Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing

Marina Blanton, Aaron Steele, and Mehrdad Aliasgari
Department of Computer Science and Engineering
University of Notre Dame
{mblanton,asteele2,maliasga}@nd.edu

ABSTRACT

This work treats the problem of designing data-oblivious algorithms for classical and widely used graph problems. A data-oblivious algorithm is defined as having the same sequence of operations regardless of the input data and data-independent memory accesses. Such algorithms are suitable for secure processing in outsourced and similar environments, which serves as the main motivation for this work. We provide data-oblivious algorithms for breadth-first search, single-source single-destination shortest path, minimum spanning tree, and maximum flow, the asymptotic complexities of which are optimal, or close to optimal, for dense graphs.

Categories and Subject Descriptors

G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*; K.6 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

Graph algorithms; oblivious execution; secure computation

1. INTRODUCTION

Cloud computing has become prevalent today and allows for convenient on-demand access to computing resources which enable clients to meet their unique needs. Such services are used for as diverse a range of applications as management of personal photos, carrying out computationally intensive scientific tasks, or outsourcing building and maintenance of an organization's computing infrastructure. Placing large volumes of one's data and computation outside one's immediate control, however, raises serious security and privacy concerns, especially when the data contains personal, proprietary, or otherwise sensitive information. To protect such information while being able to utilize external computing resources, secure processing of data used for specialized tasks has become an active area of research. Examples of such results include secure and verifiable storage outsourcing (e.g., [19]) and secure outsourcing of common tasks such as linear algebra operations (e.g., [4]) or sequence comparisons (e.g., [6]). There is, however, a lack of efficient techniques for computation on protected data in outsourced environments for most commonly used algorithms and data structures. To help eliminate this void, we develop data-oblivious algorithms for fundamental graph problems. Data-oblivious, or just oblivious, execution is defined as having the same sequence of operations regardless of the input data and data-independent memory accesses, which makes it suitable for use in outsourced tasks. Note that the great majority of data structures and algorithms commonly used in practice are not data-oblivious and thus reveal information about data, while naive approaches for achieving data-obliviousness incur a substantial increase in computation time over best-known solutions (consider, e.g., search, binary trees, etc.). Therefore, a careful design of data structures and algorithms is essential for bringing the complexity of oblivious execution as close to the complexity of its non-oblivious counterpart as possible.

While secure computation on sensitive data in outsourced and similar environments serves as the main motivation for our data-oblivious techniques, we would like to abstract the presentation of the developed techniques from a concrete setting or underlying mechanisms for securing the data. Nevertheless, to guarantee privacy of the data used in the computation, throughout this work we assume that the computation proceeds on protected data (e.g., using suitable secure multi-party computation techniques) and the only values that can be observed are memory accesses and results of the computation that lead to accessing specific memory locations. Data privacy is then guaranteed if the memory accesses are data-independent or oblivious.

In this work we focus on classical graph problems such as breadth-first search (BFS), shortest path, minimum spanning tree, and maximum flow and construct data-oblivious algorithms of practical performance. Several of them are fundamental graph problems with many uses both by themselves (e.g., for traversing a social network graph) and as building blocks for more complex problems on graphs. It therefore would be desirable to obtain data-oblivious solutions suitable for use in secure computation and outsourcing for the above problems.

1.1 Related work

One topic closely related to secure and oblivious computation is Oblivious RAM (ORAM) [23, 33, 24]. ORAM was introduced as the means of secure program execution in an untrusted environment. Only the CPU with a small amount

ASIA CCS'13, May 8–10, 2013, Hangzhou, China.

of internal memory ($O(1)$ or $O(n^a)$ for $0 < a < 1$, where n is the size of used external memory) is assumed to be trusted, while the program itself and the data are stored encrypted on untrusted storage. The goal is to ensure that no information about the program is revealed by observing its memory accesses and therefore the accesses must appear random and independent of the actual sequence of accesses made by the program. There is a renewed interest in ORAM due to the emergence of cloud computing and storage services, and recent results include [39, 40, 34, 1, 17, 27, 7, 36, 37, 29]. ORAM techniques then can be used to make a non-oblivious algorithm oblivious at the cost of polylogarithmic (in the size of the data) amortized cost per access. Current solutions achieve $O((\log n)^2)$ overhead per access (if random functions are used or server's storage is superlinear in n; otherwise, the overhead is $O((\log n)^3)$ per access). A comparison of several ORAM schemes and their complexities can be found in [1].

Another related topic is private information retrieval (PIR) (see, e.g., [14, 15, 30, 11, 22, 31, 38] among others), where a server holds a database and a clients wants to a retrieve a record at a specific position with the goal that the server should not learn what record was accessed. Symmetric PIR (SPIR) solutions also require that the user should not learn anything about any other records in the database except the record of interest. Current PIR solutions exist in both the information-theoretic and computational settings.

Privacy-preserving graph algorithms have become a recent area of interest with the prevalent adoption of location based services (LBS) for cloud services and mobile devices. With the widespread use of LBS there is an increased need for maintaining users' privacy. The approach in [32] utilizes PIR techniques to query a data-set while keeping the user's position, path, and desired destination private. Also motivated by LBS is the work proposed in [18]. It introduced data-oblivious algorithms for calculating classical geometric problems, such as convex hull and all nearest neighbors, using secure multi-party computation.

Other solutions that focus on privacy-preserving graph algorithms include algorithms designed for a public or joint graph. In particular, [8] presents a two-party solution for computing all pairs shortest distance and single source shortest path when each party holds a portion of the graph. The solution uses Yao's protocol [41] as the primary building block and achieves better efficiency than naively applying Yao's protocol to existing graph algorithms. Similarly, [20] presents a two-party method for privately performing an $A^\star$ search over a public graph where different parties hold the edge weights and the heuristics. Additionally, there are several techniques for constructing a graph privately. One such solution is [21], which shows how many clients, each possessing a portion of the distributed graph, can privately construct a graph so that an adversary is unable to deanonymize any part of it after the construction process.

There are also recent data-oblivious graph algorithms designed for the external-memory model. This model assumes that the client has access to some amount of working-memory that is inaccessible to any adversary, but the remaining storage is outsourced. The solution in [26] introduces data-oblivious algorithms for compaction, selection, and sorting. A final technique of interest is [28], which introduces data-oblivious algorithms for graph drawing problems (e.g., Euler Tours, Treemap Drawings) and develops efficient compressed-scanning techniques, which in turn allow for private data to be searched in an efficient manner.

Similar to the external-memory model is the cache-oblivious model, where algorithms are designed to perform optimally using a CPU-cache without the size of the cache as an explicit-parameter. Several cache-oblivious algorithms and data structures are described in [2, 10, 9, 3]. Of particular interest is [10], where the authors describe cache-oblivious solutions for breadth-first search and shortest path.

1.2 Our contributions

In this work, we present data-oblivious algorithms for several fundamental graph algorithms, namely, breadth-first search, single-source single-destination (SSSD) shortest path, minimum spanning tree, and maximum flow. Given graph $G = (V, E)$ as the input, our solutions assume adjacency matrix representation of the graph, which has size $\Theta(|V|^2)$ and is asymptotically optimal for dense graphs with $|E| = \Theta(|V|^2)$. Our oblivious solutions achieve the following complexities: $O(|V|^2)$ for BFS, $O(|V|^2)$ for SSSD shortest path, $O(|V|^2)$ for the minimum spanning tree, and $O(|V|^3|E|\log(|V|))$ for the maximum flow. This performance is optimal, or close to optimal, for dense graphs and outperforms applying ORAM techniques to the fastest conventional non-oblivious algorithms for these problems.

Because our algorithms use adjacency matrix representation, they are not well suited for sparse graphs. For such graphs, combining ORAM with the best non-oblivious algorithm is likely to result in superior performance, even though ORAM techniques are often not particularly fast in practice. We leave the design of efficient oblivious algorithms for sparse graphs as a direction for future work.

Our solutions assume that numeric values can be compared, multiplied, and added in a protected form; the cost of such operations is considered to be constant (see section 1.3 for a specific instantiation of these operations). We also rely on random permutation of a vector as our building block. This functionality can be accomplished by assigning random values to each element of the vector and sorting them according to the assigned values. Oblivious sorting can be accomplished, e.g., using the techniques in [25] at cost $O(n \log n)$ for a set of size n. We therefore assume that a vector can be randomly permuted at the same asymptotic cost.

