requiring less instruction, as shown in Figure 3 (b). SIMD operation works based on the formula of multiple data = multiple data $*$ multiple data. Hence, it can process a large data set using a single instruction. This means that SIMD parallel implementation does not need **for**-loop anymore for traversing overall indices of multiple data; the multiple data will be launched in computation once. Consequently, the three parties of the Sharemind platform $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$, will send messages to each other only once; the round complexity is $O(1)$.

SIMD operations implemented on top of SMC protocol yield lower round complexity than scalar operations. Moreover, SIMD operations may utilize bandwidth usage fully because of the full use of the multiple data of the vectors during processing. Indeed, if the vectors' size used in computations is larger than the bandwidth, the operating system in each party of Sharemind will split the vector into groups. Hence, each data group will be passed as a single message through secret shares; thus, the actual number of the round complexity (which is already reduced) is likely to increase—round complexity will be increased more than once as supposed to be after the reduction.



(a) Sequential implementation.



(b) Parallel implementation.

Figure 3: Framework of algorithms on Sharemind platform.

3.5. Testing and evaluations

This thesis presents many efficient protocols that solve different problems in combinatorial and algebraic graph algorithms. Extensive benchmarking and evaluations of protocols and their subroutines are conducted, including different circumstances of the protocols (with many versions) and their kinds of graphs varying sizes over various networks. The efficient performance in this work is based on the network latency of the SMC platform. Reducing the round complexity by parallel computing means lower network latency since an entire amount of network communication operations will be omitted because of the effectiveness of SIMD on top of SMC protocols, e.g., reducing the round complexity in some subroutines from $O(n)$ to $O(1)$. The performance experiments are based on fundamental measurements that research the aspects of performance gain and its evaluations. These fundamental measurements include *speed-up*, *data volume*, *serial fraction* and their *scalability*.

Speed-up

In general, the speed-up is the ratio of running time achieved by the best sequential algorithm $\mathcal{T}_s$ to the running time performed by the possible parallel algorithm $\mathcal{T}_p$, where p is the number of processors or threads—both versions of the algorithms are for solving the same problem [6]. The definition of speedup achieved by a parallel algorithm given by $\mathcal{S} = \mathcal{T}_s/\mathcal{T}_p$. The running time of the sequential version of the essential algorithms in this work is too high and inaccessible for comparison; hence, we do not implement the sequential counterpart of all algorithms. For instance, we only show the running time of the parallel algorithms. The running time of the algorithms, either for parallel counterpart or sequential counterpart, is the average time over the three parties $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ of the Sharemind platform given by $\mathcal{T} = (\mathcal{P}_1+\mathcal{P}_2+\mathcal{P}_3)/3$. The running time of the benchmark is reported in seconds. We also use the suffix $''k''$ to denote the multiplication by 1000, and the $''M''$ to denote the multiplication by 1000000.

Data Volume

The proposed SIMD parallel algorithms have changed the amount of data transferred among the computation parties of the SMC platform. The data amount transmitted by the sequential algorithms is n-bit integers (or Boolean) handled by secret shares. The transmitted data through parallel algorithms are vectors of private data, also by secret shares. Consequently, data volume is measured in all proposed algorithms and subroutines over different networks with different characteristics to study the trade-offs between data volume consumption with round complexity and the scalability of the parallel algorithms with their reduction for around complexity. The data volume of the parallel algorithms over the SMC platform is obtained by getting the sum of the transmitted data among the number of computation parties given by $\mathcal{DV} = (\mathcal{DV}_1+\mathcal{DV}_2+\mathcal{DV}_3)/3$.

Serial fraction

Some proposed protocols for finding the shortest path and minimum spanning tree can perform parallel execution on multiple graphs of similar sizes. The performance gains resulting from parallel execution on the n graphs are measured meaningfully. Hence, the experiments consider the different amounts of graphs (which scalably increased), the graph sizes, and the network environments. The serial fraction is the fundamental measurement for testing the parallel algorithm that runs n graphs in parallel. To obtain the serial fraction, which tells us about the scalability of the parallelization, we use the Karp-Flatt metric [101]. The parallelization is noticeably limited in instances of using large-size vectors that will be transmitted among parties of the SMC platform. Indeed, parallelization is effectively done on parties that produce large vectors bigger than the volume of data (that will be sent among computation parties). The operating system of the parties cannot send such a large vector; hence the operating system splits a large vector into small vectors (size is the capacity of the bandwidth or less) that will be sequentially sent one by one. In such cases, serial fraction characterizes how much of the computation has not benefited from the parallelization.

3.6. Setup experiments and Hardware architecture

The protocols described in this work are implemented on top of Sharemind's three-party protocol set secure against one passively corrupted party [32, 34], making use of the Se-

creC high-level language [30] and other development tools included with the Sharemind platform. The actual implementations used the single-instruction-multiple-data framework supported by the SecreC high-level language to write the codes. The benchmarking of all implementations is done on the Sharemind cluster of three servers connected, where each server is 12-core 3 GHz CPUs with Hyper-Threading running Linux and 48 GB of RAM, connected by an Ethernet local area network with a link speed of 1 Gbps. Single-threaded setup is used in all Sharemind's implementations; hence no usage for multiple cores performing local operations, nor the possibility of performing computations over distributed systems simultaneously.

Sharmind uses the additive secret-sharing scheme among the three servers over different rings of integers, and it has been proven that Sharemind provides strong privacy guarantees under the honest-but-curious model [94]. The most crucial feature of Sharemind is tolerating one passive corrupted party, and the system performs significantly better than similar systems—based on SMC protocol—as FairplayMP [18] and VIFF. This is the justification for using Sharemind for our implementations. Furthermore, if the Sharemind system has four computation parties, security can be achieved if there is one actively malicious party.

The central processing unit of the Sharemind system can execute operations sequentially and in parallel (SIMD) on private or public data. Algorithms can be written somehow to describe how the data should be processed using Sharemind. SecreC programming language is founded for such a purpose. Furthermore, Sharemind supports vectorized operations to allow the parallelization of algorithms to boost their performance, which is the main feature of our proposed parallel protocols. Consequently, if such frameworks—based on the secure multiparty computation—have similar features to Sharemind, our proposed parallel protocols are also run on it.

Network environments

The proposed protocols have been constructed based on achieving the critical point of the research: a reduction in the round complexity of SMC protocols. SIMD implementations yield lower round complexity, increasing the amount of data transmitted simultaneously. Thus, the protocols have many trade-offs between round complexity and communication consumption since their relative performance over various network environments may differ. For characterizing the variation in performance and checking the scalability, three different network environments are used in the implementations, throttling the connection between the servers in our Sharemind cluster.

- **High-Bandwidth Low-Latency (HBLL)**
  In the "high-bandwidth" setting, the link speeds between servers are 1Gbps, while in the "low-latency" setting, we have not delayed the messages between the servers. We use these environments to run our experiments, i.e., in our local area network.
- **High-Bandwidth High-Latency (HBHL)**
  In the "high-latency" setting, the messages are delayed by 40ms.
- **Low-Bandwidth High-Latency (LBHL)**
  In the "low-bandwidth" setting, the speeds are 100Mbps. Thus, simulating wide-area networks with different characteristics in both environments, HBHL and LBHL.

4. PRIVACY-PRESERVING SINGLE-SOURCE SHORTEST PATH PROTOCOLS

4.1. Introduction

This chapter presents the proposed single-source shortest path protocols and their related algorithms. As well as at the end of the chapter, show the performance analysis of the proposed protocols, round, and communication complexities. Also, present the security and privacy of protocols.

4.2. Privacy-preserving Dijkstra's protocols for dense graphs

4.2.1. Dijkstra's protocol for a single graph

Dijkstra's algorithm relaxes each edge only once, in the order of the distance of its start vertex from the source vertex. The edges with the same starting vertex can be relaxed in parallel. The algorithm cannot handle edges with negative weights, as described above. As Dijkstra's algorithm only holds a few edges in parallel, Laud's parallel reading and writing subroutines will be of little use here. Instead, we opt to use the dense representation of the graph, giving the weights of the edges in the adjacency matrix (weight "∞" is used to denote the lack of an edge).

Algorithm 7: Privacy-Preserving Dijkstra

Data: Number of vertices n, starting vertex s
Data: Lengths of edges $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{n \times n}$
Result: Private distances from the starting vertex

begin
- $[\![\sigma]\!] \leftarrow \mathsf{randPerm}(n)$
- **forall** $u \in \{0,\ldots,n-1\}$ **do**
  - $[\![\mathbf{G}'[u,\star]]\!] \leftarrow \mathsf{apply}([\![\sigma]\!],[\![\mathbf{G}[u,\star]]\!])$
- **forall** $v \in \{0,\ldots,n-1\}$ **do**
  - $[\![\mathbf{G}'[\star,v]]\!] \leftarrow \mathsf{apply}([\![\sigma]\!],[\![\mathbf{G}'[\star,v]]\!])$
- $s' \leftarrow \mathsf{declassify}((\mathsf{unApply}([\![\sigma]\!],[0,1,..,n-1]))[s])$
- $[\![\vec{D}]\!] \leftarrow \infty$
- $[\![D[s']]\!] \leftarrow 0$
- $\vec{M} \leftarrow \mathsf{true}$ `// length of` $\vec{M}$ `is` n
- **for** $idx = 0$ **to** $n-1$ **do**
  - $[\![\vec{L}]\!] \leftarrow NIL$
  - **for** $i = 0$ **to** $n-1$ **do**
    - **if** $M[i]$ **then** $[\![\vec{L}]\!] \leftarrow [([\![D[i]]\!],[\![i]\!])] \mathbin{@} [\![\vec{L}]\!]$
  - $u' \leftarrow \mathsf{declassify}(\mathsf{second}(\mathsf{minLs}([\![\vec{L}]\!])))$
  - $M[u'] \leftarrow \mathsf{false}$
  - $[\![\vec{E}]\!] \leftarrow [\![\mathbf{G}'[u',\star]]\!] + [\![D[u']]\!]$
  - $[\![\vec{D}]\!] \leftarrow \mathsf{choose}(\vec{M} \wedge ([\![\vec{E}]\!] < [\![\vec{D}]\!]),[\![\vec{E}]\!],[\![\vec{D}]\!])$
- **return** $\mathsf{unApply}([\![\sigma]\!],[\![\vec{D}]\!])$

Our privacy-preserving implementation of Dijkstra's protocol is presented in Algorithm 7. The main body of the algorithm is its last loop (lines 11-18). It starts by finding the unhandled vertex closest to the source vertex s. The mask vector $\vec{M}$ indicates which vertices are still unhandled. The list $[\![\vec{L}]\!]$ contains all vertices not in $\vec{M}$, with their distance from $\vec{M}$. Adding a pair to the list is by cons-function. Repeatedly, the index of an unhandled vertex is found by the function minLs given in Algorithm 8, which, when applied to a list of pairs, returns the pair with the minimal first component. We call minLs with a list where the first components are current distances, and the second components are the indices of vertices. Algorithm 8 works by dividing the vector into two parts and iterating $\log n$ times. Then, it picks up the minimum values in the vectors until the minimum pair. Moreover, it picks up the identity of the permuted vertex by second-function; the value is the second component of the pair.

In non-privacy-preserving implementations, priority queues can find the next vertex to quickly relax its outgoing edges. In privacy-preserving implementations, the queues are challenging to implement efficiently due to their complex control flow. Hence the next vertex is found by computing the minimum over the current distances for all vertices not yet handled.

Algorithm 8: minLs: Pair of the minimal first component

Data: List of pairs of private values $[\![\vec{p}]\!]$
Result: The element of $[\![\vec{p}]\!]$ with the minimal first component

```
1 begin
2     m ← length([[p⃗]]) − 1
3     if m = 0 then return [[p[0]]]
4     In parallel do
5         ([[e]], [[i]]) ← minLs([[p⃗[0 : ⌊m/2⌋]]])
6         ([[f]], [[j]]) ← minLs([[p⃗[⌊m/2⌋ + 1 : m]]])
7     return if [[e]] ≤ [[f]] then ([[e]], [[i]]) else ([[f]], [[j]])
```

Algorithm 7 declassifies the index of the unhandled vertex closest to the source vertex. This declassification greatly simplifies the computation of $[\![\vec{E}]\!]$, where the current distance of u' is added to the length of all edges starting at u'. Without declassification, one would need to use techniques for private reading here, which would be expensive. Note that both the computations of $[\![\vec{E}]\!]$ and $[\![\vec{D}]\!]$ take place in a SIMD manner, applying the same operations to $[\![E[i]]\!]$, $[\![\mathbf{G}[u',i]]\!]$, $[\![D[i]]\!]$, and $M[i]$ for each $i \in \{0,\ldots,n-1\}$.

This declassification constitutes a leak. Effectively, the last loop declassifies in which order the vertices are handled, i.e., how they are ordered concerning their distance from the source vertex. Such leakage can be neutralized by randomly permuting the graph's vertices before that last loop [3]. In this way, the indices of the vertices, when they are declassified, are random. The declassification would output a random permutation of the set $\{0,\ldots,n-1\}$, one element at each iteration.

The computation of this random permutation takes place at the beginning of Algorithm 7 (lines 2-7). It starts by performing the given undirected graph's permutation to mask the vertices' real identity and their connected edges. We first generate a private random permutation $[\![\sigma]\!]$ for n elements. We will then apply it to each row of $[\![\mathbf{G}]\!]$; the application takes place in parallel for all rows. Similarly, we apply it to each column. Such permutation also changes the index of the source vertex, and we have to find it. We

find it by taking the identity vector of length n, applying to it the *inverse* of σ, and then reading the position s of the resulting vector. Note that we declassify only the position s at this time, not the entire vector. At the end of the computation, we have to apply the inverse of σ to the computed vector of distances.

4.2.2. Vectorizing Dijkstra's protocol

Dijkstra's algorithm is used as a subroutine in specific APSD algorithms (Sec 5.2). Hence, we have also implemented a *vectorized* version of Algorithm 7 that can simultaneously compute the SSSD for several n-vertex graphs.

Algorithm 9: SIMD-nDijkstra

Data: Number of vertices n, starting vertices $\vec{s}$
Data: Lengths of edges $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{g \times n \times n}$
Result: Private distances from the starting vertices

begin
  for $i = 0$ **to** $g-1$ **do**
    $[\![\sigma[i,\star]]\!] \leftarrow \mathsf{randPerm}(n)$
    forall $u \in \{0,\ldots,n-1\}$ **do**
      $[\![\mathbf{G}'[i,u',\star]]\!] \leftarrow \mathsf{apply}([\![\sigma[i,\star]]\!], [\![\mathbf{G}[i,u,\star]]\!])$
    forall $v \in \{0,\ldots,n-1\}$ **do**
      $[\![\mathbf{G}'[i,\star,v']]\!] \leftarrow \mathsf{apply}([\![\sigma[i,\star]]\!], [\![\mathbf{G}'[i,\star,v]]\!])$
    $s'[i] \leftarrow \mathsf{declassify}(\mathsf{unApply}([\![\sigma[i,\star]]\!], [0,1,..,n-1]))$
  $[\![\vec{D}]\!] \leftarrow \infty$, $[M] \leftarrow \mathsf{false}$
  for $i = 0$ **to** $n-1$ **do**
    $[\![D[k+s'[i]]]\!] \leftarrow 0$ // $k = i \cdot n$
    for $j = 0$ **to** $n-1$ **do**
      $[\![P[k+j]]\!] = j$
      $d[k+j] = i$
  for $i = 0$ **to** $n-1$ **do**
    $[u'] \leftarrow \mathsf{declassify}((\mathsf{minLv}([\![i,[\![\vec{P}]\!]]\!], \vec{M}, \vec{d}))$
    $start = 0, end = 0, range = 0$
    for $u = 0$ **to** $g-1$ **do**
      $range = u * n + u'[u]$
      $end = start + n$
      $Ws[\![start : end]\!] = [\![\mathbf{G}'[u,u'[u],\star]]\!]$
      $nD[start : range] = range$
      $M[range] = \mathsf{true}$
      $start = start + n$
    forall $i \in \{0,\ldots,nD-1\}$ **do**
      $[\![Ds[i]]\!] \leftarrow [\![\vec{D}]\!]$
    $[\![\vec{E}]\!] \leftarrow [\![\vec{D}s]\!] + [\![\vec{W}s]\!]$
    $[M] = \mathsf{false}$
    $[\![\vec{D}]\!] \leftarrow$ **if** $\vec{M} \wedge ([\![\vec{E}]\!] < [\![\vec{D}]\!])$ **then** $[\![\vec{E}]\!]$ **else** $[\![\vec{D}]\!]$
  return $\mathsf{unApply}\ (\ [\![\sigma[\star,\star]]\!], [\![\vec{D}]\!]\ )$

We call this version of the algorithm nDijkstra, presented in Algorithm 9. Dijkstra and nDijkstra have the same round complexity. In contrast, the communication usage of the latter is n times the former (when finding SSSD for g graphs simultaneously).

Generally, the vectorized version of the privacy-preserving Dijkstra protocol consists of two main parts. The permutation part (lines 2-8) finds the permutation for each graph separately. The algorithm generates a private random permutation $[\![\sigma]\!]$ for n element for each graph. Later, the algorithm shuffles the rows and columns for each graph using permutation $[\![\sigma]\!]$ in parallel. During permutation, permuted graphs $[\![\mathbf{G}^{'}]\!]$ will be stored in a 3-dimensional adjacency matrix $[\![\mathbf{G}^{'}[g,\star,\star]]\!]$. This permutation changes the position of source vertices $\vec{s}$ for each graph. To find the new identity of the starting vertices $\vec{s}$, we apply the inverse of $[\![\sigma]\!]$ to the identity vector of length n and then read the declassified position $s^{'}$ of the resulting vector for each graph. The part (lines 10-14) is for getting the indices; vector $\vec{d}$ is public, while vector $[\![\vec{P}]\!]$ and $[\![\vec{D}]\!]$ are private. It would be better to assign them all in parallel using **forall**-loop.

Algorithm 10: minLv: Minimal values for n-vector

Data: Vector for n graphs $[\![\vec{K}]\!]$
Data: Boolean vector of unhandled vertices $\vec{M}$
Data: The private indices $[\![\vec{d}]\!]$
Result: The minimum distance $[\![\vec{u}]\!]$ for all graphs

1 **begin**
2 $\quad [\![\vec{K}]\!] \leftarrow$ **if** $(M \neg \mathsf{false})$ **then** $[\![\vec{\infty}]\!]$ else $[\![\vec{K}]\!]$
3 $\quad [\![\vec{S}]\!] \leftarrow \mathsf{min}([\![\vec{K}]\!], n)$
4 $\quad$ **forall** $i \in \{0,1,...,n-1\}$ **do**
5 $\quad\quad [\![V_i]\!] \leftarrow [\![\vec{S}]\!]$
6 $\quad [\![\vec{w}]\!] \leftarrow$ **if** $([\![\vec{V}]\!] = [\![\vec{K}]\!])$ **then** $[\![\vec{d}]\!]$ **else** $[\![\vec{\infty}]\!]$
7 $\quad [\![\vec{u}]\!] \leftarrow \mathsf{min}([\![\vec{w}]\!], n)$
8 $\quad$ **return** $[\![\vec{u}]\!]$

The second part of the algorithm is finding the shortest paths for all graphs simultaneously (lines 15-29) in Algorithm 9, all graphs $[\![\mathbf{G}]\!] \in \mathbb{Z}^{n\times n}$ will be carried and processed only once. The main **for**-loop is the main body of vectorized Dijkstra algorithm, which starts by calling the minLv-function, see Algorithm 10. This function is also vectorized version of the minim first component procedure (Algorithm 8). It can process a group of sets of vectors at one time and return a vector of elements; each element is a minimization value of a graph. The return value of minLv will be declassified to get it as public, with no threading for privacy because values are already masked in the permutation's part.

The inner for-loop (lines 18-24) is where we perform vectorization of the given matrices and the state of the visited vertices. The graphs $\mathbf{G}^{'}$ will be vectorized into vector $[\![\overrightarrow{W^{'}s}]\!]$ in each iteration. The vector has a set of rows, each from a graph, based on the value of $u^{'}[u]$. Furthermore, the Boolean vector $\vec{M}$ gets true-value for visited vertices. The last operations are similar to Algorithm 7. Finally, the return value is an adjacency matrix containing the shortest distances, each row for the corresponding graph, by applying unApply-function to get the vertices' actual identity.

4.3. Privacy-preserving Bellman-Ford protocols for sparse graphs

The section presents the two privacy-preserving Bellman-Ford protocols on a sparse graph. The main program of both protocols' versions is similar, while the main difference is the parallel algorithm of PefixMin2 subroutines.

4.3.1. Bellman-Ford protocol (Version 1)

The Bellman-Ford algorithm repeatedly relaxes all edges in parallel until the mapping $[\![\vec{D}]\!]$ changes no more. In the worst case, there may be $n-1$ iterations. We show how to run the Bellman-Ford algorithm in a privacy-preserving manner on top of the Sharemind-inspired ABB described above, applying it to graphs represented sparsely. The representation that we consider consists of two public numbers n and m of vertices and edges, and three private vectors $[\![\vec{S}]\!]$, $[\![\vec{T}]\!]$, and $[\![\vec{W}]\!]$ of length m, where the elements of the first two vectors belong to the set $\{0,\ldots,n-1\}$. In this setting, the i-th edge of the graph has the start and end vertices $[\![S[i]]\!]$ and $[\![T[i]]\!]$, and the weight $[\![W[i]]\!]$. We see that our representation hides the entire structure of the graph (besides its size given by n and m), such that even the degrees of vertices remain private.

Algorithm 11: Privacy-preserving Bellman-Ford, main program

Data: Numbers of vertices and edges n and m
Data: Starting vertex s
Data: Sources, targets, and weights $[\![\vec{S}]\!]$, $[\![\vec{T}]\!]$, and $[\![\vec{W}]\!]$
Requires: $[\![\vec{T}]\!]$ is sorted
Requires: The in-degree of each vertex is at least 1
Requires: There is a loop edge of length 0 at vertex s
Result: Private distances from vertex s

```
1  begin
2      [Z] ← GenIndicesVector([T])
3      [R_S] ← prepareRead(n, [S])
4      [R_Z] ← prepareRead(m, [Z])
5      [D] ← ∞
6      [D[s]] ← 0
7      for i = 0 to n − 1 do
8          [a] ← performRead([D], [R_S])
9          [b] ← [a] + [W]
10         [c] ← prefixMin2([b], [T])
11         [D] ← performRead([c], [R_Z])
12     return [D]
```

The algorithm for computing distances from the s-th vertex is given in Algorithm 11, with subroutines in Algorithm 12 and Algorithm 13. Suppose the requirements stated at the beginning of Algorithm 11 are not satisfied. In that case, this can be remedied easily and in a privacy-preserving manner by adding extra edges to the graph (increasing the length of the vectors $[\![\vec{S}]\!]$, $[\![\vec{T}]\!]$, and $[\![\vec{W}]\!]$), and then sorting the inputs according to $[\![\vec{T}]\!]$.

The chosen setting brings with it several challenges when relaxing edges. To relax the i-th edge, the algorithm must locate $D(S[i])$. However, $S[i]$ is private. Fortunately, the

parallel reading subroutine is applicable as we relax all edges in parallel. Moreover, as the indices $S[i]$ stay the same over the iterations of the algorithm, we can invoke the (relatively) expensive prepareRead-routine only once and use the linear-time performRead-routine in each iteration. This can be seen in Algorithm 11, where we call prepareRead on $[\![\vec{S}]\!]$ at the beginning, and then do a performRead at the beginning of each iteration. We will then compute $[\![\vec{b}]\!]$ as the sum of the current distance of the start vertex of an edge and the length of that edge.

After computing the sums $b[i] = D[S[i]] + W[i]$, the value $D[T[i]]$ has to be updated with it, if it is smaller than any other $b[j]$ where $T[i] = T[j]$. Due to the loop edge of length 0 at the starting vertex, we have simplified our computations by eliminating the need to consider the old value of $D[T[i]]$ when updating it. Such updates map straightforwardly to parallel writing. In parallel writing, concurrent writes to the same location must be resolved somehow. Currently, we want the smallest value to take precedence over others, i.e., the value is precedence. The available parallel writing routines support such precedences. However, these precedences, which change each round, are part of the inputs to prepareWrite, hence would introduce significant overhead to each iteration.

Algorithm 12: PrefixMin2 (version 1)

Data: Vector of values $[\![\vec{b}]\!]$
Data: Vector of ranges $[\![\vec{T}]\!]$
Result: Prefix minimum for elements of $\vec{b}$, separately for each range of $\vec{T}$

Function min2($[\![x]\!], [\![x']\!], [\![y]\!], [\![y']\!]$) **is**
  $[\![q]\!] \leftarrow \min([\![y]\!], [\![y']\!])$
  return choose($[\![x]\!] = [\![x']\!], [\![q]\!], [\![y']\!]$)
begin
  $n \leftarrow \text{length}([\![\vec{b}]\!])$
  if $n = 1$ **then return** $[\![\vec{b}]\!]$
  forall $i \in \{0, \ldots, \lfloor n/2 \rfloor - 1\}$ **do**
    $[\![U_i]\!] \leftarrow [\![T_{2i+1}]\!]$
    $[\![d_i]\!] \leftarrow \text{min2}([\![T_{2i}]\!], [\![T_{2i+1}]\!], [\![b_{2i}]\!], [\![b_{2i+1}]\!])$
  $[\![\vec{e}]\!] \leftarrow \text{prefixMin2}([\![\vec{d}]\!], [\![\vec{U}]\!])$
  $[\![r_0]\!] \leftarrow [\![b_0]\!]$
  forall $i \in \{1, \ldots, n-1\}$ **do**
    if *i is odd* **then**
      $[\![r_i]\!] \leftarrow [\![e_{(i-1)/2}]\!]$
    else
      $[\![r_i]\!] \leftarrow \text{min2}([\![T_{i-1}]\!], [\![T_i]\!], [\![e_{(i-2)/2}]\!], [\![b_i]\!])$
  return $[\![\vec{r}]\!]$

This work shows how to reduce the updates in parallel reading according to indices that stay the same for each iteration. It requires us first to compute the minimum distances for all vertices while being oblivious to each edge's end vertex. Thanks to $[\![\vec{T}]\!]$ being sorted, the edges ending at the same vertex form a single segment in the vector $[\![\vec{b}]\!]$. For each such segment, we will compute its minimum, which will be stored in vector $[\![\vec{c}]\!]$, at the index corresponding to the last vertex of that segment. That element can be read

out using another performRead. From where to read, the indices have been stored in the vector $[\![\vec{Z}]\!]$. We compute the minima for segments with private starting and ending points through prefix computation, where the applied associative operation is similar to the minimum. Consider the following operation:

$$\mathsf{min2}((x,y),(x',y')) = \begin{cases} (x', \min(y,y')), & \text{if } x = x' \\ (x',y'), & \text{if } x \neq x' \end{cases} \tag{4.1}$$

Suppose we zip the vectors $\vec{T}$ and $\vec{b}$ (obtaining a vector of pairs), and then compute the prefix-min2 of it. We end up with a vector of pairs whose first components give us back the vector $\vec{T}$, and whose second components are the prefix-minima of the segments of $\vec{b}$ corresponding to the segments of equal elements in $\vec{T}$. The second case of (4.1) ensures that prefix minimum computation is broken at the end of segments. It is easy to verify that min2 is associative.

We use the Ladner-Fisher parallel prefix computation method [111] to compute privacy-preserving prefix-min2 in a round- and work-efficient manner. The computation is given in Algorithm 12. The write-up of the computation is simplified by the $\vec{T}$-component not changing during the prefix computation. Hence prefixMin2 returns only the list of the second components of pairs. Similarly, the subroutine min2 returns only a single value. Our ABB supports all operations in Algorithm 12.

Algorithm 13: GenIndicesVector

Data: Sorted vector $[\![\vec{v}]\!]$
Result: Private vector of indices of the last occurrence of each value in $\vec{v}$

```
1  begin
2      m ← length([[v]])
3      [[b]] ← ([[v[0 : m−2]]] = [[v[1 : m−1]]]) @ [true]
4      [[σ]] ← randPerm(m)
5      c ← declassify(apply([[σ]], [[b]]))
6      [[k]] ← apply([[σ]], [0, 1, ..., (m−1)])
7      [[l]] ← NIL
8      for i = 0 to m−1 do
9          if c[i] then
10             [[l]] ← [[k[i]]] @ [[l]]
11     return sort([[l]]) ;                    // Use oblivious quicksort [31]
```

The computation of the vector $[\![\vec{Z}]\!]$ of the indices of the ends of the segments of equal elements in $[\![\vec{T}]\!]$ in Algorithm 13 uses standard techniques. We first compute the end position's index vector $[\![\vec{b}]\!]$. The length of that vector is m, and exactly n of its elements are true. We randomly permute $[\![\vec{b}]\!]$ using apply-routine, each element of the private vector $[\![\vec{b}]\!]$ located in the same content as the index i of the original sorted vector $[\![\vec{v}]\!]$, then convert the data type of the result from private to public locations using declassify-routine. The result of declassification is a random Boolean vector of length m, where exactly n elements are true. The distribution of this result can be sampled by knowing n and m; there is no further dependence on $[\![\vec{b}]\!]$. Hence this declassification does not break the privacy of our SSSD algorithm. We permute the identity vector similarly, resulting in

the private vector $[\![\vec{k}]\!]$. Now the indices we are looking for are located in these elements $[\![k[i]]\!]$, where $c[i]$ is true. Hence we pick them out by applying cons-routine. They have been shuffled by $[\![\sigma]\!]$; this has to be undone by sorting them.

If the graph contains negative-length cycles, there are generally no shortest paths because any path can be made cheaper by one more walk around the negative cycle. We could amend Algorithm 11 to detect these negative cycles in the standard manner by doing one more iteration of its main loop and checking whether there were any changes to $[\![\vec{D}]\!]$.

4.3.2. Bellman-Ford protocol (Version 2)

The computation of prefixMin2 is the most complex step in the main loop of Algorithm 11. The Ladner-Fisher method [111] for its computation is communication- and round-efficient—the number of rounds is logarithmic (assuming each arithmetic operation takes a constant number of rounds), and the total number of operations is only a constant times larger than a sequential implementation would have had. Still, different trade-offs between communication and round complexity are possible. To study these trade-offs, we have also implemented prefixMin2 based on the Hillis-Steele parallel prefix computation method [91]. This alternative implementation is given in Algorithm 14. Replacing its call in Algorithm 11 gives us our second version of the Bellman-Ford protocol.

Algorithm 14: prefixMin2 (version 2)

Data: Vector of values $[\![\vec{b}]\!]$
Data: Vector of ranges $[\![\vec{T}]\!]$
Result: Prefix minimum for elements of $\vec{b}$, separately for each range of $\vec{T}$

1 **begin**
2 $\quad n \leftarrow \mathsf{length}([\![\vec{b}]\!])$
3 $\quad$ **for** $j = 1$ **to** $\lfloor \log n \rfloor$ **do**
4 $\quad\quad [\![\vec{d}[0 : 2^j - 1]]\!] \leftarrow [\![\vec{b}[0 : 2^j - 1]]\!]$
5 $\quad\quad [\![\vec{U}[0 : 2^j - 1]]\!] \leftarrow [\![\vec{T}[0 : 2^j - 1]]\!]$
6 $\quad\quad [\![\vec{d}[2^j : n - 1]]\!] \leftarrow [\![\vec{b}[0 : n - 2^j - 1]]\!]$
7 $\quad\quad [\![\vec{U}[2^j : n - 1]]\!] \leftarrow [\![\vec{T}[0 : n - 2^j - 1]]\!]$
8 $\quad\quad [\![\vec{e}]\!] \leftarrow \mathsf{min}([\![\vec{b}]\!], [\![\vec{d}]\!])$ `// elementwise minimum of two vectors`
9 $\quad\quad [\![\vec{b}]\!] \leftarrow \mathsf{choose}([\![\vec{T}]\!] = [\![\vec{U}]\!], [\![\vec{e}]\!], [\![\vec{b}]\!])$
10 $\quad$ **return** $[\![\vec{b}]\!]$

Algorithm 14 has a single loop that is executed $\log n$ times. Each iteration's round complexity equals the sum of round complexities of finding the minimum and an oblivious choice. The first of these dominates, as the oblivious choice only requires a single round of communication. Compared to Algorithm 12, we have reduced the round complexity by around two times. On the other hand, we have increased data volume usage by ca. $\log n$ times.

4.4. Privacy-preserving radius-stepping protocol

In this section, we describe our algorithm for parallel privacy-preserving single-source shortest distances by radius-stepping and present its performance analysis with details. Let $G = (V, E)$ be an edge-weighted undirected graph, where V is a set of vertices, and E is a set of edges for the graph G. An edge e from the vertex u to the vertex v has a weight denoted as $w(e)$ or $w(u, v)$. The weights w in the graph G are assumed to be non-negative integers in a finite range in the case of the Radius-Stepping algorithm. The data of the graph G is presented in an adjacency matrix as integers; if there is no edge e among two vertices u and v, then the weight is $w(u, v) = \infty$. The algorithm for computing the shortest paths has three main subroutines, summation, minimum, and reweighing. The query determines the minimal cost $\delta_G(u, v)$ from vertex u to vertex $v \in V$. The radius-Stepping algorithm is an SSSD algorithm highly recommended for use on graphs edges like the planar and non-planar. The algorithm works efficiently with low time complexity compared to the Δ-Stepping algorithm, while both are parallel SSSD algorithms. Our contribution is constructing and testing a parallel privacy-preserving shortest path for different types of graphs through SIMD in the Sharemind SMC platform. The graphs' types are planar graphs (or like-planar) and dense graphs. The privacy-preserving radius-stepping algorithm is geared more toward dense graphs.

Algorithm 15: Privacy-Preserving Radius-Stepping

Data: Number of vertices and edges n, m
Data: Adjacency matrix $[\![\mathbf{G}]\!] \in \mathbb{Z}^{n \times n}$
Data: Vertex radii $[\![\vec{r}]\!] \in \mathbb{N}^n$, source vertex s
Result: The distances $[\![\vec{\delta}]\!]$ from s

begin
  $[\![\vec{S}]\!] = \mathsf{false}$ // length of $\vec{S}$ is n
  $[\![\vec{\delta}]\!] \leftarrow [\![\mathbf{G}[s, \star]]\!]$, $[\![S[s]]\!] \leftarrow \mathsf{true}$, $[\![\delta[s]]\!] \leftarrow 0$, $[\![D]\!] \leftarrow 0$
  repeat
    $[\![\vec{\delta}']\!] \leftarrow$ **Bellman-Ford-Step**$(n, [\![\mathsf{G}]\!], [\![\vec{S}]\!], [\![\vec{\delta}]\!], [\![D]\!])$
    $[\![B]\!] \leftarrow ([\![\vec{\delta}']\!] \neq [\![\vec{\delta}]\!])$ // a private Boolean
    $[\![\vec{\delta}]\!] := [\![\vec{\delta}']\!]$
    $[\![\vec{S}']\!] \leftarrow ([\![\vec{\delta}]\!] \leq D) :\in$ **kind**$(\mathsf{true}, \mathsf{false})$
    $[\![D']\!] \leftarrow \mathsf{min}(\mathsf{choose}([\![\vec{S}]\!], [\![\vec{\infty}]\!], [\![\vec{\delta}]\!] + [\![\vec{r}]\!]))$
    $[\![\vec{S}]\!] := \mathsf{choose}([\![B]\!], [\![\vec{S}]\!], [\![\vec{S}']\!])$
    $[\![D]\!] := \mathsf{choose}([\![B]\!], [\![D]\!], [\![D']\!])$
  until $\mathsf{declassify}(\bigvee \neg [\![\vec{S}]\!])$
  return $\delta(\cdot)$

The privacy-preserving radius-stepping protocol is presented in Algorithm 15. In detail, the algorithm works as follows: The data input is weighted, undirected graph $G = (V, E)$, which is presented in an adjacency matrix, source vertex s, and the value of the radius for every vertex in the graph, given as a function r: $V \rightarrow \mathbb{R}^+$. The algorithm generally has the same basic structure as the Radius-Stepping algorithm. Hence, we represent the algorithm using vectors to let the algorithm perform the calculation in single instruction multiple data with privacy-preserving. The Radius-Stepping algorithm has the same basic

structure as the Δ-Stepping algorithm. Both algorithms are a hybrid of the Bellman-Ford algorithm and Dijkstra's algorithm, as we explained in Sec 2.4.1.

The set S (Algorithm 3) stored as a vector of private Boolean $[\![\vec{S}]\!]$, and the current mapping δ as a vector of private integer $[\![\vec{\delta}]\!]$. The algorithm starts by initializing the Boolean vector $[\![\vec{S}]\!]$ with false-value, and the vector of graph distance $[\![\vec{\delta}]\!]$ with weights of the source vector s with all vertices $v \in V$. As well as, the first element in vector $[\![\vec{S}]\!]$ and $[\![\vec{\delta}]\!]$ is the source vertex s by true-value and by 0-value, respectively. The operations in the two **forall**-loop are performed in parallel.

The algorithm has only one **repeat**-loop, starting by finding the shortest path for one step by Algorithm 16, until settling all vertices with a radius less than D. The Bellman-Ford-step algorithm starts by vectorizing the elements into the vectors for $[\![\vec{\delta}]\!]$ and $[\![\vec{S}]\!]$ into the row and column vectors. Furthermore, the algorithm updates $[\![\vec{M}]\!]$ by finding the minimum weights of elements in $[\![\overrightarrow{\delta_R}]\!]$ and the summation of the elements in $[\![\overrightarrow{\delta_C}]\!]$ with edges' weights of the graph $[\![\mathbf{G}^{\mathrm{T}}]\!]$. The updating of $[\![\overrightarrow{\delta_B}]\!]$ is either getting the elements of $[\![\vec{M}]\!]$ or $[\![\overrightarrow{\delta_R}]\!]$, which is based on satisfying the three conditions of bounding the number of sub-steps $[\![\vec{c}]\!]$ as presented in Algorithm 16.

The operations of the Bellman-Ford-step algorithm are organized to be performed by the SIMD framework, which decreases the round complexity of performing the Bellman-Ford sub-steps. The vector $[\![\overrightarrow{\delta_B}]\!]$ consists of n-block while each block has also n-element (same number of the vertices in $[\![\mathbf{G}]\!]$). The minimum value is obtained by finding the minimum in all elements in a block separately, then block by block. The minimum values will be stored in $[\![\vec{\delta}']\!]$, where the size is n, and the minimum value is the shortest path for each vertex in the graph. To keep tracing over all vertices in the graph, apply the remaining operations in Algorithm 15.

Algorithm 16: Bellman-Ford-Step

Data: Number of vertices n
Data: Adjacency matrix $[\![\mathbf{G}]\!] \in \mathbb{Z}^{n\times n}$
Data: Current values of $[\![\vec{S}]\!]$, $[\![\vec{\delta}]\!]$, $[\![D]\!]$ in Alg. 15
Result: The updated distance $[\![\vec{\delta}']\!]$ from s

begin
  forall $u \in \{0,1,\ldots,n-1\}$ **do**
    forall $v \in \{0,1,\ldots,n-1\}$ **do**
      $[\![\mathbf{S}_R[u,v]]\!] \leftarrow [\![S[u]]\!]$
      $[\![\mathbf{S}_C[u,v]]\!] \leftarrow [\![S[v]]\!]$
      $[\![\delta_R[u,v]]\!] \leftarrow [\![\delta[u]]\!]$
      $[\![\delta_C[u,v]]\!] \leftarrow [\![\delta[v]]\!]$
  $[\![\vec{c}]\!] \leftarrow (\neg[\![\mathbf{S}_R]\!])\&(\neg[\![\mathbf{S}_C]\!])\&([\![\delta_C]\!] \leq [\![D]\!])$
  $[\![\vec{M}]\!] \leftarrow \min([\![\delta_R]\!], [\![\delta_C]\!] + [\![\mathbf{G}^{\mathrm{T}}]\!])$
  $[\![\delta_{\mathrm{B}}]\!] \leftarrow \mathsf{choose}([\![\vec{c}]\!], [\![\vec{M}]\!], [\![\delta_R]\!])$
  forall $u \in \{0,\ldots,n-1\}$ **do**
    $[\![\delta'[u]]\!] \leftarrow \min\{[\![\delta_{\mathrm{B}}[u,\star]]\!]\}$
  return $[\![\vec{\delta}']\!]$

In Boolean vector $[\![\vec{B}]\!]$, we store the cases of comparison the vector $[\![\vec{\delta}]\!]$ and $[\![\vec{\delta}']\!]$. The goal is to check whether all vertices are settled or not, the current iteration with the previous one. Later, the delta vector $[\![\vec{\delta}]\!]$ will be replaced by the $[\![\vec{\delta}']\!]$. The vector of visited set $[\![\vec{S}']\!]$ in Algorithm 15 represent the condition in the **repeat**-loop in Algorithm 3 (line 9). The vertices which have not been visited yet, their Boolean case will be stored in the vector $[\![\vec{S}]\!]$ after finding the minimum of $[\![\vec{\delta}]\!]$ + $[\![\vec{r}]\!]$. The last operations in the algorithm are updating the vector of visited set $[\![\vec{S}]\!]$ and the value of D, and the updating is based on the state of the Boolean B by applying choose-operations. The algorithm repeats those operations until all vertices are visited (settled). The Boolean cases of $(\bigvee \neg[\![\vec{S}]\!])$ are declassified to satisfy the condition.