All of the graph algorithms considered in this work proceed in iterations. At the core of achieving the above complexities obliviously is the idea that we do not need to touch all locations of the adjacency matrix at each iteration to hide information about the access pattern, but are able to access a single row of the matrix per iteration. Because we access each row of the matrix exactly once and the accesses are performed in random order, we show that the accesses are data oblivious. We first develop our solution for the BFS problem and then extend it with path reconstruction to obtain a solution to the SSSD shortest path problem. Both BFS and SSSD shortest path solutions are consecutively used to build our data-oblivious algorithm for the maximum flow problem. Lastly, we also present an oblivious algorithm for the minimum spanning tree problem that utilizes similar techniques.

1.3 Applications to secure computation and outsourcing

As mentioned earlier, secure outsourcing serves as the main motivation for our data-oblivious graph algorithms.

We therefore sketch how our algorithms can be used in outsourced environments for securely computing on protected data. We utilize the setting in which the computation is carried out by multiple computational nodes, which allows us to formulate the problem as secure multi-party computation.

To allow for as general problem formulation as possible, we place all participants into the following three categories: (i) the party or parties who hold private inputs; (ii) the party or parties who learn the outcome of the computation, and (iii) the parties who conduct the computation. There are no constraints on how these three groups are formed, and a single entity can be involved in a solution taking on one or more of the above roles. This framework formulation naturally fits several broad categories of collaborative and individual computing needs. For example, a number of parties with private inputs can engage in secure function evaluation among themselves and learn the result (or their respective results). They can also choose a subset among themselves, a number of outside parties, or a combination of the above to carry out the computation. Note that this includes the important use case of a single entity outsourcing its computation to computational servers, in which case the data owner is the only input and output party.

The algorithms that we present can be realized in this setting using a number of underlying techniques such as linear secret sharing, threshold homomorphic encryption, or Boolean garbled circuits. Then if we, for instance, utilize an information-theoretically secure linear secret sharing scheme (such as [35]), any linear combination of secret-shared values is computed locally and multiplication is the very basic interactive building block. Comparisons such as less-than and equality tests can be performed efficiently using, for instance, techniques of [13]. In this setting, the arithmetic is efficient and complexity is measured in the number of interactive operations (while in settings that rely on public-key cryptography, the complexity will additionally need to be measured in the number of modular exponentiations). This gives us a secure implementation of our data-oblivious graph algorithms which are suitable for outsourced environments as the sequence of operations they execute does not reveal any information about the data.

2. PROBLEM DEFINITION

In this work we consider graph problems which take graph $G = (V, E)$ as part of their input. We assume that the graph is specified in the form of the adjacency matrix M. To avoid ambiguities, we explicitly specify the adjacency matrix representation used in our description. The adjacency matrix M is a $|V| \times |V|$ matrix containing Boolean values (if the edges do not have weights), where the row corresponding to node $v \in V$ contains information about the edges leaving v. If $(v, u) \in E$, then the cell at row v and column u is set to 1; otherwise, it is set to 0. For undirected graphs, both $M_{v,u}$ and $M_{u,v}$ are set to 1, where $M_{i,j}$ refers to the cell at row i and column j. $M_{v,v}$ is set to 0. Without loss of generality, we assume that the nodes are numbered 1 through $|V|$. For problems that use weighted graph G, we will specify the differences to the adjacency matrix representation at the time of specifying the respective graph problem.

Because secure outsourced computation is used to motivate this work, we assume that the computation proceeds on properly protected data. This means that all inputs and intermediate results are not known to the party or parties carrying out the computation unless we explicitly open their values for the purposes of accessing data at specific locations. For concreteness of exposition, we use notation $[x]$ to indicate that the value of x is protected from the entities performing the computation.

To maintain data privacy, we must guarantee that the computational parties do not learn anything about the data during the execution of the algorithm. Because each private value is assumed to be adequately protected, the only way for the computational parties to deduce information about the private data is when the sequence of instructions or algorithm's memory access pattern depends on the data. Thus, to guarantee data privacy, we formally formulate data-oblivious execution of a deterministic algorithm as follows:

DEFINITION 1. *Let d denote input to a graph algorithm. Also, let $A(d)$ denote the sequence of memory accesses that the algorithm makes. The algorithm is considered data-oblivious if for two inputs d and d' of equal length, the algorithm executes the same sequence of instructions and access patterns $A(d)$ and $A(d')$ are indistinguishable to each party carrying out the computation.*

Without loss of generality, in the description that follows, we use arithmetic operations to implement Boolean operations. In particular, we write $a \cdot b$ to implement conjunction $a \wedge b$ and we write $(1-a)$ to implement complement $\overline{a}$ for Boolean a. We also use notation $(a \stackrel{?}{=} b)$ and $(a \stackrel{?}{<} b)$ to denote conditions comparing two values, which produce a bit.

3. BREADTH-FIRST SEARCH

Breadth-first search is one of the most basic algorithms for searching a graph, but it is applicable to a large number of problems and is the basis for many important graph algorithms. Given a graph $G = (V, E)$ and a source vertex $s \in V$, BFS systematically explores the edges of G to compute the distance (measured in the number of edges) from s to each reachable vertex in G, where nodes at distance i from s are discovered before nodes at distance $i + 1$.

The conventional algorithm for BFS starts from the source node s and colors the nodes white, gray, and black, where the nodes start white and may later become gray and black. The intuition is that white nodes have not been discovered yet, gray nodes have been discovered, but nodes adjacent to them may have not been discovered yet, and black nodes have been discovered themselves as well as their adjacent nodes. The conventional algorithm works by maintaining a queue with gray nodes which initially contains only the source s. When a gray node is being removed from the queue, it becomes black and all white nodes adjacent to it are colored gray and added to the queue.

Now notice that directly applying this algorithm to obtain a data-oblivious version presents problems: as we must protect the graph information, we cannot reveal how many adjacent nodes a vertex has, which means that when removing a node from the queue $O(|V|)$ nodes must be added to the queue (many of which will be just empty records not corresponding to actual nodes). This results in an exponentially growing queue and is clearly unacceptable.

To protect the structure of the graph, we design our algorithm to work on adjacency matrix representation which allows us to achieve $O(|V|^2)$ runtime. This is optimal for dense graphs with $|E| = \Theta(|V|^2)$ and any time an adjacency

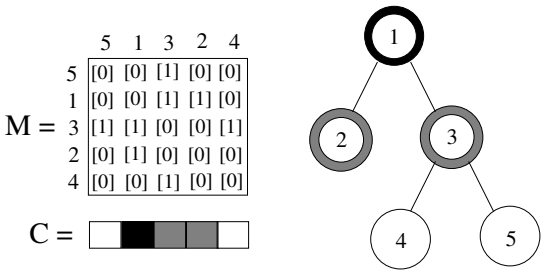


Figure 1: Illustration of the BFS algorithm.

matrix is used. We leave the problem of an efficient data-oblivious algorithm for sparse graphs open.

The intuition behind our algorithms is as follows: instead of maintaining a queue of gray nodes, node coloring is maintained in a (protected) vector C of size $|V|$. Initially, only the source is colored gray. Using the row of the adjacency matrix M corresponding to the source, we obliviously update the node coloring in vector C using the adjacency information of the source s. Once this is accomplished, we obliviously choose one of the gray nodes with the smallest distance from the source from the vector, reveal its location in the adjacency matrix, and use its adjacency information to update the vector C. This process is then repeated until all rows of the matrix have been processed.

We illustrate the idea on an example graph in Figure 1. In the figure, the (protected) matrix M contains node adjacency information where the nodes (i.e., both rows and columns) are randomly permuted. For example, row and column 1 correspond to node 5 in the graph. In the figure, the source node 1 has already been processed and colored black, and the nodes adjacent to it (nodes 2 and 3) has been colored gray. This information is stored in vector C, where the location of (black) node 1 after the permutation is equal to 2 and the locations of (gray) nodes 2 and 3 after the permutation are equal to 4 and 3, respectively.

There are subtleties in implementing the logic described above. In general, revealing what row (and thus the corresponding node) is being processed next can leak information about the structure of the graph. A naive way of hiding this information is to touch each element of matrix M when only one row is being used. We, however, can ensure that no information about the graph is revealed even if only a single row is accessed. This is achieved using two crucial ideas: (i) the order of the nodes of M is random and (ii) the next gray node to be processed is chosen from all candidates at random. The former means that the nodes are either randomly numbered in the adjacency matrix or they are randomly permuted (i.e., both the rows and columns of the matrix are randomly and consistently permuted). The latter means that rather than choosing the first node that meets the condition for the next iteration of the algorithm, it should be chosen from all candidates at random in order to hide the number of candidate nodes (and thus information about graph connectivity).

Also, the graph must be connected in order for the above algorithm to work. That is, to ensure that there is always a way to proceed with another row of the matrix, all nodes must be reachable from the source. For clarity of exposition, we first provide a solution that assumes that the graph is connected and later extend it to work with arbitrary graphs.

3.1 Basic algorithm for connected graphs

On input adjacency matrix $[M]$ of graph $G = (V, E)$ with randomly ordered nodes and source node $s \in V$:

Algorithm 1:

1. Create vector C of size $|V|$, each element of which corresponds to a node of G using the same node ordering as in M. Each element of C contains three fields: color, distance from s, and parent node in the BFS tree.
2. Initialize C to set the color of all nodes except s to white and the color of s to gray. The distance of s is set to 0 and the distance of all other nodes is set to ∞ (which for practical purposes can be set to $|V|$, i.e., larger than any actual distance). The parent of each node is set to a special character $\perp$ indicating that the parent is undefined. The values of the elements of C are protected throughout the execution.
3. Set working node v to s and retrieve row M_v of M.
4. Update C using M_v as follows:
   (a) For $i = 1$ to $|V|$, if $[M_{v,i}]$ is 1 and the color of C_i is white, set the color of C_i to gray, the parent of C_i to v, and the distance of C_i to the distance of C_v incremented by 1.
   (b) Set the color of C_v to black.

   Oblivious execution of conditional statements can be realized by executing both branches of the if-statement (in cases when only one branch is present, the second branch corresponds to keeping the original values). For instance, executing statement
   if $([cond])$ then $[a] = [b]$ else $[a] = [c]$
   becomes rewritten as
   $[a] = [cond] \cdot [b] + (1 - [cond]) \cdot [c]$
   In our case, the executed code is:
   1.for $i = 1$ to $|V|$ do
   2. $[cond] = ([M_{v,i}] \stackrel{?}{=} 1) \cdot ([C_i.color] \stackrel{?}{=} white)$
   3. $[C_i.color] = [cond] \cdot gray + (1 - [cond]) \cdot [C_i.color]$
   4. $[C_i.parent] = [cond] \cdot v + (1 - [cond]) \cdot [C_i.parent]$
   5. $[C_i.dist] = [cond]([C_v.dist] + 1) + (1 - [cond])[C_i.dist]$

5. Obliviously choose one of the gray nodes in C with the smallest distance from the source at random. Data-oblivious execution in this case means that the algorithm must be independent of the k number of nodes in C to choose from, yet random selection requires $1/k$ probability of any qualifying node to be selected.
   To achieve this, we use C to create another vector C' in which the elements that contain gray nodes with the smallest distance from s in C are set to contain their index values and all other elements are set to 0. We also choose a random permutation π of node indices and assign the key $\pi(i)$ to each element i in C' with a non-zero value. All other elements have the key of 0. We select the element with the maximum key and retrieve the index of the node stored with that element as the next node to process. In more detail, we execute:
   1. $[min] = |V|$
   2. for $i = 1$ to $|V|$ do
   3. $[cond_i] = ([C_i.color] \stackrel{?}{=} gray)$
   4. $[cond'_i] = ([C_i.dist] \stackrel{?}{<} [min])$
   5. $[min] = [cond_i][cond'_i][C_i.dist] + (1 - [cond_i] \cdot [cond'_i])[min]$
   6. for $i = 1$ to $|V|$ do
   7. $[cond''_i] = ([C_i.dist] \stackrel{?}{=} [min])$

8. $[C'_i.value] = [cond_i][cond''_i] \cdot i$
9. $[C'_i.key] = [cond_i][cond''_i] \cdot [\pi(i)]$
10. $[max] = 0$
11. $[i_{max}] = 0$
12. for $i = 1$ to $|V|$ do
13. $[cond] = ([C'_i.key] \overset{?}{>} [max])$
14. $[max] = [cond][C'_i.key] + (1 - [cond])[max]$
15. $[i_{max}] = [cond][C'_i.value] + (1 - [cond])[i_{max}]$
16. $i_{max} = open([i_{max}])$
17. $v = open([C'_{i_{max}}.value])$

In the above, $\pi : [1, |V|] \rightarrow [1, |V|]$ is a random permutation of node indices and $open(\cdot)$ corresponds to revealing (or opening) its argument. For efficiency reasons, we can skip all previously processed (i.e., black) nodes as we know that they no longer will be selected.

6. Use chosen node v to retrieve row M_v of M, and repeat steps 4–6 of the algorithm $|V| - 1$ times.

Because our algorithm always selects a gray node with the shortest distance from the source for the next iteration, it correctly implements the BFS queue.

3.2 Supporting general graphs

Algorithm 1 described in the previous section works as long as the graph G is connected, i.e., there is at least a single gray node to choose from at each iteration of the algorithm. In general, however, the input graph is not guaranteed to be connected and we next show how to modify the algorithm to ensure that it is suitable for arbitrary graphs.

The main idea behind the change is that we introduce fake nodes that will be chosen by the algorithm once there are no gray nodes to process in one of the algorithm iterations. In general, the algorithm may terminate after as few as only a single iteration, which means that we need extra $|V|-1$ nodes to simulate the remaining algorithm steps. We therefore modify the adjacency matrix M to include $|V|-1$ extra rows and columns corresponding to fake nodes which are placed at random indices within the matrix. Every fake node is made adjacent to all other fake nodes, but none of them are adjacent to the original nodes of the graph.

The modified matrix M can be formed by appending $|V|-1$ rows and columns to the original matrix and randomly and consistently permuting its rows and columns. The new location of the source node s then needs to be revealed. For the purposes of our algorithm we also store a bit vector F of size $2|V| - 1$ with the nodes ordered in the same way as in the matrix in which the element at location v is set iff node v is fake. This vector is formed by initializing its first $|V|$ elements to 0 and the remaining elements to 1 and permuting it using the same permutation as in the matrix.

With this modified setup information, we are ready to proceed with the BFS algorithm. The basic structure of Algorithm 1 and most of its steps remain unchanged, and the only modification that we introduce is the way a node is chosen for the next algorithm iteration. That is, the algorithm must account for the fact that there might be no gray nodes to choose from at any given iteration and it should proceed with choosing a fake node.

5. Create vector C' as before, but now before choosing one element from it, we check whether it contains at least one element to choose from. If it does, we leave it unmodified; otherwise, we mark all fake nodes as gray using vector F. One of the qualifying nodes of C' is chosen as before as the node to be processed in the next iteration. To test whether there is a least one node to choose from, we can simply test whether the value of $[min]$ has not been updated (i.e., it is still $|V|$). More precisely, we execute the following code between lines of 9 and 10 of the original code of step 5:
   1. $[cond] = ([min] \overset{?}{=} |V|)$
   2. for $i = 1$ to $2|V| - 1$ do
   3. $[C'_i.value] = [C'_i.value] + [cond][F_i] \cdot i$
   4. $[C'_i.key] = [C'_i.key] + [cond][F_i] \cdot [\pi(i)]$

   The remaining code remains unchanged with the exception that all for-loops now range from 1 to $2|V|-1$.

3.3 SSSD shortest path

The above algorithm already computes the distance from the source node to all other nodes in the graph. In certain problems, however, the knowledge of the shortest path itself is required. In this section we therefore show how to obliviously reconstruct the shortest path from the source s to a given destination node t. Both BFS and shortest path computation are used as the building blocks in our solution to the maximum flow problem.

For the ease of exposition, we divide our description in two parts: we first present a solution that reconstructs the path, where the information about the path itself is protected, but the length of the path is revealed. We then show how to modify this solution to also hide the length of the path.