4.5. Privacy-preserving Breadth-first search protocol

This section presents a privacy-preserving single-source shortest distance protocol that can be applied to unweighted and weighted graphs. The breadth-first search algorithm is a basement in constructing the protocols. It is a good fit for computing the shortest distances in weighted and unweighted graphs, which is an undirected connected graph. The algorithm and its operations have been designed to run smoothly on ABB using the cryptographic protocol. The sensitivity of the private data in SMC protocols requiring several round-trips between the computing parties for each operation affects the implementation. It is possible to compute the shortest distance on a weighted graph by applying fewer operations than it is to compute the shortest distance on an unweighted graph. We, therefore, present a version of the algorithm that computes the shortest distances faster in the weighted graphs. In the first case, we present a privacy-preserving SSSD algorithm for unweighted graphs, called Unweighted Breadth-First Search (UBFS), while in the second case, we present a privacy-preserving SSSD in weighted graphs, called Weighted Breadth-First Search (WBFS). In parallel, the algorithms use the SIMD framework and are based on the BFS algorithm.

4.5.1. UBFS protocol for unweighted graph

The breadth-first search algorithm relaxes the edges with the same starting vertex s in parallel. The algorithm relaxes the edges of the frontier vertices with each iteration. The graph is represented densely. In the adjacency matrix, weights are assigned to edges with size $n \times n$, where the weight "∞" indicates the lack of a given edge. We use an adjacency matrix to fit our proposed algorithm as a data structure. However, our benchmarks use different graphs on various network environments. The implementation of the Breadth-First search for finding the shortest distances in an unweighted graph that preserves privacy is presented in Algorithm 17.

In detail, the algorithm works as follows: the input data is a private unweighted undirected graph $G = (V,E)$ represented in an adjacency matrix, and the second input is the source vertex. The return value is the distance $[\![\vec{\delta}]\!]$, which is a private vector with size n. The adjacency matrix's data structure was vectorized to make it a good fit for our proposed parallel technique. Two private vectors of size n^2 are initialized to obtain the edges' weights. The rows of the matrix are represented by the first vector, $[\![\overrightarrow{\delta r}]\!]$, and the columns by the second vector, $[\![\overrightarrow{\delta c}]\!]$. The initial values are, storing the first row in the adjacency matrix $[\![\mathbf{G}[s,*]]\!]$ into distance vector $[\![\vec{\delta}]\!]$, and source vertex. It is worth noting that the adjacency matrix has already been vectorized into a $[\![\vec{e}]\!]$ vector of size n^2.

The two vectors $[\![\overrightarrow{\delta r}]\!]$ and $[\![\overrightarrow{\delta c}]\!]$ allow traversing over vertices to relax their edges using sets. As a result, the **foreach**-loops (in Algorithm 4) are not required. This new algorithmic structure minimizes the algorithm's round complexity. The initial values of weight sets are given in parallel, with the vector $[\![\overrightarrow{\delta r}]\!]$ containing the elements of the u-row and the vector $[\![\overrightarrow{\delta c}]\!]$ containing the elements of the v-column, $u, v \in V$. Getting the indices $[\![\overrightarrow{\delta r}]\!]$ happens only once, out of the **repeat**-loop, while getting the indices of $[\![\overrightarrow{\delta c}]\!]$ must take place at each iteration inside the **repeat**-loop for updates. The algorithm consists of a single **repeat**-loop that begins by collecting the edges' weights of columns $[\![\overrightarrow{\delta c}]\!]$ in SIMD parallel.

Algorithm 17: Privacy-Preserving UBFS for unweighted graph

Data: Adjacency matrix $[\![\mathbf{G}]\!] \in \mathbb{Z}^{n \times n}$
Data: Source vertex s
Result: The graph distance $\delta(\cdot)$ from s

```
1  begin
2      [[δ⃗]] ← [[G[s, ∗]]]
3      δ[s] ← 0
4      forall u ∈ {0, 1, ..., n^2 − 1} do
5          [[δ_r[u]]] ← [[δ⃗]]
6      repeat
7          forall v ∈ {0, 1, ..., n^2 − 1} do
8              [[δ_c[v]]] ← [[δ⃗]]
9          [[C⃗]] ← ([[δ⃗_c]] ≠ ∞)
10         [[M⃗]] ← ([[δ⃗_c]] + [[e⃗]])
11         [[δ⃗′]] := choose([[C⃗]], [[M⃗]], [[δ⃗_r]])
12         [[δ⃗]] ← min([[δ⃗′]], n)
13     until declassify(all([[δ]] ≠ ∞))
14     return [[δ(·)]]
```

Regardless of whether there is an edge or not (denoted by "∞"), all elements of column vector $[\![\overrightarrow{\delta c}]\!]$ will be computed. To distinguish this, the condition $[\![\vec{C}]\!]$ will label the indices of existing edges with Boolean values. If its index is used in computation, the elements in the vector are true. This index will be disregarded if the value is false. The Boolean vector has a size of n^2, the same as the adjacency matrix $[\![\mathbf{G}]\!]$. This adjacency matrix may be readily vectorized, and the SIMD technology can be easily applied to the algorithm. The elements of the columns vector $[\![\overrightarrow{\delta c}]\!]$ and the elements of $[\![\vec{e}]\!]$ will then be summed and put in the vector $[\![\vec{M}]\!]$ in the next step. Because all elements are summed in a single parallel instruction, the technique uses the SIMD approach.

UBFS's algorithm will implement the vectorized choose-operation from Sharemind's protocol set. The outcome of the choose-operation is stored in the $[\![\vec{\delta}']\!]$. Among the values stored in the vector, $[\![\vec{\delta}']\!]$ are the current shortest distances $\delta(\cdot)$ from source vertex s to all vertices in the given graph.

The final step is to obtain these minimum values (shortest distances) for each vertex. The min-operation has two arguments: a vector of private values and a public integer n. It

replaces each input vector's n elements with a single value, the cheapest of these n values. In other words, vector $[\![\delta']\!]$ has n-block, min-operation finds the cheapest weights for each block, which is for a vertex. Thus, repeat all operations until all vertices are settled. All of the UBFS algorithm's operations can be applied to vectors of values stored in the ABB. The operation's parameters (which are private) must all be the same length. The SIMD method is satisfied by this aspect of the operations.

4.5.2. WBFS protocol for weighted graph

The sequential breadth-first search algorithm's feature can be suitable for creating a parallel version of the BFS on top of SMC protocols. The data structure of the frontier stores the vertices from the source vertex s simultaneously. This function can be parallelized, reducing the quantity of layer traversal. As a result, the shortest path can be computed in a weighted graph with fewer operations than in an unweighted graph. This section shows another variation of BFS's privacy-preserving shortest path algorithm for a weighted graph with fewer operations. This has the effect of shortening the algorithm's execution time. In Algorithm 18, the privacy-preserving BFS shortest path for the weighted graph is shown.

Algorithm 18: Privacy-preserving WBFS for weighted graph

Data: Adjacency matrix $[\![\mathbf{G}]\!] \in \mathbb{Z}^{n\times n}$
Data: Source vertex s
Result: The graph distance $\delta(\cdot)$ from s

begin
  $[\![\vec{\delta}]\!] \leftarrow [\![\mathbf{G}[s,*]]\!]$
  repeat
    forall $u \in \{0,1,...,n^2-1\}$ **do**
      $[\![\delta_r[u]]\!] \leftarrow [\![\vec{\delta}]\!]$
    $[\![\vec{\delta}']\!] \leftarrow \mathbf{min}((([\![\overrightarrow{\delta_r}]\!] + [\![e(v,u)]\!]), n)$
    $[\![\vec{D}]\!] := [\![\vec{\delta}]\!]$
    $[\![\vec{\delta}]\!] := [\![\vec{\delta}']\!]$
  until **declassify**(**all**($[\![\vec{\delta}']\!] = [\![\vec{D}]\!]$))
  return $[\![\delta(\cdot)]\!]$

The data input is source vertex s, and the weights of their edges and vertices, stored in the private adjacency matrix, are also input to the algorithm. The algorithm's output is the shortest path from the source vertex s to all other vertices in the given graph $[\![\mathbf{G}]\!]$. The algorithm's initial step is to insert the first row of the given graph $[\![\mathbf{G}]\!]$ (with source vertex s) into the shortest distances vector $[\![\vec{\delta}]\!]$, which will be updated during the algorithm's execution until all vertices have been processed. The algorithm includes only one repeat-loop, which begins by allocating all rows of the provided graph $[\![\mathbf{G}]\!]$ to the $[\![\vec{\delta}_r]\!]$ rows vector. In our algorithms, we utilize the **forall**-loop to signal that all assign-operation in that loop should be executed in parallel.

Later, the adjacency matrix of the given graph $[\![\mathbf{G}]\!]$ has already been vectorized into vector $[\![\vec{e}]\!]$, which we have vectorized column by column. The algorithm's core operation is the summing of the two vectors $[\![\overrightarrow{\delta r}]\!]$ and $[\![\vec{e}]\!]$, followed by the min-operation to obtain

the minimal distance for each vertex. The algorithm's final mission is to swap the vectors used to make the updates. The goal is to ensure that all vertices have been handled. The end-of-loop equality check returns a vector of Boolean values. The all-operation does not reveal the values of $[\![\vec{\delta}']\!]$ and $[\![\vec{D}]\!]$. The number of iterations in this algorithm is unknown. It is based on how many related edges there are. The algorithm performs efficiently when the number of edges is close to the maximum possible, such as in a dense graph.

4.6. Performance Analysis

Two kinds of communication-related complexities exist in secure multiparty computation applications that might constitute bottlenecks in deployments among computation parties. This is why secure multiparty computation applications have such high network latency. This section discusses the performance analysis of our proposed protocols in terms of these complexities. The number of round trips that have to be made among the SMC computation parties is called the *round complexity* of the algorithms. The second complexity is the *communication complexity* for an algorithm which refers to the number of bits that computation parties may transport. This complexity is based on the algorithm's structure and the number of times the algorithm will be iterated to complete the computation. Let g denote the number of given graphs—or the number of components in an unconnected graph $[\![\mathbf{G}]\!]$, while n is the number of the vertices, and m is the number of the edges in the graph.

4.6.1. Round complexity

We start with the Privacy-preserving Radius-stepping protocol. The radius r determines the number of steps and sub-steps in the calculation. Thereby, the Round complexity is based on the radius r. The number of iterations is based on the number of visited vertices. The protocol of radius-stepping has only one **repeat**-loop that indicates the logarithmic complexities of the algorithm. Algorithm 15 has a subroutine, and the round complexity of the subroutine—Bellman-Ford-step (Algorithm 16)—is constant. There are three cases of the radii r. In the case that $r(v) = 0$, the round complexity of the algorithm is $O(n + \log m)$. If $r(v) = \infty$, the round complexity is $O(\log n)$. As well as, the round complexity is $O(\log n + \log m)$ if the $r(v) = \delta$.

We suppose m is somewhere between n and $O(n^2)$, which means $n + m$ is $O(m)$ and $\log m$ is $O(\log n)$. In the privacy-preserving Bellman-Ford algorithm (Algorithm 11), the part before the main loop requires $O(\log n)$ rounds. Indeed, both prepareRead-statements have this complexity, while the complexity of GenIndicesVector-function is dominated by the sorting of n private values. One iteration of the main loop of the Bellman-Ford algorithm requires $O(\log n)$ rounds, with prefixMin2-function (Version 1) being the only operation working in non-constant rounds. The number of iterations is $(n-1)$, hence the total round complexity is $O(n \log n)$. However, the number of iterations represents the worst-case scenario, which must be used only if there is no further information. We can reduce the number of iteration cycles by assuming that the shortest path we are concerned which consists of at most $k < n$ edges, where the number of iterations can be reduced. The round complexity, in this case, is $O(k \log n)$.

In the privacy-preserving Dijkstra's protocol (Algorithm 7) where input data is an adjacency matrix $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{n \times n}$, number of edges is less than n^2. The main **for**-loop requires $O(\log n)$ rounds for each iteration, minLs is the only operation working in non-constant rounds. The entire loop thus requires $O(n \log n)$ rounds. The permutation

which occurs before the main loop requires a constant number of rounds. The last part in the algorithm, which is the private read, has $O(\log n)$.

The running in the nDijkstra's algorithm is g time in Dijkstra's algorithm; however, the running in parallel, nDijkstra's algorithm carries g graphs simultaneously. Hence, the round complexity in the implementation of nDijkstra's protocol (Algorithm 9) is $O(n^2)$. Furthermore, the round complexity of the permutation part is $O(n)$, where $g = n$. Moreover, the private read also requires $O(n \log n)$.

In privacy-preserving BFS protocols, Round complexity is based on how often the algorithm has to iterate until the result is obtained. Based on empirical tests in Sec 8.2.5, round complexity is based on the number of vertices n and the number of edges m. If two graphs have the same number of vertices, one dense and the other sparse, then the round complexity of the sparse graph's algorithm is more than running on a dense graph. In other words, the best case of the algorithm is on a dense graph, while the worst case is on a sparse graph. The round complexity for both versions of BFS is $O(\log n)$. In the UBFS version that uses the dense graph, the round complexity is constant $O(1)$. The round complexities of privacy-preserving SSSD protocols are presented in Table 1.

Table 1: Round and communication complexities for privacy-preserving SSSD protocols.

Protocol		**Complexities**	
		Round	Communication
Radius-Stepping	r = 0	$O(n+\log m)$	$O(n^2 \log n)$
	r =∞	$O(\log n)$	$O(n^2 \log n)$
	r=δ	$O(\log n+\log m)$	$O(n^2 \log n)$
BFS	WBFS	$O(\log n)$	$O(n^2 \log n)$
	UBFS	$O(\log n)$	$O(n^2 \log n)$
Bellman-Ford V1	Pre-loop	$O(\log n)$	$O(m \log n)$
	Loop	$O(n \log n)$	$O(mn)$
Bellman-Ford V2	Pre-loop	$O(\log n)$	$O(m \log n)$
	Loop	$O(n \log n)$	$O(mn)$
Dijkstra	Perm.	$O(1)$	$O(n^2)$
	Loop	$O(n \log n)$	$O(n^2)$
	PerfR.	$O(\log n)$	$O(n \log n)$
nDijkstra	Perm.	$O(n)$	$O(n^3)$
	Loop	$O(n^2)$	$O(n^3)$
	PerfR.	$O(n \log n)$	$O(n^2 \log n)$

4.6.2. Communication complexity

Bellman-Ford and Dijkstra's protocols have linearithmic round complexities, while radius-stepping and breadth-first search protocols have logarithmic rounds.

We start by discussing the communication complexities of the linearithmic protocols based on the discussions in Sec 4.6.1. The steps of the Bellman-Ford algorithm before the main loop of the algorithm require $O(m \log n)$ of communication. In the main **for**-loop requires $O(m)$ of communication for each iteration. The number of iterations in the main loop is $n-1$; hence, the total communication requires $O(mn)$. In the case that shortest paths consist of at most $k < n$ edges, the total communication is $O(m(k+\log n))$.

In privacy-preserving Dijkstra's protocol, shuffling rows and columns in the permutation part requires $O(n^2)$ of communication. The main **for**-loop requires $O(n^2)$ of communication. Applying Laud's protocols for private read thus requires $O(n \log n)$ of communication.

The communication of privacy-preserving Dijkstra's protocol for multiple graphs (nDijkstra), where $g = n$ as follows: The shuffling of rows and columns requires $O(n^3)$ of communication. The part for finding the shortest path for several graphs has two loops. The inner **for**-loop requires $O(n^2)$ of communication, while each iteration in outer **for**-loop also require $O(n^2)$ of communication. The entire loop thus requires $O(n^3)$ of communication.

In the radius-stepping protocol with logarithmic round complexities based on radii, the adjacency matrix is entirely vectorized into a large vector, where the size is $n \times n$. Each iteration of the **repeat**-loop requires $O(n^2)$ of communication. The entire loop thus requires $O(n^2 \log n)$ of communication.

The versions of the privacy-preserving BFS protocol have an algorithmic structure similar to the Radius-stepping algorithm. Hence, we assume the number of iterations is p. The vectorization is done into a large vector with size $n \times n$ that will handle the operations. Thus, one iteration requires $O(n^2)$ of communication. Then, the entire **repeat**-loop requires $O(p \cdot n^2)$ of communication. The communication complexities of privacy-preserving SSSD protocols with their parts are presented in Table 1.

4.7. Security and privacy of protocols

4.7.1. Security of protocols built on top of ABB

As shown above, this chapter presents four new protocols for privacy-preserving SSSD with their related algorithms. The input data of the protocols are privately structured either in an adjacency matrix or in three private vectors. The proposed protocol is built on top of a universally composable ABB. If there were no declassification operations in the protocol, declassification means converting data type of private data from private to public. Then, its composition with an SMC protocol set that is a secure implementation of the ABB would inherit that protocol set's security and privacy properties [112].

The proposed SSSD protocols contain declassification operations. We leak the number of iterations in Algorithm 15, Algorithm 17, and Algorithm 18. Also, we leak the identity of source vertices s and start-point edges u in versions of Dijkstra's protocol, Algorithm 7 and Algorithm 9—the vertices s and u are masked as a result of permutation. Nevertheless, we can state the following security theorem.

Theorem 1. *Suppose that our SMC protocol set implements an ABB for k computing parties that is secure against an active [resp. passive] adversary that corrupts at most t computing parties. Then, an active [resp. passive] adversary that runs in parallel with k parties executing the SSSD protocol and this SMC protocol set to implement private computations, corrupting at most t parties, will not learn anything about the inputs to the protocol besides the number of vertices and edges of the graph, and the starting vertex s (for Bellman's and Dijkstra's protocols). For some protocols (the BFS kind and radius-stepping), the adversary also learns the number of iterations.*

Proof. We need to construct a simulator that takes the adversary's view in the ideal world and returns the adversary's view in the real world. A copy of the ABB's ideal functionality

is running inside the simulator. The adversary would only receive n and s in an ideal world. In the real world, if several computation parties are corrupted, the adversary sees several things:

- the inputs, which are the number of the vertices n and source vertices s;
- the *handles* to the private values, stored in the variables marked with $[\![\cdot]\!]$ in the algorithm. Regardless, it is elements, vectors, or adjacent matrix.
- the declassified values. It can be an integer number, element, or vector.
- if the adversary is active: the reactions of the ABB to the attempts of the corrupted parties to deviate from an SSSD protocol.

The composition theorem of the universal composability framework takes care of the values the real-world adversary observes (through corrupted parties) throughout the execution of the SMC protocols implementing the ABB and the implications of any deviation from these protocols by the corrupted parties.

Ideal-world adversary outputs the numbers n and s to the simulator. Since the handles' values (as opposed to the values they are pointing to within the ABB) are public, the simulator can compute those values. The simulator picks up on the commands sent to the ABB by all k parties. The simulator learns whether a corrupted party deviates from the SSSD protocol and its subroutines, as well as the reaction of the ABB.

In the SSSD protocols, the declassification values are either for the number of iterations or the source vertices. In order to simulate the declassified values for the source vertices, the simulator generates a random permutation of the numbers $0, 1, \ldots, n-1$. Later, release them either as a single value at the declassification in line 7 in Algorithm 7, or release the single values one by one at declassification in line 15 in Algorithm 7. Also, release them as vectors, vector by vector, such as at declassification in line 16 in Algorithm 9.

In the algorithms where we declassified the source vertices, we randomly permuted the vertices at the begging of the function as a prerequisite computation, keeping the permutation itself private, hence out of the adversary's view. Thus the order in which the vertices are relaxed (i.e., added to the set M) in the main loop is random.

In the algorithm where we declassified the number of the iteration, the algorithms' outcomes are not directly deduced from the public parameters of the graph. By experimentally determining a reasonable upper bound of the iterations and always doing at least this number of iterations in such algorithms, it is likely to reduce this leakage while still obtaining the advantages of iterating only as long as something changes.

This number of iterations depends on many parameters, including the graph's structure (and perhaps radii in the radius-stepping algorithm) and the number of edges in graphs such as the BFS algorithm. As a particular case, the UBFS algorithm with dense graph, the algorithm has constant round complexity, and the computation is ended by only one iteration, the result documented in Sec 8.2.5. Thus, the leakage of the iterations' number is useless and will not lead the adversary to any knowledge about computation.

4.7.2. Detailed security proof for privacy-preserving Bellman-Ford

The pure computation of finding the SSSD on top of SMC is privacy-preserving because no private data has been declassified. However, we use declassification, which is used only for permuted private data. The most important feature of the UC security definition

is composability. This feature allows developers to create large privacy-preserving applications, enabling researchers to add new operations to ABB. The basic ABB operations are arithmetical and logical, and comparisons and permutations are discussed in Sec 3.2.3. Other researchers have extended an ABB to use new operations such as sorting [158, 31], and private read/write discussed in Sec 3.2.1, which operates mainly in some research contributions of this thesis.

We assume the existence of a secure implementation Ξ of an ideal functionality $\mathcal{G}_{ABB}$ (for n computation parties) that supports the following operations:

- Arithmetic. The arithmetic operations, addition and multiplication, take two handles to integers and return a handle to an integer because both integers are private.
- Comparison. The comparison operations, "less than" ($<$), "less or equal than" ($\leq$), "equal to" ($==$), "not equal to" ($\neq$), etc., take two handles to integers (or float) and returns a handle to a Boolean value. In the case of operations "equal to" ($==$) and "not equal to" ($\neq$), the data input can also be Boolean.
- Logic. We apply the logic operation for private vectors using choose ($[\![\vec{a}]\!]$, $[\![\vec{x}]\!]$, $[\![\vec{y}]\!]$). So, $[\![\vec{a}]\!]$, $[\![\vec{x}]\!]$, and $[\![\vec{y}]\!]$ are all vectors of handles. The operation returns a vector of handles, either $[\![\vec{x}]\!]$ or $[\![\vec{y}]\!]$. These new handles point to elements of $[\![\vec{x}]\!]$ and $[\![\vec{y}]\!]$, depending on the values that the elements of $[\![\vec{a}]\!]$ point to. The data type of the vectors $[\![\vec{x}]\!]$ and $[\![\vec{y}]\!]$ can be integers, float, and Boolean accessed only through handles, while the results of the choose-operation are only Boolean, which also through handles.
- Declassification. This operation removes the privacy of private data and makes it public. The input data of this operation is private integers $[\![\vec{y}]\!]$—a vector of handles. The output is a public vector of integers x.
- Sorting. The input value of this operation is private integers located into vector $[\![\vec{x}]\!]$. The operation returns sorted private integers located in private vector $[\![\vec{y}]\!]$. The sorted vector is a vector of handles to values that represents the result of sorting the vector of values pointed to by handles in $[\![\vec{x}]\!]$.
- Random permutation. This operation is for randomly generating a private permutation of n elements. The input is a public integer number n and returns a private permutation $[\![\sigma]\!]$ through the handle. We apply the permutation for a private vector $[\![\vec{v}]\!]$ using apply ($[\![\sigma]\!]$,$[\![\vec{v}]\!]$). It results in a vector $[\![\vec{w}]\!]$ through handle, where $w_i = v_{\sigma(i)}$ for all $i \in \{1,\ldots,n\}$. Furthermore, applying the inverse of σ to $\vec{v}$ is also possible using unApply ($[\![\sigma]\!]$,$[\![\vec{v}]\!]$) through handle.
- prepareRead and performRead. This operation is for reading from a vector by a private index. We perform the reading by applying performRead-operation, which has two arguments, integer vector $[\![\vec{v}]\!]$ of length n, and the second argument comes from applying prepareRead(n, $[\![\vec{y}]\!]$), where $[\![\vec{y}]\!]$ is indices of the m elements that we want to read. This operation returns private vector $[\![\vec{w}]\!]$ of length m through the handle. The length of $[\![\vec{v}]\!]$ is equal to the first argument to prepareRead; the second argument of performRead is the output of prepareRead. However, if these conditions are not satisfied, then $\mathcal{F}_{ABB}$ can behave arbitrarily. The return value of performRead-operation is a handle to a private vector of integers. The vector of the return value is $[\![\vec{w}]\!]$, and each element of the vector is $w_i = v_{y_i}$.

We use the composition theory [48] to add a new operation, SSSD, to the set of supported ABB. The implementation of SSSD on top of ABB has been described in four main protocols. These SSSD protocols are as follows:

- Privacy-preserving Dijkstra protocol (Algorithm 7).
- Privacy-preserving Bellman-ford protocol (Algorithm 11).
- Privacy-preserving Radius-stepping protocol (Algorithm 15).
- Privacy-preserving Breadth-first search protocol (Algorithm 18).

Those protocols have different properties in terms of the graphs' kinds, relaxing procedures, and the structure of private input data. We give the example of detailed security proof for the Bellman-Ford protocol because it uses the most interesting privacy-preserving computation subroutines (perfixMin2 presented in Algorithms 12 and 14) out of these four proposed protocols. The new operation SSSD takes the handles pointing to the edge locations and their weights (as in the definition of Bellman-Ford) and returns the handles pointing to the distances of vertices from the source vertex s. We are now going to present a secure implementation of an ABB $\mathcal{F}_{ABB}$ (for n parties) that supports all the same operations as $\mathcal{G}_{ABB}$, but additionally supports the SSSD operation. The details of the inputs and outputs of the SSSD operation and is executed as follows:

- On input from all honest computing parties, $\mathcal{P}_i$ (for $i \in \{1, \ldots, n\}$) does not compute anything themselves. They only give inputs to and receive outputs from either the machines $\mathcal{M}_1$,...,$\mathcal{M}_n$, or the ideal functionality $\mathcal{F}_{ABB}$. In the SSSD operation by Bellman-ford, as supported by $\mathcal{F}_{ABB}$, when all honest parties invoke the SSSD operation with arguments $[\![\vec{S}]\!]$, $[\![\vec{T}]\!]$, $[\![\vec{W}]\!]$, s, N, where $[\![\vec{S}]\!]$ and $[\![\vec{T}]\!]$ are the edges' locations, $[\![\vec{W}]\!]$ is their weights, and s is the source vertex, and N is the number of vertices. Then $\mathcal{F}_{ABB}$ computes the SSSD and returns its result as a vector of handles to each of the computing parties, where the handles point to the shortest distances from s. The length of the returned vector is equal to N. The ideal functionality $\mathcal{F}_{ABB}$ keeps the adversary $\mathcal{A}$ updated on the computations, see Sec 2.1.3.
- On output from all honest computing parties, each the computation party $\mathcal{P}_i$ sends its results SSSD to the corresponding party $\mathcal{P}'$. The result is a vector of the shortest distances for all vertices $[\![\vec{T}]\!]$ from single source s.

Theorem 2. *There exists a protocol Π^{SSSD} that perfectly securely implements $\mathcal{F}_{ABB}$ in the $\mathcal{G}_{ABB}$-hybrid model.*

Proof. The protocol Π^{SSSD} is the composition of machines $\mathcal{M}_1, \ldots, \mathcal{M}_n$. Whenever $\mathcal{M}_1, \ldots, \mathcal{M}_n$ receive a command that is not SSSD, then they will forward that command over to $\mathcal{G}_{ABB}$, whenever $\mathcal{M}_1, \ldots, \mathcal{M}_n$ receive an SSSD command, then they will follow Algorithm 11 (with subroutines in Algorithms 12 and 13). This means that they invoke the operations of $\mathcal{G}_{ABB}$ in the order specified in Algorithm 11-13.

In order to show that the composition of $\mathcal{M}_1, \ldots, \mathcal{M}_n$ and $\mathcal{G}_{ABB}$ is at least as secure as $\mathcal{F}_{ABB}$, we will present a simulator that translates any actions of an adversary attacking the real system to actions against $\mathcal{F}_{ABB}$, such that the parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ cannot see the difference.

The simulator *Sim* is placed between ideal functionality $\mathcal{F}_{ABB}$, and a real adversary $\mathcal{A}$ that expects to be connected to Turning machines $\mathcal{M}_1, \ldots, \mathcal{M}_n$ and $\mathcal{G}_{ABB}$. The simulator gets from $\mathcal{F}_{ABB}$ the commands that computation parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$ enter, and translates

them either to messages $\mathcal{G}_{ABB}$ –> $\mathcal{A}$ (if the command was not SSSD), or to an SSSD command from each of $\mathcal{M}_1, \ldots, \mathcal{M}_n$ and a sequence of commands from $\mathcal{G}_{ABB}$ (if the command from $\mathcal{P}_1, \ldots, \mathcal{P}_n$ were SSSD).

The simulator effectively has to run a copy of $\mathcal{G}_{ABB}$ inside it. Each copy of $\mathcal{G}_{ABB}$ on Turning machine $\mathcal{M}_i$ of the computation party $\mathcal{P}_i$. The aim is to check that there is no corrupting party by making adversary $\mathcal{A}$ would think it was communicating with the components of protocol Π^{SSSD}. The simulator does not know the sensitive input values that go into $\mathcal{G}_{ABB}$ because $\mathcal{F}_{ABB}$ does not forward them to the simulator. Hence the simulator executes its internal $\mathcal{G}_{ABB}$ with arbitrary values. However, the simulator learns the declassified values if $\mathcal{P}_1, \ldots, \mathcal{P}_n$ ask $\mathcal{F}_{ABB}$ to declassify something. At this point, the simulator can fill in the correcting values and forward them to adversary $\mathcal{A}$ over the $\mathcal{G}_{ABB}$ –> $\mathcal{A}$ connection.

The declassification command allows parties to get the actual data using a handle. There is a declassification command in step 5 of Algorithm 13. It is important to note that $\mathcal{F}_{ABB}$ will only correctly react if it receives the same instructions from all computing parties in the same $\mathcal{G}_{ABB}$. A sorted vector $[\![\vec{v}]\!]$ is provided as input to the simulator, which then invokes $\mathcal{G}_{ABB}$ for steps 2-5 of Algorithm 13. The simulator first gets the size m of the sorted vector $[\![\vec{v}]\!]$. Then, computes the end-point index vector $[\![\vec{b}]\!]$. Using apply-routine, randomly permute vector $[\![\vec{b}]\!]$ based on σ elements generated by step 4. Public vector $\vec{c}$ is a random 0-1 vector with a given number of entries "1". It handles declassified values, which indicates that a vertex has an edge, but it does not matter that this value has to be private. The declassification produces a Boolean vector with exactly n elements of "true", where m is the length of the vector. As a result, there is no further dependency on $[\![\vec{b}]\!]$. This declassification in Algorithm 13 will not compromise our SSSD Bellman-Ford's privacy.

5. PRIVACY-PRESERVING ALL-PAIRS SHORTEST PATH PROTOCOLS

5.1. Introduction

This chapter presents our proposed protocols in All-pairs shortest distance on top of secure multiparty computation protocols. We first discuss the proposed protocol of privacy-preserving APSD's Johnson with its versions. For benchmarking and evaluation of the proposed privacy-preserving APSD Johnson's protocol, we introduce an implementation of Floyd-warshall and transitive closure of the graph in SIMD, which of on top of SMC protocols.

5.2. Johnson protocol

Dijkstra's algorithm is applied at each vertex via Johnson's APSD algorithm. It will first adjust the weights to copy with the negative-weight edges so that the shortest paths remain unchanged. To accomplish this, a new vertex is added to the graph, with 0-weight edges connecting it to all existing vertices, and the shortest distances $h(v)$ between the new vertex and all existing vertices v are determined. The weight of an edge $w(u,v)$ is then updated to $\tilde{w}(u,v) = w(u,v) + h(u) - h(v)$, resulting in a change in the lengths of all paths from u to v of $h(u) - h(v)$. As a result, the shortest distances in the updated graph, when combined with h, yield the shortest distances in the original graph. The Bellman-Ford algorithm is used to find the shortest distances from an extra vertex.

Algorithm 19: Privacy-preserving Johnson

Data: Number of vertices n and edges m
Data: Sources, targets, and weights of edges $[\![\vec{S}]\!]$, $[\![\vec{T}]\!]$, and $[\![\vec{W}]\!]$
Result: Private distances of all pairs of vertices

begin
- $[\![\vec{S}_Q]\!] \leftarrow [\![\vec{S}]\!]$ @ $[[\![n]\!], \ldots, [\![n]\!]]$
- $[\![\vec{T}_Q]\!] \leftarrow [\![\vec{T}]\!]$ @ $[[\![0]\!], \ldots, [\![n-1]\!]]$
- $[\![\vec{W}_Q]\!] \leftarrow [\![\vec{W}]\!]$ @ $[[\![0]\!], \ldots, [\![0]\!]]$
- $[\![\vec{h}]\!] \leftarrow \mathsf{Bellman\text{-}Ford}(n+1, m+n, n, [\![\vec{S}_Q]\!], [\![\vec{T}_Q]\!], [\![\vec{W}_Q]\!])$
- $[\![\vec{h}_\mathrm{s}]\!] \leftarrow \mathsf{performRead}([\![\vec{h}]\!], \mathsf{prepareRead}(n, [\![\vec{S}]\!]))$
- $[\![\vec{h}_\mathrm{t}]\!] \leftarrow \mathsf{performRead}([\![\vec{h}]\!], \mathsf{prepareRead}(n, [\![\vec{T}]\!]))$
- $[\![\vec{W}']\!] \leftarrow [\![\vec{W}]\!] + [\![\vec{h}_\mathrm{s}]\!] - [\![\vec{h}_\mathrm{t}]\!]$
- $[\![\mathbf{G}]\!] \leftarrow \infty$ `// size of` $\mathbf{G}$ `is` $n \times n$
- $[\![\mathbf{G}]\!] \leftarrow \mathsf{performWrite}([\![\mathbf{G}]\!], [\![\vec{W}']\!], \mathsf{prepareWrite}(n^2, n \cdot [\![\vec{S}]\!] + [\![\vec{T}]\!]))$
- **for** $i = 0$ **to** $n-1$ **do**
  - $[\![\mathbf{D}'[i,\star]]\!] \leftarrow \mathsf{Dijkstra}(n, i, [\![\mathbf{G}]\!])$
  - **forall** $j \in \{0, \ldots, n-1\}$ **do**
    - $[\![\mathbf{D}[i,j]]\!] \leftarrow [\![\mathbf{D}'[i,j]]\!] - [\![\vec{h}[i]]\!] + [\![\vec{h}[j]]\!]$
- **return** $[\![\mathbf{D}]\!]$

Johnson's algorithm consists of three parts, Bellman-Ford, Dijkstra, and rewriting.

Having a privacy-preserving version of both algorithms, Bellman-Ford and Dijkstra, will open the door for creating a privacy-preserving version of the APSD Johnson protocol. However, re-weighing edges is still a challenge, as the locations of the edges are private. Laud protocol solves this problem by applying performWrite-function. We integrated the Bellman-Ford and Dijkstra algorithms as in Johnson's approach since we had privacy-preserving implementations of each. Like our Bellman-Ford implementation, our Johnson protocol implementation expects the graph to be sparsely represented. Because our implementation of Dijkstra's method requires a dense graph representation, we must do the conversion in the middle. Algorithm 19 is our privacy-preserving implementation of Johnson's protocol.

The steps of Algorithm 19 closely follow the description above. We augment the original representation of the graph with an extra vertex and edges, the index of the added vertex being n, as an example $[\![\vec{S}_Q]\!] \leftarrow [\![\vec{S}]\!] \mathbin{@} [[\![n]\!], \ldots, [\![n]\!]]$, where @ represent the augment operation. After finding the shortest paths from this vertex to all other vertices, we use the parallel reading subroutines to find the updates for edge lengths. We then use the parallel writing subroutines to convert from a sparse to a dense graph representation. We store the all-pairs shortest distances in the modified graph inside the matrix $[\![\mathbf{D}']\!]$, and then remove the updates to the lengths of edges (and paths).

We see that the last loop in Algorithm 19 can be performed in parallel, as there are no data dependencies between the iterations. At this point, we can use the nDijkstra procedure, discussed in Sec 4.2. We use this procedure to fill in the entire matrix $[\![\mathbf{D}']\!]$ at once and then compute $[\![\mathbf{D}]\!]$ in a SIMD manner. We call the algorithm in Algorithm 19 the Version 1 of the implementation of Johnson's protocol, while the version that uses the nDijkstra procedure (in Algorithm 9) is called Version 2. We present benchmark results for both versions. The main difference in Johnson versions is finding APSD using versions of privacy-preserving Dijkstra protocols; no change has occurred in the Bellman-Ford part, and rewriting remains.

5.3. Floyd-Warshall algorithm

The Floyd-Warshall is a greedy algorithm to solve the APSD problem. It compares all possible paths via the graph $[\![\mathbf{G}]\!] = (V, E)$ between each pair of vertices i and $j \in V$. It has three for-loop to find the shortest path for all pairs in a weighted graph.

Algorithm 20: Privacy-preserving Floyd-Warshall

Data: Number of vertices n
Data: Lengths of edges $[\![\mathbf{G}]\!] \in (\mathbb{Z} \cup \{\infty\})^{n \times n}$
Result: Private distances of all pairs of vertices

```
1 begin
2     [[D]] ← [[G]]
3     for k = 0 to n − 1 do
4         forall i = 0 to n − 1 do
5             forall j = 0 to n − 1 do
6                 [[T[i, j]]] ← [[D[i, k]]] + [[D[k, j]]]
7         [[D]] ← min([[D]], [[T]])
8     return [[D]]
```

The adjacency matrix has two dimensions, i and j. The third loop is for the intermediate vertices k that the algorithm used to find the shortest path between any two pairs, vertex i and j through intermediate vertex k. To compare our implementation of privacy-preserving Johnson with previous techniques, we constructed the Floyd-Warshall algorithm on top of the Sharemind-based ABB. Our implementation is standard [45, 106], attempting to parallelize as many operations as possible. It uses the dense representation of the graph.

Our implementation of the SIMD-Floyd-Warshall algorithm is presented in Algorithm 20. The input data of the Floyd-Warshall algorithm is a private matrix $[\![\mathbf{G}]\!]$, which contains the weights of the edges. The output of the algorithm is a private matrix $[\![\mathbf{D}]\!]$, which has the lengths of shortest paths among all vertices in V. Although the algorithm has three nested loops, the two inner loops are only used to rearrange the already computed private values and can be performed simultaneously. The actual computations (summation and minimum) are made in a SIMD manner for all entries of $[\![\mathbf{T}]\!]$ resp. $[\![\mathbf{D}]\!]$.

5.4. Transitive closure algorithm

In a given directed graph, the algorithm checks if a path from vertex i to j for all vertex pairs (i, j) means vertex j is reachable. The reachability matrix is called the transitive closure of a given directed graph. As in Johnson's APSD algorithms, we have implemented the transitive closure computation on top of Sharemind-based ABB to investigate another means for computing APSD.

The transitive closure computation presents another trade-off between communication and round complexity. Our implementation of privacy-preserving APSD through transitive closure is presented in Algorithm 21. The only sequentially-run loop of the computation has a logarithmic number of iterations, so this algorithm has low round complexity compared to the Floyd-Warshall algorithm.