3.3.1 Basic solution

On input $G = (V, E)$ described by its adjacency matrix M, source node $s \in V$, and destination node $t \in V$, our solution first uses BFS to compute the distances from s to all other nodes and then starts from node t and retrieves parent node information from vector C. A simple oblivious solution to the path reconstruction problem can be implemented in $O(|V|^2)$ time. In particular, we can scan the vector C up to $|V|$ times, each time retrieving and storing the parent of the current node on the path. Then because in each round we touch all $|V|$ nodes (or all $2|V| - 1$ nodes when fake nodes are used), no information about the node added to the path is revealed. Also note that because the BFS algorithm requires $\Omega(|V|^2)$ time, this solution would not increase the asymptotic complexity of the overall solution.

An asymptotically more efficient solution would be to retrieve one element of C at a time, where each consecutive element is set to the parent node of the previously retrieved node, and use ORAM techniques to hide information about the elements of C that have been accessed. This increases both the storage necessary to maintain ORAM for C as well as adds polylogarithmic computational overhead for each memory access. Because of the complexity of ORAM techniques and their simulation in a privacy-preserving framework in particular, this approach would provide computational advantage only for very large graphs.

Also note that logic similar to what we have previously used for BFS does not work here. That is, suppose we randomly shuffle vector C and directly access its element to retrieve the next node on the path. This shuffling guarantees that there is no correlation between the row accessed during BFS and vertices accessed during path reconstruction. Unfortunately, translating parent information of the accessed location to its shuffled version cannot be performed obliviously in constant time and we are back to either scanning

the entire vector or applying ORAM techniques to access the mapping information. For these reasons, we next describe a simple solution of $O(|V|^2)$ complexity.

In the description that follows, we conservatively assume that the location of destination node t in the adjacency matrix is not known (i.e., it is protected). This algorithm can then be easily modified for the case when the location of t in M and C is known. The initial solution, i.e., the one that does not hide the size of the path, is as follows:

Algorithm 2:

1. Execute BFS to compute shortest distances from source s to all other nodes.
2. Initialize the path P to $\langle [t] \rangle$. Set the current working node $[v]$ to $[t]$.
3. Scan vector C to find node $[w]$ such that $[w] = [C_v.parent]$. In more detail, we execute:
   1. $[w] = 0$
   2. for $i = 1$ to $2|V| - 1$ do
   3. $\quad [cond] = ([v] \stackrel{?}{=} i)$
   4. $\quad [w] = [cond] \cdot [C_i.parent] + (1 - [cond]) \cdot [w]$
4. Update the path as $P = \langle [w], P \rangle$ and set $[v]$ to $[w]$.
5. Repeat steps 3–5 until $[v] = s$.

When the location of t in the adjacency matrix is known, we can skip step 3 in the first iteration and directly add $[C_t.parent]$ to P and set $[v]$ to value $[C_t.parent]$ in step 4.

3.3.2 Hiding the length of the path

To protect information about the length of the path, we need to ensure that the algorithm always performs $|V| - 1$ iterations and produces a path of length $|V|$. To achieve this, we instruct the algorithm to continue adding nodes to the path if the source s is reached, but the path is shorter than $|V|$ vertices long.

To be able to hide the length of the path from s to t, we must hide the fact that $v = s$ and proceed with adding nodes to the path. For that reason, we add s to the path, but set one of the fake nodes as the current working node $[v]$. The algorithm will then keep adding fake nodes to the path until its length becomes $|V| - 1$.

To ensure that a sufficient number of fake nodes is added without repetitions, we set their parent information to form a cycle of length $|V| - 1$. Because the algorithm is oblivious, the parents of the fake nodes can be assigned in an arbitrary manner. We choose to assign them sequentially to form one cycle. More precisely, the parent of fake node v is set to fake node w with the highest index less than v (and the parent of the fake node with the lowest index is set to the fake node with the highest index). For that purpose, we utilize vector F to update parent information of fake nodes in vector C.

In more detail, the computation that we need to perform after step 1 of Algorithm 2 is as follows. We first scan F to find the index of the fake node with the highest number. It will be used to initialize the parent value (to be used for the fake node with the lowest index). We then scan the nodes in C updating parent values of fake nodes and the current parent value.

1. $[parent] = 0$
2. for $i = 1$ to $2|V| - 1$ do
3. $\quad [parent] = [F_i] \cdot i + (1 - [F_i])[parent]$
4. for $i = 1$ to $2|V| - 1$ do
5. $\quad [C_i.parent] = [F_i] \cdot [parent] + (1 - [F_i])[C_i.parent]$
6. $\quad [parent] = [F_i] \cdot i + (1 - [F_i])[parent]$

The only other change that we need to make to Algorithm 2 is to hide the fact that source s has been reached by selecting a fake node. To achieve this, we randomly select one of the fake nodes and set the current working node to that fake node if the source has been reached. Because we do not know at which step of the computation the source might be reached, we have to perform such testing at each iteration of the algorithm. Fortunately, because the source node can be reached only once, we can pre-select one of the fake nodes at the beginning of the algorithm (instead of doing it at each iteration), but test each time whether it should be chosen. This means that after assigning parent node information to the fake nodes as described above, we select one of the nodes at random as follows:

1. choose (protected) random permutation $\pi(\cdot)$
2. for $i = 1$ to $2|V| - 1$ do $[v_i] = [F_i] \cdot [\pi(i)]$
3. $[u] = 0$
4. for $i = 1$ to $2|V| - 1$ do
5. $\quad [cond_i] = ([v_i] \stackrel{?}{>} [u])$
6. $\quad [u] = [cond_i] \cdot i + (1 - [cond_i])[u]$

Lastly, we modify steps 4–5 of Algorithm 2 to the following:

4. Prepend $[w]$ to the path, i.e., $P = \langle [w], P \rangle$. If $[w]$ is different from s, set $[v]$ to $[w]$; otherwise, set $[v]$ to a randomly chosen fake node. That is, compute:
   1. $[cond] = ([w] \stackrel{?}{=} s)$
   2. $[v] = [cond] \cdot [u] + (1 - [cond])[w]$
5. Repeat steps 3–5 $|V| - 2$ times.

3.3.3 Handling unreachable destination

The algorithm that we described so far for shortest path computation works if there is a path from s to t. If, however, node t cannot be reached from the source node, the algorithm must still proceed with the computation protecting the fact that there is no path. According to our BFS algorithm, node t will have its parent set to $\perp$ if there is no path from s to t. We can use this information to slightly modify the algorithm and proceed with one of the fake nodes if $\perp$ has been reached. This introduces a very minor change to the algorithm described in section 3.3.2 because we can reuse the selected fake node for our purposes. This is a safe modification to the algorithm because a path will never simultaneously contain s and $\perp$. We therefore update step 4 of Algorithm 2 to the following:

4. Prepend $[w]$ to the path, i.e., $P = \langle [w], P \rangle$. If $[w]$ is different from s or $\perp$, set $[v]$ to $[w]$; otherwise, set $[v]$ to a randomly chosen fake node. That is, compute:
   1. $[cond_1] = ([w] \stackrel{?}{=} s)$
   2. $[cond_2] = ([w] \stackrel{?}{=} \perp)$
   3. $[cond] = [cond_1] + [cond_2] - [cond_1][cond_2]$
   4. $[v] = [cond] \cdot [u] + (1 - [cond])[w]$

In the above, we use $a + b - a \cdot b$ to implement $a \vee b$.

3.4 Analysis

For all of our oblivious graph algorithms, we first analyze their time complexity followed by their security analysis.

3.4.1 Complexity analysis

In this section, we analyze the complexities of our BFS algorithm and shortest path reconstruction.

It is easy to see that the complexity of steps 1–2 and 4–5 of the BFS algorithm (both the basic and general versions) is

$O(|V|)$, while the complexity of steps 3 and 6 is $O(1)$. Then because steps 4–6 are executed $|V|$ times, the overall runtime of $O(|V|^2)$, which is optimal for the adjacency matrix representation and for graphs with $|E| = \Theta(|V|^2)$. It also outperforms any solution that combines the conventional algorithm with ORAM when $|E| = \Omega(|V|^2/\log(|V|)^2)$.

If the labeling of the graph nodes is not guaranteed to be random (i.e., the labeling can reveal information about the structure of the graph) or the graph is not guaranteed to be connected and the fake nodes need to be placed at random locations in the graph, the nodes of the graph will need to be randomly permuted. When this step is implemented using oblivious sorting, its complexity is $O(|V|^2 \log(|V|))$, which dominates the complexity of the algorithm.