Algorithm 21: Transitive Closure

Data: Number of vertices n
Data: Lengths of edges $[\![\mathbf{G}]\!] \in (\mathbb{Z} \cup \{\infty\})^{n \times n}$
Result: Private distances of all pairs of vertices

```
1 begin
2     [[D→]] ← [[G]]
3     for l = 1 to log n do
4         forall i ∈ {0,...,n−1} do
5             forall j ∈ {0,...,n−1} do
6                 forall k ∈ {0,...,n−1} do
7                     [[T_{i,j}[k]]] ← [[D[i,k]]] + [[D[k,j]]]
8                 [[D[i,j]]] ← min([[T→_{i,j}]])    // minimum element of a vector
9     return [[D]]
```

5.5. Performance Analysis

5.5.1. Round complexity

The privacy-preserving Johnson's protocol has around a complexity of $O(n \log n)$, which is the same as the complexity of both the Bellman-Ford and Dijkstra algorithms. Private reading and writing operations between them take just $O(\log n)$ rounds. The Floyd-Warshall algorithm has a round complexity of $O(n)$. In the transitive closure, because the outer loop iterates $\log n$ times, the round complexity of transitive closure is $O(\log^2 n)$. The smallest elements of length vectors n can be found in $O(\log n)$ rounds at each iteration. The round complexities of privacy-preserving APSD protocols are presented in Table 2.

Table 2: Round and communication complexities for privacy-preserving APSD protocols.

Protocol		**Complexities**	
		Round	Communication
Floyd-Warshall		$O(n)$	$O(n^3)$
Transitive Closure		$O(\log^2 n)$	$O(n^3 \log n)$
Johnson	Alg	$O(n \log n)$	$O(n^3)$
	PerformRead/-Write	$O(\log n)$	$O(n^2 \log n)$

5.5.2. Communication complexity

The Communication consumption of Johnson's protocol is dominated by n instances of Dijkstra's algorithm, hence being $O(n^3)$ in total. This is also the asymptotic Communication consumption of the Floyd-Warshall algorithm, but as we can see in Sec 8.3, the constant hidden in the O-notation is less. Johnson's algorithm, on the other hand, is more versatile: if we are not interested in all-pairs shortest distances, but only in shortest distances from $k < n$ vertices, then the Communication consumption of Johnson's algorithm is only $O(mn + n^2(k + \log n))$, with $O(mn)$ being the Communication use of the Bellman-Ford algorithm and $O(n^2 \log n)$ being the Communication use of the private writing into an array of size n^2. The Communication consumption used by the transitive closure is $O(n^3 \log n)$. The communication complexities of privacy-preserving APSD protocols are presented in Table 2.

5.6. Security and privacy of protocols

As we mentioned before, the most critical feature of privacy preservation is that the execution is done on the ABB, and the implementation has no declassification. Following the discussion in Sections 3.2.3 and 4.7, we conclude that APSD protocols 18–20 are all privacy-preserving, as none contain any declassification statements.

6. PRIVACY-PRESERVING MINIMUM SPANNING TREE AND FOREST PROTOCOLS

6.1. Introduction

Besides the shortest path problem, this thesis introduces solutions for minimum spanning trees and forests on privacy preservation, which use big data sets. The chapter starts with presenting the privacy-preserving Prim's MST protocol. Furthermore, we describe the optimized version of Prim's protocol. We benefit from our suggested protocol for finding a minimum spanning tree on top of the SMC protocol to create a new protocol for finding a minimum spanning forest. As well as we present a privacy-preserving MSF protocol, which means a minimum spanning tree n times with two versions of MSF, sequential and parallel. Besides describing the protocols with their related algorithms, this chapter discusses their performance, round and communication complexities, and security and privacy.

6.2. Privacy-preserving Prim's protocol

The privacy-preserving Prim's MST protocol has been constructed based on the general feature of Prim's essential algorithm, as described above. In detail, two main sets of the vertices V in a given private graph $[\![\mathbf{G}]\!]$ should be defined for included vertices in the MST and the other for not yet included ones. In the parallel implementation, the protocol was constructed using SIMD instruction considering the Sharemind protocols set—which is parallel implementation on top of SMC protocols.

Parallel Prim's Algorithm maintains two sets of vertices. The first set M, which in our privacy-preserving implementation is represented as a vector of Boolean $\vec{M}$ of length $|V|$, with the value false meaning that the corresponding vertex is included in the set. This Boolean vector $\vec{M}$ contains the vertices already included in the private MST. For reasons shown later, the set $\vec{M}$ is public.

For the remaining vertices, we record in the vector $[\![\vec{K}]\!]$ the length of the shortest edge that connects them with some vertex in $\vec{M}$. The Algorithm finds the minimum-weight edge between the sets M and $V \backslash M$ at each step of Prim's Algorithm and includes it in the MST, updating the set $\vec{M}$. We record the tree that we created in the parent vector $[\![\vec{P}]\!]$ (of length $|V|$)—for each vertex v, the value $P[v]$ represents a vertex; thus $(P[v], v)$ was added to the tree when v was an element of $V \backslash M$. The privacy-preserving computation of Prim's minimum spanning tree protocol is presented in Algorithm 22. The protocol has three main parts. It starts with a permutation to mask the real identity of the vertices; the second part is finding the MST in privacy-preserving, and the last part is returning the original order of the vertices.

The first part (lines 2-8) hides the vertices' identities using a random, private permutation. By permuting their identities, we can mask the order in which the vertices are added to the set M. In this way, the procedures are similar to [3], and the reasons are the same—to avoid expensive memory accesses dependent on private data in the following steps, this procedure is also used in privacy-preserving Dijkstra in Sec 4.2.

We generate a random private permutation $[\![\sigma]\!]$ of length n to permute the vertices. We use $[\![\sigma]\!]$ to permute all rows and columns of $[\![\mathbf{G}]\!]$; all rows can be permuted simultaneously, and the same goes for columns.

Algorithm 22: Privacy-preserving Prim's MST

Data: Number of vertices n
Data: Starting vertex s
Data: edge weights $[\![\mathbf{G}]\!]$ (a $n \times n$ array)
Result: Minimum Spanning Tree $[\![\vec{T}]\!]$

1 **begin**
2 $\quad [\![\sigma]\!] \leftarrow \mathsf{randPerm}(n)$
3 $\quad$ **forall** $u \in \{0,\ldots,n-1\}$ **do**
4 $\quad\quad [\![\mathbf{G}'[u',\star]]\!] \leftarrow \mathsf{apply}([\![\sigma]\!],[\![\mathbf{G}[u,\star]]\!])$
5 $\quad$ **forall** $v \in \{0,\ldots,n-1\}$ **do**
6 $\quad\quad [\![\mathbf{G}'[\star,v']]\!] \leftarrow \mathsf{apply}([\![\sigma]\!],[\![\mathbf{G}'[\star,v]]\!])$
7 $\quad [\![\vec{I}]\!] \leftarrow \mathsf{unApply}([\![\sigma]\!],[0,1,..,n-1])$
8 $\quad s' \leftarrow \mathsf{declassify}([\![I[s]]\!])$
9 $\quad [\![\vec{K}]\!] \leftarrow \infty$
10 $\quad [\![K[s']]\!] \leftarrow 0$
11 $\quad [\![P[s']]\!] \leftarrow s'$
12 $\quad \vec{M} \leftarrow \mathsf{true}$
13 $\quad$ **for** $idx = 0$ **to** $n-1$ **do**
14 $\quad\quad [\![\vec{L}]\!] \leftarrow NIL$
15 $\quad\quad$ **for** $i = 0$ **to** $n-1$ **do**
16 $\quad\quad\quad$ **if** $M[i]$ **then**
17 $\quad\quad\quad\quad [\![\vec{L}]\!] \leftarrow \mathsf{cons}((([\![K[i]]\!],[\![i]]\!]),[\![\vec{L}]\!])$
18 $\quad\quad u' \leftarrow \mathsf{declassify}(\mathsf{second}(\mathsf{minL}([\![\vec{L}]\!])))$
19 $\quad\quad M[u'] \leftarrow \mathsf{false}$
20 $\quad\quad [\![\vec{D}]\!] \leftarrow [\![\mathbf{G}'[u,\star]]\!]$
21 $\quad\quad [\![\vec{C}]\!] \leftarrow ([\![\vec{D}]\!] < [\![\vec{K}]\!])$
22 $\quad\quad [\![\vec{K}]\!] \leftarrow$ **if** $\vec{M} \wedge [\![\vec{C}]\!]$ **then** $[\![\vec{D}]\!]$ **else** $[\![\vec{K}]\!]$
23 $\quad\quad [\![\vec{P}]\!] \leftarrow$ **if** $\vec{M} \wedge [\![\vec{C}]\!]$ **then** u' **else** $[\![\vec{P}]\!]$
24 $\quad [\![\vec{R}]\!] \leftarrow \mathsf{prepareRead}(n,[\![\vec{I}]\!])$
25 $\quad [\![\vec{T}]\!] \leftarrow \mathsf{performRead}([\![\vec{P}]\!],[\![\vec{R}]\!])$
26 $\quad$ **return** $[\![\vec{T}]\!]$

For the u-th row, we write $\mathbf{G}[u,\star]$, and for the v-th column, we write $\mathbf{G}[\star,v]$. Finding the new identity of the beginning vertex s is the final step in the permutation process. Consequently, we apply the inverse of $[\![\sigma]\!]$ to the identity vector $[0,1,\ldots,n-1]$ (which will be classified in the process), and take its s-th element. This element is a random number picked from the set $\{0,\ldots,n-1\}$. Thereby, this element may be declassified.

The main part of the algorithm (lines 9-23) is finding MST in parallel (using the SIMD framework). We set up the private vectors $[\![\vec{K}]\!]$ and $[\![\vec{P}]\!]$, and the Boolean public vector $\vec{M}$ by initial values; this can be seen in Algorithm 22. The outer **for**-loop has n iterations, which are based on the number of the given vertices.

The algorithm adds one vertex to the MST in each iteration: vector $[\![\vec{T}]\!]$. The operations inside this loop are suitably organized in parallel. The inner for-loop is used to retrieve the permuted vertices one by one until the spanning is complete. Select the

$u' \in V \setminus M$ vertex closest to $\vec{M}$ vertices. All vertices not in $\vec{M}$ are listed in the list $[\![\vec{L}]\!]$, along with their distance from $\vec{M}$. The cons-function is for adding a pair to the list. Repeatedly, the algorithm finds the pair with the minimum first component by function calling (minL), presented in Algorithm 23. The vector is divided into two parts and iterated $\log n$ times in this subroutine. Then select the smallest values in the vectors, and so on, until the last element, which is the smallest pair. The second-function detects the identity of the vertex u'; the value is the pair's second component.

Algorithm 23: minL: minimal first component pair

Data: List of pairs of private values $[\![\vec{w}]\!]$
Result: the element of $[\![\vec{w}]\!]$ with the minimal first component

```
1  begin
2      m ← length([[w]])
3      if m = 1 then return [[w[0]]]
4      begin in parallel
5          ([[e]], [[i]]) ← minL(left_⌊m/2⌋([[w]]))
6          ([[f]], [[j]]) ← minL(right_⌈m/2⌉([[w]]))
7      if [[e]] ≤ [[f]] then
8        return ([[e]], [[i]])
9      else
10       return ([[f]], [[j]])
```

The algorithm declassifies the return values of minL-function to utilize it later as public since the element is private. This is the reason we perform the permutation to the given graph. We have to declassify the vertex; however, we should keep the private data. Later, the remaining operations in the algorithm are like the same operations in classical Prim's algorithm with some proper exceptions, which are done in parallel. The vertex u' will take place as indices in vector $\vec{M}$, and the vector $[\![\vec{D}]\!]$ gets the u-th row from the adjacency matrix $[\![\mathbf{G}]\!]$. The condition (line 21) is that the components of $[\![\vec{D}]\!]$ are less than those of $[\![\vec{K}]\!]$. This condition is used in the algorithm's following two operations (lines 22-23) to update the two vectors $[\![\vec{K}]\!]$ and $[\![\vec{P}]\!]$. For updating $[\![\vec{K}]\!]$, the result is $[\![\vec{D}]\!]$ that will be stored in $[\![\vec{K}]\!]$, if the condition is "true", else, assign $[\![\vec{K}]\!]$. For updating, $[\![\vec{P}]\!]$ is similar.

The real identity of the vertices with their minimal edges is obtained in the final stage of the algorithm (lines 24-25). To get the MST, Laud's protocol for private reading is used. The vector $[\![\vec{I}]\!]$ represents the original order of the vertices saved in ABB. The prepareRead subroutine of the protocol is used to obtain the indices of the vertices using $[\![\vec{I}]\!]$ and a length of n. After that, use performRead to obtain the $[\![\vec{T}]\!]$, which is the MST.

6.3. Optimized Prim's protocol

There is a more efficient way of doing the privacy-preserving computation of MST following Prim's protocol. Hence, we propose our optimized version of the privacy-preserving Prim's MST protocol in Algorithm 24. The improved algorithm is distinguished by the absence of a list of pairs for the vertices and their distances from $\vec{M}$. As a result, the cons-function has been removed, and there is no longer a requirement to set up $\vec{M}$ with "true" values. Furthermore, there is no need to use the inner for-loop. The algorithmic structure of the optimized algorithm makes meaningful sense reducing the round complexity of the

SMC protocols. We provide the privacy-preserving optimized Prim's MST as a pure MST protocol. The permutation and Laud's parts of the optimized method have been relocated to the main program.

Algorithm 24: Optimized-Prim

Data: Number of vertices n
Data: Starting vertex s'
Data: Edge weights $[\![\mathbf{G}']\!]$ (a $n \times n$ array)
Result: Masked Minimum Spanning Tree $[\![\vec{P}]\!]$

1 **begin**
2 $\quad [\![\vec{K}]\!] \leftarrow \infty$
3 $\quad [\![K[s']]\!] \leftarrow 0$
4 $\quad [\![P[s']]\!] \leftarrow s'$
5 $\quad \vec{M} \leftarrow \mathsf{false}$
6 $\quad$ **for** $i \leftarrow 0$ *to* $n-1$ **do**
7 $\quad\quad u' \leftarrow \mathsf{declassify}(\mathsf{second}(\mathsf{minLs}([\![\vec{k}]\!], \vec{M})))$
8 $\quad\quad M[u'] \leftarrow \mathsf{true}$
9 $\quad\quad [\![\vec{U}]\!] \leftarrow u'$
10 $\quad\quad [\![\vec{D}]\!] \leftarrow [\![\mathbf{G}'[u', \star]]\!]$
11 $\quad\quad [\![\vec{C}]\!] \leftarrow ([\![\vec{D}]\!] < [\![\vec{K}]\!])$
12 $\quad\quad [\![\vec{K}]\!] \leftarrow$ **if** $\vec{M} \wedge [\![\vec{C}]\!]$ **then** $[\![\vec{D}]\!]$ **else** $[\![\vec{K}]\!]$
13 $\quad\quad [\![\vec{P}]\!] \leftarrow$ **if** $\vec{M} \wedge [\![\vec{C}]\!]$ **then** $[\![\vec{U}]\!]$ **else** $[\![\vec{P}]\!]$
14 $\quad$ **return** $[\![\vec{P}]\!]$

The optimization targeted the algorithm's operations, not the functionality. In other words, permutation and performRead should be used in the real implementation of the optimized version. Consequently, the permuted adjacency matrix $[\![\mathbf{G}']\!]$, starting vertex s', and the matrix's size n come from the main program. The algorithm starts by setting up the vectors $[\![\vec{K}]\!]$, $[\![\vec{P}]\!]$, and the public vector $\vec{M}$ is by "false"-value. The optimized version has only one **for**-loop. Each iteration relaxes all connected edges of a single vertex using a single instruction.

The minLs-function carried their parameters, vectors of $[\![\vec{K}]\!]$ and $\vec{M}$. Algorithm 25 presents the minimal index (minLs) algorithm for each vertex. This algorithm operates similarly to Algorithm 23, with the distinction that it returns the second element (not a pair), which is either $[\![j]\!]$ or $[\![i]\!]$. The return value of the algorithm will be picked up by second-function; this function picks up the second value. Also, the private value will be declassified to get the public vertex u'. The vertex u' will indicate the indices of vector $\vec{M}$ that will take "true"-value. As well, assigns the vertex u' into vector $[\![\vec{U}]\!]$, and assigns the u'-th row from permuted graph $[\![\mathbf{G}']\!]$ into vector $[\![\vec{D}]\!]$. Lastly, the remains of the operation are similar to a previous algorithm Algorithm 23. The iterations end up by the return value of the vector $[\![\vec{P}]\!]$, which has masked MST. Getting the real identity of MST is by applying Laud's protocol in the main program.

Algorithm 25: minLs: minimal index

Data: Vectors $[\![\vec{k}]\!]$ and $\vec{M}$
Result: The minimal index x

begin
  $m \leftarrow \mathsf{length}(\mathsf{sum}(M == \mathsf{false}))$
  if $m = 1$ **then return** $[\![k[0]]\!]$
  begin *in parallel*
    $([\![e]\!], [\![i]\!]) \leftarrow \mathsf{minLs}(\mathsf{left}_{\lfloor m/2 \rfloor}([\![\vec{k}]\!]), \mathsf{left}_{\lfloor m/2 \rfloor}([\![\vec{M}]\!]))$
    $([\![f]\!], [\![j]\!]) \leftarrow \mathsf{minLs}(\mathsf{right}_{\lceil m/2 \rceil}([\![\vec{k}]\!]), \mathsf{right}_{\lfloor m/2 \rfloor}([\![\vec{M}]\!]))$
  if $[\![e]\!] \leq [\![f]\!]$ **then**
    return ($[\![e]\!]$, $[\![i]\!]$)
  else
    return ($[\![f]\!]$, $[\![j]\!]$)

6.4. Minimum spanning forest protocol

The computation of the Minimum spanning forest is called computational squaring of the MST, which increases the computational cost n times. Greedy algorithm-based computations have a high round complexity based on the algorithm's iterations. Prim's Minimum spanning tree (MST) algorithm has $O(n^2)$ in time complexity. Furthermore, Prim's Minimum spanning forest has time complexity n times the Prim algorithms. The proposed privacy-preserving MSF protocol is effective on both sparse and dense graphs. When dealing with sparse or dense graphs, using the SIMD framework to design the protocol makes no difference. The number of connecting edges between vertices does not affect the entire running time. In other words, regardless of the number of edges, the running time of dense or sparse graphs with the same number of vertices is similar.

We created two versions of the privacy-preserving MSF protocol in the implementations. The MSF's privacy-preserving sequential computation is the first version. The privacy-preserving parallel computation of the MSF is the second, and it is more efficient than the first. We put them against each other to see how efficient parallel computing is. We employ a variety of graphs of varied sizes in evaluations.

6.4.1. Sequential minimum spanning forest protocol

Our proposed privacy-preserving MSF protocol entails performing the privacy-preserving Prim's MST algorithm for each component in the given unconnected graph. The return value of running MST protocol for each component is the minimum total edge weight. Consequently, the entire edges' weights for all given components are minimum. The privacy-preserving MSF is presented in Algorithm 26. The parameters are a 3-dimensional matrix $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{g \times n \times n}$, and sources $\vec{s}$, each source for a graph. The first two dimensions are an adjacency matrix for one component, and the last is several components.

The algorithm has only one **for**-loop. The operations are similar to the functionality of finding MST, except we utilize the optimized version of Prim's algorithm. For each graph, the algorithm generates a random private permutation $[\![\vec{\sigma}]\!]$ of length n. Moreover, the algorithm performs the permutation for each graph, shuffling rows and columns sep-

arately and getting the new identity of the starting vertex s after permutation. The return value of the optimized Prim's algorithm is masked MST $[\![\vec{P}]\!]$ into $[\![\vec{F}']\!]$. The last part of the algorithm applies Laud's protocol to get the real identity of the vertices in MST for each graph. Then, assign the MST vector to the adjacency matrix $[\![\mathbf{F}]\!]$. Each MST vector will take place as a row in $[\![\mathbf{F}]\!]$.

Algorithm 26: MSF Algorithm, Main Program

Data: Number of vertices n
Data: Starting vertices $[s]$
Data: Lengths of edges $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{g \times n \times n}$
Result: Minimum Spanning Forest $[\![\mathbf{F}]\!]$

1 **begin**
2 **for** $i = 0$ *to* $g-1$ **do**
3 $[\![\sigma]\!] \leftarrow \mathsf{randPerm}(n)$
4 **forall** $u \in \{0,\ldots,n-1\}$ **do**
5 $[\![\mathbf{G}'[i,u',\star]]\!] \leftarrow \mathsf{apply}([\![\sigma]\!],[\![\mathbf{G}[i,u,\star]]\!])$
6 **forall** $v \in \{0,\ldots,n-1\}$ **do**
7 $[\![\mathbf{G}'[i,\star,v']]\!] \leftarrow \mathsf{apply}([\![\sigma]\!],[\![\mathbf{G}'[i,\star,v]]\!])$
8 $[\![\vec{I}]\!] \leftarrow \mathsf{unApply}([\![\sigma]\!],[0,1,..,n-1])$
9 $s' \leftarrow \mathsf{declassify}([\![I[s[i]]]\!])$
10 $[\![\vec{F}']\!] \leftarrow \mathsf{Optimized\text{-}Prim}(s',n,[\![\mathbf{G}'[i,\star,\star]]\!])$
11 $[\![\vec{R}]\!] \leftarrow \mathsf{prepareRead}(n,[\![\vec{I}]\!])$
12 $[\![\mathbf{F}[i,\star]]\!] \leftarrow \mathsf{performRead}([\![\vec{F}']\!],[\![\vec{R}]\!])$
13 **return** $[\![\mathbf{F}]\!]$

6.4.2. Parallel minimum spanning forest protocol

This section presents the parallel version of the privacy-preserving MSF protocol. The computation of the privacy-preserving MSF has multiple components to be processed simultaneously. The large unconnected graphs can often be processed using a single instruction, which can be solved simultaneously with fewer iterations. Such an approach will reduce the round complexity of finding the MSF on top of the SMC protocol. The privacy-preserving parallel computation of the minimum spanning forest is presented in Algorithm 27. The parallel MSF protocol has three main parts.

The permutations part (lines 2-9) finds the permutation for each graph separately and by using different randomness each time. The permuted graphs will be stored in a 3-dimensional adjacency matrix $[\![\mathbf{G}'[g,\star,\star]]\!]$. The three parameters will be carried to the nPrim function. The processing of this function is only one time and for all components. In contrast, the sequential version of the protocols process Optimized-Prim function n times separately for each component; this feature of the parallel version will reduce the round complexity. More specifically, parallel MSF performs a single instruction to a group of sub-vectors combined in a large vector $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{n \times n}$. In other words, we perform the same instruction on multiple data sets simultaneously. The large vector has n segments of the data set, each segment for a graph. Each segment is a row in the graph $[\![\mathbf{G}[i]]\!]$. Algorithm 28 is a vectorized version of Algorithm 24 that can compute the MST for several n-vertex graphs simultaneously.

Algorithm 27: SIMD-MSF, main program

Data: Number of vertices n
Data: Starting vertices $\vec{s}$
Data: Lengths of edges $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{g \times n \times n}$
Result: Minimum Spanning Forest $[\![\mathbf{F}]\!]$

1 **begin**
2 **for** $i = 0$ *to* $g-1$ **do**
3 $[\![\sigma[i,\star]]\!] \leftarrow \mathsf{randPerm}(n)$
4 **forall** $u \in \{0,\ldots,n-1\}$ **do**
5 $[\![\mathbf{G}'[i,u',\star]]\!] \leftarrow \mathsf{apply}([\![\sigma[i,\star]]\!], [\![\mathbf{G}[i,u,\star]]\!])$
6 **forall** $v \in \{0,\ldots,n-1\}$ **do**
7 $[\![\mathbf{G}'[i,\star,v']]\!] \leftarrow \mathsf{apply}([\![\sigma[i,\star]]\!], [\![\mathbf{G}'[i,\star,v]]\!])$
8 $[\![I[i,\star]]\!] \leftarrow \mathsf{unApply}([\![\sigma[i,\star]]\!], [0,1,..,n-1])$
9 $s'[i] \leftarrow \mathsf{declassify}([\![I[i,s[i]]]\!])$
10 $[\![\vec{F}']\!] \leftarrow \mathsf{nPrim}([s'], n, [\![\mathbf{G}]\!])$
11 $[\![\mathbf{F}']\!] \leftarrow \mathsf{Matrix}([\![\vec{F}']\!])$
12 **for** $i = 0$ *to* $g-1$ **do**
13 $[\![\mathbf{R}[i,\star]]\!] \leftarrow \mathsf{prepareRead}(n, [\![I[i,\star]]\!])$
14 $[\![\mathbf{F}[i,\star]]\!] \leftarrow \mathsf{performRead}([\![\mathbf{F}'[i,\star]]\!], [\![\mathbf{R}[i,\star]]\!])$
15 **return** $[\![\mathbf{F}]\!]$

We have designed Algorithm 28 that can simultaneously process multiple components of an unconnected graph. The input data is a three-dimension adjacency matrix, while each matrix contains a graph's data (size is $n \times n$). The second parameter is the sources vector $\vec{s}'$, while every source is for a graph (or a component). The output is an adjacency matrix (size is $n \times n$), and each row is the minimum spanning tree for a component. In other words, the $n \times n$ adjacency matrix is a minimum spanning forest for an unconnected graph.

The algorithm begins (lines 2-3) by replacing "∞" with "false" for the vectors $[\![\vec{D}]\!]$ and "false" for the Boolean public vector for unhandled vertices $\vec{M}$. The part (lines 4-8) is for getting the sources of vector $[\![\vec{K}]\!]$ and $[\![\vec{P}]\!]$ as initial values, and the private indices of $[\![\vec{d}]\!]$. The second for-loop is the main body of the nPrim algorithm. This portion starts by calling the minLv-function, represented in Algorithm 10. This function is to determine the minimum key values (from the set of vertices not yet included in MST) for all components simultaneously in parallel. Note, Algorithm 10 is used as a subroutine in both algorithms, nPrim for finding MST, and in nDijkstra for finding the shortest path. The minLv-function's parameters are private vector $[\![\vec{K}]\!]$, Boolean vector $\vec{M}$ for unhandled vertices, and the indices $[\![\vec{d}]\!]$, where the length of all vectors is $n \times n$.

The return value of Algorithm 10 is the minimum value from the set of vertices not yet processed for each adjacency matrix. In the **for**-loop (lines 9-21) calls Algorithm 10 for each iteration. A private vector of the vertices is the return value. To make $[\![\vec{u}]\!]$ public, the algorithm declassifies it. Because the algorithm runs with masked vertices, declassifying will not affect the vertices' privacy (not the real identity). Each element in this vector represents the identity of a vertex in a graph. The algorithm then relaxes all neighboring vertices in parallel for each component.

Algorithm 28: nPrim

Data: Number of vertices n
Data: Starting vertices $\vec{s}$
Data: Lengths of edges $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{g \times n \times n}$
Result: Private distances $[\![\vec{P}]\!]$

1 **begin**
2 $\quad [\![\vec{D}]\!] \leftarrow \infty$
3 $\quad \vec{M} \leftarrow \mathsf{false}$
4 $\quad$ **for** $i = 0$ *to* $n-1$ **do**
5 $\quad\quad [\![K[i*n+s[i]]]\!] \leftarrow 0$
6 $\quad\quad [\![P[i*n+s[i]]]\!] \leftarrow s[i]$
7 $\quad\quad$ **for** $j = 0$ *to* $n-1$ **do**
8 $\quad\quad\quad [\![d[i*n+j]]\!] = j$
9 $\quad$ **for** $i = 0$ *to* $n-1$ **do**
10 $\quad\quad [u'] \leftarrow \mathsf{declassify}((\mathsf{minLv}([\![\vec{K}]\!], \vec{M}, [\![\vec{d}]\!]))$
11 $\quad\quad start = 0, end = 0, range = 0$
12 $\quad\quad$ **for** $u = 0$ *to* $g-1$ **do**
13 $\quad\quad\quad range = u*n+u'[u]$
14 $\quad\quad\quad end = start+n$
15 $\quad\quad\quad [\![D[start:end]]\!] = [\![\mathbf{G}'[u,u'[u],\star]]\!]$
16 $\quad\quad\quad M[range] = \mathsf{true}$
17 $\quad\quad\quad [\![U[start:end]]\!] \leftarrow u'[u]$
18 $\quad\quad\quad start = start+n$
19 $\quad\quad [\![\vec{C}]\!] \leftarrow ([\![\vec{D}]\!] < [\![\vec{K}]\!])$
20 $\quad\quad [\![\vec{K}]\!] \leftarrow$ **if** $\vec{M} \wedge [\![\vec{C}]\!]$ **then** $[\![\vec{D}]\!]$ **else** $[\![\vec{K}]\!]$
21 $\quad\quad [\![\vec{P}]\!] \leftarrow$ **if** $\vec{M} \wedge [\![\vec{C}]\!]$ **then** $[\![\vec{U}]\!]$**else** $[\![\vec{P}]\!]$
22 $\quad$ **return** $[\![\vec{P}]\!]$

In the **for**-loop (lines 12-18), the algorithm vectorizes the given adjacency matrices and gets the state of the vertices. In detail, based on the identity of $u'[u]$, the algorithm receives the rows from all matrices. Then the rows will be vectorized as segments into vector $[\![\vec{D}]\!]$. Also, the Boolean vector $\vec{M}$ for visited vertices is updated by "true". The other operations in this section will assign the components' real places and ranges. This for-loop iterates g times, where g is the number of unconnected graph components. In part (lines 19-21), the vectors $[\![\vec{K}]\!]$ and $[\![\vec{P}]\!]$ will be updated if $[\![\vec{D}]\!]$ is smaller than $[\![\vec{K}]\!]$, and if $\vec{M}$ is "false" for vertices not yet included in masked MSF, which is $[\![\vec{P}]\!]$. This update is for all components simultaneously.

Finally, the return value of the nPrim function is masked MSF $[\![\vec{P}]\!]$ that will be stored in vector $[\![\vec{F}']\!]$ in the main program. Line 11 in Algorithm 27 is converting the vector $[\![\vec{F}']\!]$ to the adjacency matrix $[\![\mathbf{F}']\!]$. The last part (lines 12-14) in the Algorithm 27 applies Laud's protocol to get the real identity of the MSF. The number of iterations is g times based on the number of unconnected graphs. The algorithm will end up with the private minimum spanning forest located in an adjacency matrix $[\![\mathbf{F}]\!]$.

6.5. Performance Analysis

This section presents the two communication-related complexities of our proposed MST and MSF protocols in secure multiparty computation applications, round, and communication, as discussed in Sec 4.6.

6.5.1. Round complexity

Let us start with the privacy-preserving Prim's protocol (Algorithm 22), where the private data is structured in an adjacency matrix $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{n \times n}$. The number of vertices is n, and the number of edges is no more than n^2. One iteration of the main **for**-loop requires $O(\log n)$ rounds, minLs is the only operation working in non-constant rounds. The entire loop thus requires $O(n \log n)$ rounds. The permutation which occurs before the main loop requires a constant number of rounds. The last part in the algorithm, which is the private read, has $O(\log n)$.

The optimized version of Prim's MST algorithm (Algorithm 24) has only one **for**-loop that requires $O(n)$ rounds, each iteration has minLs-function which requires $O(\log n)$. The entire algorithm thus $O(n \log n)$, but without the permutation part. The permutation part for the Optimized-Prim's algorithm will be processed in the main program with a constant number of rounds. As well no change in the round complexity of the private read, which is still $O(\log n)$, but in the main program.

For the sequential version of the MSF protocol (Algorithm 26), the data of the given unconnected graph presented in the 3-dimensional adjacency matrix $[\![\mathbf{G}]\!] \in (\mathbb{N} \cup \{\infty\})^{g \times n \times n}$. Assume that the number of components in an unconnected graph g is comparable to the number of vertices in a connected graph (one component) n. Thereby, each iteration requires $O(n)$ rounds for permutation parts.

The round complexity of running Optimized-Prim function is $O(n^2 \log n)$. The last part is Laud's protocol requires $O(\log n)$ round for each graph. In the case of g components in an unconnected graph, the private read has $O(n \log n)$ round complexity. In the case, $g \neq n$, round complexity of permutation is $O(g)$, Optimized-Prim function requires $O(gn \log n)$ rounds, private read is $O(g \log n)$. Note the smallest MSF can be constructed only by two graphs (or two components in an unconnected graph.

The parallel MSF protocol is similar to sequential MSF in the main program, except for Prim's function. Consequently, the round complexity of the Parallel MSF's permutation part is $O(n)$, where $g = n$. Moreover, the private read also requires $O(n \log n)$. The calling of the nPrim function is the only constant round in the main program. However, the nPrim-function has two loops. The round complexity is $O(n^2)$, and minLv-function is only constant round. Anyway, the entire nPrim-function thus requires $O(n^2)$ rounds. This is the justification for why parallel MSF protocol has a low running time compared with sequential MSF protocol. The round complexities for privacy-preserving MST and MSF protocols with their parts are presented in Table 3.

6.5.2. Communication complexity

Besides the round complexity of the protocols, we highlight the performance in terms of communication complexity. In privacy-preserving Prim's protocol, each iteration of the main **for**-loop has $O(n)$ communication. The entire loop requires $O(n^2)$ communication. The permutation part before that loop has also $O(n^2)$ communication. The last part in the algorithm (private read) has $O(n \log n)$ communication.

Table 3: Round and communication complexities for privacy-preserving MST and MSF protocols.

Protocol		**Complexities**	
		Round	Communication
Prim	Prem.	$O(1)$	$O(n^2)$
	Loop	$O(n \log n)$	$O(n^2)$
	PerfR.	$O(\log n$	$O(n \log n)$
Optimized-Prim	Prem.	$O(1)$	$O(n^2)$
	Loop	$O(n \log n)$	$O(n)$
	PerfR.	$O(\log n$	$O(n \log n)$
MSF V1	Perm.	$O(n)$	$O(n^3)$
	Loop	$O(n^2 \log n)$	$O(n^2)$
	PerfR.	$O(n \log n)$	$O(n^2 \log n)$
MSF V2	Perm.	$O(n)$	$O(n^3)$
	nPrim	$O(n^2)$	$O(n^3)$
	PerfR.	$O(n \log n)$	$O(n^2 \log n)$

In the optimized version of the privacy-preserving Prim's protocol, the entire for-loop thus requires $O(n)$ communication. We notice that Optimized-Prim algorithm has less communication than Prim's algorithm; the benchmark result documented in Sec 9.3 confirms that.

In the sequential MSF protocol, the lowest number of the components of the MSF is only two components in an unconnected graph (or two graphs only). In this case, the sequential MSF's protocol requires $O(n^2)$ communication, the permutation part in the main program requires $O(n^2)$ communication, and $O(n \log n)$ communication for the last part of the algorithm—laud protocol.

Let us assume that the components' number in an unconnected graph is similar to the number of the vertices in a single component, $g = n$. Consequently, the communication complexity for the sequential MSF's protocol is as follows: the permutation is $O(n^3)$, the Optimized-Prim is $O(n^2)$ communication, and the Laud's protocol is $O(n^2 \log n)$. In other words, for both round and communication, complexities of the sequential MSF version are similar to the optimized-Prim algorithm but n times.

For the parallel MSF protocol, the operations in the first and last parts are done in **for**-loop, while nPrim function has the constant call. In the case that $g = n$, the permutation part requires $O(n^3)$ communication, while the Laud's protocol is $O(n^2 \log n)$. The nPrim function might be reduced the round complexity compared with running optimized-Prim function n times. In contrast, this function increased communication. The length of the working vectors is $n \times n$. Moreover, the nPrim function has two for loops.

The inner for loop (lines 12-18) requires $O(n^2)$ communication, while each iteration in the outer loop requires $O(n^2)$ communication. The outer loop runs the inner loop n times. Consequently, the communication of the nPrim algorithm is $O(n^3)$. The communication complexities for privacy-preserving MST and MSF protocols with their parts are presented in Table 3.

6.6. Security and privacy of protocols

The algorithm preserves the security and privacy properties of the underlying implementation of the ABB, including its resistance against semi-honest or malicious adversaries, as long as the algorithms constructed on top of the ABB do not declassify any information. This chapter proposes that the privacy-preserving Prim's protocol is a crucial building block for other proposed protocols, including MSF.

The discussion in Sec 4.2 is presented for SSSD protocols, including Dijkstra's algorithm. As is well-known, Dijkstra's algorithm and Prim's algorithm have different functionalities. However, both algorithms share the same structure of data input. Both have the same number of iterations and the same subroutine. Our SIMD implementation for both algorithms is also similar. The gain of parallelism as a round complexity for both algorithms is similar, and the declassification statements with permutations are similar. Consequently, the proposed protocol of Prim's algorithm—which is the critical point in constructing MSF protocol—is privacy-preserving. In addition, nDijkstra's and nPrim's algorithms are similar in properties, parallelism gain, and SIMD implementation, hence different in functionality. Following the discussion in Sections 3.2.3 and 4.7, we conclude that MST and MSF protocols are all privacy-preserving.

7. PRIVACY-PRESERVING ALGEBRAIC PATH COMPUTATION PROTOCOL

7.1. Introduction

In previous chapters, we have seen privacy-preserving shortest-path algorithms that hide the edges' length and the edges' existence. However, the problem of privacy-preserving SSSD is also interesting if the edges' end-points are public, and only their weights are private. Some algorithms from previous chapters cannot significantly benefit from the knowledge of edge locations, while others can. Additionally, we can propose algorithms with even higher parallelism if we know the location of edges. In this chapter, we propose one that achieves high parallelism for certain graphs, including planar graphs. Even though this algorithm works only for undirected graphs. However, it also works for directed graphs but needs some changes in the algorithm. This technique of finding the shortest distances is algebraic path computation using a semiring framework in SIMD parallel with public edges on SMC protocol. Furthermore, we present a version of the Bellman-Ford protocol with public edges; its purpose here is to serve as a baseline.

7.2. Overview of algebraic path computation

In the algebraic path computation protocol that we propose in this chapter, we follow a strategy for constructing this protocol in parallel. The essential algorithms and framework in building the solution of path algebra problems through the algorithms of Pan and Reif [136, 137, 135], and the correctness of the formulas are proven there. An efficient algorithm for computing the minimum cost path for an adjacency matrix associated with an undirected graph $G(A)$ is proposed in [137]. Pan and Reif algorithm of computing algebraic paths is based on recursive factorization presented in [135], the aim is to compute matrix A^*, by following equations, $h = 0, 1, \ldots, d$, where $d = O(\log n)$:

$$A_h = \begin{bmatrix} X_h & Y_h^T \\ Y_h & Z_h \end{bmatrix} \qquad A_{h+1} = Z_h \oplus Y_h X_h^* Y_h^T \tag{7.1}$$

$$A_h^* = \begin{bmatrix} I & X_h^* Y_h^T \\ O & I \end{bmatrix} \begin{bmatrix} X_h^* & O \\ O & A_{h+1}^* \end{bmatrix} \begin{bmatrix} I & O \\ Y_h X_h^* & I \end{bmatrix} \tag{7.2}$$

Indeed, the special recursive factorization in [135] and Cholesky factorization in [122] can not be extended to the case of semiring framework (dioids) because of a lack of subtraction and division. In [136] a particular recursive factorization of the inverse matrix A^{-1} of [135] has extended to the similar factorization of the quasi-inverse A^*. This extension is sufficient in many path algebra computations, particularly the one we use in this work. Using the concept of the inverse matrix $(I\text{-}A)^{-1}$, the quasi-inverse A^* is defined for the case of semiring framework (or dioids). The matrix equations 7.1 and 7.2 generalize the recursive factorization of matrix A. This recursive factorization easily solves the system of linear equations $Ax = b$ for any given vector b. The result of finding a single-source shortest path is the multiplication of a unit vector b with a matrix of A^*; the SSSD is given by $x = A^* b$. The partition of the $n_h \times n_h$ adjacency matrix A_h is based on the separator structure of the graph into four parts, X_h, Y_h, Z_h and Y_h^T—the transpose of a matrix Y_h.

The adjacency matrix A of an undirected graph is symmetric, and this is the one we use in the implementation and benchmarking—implementation and benchmarking is only for undirected graph. For instance, of directed graphs, the algebraic path computation protocol on top of SMC can be extended to the case of a non-symmetric linear system with directed graphs. Some changes should be made, replacing the matrix Y_h^T with the matrix W_h for all levels of h. Matrix W_h is given by $U_h \cdot Y_h^T$, while matrix U_h is given by $Y_h \cdot X_h^{-1}$. Hence, the assumption that matrix X_h in all levels of h is symmetric should be removed.

The four submatrices X_h, Y_h, Z_h, and Y_h^T can be obtained by applying an efficient parallel algorithm in [135] that computes the recursive factorization of matrix A. A recursive s(n)-factorization of an adjacency symmetric A is a sequence of sub-matrices $A_0, A_1, \ldots,$ A_d, such that $A_0 = PAP^T$, where P is an $n \times n$ permutation adjacency matrix A_h, the size of matrix A_h is $n_{d-h} \times n_{d-h}$. In any n-vertex planar graph $G = (V, E)$, the symmetric matrix A associated with a graph $G = G(\alpha)$ having an s(n)-separator family with respect to two constants α and n_0, the graph G have s(n)-separator family if either $|V| \leq n_0$ or by erasing some separator set of vertices $O(\sqrt{n})$ [123]. The partitioning of the graph G (which is also called separation) into two disconnected subgraphs G_1 and G_2 that have at most $2n/3$ with two sets of vertices $|V_1|$ and $|V_2|$, and the Separator S which has $O(\sqrt{n})$ vertices. The Separator S is a vector of vertices shared between the two partitioned new graphs, G_1 and G_2, which is responsible for the algorithm's performance. Once the separation produces three sets A, S, and B, the edges-endpoint in A belongs to subgraph G_1, and the edges-endpoint in B belongs to subgraph G_2. In contrast, the edges-endpoint of Separator S and the remaining edges are separated arbitrarily. The separator tree is the adjacency matrix of subgroups G_1 and G_2 resulting from partitioning. Furthermore, each of the two subgraphs also has an s(n)-separator family, and it is not required that G_1 and G_2 have connected subgraphs of the parent graph G. The algorithm recursively keeps portioning until obtaining both subgraphs that have at least $n/3$ vertices set.

Example: Consider a grid graph with $n \times n$ size, with rows numR and columns numC. The number of the vertices in the adjacency matrix equals $\mathsf{numR} \times \mathsf{numC}$. For instance, the illustration in Figure 4, $\mathsf{numR} = 5$, and $\mathsf{numC} = 5$, the number of vertices $N = \mathsf{numR} \times \mathsf{numC} = 25$. The partitioning starts by selecting the central row or column in the adjacency matrix A. Suppose rows numR is an odd number, and the single central row is separator S. Otherwise, two rows are equally near the center. Vertically, if numC is an even number, there are two columns near the center; otherwise, the single center column is separator S. Choosing separator S to be any of these central rows or columns. Next, graph G will be partitioned into two smaller connected subgraphs, G_1 and G_2. Consequently, the result of partitioning graph G is two subgraphs G_1, G_2, and separator S, all called s(n)-separator family. The separator S tree will be shared in the two subgraphs as connectors. We expect the rows/columns of the adjacency matrix to be labeled with the graph's vertices G in a certain order based on the separator tree.

Figure 4 shows the process of the partitioning and the level of the partitioning h. In other words, the depth of the separator tree d. The recursive factorization produces the separator families. It also shows the arguments that will be used in the main computation of the algebraic path. The separator trees for each level with the order of vertices are illustrated, and the elements will be stored in public vector $\overrightarrow{ST}$. Moreover, the size of blocks $\overrightarrow{SB}$ and their values and indices are also obtained. In detail, the main program of privacy-preserving algebraic path computation is presented in Algorithm 32.

It has eight arguments from the prerequisite computation—s(n)-separator tree and its

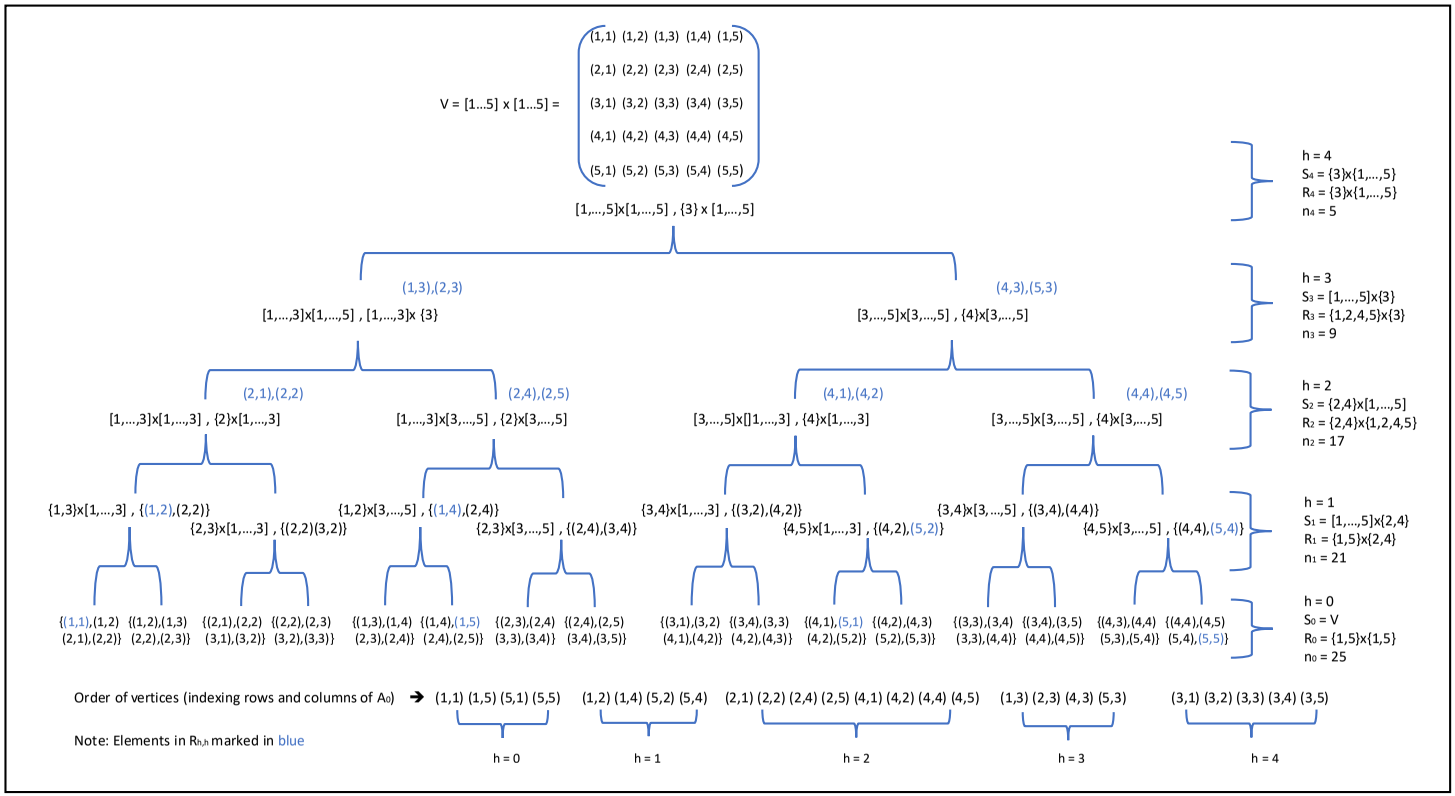


Figure 4: Separator tree and its arguments.

properties. The primary prerequisite functions are separator-tree and its properties in computing R. The set of vertices $R_{h,k}$ denote the set of all vertices of separator tree $S_{h,k}$ that are not in S_{h^*} for $h^* > h$, for each $k = 1, \ldots, N_h$, for more details about R, refer to [135]. Practically, R-function returns the set of R and its properties, which are indices of R in its s(n)-separator family, the size of separator tree $\overrightarrow{ST}$, and the size of block-diagonal matrices $\overrightarrow{BS}$ for all separator trees in an adjacency matrix A.

7.3. Privacy-preserving algebraic path computation protocol

In Sec 7.2, we presented the essential algorithm, definitions, and equations that end up by proposing the algebraic path computation protocol using a semiring framework. The algorithm in [136] is based on the extended definitions of solving sparse linear systems from [135], then a parallel version of the algorithm has been proposed [137]. This section presents a privacy-preserving implementation of the parallel version of the algebraic path computation using a semiring framework. The main feature of our proposed implementation is reshaping the whole computations and data input in sparse representation. This representation is fit to process the given private graph on SIMD parallel computation over a secret-sharing based SMC Sharemind platform.

The sparse representation of matrices with the operations has been done based on the prerequisite operations over a private undirected graph. Those prerequisite operations are s(n)-separator tree and its properties that can be obtained using the public elements of the given graph and its adjacency matrices. The graph's edges E are assumed to be public, while the private data consists only of the edge weights $W\colon E \to \mathbb{R}^+$.

The s(n)-separator tree and its properties can be obtained using the indices of the public edges. Due to this setting, operations of s(n)-separator tree can be done in a local server of SMC Sharemind with no communication with other servers—there are no round and communication complexities.

The data input is symmetric matrix $[\![\mathbf{A}]\!]$ that has been represented sparsely associated with an undirected graph G. We rearrange given data in the adjacency matrix into a sparse representation of matrices. To convert from a dense to a sparse representation of matrices, a Struct that grouped different matrix elements is defined. It has four public elements and one private, which is weight. This data model vectorizes the matrix $[\![\mathbf{A}]\!]$ into three vectors/lists, rows $\vec{R}$, columns $\vec{C}$ and the vector for weights' edges $[\![\vec{W}]\!]$. The number of rows and columns of the matrix should be given, which are denoted numR and numC will be used on related functions arguments of the main program. This structure is indicated by a function that transforms graph coordinates into a sparse representation of matrices. The vectors for both rows $\vec{R}$ and columns $\vec{C}$ are from the matrix $[\![\mathbf{A}]\!]$, while the size of the sparse representation of the matrix is $n \times n$ vertices of a graph G.

Although the SIMD operations have been applied in computation, we omitted the infinite edges (which means no edges between two vertices) to reduce the size of the vectors that can only handle the meaningful edges (non-infinity). This will reduce the bottleneck of the SMC Sharemind during communication between servers. It is important to note that in the operations in the algebraic path computation, no processing has occurred for the dense representation of $\mathbf{A}$, all processing on the sparse representation of the matrices, e.g., Y_h. Hence we need both numR and numC to be obtained before the beginning of the computation. This section presents the related functions and their algorithms for the main computation of the algebraic path; these functions are constructed in parallel.

7.3.1. Related functions

The whole related functions of the main computation carry a sparse representation of the given graphs. First, we present the parallel version of the factorization and Block diagonal matrix functions. Those functions can be used in the main computation of algebraic paths. Hence it can be used in different computations in algebraic computation. As well as the principal operations in the algebraic path, which are computing the Sum and Min in sparse representation, both operations are also constructed in SIMD parallel.

The last two related functions are the First and Second normalization. Both functions are also constructed in SIMD parallel and their input data-sparse representation.

Factorization

The first related function in the main program is Factorization, the Factorization for matrix A—which is represented sparsely—that returns four matrices, X, Y, Z and Y^T which is the transpose of Y. Those matrices will be stored in a special Struct called sparseF, and all matrices are sparse. The function has two arguments, the sparse representation of the matrix A and the number of vertices in separator tree $\overrightarrow{ST}$ for level h. The function splits the given sparse representation of A into four different-sized matrices. The given vertices' number of $\overrightarrow{ST}$ determines the size of the matrix X. Suppose $k = ST[cyc]$, and the size of the matrix A is $n \cdot n$, then the size of X is $k \cdot k$. The sizes of remains matrices are based on the size of matrix X; this can be seen in Figure 5.

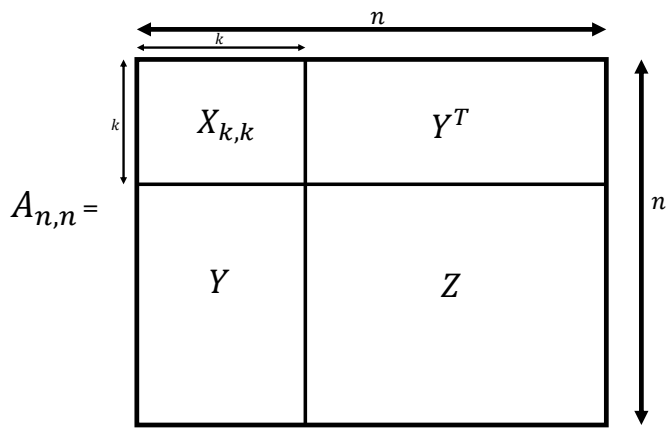


Figure 5: Blocks of recursive factorization.

First-normalization

The first normalization is for sorting the elements of the matrix by rows and columns—puts entries corresponding to the same cell next to each other. We constructed the algebraic path protocol with related functions to process a sparse representation of matrices on the SIMD framework. The elements of the sparse representation of matrices need to be sorted before applying the second normalization. To solve this problem, we propose the first normalization of numbers for the sparse representation of the matrices' elements—rows, columns, and weights of a graph. Using the first normalization, numbers can be

presented differently, bringing the sparse representation of matrices into a more canonical form. Furthermore, in particular, it enables Second-normalization.

Second-normalization

The sparse representation of matrices increases the size of the vectors processed in parallel SIMD. Consequently, this causes a new problem regarding the vectors' size and the size of a graph. We propose the second normalization for duplicate numbers of the sparse representation of matrices to solve this problem. The aim is to remove the duplicate occurrences of the same cell; hence vectors can carry more data, and bottleneck problems among the servers of the SMC platform will be reduced, particularly when using big graphs.

The adjacency matrix's second normalization, represented in sparse representation, is illustrated in Algorithm 29. In general, getting the indices of the $\vec{R}$ and $\vec{C}$ is based on conditional expression $A.R[i\text{-}1] \neq A.R[i]$ or $A.C[i\text{-}1] \neq A.C[i]$. The elements of private vector $A.W$ are based on the minimal values of $A.W$ and t, which are obtained by applying the getMin-function. In general, lines 4-8 compute the coordinates of the cells in the resulting matrix. Lines 13 and 14 compute the values in these cells.

Algorithm 29: Second-normalization

```
   Data: Struct sparse A
   Result: Struct sparse val
1  begin
2      if size(A.R) == 0 then
3          return A
4      for i ← 1 to size(A.R) do
5          if (A.R[i-1] ≠ A.R[i]) || (A.C[i-1] ≠ A.C[i]) then
6              R[c] = A.R[i-1]
7              C[c] = A.C[i-1]
8              c++
9      R[c] = A.R[size(A.R)-1]
10     C[c] = A.C[size(A.R)-1]
11     val.R = R[0 : c + 1]
12     val.C = C[0 : c + 1]
13     [t] = A.R × A.numC + A.C + 1
14     val.W = getMin(A.W, t)
15     val.numR = A.numR
16     val.numC = A.numC
17     return val
```

Block diagonal matrix

The second related function in the main program is the block diagonal matrix. It has two arguments, the matrix $X_{k,k}$, and square blocks matrices in a level h. Both arguments are matrices but in sparse representation—elements of the matrices stated in vectors. The **while**-loop (lines 5-7) in Algorithm 32 is to feed the second argument in the block diagonal matrix function. It picks up indices of the blocks' matrices from $\overrightarrow{BS}$ in a level h.

The number of the blocks' matrices is obtained from $\overrightarrow{ST}$ in that level h. The block diagonal matrix is presented in Algorithm 30. We build this algorithm in parallel to perform the computation over vectors to reduce the iteration over private elements. The getSlice-function is to determine the dimensions and sizes of the block matrices in the given vector A, then determine the indices of the elements located in different locations.

Later, we transform the data from the sparse representation of matrix in A to a dense representation in B by applying sparse-to-dense-function. The next step is to multiply the private elements of the blocks' matrices in parallel (line 5). This computation can be done in the same algorithmic structure as the Floyd-Warshall algorithm with some changes based on the semiring framework $\oplus$. Thereby, we use the parallel version of the Floyd-Warshall algorithm presented in Algorithm 20 to perform this computation. It is important to note that Algorithm 20 performs the computation on one adjacency matrix. Hence, we run the Floyd-Warshall algorithm over n blocks matrices simultaneously.

Next, the operation in the algorithm is to get the elements of $\vec{C}$ and its rows and columns indices. $C[i]$ has a similar size to A, and the content of $A'[i]$ goes to the same place (i.e., into the same block) from where $A[i]$ was read. We use the dense-to-sparse-function for transforming to sparse representation before applying overlay-function. The return value of the Block diagonal matrix is X_h^*, and its data is presented in sparse representation.

Algorithm 30: Block-Diagonal-Matrix-inv

Data: Struct sparse A, $\overrightarrow{BS}$
Result: Struct sparse S

```
1 begin
2     forall i ∈ {1,...,|BS|} do
3         A[i] ← getSlice(A, 0, Σ_{j=1}^{i-1} BS[i], 1, BS[i])
4         B[i] ← sparse-to-dense(A[i])
5         B'[i] ← FloydWarshall-nSIMD(B[i])
6         B[i] ← dense-to-sparse(B'[i])
7         C[i] ← overlay(B[i],...,...)
8     return overlap(C_1,...,C_|BS|))
```

Sum-sparse operation

In the semiring framework, the main mathematical operations are **sum** and **min**, which will be executed five times for Sum, while Min appears only once for each recursive cycle in the main program. The sum operation is constructed sparsely as input data and processes in parallel, which can reduce the round complexity of SMC protocol. It has two arguments in sparse representation, Y and X_h^*, in its first use in the main computation. The parallel **Sum**-operation—in sparse representation—based on a semiring framework is presented in Algorithm 31. The algorithm supposes that the first argument has the same number of columns as the second argument's number of row assert(X.numC == Y.numR), similar to the matrix multiplication in linear algebra. The portion (lines 2-7) is to get the elements for both sparse matrices X and Y into $\vec{R}$ and $\vec{C}$, respectively, using cons-function. Then, apply the summation for both $[\![\vec{W}]\!]$ of X and Y.

Algorithm 31: Sum-sparse

```
Data: Struct sparse X, Y
Result: Struct sparse B
1  begin
2      for i ← 0 to size(X.R) do
3          for j ← 0 to size(Y.R) do
4              if X.C[i] == Y.R[j] then
5                  S.R ← cons(X.R[i], S.R)
6                  S.C ← cons(Y.C[j], S.C)
7                  S.W ← cons(X.W[i] + Y.W[j], S.W)
8      S.numR = X.numR
9      S.numC = Y.numC
10     A = First-normalization(S)
11     B = Second-normalization(A)
12     return B
```

The double **for**-loop may take much running time, particularly if the matrices X and Y are large. To optimize the portion, we assume that Y has been normalized, and then we reorder the points in X by columns. Ordering the points of X by the columns corresponds to transposing X and then First-normalizing it. We then do a single loop, moving forward along the columns of X and rows of Y. Whenever we find a column index of X that equals a row index of Y, we add things into S. This change lets the algorithm get the elements' indices of X identically with Y to perform the sum in parallel.

The single **for**-loop in the algorithm is for getting the elements of the public vectors $\vec{R}$ and $\vec{C}$, and indices of the private vector $[\![\vec{W}]\!]$. Using single instruction, a summation operation will be performed for vectors X and Y, and save the result in new struct S. The last operation in the **sum** algorithm is the First and Second normalization.

Min-sparse operation

The min operation in our algebraic path computation is used only once. The algorithm is represented sparsely to deal with the sparse representation of the adjacency matrix located in Struct sparse. The data input is two arguments in sparse representation X and Y. The algorithm starts by checking that the first argument X rows have the same number as the second argument rows Y—assert(X.numR == Y.numR). As well as checking the number of columns in both arguments X and Y—assert(X.numC == Y.numC). When both conditional expressions in an assert statement are true, the algorithm indicates the concatenation for three elements of the two arguments, the public vectors $\vec{R}$ and $\vec{C}$, and the private vector $[\![\vec{W}]\!]$. The next step is assigning the three vectors with their sizes in a Struct sparse XY, finding the first normalization of the XY. Later on, finding the second normalization, which has the getMin-function that will find the minimum values for both concatenated arguments.

7.3.2. Main computation

We begin with the input to Algorithm 32, and the given adjacency matrix should be represented in a sparse representation A, as mentioned above. All elements in Struct A are pubic except weights $A.W$ is private. The arguments $\overrightarrow{ST}$ and $\overrightarrow{BS}$ comes from prerequisite

computation (R-function). In obtaining $\overrightarrow{ST}$, the set of R should be obtained by the vertices of separator tree $S_{h,k}$ that their vertices are not in that level of h. Later, obtaining the indices of the R elements. Finally, accounting for the number of the elements with the same indices of a level h, we obtain the $\overrightarrow{ST}$. In the case of $\overrightarrow{BS}$, given the list of separator trees, the R-function accounts for the number of vertices S in each separator tree that their vertices are not in R to determine the blocks of diagonal matrices.

For the three arguments whose initial value is zero, *cyc* is a counter for the recursive iteration, M indicates the range of Block diagonal matrices for each iteration, and $cont_1$ indicates the indices of the Block diagonal matrices. The argument of *Level* (h) represents the number of iterations in the main program and the number of portioning levels in the prerequisite computation of the separator tree. The last argument is a Struct of $v1''$, which carries the graph's algebraic shortest path that will be updated in each iteration.

The return value of the main computation of the algebraic path is the shortest distance for all vertices from the source vertex. In general, the algorithm provides a solution of a linear system $Ax = b$ with a sparse $n \times n$ symmetric positive matrix A. We replace vector b in the equation by the initial value for the shortest distances vector v, and the solution is $x = A^{-1}v$, ending up by shortest distances located in Struct v. The main computation of the algebraic path has a recursive (lines 1-28) and contains different related functions. Performing the recursion is while the condition is satisfied, which is the value of level h, thus requiring $O(\log^2 n)$ time.

The algorithm computes the recursive factorization in each recursive cycle that will return four matrices. The first matrix, X, will be used as an argument to find the block diagonal matrix of X with its matrices blocks; it computes X_h^*. The next step is the summation of two matrices, Y and X_h^*, obtaining the W_h matrix. The components of Struct W_h will be swapped into res_2, the vector columns $W_h.C$ into $res_2.R$, the vector rows into $res_2.C$, number of rows into several columns, and number of columns into several rows, $res_2.numR = W_h.numC$, $res_2.numC = W_h.numR$, respectively. No change in weights private vector, $res_2.W = W_h.W$.

The algorithm performs a First-normalization for Struct res_2. Another summation will be performed to the Struct W_h with a transpose matrix of Y; it returns res_3. To obtain the matrix A_h that will be used in the next recursive cycle, the minimum of two Struct Z and res_3 will be performed by Min-sparse.

The algorithm builds two matrices, U and L, based on the size of matrix A in each recursive cycle. The matrix U localizes the matrix res_2 in its upper right quadrant, While matrix L localizes the matrix W_h in its lower left quadrant. The remaining elements in matrices U and L are "∞", while the diagonals are 0's; matrices are represented in sparse representation.

One of the secondary related functions is getSlice; this function is to reshape the struct of the shortest path $v1'$ and $v1''$ that will be used in next operations. The first argument $v1$ is the struct of the shortest path that will be reshaped. The second and third arguments are rows-to-remove and cols-to-remove, respectively. The fourth and fifth arguments are rows-to-keep and cols-to-keep, respectively.

The new elements of rows $\vec{R}$ is $A.R_i$ - rows-to-remove, while the vector $\vec{C}$ is $A.C_i$ - cols-to-remove. The private vector $[\![\vec{W}]\!]$ gets its elements based on indices' values of i. Those three vectors will be constructed if the conditional expression is set to true; the condition is two parts, over rows and columns. In detail, $A.R_i \geqslant$ rows-to-remove & $A.R_i <$ rows-to-remove + rows-to-keep, the second part of the condition is over columns, similar to the rows. The getSlice-function provides $v1'$ and $v1''$, the Struct $v1'$ will be summed

with Xh^* to get $v2^{'}$. Those intermediate shortest paths $v1^{''}$ will be used as an argument in the recursive call, and $v2^{'}$ will be used as an argument in the overlap-function after the recursive call.

Algorithm 32: Main computation of Algebraic paths

Data: Struct sparse A, $\overrightarrow{ST}$, $\overrightarrow{BS}$
Data: $level(h)$, struct $v1^{''}$
Data: $cyc = 0$, $M = 0$, $cont_1 = 0$
Result: shortest paths Struct sparse v

1 **Function** Algebraic-paths$(A, \overrightarrow{ST}, \overrightarrow{BS}, cyc, M, cont_1, level, v1^{''})$ **is**
2 **if** cyc ! = $level$-1 **then**
3 $rang = 0$
4 Struct sparseF F = Factorization$(A, ST[cyc])$
5 **while** $ST[cyc]$! = $rang$ **do**
6 $rang$ = $rang$ + $BS[cont_1{+}{+}]$
7 $cont_2{+}{+}$
8 sparse Xh^* = Block-Diagonal-Matrix-inv$(F.X, BS[M : M + cont_2])$
9 $M = M + cont_2$
10 $cont_2 = 0$
11 Wh = Sum-sparse$(F.Y, Xh^*)$
12 sparse res_2 = Wh; // res2.R = Wh.C & res2.C = Wh.R
13 res_2 = First-normalization(res_2)
14 sparse res_3 = Sum-sparse$(Wh, F.Y^T)$
15 sparse Ah = Min-sparse$(F.Z, res_3)$
16 sparse U = getUpper$(A.numR, res_2)$
17 sparse L = getLower$(A.numR, Wh)$
18 sparse $v1$ = Sum-sparse$(v1^{''}, U)$
19 $row = F.X.numR$
20 $col = F.X.numC$
21 sparse $v1^{'}$ = getSlice$(v1, 0, 0, 1, col)$
22 sparse $v1^{''}$ = getSlice$(v1, 0, col, 1, v1.numC\text{-}col)$
23 sparse $v2^{'}$ = Sum-sparse$(v1^{'}, Xh^*)$
24 $cyc{+}{+}$
25 sparse $v2^{''}$ = Algebraic-paths$(Ah, [ST], [BS], cyc, M, cont_1, level, v1^{''})$
26 *sparse* $v2$ = overlap(overlay$(v2^{'}, 0, v1.numC$ - $col)$, overlay$(v2^{''}, col, 0)$)
27 sparse v = Sum-sparse$(v2, L)$
28 **return** v
29 $[B] = A.numR$
30 sparse A^* = Block-Diagonal-Matrix-inv$(A, [B])$
31 sparse v = Sum-sparse$(v1^{''}, A^*)$
32 **return** v

The overlap-function has the same functionality and structure as Min-sparse. The difference is that it has no Second-normalization and assumes that no position X and Y have non-INF simultaneously. This function has two arguments, which are the interme-

diate shortest paths. The two arguments must be modified before being carried out to the overlap-function.

The third secondary related function is overlay that increases the size of the intermediate shortest paths $v2'$ and $v2''$ in terms of the columns $\vec{C}$ and the number of columns *.numC*. The aim of increasing the size is to make it the same as the other arguments in the overlap function. In detail, the second and third arguments in the overlap function will be summed with $v2.numC$, and the second argument will be summed with $v2.C$. No change in rows $\vec{R}$ and weights $[\![\vec{W}]\!]$.

The shortest path v is returned value of the last Sum-sparse inside the conditional expression that summed Struct L and the intermediate shortest path v. If the conditional expression is set to false, we define a single block (line 29) which will be carried to the Block-diagonal-matrix function to find the A^*. The shortest path v is returned value of the Sum-sparse which is out of conditional expression, the sum is $v1''$ with A^*.

7.4. Optimized Bellman-Ford public edges protocol

Chapter 4 presented the privacy-preserving single-source shortest distances protocols on private dense and sparse graphs. The three elements of the given graphs, vertices V, edges E, and weight W, are private. The operations over private elements, either integer and Boolean values or vectors, have high computational costs. In most applications that use shortest paths algorithms, the application is privacy-preserving if the weights—the most critical elements in a given graph—are private as input and output to the arithmetic black box.

Algorithm 33: Bellman-Ford (Version 3)

Data: Number of vertices and edges n and m
Data: Public Sources $\vec{S}$ and targets $\vec{T}$
Data: Private weights $[\![\vec{W}]\!]$
Data: starting vertex s
Requires: $[T]$ is sorted
Result: Private distances $[\![\vec{D}]\!]$ from vertex s

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

The vertices and edges can be public elements in a privately given graph. For example, the navigation on city streets and the layout of the streets is available to the public [169], and in shortest paths and distances with differential privacy [150], weights are only private. Even if the given edges and vertices are public, the computation is still privacy preservation, with low running time than the same private computation using complete

private graphs—edges, vertices, and weights are private. To use an efficient protocol in solving such a problem, we propose a version of the privacy-preserving Bellman-Ford protocol with public edges, presented in Algorithm 33.

In general, Version 3 of the Bellman-Ford protocol has the same algorithmic structure and functionality as Algorithm 11. The difference between them is that we replaced PrefixMin2 by getMin, both functions have the same functionality with a difference that getMin-function deals with public edges m and vertices n. This is the reason why getMin-function is faster than the two versions of the prefixMin2-function (Algorithm 12 and Algorithm 14). The second difference is that there is no use for Laud's protocol with its functions, prepareRead and performRead; this will reduce the round complexity a bit and provide more detail about the reduction in round complexity of the Version 3 in Sec 7.5. The data input is three vectors of a graph $[\![\mathbf{G}]\!]$, the source $\vec{S}$ and target $\vec{T}$ vertices are public, while weights $[\![\vec{W}]\!]$ of edges is private. The vector $\vec{T}$ should be sorted, and then the input of all vectors should be sorted according to $\vec{T}$. Then, continue regularly performing the computation, such as in the Algorithm 11.

7.5. Complexity of algorithms

This section discusses the protocol's performance for finding the shortest paths in the algebraic path computation technique and related algorithms. The complexity of the algorithms has two sides, round and communication complexities. Let n denote the number of vertices in the given graph, while m is the number of edges.

7.5.1. Round complexity

The main computation of the algebraic path has no iteration control structure, while it has recursive iterations, and the related algorithms call in each iteration. First, the round complexity of the main computation requires $O(\log^2 n)$. Second, the related functions will be executed during each iteration, while each has round complexities separately. Some of these secondary functions have zero round complexities, getSlice, overlap and overlay. As well as, the getUpper and getLower functions require zero round complexities. Each iteration in a recursive call has the following round complexities:

The first related function is recursive factorization which has zero round complexity. The algorithm splits the Struct A into four matrices in sparse representation. The public operations have zero round complexity, and assigning the private vector weights $[\![\vec{W}]\!]$ into four sub-vectors is done in parallel, with zero round complexity.

The Block-diagonal-matrix-inv function has zero round complexity for all **for**-loops. The function also has a subroutine of FloydWarshall-nSIMD, which processes t block matrices simultaneously; hence the number of blocks does not influence the round complexity because all blocks are handled in parallel. We suppose k is the size of the largest block. Thus, total round complexity of FloydWarshall-nSIMD function is $O(k)$.

It is important to present the complexities of First- and Second-normalization functions before the Min-sparse and Sum-sparse functions. The round complexity of the First-normalization is zero, the whole operations are public, and assigning the private vector has zero round complexity. The Second-normalization has getMin-function as subroutine, which has $O(\log n)$ round complexity. Thus, each Second-normalization call has $O(\log n)$ round complexity.

The Sum-sparse function has assigning operations for private vectors, which requires zero round complexity, and it has Second-normalization that requires $O(\log n)$ round

complexity. Thereby, each Sum-sparse in one recursive cycle in the main computation has $O(\log n)$ round complexity. The Min-sparse requires the same round complexity as in Sum-sparse, which is $O(\log n)$. Both have the same algorithmic structure regarding public and private operations and subroutines. The round complexities for each iteration of privacy-preserving APC and Bellman-Ford V3 protocols are presented in Table 4.

Table 4: Round and communication complexities for each iteration of privacy-preserving APC, and Bellman-Ford V3 protocols.

Protocol		**Complexity**	
		Round	Communication
APC	getSlice	Zero	Zero
	overlap	Zero	Zero
	overlay	Zero	zero
	getUpper	Zero	Zero
	getLower	Zero	Zero
	recursive factorisation	Zero	Zero
	Block diagonal-matrix-inv	Zero	Zero
	Floyd-Warshall-nSIMD	$O(k)$	$O(k^3t)$
	First-normalization	Zero	Zero
	Second-normalization	$O(\log n)$	$O(m)$
	Sum-sparse	$O(\log n)$	$O(m^2)$
	Min-sparse	$O(\log n)$	$O(m)$
Bellman-Ford V3		$O(\log n)$	$O(n^3)$

7.5.2. Communication complexity

Let n denote the number of the vertices in the given graph $[\![\mathbf{G}]\!]$, and m is the number of edges. In the algebraic path computation, we use the First-normalization to sort the elements. We also use Second-normalization to remove the occurrence of the duplicate cells; this reduction can help the implementation carry a big graph. We consider e the number of edges in a graph used in algebraic path computation protocol, and v is the number of vertices.

The functions that require zero round complexity also require zero communication—no communication had occurred among the computation parties of the SMC platform. These functions are Recursive-factorization, First-normalization, getSlice, overlap and overlay, getUpper, getLower and Block-diagonal-matrix-inv.

Initially, the size of the given adjacency matrix is $n \times n$, which is represented sparsely into vectors; the size of each vector is m. The communications of the main computation (Algorithm 32) requires $O(m\log^2 n)$. Each iteration in a recursive call has the following communication complexities:

The Block-diagonal-matrix-inv function has FloydWarshall-nSIMD function, which processes t blocks matrices simultaneously, the size of the largest Block is also k. The total communication is $O(k^3t)$. The Second-normalization function has getMin as a subroutine, the communication requires $O(m)$. The Sum-sparse in one recursive cycle in the main computation requires $O(m^2)$ communication. The Min-sparse function has $O(m)$ communication. The communication complexities for each iteration in the main computation of privacy-preserving APC protocol are presented in Table 4.

7.6. Security and privacy of protocols

As we mentioned in Sec 4.7, the protocol is built on top of a universally composable ABB. It inherits the same security properties against various adversaries as the underlying secure computation protocol. The privacy-preserving algebraic path parallel computation protocol and its related functions are privacy-preserving since they do not contain any declassification statements. The given graph in the implementation has public edges, while the weights are only private. The private result is no longer determined by the private values of all elements in a graph; some can be public (edges and vertices) and hence do not leak the privacy preservation. The private values in the algebraic path computation protocol have no declassification statements. Therefore our implementations are privacy-preserving.

8. EXPERIMENTS OF SHORTEST PATH PROTOCOLS

8.1. Introduction

This chapter presents the benchmarking results and empirical analysis of the shortest path protocols. Firstly, it shows the investigations and analysis of all privacy-preserving single-source shortest path protocols. Secondly, presenting the experiments and analysis of the privacy-preserving All-pairs shortest path protocols. Benchmarking the results with previous works is also given. We perform the tests and experiments on the Sharemind system over different network environments. The set-up experiments for all protocols, hardware, and network architectures can be seen in Sec 3.6.

8.2. Single-source shortest paths experiments

8.2.1. Benchmarking results for single-source shortest path

There have been no extensive benchmarking studies of privacy-preserving shortest-path protocols. The outcomes still need to be stronger because the proposed protocols are sequentially constructed, which requires high round complexities among the computational parties; such protocols do not lead to building efficient real-world applications.

Aly et al. [5] have benchmarked their implementations of Dijkstra's and Bellman-Ford algorithms on dense representations of small graphs. They implemented the algorithms on top of VIFF [78] with BGW protocol [84] used for multiplication and Toft's protocol [160] for comparison. They claim a runtime of little more than an hour for Dijkstra's algorithm on 128 vertices and more than 8 hours for the Bellman-Ford algorithm on 64 vertices.

Aly and Cleemput [3] benchmark their implementation of Dijkstra's algorithm on graphs of up to 64 vertices, reporting running times in a range of 20 seconds. On a 32-vertex graph, the running duration is roughly 5 seconds. The permutation procedure is also employed to disguise the vertices' true identities, increasing the total running time. It is an excellent technique to maintain privacy but also an extra operation that will lengthen the computation's overall run time.

Keller and Scholl [104] have implemented the operations of oblivious RAM (ORAM) on top of the SPDZ protocol set [63] and used them to implement a privacy-preserving version of Dijkstra's algorithm. Cycle graphs of ca. 2000 vertices that are represented sparsely report the running times in a few minutes. Their implementation requires a couple of hours for graphs with 500 vertices represented densely.

Carter et al. [50] use garbled circuits (hence removing considerations about round complexity but consuming more communication) to evaluate Dijkstra's algorithm privately. They report a running time of 26 seconds for a 20-vertex graph and ca. 15 minutes for 100-vertex graphs—their parallel implementation handles 32 circuits simultaneously on a 32-core server. The 64-circuit evaluation requires almost 50 seconds for the 20-vertex graph and nearly 22 minutes for a 100-vertex graph.

Similarly, Liu et al. [125] make use of garbled circuits to evaluate Dijkstra's algorithm on sparse graphs, employing oblivious priority queues [165] to increase efficiency. The estimate is that together with JustGarble [16], their running time on a 1000-vertex, 3000-edge graph is maybe 20 minutes. In [25], theoretical work has been proposed, but no evaluation is reported.

8.2.2. Dijkstra's protocol experiments

Dijkstra's protocol for single graph

The running time of our privacy-preserving Dijkstra's protocol depends only on the number n of the vertices of the input graph, while there is no influence on the number of edges m. We report the running times in Table 5 for various values of vertices n—different sizes of graphs. For interest and similarly to [3], for some of the instances, we report separately the time it takes to permute the vertices and the time it takes to execute the main loop of Alg 7. This split is only informative for most use cases because all main loop iterations have to be executed to find the shortest distances to all vertices. It may be helpful only if we are interested in the shortest path from the source vertex s to some target vertex t, and we somehow know that t is one of the closest vertices to s. Again, the running times are given for the HBLL environment.

In different network environments, we expect Dijkstra's protocol to behave similarly to the Bellman-Ford algorithm—it will also be latency-bound to an even greater extent.

Table 5: Running times (in seconds) of privacy-preserving Dijkstra's protocol.

	Graph		**Dijkstra**		
	n	**m**	Perm.	Loop	Total
Sparse	10	25	0.01	0.08	0.09
	20	100	0.02	0.016	0.18
	700	3k	8.2	29.0	37.2
	900	244k	13.3	40.2	53.5
	8.5k	200k	1176	3230	4406
	9.5k	200k	1441	4014	5456
Dense	50	1225	0.09	0.48	0.57
	64	2016	0.12	0.69	0.81
	85	3500	0.19	1.02	1.2
	150	11k	0.5	2.5	3.0
	300	44.8k	1.63	6.42	8.1
	450	100k	3.43	13.7	17.1
	700	244k	7.94	29.3	37.2
	2k	1.9M	57.5	196	253
	3k	4.4M	137	479	617
	4.5k	10M	312	1006	1319
	5k	12.4M	380	1196	1577
	7k	24M	745	2266	3012
	10k	49.9M	1572	4488	6061
	15K	112M	3601	9807	13.4k

In Figure 6, we establish the baseline for our experiments by measuring the running time of Dijkstra's protocol on graphs of various sizes in different network environments. For the benchmark, we use both small and relatively large graphs, and the number of edges in the graph is not a determining factor. These running times are for finding the privacy-preserving SSSD in a single graph. We observe that the performance is highly latency-bound, meaning that the available data volume does not significantly affect the performance on most graphs in high-latency environments.

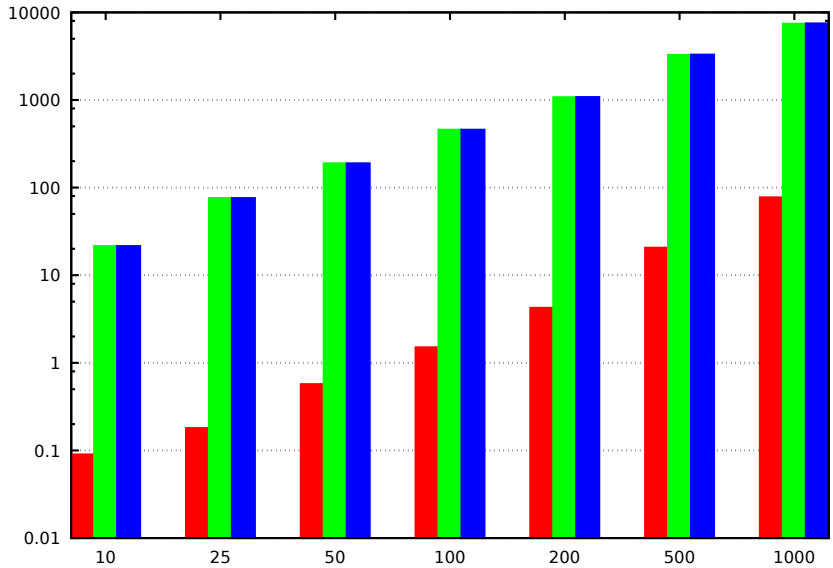


Figure 6: Performance of Dijkstra's algorithm on graphs with given numbers of vertices in different network environments (red: HBLL, green: HBHL, blue: LBHL).