Our algorithm for SSSD shortest path computation also has $O(|V|^2)$ time. In particular, after executing BFS in step 1 of the algorithm, the only steps that have non-constant time is one-time pre-processing of fake nodes and step 3, both with complexity $O(|V|)$. Because steps 3–4 are performed $|V|-1$ times, we obtain the overall runtime of $O(|V|^2)$.

3.4.2 Security analysis

To show security of our BFS and shortest path algorithms, we show that they are oblivious with respect to Definition 1.

THEOREM 1. *The BFS algorithm is data-oblivious.*

PROOF. To prove this theorem, we analyze each major operation in the algorithm with respect to Definition 1. We show that for any given input graph $G = (V, E)$ and source s, (i) the sequence of executed instructions is the same as for all other input graphs with the same number of nodes $|V|$ and (ii) the memory accesses are indistinguishable from the memory accesses when the input G is a randomly generated graph with $|V|$ nodes.

The first three steps of the algorithm are independent of the input graph G, and step 1 is also independent of the source node s. This means that step 1 is exactly the same for all possible inputs. Step 2 performs the same operations for all inputs, but accesses and updates node s in C with different information than other nodes. Because according to the solution, s has a random location in the graph, its position is indistinguishable for real and randomly generated input graphs G. The same applies to step 3 of the algorithm.

Steps 4 and 5 execute the same sequence of instructions for all input graphs and access all memory locations of C and M_v in exactly the same manner, therefore they are identical for all input graphs. The only part that remains is to show that revealing the locations of $|V|$ nodes which are being processed by the algorithm cannot be used to extract information about the input $\langle G, s\rangle$. In particular, because of randomized order of the nodes, the revealed locations are random and cannot be used to extract information about the structure of the graph. Furthermore, when selecting the next candidate node, one of them is selected at random, which also protects information about the number of candidate nodes. We obtain that the revealed locations are random and the memory accesses are indistinguishable from those of randomly generated graphs. □

THEOREM 2. *The SSSD shortest path algorithm is data-oblivious.*

PROOF. Similar to the proof of BFS algorithm, we consider all steps of the SSSD shortest path algorithm and show that they are data-oblivious according to our definition.

Step 1 executes BFS and is data-oblivious according to Theorem 1. All remaining steps (including fake node selection) execute exactly the same sequence of operations for all inputs graphs and access exactly the same memory locations for all input graphs. This means that the execution and memory accesses are identical for all inputs and the algorithm is data-oblivious. □

4. MAXIMUM FLOW

In this section we provide an oblivious solution to another graph problem, namely, maximum flow. Before we can proceed with its description, we need to provide background information.

4.1 Background

A flow network is a directed graph $G = (V, E)$, where each edge $(v, u) \in E$ has a non-negative capacity $c(v, u) \geq 0$ (and if $(v, u) \notin E$, $c(v, u) = 0$). Given a source vertex $s \in V$ and a sink vertex $t \in V$, a flow f in G is a function $f : V \times V \to \mathbb{R}$ that must satisfy the properties of capacity constraint (i.e., for all $v, u \in V$, $f(v, u) \leq c(v, u)$), skew symmetry (i.e., for all $v, u \in V$, $f(v, u) = -f(u, v)$), and flow conservation (i.e., for all $v \in V \setminus \{s, t\}$, $\sum_{u \in V} f(v, u) = 0$). The value of a flow in the flow network is defined as the total flow out of the source s $|f| = \sum_{u \in V} f(s, u)$, and the maximum-flow problem is to find a flow of maximum value.

A standard way of computing maximum flow relies on the Ford-Fulkerson method, which proceeds as follows: We initialize the flow to 0, and while there exists an augmenting path p, we augment the flow f along the path p. Here an augmenting path is defined as a path from the source s to the sink t which can admit more flow and therefore can be used to increase the overall flow of the network.

In implementing the above high-level logic, existing algorithms rely on the notion of residual network, which intuitively consists of edges that can admit more flow. In more detail, given a flow network $G = (V, E)$ and a flow f, the residual network of G induced by f is $G_f = (V, E_f)$ with edges of positive capacity $E_f = \{(v, u) \in V \times V \mid c_f(v, u) > 0\}$, where $c_f(v, u) = c(v, u) - f(v, u)$ is the residual capacity of (v, u). An augmenting path p is then a simple path from s to t in the residual network G_f. This means that each edge on the path admits additional positive flow, and the (residual) capacity of p is defined as $c_f(p) = \min\{c_f(v, u) \mid (v, u) \text{ is on } p\}$.

The basic structure of the approach, which corresponds to the Ford-Fulkerson algorithm, is then as follows: On input $G = (V, E)$, $s \in V$, and $t \in V$,

1. for each $(v, u) \in E$ do
2. $\quad f(v, u) = 0$
3. $\quad f(u, v) = 0$
4. while there exists path p from s to t in residual network G_f do
5. $\quad c_f(p) = \min\{c_f(v, u) \mid (v, u) \text{ is on } p\}$
6. $\quad$ for each (v, u) in p do
7. $\quad\quad f(v, u) = f(v, u) + c_f(p)$
8. $\quad\quad f(u, v) = -f(v, u)$

This algorithm has complexity $O(|E| \cdot |f_{max}|)$, where f_{max} is the maximum flow returned by the algorithm, as finding a path in each iteration of the algorithm involves $O(|E|)$ time. For networks with integral capacities and small f_{max}, the runtime of this algorithm is good; however, in the gen-

eral case a variant known as the Edmonds-Karp algorithm is preferred. If we use a shortest path from s to t on line 4 of the algorithm, we obtain the Edmonds-Karp algorithm with complexity $O(|V| \cdot |E|^2)$. It is guaranteed to find the maximum flow in $O(|V| \cdot |E|)$ iterations, each of which involves $O(|E|)$ time (see, e.g., [16]). Finding a shortest path can be accomplished using BFS.

4.2 Oblivious algorithm

In our oblivious solution, we follow the overall structure of the algorithm and use the implementation of BFS and shortest path computation from Section 3. Because our oblivious BFS algorithm processes one node at a time and reveals its location in the adjacency matrix, we need to shuffle the rows and columns of the matrix between each iteration of the maximum flow solution to hide all access patterns. We also now must maintain the residual network and obliviously update it after computing the augmenting path p.

We obtain the solution, which takes as the input a flow network $G = (V, E)$ with the capacity function stored in the (protected) adjacency matrix M, source node s, and sink node t. We assume that positive capacity of edge (v, u) is stored in $[M_{v,u}]$, and $M_{v,u} = 0$ indicates that there is no edge from v to u. The algorithm proceeds as follows:

Algorithm 3:

1. Expand matrix M with $|V|-1$ fake nodes inserted into M consistently as rows and columns. The capacity of all edges (v, u) and (u, v) connecting two fake nodes v and u is set to c_{max}, where c_{max} refers to the maximum possible capacity of an edge. The capacity of all edges (v, u) and (u, v), where v is a node of the original graph G and u is a fake node is set to 0. As before, the information about which nodes of M are fake is maintained in bit vector F. The location of fake nodes does not need to be randomized at this step.
2. Create (protected) matrix M' for storing residual network G_f and initialize each element of it $M'_{i,j}$ to $M_{i,j}$. Also create (protected) matrix L for storing the flow function and initialize each element of it $L_{i,j}$ to 0.
3. Repeat the computation that follows $|V| \cdot |E|$ times.
4. Apply a random permutation to the rows and columns of M' and L as well as to the elements of F consistently. Reveal the location of s after the permutation, but keep the location of t protected.
5. Execute SSSD shortest path algorithm on the graph defined by M' using source s and destination $[t]$. Because our algorithms in Section 3 were defined for unweighted graphs, we need to introduce changes to work with weighted graphs and also preserve information about the capacity of each edge on the path. For that reason, for the purposes of the BFS algorithm, node u is considered to be adjacent to node v if the capacity of the edge (v, u) is positive, i.e., $M'_{v,u} > 0$. In addition, when updating vector C in step 4 of Algorithm 1 with parent and distance from the source information, we also preserve information about the capacity of the edge from the parent to the current node using matrix M'. This information is stored in field $C_i.capacity$ and is updated together with $C_i.parent$ and $C_i.dist$. That is, we execute:
   1. for $i = 1$ to $|V|$ do
   2. $\quad [cond] = ([M'_{v,i}] \stackrel{?}{>} 0) \cdot ([C_i.color] \stackrel{?}{=} white)$
   3. $\quad [C_i.color] = [cond] \cdot gray + (1 - [cond]) \cdot [C_i.color]$
   4. $\quad [C_i.parent] = [cond] \cdot v + (1 - [cond]) \cdot [C_i.parent]$
   5. $\quad [C_i.dist] = [cond]([C_v.dist] + 1) + (1 - [cond]) \cdot [C_i.dist]$
   6. $\quad [C_i.capacity] = [cond][M'_{v,i}] + (1 - [cond]) \cdot [C_i.capacity]$