Dijkstra's protocol for multiple graphs

Sequentially, we can run version 1 of privacy-preserving Dijkstra's protocol several times over graphs of the same size. In contrast, version 2 of the privacy-preserving SSSD Dijkstra's protocol executes several graphs of the same size in parallel.

The running times of privacy-preserving nDijkstra's protocol (version 2) using different Networks are presented in Table 6. In the benchmark, we run different kinds of graphs: dense and sparse, with different sizes: big and small. The preferable network case (High-Bandwidth, Low-Latency) is presented in Table 6 with other network environments.

The performance gains resulting from parallel execution over multiple graphs should be meaningful and comparable over various graph sizes, several graphs, and network environments. We mean in different graph sizes is the number of the vertices n regardless of their connected edges m.

To estimate how scalable the parallelization is, we use the Karp-Flatt metric, considering the number of graphs k, the graph size n, and network environments. Suppose the time to execute the parallel nDijkstra's protocol on one graph is T_1. The time to execute the same algorithm on k graphs is T_k. In that case, both implementations have the same graph size on the same network environment. Consequently, the parallel gains are maximal if $T_k \approx T_1$, while it is minimal if $T_k \approx k.T_1$. We then map the gains to the segment $[0, 1]$.

Then, the *serial fraction* is given by $F_k = (T_k/T_1 - 1)\ /\ (k - 1)$. It presents the measurement of the serial fraction when running the parallel nDijkstra's protocol for different numbers of the graphs k and different sizes of the graphs n and over different network environments, see Figure 7. The parallelization is perfectly scalable if the number of graphs k grows at a rate no more than the number of vertices n in a graph. In contrast, parallelization is not scalable if the number of graphs k grows more than the number of vertices n in a graph. The data underlying this Figure 7 comes from Table 6.

Table 6: Benchmarking results for the parallel execution of nDijkstra's protocol on several graphs of the same size in different network environments.

Num. of graphs	Size of graph	Running time (s)		
		HBLL	HBHL	LBHL
1	10	0.09	21.6	29.3
10	10	0.2	54.1	54.1
1	25	0.23	73.4	100
25	25	1.0	158	160
1	50	0.53	166	228
5	50	1.6	326	326
10	50	2.6	329	331
25	50	3.9	336	341
50	50	6.3	371	388
1	100	1.29	373	513
10	100	7.5	745	755
25	100	16.1	764	792
50	100	28.6	793	852
75	100	42.6	834	920
100	100	42.3	898	1019
1	200	3.37	828	1144
20	200	46.0	1699	1792
50	200	104	1800	2075
100	200	189	2138	2787
200	200	337	2657	3669
1	500	12.5	2884	4042
100	500	1049	7524	10.8k
500	500	3715	17.2k	34.6k
1	1000	46.1	6356	8933
50	1000	2129	15.7k	21.0k
500	1000	21.1k	62.2k	125k
1000	1000	42.5k	109k	235k
1	5000	853	40.8k	61.2k
100	5000	230k	390k	648.9k

We have not tested the parallel execution for several graphs larger than the number of vertices in a graph because this case will not be needed in any sensible executions of Johnson's protocol. As expected, we see the most significant gains for the HBHL environment. Interestingly, the largest gains are obtained for graphs with ca. 200 vertices, at least for high-latency environments. This may mean that the parallelization possibilities for computations with a single graph for larger graphs are already significant.

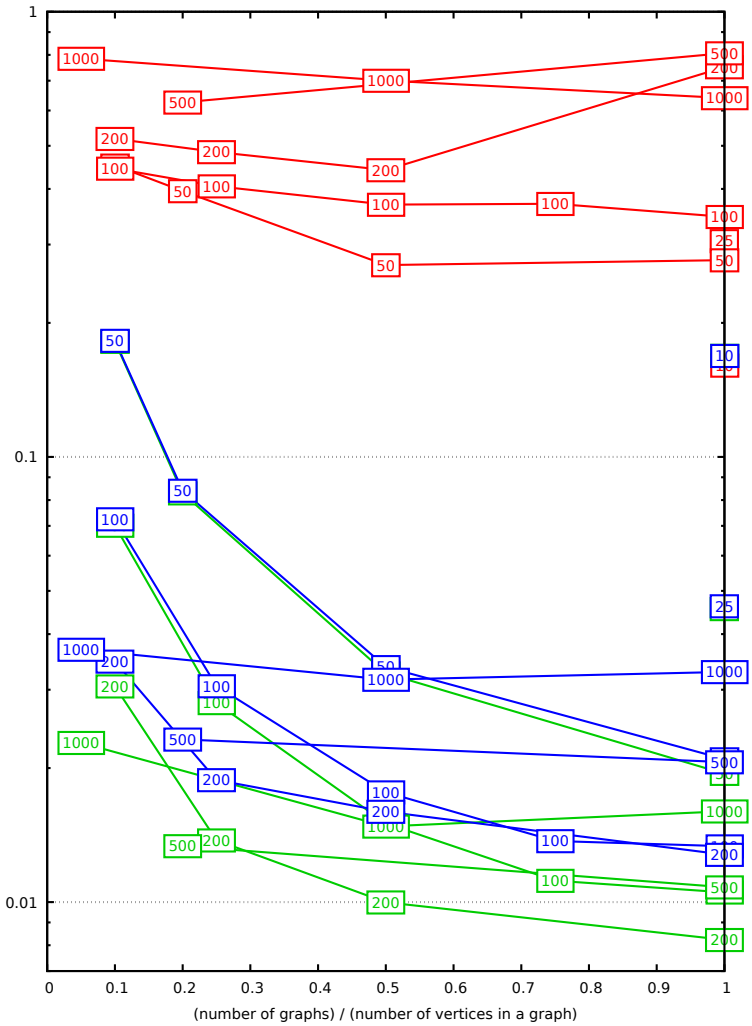


Figure 7: Dijkstra's algorithm performance (as a serial fraction; lower is better) on multiple graphs of various sizes (number of vertices given on the graph) in different network environments (red: HBLL, green: HBHL, blue: LBHL).

8.2.3. Bellman-Ford protocol experiments

We ran the codes for the two versions of the Bellman-Ford protocol and used different graphs with different sizes in both versions. The execution time of our privacy-preserving Bellman-Ford protocols depends only on its public inputs, i.e., the number of vertices n and the number of edges m of the given graph $[\![\mathbf{G}]\!]$. The given vertices, edges, and weights are private in both versions of Bellman-Ford. The main difference between the two versions of Bellman-Ford protocols is a subroutine which is prefixMin2. Version 1 of the algorithm uses the implementation of prefixMin2 shown in Algorithm 12, while Version 2 uses the implementation shown in Algorithm 14. The algorithm executions are made in the *high-bandwidth low-latency* environment. The running times for various sizes of the input graph are in Table 7.

Table 7: Running times (in seconds) of privacy-preserving Bellman-Ford protocol.

Graph			BF Version 2		BF Version 1	
n	m	Pre.	Loop	Total	Loop	Total
10	25			0.18		0.27
20	100			0.53		0.70
50	400			2.68		2.70
85	1.2k			11.3		8.10
170	2.5k			37.8		25.8
350	1050	9.4	42.7	52.1	27.5	36.9
350	2k	9.4	68.9	78.3	41.6	51.0
500	1.5k	18.9	87.3	106	54.1	73.0
500	5k	19.7	195	214	121	140
700	2.1k	37.3	165	202	109	146
700	10k	38.2	476	514	291	329
3k	9k	663	2541	3204	1545	2208
3k	50k	676	10511	11187	5415	6091
4.5k	13.5k	1515	5830	7345	3239	4754
4.5k	100k	1.5k			16.2k	17.7k
7k	21k	3.6k			7.7k	11.3k
7k	200k	3.6k			46.6k	50.2k
8.5k	25.5k	5.2k			11.2k	16.4k
8.5k	300k	5.3k			81.7k	87k
9.5k	28.5k	6.6k			12.9k	19.4k
9.5k	500k	6.6k			144k	151k

The execution of the Bellman-Ford protocols consists of preparatory steps for subsequent array accesses according to private indices, followed by the algorithm's main loop executed at most $(n\text{-}1)$ times. In Table 7, we report separately the time it took to run the preparatory steps of Algorithm 11 (everything up to the main loop), as well as the main loop (all $(n\text{-}1)$ iterations). Moreover, the total execution time for every version is reported. The preparatory steps (the part before the main loop) is similar for both versions.

As part of more extensive applications, we may be interested in executing the Bellman-Ford protocol for less than $(n\text{-}1)$ iterations. We know from the context that fewer iterations are sufficient, or it is acceptable to leak either the precise number of iterations or some padded version. Given the number of vertices and edges and the expected number of iterations, Table 7 can be used to find the scheduled running time. The running time of the preprocessing must be added to the running time of the main loop, where the fraction is equal to the proportion of iterations to vertices.

Both dense and sparse graphs were taken into consideration when benchmarking. Planar graphs are expected to appear in various applications as a class of sparse graphs. Hence half of our tests have been run with graphs whose number of edges matches that of typical planar graphs having mostly triangles as faces—we let the number of edges be thrice its number of vertices.

Version 1 of our implementation of the Bellman-Ford protocols consumes less communication, while Version 2 has better round complexity. This difference can be seen in Table 7, where either implementation may have a shorter running time for certain sizes of inputs. We have benchmarked both versions of the Bellman-Ford protocol in different

network environments for graphs of various sizes. We give both the running time and data volume consumption in Table 8. We also depict the running time in Figure 8.

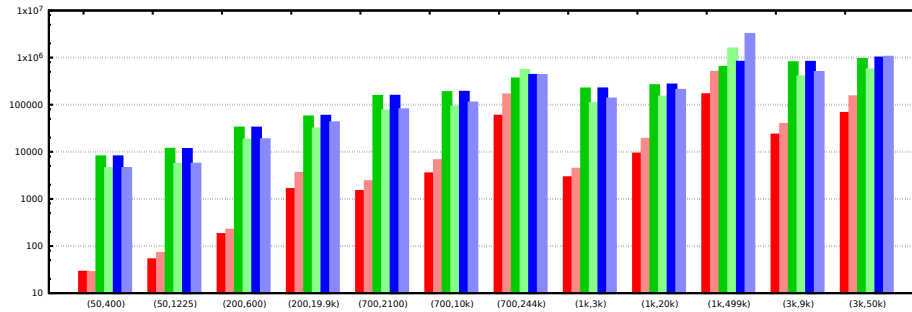


Figure 8: Bellman-Ford algorithm performance (time in seconds) in different networks for different (n, m) (red: HBLL, green: HBHL, blue: LBHL, dark: Version 1, light: Version 2).

In particular, we have focused primarily on the functional aspects of networks for running our privacy-preserving Bellman-Ford protocols. We tested our work over different networks considering the two fundamental measuring tools, data volume and running time. In Table 8, we have implemented the two versions of privacy-preserving Bellman-Ford protocols over three main networks. In the implementations, we run different graphs: sparse and dense, with different sizes. We documented the results for Bandwidth and Latency over the three servers of the SMC Sharemind system for the versions of Bellman-Ford. Note that the running time of the three servers is similar; they start and finish together.

Table 8: Benchmarking results (data volume for a single computing server) for Bellman-Ford algorithms in different network environments.

Size of graph		**Version 1 (with Alg. 12)**				**Version 2 (with Alg. 14)**			
		Data	**Running time (s)**			**Data**	**Running time (s)**		
n	m	**volume**	**HBLL**	**HBHL**	**LBHL**	**volume**	**HBLL**	**HBHL**	**LBHL**
50	400	32 MB	2.8	799	799.5	65 MB	2.8	448	451
50	1225	74 MB	5.2	1155	1145	206 MB	7.2	550	559
200	600	165 MB	17.9	3270	3261	465 MB	22.2	1823	1841
200	19.9k	2900 MB	162	5626	5806	15.8 GB	356.2	3122	4219
700	2100	1570 MB	147	15.4k	15.4k	6.3 GB	237.7	7529	7956
700	10k	5 GB	348	18.5k	18.7k	27 GB	661.7	9158	11.1k
700	244k	110 GB	5823	35.9k	42.9k	810 GB	16.4k	54.2k	111k
1k	3k	4 GB	288	22k	22.2k	12.7 GB	435.8	10.8k	13.5k
1k	20k	13 GB	917	25.9k	26.7k	90 GB	1879	14.8k	20.6k
1k	499k	320 GB	16.6k	63.4k	81.7k	2.4 TB	49.8k	156k	318k
3k	9k	65 GB	2318	79.6k	81.1k	145 GB	3889	39.8k	49.3k
3k	50k	133 GB	6675	93.3k	100k	630 GB	15k	56.3k	104k

As we explained in Sec 4.3, privacy-preserving Bellman-Ford Version 1 is efficient in terms of running time, while Version 2 is efficient in terms of communication. In Table 8, the running time for all graphs in Version 1 is lower than Version 2 of Bellman-Ford (in the preferable case HBLL), while the communication for Version 2 is higher than Version 1. We notice the change in latency and Bandwidth for different graphs over

different instances of the networks. Those results are practical proof of our methodology in implementing different parallel algorithms in finding the prefix minimum.

The timing results clearly show that the two versions of the protocol behave very differently in various network environments. While Version 1 of the protocol is always better in the HBLL environment, it depends on the number of edges of the graph for the environments with high latency. Indeed, the number of edges essentially determines the parallelism available for the algorithm; a small number of edges makes Version 2 preferable because of its smaller round complexity.

The timing results also show that for all sizes of the graph shown in Figure 8, the algorithm's performance is essentially bounded by the latency of the network. Indeed, for smaller examples, we see very little difference between the HBHL and LBHL environments. Also, even for the largest examples, there is still a very significant difference between the HBLL and HBHL environments.

8.2.4. Radius-stepping protocol experiments

We have implemented the shortest path algorithm on the SMC Sharemind platform for both versions of the protocol, sequential and parallel SIMD. We use speed-up to measure the relative performance of parallel and sequential versions of the proposed protocol [147]. We find the speed-up of our parallel radius-stepping protocols on top of SMC protocols by dividing the standard version of the radius-stepping by the execution time of the parallel version (SIMD). The formula is $\textit{Speed-up} = \mathcal{T}_{\mathcal{S}}/\mathcal{T}_{\mathcal{P}}$, where $\mathcal{T}_{\mathcal{S}}$ is the execution time of the sequential protocol, and $\mathcal{T}_{\mathcal{P}}$ is the execution time of parallel SIMD version of the radius-stepping protocol.

Table 9: Running times (in seconds) of privacy-preserving Radius-Stepping protocols for sparse and dense graphs.

	Graph		**Radius-Stepping**		
	n	m	Serial	Parallel	Speed-up
Sparse	200	1500	914.7	20.5	44x
	400	4k	6786	97.2	69x
	500	6k	13062	203	64x
	500	10k	35346	186	189x
	900	20k	77580	825	94x
	1k	40k	–	920	–
	2k	40k	–	10206	–
Dense	10	45	0.8	0.1	8x
	35	400	20.6	0.5	41x
	50	1225	74.6	0.6	124x
	75	2775	247	1.2	206x
	100	4950	593	1.5	395x
	300	44850	16059	19.4	827x
	500	124.7k	74453	78.6	947x
	1000	499.5k	–	188.4	–

We use different weighted undirected graphs in our implementation, sparse and dense, with varying sizes. We also used unweighted graphs with different sizes in the implementation. The execution time of our privacy-preserving SIMD-Radius-Stepping protocol depends on the number of vertices and the edges of the input graph. We report the run-

ning time in Table 9 for several sparse and dense graphs of different sizes for the "∞"-radii case.

We ran two versions of codes, sequential and parallel (SIMD), while both used a private access memory for privacy preservation. The speed-up of the first graph $[\![\mathbf{G_1}]\!]$ is around 44 times—the parallel version is faster than the sequential version. The speed-up of the others graphs $[\![\mathbf{G_2}]\!]$, $[\![\mathbf{G_3}]\!]$, $[\![\mathbf{G_4}]\!]$, and $[\![\mathbf{G_5}]\!]$ are 69x, 64x, 189x, 94x, respectively. The result shows that the speed-up is increased by increasing the input graph size, and the SIMD-Radius-stepping algorithm's speed-up is scalable. The aim of presenting a sequential version of radius-stepping in privacy preservation is to analyze the parallel method we use and how much it makes a difference in the latency of the network. One of the most impressive uses of our proposed protocol is finding the privacy-preserving shortest path for dense graphs. The results show that the speed-up for parallel implementation of the dense graph may be hundreds of times. The graph with 300-vertex and 44850-Edge has a speed of 827 times faster than the sequential version of the algorithm. It is also important to note that reporting the speed-up for a huge graph is difficult because the running time of the sequential version of the protocol is too high. We estimate the running time of the sequential version of the protocol to be in a few days or weeks on the SMC platform. This is the motivation of our work; by using our SIMD-Radius-Stepping, we can find the shortest path for vast graphs in privacy-preserving parallel computation in less running time compared with the running time of the sequential version of the protocol.

Table 10: Running times of SIMD-radius-stepping for planar-like graphs.

	Graph		**Radius-Stepping (planar-like)**		
	n	m	Delta	Stand	SIMD
E = 2V	20	40	2.28	1.62	0.24
	40	80	6.84	8.4	0.54
	80	160	89.2	54.6	4.6
	120	240	254	170	13.4
E = 2.5V	20	50	5.28	1.74	0.24
	40	100	20.7	9.42	0.84
	80	200	113	56.7	5.6
	120	300	296	178	9.48
E = 4V	20	80	10.1	2.28	0.24
	40	160	30.9	10.7	0.9
	80	320	109	66.8	3.78
	120	480	360	192	9.5

Another series of tests that we report in Table 10 considers graphs whose number of edges (for the given number of vertices) is similar to planar graphs. This series implemented three privacy-preserving single-source shortest path protocols: the Radius-Stepping and Δ-Stepping, and our SIMD-Radius-Stepping. There are three groups of graphs that have been benchmarked. The groups are based on the ratio between the number of edges and the number of vertices of the graph. In the first group, the number of edges is two times the number of vertices. In the second group, the number of edges is 2.5 times the number of vertices, being close to the maximum number of edges a planar graph may have. The graphs of the last group are denser than planar graphs. The running times of the standard Radius-Stepping protocol are more minor than the running times of

the Δ-Stepping protocol, while the SIMD-Radius-Stepping protocol is less than both.

We tested our protocol also for unweighted graphs with "∞"-radii case. Table 11 shows the running times for SIMD-Radius-Stepping protocols on several unweighted graphs of different sizes. The result shows our protocol has an enormous increase in speed-up for finding the shortest paths in unweighted graphs compared with the speed-up for finding shortest paths in weighted graphs. For example, the graph with 50 vertices and 1225 edges has a 124x speed-up for the weighted graph and a 359x speed-up for the unweighted graph.

Table 11: Running times (in seconds) of privacy-preserving Radius-Stepping algorithms for unweighted graphs.

Graph		**Radius-Stepping (unweighted)**		
n	m	Serial	Parallel	Speed-up
25	100	5.1	0.15	34x
25	300	8.7	0.1	87x
50	1225	71.8	0.2	359x
100	2k	357	0.7	510x
200	19.9k	4769	4.3	1109x
500	20k	26372	14.8	1781x
1k	20k	–	75.9	–
5k	12.4M	–	2304	–
10k	49.9M	–	9087	–

Table 12: Running times (in seconds) of SIMD-Radius-Stepping with radii cases.

	Graph		**Radius-Stepping (radii cases)**					
			r = ∞		**r = 0**		**r = rnd**	
k	n	m	Iter	Time	Iter	Time	Iter	Time
Sparse	25	100	6	0.17	24	0.78	7	0.25
	50	400	6	0.5	44	3.9	8	0.66
	100	2k	5	1.3	41	9.8	9	2.1
	200	2k	9	8.1	100	94.5	20	17.3
	500	20k	6	29.4	121	597	27	134
	1k	200k	6	106	69	1266	19	451
Dense	50	1225	5	0.4	24	2.1	6	0.5
	100	4950	5	1.1	25	6.1	8	1.9
	200	19.9k	5	4.5	27	22.8	8	8.1
	500	124.7k	5	23.6	31	153	10	49.1
	1k	499.5k	5	87.8	38	690	15	256
Planar-like	50	100	8	0.75	78	6.9	11	0.9
	50	150	8	0.7	70	6.2	10	0.8
	100	200	9	2.3	165	40.1	23	5.8
	100	300	9	2.1	155	38.9	21	5.0
	200	400	13	11.1	380	276	40	43.6
	200	600	11	9.8	251	260	41	35.1
	500	1500	13	63.5	691	3420	100	489

Algorithm 15 has a declassification statement in a location that causes some graph details to be leaked through the algorithm's running. The running time is characterized

by the time it takes to run a single iteration of that algorithm (which only depends on the number of vertices and edges of the graph) and the number of iterations the algorithm does (which depends on the structure of the graph, as well as on the radius **r**). In Table 12, we report the average number of iterations for random weighted graphs with a given number of vertices and edges for three possible choices of the radii—either infinite, zero, or randomly generated. The average execution time has been reported, too. We see that the first choice consistently beats the other for these choices.

8.2.5. Breadth-first search protocol experiments

Similar to the implementations above, we have implemented our proposed protocol of the privacy-preserving breadth-first search with its two versions—Weighted Breadth-First Search (WBFS) and Unweighted Breadth-First Search (UBFS)—on the Sharemind SMC platform. Although our algorithm fits dense graphs, we use different types of graphs in the implementations on the top of the SMC protocol set, sparse and dense ones, with varying sizes. Among the sparse graphs, we consider graphs whose number of edges is similar to planar graphs, i.e., two or three times the number of vertices. Using different kinds of graphs is for the benchmarking of the two versions of our privacy-preserving BFS protocol. We reported the running time of our experiments in seconds. The parallel codes of the implementation are written using the SIMD framework; after vectorization of the given adjacency matrices. Besides the running time as a measurement tool for our benchmarks, we report the data volume for running the two versions of the proposed protocol.

Privacy-preserving WBFS protocol

We report the running times of the privacy-preserving WBFS's protocol for various weighted graphs, sparse, dense, and planar-like, with various sizes in Table 13. This table presents the benchmark for the privacy-preserving parallel versions of the WBFS and Radius-stepping protocols. In the implementations of the radius-stepping, we use the best two cases of the radii, which are random (rnd) and infinity ("∞"). The table shows the number of possible iterations (Iter) for each algorithm with their total running time. A function is used to generate random private graphs.

Therefore, the number of iterations in all implementations can differ based on the weights of edges and which vertices are connected by edges. The number of iterations in WBFS's protocol is smaller than in both versions of the radius-stepping protocol in all experiments. The empirical test clearly shows that WBFS's protocol is faster than the radius-stepping protocol in two cases of radii.

Moreover, Table 13 shows how much WBFS's protocol is faster than the best radii case ("∞") between 3 and 5 times. Although, the number of iterations in WBFS is one time less than that in the ∞-radii version (with some exceptions). For rnd-radii, the Improvement of WBFS's protocol is 50 times or more.

Lastly, the number of iterations running the protocol with dense graphs (regardless of size) is less than that running the protocol with sparse graphs. The number of iterations is decreased by increasing the number of edges in the given graphs $[\![\mathbf{G}]\!]$. Thus, the privacy-preserving BFS protocols are efficient over dense graphs with low running time. The protocol is still more efficient for other graphs than radius-stepping and different algorithms in previous works—Bellman-Ford, and Dijkstra protocols.

Table 13: Running times (in seconds) of privacy-preserving WBFS and radius-stepping protocols for various graphs.

Graphs			Parallel Radius				Parallel WBFS		Improvement BFS vs.	
			Radii = rnd		Radii = ∞					
k	n	m	Iter.	Time	Iter.	Time	Iter.	Time	rnd	∞
Sparse	25	100	8	0.33	6	0.24	5	0.07	5.0x	3.6x
	50	400	7	0.78	5	0.52	4	0.13	6.1x	4.1x
	100	900	14	4.67	6	2.05	5	0.47	10.0x	4.4x
	200	1k	33	41.4	11	13.8	10	2.98	13.9x	4.6x
	500	5k	38	275	12	86.9	11	21.5	12.8x	4.1x
	750	10k	58	916	9	142	8	35.4	25.9x	4.0x
	1k	10k	94	2620	14	392	13	98.9	26.5x	4.0x
	1k	40k	41	1141	8	224	7	56.6	20.2x	4.0x
	3k	100k	147	36.4k	11	2724	10	685	53.0x	4.0x
	3k	1M	44	10.9k	7	1733	6	429	25.4x	4.0x
	5k	1M	95	41.8k	7	4778	6	1198	34.8x	4.0x
	5k	5M	31	14.1k	5	3459	4	852	16.5x	4.1x
	10k	15M	51	91.3k	6	16.5k	5	4090	22.2x	4.0x
Dense	50	1225	5	0.56	4	0.43	3	0.11	5.0x	3.8x
	100	4950	8	2.67	6	1.99	5	0.44	6.0x	4.5x
	200	19.9k	9	11.3	5	6.35	4	1.45	7.8x	4.4x
	750	280k	16	254	6	94.9	5	23.5	10.8x	4.0x
	1k	499k	20	560	5	140	4	35.3	15.9x	4.0x
	5k	12.4M	21	14.4k	5	3419	4	852	16.9x	4.4x
	6k	17.9M	28	17.7k	6	5953	5	1465	12.0x	4.1x
	10k	49.9M	27	48.9k	5	13.7k	4	3377	11.9x	4.1x
Planar-like	50	100	11	1.14	9	1.01	8	0.24	4.8x	4.3x
	50	150	11	0.98	7	0.69	5	0.16	6.2x	4.4x
	100	200	21	6.93	10	3.31	9	0.72	9.6x	4.6x
	100	300	21	7.08	8	2.72	7	0.61	11.7x	4.5x
	500	1k	146	1056	11	79.9	10	19.8	53.0x	4.0x
	500	1500	105	759	15	108	14	26.9	28.2x	4.0x
	1k	2k	245	6834	14	271	13	61.5	111x	4.4x
	1k	3K	169	4713	15	285	13	61.6	77.3x	4.6x

Privacy-preserving UBFS protocol

The implementations of our proposed privacy-preserving parallel computation of UBFS's protocol for an unweighted graph are presented in Table 14. We have benchmarked the privacy-preserving UBFS protocol with a privacy-preserving radius-stepping protocol for unweighted graphs. Different kinds of graphs were used in the implementations, sparse, dense, and planar-like, with various sizes.

The running time in seconds is reported for both protocols. The benchmark of the radius-stepping protocol is noted for the ∞-radii case, which is considered the most efficient case of radii. In both protocols, the result shows the possible iterations (Iter) based on the number of given edges. The results also show the number of vertices with the exact distances (levels) for both protocols. The results generally indicate the running times of the UBFS's protocol and its improvement compared to the radius-stepping protocol. This speed-up is between 4 and 5.6 times.

The UBFS protocol has the lowest possible number of iterations in experiments with dense graphs, regardless of the size. The number of iterations is decreased by increasing the number of edges in the given graph. In the case of the maximal possible number of

edges in the dense graph, the number of iterations will be only one. This is the significance of our work; for the maximum possible number of edges for a given graph on the UBFS protocol, the round complexity is constant.

Table 14: Running times (in seconds) of privacy-preserving radius-stepping and UBFS protocols over Sharemind.

	Size		**Parallel Radius**		**Parallel UBFS**		**Level**	**Improvement**
k	n	m	Iter	Time	Iter	Time		
Sparse	50	400	3	0.32	1	0.07	3	4.8x
	100	2k	3	1.02	1	0.21	3	4.9x
	150	750	5	3.59	3	0.71	4	5.1x
	200	1k	5	6.28	3	1.25	4	5.0x
	500	20k	3	21.9	1	4.02	3	5.5x
	750	10k	4	63.5	2	13.1	4	4.8x
	1k	20k	4	113	2	23.5	4	4.8x
	1k	100k	3	84.9	1	15.5	3	5.5x
	3k	5k	3	762	1	136	3	5.6x
	10K	15M	3	8425	1	1524	3	5.5x
Dense	50	1225	3	0.33	1	0.06	3	5.2x
	100	4950	3	1.04	1	0.21	3	5.5x
	200	19.9k	3	3.78	1	0.69	3	5.5x
	500	124k	3	22.1	1	3.96	3	5.6x
	750	280k	3	48.5	1	9.01	3	5.4x
	1k	499k	3	85.1	1	15.7	3	5.4x
	3k	4.49M	3	758	1	135	3	5.6x
	5k	12.4M	3	2091	1	374	3	5.6x
	10k	49.9M	3	8312	1	1490	3	5.6x
Planar-like	50	100	6	0.62	4	0.15	6	4.2x
	50	150	5	0.55	3	0.11	5	5.0x
	100	200	6	1.95	4	0.41	6	4.7x
	100	300	5	1.72	3	0.36	6	4.8x
	500	1k	10	72.6	8	18.1	10	4.0x
	500	1.5k	6	43.5	4	10.0	6	4.0x
	1k	2k	9	250	7	62.4	9	4.0x
	1k	3k	7	195	5	49.9	7	4.2x

The performance of the secure multiparty computation protocols feature is measured by two parameters—the round complexity and the data volume. Table 15 presents the running time and the data volume for two versions of the privacy-preserving BFS protocol with different data inputs. We performed the computation over the preferred network with high bandwidth and low latency. It gives the running time, data volume for each round (r), and the number of possible iterations for the entire computation, indicated by r/Iter.

The number of Iter is generally slightly higher. These cause a higher running time and communication. The data volume and running time for first-round $\mathsf{r} = 1$ are similar in both sparse and dense graphs (those have the same number of vertices). The number of rounds r will increase running time and data volume. Thus, until the last round of computation takes place ($\mathsf{r} = \mathsf{Iter}$). Computation in dense graphs finishes early, while in sparse and planar-like graphs, iterating until the last round. This causes higher round complexity and communication for a given graph.

Table 15: Running times (in seconds) and data volume of different rounds for the privacy-preserving versions of BFS.

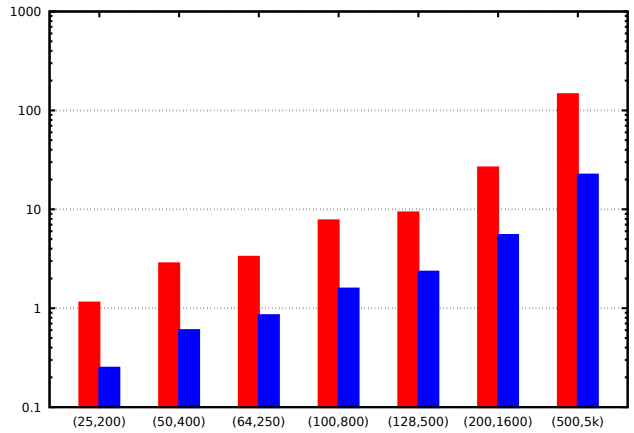
	Size		WBFS			UBFS		
k	n	m	r / Iter.	Time	Data volume	r / Iter	Time	Data volume
Sparse	50	400	1/4	0.07	0.8 MB	1/1	0.08	0.7 MB
	50	400	2/4	0.09	1.3 MB	-	-	-
	50	400	3/4	0.12	1.7 MB	-	-	-
	50	400	4/4	0.14	2.1 MB	-	-	-
	500	5k	1/11	3.9	43.9 MB	1/3	4.20	49.3 MB
	500	5k	2/11	5.7	83.3 MB	2/3	6.11	102.7 MB
	500	5k	3/11	7.42	109.0 MB	3/3	8.31	146.2 MB
	500	5k	7/11	14.4	277.9 MB	-	-	-
	500	5k	11/11	21.5	429.4 MB	-	-	-
	1k	40k	1/7	14.9	176.1 MB	1/2	16.2	197.7 MB
	1k	40k	2/7	22.0	319.2 MB	2/2	24.5	399.1 MB
	1k	40k	3/7	28.9	492.0 MB	-	-	-
Dense	100	4950	1/4	0.20	2.0 MB	1/1	0.21	2.6 MB
	100	4950	4/4	0.41	7.2 MB	-	-	-
	500	124k	1/4	3.87	49.7 MB	1/1	4.24	55.8 MB
	500	124k	2/4	5.73	90.7 MB	-	-	-
	500	124k	4/4	9.32	164.4 MB	-	-	-
	1k	499k	1/5	14.7	190.8 MB	1/1	15.6	226 MB
	1k	499k	5/5	43.4	562.7 MB	-	-	-
	2k	1.9M	1/5	56.9	753 MB	1/1	61.2	899 MB
	2k	1.9M	5/5	164	3.2 GB	-	-	-
	5k	12.4M	1/4	362	4.5 GB	1/1	391	5.6 GB
	5k	12.4M	4/4	868	16.0 GB	-	-	-
Like-Planar	50	100	1/7	0.07	0.5 MB	1/4	0.07	0.8 MB
	50	100	2/7	0.10	1.0 MB	2/4	0.10	1.2 MB
	50	100	7/7	0.22	3.1 MB	4/4	0.16	2.3 MB
	50	150	1/5	0.07	0.8 MB	1/3	0.07	0.8 MB
	100	200	1/9	0.21	1.9 MB	1/4	0.22	2.0 MB
	100	300	1/7	0.21	2.2 MB	1/3	0.22	2.2 MB
	500	1k	1/12	3.84	44.7 MB	1/8	4.04	50.5 MB
	500	1k	12/12	23.1	471.8 MB	8/8	17.9	391 MB
	1k	2k	1/12	14.7	158.8 MB	1/9	15.8	200 MB
	1k	2k	12/12	91.6	1.8 GB	9/9	76.4	1.7 GB

8.2.6. Evaluation of the privacy-preserving SSSD protocols

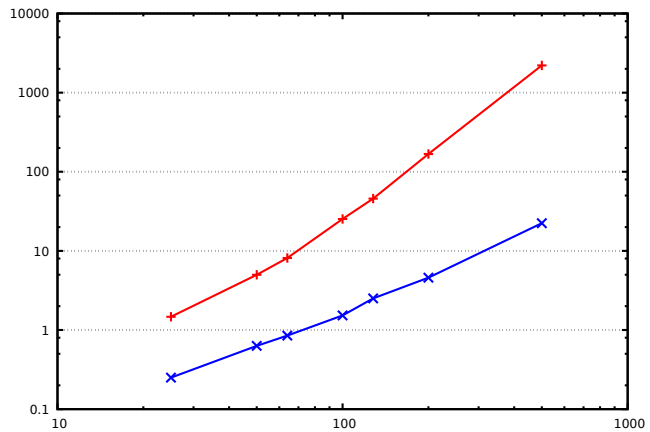
The previous section presented the Breadth first-search protocol benchmarked with the radius-steeping protocol. Both protocols have partially similar algorithmic structures and have $O(\log n)$ round complexities. The privacy-preserving breadth first-search protocol is more efficient than radius-stepping protocols over sparse and dense graphs.

In Bellman-Ford and Dijkstra protocols, both have partially similar algorithmic structures and have linearithmic round complexities. By comparing Table 7 and Table 5, we see that, generally, Dijkstra's protocol performs faster than the Bellman-Ford protocol. This is expected because the main loop of both algorithms makes n iterations. Still, the amount of work done in one iteration of the Bellman-Ford protocol is $O(m)$, while the amount of work done in one iteration of Dijkstra's protocol is $O(n)$. On the other hand, the Bellman-Ford protocol works on the sparse representation of the graph, while Dijkstra's algorithm requires a dense representation. Hence the memory consumption of the

latter may be significantly higher for sparse graphs. Dijkstra's protocol can work on a sparse graph representation, using oblivious RAM and loop coalescing [125]. However, this will increase the number of iterations to $O(m)$, which is undesirable when the actual performance of the algorithm is latency-bound.



(a) Sparse graphs.



(b) Dense graphs.

Figure 9: Performance (in seconds) comparison of Dijkstra's (blue) and Bellman-Ford (red) algorithms on sparse and dense graphs.

On a set of small and medium-sized graphs, we compare the performance of Dijkstra's and Bellman-Ford protocols in the HBLL environment, which can be seen in Figure 9 and Figure 10. We have evaluated the performance for specific pairs (n,m) of the number of vertices and edges for sparse graphs. For dense graphs, the horizontal axis shows the number of vertices n, while the number of edges is $n(n-1)/2$—the maximal possible number of edges in a graph. For graphs with several edges similar to planar graphs, we picked two or three times the number of vertices n as the number of edges m. Dijkstra's approach is more efficient in all circumstances, but the difference is less pronounced for sparse graphs.

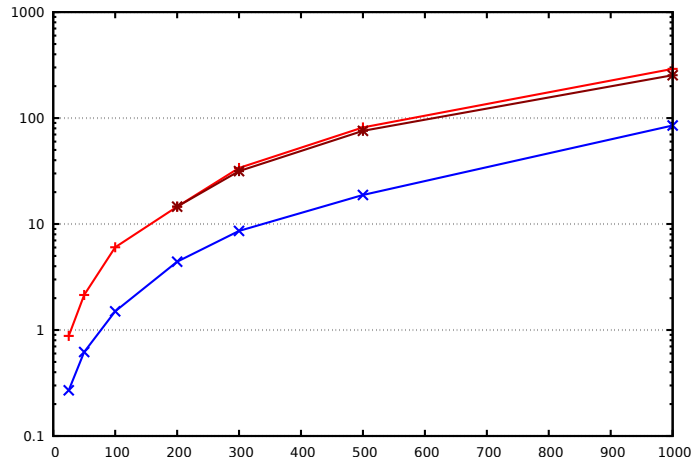


Figure 10: Performance comparison of Dijkstra's (blue) and Bellman-Ford (light red: $m = 3n$; dark red: $m = 2n$) algorithms on planar-like graphs.

Sec 8.2 presents experiments for the privacy-preserving single-source shortest distances protocols with their different versions using small and big graphs, sparse and dense. It is crucial to evaluate all privacy-preserving SSSD protocols together over sparse and dense graphs. Figure 11 illustrates the benchmark results for the four SSSD algorithms in privacy-preserving over dense graphs. In this evaluation, we benchmarked the most efficient version of the Radius-stepping protocol (∞), the Version 1 of Bellman-ford, Dijkstra, WBFS, and the most efficient protocol is WBFS, as can be seen in Figure 11.

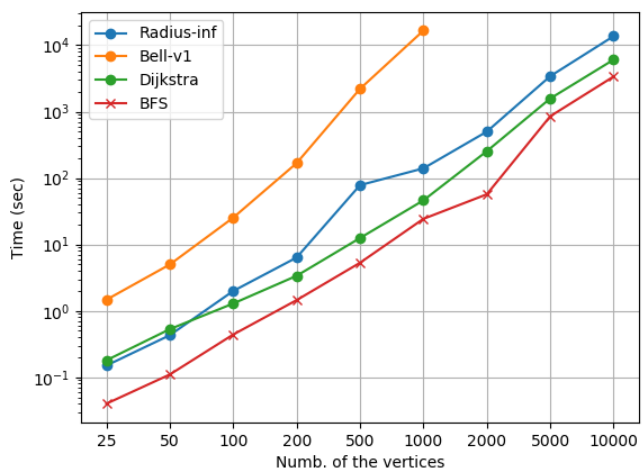


Figure 11: Running time of the privacy-preserving SSSD protocols over the dense graphs in different sizes.

In contrast, the privacy-preserving SSSD protocols for the sparse graph are presented in Figure 12. The number of edges m in the sparse graphs is three times the number of the vertices n. Dijkstra's protocol competes with the BFS protocol in a relatively big

graph. In the graphs, which have 2000-vertex and more, Dijkstra protocols become more efficient, as shown in Figure 12. There is no effect on the number of edges m in the Dijkstra protocol, but it affects the BFS protocol. Increasing the number of edges in the given graph makes the BFS protocol more efficient than Dijkstra.

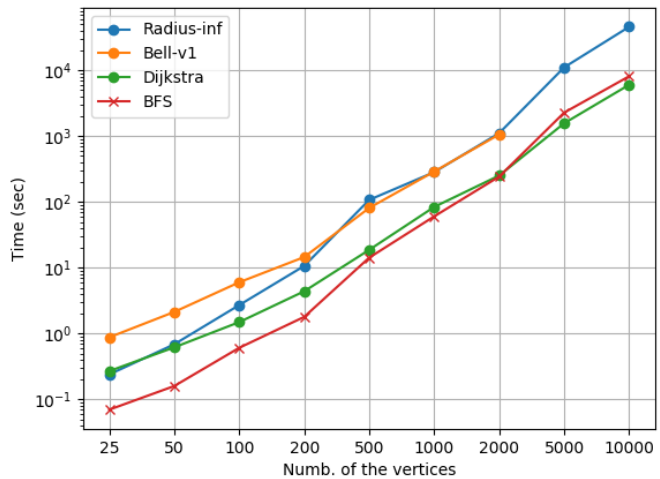


Figure 12: Running time of the privacy-preserving SSSD protocols over the sparse graphs in different sizes.