   To preserve edge capacity in the reconstructed path P, we also introduce changes to Algorithm 2. In particular, instead of storing a single node $[w]$ in P, we now store a tuple of the form $\langle [v_1], [v_2], [v_3], [c] \rangle$. Here $[v_1]$ is the same as previously stored, i.e., the first elements of the stored tuples form a path in the graph padded in the beginning with fake nodes to be of length $|V|$. Nodes v_2 and v_3 represent an edge, and c is capacity. In almost all cases, $v_1 = v_2$ and (v_3, v_2) is the edge on the path of capacity c, i.e., v_3 is stored as the first element of the preceding tuple. The only time when this does not hold is during special cases when the source or no path symbol $\perp$ has been reached during path computation. In those cases, we set v_1 to s or $\perp$, but the edge (v_3, v_2) is between two fake nodes, and c is the capacity of that edge in M' when $v_1 = s$ and $c = 0$ when $v_1 = \perp$. Setting the capacity in such a way will allow us to ensure that the capacity of the path is computed correctly. For reasons that will soon become apparent, we also set $[v_2] = [v_1]$, $[v_3] = [t]$, and $[c] = 0$ in the last iteration of path reconstruction (i.e., the first tuple on the path). The path reconstruction algorithm becomes:

   (a) Initialize the path P to empty and set $[v] = [v'] = [t]$. Randomly choose a fake node as before and store it in $[u]$.

   (b) Repeat $|V| - 1$ times:

      i. Scan vector C to retrieve parent and edge capacity information and compute data to be added to the path by executing the following:
         1. $[w] = 0$
         2. $[c] = 0$
         3. for $i = 1$ to $2|V| - 1$ do
         4. $\quad [cond] = ([v] \stackrel{?}{=} i)$
         5. $\quad [w] = [cond][C_i.parent] + (1 - [cond])[w]$
         6. $\quad [c] = [cond][C_i.capacity] + (1 - [cond])[c]$
         7. $[cond] = (v' \stackrel{?}{=} \perp)$
         8. $[v_1] = [v']$
         9. $[v_2] = [v]$
         10. $[v_3] = [w]$
         11. $[c] = (1 - [cond])[c]$
         12. $[cond_1] = ([w] \stackrel{?}{=} s)$
         13. $[cond_2] = ([w] \stackrel{?}{=} \perp)$
         14. $[cond] = [cond_1] + [cond_2] - [cond_1][cond_2]$
         15. $[v'] = [w]$
         16. $[v] = [cond] \cdot [u] + (1 - [cond])[w]$

         where, as before u, is a randomly selected fake node. The purpose of v' here is simply to store node information from the previous round (in case v is set to u, while we need to remember w as well).

      ii. Prepend $\langle [v_1], [v_2], [v_3], [c] \rangle$ to P.

   (c) Set $[v_1] = [v']$, $[v_2] = [v']$, $[v_3] = [t]$, and $[c] = [0]$ and prepend $\langle [v_1], [v_2], [v_3], [c] \rangle$ to P.

6. Compute the residual capacity $[c_f]$ of the path in P. Oblivious execution of this step is not difficult:
   1. $[c_f] = [c_{max}]$
   2. for $i = 2$ to $|V|$ do
   3. $\quad [cond] = ([P_i.capacity] \overset{?}{<} [c_f])$
   4. $\quad [c_f] = [cond] \cdot [P_i.capacity] + (1 - [cond])[c_f]$

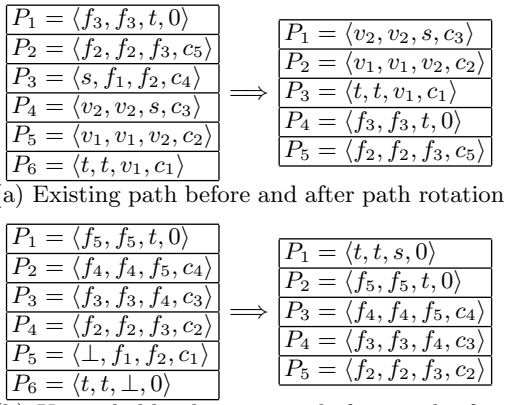
   Here P_i is the ith element of the path P. Note that the first tuple on the path is ignored as its capacity is always 0, i.e., only $|V|-1$ edges are used in computing the path's capacity.
7. To be able to update the residual network M' and flow function L, we obliviously rotate the entries in the path P to have the first record contain an edge that leaves the source s. This will allow us to proceed with one edge of P at a time updating cells of M' and L in the next step.
   When there is no path between s and t, vertex s does not appear in P. To hide this information, we replace $\perp$ with s in tuple $\langle [t], [t], [\perp], [0] \rangle$.[1] This does not affect correctness of the algorithm. We also ignore the preceding record (of the form $\langle [s], [v_2], [v_3], [c] \rangle$ or $\langle [\perp], [v_2], [v_3], [c] \rangle$), i.e., process only $|V| - 1$ edges.
   A simple way of performing oblivious rotation of the tuples in P is as follows:
   1. for $i = 1$ to $|V| - 1$ do
   2. $\quad [cond] = ([P_i.v_3] \overset{?}{=} s)$
   3. $\quad [temp] = [P_1]$
   4. $\quad$ for $j = 1$ to $|V| - 1$ do
   5. $\quad\quad [P_j] = [cond][P_j] + (1 - [cond])[P_{j+1}]$
   6. $\quad [P_{|V|}] = [cond][P_{|V|}] + (1 - [cond])[temp]$
   7. remove P_1 from P

   For simplicity of exposition, we use assignment $[x] = [P_i]$ to indicate that the entire tuple at the ith position of P is stored in variable x (where the value of i is known, while the content of values at P_i is not).
   After rotating the elements of P in this manner, we have that P stores a path from s to t padded at the end with fake nodes and edges between them (and if no path exists, there will be a path in P, but its capacity is 0, which means that the residual network will not be modified). Note that there is an edge leaving t of capacity 0 (i.e., the first element of P prior to its rotation) which correctly forms the path and transitions to a fake node (after which all nodes on the path are fake). Figure 2 demonstrates how P is formed before and after path rotation for two cases: (i) when a path from s to t is present and (ii) when there is no path from s to t. Notation v_i refers to an original node of the graph, and notation f_i is used for fake graph nodes.
8. Update residual network M' and flow function L. Now because we use random labeling of vertices in the graph, we can update M' and L by revealing the path information (i.e., open edges (v_3, v_2) in each element of P) and update the corresponding cells of M' and L with (still protected) path capacity information. We, however, already revealed a certain sequence of nodes during BFS computation, and to ensure that there is no correlation between memory accesses during BFS and during residual network update, we need to randomly shuffle the nodes again. This time, in addition to permuting the rows and columns of M' and L, as well as vector F, we also need to map each node v contained in path P to its permuted index $\pi(v)$. A simple way of accomplishing this is to scan the $\langle i, [\pi(i)] \rangle$ pairs for each v in P and replace v with $\pi(i)$ when $v = i$. That is, for each node v included in P, we execute:
   1. for $i = 1$ to $2|V| - 1$ do
   2. $\quad [cond] = ([v] \overset{?}{=} i)$
   3. $\quad [\pi(v)] = [cond][\pi(i)] + (1 - [cond])[\pi(v)]$

   It is important that the location of t is not known after the random permutation in each iteration and the nodes on any given path do not repeat. The cells of M' and L are then updated as follows:
   1. for $i = 1$ to $|V| - 1$ do
   2. $\quad$ open v_3 and v_2 in P_i
   3. $\quad [L_{v_3,v_2}] = [L_{v_3,v_2}] + [c_f]$
   4. $\quad [L_{v_2,v_3}] = -[L_{v_3,v_2}]$
   5. $\quad [M'_{v_3,v_2}] = [M'_{v_3,v_2}] - [c_f]$
   6. $\quad [M'_{v_2,v_3}] = [M'_{v_2,v_3}] + [c_f]$

   At the end of the computation we output the maximum flow as the total flow $\sum_{i=1}^{|V|} [L_{s,i}]$ leaving the source node s. If the flow information is desired with respect to the original node labeling, the algorithm can maintain the composition of random permutations used at each iteration.

[1]This step can be performed at the time of forming the tuples in P, but for clarity of presentation we choose to form the path correctly in step 5.



(a) Existing path before and after path rotation

(b) Unreachable destination before and after path rotation

Figure 2: Example of path computation for $|V| = 6$.

4.3 Analysis

4.3.1 Complexity analysis

To show that our algorithm runs in $O(|V|^3|E|\log(|V|))$ time, we analyze the complexities of each step of it. In the algorithm, steps 1, 2, 5, and 7 take $O(|V|^2)$ time, while the complexity of step 6 is $O(|V|)$ and that of steps 4 and 8 is $O(|V|^2 \log(|V|))$. Because steps 4–8 are repeated $O(|V||E|)$ times, we obtain the overall complexity as claimed. Our complexity is higher by a factor $\log(|V|)$ than that of Edmonds-Karp's algorithm when the adjacency matrix is used or the graph is dense with $|E| = O(|V|^2)$. It also outperforms any solution that combines the Edmonds-Karp algorithm with ORAM when $|E| = \Omega(|V|^2/\log(|V|))$.

4.3.2 Security analysis

We show the security of the maximum-flow algorithm as before using Definition 1.

THEOREM 3. *The maximum-flow algorithm is data-oblivious.*

PROOF. As before, we analyze each major operation in the algorithm with input $G = (V, E)$, s, and t. In steps 1 and 2, we perform identical operations and touch exactly the same locations for all inputs with $|V|$ nodes and therefore the steps are data-oblivious. Step 3 is merely an iterator with the same number of iterations for all graphs with $|V|$ nodes and $|E|$ edges, so we proceed directly to step 4. Step 4 only executes a data-oblivious shuffle operation and is therefore oblivious. Step 5 calls BFS and SSSD shortest path algorithms with small changes to preserve edge capacity information. These algorithms have been shown to be data-oblivious in Theorems 1 and 2, respectively. Step 6 performs identical operations and accesses the same memory locations for all graphs with $|V|$ nodes and is therefore data-oblivious. In step 7, the algorithm rotates the path, once again executing identical operations for all inputs and accessing the same memory locations. Lastly, in step 8, we execute the same operations for all graphs, but accessed memory locations differ. Our algorithm crucially relies on two facts to achieve data-obliviousness: (i) the vertices are randomly permuted prior to any access is made and (ii) all updates follow a path of size $|V|$, which always starts with the source s and has no repeated nodes on it. This means that, besides for the first accessed node s, all other memory accesses are random and cannot be distinguished from accesses used for a randomly generated input graph. Lastly, because the algorithm proceeds in iterations, we ensure that every time a subset of memory locations is accessed (i.e., during BFS and when updating the residual network), the nodes are randomly permuted right before that operations. This ensures that all accesses cannot be distinguished from a random sequence of vertex accesses. □

5. MINIMUM SPANNING TREE

Given the already developed techniques, it is not difficult to design an oblivious algorithm for computing the minimum spanning tree that runs in $O(|V|^2)$ time (unless the node labeling reveals information about the structure of the graph). Our algorithm uses the structure of Prim's algorithm, and we start by describing background information.

5.1 Background

On input connected undirected weighted graph $G = (V, E)$, Prim's algorithm initializes the spanning tree $T = (V', E')$ to a single arbitrary vertex from V and no edges. Then until $V' \neq V$, the algorithm selects an edge (v, u) with the minimum weight such that $v \in V'$ and $u \notin V'$ and sets $V' = V' \cup \{u\}$ and $E' = E' \cup \{(v, u)\}$. The complexity of this algorithm depends on the graph representation and data structures used. Using the adjacency matrix, the algorithm runs in $O(|V|^2)$ time; using the adjacency list, the algorithm's performance is $O(|E| \log(|V|))$ or $O(|E| + |V| \log(|V|))$ if a binary heap or Fibonacci heap is used, respectively.

5.2 Oblivious algorithm

In order to compute a minimum spanning tree obliviously without incurring a large amount of additional overhead, we maintain a vector with candidate nodes to potentially be added to the tree T and select one of them which has an edge with the minimum weight connecting it to T. The vector is updated each time a node is added to V'. To efficiently implement this step, we crucially rely on the ability to process a single row of the matrix, which can be safely performed only when the node labeling is random and does not reveal information about the structure of the graph. We obtain the algorithm given below, which takes as input adjacency matrix M representing the graph G. Because edge weights in G can be arbitrary, we assume that each cell $M_{v,u}$ of M contains two fields: (i) Boolean field *adjacent*, which is set if $(v, u) \in E$ and (ii) numeric value *weight* indicating the weight of the edge (v, u).

Algorithm 4:

0. If the numbering of the nodes in G may convey information about the structure of the graph, randomly and consistently permute the rows and columns of adjacency matrix M.
1. Choose a single node v from G, e.g., at position 1 in M. Set $V' = \{v\}$ and $E' = \emptyset$. Create (protected) vector C of size $|V|$, where each element of C, C_i, contains two fields: *weight* and *parent*. Initialize each $[C_i.weight]$ to ∞ (which for practical purposes can be any constant larger than any weight in M) and each $[C_i.parent]$ to $\perp$.
2. Retrieve row at position v from M and update C using M_v. That is, for each i we store in C_i the minimum of C_i and the weight at $M_{v,i}$ if v is adjacent to i. Then if $C_i.weight$ is updated to the weight stored at $M_{v,i}$, we also store v in $C_i.parent$. The pseudo-code for oblivious implementation of this step is:
   1. for $i = 1$ to $|V| - 1$ do
   2. $\quad [cond] = ([M_{v,i}.weight] \overset{?}{<} [C_i])[M_{v,i}.adjacent]$
   3. $\quad [C_i.weight] = [cond][M_{v,i}.weight] + (1 - [cond]) \cdot [C_i.weight]$
   4. $\quad [C_i.parent] = [cond] \cdot v + (1 - [cond])[C_i.parent]$
3. Locate the minimum element of C, $C_{i_{min}}$, considering only locations i such that $i \notin V'$. That is, execute:
   1. $[min] = \infty$
   2. $[i_{min}] = 0$
   3. for $i = 1$ to $|V| - 1$ do
   4. $\quad$ if $i \notin V'$ then
   5. $\quad\quad [cond] = ([C_i.weight] \overset{?}{<} [min])$
   6. $\quad\quad [min] = [cond][C_i.weight] + (1 - [cond])[min]$
   7. $\quad\quad [i_{min}] = [cond] \cdot i + (1 - [cond])[i_{min}]$
   8. $i_{min} = open(i_{min})$
4. Update the minimum spanning tree by setting $V' = V' \cup \{i_{min}\}$, $E' = E' \cup \{([C_{i_{min}}.parent], [i_{min}])\}$, and $v = i_{min}$. Repeat steps 2–4 $|V| - 2$ times and output the edges E'.

Recall that the nodes of the graph are randomly permuted and the order in which they are added to V' does not reveal information about the graph. This means that the nodes that form the set V' are not protected, where each node is added to V' exactly once. This means that in step 3 we need to consider only the nodes that have not yet been added to the minimum spanning tree.

Correctness of this algorithm follows from the correctness of Prim's algorithm.

5.3 Analysis

5.3.1 Complexity analysis

The performance of our algorithm is not difficult to assess: steps 1–3 take $O(|V|)$ time, while step 4 takes $O(1)$ time. Because these steps are repeated $O(|V|)$ times, we arrive at $O(|V|^2)$ overall time. It is optimal with matrix adjacency representation and when $|E| = \Theta(|V|^2)$. If step 0 is necessary, however, the overall time raises to $O(|V|^2 \log(|V|))$.

5.3.2 Security analysis

To show that the minimum spanning tree algorithm is oblivious, we as before follow Definition 1 and show that the revealed information is indistinguishable from random.