8.3. All pairs shortest paths experiments

8.3.1. Privacy-preserving Johnson's protocols experiments

The data input to the privacy-preserving Johnson's protocols starts from the sparse representation. Its running time depends on the number n of vertices and the number m of edges of the input graph. We report the running times for certain dense graphs in Table 16. The protocol consists of three identifiable stages—the execution of the Bellman-Ford algorithm (which has sparse representation), updating the lengths of edges, and execution of n copies of Dijkstra's algorithm. We report the running time of each stage for two versions of Johnson's protocol. Recall that in Version 1, n copies of Dijkstra's algorithm are executed—we use the privacy-preserving Dijkstra protocol presented in Algorithm 7. In Version 2, they are all performed in parallel—we use the privacy-preserving nDijkstra protocol presented in Algorithm 9. The benchmarking is done in the HBLL environment. The parallelization gains are smaller for small graphs due to the first non-parallelism steps taking relatively more time. We also see the parallelization gains drop for larger graphs, similar to the benchmarking outcomes shown in Figure 7 (HBLL plots, right edge of the figure).

As well as Table. 16 presents the speed-up, which is the difference between the two versions of Johnson's protocols. In other words, the speed-up is how often Version 2 is faster than Version 1. The versions of privacy-preserving Johnson's protocols are constructed based on the third stage, which is the privacy-preserving Dijkstra's protocols.

There is no change in the first stage—the Bellman-Ford's execution nor updating the edges' lengths in the second stage.

Table 16: Running Time (in seconds) of privacy-preserving APSD protocol.

Graph		Privacy-Preserving Johnson V1				Privacy-Preserving Johnson V2				V1 , V2
n	m	BF.	upd.	Dijk.	Total	BF.	upd.	Dijk.	Total	Speed-up
5	10	0.18	0.03	0.10	0.31	0.18	0.03	0.06	0.27	1.2x
10	45	0.45	0.45	0.51	1.41	0.45	0.45	0.18	1.08	1.3x
20	190	1.04	0.07	2.22	3.33	1.04	0.07	0.55	1.66	2.0x
50	1225	5.28	0.23	20.7	26.2	5.28	0.23	6.19	11.7	2.2x
100	4950	27.2	1.0	109	138	27.2	1.0	43.6	71.8	1.9x
200	19.9k	166	3.55	583	752	166	3.55	339	508	1.5x
500	124k	2282	26.9	6644	8954	2282	26.9	5015	7324	1.2x
1k	499k	16392	117	48582	65477	16392	117	43599	60494	1.08x

8.3.2. Floyd-Warshall and transitive closure experiments

The main goal of implementing the Floyd-Warshall and transitive closure in privacy preservation on the SMC Sharemind platform is to benchmark it with the privacy-preserving APSD Johnson protocols. The execution time of our privacy-preserving Floyd-Warshall algorithm and transitive closure computation also depends only on the number n of the vertices of the input graph $[\![\mathbf{G}]\!]$. We report the running times and data volume in Table 17 for various values of vertices n for different network environments.

Table 17: Benchmarking results (data volume for a single computing server) for Floyd-Warshall and transitive closure algorithms in different network environments.

Size of graph	Floyd-Warshall				Transitive closure			
	Data volume	Running time (s)			Data volume	Running time (s)		
		HBLL	HBHL	LBHL		HBLL	HBHL	LBHL
5	0.08 MB	0.01	2.22	2.22	0.46 MB	0.02	4.01	4.01
10	0.48 MB	0.03	4.44	4.46	1.64 MB	0.05	7.37	7.53
20	3.52 MB	0.1	9.1	9.35	16.4 MB	0.29	12.9	14.2
50	54.1 MB	0.92	23.6	28.6	318 MB	5.56	26.9	66.3
100	402 MB	6.91	52.9	90.5	3019 MB	51.1	157	426
200	3417 MB	62.4	153	526	27.3 GB	460	1336	2529
500	53.3 GB	934	2753	7469	490 GB	7987	23.5k	–
1k	426 GB	7268	21.5k	57.4k	–	–	–	–

In Figure 13, we compare Floyd-Warshall and transitive closure algorithms for different network environments. Despite the lower round complexity, transitive closure is still slower in high-latency environments. We have not included Johnson's protocol in this comparison due to the significant number of tunable parameters. The previous benchmarking results allow one to estimate the performance of different stages of Johnson's protocol for various values of these parameters and in other networking environments.

8.3.3. Evaluation of the protocols

The running times of all privacy-preserving APSD protocols on HBLL are illustrated in Table 18. It shows that transitive closure is not competitive with Floyd-Warshall regarding

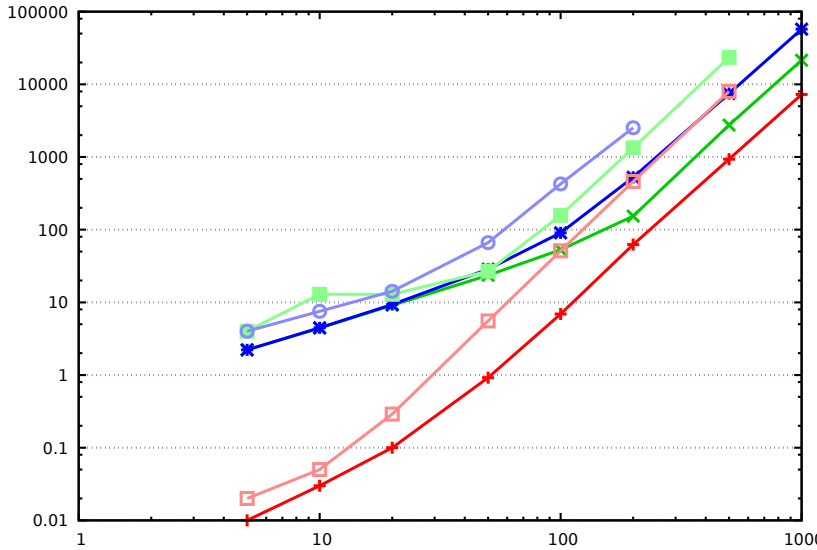


Figure 13: Performance (time in seconds) of Floyd-Warshall and transitive closure algorithms on graphs of different sizes in different network environments (red: HBLL, green: HBHL, blue: LBHL, dark: Floyd-Warshall, light: transitive closure).

running time, even though its round complexity is smaller. It also indicates Johnson's protocols (both versions) require significantly more time than Floyd-Warshall and transitive closure.

Table 18: Running times (in seconds) of privacy-preserving APSD protocols.

Graphs		**Privacy-preserving APSD**			
n	m	Johnson V1	Johnson V2	Floyd-Warshall	Transitive Closure
5	10	0.31	0.27	0.01	0.02
10	45	1.41	1.08	0.03	0.05
20	190	3.33	1.66	0.10	0.29
50	1225	26.2	11.7	0.92	5.56
100	4950	138	71.8	6.91	51.1
200	19.9k	752	508	62.4	460
500	124k	8954	7324	933	7987
1k	499k	65477	60494	7268	–

However, the latter is not a good comparison, as Table 18 only presents the worst-case running time for Johnson's protocol for graphs with a given number of vertices n. Indeed, the following aspects may improve the running time:

- If the number of edges is smaller, then the execution of the Bellman-Ford step needs less time. We also use Bellman-Ford Version 1, which has low running time than Version 2.
- The execution time of the Bellman-Ford step may be shorter if it is run for a smaller number of iterations (see discussion in Sec 8.2.3).
- If the shortest distances have to be found only from a subset of vertices, then a smaller number of instances of Dijkstra's algorithm has to be executed.

- It's highly recommended to use Version 2, which is more parallel than Version 1.

None of these aspects applies to the Floyd-Warshall algorithm. The consequences of running time from all of these aspects have been covered in our benchmarks in Table 7 and Figure 7.

The Floyd-Warshall and Transitive Closure benchmarks show lower running time than the versions of Johnson's protocols because both have a simple structure. The running time in the privacy-preserving Floyd-Warshall algorithm is lower than in the privacy-preserving Transitive Closure algorithm. In contrast, the Transitive Closure has $O(\log n)$-round complexity lower than the $O(n)$-round complexity of the Floyd-Warshall. This exciting result shows how the time complexity can significantly affect the total running time of the algorithm. In such a case, high time complexity considers an obstacle to reducing the latency.

The communication among computation parties takes place in Transitive closure after performing the three for-loops (time complexity is $O(n^3)$), while in Floyd-Warshall only takes place after running two for-loops (time complexity is $O(n^2)$). At the same time, the Transitive Closure algorithm has more communication than the Floyd-Warshall algorithm. In general, the privacy-preserving Floyd-Warshall is the most efficient method in terms of latency in comparison with other algorithms. The privacy-preserving Transitive Closure algorithm is the most efficient method in terms of communication in comparison with other algorithms. The importance of privacy-preserving Johnson's protocol is that it can also be efficient for finding the All-pairs shortest Distance for a subset of the vertices in the given graph, while Floyd-Warshall and transitive closure run the whole graph.

9. EXPERIMENTS OF MINIMUM SPANNING TREE AND FOREST PROTOCOLS

9.1. Benchmarking results for minimum spanning tree

Some benchmarking results for privacy-preserving minimum spanning tree protocols have already been documented. Implementation of the Awerbuch and Shiloach algorithm for computing the minimum spanning tree in privacy-preserving computation is proposed in [114]. The private read and write protocol (Laud's protocol) is used in the privacy-preserving MST's implementation—the same protocol used in some implementations in this thesis. The proposed algorithm has a logarithmic time on different graph sizes. The number of processors is based on the number of given edges in the private graph $[\![\mathbf{G}]\!]$. In detail, a dense graph with 2k vertices (and 1999k edges) is benchmarked in his work, and the running time is more than 10^4 seconds, while the running in our implementation is around 475 seconds. More details about benchmarking for this proposed protocol with our work are presented in Sec 9.4. In the sparse graphs used on his benchmark, the edges are only six times the number of vertices. This means the algorithm is inefficient for using more edges, and thereby, the algorithm is inefficient on a dense graph.

In [146], sequential implementations of the privacy-preserving minimum spanning tree are proposed by separately implementing two classical MST algorithms, Prim and Kruskal. There is no actual implementation in their work; the time complexity for both algorithms is $O(m \log n)$.

9.2. Privacy-preserving Prim's protocol experiments

This section presents the experiments of the parallel Prim's minimum spanning tree protocol on the Sharemind secure multiparty computation platform, shown in Sec 6.2. The running time of our parallel privacy-preserving Prim's MST protocol depends on the number of vertices in the graph, while the number of edges does not influence the running time. More specifically, each row in the adjacency matrix will be taken in computation in each iteration using single instruction, regardless of the contents of the row. If there is an edge, then the record in the row has a weight. If there is no edge, the record will be "∞". We report the execution time (in seconds) for running on several graphs with different sizes in Table 19. Note that our algorithm uses the adjacency matrix to represent the private data of the graphs, i.e., the data structure of the dense graph is used, but for the different kinds of graphs depends on the number of vertices and edges. We used three types of graphs in our implementation: sparse, dense, and planar. We also use the fourth set of big graphs with tens of thousands of vertices with millions of edges. This is the first implementation with an enormous number of edges and vertices, particularly for dense graphs. We split the running time into three calculation groups to analyze the algorithm's actual behavior. The three groups are Permutation-operation which is shuffling the rows and columns in the given graphs before finding the MST—called Perm, the entire loop-operation is for finding MST—Loop, and the last part performRead-operation for reading the private array—PerfR. Then, the last column in the table is the total running time.

In the first group of graphs, we used the data of the sparse graph with dense representation in the implementation. The smallest graph is a graph with 20 vertices and 75 edges. The number of edges is around 3x times the number of vertices.

Table 19: Running time (in seconds) of privacy-preserving Prim's algorithm

	Graph		Privacy-preserving MST's Prim			
	n	m	Perm.	Loop	PerfR.	Total
Sparse	20	75	0.02	0.2	0.02	0.24
	50	150	0.08	0.7	0.02	0.81
	50	250	0.09	0.71	0.02	0.81
	50	1k	0.08	0.72	0.02	0.83
	200	1k	0.81	6.07	0.04	6.91
	200	5k	0.85	6.05	0.04	6.93
	1k	5k	15.3	110	0.1	126
	3k	10k	136	920	0.2	1056
	3k	15k	137	914	0.2	1052
	3k	50k	133	920	0.3	1053
	3k	100k	136	910	0.2	1046
	5k	20k	375	2478	0.4	2854
	5k	50k	371	2516	0.4	2887
	5k	100k	375	2466	0.4	2814
Dense	50	1225	0.09	0.75	0.02	0.86
	100	4950	0.23	1.9	0.02	2.18
	250	31.1k	1.2	8.9	0.04	10.1
	500	124.7k	4.3	31.3	0.07	35.7
	1k	499.5k	14.5	107	0.1	121
	2k	1999k	61.3	413	0.2	475
	5k	12497k	375	2502	0.4	2879
	10k	49.9M	1573	10069	0.9	11642
Planar-like	100	200	0.3	2.1	0.03	2.43
	100	300	0.25	2.3	0.03	2.58
	500	1000	4.1	32.2	0.06	36.36
	500	1500	4.1	32.1	0.06	36.3
	1k	2k	16.6	111	0.1	128
	1k	3k	15.5	109	0.1	125
	2k	4k	58.4	418	0.2	476
	2k	6k	57.4	416	0.2	473
	3k	6k	133	913	0.2	1047
	3k	9k	136	909	0.3	1046
Big	8500	300k	1.1k	7.1k	0.6	8.2k
	9500	500k	1.4k	8.8k	0.8	10.2k
	10k	1M	1.5k	9.8k	0.7	11.3k
	20k	5M	5.9k	39.1k	2.1	45.0k
	20k	10M	6.1k	39.6k	2.3	45.7k
	30K	5M	13.4k	89.8k	3.6	103.2k

The graphs in the group are based on the number of edges which is given by *m=xn*. The biggest graphs we processed are the graphs where the number of edges is around 20x and 33x times the number of vertices. The result shows that the number of edges does not influence the algorithm's running time using the SIMD. In the group of dense graphs, the number of edges is given by $n(n-1)/2$, where n is the number of vertices. The result shows that our protocol for finding MST is more efficient than the one presented in Sec 9.1. The third group of the graphs is a planar graph, where the edges are 2 or 3 times the number of the vertices. Big graphs are also implemented in our algorithm, and the result shows how our algorithm is efficient for finding MST in the private calculation for big graphs with up to ten million edges.

9.3. Privacy-preserving optimized-Prim's protocol

The optimized version of the privacy-preserving Prim's MST protocol is also benchmarked with one in previous Sec 9.2. The benchmark is also performed over the SMC Sharemind platform. The data input is different graphs represented in adjacency matrices of various sizes. Although the given graph is described in an adjacency matrix, the algorithm will vectorize the adjacency matrix into vectors and fit with the SIMD. We use a function to generate random graphs with a specific size as vertices with their possible number of edges. Moreover, the benchmark is not just appointed towards the running time. The communication among the cluster's machines is also represented in secret-shared manners for inputs and outputs.

The feature of the privacy-preserving optimized Prim and Prim's algorithms is that there is no effect on the number of connected edges with vertices in the running time. The algorithm relaxes its edges for all possible connected edges with the vertices. Regardless if there are edges connected to some vertices or there is no "∞". The maximal possible number of the edges for each vertex is n-1. So, all the elements in the row will be taken in a relaxing operation in parallel (SIMD) regardless of how many non "∞" elements are in the u-th row. Consequently, the running time for privacy-preserving Prim's and optimized Prim's algorithms over sparse and dense graphs is similar. Optimizing Prim's algorithm might be a little done (no significant change between optimized Prim and Prim's algorithms). Hence the results show how much such optimization effect the total running time of the algorithm. This is because of the sensitivity of the private data and reducing the round complexities that positively affect in reducing the running time in the optimized Prim's algorithm.

In detail, the privacy-preserving optimized Prim's protocol has three main parts, permutation, finding the MST, and applying Laud's protocol for the private read. The optimization in the algorithm targeted the finding of MST, neither permutation nor Laud's protocol. The running time and data volume for the privacy-preserving optimized prim protocol with their parts are presented in Table 20, which is benchmarked with the privacy-preserving Prim algorithm with their parts. The empirical test shows the running time for the algorithms' parts with speed-up and data volume.

Table 20: Running time (in seconds) and data volume of privacy-preserving prim's protocols.

	Graph		Prim's MST					Optimized Prim's MST					Speed-up
	k	n	m	Data volume	Perm.	Loop	PerfR.	Total	Data volume	Perm.	Loop	PerfR.	Total	x-time
Sparse	20	75	0.1 MB	0.02	0.16	0.02	0.24	0.2 MB	0.02	0.2	0.02	0.2	1.2x
	50	250	1.2 MB	0.09	0.71	0.02	0.81	0.7 MB	0.1	0.5	0.02	0.56	1.4x
	200	1k	15.2 MB	0.81	6.07	0.04	6.91	6.0 MB	0.8	2.5	0.04	3.36	2.0x
	1k	5k	350 MB	15.3	110	0.1	126	129 MB	15.6	36.7	0.17	52.4	2.4x
	3k	100k	2.95 GB	136	910	0.2	1046	963 MB	145.7	237	0.53	383	2.7x
	5k	20k	8.38 GB	375	2478	0.4	2854	2.56 GB	363	632	0.88	997	2.8x
Dense	50	1225	1.4 MB	0.09	0.75	0.02	0.86	0.7 MB	0.1	0.4	0.03	0.56	1.5x
	100	4950	4.2 MB	0.23	1.9	0.02	2.18	2.1 MB	0.2	1.2	0.04	1.47	1.4x
	250	31k	23.2 MB	1.2	8.9	0.04	10.1	8.7 MB	1.3	3.7	0.06	5.15	1.9x
	500	124k	89.2 MB	4.3	32.3	0.07	36.7	31.0 MB	4.2	8.9	0.1	13.1	2.8x
	1k	499k	344 MB	14.5	107	0.1	121	120 MB	15.7	30.7	0.16	46.6	2.6x
	2k	1.9M	1.3 GB	61.3	413.6	0.2	475	436 MB	58.2	123	0.32	181	2.6x
	5k	12.4M	8.2 GB	375	2502	0.4	2879	2.59 GB	386	880	0.89	1267	2.3x
	10k	49.9M	32.8 GB	1573	10069	0.9	11642	10.2 GB	1616	2280	1.74	3897	2.9x

The result shows how often the optimized Prim is more efficient than Prim's. The optimized Prim protocol's total speed-up is between 1.2 and 3 times more, including the permutation and laud's protocol parts. In both protocols, the running time of both parts, permutation and Laud protocol, is less than the loop. For finding MST only without

permutation and Laud's protocol, the result shows that the speed-up of the optimized Prim's protocol is 4.5 times faster than the same part in Prim's protocol.

In the case of network communication, we reported the data volume benchmarking for the optimized Prim algorithm and Prim's algorithm over the Sharemind SMC cluster. These tests were conducted using sparse and dense graphs of different sizes. It is important to note that the running time was reported as an average of three cluster machines. As well as the data volume, the average total maximum data transfer rate across the computation parties is taken. The tests are done over the preferable network case, a *high-bandwidth* and *low-latency* setting. In the *high-bandwidth* setting, the link speed between the cluster machines is 1 Gbps, while in *low-latency*, there is no delay in transferring, which is 0 ms. The result appears that the optimized Prim protocol has low data volume compared with Prim's protocol. The difference is around 2 to 3 times less. This is because of the optimized prim protocol's reduction in the round complexity.

9.4. Evaluation of the protocols

In this section, we present the benchmark of our privacy-preserving prim's MST protocols, with the previous work, privacy-preserving Awerbuch-Shiloach's MST algorithm [114]. The benchmark results for the three MST protocols in privacy preservation over dense graphs are presented in Figure 14. The algorithms have been implemented on top of the SMC protocol and built using Laud's protocol for the private read. The optimized-prim protocol is more efficient than the other cases of dense graphs with various sizes.

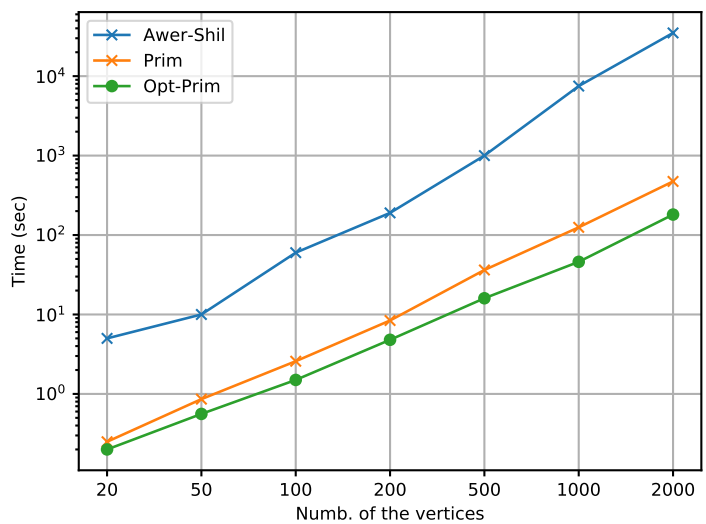


Figure 14: Running time of the privacy-preserving MST protocols over dense graphs.

Figure. 15 shows the running time for the sparse graphs over the three protocols. The sparsity is that the number of connected edges is three times the number of vertices ($m = 3n$). Prim's protocols (the prim and the optimized) are more efficient than Awer-Shil's for small graphs. Although the prim's protocols fit the dense graph (and Like-dense), the result shows that it is efficient for the small sizes of the sparse graphs.

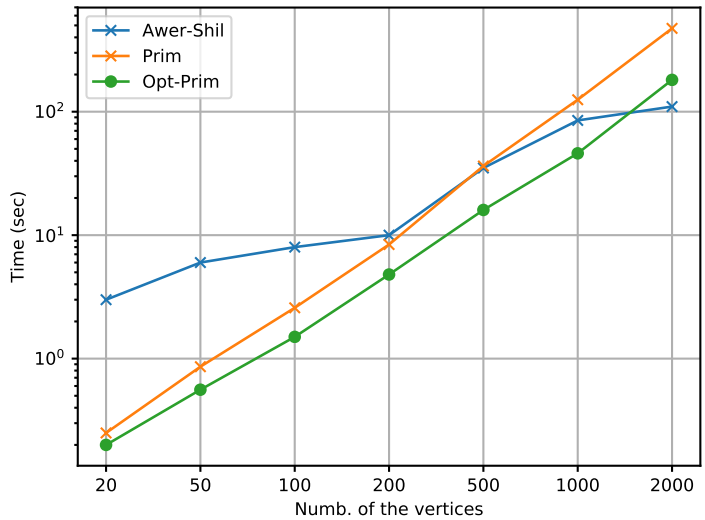


Figure 15: Running time of the privacy-preserving MST protocols over sparse graphs

9.5. Minimum spanning forest experiments

This section presents the empirical tests and the sequential and parallel effect of the privacy-preserving MSF protocols. Intuitively, parallel MSF is faster than sequential MSF protocols. However, we created the sequential version of the MSF protocol to evaluate it and benchmark it with the parallel MSF protocol. In the test, we use different graphs, sparse and dense, with various sizes. We used the optimized Prim's protocol as a subroutine in MSF computing in both versions of the privacy-preserving MSF protocol. The running time of the privacy-preserving MSF protocols using various graphs of different sizes is presented in Table 21. The graphs we used in this test have different sizes: the number of vertices n and the number of components g. The table shows the running time of the three parts of the MSF protocol separately and the total running time for both versions. As well the speed-up of the parallel MSF protocol is presented. The speed-up is how much time the parallel MSF version is faster than the sequential.

There is no effect for the number of edges in the running time, and we used the maximal possible number of the edges m in the test, which is given by $n(n\text{-}1)/2$. We use a function to generate random graphs with specific sizes in all implementations. Each graph has g components, with n vertices with their maximal connected weighted edges m. Then, the number of components is similar to the number of the vertices $g = n$. The sequential MSF protocol is implemented by calling the three parts sequentially g times in for-loop. There is no running for the permutation and laud parts in parallel, calling the optimized-prim a subroutine of the MSF g times. Hence, the operations in the optimized Prim are in parallel. The parallel MSF protocol is implemented by calling nPrim only once but using g graphs. So, both versions run the same g graphs, but each one is based on its structure. While running, the permutation and Laud parts are also sequential. Precisely, the speed-up gain in parallel MSF protocol is because of running the nPrim.

The results show the running time of both versions of MSF protocol with speed up, which reached three times. The running time of the laud's protocol in both versions is

Table 21: Running Time (in seconds) of privacy-preserving computation of minimum spanning forest protocols.

Graphs			Privacy-Preserving MSF V1				Privacy-Preserving MSF V2				V1 vs. V2
g	**n**	**m**	Perm.	Loop	PerfR.	Total	Perm.	Loop	PerfR.	Total	**Speed-up**
16	16	120	0.17	1.37	0.25	1.79	0.19	0.19	0.23	0.61	2.9x
32	32	496	0.76	6.53	0.7	7.99	0.72	1.21	0.72	2.65	3.0x
50	50	150	3.2	20.3	1.4	24.9	3.2	5.9	1.3	10.5	2.4x
50	50	1225	2.80	18.3	1.33	22.4	2.60	3.42	1.33	7.33	3.1x
64	64	200	5.9	34.8	1.7	42.4	5.9	11.1	1.9	18.8	2.2x
64	64	2016	5.70	33.3	1.82	40.9	6.40	6.66	1.70	14.8	2.8x
100	100	300	20.2	100	3.2	123	20.2	35.6	3.1	58.9	2.1x
100	100	4950	21.1	94.1	4.50	119	18.8	22.8	3.26	44.8	2.7x
128	128	8128	40.8	206	4.48	251	48.1	47.4	4.65	100	2.5x
250	250	2000	244	1145	16.1	1406	298	539	18.1	856	1.6x
250	250	31k	244	964	18.5	1227	280	378	14.9	674	1.8x
500	500	124k	1980	5069	55.7	7105	2241	3049	54.8	5346	1.3x
1000	1000	499k	15205	32061	210	47476	17952	24670	198	42821	1.11x

similar. In contrast, the running of the permutation part is different, especially in big graphs. This is because of running the assign-operation for the graphs into 3-dimensional matrix $[\![\mathbf{G}]\!]$ and assigning the sources as well. The assign is an extra operation that does not exist in the sequential version. Additionally, we benchmark the parallel MSF with the sequential version using different graphs that might be less than the number of vertices ($g \leq n$) in various network specifications. We consider three network environments as we described them in Sec 3.6. We also run different dense graphs: big and small, with various components. Figure 16 presents the running time for the privacy-preserving parallel MSF protocol over different network environments.

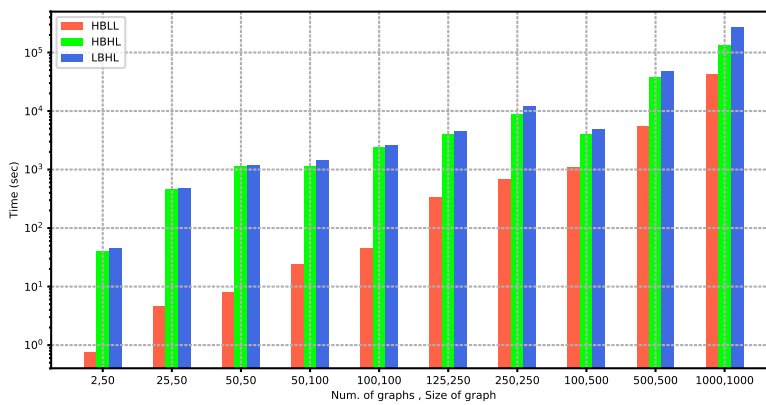


Figure 16: Running time (in seconds) of the privacy-preserving parallel MSF protocol (Version 2) over different networks.

It shows the number of the graphs with the size of the graph, and it does not matter the number of their connected edges. The running time for both versions of the privacy-preserving MSF protocols on different network environments is presented in Table 22. In general, the running time for the parallel MSF protocol is less than the running time of the sequential version. In contrast, the sequential version of the algorithm has less communication than the parallel version of the MSF protocol. In particular, we have focused

primarily on the functional aspects of networks for running our privacy-preserving MSF protocols in terms of latency. Those results are a practical proof of our methodology in implementing parallel computing for finding MSF. Moreover, the speed-up in the second and third networks reached hundreds of times. It shows how much parallel strategy is efficient and scalable. Hence, the challenge is using a higher bandwidth link to transfer the whole big vector without splitting. Applying a single instruction yields less round complexity, while vectorization of the data into a vector (in case the vector size is bigger than bandwidth) yields round complexity again.

Notably, the size of the working vectors in the nPrim in parallel MSF is n^2, which is larger than in the Optimized-Prim in sequential MSF protocol. The computation parties may only send part of the vector in a single trip between them, particularly when executing a big vector that has a size larger than the bandwidth. The large vector will be divided into small vectors by the platforms. Then each small vector will be sent one at a time. This suggests that the round's complexity will rise. As a result, the parallel version of the MSF protocol still runs faster.

Table 22: Running Time (in seconds) of privacy-preserving computation MSF's protocols on different networks.

Num. of graphs	**Size of graph**	**Sequential MSF V1**			**Parallel MSF V2**		
		HBLL	HBHL	LBHL	HBLL	HBHL	LBHL
8	16	0.98	427	365	0.32	99.6	120
16	16	1.71	881	730	0.51	215	224
32	32	8.82	4001	820	2.87	524	611
2	50	1.16	415	365	0.76	40.5	43.9
10	50	5.13	2477	1826	2.42	192	210
25	50	11.6	6194	4565	4.49	457	476
50	50	22.6	12388	9130	7.8	1118	1167
2	64	1.49	407	562	0.95	42.6	38.0
64	64	45.0	13030	17984	14.1	1181	1263
2	100	2.70	790	1085	1.80	50.4	44.3
10	100	12.3	3954	5427	6.84	212	262
25	100	31.9	9886	13569	13.1	532	586
50	100	66.6	19772	27139	23.5	1127	1444
100	100	116	39454	54278	45.1	2400	2579
2	250	12.7	2749	2802	11.1	52.3	61.7
125	250	653	171k	237k	335	3968	4440
250	250	1227	343k	475k	674	8890	11947
5	500	70.9	14822	20419	62.8	172	193
10	500	141	29644	40839	110	338	404
100	500	1423	296k	408k	1067	3907	4916
500	500	7105	1482k	2041k	5346	38114	46608
2	1000	98.5	12906	18138	105	118	136
10	1000	478	64534	90694	409	402	651
1000	1000	47476	6453k	9069k	42821	134k	264k

10. EXPERIMENTS OF ALGEBRAIC PATH COMPUTATION PROTOCOL

This chapter presents the extensive benchmarks and analysis of the secret-shared based secure multiparty computation of the Algebraic path computation protocol. The implementation and benchmarking of this protocol are done on various graph sizes, providing an overview of how they stack up on top of secure multiparty computation protocols in different deployments. For analyzing and evaluation, the experiments and analysis of the privacy-preserving public edges' version of the Bellman-Ford protocol (version 3) are also done over the SMC Sharemind platform. The different sizes of graphs used in these experiments are generated using a random generating function.

10.1. Algebraic path computation experiments

We have implemented our privacy-preserving algebraic path computation protocol and its related algorithms presented in Sec 7.3 and have tested them on different sizes of graphs obtained from a random generating function. The generated gird graphs are given by $\mathbf{G}(A)$, where A is a $R \cdot C$ adjacency matrix, while R and C are the numbers of rows and columns in the grid graph, respectively. The number of edges in the graph is given by $2RC$ - R - C, where R and C are the numbers of rows and columns in a graph, respectively. Furthermore, the depth of tree is given by $d = 2 \cdot k$, where $k \in \{2,3,\ldots,\infty\}$. In the algebraic path computation protocol, we use only grid graphs (with different sizes); the construction of a separator tree is a task that is at the same time non-trivial and peripheral to the goal of secure computation; hence we do not want to put significant effort into programming it—construction of a separator tree is public computation.

The protocol is designed to perform the computation sparsely (which means performing the computation on a sparse representation of matrices), and we use sparse graphs. Nevertheless, the running time of the privacy-preserving algebraic path computation protocol depends on the number of vertices n and edges m. Note that the number of edges in

Table 23: Running times (in seconds) and data volume of privacy-preserving algebraic path computation protocol.

Graph		**Recursive-**	**Algebraic Path Computation**	
G(A)	A×A	**cycle**	Data volume	Time
5	25	4	0.16 MB	0.1
9	81	6	0.30 MB	0.3
17	289	8	2.31 MB	1.2
33	1089	10	27.3 MB	8.2
50	2500	12	90.3 MB	30.1
65	4225	12	366 MB	66.4
100	10000	14	874 MB	244
129	16641	14	1972 MB	522
150	22500	16	3136 MB	838
200	40000	16	7792 MB	2029
257	44049	16	16.4 GB	4280
513	263169	18	138.3 GB	35341
600	360000	20	224.6 GB	58082
1025	1050625	–	–	–

a given graph also depends on the number of vertices n. The running times and data volume of the secret-sharing based security multiparty computation protocol of the algebraic path computation are illustrated in Table 23. Running times and data volume are given in the HBLL environment. The total running times are recorded to the main computation of the Algebraic path (Algorithm 32) with its related functions. We did not record the preparatory step, a public operation with no round complexities. The data volume among the computation parties of the SMC Sharemind platform will be reduced.

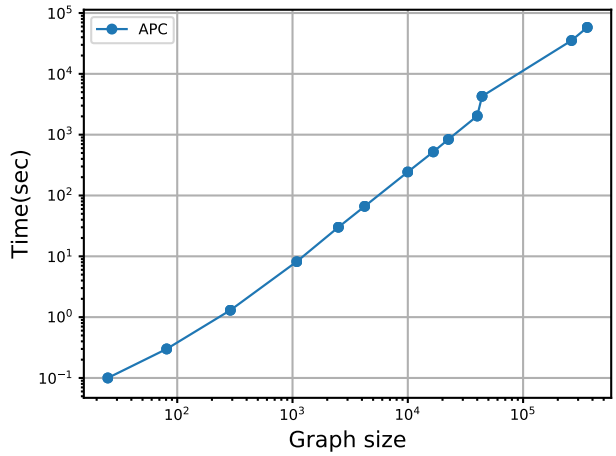


Figure 17: Relation of running time and graph size for the privacy-preserving APC protocol.

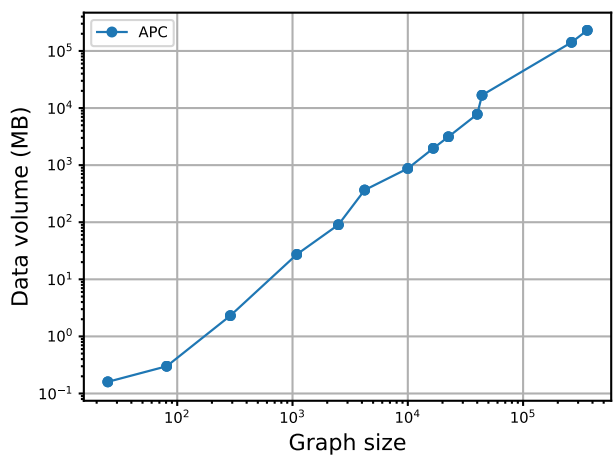


Figure 18: Relation of data volume and graph size for the privacy-preserving APC protocol.

The parallelization targets not only the main computation but also the related functions of the proposed protocol. Those related functions are constructed to deal with the sparse representation of matrices in privacy preservation. Furthermore, some of those related

functions have no private operations over private data, which means they have no round complexity, see Sec 7.5.1. It is important to note that such parallel functions can be used as a subroutine in constructing other protocols in the sparse-linear system on top of secure multiparty computation. Note that each Sum-sparse-function may have a different execution time depending on the size of the matrices (which are represented sparsely), while the size of matrices is based on the separator-tree.

In Figure 17, we show the relationship between the grid graph size and the computation parties' average running time. Furthermore, we present in Figure 18 the relation between the data volume and size of the grid graph. The data volume is the average transferred data for the three computation parties of the SMC Sharemind platform. In Figure 19, we establish the baseline for our experiments, measuring the running time of privacy-preserving Algebraic path computation protocol on graphs of different sizes in different network environments. The performance is very much latency-bound, such that the available bandwidth (even without First- and Second- normalization) even does not affect the performance on most graphs in high-latency environments.

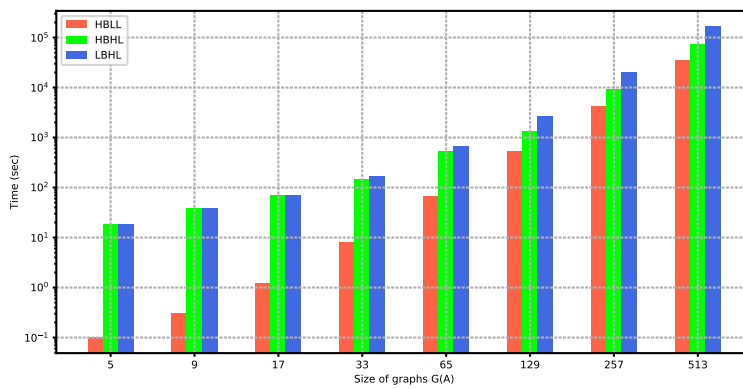


Figure 19: Performance of algebraic path computation protocol on graphs with given numbers of vertices in different network environments (red: HBLL, green: HBHL, blue: LBHL).

10.2. Bellman-Ford public edges (Version3)

The performance of the privacy-preserving Bellman-Ford version 3 protocol depends on the number of vertices n and edges m of the given private $[\![\mathbf{G}]\!]$. Increasing the number of edges in a graph will increase the running time. This version of the Bellman-Ford protocol has public elements, the number of the vertices n, edges m, and both vectors of vertices $\vec{R}$ and edges $\vec{C}$, while weights $[\![\vec{W}]\!]$ are private. The lonely preparatory step in Version 3 is the sorted vector $\vec{T}$, followed by the main loop of the proposed algorithm, also executed at most $(n-1)$ times. The running time and data volume of the Bellman-Ford Version 3 protocol are presented in Table 24. Various graph sizes are used in this implementation, and the running time is given in the HBLL environment. The Bellman-Ford protocol with its versions is fitter for sparse representation than dense, as shown in Table 24. Although Bellman-Ford is suitable for sparse graphs, the benchmarking is done over sparse and dense graphs. The lowest running time of the Bellman-Ford protocols (version 1, 2, and

3) is when the edges are minimum—in like-planar graphs, running time is lowest than in graphs with the same number of vertices for sparse and dense graphs.

Table 24: Running times (in seconds) and data volume of privacy-preserving Bellman-Ford version 3 protocol.

	Graph		Bellman-Ford V3	
k	**n**	**m**	Data volume	Time
Sparse	10	25	0.3 MB	0.04
	20	100	0.5 MB	0.11
	50	400	1.6 MB	0.55
	100	400	2.9 MB	1.14
	200	900	11.0 MB	3.31
	500	5k	140 MB	24.4
	1k	10k	538 MB	74.2
	2k	50k	5.44 GB	688
Dense	10	45	0.2 MB	0.04
	25	300	1.0 MB	0.23
	50	1225	4.6 MB	1.07
	100	4950	32.4 MB	4.55
	200	19.9k	237 MB	27.1
	500	124k	3.4 GB	434
	1k	499k	28.4 GB	3368
	2k	1999k	232 GB	26325

We also benchmarked Version 3 using two fundamental tools in measure, data volume and running time, as shown in Table 24. We benchmarked our work over different network environments, HBLL, HBHL, and LBHL. The running times of the Bellman-Ford Version 3 over different network environments are presented in Figure 20. In benchmarking in this test, the edges of the given graph depend on the maximum possible number of the edges in the $n \times n$ grid graph (see Sec 10.1), where n is the number of the vertices.