THEOREM 4. *The minimum spanning tree algorithm is data-oblivious.*

PROOF. We analyze each major operation in the algorithm with input $G = (V, E)$. Step 0 calls random shuffling, which can be performed obliviously. Step 1 does not use the input, only the input's size. In step 2, after retrieving a row of the adjacency matrix, the execution that follows uses the same instructions and accesses the same memory for all input graphs. Then, because this step is executed multiple times, each iteration accesses a row corresponding to a unique node. Since we know that the node ordering is random and each node is used exactly once, the access pattern induces a random permutation on the set of nodes and is therefore indistinguishable from access patterns of randomly chosen graphs. Steps 3 and 4 execute the same instructions and access the same memory locations for all input graphs and are therefore data-oblivious. □

6. BUILDING SECURE PROTOCOLS

In this section we briefly illustrate how our oblivious algorithms can be used to build protocols for graph problems suitable for secure computation and outsourcing. We use the BFS algorithm for illustration purposes.

Using the setting of section 1.3 with respect to computation participants, we denote the computational parties as $P_1, \ldots, P_n$ and define security in presence of semi-honest participants (who follow the prescribed computation, but might attempt to learn additional information about private data from the messages that they receive) as follows:

DEFINITION 2. *Let parties $P_1, \ldots, P_n$ engage in a protocol Π that computes function $f(\mathsf{in}_1, \ldots, \mathsf{in}_n) = (\mathsf{out}_1, \ldots, \mathsf{out}_n)$, where in_i and out_i denote the input and output of party P_i, respectively. Let $\mathrm{VIEW}_\Pi(P_i) = (\mathsf{in}_i, r_i, m_1, \ldots, m_k)$ denote the view of participant P_i during the execution of protocol Π, which is formed by its input, internal random coin tosses r_i, and messages $m_1, \ldots, m_k$ passed between the parties during protocol execution. Let $I = \{P_{i_1}, P_{i_2}, \ldots, P_{i_t}\}$ denote a subset of the participants for $t < n$ and $\mathrm{VIEW}_\Pi(I)$ denote the combined view of participants in I during the execution of protocol Π (i.e., the union of the views of the participants in I). We say that protocol Π is t-private in presence of semi-honest adversaries if for each coalition of size at most t there exists a probabilistic polynomial time simulator S_I such that $\{S_I(\mathsf{in}_I, f(\mathsf{in}_1, \ldots, \mathsf{in}_n)\} \equiv \{\mathrm{VIEW}_\Pi(I), \mathsf{out}_I\}$, where $\mathsf{in}_I = \bigcup_{P_i \in I}\{\mathsf{in}_i\}$, $\mathsf{out}_I = \bigcup_{P_i \in I}\{\mathsf{out}_i\}$, and $\equiv$ denotes computational or statistical indistinguishability.*

To obtain a solution that complies with the above security definition, we employ a (n, t) threshold linear secret sharing scheme (such as [35]), using which a secret s is split into n shares. Then any $t + 1$ shares can be used to reconstruct s exactly, while possession of t or fewer shares information-theoretically reveals no information about s. Using such techniques for data protection, we obtain the following secure protocol for BFS computation:

1. The input party/parties distribute shares of the graph (in the form of adjacency matrix M) to the computational parties and indicate what node is the source.
2. If the node labels are not random, the computational parties use a random permutation to randomize the graph representation.
3. The parties execute Algorithm 1 on shares using the following building blocks for computing with shares:
   (a) addition and subtraction of shares, multiplication of a shared value by a known or shared value constitute elementary operations;
   (b) comparisons and equality tests are implemented using protocols LT and Eq, respectively, from [13];
   (c) random permutation is achieved by first choosing random labels for all elements to be permuted, e.g., by calling PRandFld from [13], and obliviously sorting the elements using the chosen random labels as the sorting key as, e.g., shown in [5].
4. The parties send their shares of the result to the output party/parties.

Security of the above solution can be shown based on the facts that (1) the building blocks have been previously proven secure in the same security model and (2) composition of secure building blocks leads to the security of the overall solution using the composition theorem from [12]. In more detail, to build a simulator as specified in Definition 2, we can invoke the corresponding simulators of the building blocks to simulate the view of a coalition of computational parties which is indistinguishable from the real protocol execution.

7. CONCLUSIONS

In this work we design data-oblivious algorithms for several classical graph problems, namely, breadth-first search, single-source single-destination shortest path, minimum spanning tree, and maximum flow. The algorithms are designed to work on protected data and have applications to secure computation and outsourcing. The algorithms have optimal, or close to optimal, performance for dense graphs or when adjacency matrix is used to represent the input graphs. It is an open problem to design efficient data-oblivious algorithms for sparse graphs.

8. ACKNOWLEDGMENTS

This work was supported in part by grants CNS-1223699 from the National Science Foundation, FA9550-13-1-0066 from the Air Force Office of Scientific Research, and FA8750-11-2-0218 from Air Force Research Laboratory. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding agencies.

9. REFERENCES

[1] M. Ajtai. Oblivious RAMs without cryptographic assumptions. In *STOC*, pages 181–190, 2010.

[2] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. Munro. Cache-oblivious priority queue and graph algorithm applications. In *STOC*, pages 268–276, 2002.

[3] L. Arge, M. Bender, E. Demaine, B. Holland-Minkley, and J. Munro. An optimal cache-oblivious priority queue and its application to graph algorithms. *SIAM Journal on Computing*, pages 1672–1695, 2007.

[4] M. Atallah, K. Frikken, and S. Wang. Private outsourcing of matrix multiplication over closed semi-rings. In *SECRYPT*, pages 136–144, 2012.

[5] M. Blanton and E. Aguiar. Private and oblivious set and multiset operations. In *ASIACCS*, 2012.

[6] M. Blanton, M. Atallah, K. Frikken, and Q. Malluhi. Secure and efficient outsourcing of sequence comparisons. In *ESORICS*, pages 505–522, 2012.

[7] D. Boneh, D. Mazieres, and R. Popa. Remote oblivious storage: Making oblivious RAM practical. Technical Report MIT-CSAIL-TR-2011-018, MIT, 2011.

[8] J. Brickell and V. Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *ASIACRYPT*, pages 236–252, 2005.

[9] G. Brodal. Cache-oblivious algorithms and data structures. In *SWAT*, pages 3–13, 2004.

[10] G. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *SWAT*, pages 480–492, 2004.

[11] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylog communication. In *EUROCRYPT*, pages 402–414, 1999.

[12] R. Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[13] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In *SCN*, pages 182–199, 2010.

[14] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, pages 41–50, 1995.

[15] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, pages 965–981, 1998.

[16] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. The MIT Press, 2009.

[17] I. Damgård, S. Meldgaard, and J. Nielsen. Perfectly secure oblivious RAM without random oracles. In *TCC*, pages 144–163, 2011.

[18] D. Eppstein, M. Goodrich, and R. Tamassia. Privacy-preserving data-oblivious geometric algorithms for geographic data. In *GIS*, pages 13–22, 2010.

[19] C. Erway, A. Kupcu, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *CCS*, pages 213–222, 2009.

[20] P. Failla. Heuristic search in encrypted graphs. In *SECURWARE*, pages 82–87, 2010.

[21] K.B. Frikken and P. Golle. Private social network analysis: how to assemble pieces of a graph privately. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, pages 89–98, 2006.

[22] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, pages 803–815, 2005.

[23] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987.

[24] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996.

[25] M. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. In *SODA*, pages 1262–1277, 2010.

[26] M. Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, pages 379–388, 2011.

[27] M. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, pages 576–587, 2011.

[28] M. Goodrich, O. Ohrimenko, and R. Tamassia. Data-oblivious graph drawing model and algorithms. Arxiv preprint arXiv:1209:0756, 2012.

[29] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156, 2012.

[30] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, pages 364–373, 1997.

[31] H. Lipmaa. An oblivious transfer protocol with log-squared total communication. In *Information Security Conference (ISC)*, pages 314–328, 2005.

[32] K. Mouratidis and M. Yiu. Shortest path computation with no information leakage. *VLDB Endowment*, 5(8):692–703, 2012.

[33] R. Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523, 1990.

[34] B. Pinkas and T. Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519, 2010.

[35] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[36] E. Shi, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.

[37] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[38] S. Wang, X. Ding, R. Deng, and F. Bao. Private information retrieval using trusted hardware. In *ESORICS*, pages 49–64, 2006.

[39] P. Williams and R. Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[40] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: Practical access pattern privacy and correctness on untrusted storage. In *CCS*, pages 139–148, 2008.

[41] A. Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.