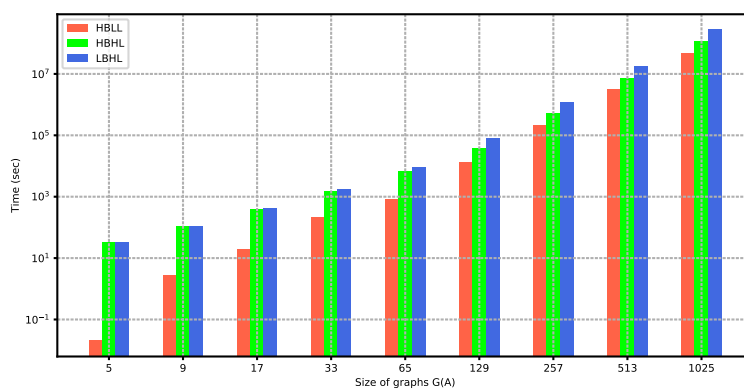


Figure 20: Performance of Bellman-Ford Version-3 protocol on graphs with given numbers of vertices in different network environments (red: HBLL, green: HBHL, blue: LBHL).

Version 3 is more efficient than Version 1 and Version 2 on the SMC Sharemind platform. The computation of Version 3 is the lowest round complexity among the compu-

tation parties of Sharemind because of using the public edges. The benchmarking of the privacy-preserving Bellman-Ford protocol versions is presented in Table 25. It shows the running times and data volume for different graph sizes in sparse and dense graphs. In the sparse graphs, the edges are four times the number of the vertices in the given graphs—$m = 4n$. In terms of running time, Version 3 is more efficient than other versions.

Table 25: Running time (in seconds) and data volume for privacy-preserving Bellman-Ford protocol versions.

	Graph		**Bellman-Ford V1**		**Bellman-Ford V2**		**Bellman-Ford V3**		**Speed-up V3**	
K	n	m	Data volume	Time	Data volume	Time	Data volume	Time	*vs. V2*	*vs. V1*
Sparse	20	80	0.85 MB	0.66	0.98 MB	0.47	0.38 MB	0.15	3.1x	4.4x
	50	200	3.1 MB	1.97	4.41 MB	1.50	0.88 MB	0.41	3.6x	4.8x
	100	400	8.1 MB	4.72	17.3 MB	5.12	3.12 MB	1.25	4.0x	3.7x
	500	2k	177 MB	67.2	502 MB	101	56.1 MB	13.2	7.6x	5.1x
	1k	4k	449 MB	250	2.1 GB	351	216 MB	38.6	9.1x	6.5x
Dense	20	190	1.37 MB	0.76	2.1 MB	0.59	0.7 MB	0.17	3.4x	4.4x
	50	1225	9.58 MB	3.88	26.3 MB	5.57	4.7 MB	1.20	4.6x	3.2x
	100	4950	53.9 MB	15.9	224 MB	30.9	32 MB	4.38	7.1x	3.6x
	500	124k	4.96 GB	1391	33.1 GB	3895	3.3 GB	435	8.9x	3.2x
	1k	499k	239 GB	9237	456 GB	28618	26 GB	3004	9.5x	3.1x

Figure 21 presents the benchmark results for the three versions of the Bellman-Ford privacy preservation protocol over sparse graphs. The edges in the sparse graphs are four times the number of the vertices, given by $m = 4n$. The result shows the influence of using public edges in computation and replacing prefixMin2 shown in Algorithm 12 and Algorithm 14 by getMin-function, which is constructed sparsely based on the publicity of the edges and vertices. In contrast, the privacy-preserving SSSD Bellman-Ford protocol versions for the dense graphs are presented in Figure 22.

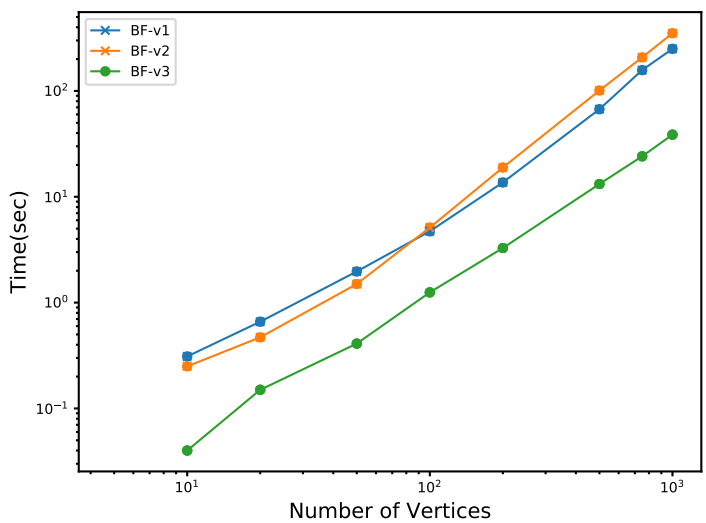


Figure 21: Running time (in seconds) of the privacy-preserving Bellman-Ford protocol versions on sparse graphs.

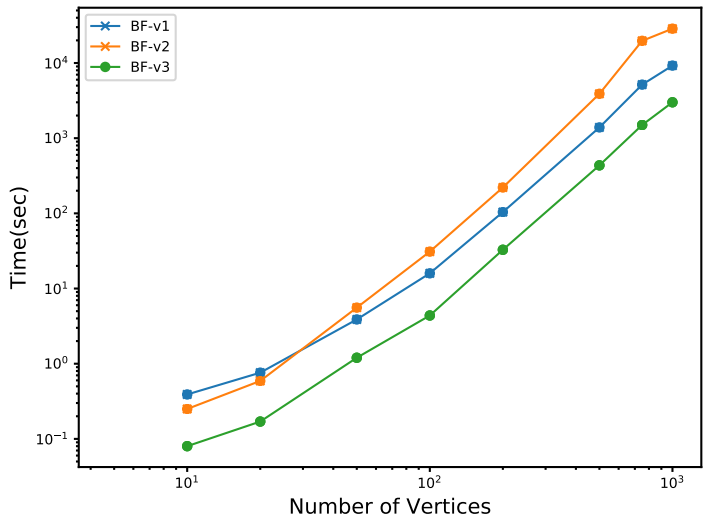


Figure 22: Running time (in seconds) of the privacy-preserving Bellman-Ford protocol versions on dense graphs.

10.3. Evaluation of the 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 26. The experiments also show data volume in different network environments. The running times of all graphs in various network environments for Algebraic path computation are lower than the running times of the Bellman-Ford protocol Version 3. As well as, the data volume in Algebraic path computation is less than the data volume in Bellman-Ford protocol Version 3 despite both protocols having a similar input data structure. Both have been designed to be fit for sparse representation of a graph. Also, both protocols have been constructed in the parallel SIMD framework.

Table 26: Estimated running time (data volume for a single computing server) for Bellman-Ford Version 3 and Algebraic path protocol in different network environments.

	Bellman-Ford Version 3 (BF-v3)				**Algebraic Path Computation (APC)**				**Improvement**		
	Data	**Running time (s)**			**Data**	**Running time (s)**			**BF-v3 vs. APC**		
G(A)	**volume**	HBLL	HBHL	LBHL	**volume**	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
1025	349 TB	46914854	116623074	273458923	–	–	–	–	–	–	–

In Table 26, the largest execution time is already measured for the Bellman-Ford Version 3, which is more than eight years. We benchmarked the larger examples by running only a few iterations, estimated the running time of a single iteration, and then multiplied with the total number of iterations, given by $(k \cdot k)$, where k is several rows/column

in a grid graph. Moreover, in Table 26, we documented the improvements of privacy-preserving APC protocol in different network environments compared with the running time of the Bellman-Ford Version 3. The APC protocol is faster than Bellman-Ford Version 3 tens of time, in particular, using big graphs.

In Figure 23, we present the comparison of Algebraic path computation and Version 3 of the Bellman-Ford protocol for different network environments. We see that despite the simple structure of the Bellman-Ford Version 3, Algebraic path computation is still faster also in high-latency settings. The running times that we measured are given in Appendix C.1.

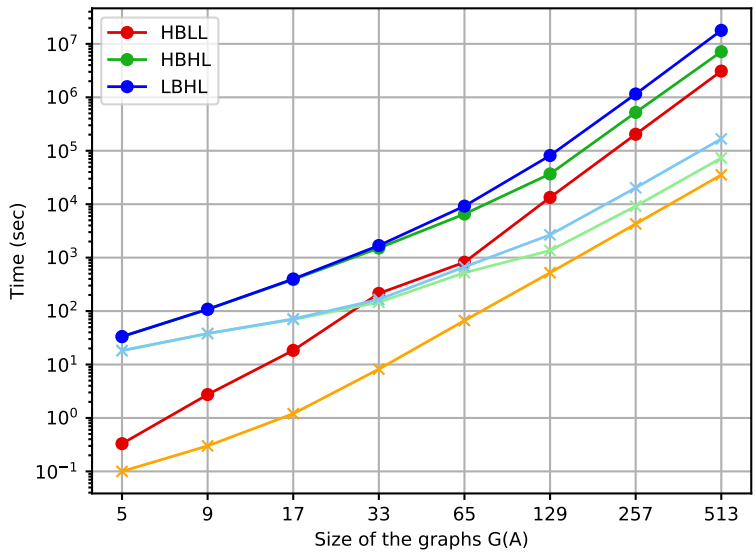


Figure 23: Performance (time in seconds) of Bellman-Ford Version 3 and Algebraic path computation protocols on graphs of different sizes in different network environments (red: HBLL, green: HBHL, blue: LBHL, dark: Bellman-Ford, light: Algebraic path computation).

11. CONCLUSIONS AND FUTURE WORK

This work presented how to use state-of-the-art algorithmic techniques of private parallel computation to implement privacy-preserving versions of some combinatorial and algebraic graph algorithms. These classical algorithms are SSSD and APSD, minimum spanning tree and forest, and finding the shortest path by the sparse linear system with a semiring framework. The SIMD parallel framework under the SecreC language has been used to implement those our new proposed protocols. In addition to securing SMC protocols in different deployments, we have implemented and benchmarked these protocols to understand their efficiency better. Extensive experiments have been conducted using different sizes and types of graphs for different purposes. Benchmarks are also conducted in different network environments to test the performance and analyze the proposed parallel methods.

We constructed different SSSD protocols based on essential algorithms on top of SMC protocols. The performance of these protocols depends on the size of the input graph and the number of edges. For example, Dijkstra's protocol for dense graphs is more efficient than Bellman-Ford's protocol for sparse graphs. If the shortest path is known to have only a few edges, the running time of the latter can be significantly reduced, often to much less than the total number of vertices in the graph. The SIMD parallel methods we proposed for constructing the SSSD protocols made changes to the main features of the SSSD algorithms. Some of the proposed algorithms are no longer greedy algorithms, in terms of round complexity. For example, the Radius-stepping and BFS protocols generally have logarithmic round complexities. This optimization reduced the round complexity of SMC protocols. Benchmark results show that the BFS protocol is more efficient than other protocols for dense graphs, and even for sparse graphs, it is still more efficient than others for more than the possible minimal number of edges in a graph. The running times we achieved are orders of magnitude faster than earlier studies in similar settings. The proposed SSSD protocols solved different problems for sparse, dense, like-planar, weighted, and unweighted graphs.

By exploiting parallel optimization for Dijkstra and Bellman-Ford protocols, we were able to construct the APSD Johnson protocol on top of secure multiparty computation protocols. Similar considerations apply to APSD and the choice between Floyd-Warshall, transitive closure, and Johnson protocols. Based on our benchmark results, we can gain insight into the possible running times of our protocols as subroutines for various applications with different network settings. No previous compilations and evaluations of privacy-preserving shortest paths have been performed. In particular, our extensive benchmark of APSD algorithms is unlike any previous work.

n addition to constructing shortest path protocols on top of secure multiparty computation protocols, we also developed minimum spanning tree and forest protocols on SMC protocols. We created a privacy-preserving protocol for finding MST using parallel SIMD, which is suitable for dense graph representation. Its running time has never been achieved before, especially for large dense graphs. To further optimize the protocol, we presented an optimized version of the privacy-preserving Prim's MST protocol, which can be faster with low round complexity. Additionally, we introduced a new protocol for simultaneously finding a minimum spanning tree for groups of graphs on top of secure multiparty computation protocols, which is called a minimum spanning forest. The proposed secure multiparty parallel computation of MSF protocol is noticeably double parallel, resulting in low running time. A privacy-preserving minimum spanning forest

protocol of this caliber has not been developed before. This state-of-the-art protocol can perform computations on big graphs that contain the maximum possible number of edges. These computations were previously avoided due to the high computational cost, such as for applications like hyperspectral images. We extensively benchmarked the proposed protocols using different graph sizes and types on various network settings. Moreover, in this thesis, we tried to construct and study another protocol for solving some problems in graph algorithms using different techniques. We exploited the solver of the sparse-linear system with a semiring framework to build a secure multiparty parallel computation of algebraic path protocol. We constructed the protocol and its related function sparsely, and its data input is also the sparse representation of a graph. The structure of the protocol and its related algorithm built sparsely to be fit with the efficiency of applying the parallel SIMD framework. This protocol has a different feature: using public edges of a graph with only private data for weights. We evaluated and benchmarked the algebraic path parallel computation with a proposed third version of the Bellman-Ford protocol with a similar feature, which is also the public edges of a given graph.

The variety of features offered by the proposed shortest path protocols provides a foundation for developing real-world applications. Shortest path protocols with public edges can be used to create navigation applications that do not require privacy over vertices and edges (locations). Our privacy-preserving parallel graph protocols contain novel essential details, particularly in adapting the Bellman-Ford algorithm to privacy-preserving computations. In this regard, the number of edges has no effect on running the minimum spanning tree and forest protocols

BIBLIOGRAPHY

[1] AS aCybernetica. “Basic constructions of secure multiparty computation”. In: *Applications of Secure Multiparty Computation* 13 (2015), p. 1.

[2] Rakesh Agrawal and Ramakrishnan Srikant. “Privacy-preserving data mining”. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 2000, pp. 439–450.

[3] Abdelrahaman Aly and Sara Cleemput. “An improved protocol for securely solving the shortest path problem and its application to combinatorial auctions”. In: *Cryptology ePrint Archive* (2017).

[4] Abdelrahaman Aly and Mathieu Van Vyve. “Securely solving classical network flow problems”. In: *International Conference on Information Security and Cryptology*. Springer. 2014, pp. 205–221.

[5] Abdelrahaman Aly et al. “Securely solving simple combinatorial graph problems”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2013, pp. 239–257.

[6] Gene M Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[7] Mohammad Anagreh and Peeter Laud. “A Parallel Privacy-Preserving Shortest Path Protocol from a Path Algebra Problem”. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2022 International Workshops, DPM 2022 and CBT 2022, Copenhagen, Denmark, September 26–30, 2022, Revised Selected Papers*. Springer. 2023, pp. 120–135.

[8] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. “Parallel Privacy-Preserving Shortest Path Algorithms”. In: *Cryptography* 5.4 (2021), p. 27.

[9] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. “Privacy-Preserving Parallel Computation of Minimum Spanning Forest”. In: *SN Computer Science* 3.6 (2022), pp. 1–19.

[10] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. “Privacy-preserving Parallel Computation of Shortest Path Algorithms with Low Round Complexity”. In: *Proceedings of the 8th International Conference on Information Systems Security and Privacy - ICISSP,* INSTICC. SciTePress, 2022, pp. 37–47. ISBN: 978-989-758-553-1. DOI: 10.5220/0010775700003120.

[11] Mohammad Anagreh, Eero Vainikko, and Peeter Laud. “Parallel Privacy-preserving Computation of Minimum Spanning Trees.” In: *ICISSP*. 2021, pp. 181–190.

[12] Mohammad Anagreh, Eero Vainikko, and Peeter Laud. “Parallel Privacy-Preserving Shortest Paths by Radius-Stepping”. In: *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2021, pp. 276–280.

[13] Baruch Awerbuch and Yossi Shiloach. “New connectivity and MSF algorithms for shuffle-exchange network and PRAM”. In: *IEEE Transactions on Computers* 36.10 (1987), pp. 1258–1263.

[14] Tim Baer, Raghavendra Kanakagiri, and Edgar Solomonik. “Parallel Minimum Spanning Forest Computation using Sparse Matrix Kernels”. In: *Proceedings of the 2022 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM. 2022, pp. 72–83.

[15] Scott Beamer, Krste Asanovic, and David Patterson. “Direction-optimizing breadth-first search”. In: *SC’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–10.

[16] Mihir Bellare et al. “Efficient garbling from a fixed-key block cipher”. In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 478–492.

[17] Richard Bellman. “On a routing problem”. In: *Quarterly of applied mathematics* 16.1 (1958), pp. 87–90.

[18] Assaf Ben-David, Noam Nisan, and Benny Pinkas. “FairplayMP: a system for secure multi-party computation”. In: *Proceedings of the 15th ACM conference on Computer and communications security*. 2008, pp. 257–266.

[19] Kevin Bernard et al. “A stochastic minimum spanning forest approach for spectral-spatial classification of hyperspectral images”. In: *2011 18th IEEE International Conference on Image Processing*. IEEE. 2011, pp. 1265–1268.

[20] Norman Biggs, Norman Linstead Biggs, and Biggs Norman. *Algebraic graph theory*. 67. Cambridge university press, 1993.

[21] Stefano Bistarelli. *Semirings for soft constraint solving and programming*. Vol. 2962. Springer Science & Business Media, 2004.

[22] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. “Semiring-based constraint satisfaction and optimization”. In: *Journal of the ACM (JACM)* 44.2 (1997), pp. 201–236.

[23] Stefano Bistarelli and Francesco Santini. “C-semiring frameworks for minimum spanning tree problems”. In: *International Workshop on Algebraic Development Techniques*. Springer. 2008, pp. 56–70.

[24] George Robert Blakley. “Safeguarding cryptographic keys”. In: *Managing Requirements Knowledge, International Workshop on*. IEEE Computer Society. 1979, pp. 313–313.

[25] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. “Data-oblivious graph algorithms for secure computation and outsourcing”. In: *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. 2013, pp. 207–218.

[26] Guy E Blelloch et al. “Parallel shortest paths using radius stepping”. In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*. 2016, pp. 443–454.

[27] Dan Bogdanov. “Sharemind: programmable secure computations with practical applications”. PhD thesis. Tartu University, 2013.

[28] Dan Bogdanov, Roman Jagomägis, and Sven Laur. “A universal toolkit for cryptographically secure privacy-preserving data mining”. In: *Pacific-Asia Workshop on Intelligence and Security Informatics*. Springer. 2012, pp. 112–126.

[29] Dan Bogdanov, Peeter Laud, and Jaak Randmets. “Domain-polymorphic programming of privacy-preserving applications”. In: *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*. 2014, pp. 53–65.

[30] Dan Bogdanov, Peeter Laud, and Jaak Randmets. “Domain-polymorphic programming of privacy-preserving applications”. In: *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*. 2014, pp. 53–65.

[31] Dan Bogdanov, Sven Laur, and Riivo Talviste. “A practical analysis of oblivious sorting algorithms for secure multi-party computation”. In: *Nordic Conference on Secure IT Systems*. Springer. 2014, pp. 59–74.

[32] Dan Bogdanov, Sven Laur, and Jan Willemson. “Sharemind: A framework for fast privacy-preserving computations”. In: *European Symposium on Research in Computer Security*. Springer. 2008, pp. 192–206.

[33] Dan Bogdanov, Riivo Talviste, and Jan Willemson. “Deploying secure multi-party computation for financial data analysis”. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2012, pp. 57–64.

[34] Dan Bogdanov et al. “High-performance secure multi-party computation for data mining applications”. In: *International Journal of Information Security* 11.6 (2012), pp. 403–418.

[35] Dan Bogdanov et al. “How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation”. In: *International conference on financial cryptography and data security*. Springer. 2015, pp. 227–234.

[36] Dan Bogdanov et al. “Implementation and evaluation of an algorithm for cryptographically private principal component analysis on genomic data”. In: *IEEE/ACM transactions on computational biology and bioinformatics* 15.5 (2018), pp. 1427–1432.

[37] Dan Bogdanov et al. “Privacy-preserving statistical data analysis on federated databases”. In: *Annual Privacy Forum*. Springer. 2014, pp. 30–55.

[38] Dan Bogdanov et al. “Secure multi-party data analysis: end user validation and practical experiments”. In: *Cryptology ePrint Archive* (2013).

[39] Dan Bogdanov et al. “Students and Taxes: a Privacy-Preserving Study Using Secure Computation.” In: *Proc. Priv. Enhancing Technol.* 2016.3 (2016), pp. 117–135.

[40] Bruce Boldon, Narsingh Deo, and Nishit Kumar. “Minimum-weight degree-constrained spanning tree problem: Heuristics and implementation on an SIMD parallel machine”. In: *Parallel Computing* 22.3 (1996), pp. 369–382.

[41] Béla Bollobás. *Modern graph theory*. Vol. 184. Springer Science & Business Media, 1998.

[42] Otakar Borvka. “O jistém problému minimálnım”. In: (1926).

[43] Elette Boyle, Kai-Min Chung, and Rafael Pass. “Large-scale secure computation: Multi-party computation for (parallel) RAM programs”. In: *Annual Cryptology Conference*. Springer. 2015, pp. 742–762.

[44] Elette Boyle et al. “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.

[45] Justin Brickell and Vitaly Shmatikov. “Privacy-preserving graph algorithms in the semi-honest model”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2005, pp. 236–252.

[46] Martin Burkhart et al. “{SEPIA}:{Privacy-Preserving} Aggregation of {Multi-Domain} Network Events and Statistics”. In: *19th USENIX Security Symposium (USENIX Security 10)*. 2010.

[47] Ran Canetti. “Security and composition of multiparty cryptographic protocols”. In: *Journal of CRYPTOLOGY* 13.1 (2000), pp. 143–202.

[48] Ran Canetti. “Universally composable security: A new paradigm for cryptographic protocols”. In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE. 2001, pp. 136–145.

[49] Ran Canetti et al. “Adaptively secure multi-party computation”. In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 639–648.

[50] Henry Carter et al. “Secure outsourced garbled circuit evaluation for mobile devices”. In: *Journal of Computer Security* 24.2 (2016), pp. 137–180.

[51] Chein-I Chang. *Hyperspectral data exploitation: theory and applications*. John Wiley & Sons, 2007.

[52] Chein-I Chang. *Hyperspectral imaging: techniques for spectral detection and classification*. Vol. 1. Springer Science & Business Media, 2003.

[53] David Chaum, Claude Crépeau, and Ivan Damgard. “Multiparty unconditionally secure protocols”. In: *Proceedings of the twentieth annual ACM symposium on Theory of computing*. 1988, pp. 11–19.

[54] David Cheriton and Robert Endre Tarjan. “Finding minimum spanning trees”. In: *SIAM journal on computing* 5.4 (1976), pp. 724–742.

[55] Sun Chung and Anne Condon. “Parallel implementation of Bouvka’s minimum spanning tree algorithm”. In: *Proceedings of International Conference on Parallel Processing*. IEEE. 1996, pp. 302–308.

[56] Ran Cohen et al. “Round-preserving parallel composition of probabilistic-termination cryptographic protocols”. In: *Journal of Cryptology* 34.2 (2021), pp. 1–57.

[57] Leiserson Cormen, Charles E Leiserson, and L Ronald. *Rives: Introduction to Algorithms*. 1990.

[58] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[59] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

[60] Ronald Cramer, Ivan Bjerre Damgård, et al. *Secure multiparty computation*. Cambridge University Press, 2015.

[61] Ivan Damgård and Jesper Buus Nielsen. “Universally composable efficient multiparty computation from threshold homomorphic encryption”. In: *Annual International Cryptology Conference*. Springer. 2003, pp. 247–264.

[62] Ivan Damgård et al. “Asynchronous multiparty computation: Theory and implementation”. In: *International workshop on public key cryptography*. Springer. 2009, pp. 160–179.

[63] Ivan Damgård et al. “Multiparty computation from somewhat homomorphic encryption”. In: *Annual Cryptology Conference*. Springer. 2012, pp. 643–662.

[64] Daniel Demmler, Thomas Schneider, and Michael Zohner. “ABY-A framework for efficient mixed-protocol secure two-party computation.” In: *NDSS*. 2015.

[65] Edsger W Dijkstra et al. “A note on two problems in connexion with graphs”. In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[66] K Erciyes. *Algebraic Graph algorithms*. Springer, 2021.

[67] Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.

[68] Jittat Fakcharoenphol and Satish Rao. “Planar graphs, negative weight edges, shortest paths, and near linear time”. In: *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE. 2001, pp. 232–241.

[69] Pedro F Felzenszwalb and Daniel P Huttenlocher. “Efficient graph-based image segmentation”. In: *International journal of computer vision* 59.2 (2004), pp. 167–181.

[70] Eugene Fink. *A survey of sequential and systolic algorithms for the algebraic path problem*. Faculty of Mathematics, University of Waterloo, 1992.

[71] Robert W Floyd. "Algorithm 97: shortest path". In: *Communications of the ACM* 5.6 (1962), p. 345.

[72] Michael J Flynn. "Some computer organizations and their effectiveness". In: *IEEE transactions on computers* 100.9 (1972), pp. 948–960.

[73] Michael J Flynn. "Very high-speed computing systems". In: *Proceedings of the IEEE* 54.12 (1966), pp. 1901–1909.

[74] Leslie R Foulds. *Graph theory applications*. Springer Science & Business Media, 2012.

[75] Tore Kasper Frederiksen et al. "Minilego: Efficient secure two-party computation from general assumptions". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2013, pp. 537–556.

[76] Michael L Fredman and Robert Endre Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[77] Tanvi Garg et al. "A survey on security and privacy issues in IoV." In: *International Journal of Electrical & Computer Engineering (2088-8708)* 10.5 (2020).

[78] Martin Geisler. "Cryptographic protocols: theory and implementation". In: *PhD thesis, University of Aarhus* (2010).

[79] Rosario Gennaro, Michael O Rabin, and Tal Rabin. "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*. 1998, pp. 101–111.

[80] Craig Gentry et al. "Garbled RAM revisited". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2014, pp. 405–422.

[81] Jonathan S Golan. *Semirings and affine equations over them: theory and applications*. Vol. 556. Springer Science & Business Media, 2013.

[82] Oded Goldreich, Silvio Micali, and Avi Wigderson. "How to play any mental game, or a completeness theorem for protocols with honest major-

ity". In: *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*. 2019, pp. 307–328.

[83] Oded Goldreich and Rafail Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM (JACM)* 43.3 (1996), pp. 431–473.

[84] S Goldwasser, M Ben-Or, and A Wigderson. "Completeness theorems for non-cryptographic fault-tolerant distributed computing". In: *Proc. of the 20th STOC*. 1988, pp. 1–10.

[85] Ronald L Graham and Pavol Hell. "On the history of the minimum spanning tree problem". In: *Annals of the History of Computing* 7.1 (1985), pp. 43–57.

[86] Alexander E Guterman. "over semirings". In: *Surveys in contemporary mathematics* 347 (2008), p. 1.

[87] Sung-Chul Han and SC Kang. *Optimizing all pairs shortest path algorithm using vector instructions*. 2005.

[88] Robert M Haralick and Linda G Shapiro. "Image segmentation techniques". In: *Computer vision, graphics, and image processing* 29.1 (1985), pp. 100–132.

[89] Wilko Henecka et al. "TASTY: tool for automating secure two-party computations". In: *Proceedings of the 17th ACM conference on Computer and communications security*. 2010, pp. 451–462.

[90] Monika R Henzinger et al. "Faster shortest-path algorithms for planar graphs". In: *journal of computer and system sciences* 55.1 (1997), pp. 3–23.

[91] W Daniel Hillis and Guy L Steele Jr. "Data parallel algorithms". In: *Communications of the ACM* 29.12 (1986), pp. 1170–1183.

[92] Andrew Hodges. *Alan Turing: the enigma*. Princeton University Press, 2014.

[93] Joosep Jääger and Alisa Pankova. "PrivaLog: a privacy-aware logic programming language". In: *23rd International Symposium on Principles and Practice of Declarative Programming*. 2021, pp. 1–14.

[94] R Jagomägis. "A programming language for creating privacy-preserving applications". In: *Bachelor's thesis, Institute of Computer Science, University of Tartu* (2008).

[95] Pelle Jakovits. “Adapting scientific computing algorithms to distributed computing frameworks”. PhD thesis. Ph. D. thesis, University of Tartu, Tartu, 2017.

[96] Donald B Johnson. “Efficient algorithms for shortest paths in sparse networks”. In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 1–13.

[97] Donald B Johnson and Panagiotis Metaxas. “A parallel algorithm for computing minimum spanning trees”. In: *Journal of Algorithms* 19.3 (1995), pp. 383–401.

[98] Liina Kamm. “Privacy-preserving statistical analysis using secure multiparty computation”. In: *Ph.D. dissertation* (2015).

[99] Liina Kamm et al. “A new way to protect privacy in large-scale genome-wide association studies”. In: *Bioinformatics* 29.7 (2013), pp. 886–893.

[100] Vidhi Kapoor et al. “Privacy issues in wearable technology: An intrinsic review”. In: *Proceedings of the International Conference on Innovative Computing & Communications (ICICC)*. 2020.

[101] Alan H Karp and Horace P Flatt. “Measuring parallel processor performance”. In: *Communications of the ACM* 33.5 (1990), pp. 539–543.

[102] Jonathan Katz and Chiu-Yuen Koo. “Round-efficient secure computation in point-to-point networks”. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2007, pp. 311–328.

[103] Jonathan Katz, Rafail Ostrovsky, and Adam Smith. “Round efficiency of multi-party computation with a dishonest majority”. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2003, pp. 578–595.

[104] Marcel Keller and Peter Scholl. “Efficient, oblivious data structures for MPC”. In: *International Conference on the Theory and Application of Cryptology and Information Security*. Springer. 2014, pp. 506–525.

[105] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.

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

[107] Philip Klein and Clifford Stein. “A parallel algorithm for eliminating cycles in undirected graphs”. In: *Information Processing Letters* 34.6 (1990), pp. 307–312.

[108] Philip N Klein, Shay Mozes, and Oren Weimann. “Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n\log^2 n)$-time algorithm”. In: *ACM Transactions on Algorithms (TALG)* 6.2 (2010), pp. 1–18.

[109] Philip N Klein and Sairam Subramanian. “A randomized parallel algorithm for single-source shortest paths”. In: *Journal of Algorithms* 25.2 (1997), pp. 205–220.

[110] Joseph B Kruskal. “On the shortest spanning subtree of a graph and the traveling salesman problem”. In: *Proceedings of the American Mathematical society* 7.1 (1956), pp. 48–50.

[111] Richard E Ladner and Michael J Fischer. “Parallel prefix computation”. In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 831–838.

[112] P Laud and L Kamm. “Stateful abstractions of secure multiparty computation”. In: *Applications of Secure Multiparty Computation. Cryptology and Information Security* 13 (2015), pp. 26–42.

[113] Peeter Laud. “Linear-time oblivious permutations for SPDZ”. In: *International Conference on Cryptology and Network Security*. Springer. 2021, pp. 245–252.

[114] Peeter Laud. “Parallel oblivious array access for secure multiparty computation and privacy-preserving minimum spanning trees”. In: *Proceedings on Privacy Enhancing Technologies* 2015.2 (2015), pp. 188–205.

[115] Sven Laur and Pille Pullonen-Raudvere. “Foundations of Programmable Secure Computation”. In: *Cryptography* 5.3 (2021), p. 22.

[116] Sven Laur, Jan Willemson, and Bingsheng Zhang. “Round-efficient oblivious database manipulation”. In: *International Conference on Information Security*. Springer. 2011, pp. 262–277.

[117] M Leyzorek et al. “Investigation of model techniques–first annual report–6 June 1956–1 July 1957–a study of model techniques for communication systems”. In: *Case Institute of Technology, Cleveland, Ohio* (1957).

[118] Yehida Lindell. “Secure multiparty computation for privacy preserving data mining”. In: *Encyclopedia of Data Warehousing and Mining*. IGI global, 2005, pp. 1005–1009.

[119] Yehuda Lindell. “Fast cut-and-choose-based protocols for malicious and covert adversaries”. In: *Journal of Cryptology* 29.2 (2016), pp. 456–490.

[120] Yehuda Lindell and Benny Pinkas. “Privacy preserving data mining”. In: *Annual International Cryptology Conference*. Springer. 2000, pp. 36–54.

[121] Yehuda Lindell and Ben Riva. “Cut-and-choose Yao-based secure computation in the online/offline and batch settings”. In: *Annual Cryptology Conference*. Springer. 2014, pp. 476–494.

[122] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. “Generalized nested dissection”. In: *SIAM journal on numerical analysis* 16.2 (1979), pp. 346–358.

[123] Richard J Lipton and Robert Endre Tarjan. “A separator theorem for planar graphs”. In: *SIAM Journal on Applied Mathematics* 36.2 (1979), pp. 177–189.

[124] Chang Liu et al. “Automating efficient RAM-model secure computation”. In: *2014 IEEE Symposium on Security and Privacy*. IEEE. 2014, pp. 623–638.

[125] Chang Liu et al. “Oblivm: A programming framework for secure computation”. In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 359–376.

[126] Kazuya Matsumoto, Naohito Nakasato, and Stanislav G Sedukhin. “Blocked united algorithm for the all-pairs shortest paths problem on hybrid CPU-GPU systems”. In: *IEICE TRANSACTIONS on Information and Systems* 95.12 (2012), pp. 2759–2768.

[127] Ricardo Mendes and João P Vilela. “Privacy-preserving data mining: methods, metrics, and applications”. In: *IEEE Access* 5 (2017), pp. 10562–10582.

[128] Ulrich Meyer. “Design and analysis of sequential and parallel single-source shortest-paths algorithms”. In: (2002).

[129] Ulrich Meyer and Peter Sanders. “Δ-stepping: a parallelizable shortest path algorithm”. In: *Journal of Algorithms* 49.1 (2003), pp. 114–152.

[130] Mehryar Mohri et al. “Semiring frameworks and algorithms for shortest-distance problems”. In: *Journal of Automata, Languages and Combinatorics* 7.3 (2002), pp. 321–350.

[131] Shay Mozes and Christian Wulff-Nilsen. “Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log \log n)$ time”. In: *European Symposium on Algorithms*. Springer. 2010, pp. 206–217.

[132] S Nagavalli. “Dynamic Optimization—Using Hardware Parallelism for Faster Search via Dynamic Programming”. In: *Project Report, Carnegie-Mellon University. 2013. Available online:* (2013). DOI: https://www.andrew.cmu.edu/user/snagaval/16-745/Project/16-745-Project-Report-SasankaNagavalli.pdf.

[133] AS Nepomniaschaya. “Concurrent selection of the shortest paths and distances in directed graphs using vertical processing systems”. In: *Bulletin of the Novosibirsk Computing Center* 19 (2003), pp. 61–72.

[134] Andre Ostrak et al. “Implementing Privacy-Preserving Genotype Analysis with Consideration for Population Stratification”. In: *Cryptography* 5.3 (2021), p. 21.

[135] Victor Pan and John Reif. “Fast and efficient parallel solution of sparse linear systems”. In: *SIAM Journal on Computing* 22.6 (1993), pp. 1227–1250.

[136] Victor Pan and John Reif. “Fast and efficient solution of path algebra problems”. In: *Journal of Computer and System Sciences* 38.3 (1989), pp. 494–510.

[137] Victor Pan and John Reif. “The parallel computation of minimum cost paths in graphs by stream contraction”. In: *Information Processing Letters* 40.2 (1991), pp. 79–83.

[138] Alisa Pankova. “Efficient multiparty computation secure against covert and active adversaries”. In: *Ph. D. dissertation* (2017).

[139] Alisa Pankova and Joosep Jääger. “Short Paper: Secure Multiparty Logic Programming”. In: *Proceedings of the 15th Workshop on Programming Languages and Analysis for Security*. 2020, pp. 3–7.

[140] Robert Pike et al. “A minimum spanning forest based hyperspectral image classification method for cancerous tissue detection”. In: *Medical Imaging 2014: Image Processing*. Vol. 9034. SPIE. 2014, pp. 481–488.

[141] Alessandro Pinto, Luca P Carloni, and Alberto L Sangiovanni-Vincentelli. “Efficient synthesis of networks on chip”. In: *Proceedings 21st International Conference on Computer Design*. IEEE. 2003, pp. 146–150.

[142] Maurice Pollack and Walter Wiebenson. “Solutions of the shortest-route problem—A review”. In: *Operations Research* 8.2 (1960), pp. 224–230.

[143] Robert Clay Prim. “Shortest connection networks and some generalizations”. In: *The Bell System Technical Journal* 36.6 (1957), pp. 1389–1401.

[144] Sara Ramezanian, Tommi Meskanen, and Valtteri Niemi. “Privacy preserving shortest path queries on directed graph”. In: *2018 22nd Conference of Open Innovations Association (FRUCT)*. IEEE. 2018, pp. 217–223.

[145] Jaak Randmets. “Programming Languages for Secure Multi-party Computation Application Development”. PhD thesis. Tartu University, 2017.

[146] CH Koteswara Rao and Kunwar Singh. “Securely solving privacy preserving minimum spanning tree algorithms in semi-honest model”. In: *International Journal of Ad Hoc and Ubiquitous Computing* 34.1 (2020), pp. 1–10.

[147] David P Rodgers. “Improvements in multiprocessor system design”. In: *ACM SIGARCH Computer Architecture News* 13.3 (1985), pp. 225–231.

[148] Ali Saglam and Nurdan Akhan Baykan. “Sequential image segmentation based on minimum spanning tree representation”. In: *Pattern Recognition Letters* 87 (2017), pp. 155–162.

[149] Robert Scheichl and Eero Vainikko. “Additive Schwarz with aggregation-based coarsening for elliptic problems with highly variable coefficients”. In: *Computing* 80.4 (2007), pp. 319–343.

[150] Adam Sealfon. “Shortest paths and distances with differential privacy”. In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2016, pp. 29–41.

[151] Robert Sedgewick and Kevin Daniel Wayne. *Algorithms (4th ed.)* Addison-Wesley, 2011.

[152] Adi Shamir. “How to share a secret”. In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[153] Adi Shamir, Ronald L Rivest, and Leonard M Adleman. “Mental poker”. In: *The mathematical gardner*. Springer, 1981, pp. 37–43.

[154] Jean Stawiaski. “Mathematical morphology and graphs: Application to interactive medical image segmentation”. PhD thesis. École Nationale Supérieure des Mines de Paris, 2008.

[155] Francis Suraweera. “A fast algorithm for the minimum spanning tree”. In: *Computers in industry* 13.2 (1989), pp. 181–185.

[156] Francis Suraweera and Prabir Bhattacharya. “A parallel algorithm for the minimum spanning tree on an SIMD machine”. In: *Proceedings of the 1992 ACM annual conference on Communications*. 1992, pp. 473–476.

[157] Yasuhiro Takei, Masanori Hariyama, and Michitaka Kameyama. “Evaluation of an FPGA-based shortest-path-search accelerator”. In: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*. The Steering Committee of The World Congress in Computer Science, Computer . . . 2015, p. 613.

[158] Riivo Talviste et al. “Applying secure multi-party computation in practice”. In: *Ph. D. dissertation* (2016).

[159] Mikkel Thorup. “Integer priority queues with decrease key in constant time and the single source shortest paths problem”. In: *Journal of Computer and System Sciences* 69.3 (2004), pp. 330–353.

[160] Tomas Toft et al. “Primitives and applications for multi-party computation”. In: *Unpublished doctoral dissertation, University of Aarhus, Denmark* (2007).

[161] Jesper L Träff and Christos D Zaroliagis. “A simple parallel algorithm for the single-source shortest path problem on planar digraphs”. In: *Journal of Parallel and Distributed Computing* 60.9 (2000), pp. 1103–1124.

[162] Vibhav Vineet et al. “Fast minimum spanning tree for large graphs on the GPU”. In: *Proceedings of the Conference on High Performance Graphics 2009*. 2009, pp. 167–171.

[163] Bin Wang and Jennifer C Hou. “Multicast routing and its QoS extension: problems, algorithms, and protocols”. In: *IEEE network* 14.1 (2000), pp. 22–36.

[164] Wei Wang et al. “GPU-based fast minimum spanning tree using data parallel primitives”. In: *2010 2nd International Conference on Information Engineering and Computer Science*. IEEE. 2010, pp. 1–4.

[165] Xiao Shaun Wang et al. “Oblivious data structures”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 215–226.

[166] Jan Wassenberg, Wolfgang Middelmann, and Peter Sanders. “An efficient parallel algorithm for graph-based image segmentation”. In: *International*

Conference on Computer Analysis of Images and Patterns. Springer. 2009, pp. 1003–1010.

[167] Douglas Brent West et al. *Introduction to graph theory*. Vol. 2. Prentice hall Upper Saddle River, 2001.

[168] Avi Wigderson, MB Or, and S Goldwasser. "Completeness theorems for noncryptographic fault-tolerant distributed computations". In: *Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC'88)*. 1988, pp. 1–10.

[169] David J Wu et al. "Privacy-preserving shortest path computation". In: *arXiv preprint arXiv:1601.02281* (2016).

[170] Takeo Yamada. "A mini–max spanning forest approach to the political districting problem". In: *International Journal of Systems Science* 40.5 (2009), pp. 471–477.

[171] Takeo Yamada, Hideo Takahashi, and Seiji Kataoka. "A heuristic algorithm for the mini-max spanning forest problem". In: *European Journal of Operational Research* 91.3 (1996), pp. 565–572.

[172] Andrew C Yao. "Protocols for secure computations". In: *23rd annual symposium on foundations of computer science (sfcs 1982)*. IEEE. 1982, pp. 160–164.

Appendix A. SINGLE-SOURCE SHORTEST DISTANCE PROTOCOLS

A.1. Running time (in second) and data volume (in MB) for Privacy-preserving Bellman-Ford Protocols via Sharemind Cluster

Table 27: Privacy-preserving Bellman-Ford version 1 and 2 in HB-LL environment.

Graph		Privacy-Preserving Bellman-Ford V.2						Privacy-Preserving Bellman-Ford V.1					
		Server-1		Server-2		Server-3		Server-1		Server-2		Server-3	
n	m	Time	$\mathcal{DV}_1$	Time	$\mathcal{DV}_2$	Time	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	Time	$\mathcal{DV}_2$	Time	$\mathcal{DV}_3$
50	400	2.8	36	2.8	35	2.8	34.2	2.8	34.3	2.8	34.0	2.8	31.3
50	1225	7.2	206	7.2	193.4	7.2	203	5.2	74.5	5.2	71.4	5.2	74.1
200	600	22.2	462	22.2	434	22.2	455	17.9	162	17.9	155	17.9	160
200	19.9k	356	15922	356	14889	356	15761	162	2921	162	2773	162	2905
700	2100	237	6320	237	5981	237	6324	147	1531	147	1481	147	1548
700	10k	661	27438	661	25647	661	27155	348	5045	348	4783	348	5011
700	244k	16.4k	821826	16.4k	756706	16.4k	800287	5823	136308	5823	117524	5823	212601
1k	3k	435	12845	435	12015	435	12709	287	2970	287	2861	287	2989
1k	20k	1879	81816	1879	76432	1879	80938	916.5	13576	916.5	12880	916.5	13506
1k	499k	49.8k	2328681	49.8k	2152942	49.8k	2266266	16.6k	647408	16.6k	591950	16.6k	585171
3k	9k	3889	132470	3889	123805	3889	131038	2318	24707	2318	23793	2318	24892
3k	50k	15k	656517	15k	613129	15k	649490	6675	99518	6675	94588	6675	99221

Table 28: Privacy-preserving Bellman-Ford version 1 and 2 in LB-HL environment.

Graph		Privacy-Preserving Bellman-Ford V.2						Privacy-Preserving Bellman-Ford V.1					
		Server-1		Server-2		Server-3		Server-1		Server-2		Server-3	
n	m	Time	$\mathcal{DV}_1$	Time	$\mathcal{DV}_2$	Time	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	Time	$\mathcal{DV}_2$	Time	$\mathcal{DV}_3$
50	400	451	66.5	451	62.9	451	66.0	799	32.7	799	31.6	799	32.5
50	1225	558	210	558	197	558	208	1145	77.0	1144	74.2	1145	76.7
200	600	1841	470	1841	442	1841	465	3260	169	3260	162	3261	167
200	19.9k	4219	15883	4219	14967	4219	15996	5806	2941	5806	2791	5806	2924
700	2100	7956	13651	7956	12920	7956	13538	15.4k	1580	15.4k	1506	15.4k	1565
700	10k	11.1k	25824	11.1k	27611	11.1k	25824	18.7k	5112	18.7k	4849	18.7k	5073
700	244k	42.5k	819421	42.5k	778308	42.5k	820166	42.9k	110443	42.9k	104449	42.8k	109567
1k	3k	13.5k	13375	13.5k	45433	13.5k	45617	22.2k	5112	22.2k	4849	22.2k	5074
1k	20k	20.6k	82711	20.6k	84188	20.6k	81755	26.7k	13759	26.7k	12980	26.7k	13608
1k	499k	318k	2504753	318k	2321886	318k	2461767	81.7k	295402	81.7k	296271	81.7k	300797
3k	9k	49.3k	177412	49.3k	229888	49.3k	221851	81.1k	31449	81.1k	172817	81.1k	175074
3k	50k	104k	665504	104k	571230	104k	608287	100k	157142	100k	229054	100k	235415

Table 29: Privacy-preserving Bellman-Ford version 1 and 2 in HB-HL environment.

Graph		Privacy-Preserving Bellman-Ford V.2						Privacy-Preserving Bellman-Ford V.1					
		Server-1		Server-2		Server-3		Server-1		Server-2		Server-3	
n	m	Time	$\mathcal{DV}_1$	Time	$\mathcal{DV}_2$	Time	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	Time	$\mathcal{DV}_2$	Time	$\mathcal{DV}_3$
50	400	448	65.9	448	62.2	448	65.3	799	32.5	799	31.5	799	32.4
50	1225	550	207.6	550	195.2	550	205.8	1155	77.6	1155	74.7	1155	77.3
200	600	1823	466.5	1823	437.9	1823	461.3	3270	169.8	3270	163.1	3270	168.5
200	19.9k	3122	15960	3122	14927	3122	15799	5626	2945.1	5262	2797.3	5626	2926.8
700	2100	7529	6365.0	7529	5958.8	7529	6299.8	15.4k	1577.9	15.4k	1711.1	15.4k	1774.0
700	10k	9158	27513	9158	25701	9158	27212	18.5k	5086.2	18.5k	4816.9	18.5k	5042.5
700	244k	54.8k	422087	54.8k	368673	54.8k	368673	35.9k	110857	35.9k	104008	35.9k	109986
1k	3k	10.8k	12727	10.8k	8544.9	10.8k	11749	22.0k	3274.3	22.0k	3670.4	22.0k	4070.9
1k	20k	14.8k	100221	14.8k	92452	14.8k	100096	25.9k	11918	25.9k	3958.8	25.9k	9522.2
1k	499k	156k	2508184	156k	2398777	156k	2544635	63.4k	325231	63.4k	302641	63.4k	320396
3k	9k	39.8k	151334	39.8k	144723	39.8k	160039	79.6k	48175	79.6k	70630	79.6k	65083
3k	50k	56.3k	620549	56.3k	433511	56.3k	468711	93.3k	131648	93.3k	133118	93.3k	131361

A.2. Running time (in second) and data volume (in MB) for Privacy-preserving Dijkstra Protocols via Sharemind Cluster.

Table 30: Privacy-preserving Dijkstra version 1 and 2 in HB-LL environment.

Graph			Sequential SIMD-Dijkstra (V1)				SIMD nDijkstra (V2)				Speed-up
			S1	S2	S3	Time	S1	S2	S3	Time	
G	n	m	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	x-time
10	10	25	3.14	2.48	3.41	0.57	3.8	3.3	2.1	0.22	2.6x
10	10	45	3.1	2.97	3.4	0.56	3.0	2.9	2.1	0.21	2.6x
25	25	300	34.6	33.4	34.9	4.40	25.6	24.1	25.1	1.5	2.9x
50	50	400	215	206	207	24.2	195	184	189	8.1	3.0x
50	50	1225	211	204	207	24.7	196	185	191	8.3	2.9x
100	100	1500	1329	1274	1301	137	1550	1458	1517	52.9	2.6x
100	100	4950	1335	1281	1310	138	1550	1459	1520	54.1	2.6x
200	200	19.9k	8802	8401	8638	847	12133	11727	12587	638	1.3x
500	500	1500	116368	110496	113907	10.2k	193497	178121	189132	8319	1.2x
500	500	124k	138050	134935	156069	10.4k	230800	2000093	207747	8560	1.2x
1k	1k	499k	873140	853441	819238	77.3k	1600555	1473759	1568905	49.5k	1.5x

Table 31: Privacy-preserving Dijkstra version 1 and 2 in LB-HL environment.

Graph			Sequential SIMD-Dijkstra (V1)				SIMD nDijkstra (V2)				Speed-up
			S1	S2	S3	Time	S1	S2	S3	Time	
G	n	m	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	x-time
10	10	25	3.8	3.6	3.8	214	2.17	2.08	2.16	54.1	3.9x
10	10	45	3.8	3.6	3.7	214	2.18	2.08	2.16	54.1	3.9x
25	25	300	38.6	36.7	37.8	1895	26.1	24.6	25.1	159	11.8x
50	50	400	223	226	216	9445	198	185	194	382	24.7x
50	50	1225	226	214	221	9446	198	186	194	382	24.7x
100	100	1500	2970	2808	2911	45.7k	1564	1470	1530	1083	42.1x
100	100	4950	2971	2858	2903	45.7k	1615	1520	1581	1061	43.1x
200	200	19.9k	9171	8702	8981	216k	12485	11724	12207	3826	56.5x
500	500	1500	119694	113771	117458	1643k	193728	188411	195987	37.3k	44.0x
500	500	124k	119870	113938	117625	1642k	202755	199403	207053	37.1k	44.2x
1k	1k	499k	886069	841861	869319	7473k	1619544	1579515	1657197	253k	29.4x

Table 32: Privacy-preserving Dijkstra version 1 and 2 in HB-HL environment.

Graph			Sequential SIMD-Dijkstra (V1)				SIMD nDijkstra (V2)				Speed-up
			S1	S2	S3	Time	S1	S2	S3	Time	
G	n	m	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	x-time
10	10	25	3.7	3.6	3.7	214	2.18	2.08	2.17	54.1	3.9x
10	10	45	3.7	3.6	3.7	214	2.18	2.08	2.16	54.1	3.9x
25	25	300	38.5	36.6	37.7	1894	25.8	24.3	25.3	158	11.9x
50	50	400	226	215	221	9444	197	185	193	368	25.6x
50	50	1225	228	217	224	9446	197	185	193	368	25.6x
100	100	1500	294	295	304	45.7k	1558	1464	1525	946	48.3x
100	100	4950	1398	1332	1373	45.7k	1558	1464	1525	932	49.1x
200	200	19.9k	9144	8671	8957	215k	12423	11672	12161	2835	75.8x
500	500	1500	118664	112755	116390	1637k	199324	192945	205934	20.9k	78.3x
500	500	124k	119263	113845	117131	1636k	198790	195059	208614	20.9k	78.2x
1k	1k	499k	904772	852524	884220	7421k	1619420	1566238	1675519	126k	58.5x

A.3. Privacy-preserving SIMD-nDijkstra via Shremind Cluster

Table 33: Running time (in second) of nDijkstra via various Networks environments.

Graphs			High Band.- Low Latency			Low Band.- High Latency			High Band.- High Latency		
G	n	m	Perm.	Loop	Total	Perm.	Loop	Total	Perm.	Loop	Total
5	50	1225	0.39	1.21	1.6	3.20	323	326	3.2	322	326
10	50	1225	0.71	1.87	2.60	6.40	325	331	6.4	323	329
25	50	1225	0.15	2.48	2.60	12.9	329	341	12.7	324	336
10	100	4950	2.30	5.30	7.50	8.90	746	755	8.7	736	745
25	100	4950	5.70	10.3	16.1	22.5	769	792	21.7	742	764
50	100	4950	10.5	18.2	28.6	44.8	807	852	43.3	750	793
75	100	4950	16.2	26.4	42.6	70.3	850	920	68.9	765	834
20	200	19k	16.2	29.6	46.0	35.1	1757	1792	34.7	1664	1698
50	200	19k	38.1	66.7	104	90.3	1985	2075	83.0	1717	1800
100	200	19k	75.1	122	189	194	2592	2787	181	1956	2138
100	500	5k	496	800	1296	940	9892	10832	922	6724	7647
100	500	124k	456	592	1049	927	9907	10835	900	6624	7524
50	1k	5k	967	2078	3045	1535	21296	22832	1189	14221	15410
50	1k	499k	961	2088	3049	1530	19460	20990	1234	14474	15708
500	1k	499k	9477	17.6k	27.1k	16239	109093	125332	12547	49632	62179
100	5k	12.4M	48.7k	181.3k	230k	59.2k	589.7k	648.9k	50983	338632	389616

Appendix B. ALL-PAIRS SHORTEST DISTANCE PROTOCOLS

B.1. Privacy-preserving APSD protocols via Shremind Cluster

Table 34: Running time (in second) and data volume (in MB) for Privacy-preserving APSD protocols on HB-LL

Graph	Privacy-Preserving Johnson V1				Privacy-Preserving Johnson V2				Privacy-Preserving FW				Privacy-Preserving TC			
	S1	S2	S3		S1	S2	S3		S1	S2	S3		S1	S2	S3	
n	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time
5	1.7	1.6	2.8	0.37	1.8	1.8	2.1	0.27	0.6	0.2	0.1	0.01	1.02	0.52	0.16	0.02
10	11.6	11.4	12.6	1.16	6.0	6.1	6.9	0.57	2.9	2.9	2.8	0.03	3.76	3.67	1.59	0.05
20	72.4	69.68	71.4	5.21	32.2	30.1	33.9	1.67	5.1	4.9	3.5	0.10	16.9	16.1	16.1	0.29
50	1021	1023	966	53.9	539	487	533	13.9	53.2	52.1	50.8	0.92	3132	299	308	5.56
100	9361	8760	9366	357	5954	5240	5932	82.6	423	403	419	6.91	3012	2817	2996	51.1
200	11783	11233	11604	1067	15115	14559	15554	858	3402	3215	3345	62.4	27347	25607	26923	460
500	221624	205002	212941	34845	170474	157749	163785	8498	53714	50423	53061	933	482389	451801	253017	7987
1k	3870577	3577924	3903932	127.3k	1591947	1557064	1560411	66.2k	430824	402409	424660	7268	–	–	–	–

Table 35: Running time (in second) and data volume (in MB) for Privacy-preserving APSD protocols on LB-HL.

Graph	Privacy-Preserving Johnson V1				Privacy-Preserving Johnson V2				Privacy-Preserving FW				Privacy-Preserving TC			
	S1	S2	S3		S1	S2	S3		S1	S2	S3		S1	S2	S3	
n	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time
5	3.1	3.1	3.1	135	2.2	2.2	2.2	106	0.08	0.08	0.08	2.22	0.46	0.45	0.46	4.01
10	13.8	13.5	13.7	633	7.5	7.3	7.5	281	0.48	0.45	0.47	4.46	1.64	1.55	1.61	7.53
20	77.1	74.7	76.7	3008	35.7	33.7	35.7	691	3.52	3.32	3.47	9.35	16.4	15.5	16.1	14.2
50	271	258	266	9446	243	230	239	1586	54.1	51.1	53.1	28.6	321	307	326	66.3
100	3210	3092	3143	45.9k	1854	1754	1821	3773	402	405	421	90.5	2976	2992	3124	426
200	12532	11902	12323	221k	15846	14924	1551	9748	3417	3231	3631	526	27187	27636	28691	2529
500	134419	128368	132241	1664k	217304	213833	221669	59.1k	53343	55171	57212	7469	–	–	–	–
1k	1196312	1152269	1184117	7555k	1929787	1889923	1971995	335k	426616	434245	450533	57.4k	–	–	–	–

Table 36: Running time (in second) and data volume (in MB) for Privacy-preserving APSD protocols on HB-HL.

Graph	Privacy-Preserving Johnson V1				Privacy-Preserving Johnson V2				Privacy-Preserving FW				Privacy-Preserving TC			
	S1	S2	S3		S1	S2	S3		S1	S2	S3		S1	S2	S3	
n	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Time
5	2.5	2.5	2.5	146	2.2	2.1	2.2	99.3	0.08	0.07	0.08	2.22	0.17	0.16	0.17	4.01
10	12.9	12.6	12.8	589	6.3	6.1	6.3	224	0.48	0.45	0.48	4.44	1.6	1.5	1.6	7.37
20	28.4	27.3	28.6	1471	20.4	19.3	20.6	482	3.5	3.3	3.5	9.1	16.4	15.5	16.1	12.9
50	274	262	270	9446	240	230	239	1569	53.8	50.8	52.8	23.6	318	296	312	26.9
100	1830	1742	1791	48.4k	1990	1874	1943	3611	427	403	419	52.9	3055	3019	3235	157
200	12500	11872	12301	220k	15858	14961	15575	8556	3411	3225	335	153	27960	28650	30526	1336
500	180493	172339	178318	1566k	260020	253553	269801	40.4k	54706	52774	56274	2753	493758	514058	547932	23.5k
1k	1247844	1172302	1221617	7484k	1962492	1886016	2012916	189k	443198	456140	489730	21.5k	–	–	–	–

Appendix C. ALGEBRAIC PATH COMPUTATION VS. BELLMAN-FORD VERSION-3

C.1. Running time (in second) and data volume (in MB) for Privacy-preserving Bellman-Ford and Algebraic path Protocols via Sharemind Cluster

Table 37: Bellman-Ford and Algebraic path protocols on HB-LL environment.

Graph		Bellman-Ford Version-3				Algebraic path computation				Improvement
		S1	S2	S3	Time	S1	S2	S3	Time	
$G(A)$	$A \times A$	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	x-time
5	25	0.6 MB	0.5 MB	0.3 MB	0.2	0.33 MB	0.2 MB	0.1 MB	0.1	3.3x
9	81	1.8 MB	1.7 MB	1.7 MB	2.74	0.3 MB	0.3 MB	0.3 MB	0.3	9.1x
17	289	31 MB	30 MB	21 MB	18.4	2.4 MB	2.2 MB	2.3 MB	1.2	15.3x
33	1089	212 MB	212 MB	219 MB	214	22.9 MB	29.1 MB	29.9 MB	8.2	26.1x
65	4225	5762 MB	5665 MB	5755 MB	819	276 MB	376 MB	445 MB	66.4	12.3x
129	16641	126 GB	144 GB	131 GB	13395	1972 MB	1857 MB	1955 MB	522	25.6x
257	44049	4005 GB	4064 GB	3954 GB	203428	16.7 GB	15.7 GB	16.7 GB	4280	47.5x
513	263169	27.9 TB	25.6 TB	29.5 TB	3092314	141.0 GB	133.0 GB	138.3 GB	35341	87.4x
1025	1050625	691.5 TB	750.0 TB	593.3 TB	46914854	–	–	–	–	–

Table 38: Bellman-Ford and Algebraic path protocols on LB-HL environment.

Graph		Bellman-Ford Version-3				Algebraic path computation				Improvement
		S1	S2	S3	Time	S1	S2	S3	Time	
$G(A)$	$A \times A$	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	x-time
5	25	0.4 MB	0.4 MB	0.4 MB	33.3	0.1 MB	0.1MB	0.1 MB	18.2	1.8x
9	81	3.5 MB	3.4 MB	3.5 MB	108	0.3 MB	0.3 MB	0.3 MB	38.0	2.8x
17	289	36 MB	35 MB	36 MB	399	2.4 MB	2.3 MB	2.4 MB	71.4	5.5x
33	1089	306 MB	291 MB	304 MB	1684	23.1 MB	22.0 MB	22.8 MB	165	10.2x
65	4225	2646 MB	2490 MB	2692 MB	9205	217 MB	214 MB	222 MB	670	13.7x
129	16641	269 GB	284 GB	270 GB	81346	1960 MB	1968 MB	2046 MB	2669	30.4x
257	44049	3854 GB	3728 GB	3949 GB	1154261	16.5 GB	16.7 GB	17.3 GB	20276	56.9x
513	263169	50.3 TB	49.7 TB	52.1 TB	17883699	139.2 GB	141.7 GB	147.1 GB	166643	107.3x
1025	1050625	500.0 TB	504.5 TB	524.8 TB	273458923	–	–	–	–	–

Table 39: Bellman-Ford and Algebraic path protocols on HB-HL environment.

Graph		Bellman-Ford Version-3				Algebraic path computation				Improvement
		S1	S2	S3	Time	S1	S2	S3	Time	
$G(A)$	$A \times A$	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	$\mathcal{DV}_1$	$\mathcal{DV}_2$	$\mathcal{DV}_3$	Sec.	x-time
5	25	0.37 MB	0.36 MB	0.37 MB	33.3	0.1 MB	0.1 MB	0.1 MB	18.2	1.8x
9	81	2.8 MB	2.61 MB	2.77 MB	108	0.3 MB	0.3 MB	0.3 MB	38.0	2.8x
17	289	5.71 MB	5.24 MB	5.44MB	388	2.4 MB	2.3 MB	2.4 MB	69.4	5.6x
33	1089	428 MB	410 MB	425 MB	1509	23.0 MB	21.8 MB	22.7 MB	146	10.3x
65	4225	7731 MB	7397 MB	7703 MB	6542	223 MB	234 MB	249 MB	522	12.5x
129	16641	117 GB	117 GB	110 GB	36835	2026 MB	2057 MB	2204 MB	1355	27.1 x
257	44049	2843 GB	2557 GB	2767 GB	521491	17.0 GB	17.6 GB	18.8 GB	9182	53.1x
513	263169	32.5 TB	32.5 TB	34.9 TB	7147049	143.9 GB	149.6 GB	159.9 GB	73215	97.6x
1025	1050625	238.1 TB	285.9 TB	303.7 TB	116623074	–	–	–	–	–

C.2. Privacy-preserving MST prim via Sharemind Cluster

Table 40: High-Bandwidth (in MB) Low-Latency (in second)

Graph		Privacy-Preserving MST Prim					
		Server-1		Server-2		Server-3	
n	m	Time	$\mathcal{DV}_1$	Time	$\mathcal{DV}_2$	Time	$\mathcal{DV}_3$
50	300	0.81	10.0	0.81	9.7	0.81	9.9
50	1225	0.84	12.1	0.84	11.9	0.84	10.1
100	1K	2.21	35.9	2.27	35.0	2.21	34.2
200	5K	7.10	129	7.10	123	7.10	124
200	19.9K	7.10	128	7.10	124	7.10	127
1k	2K	129	3554	129	3075	129	3183
1k	3K	130	2820	130	2733	130	2799
1k	40K	129	2973	129	2809	129	2880
1k	499K	131	2887	131	2798	131	2863
3k	6K	1065	37064	1065	31230	1065	29723
3k	9K	1069	24983	1069	24000	1069	24770
5k	1M	2944	71666	2944	67993	2945	69664
5k	12.4M	2982	72636	2981	69055	2983	71023
10k	49.9M	11745	346330	11745	304688	11750	312186

ACKNOWLEDGEMENTS

I would like to express my appreciation and gratitude to my supervisors, Eero Vainikko and Peeter Laud, during my Ph.D studies. Eero Vainikko always encouraged me and provided me with the knowledge, advice, motivation, and all the necessary resources for success. I have been extremely lucky to work as his PhD student under his supervision. Peeter Laud was there for me throughout the process. He taught me many things and how to overcome challenges, conduct research, and solve problems; this thesis would never have been possible without his outstanding efforts and supervision.

I am very thankful to the internal reviewer at the University of Tartu Pelle Jakovits, and the opponents Shin'ichiro Matsuo and Mustafa A. Mustafa, whose valuable suggestions and comments have contributed remarkably to improving the quality of this thesis.

I am extremely thankful to the European Regional Development Fund for supporting me during my study through the Estonian Centre of Excellence in ICT research (EXCITE). I also thank the Estonian Research Council (ETAG) for supporting this work through grant number PRG920.

I would like to thank all staff members in the Chair of Distributed Systems for all help and support that encouraged me during my study.

I would also like to thank all members of the staff at the University of Tartu who helped and supported me during all this time.

I would like to thank Cybernetica AS and its staff members for all facilities and support while working there, especially the research department staff.

I would like to thank my colleague Benson Muite from the Institute of Computer Science; I will not forget his seeking to guide, teach, and help me in everything I asked for, during this work, especially in the first year.

Finally, I express my most profound appreciation and gratitude to my family for their support and encouragement and for standing with me all the time and in all circumstances.

SISUKOKKUVÕTE

Privaatsust säilitavad paralleelarvutused graafiülesannete jaoks

Turvalise ühisarvutuse protokollide abil reaaleluliste rakenduste koostamine on keeruline nende protokollide raundikeerukuse tõttu. See on üks kõige olulisemaid väljakutseid üldiste funktsionaalsuste jaoks turvalise ühisarvutuse protokollide konstrueerimisel. Eriti oluline on see probleem ühissalastusel põhinevate protokollistike juures, kus arvutusosapooled peavad üksteisega iga operatsiooni juures suhtlema. Ühissalastusel põhinevad protokollid võivad olla kõikvõimalikest turvalise ühisarvutuse protokollidest kõige efektiivsemad, kuid see efektiivsus materjaliseerub ainult suure paralleelsusega ülesannete juures. Raundikeerukuse vähendamisega saavutatav võrgulatentsi vähenemine on teaduslik väljakutse, millele me oma töös keskendume, arendades ja optimeerides uudseid turvalise ühisarvutuse protokolle. Me püstitasime hüpoteesi, et paralleelarvutustest pärit tehnikate kasutamine aitab turvalise ühisarvutuse protokollide raundikeerukust vähendada, selle hüpoteesi uurimine on käesoleva doktoritöö sisu. Privaatsust säilitavate tehnoloogiate uudsuse ja nendega seotud arvutuste suure keerukuse tõttu pole paralleelseid privaatsust säilitavaid graafialgoritme veel olulisel määral uuritud. Graafialgoritmid on paljude arvutiteaduse rakenduste aluseks, nagu näiteks navigatsioonisüsteemid, gruppide tuvastamine, tarneahelate optimeerimine, hüperspektraalsete kujutiste töötlus, hõredate lineaarvõrrandisüsteemide lahendamine. Suurte privaatsete andmekogumite töötlemise hõlbustamiseks vajame me privaatsust säilitavaid graafialgoritme. Käesolevas doktoritöös me esitame privaatsust säilitava paralleelarvutuse tehnikaid mitmesuguste graafialgoritmide jaoks, mis võivad olla lühimaid teid ja minimaalse kaaluga aluspuid vajavate päriselurakenduste aluseks.

Ma pakume välja paralleelsed turvalise ühisarvutuse protokollid graafi tippude kauguste leidmiseks ühest konkreetsest tipust (SSSD), graafi kõigitipupaaride omavaheliste kauguste leidmiseks (APSD), minimaalse kaaluge aluspuude (MST) ja -metsade (MSF) leidmiseks ning hõredate graafide töötlemiseks poolringiraamistikus. Turvalise ühisarvutuse raundikeerukuse vähendamiseks kasutame ühe käsuvoo ja mitme andmevooga (SIMD) lähenemist, mis vähendab turvalise ühisarvutuse arvutusplatvormi osapoolte vahelist võrgulatentsi. Me esitame parimad teadaolevad privaatsust säilitavad protokollid Bellman-Fordi, Dijkstra, laiuti otsimise, raadiussammumise SSSD algoritmide jaoks ning Johnsoni APSD algoritmi jaoks, võrreldes viimase jõudlust Floyd-Warshalli ja transitiivse sulundi leidmise algoritmidega. Samuti esitame me parimad teadaolevad protokollid minimaalse kaaluga alusmetsa jaoks kasutades Primi algoritmi, seda nii tihedate kui hõredate graafide jaoks. Doktoritöö kolmas osa esitab privaatsust säilitava algoritmi algebraliseks teeleidmiseks koos seotud funktsioonidega. Need funktsioonid on kasutatavad hõredate lineaarsüsteemidega seotud erinevate ülesannete lahendamiseks poolringiraamistikus. Käesolevas doktoritöös me hindame väljapakutud algoritmide jõudlust ja järeldame, millised neist on eri tüüpi graafide jaoks efektiivsemad, võttes arvesse servade arvu suhet tippude arvu, servade asukoha privaatsust, kaalude olemasolu servadel, jne.

Detailsemalt: me pakume välja kaks eri varianti privaatsust säilitavast Bellman-Fordi algoritmist hõredate graafide jaoks, nende variantide põhiline erinevus on paralleelse prefiksmiinimumi leidmise alamprogrammi detailides. Privaatsust säilitavast Dijkstra algoritmist esitame samuti kaks eri varianti, kus teine variant töötab mitme graafi jaoks samaaegselt. Ka privaatsust säilitavast laiutiläbimise algoritmist esitame me kaks versiooni, kus üks on kaalutud ja teine kaaludeta servadega graafide jaoks. Privaatsust säilitava John-

soni algoritmi komponeerime Bellman-Fordi ja paralleelsest Dijkstra algoritmist. Sellestki algoritmist on meil kaks versiooni, vastavalt Dijkstra algoritmi versioonidele. Kahe komponendi vahelise, servade ümberkaalumisel põhineva ühenduse loome Laua mälupöördusprotokolli abil. Minimaalse alusmetsa jaokski esitame kaks algoritmi: järjestiku ja paralleelse.

SIMD-põhimõtet kasutavad paralleelalgoritmid ei ole ainult klassikaliste ja algebraliste graafiülesannete jaoks. Küll aga on nad heaks näiteks, kuidas seda tehnikat kasutada erinevate keerukate probleemide lahendamiseks. Kuna turvalise ühisarvutuste protokollistike defineeritud arvutusplatvormide jõudlusprofiil erineb olulisel määral tavaliste protsessorite omast, siis tuli meil vastuvõetava jõudluse saavutamiseks lahendusalgoritme olulisel määral muuta, kaasa arvatud nende kontrollivoogu ja mälupöördusmustreid. Me realiseerisime oma protokollid ühissalastusel põhineva Sharemind MPC platvormi peal, kasutades selle platvormiga kaasasolevat SecreC-programmeerimiskeelt ja arendusvahendeid. Me viisime läbi oma protokollide ja nende oluliste alamosade põhjalikud jõudlustestid eri suurusega graafidel ja eri võrgukeskkondades. Kui eeldada, et turvalise ühisarvutuse protokollistikke realiseerivad osapooled on eri geograafilistes punktides, siis saab meie jõudlustestide abil teha mõistlikke oletusi nii meie protokollide kui ka suuremate privaatsust säilitavate rakenduste jõudluse kohta sellise juurutuse korral. Mõnede meie väljapakutud paralleelprotokollide kiirus on eelmiste tööde ja neis esitatud standardsete realisatsioonidega võrreldes tuhandekordne. Lisaks sellele ei ole minimaalse kaaluga alusmetsa ja algebralise teeleidmise protokolle kunagi varem välja pakutud.

CURRICULUM VITAE

Personal data

Name:	Mohammad Anagreh
Birth:	6. August 1984
	Irbid, Jordan
Citizenship:	Jordanian
Languages:	Arabic, English
E-mail:	mohammad.anagreh@ut.ee

Education

2017–	University of Tartu, Ph.D. candidate in Computer Science
2010–2012	Universiti Sains Malaysia, M.Sc. in Computer Science
2003–2007	Irbid National University, B.Sc. in Computer Science

Employment

2018–	Cybernetica As, Junior Researcher
2013–2017	Prince Sattam Bin Abdualaziz University, Lecturer of Computer Skills and information technology

Scientific work

Main fields of interest:

- Privacy-preserving parallel computation
- Parallel computing
- Cryptography and security

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Mohammad Anagreh
Sünniaeg ja -koht:	6. augustil 1984 Irbid, Jordaania
Kodakondsus:	Jordaanlane
Keelteoskus:	Araabia, Inglise,
Kontaktandmed:	mohammad.anagreh@ut.ee

Haridus

2017–	Tartu ülikool, informaatika doktorant
2010–2012	Malaisia ülikooliteadus, Arvutiteadus
2003–2007	Irbidi riiklik ülikool, Arvutiteadus

Teenistuskäik

2018–	Cybernetica As, nooremteadur
2013–2017	Prints Sattam Bin Abdualazizi ülikool, arvutioskuste ja infotehnoloogia lektor

Teadustegevus

Peamised uurimisvaldkonnad:

- Privaatsust säilitav paralleelarvutus
- Paralleelne andmetöötlus
- Krüptograafia ja turvalisus

LIST OF ORIGINAL PUBLICATIONS

Published papers included in the thesis

- Anagreh, M., Laud, P.: A Parallel Privacy-Preserving Shortest Path Protocol from a Path Algebra Problem. The 17th International Workshop on Data Privacy Management (DPM 2022), DPM 2022/CBT 2022, LNCS 13619, pp. 1–16 (2023).
- Anagreh, M., Laud, P. and Vainikko, E.: Privacy-Preserving Parallel Computation of Minimum Spanning Forest. SN Computer Science, 3(6), pp.1-19 (2022).
- Anagreh, M., Laud, P. and Vainikko, E.: Privacy-Preserving Parallel Computation of Shortest Path Algorithms with Low Round Complexity. In Proceedings of the 8th International Conference on Information Systems Security and Privacy - ICISSP, ISBN 978-989-758-553-1, pp.37-47 (2022),
  DOI: 10.5220/0010775700003120
- Anagreh, M., Laud, P. and Vainikko, E.: Parallel Privacy-Preserving Shortest Path Algorithms. Cryptography, 5(4), p.27 (2021).
- Anagreh, M.; Vainikko, E. and Laud, P.: Parallel Privacy-preserving Computation of Minimum Spanning Trees. In Proceedings of the 7th International Conference on Information Systems Security and Privacy - ICISSP, ISBN 978-989-758-491-6; ISSN 2184-4356, pp. 181-190 (2021),
  DOI: 10.5220/0010255701810190
- Anagreh, M., Laud, P. and Vainikko, E.: Parallel Privacy-Preserving Shortest Paths by Radius-Stepping. In: 2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP). IEEE. 2021, pp. 276–280 (2021).

Publications not included in the thesis

- Anagreh, M.; Vainikko, E. and Laud, P. (2019). Accelerate Performance for Elliptic Curve Scalar Multiplication based on NAF by Parallel Computing. In Proceedings of the 5th International Conference on Information Systems Security and Privacy - ICISSP, ISBN 978-989-758-359-9; ISSN 2184-4356, pages 238-245.
  DOI: 10.5220/0007312702380245

• Anagreh, M.; Vainikko, E. and Laud, P. (2020). Speeding Up the Computation of Elliptic Curve Scalar Multiplication based on CRT and DRM. In Proceedings of the 6th International Conference on Information Systems Security and Privacy - ICISSP, ISBN 978-989-758-399-5; ISSN 2184-4356, pages 176-184.
DOI: 10.5220/0009129501760184

DISSERTATIONES INFORMATICAE PREVIOUSLY PUBLISHED IN DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 lk.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.

113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.
114. **Alisa Pankova.** Efficient Multiparty Computation Secure against Covert and Active Adversaries. Tartu, 2017, 316 p.
116. **Toomas Saarsen.** On the Structure and Use of Process Models and Their Interplay. Tartu, 2017, 123 p.
121. **Kristjan Korjus.** Analyzing EEG Data and Improving Data Partitioning for Machine Learning Algorithms. Tartu, 2017, 106 p.
122. **Eno Tõnisson.** Differences between Expected Answers and the Answers Offered by Computer Algebra Systems to School Mathematics Equations. Tartu, 2017, 195 p.

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

1. **Abdullah Makkeh**. Applications of Optimization in Some Complex Systems. Tartu 2018, 179 p.
2. **Riivo Kikas**. Analysis of Issue and Dependency Management in Open-Source Software Projects. Tartu 2018, 115 p.
3. **Ehsan Ebrahimi**. Post-Quantum Security in the Presence of Superposition Queries. Tartu 2018, 200 p.
4. **Ilya Verenich**. Explainable Predictive Monitoring of Temporal Measures of Business Processes. Tartu 2019, 151 p.
5. **Yauhen Yakimenka**. Failure Structures of Message-Passing Algorithms in Erasure Decoding and Compressed Sensing. Tartu 2019, 134 p.
6. **Irene Teinemaa**. Predictive and Prescriptive Monitoring of Business Process Outcomes. Tartu 2019, 196 p.
7. **Mohan Liyanage.** A Framework for Mobile Web of Things. Tartu 2019, 131 p.
8. **Toomas Krips.** Improving performance of secure real-number operations. Tartu 2019, 146 p.
9. **Vijayachitra Modhukur.** Profiling of DNA methylation patterns as biomarkers of human disease. Tartu 2019, 134 p.
10. **Elena Sügis.** Integration Methods for Heterogeneous Biological Data. Tartu 2019, 250 p.
11. **Tõnis Tasa.** Bioinformatics Approaches in Personalised Pharmacotherapy. Tartu 2019, 150 p.
12. **Sulev Reisberg.** Developing Computational Solutions for Personalized Medicine. Tartu 2019, 126 p.
13. **Huishi Yin.** Using a Kano-like Model to Facilitate Open Innovation in Requirements Engineering. Tartu 2019, 129 p.
14. **Faiz Ali Shah.** Extracting Information from App Reviews to Facilitate Software Development Activities. Tartu 2020, 149 p.
15. **Adriano Augusto**. Accurate and Efficient Discovery of Process Models from Event Logs. Tartu 2020, 194 p.
16. **Karim Baghery.** Reducing Trust and Improving Security in zk-SNARKs and Commitments. Tartu 2020, 245 p.
17. **Behzad Abdolmaleki.** On Succinct Non-Interactive Zero-Knowledge Protocols Under Weaker Trust Assumptions. Tartu 2020, 209 p.
18. **Janno Siim.** Non-Interactive Shuffle Arguments. Tartu 2020, 154 p.
19. **Ilya Kuzovkin.** Understanding Information Processing in Human Brain by Interpreting Machine Learning Models. Tartu 2020, 149 p.
20. **Orlenys López Pintado.** Collaborative Business Process Execution on the Blockchain: The Caterpillar System. Tartu 2020, 170 p.
21. **Ardi Tampuu.** Neural Networks for Analyzing Biological Data. Tartu 2020, 152 p.

22. **Madis Vasser.** Testing a Computational Theory of Brain Functioning with Virtual Reality. Tartu 2020, 106 p.
23. **Ljubov Jaanuska.** Haar Wavelet Method for Vibration Analysis of Beams and Parameter Quantification. Tartu 2021, 192 p.
24. **Arnis Parsovs.** Estonian Electronic Identity Card and its Security Challenges. Tartu 2021, 214 p.
25. **Kaido Lepik.** Inferring causality between transcriptome and complex traits. Tartu 2021, 224 p.
26. **Tauno Palts.** A Model for Assessing Computational Thinking Skills. Tartu 2021, 134 p.
27. **Liis Kolberg.** Developing and applying bioinformatics tools for gene expression data interpretation. Tartu 2021, 195 p.
28. **Dmytro Fishman.** Developing a data analysis pipeline for automated protein profiling in immunology. Tartu 2021, 155 p.
29. **Ivo Kubjas.** Algebraic Approaches to Problems Arising in Decentralized Systems. Tartu 2021, 120 p.
30. **Hina Anwar.** Towards Greener Software Engineering Using Software Analytics. Tartu 2021, 186 p.
31. **Veronika Plotnikova.** FIN-DM: A Data Mining Process for the Financial Services. Tartu 2021, 197 p.
32. **Manuel Camargo.** Automated Discovery of Business Process Simulation Models From Event Logs: A Hybrid Process Mining and Deep Learning Approach. Tartu 2021, 130 p.
33. **Volodymyr Leno.** Robotic Process Mining: Accelerating the Adoption of Robotic Process Automation. Tartu 2021, 119 p.
34. **Kristjan Krips.** Privacy and Coercion-Resistance in Voting. Tartu 2022, 173 p.
35. **Elizaveta Yankovskaya.** Quality Estimation through Attention. Tartu 2022, 115 p.
36. **Mubashar Iqbal.** Reference Framework for Managing Security Risks Using Blockchain. Tartu 2022, 203 p.
37. **Jakob Mass.** Process Management for Internet of Mobile Things. Tartu 2022, 151 p.
38. **Gamal Elkoumy.** Privacy-Enhancing Technologies for Business Process Mining. Tartu 2022, 135 p.
39. **Lidia Feklistova.** Learners of an Introductory Programming MOOC: Background Variables, Engagement Patterns and Performance. Tartu 2022, 151 p.
40. **Mohamed Ragab.** Bench-Ranking: A Prescriptive Analysis Approach for Large Knowledge Graphs Query Workloads. Tartu 2022, 158 p.