Graph Drawing in the Cloud: Privately Visualizing Relational Data Using Small Working Storage*

Michael T. Goodrich[1], Olga Ohrimenko[2], and Roberto Tamassia[2]

[1] Dept. Computer Science, Univ. of California, Irvine
goodrich@acm.org
[2] Dept. Computer Science, Brown University
{olya,rt}@cs.brown.edu

Abstract. We study graph drawing in a cloud-computing context where data is stored externally and processed using a small local working storage. We show that a number of classic graph drawing algorithms can be efficiently implemented in such a framework where the client can maintain privacy while constructing a drawing of her graph.

1 Introduction

In this paper, we present techniques that allow a client to efficiently execute various classic graph drawing algorithms, and variations of them, in a cloud computing environment, where the storage of the graph is outsourced to an online storage service.

We are particularly interested in allowing a client to access her data and perform computations on them in a *privacy-preserving* way. For example, an administrator for a fast-growing company may be revising (and visualizing) the organizational chart for the leadership of her company, and leaking this chart to the press or a rival could negatively impact the company. Thus, we view the storage server as an *honest-but-curious* adversary, who correctly performs the storage and retrieval operations requested by the client, but is nevertheless interested in learning as much from her data as possible (indeed, some cloud computing companies are basing their business model on this goal).

Of course, in a cloud computing scenario, the client would encrypt the data she outsources, decrypting it when she retrieves it, and re-encrypting it when she stores it back (using a probabilistic cipher that is unlikely to repeat the same cipher text for a re-encryption of the same plaintext). But she may also be leaking information to the server from the pattern of her data accesses to the storage server. For example, accessing the memory associated with a certain department while preparing a new organizational chart leaks the fact that that department is being reorganized. So the client should additionally aim at completely hiding her access patterns in order to achieve privacy protection for her data.

* Research supported in part by the National Science Foundation under grants 0830149, 0830403, 1228485, 1228639, and 1212508 and by a NetApp Faculty Fellowship. We would like to thank Giuseppe Di Battista for useful discussions.

W. Didimo and M. Patrignani (Eds.): GD 2012, LNCS 7704, pp. 43–54, 2013.

Oblivious Algorithms and Storage. The general techniques of *oblivious RAM simulation* and *oblivious storage* allow a client to simulate an arbitrary algorithm in such a cloud-computing environment so as to hide both the content and access patterns for her computation (e.g., see [14–17]). But these solutions involve fairly complicated simulation techniques for generic algorithms that increase the running time of the client's algorithm by a polylogarithmic factor when the client has a small workspace.

Privacy-preserving algorithms in the cloud computing scenario have been developed for sorting [13] and for fundamental computational geometry problems on planar point sets, including convex hull, well-separated pair decomposition, compressed quadtree construction, closest pairs, and all nearest neighbors [9]. These algorithms also hide the access pattern from the server and are referred to as *data-oblivious*. In this paper, we develop simple privacy-preserving algorithms for some classic graph drawing problems that fully obfuscate the access pattern from the data server. Our algorithms are provably data-oblivious and utilize small workspace.

Related Work. There are existing web-based systems that can perform graph drawing services for clients, such as the Brown Graph Server [5] and Grappa [2]. These differ from the framework we are describing in this paper in two ways. First, our model involves the client storing her data in an outsourced data server and accessing that data remotely, whereas the web-based graph drawing services involve a client storing her data locally and temporarily shipping it to the server. Second, in the framework we are describing here, the client performs the graph drawing algorithm herself, not the server (because of privacy concerns), whereas the web-based drawing services employ their own graph drawing algorithms to produce layouts for the client.

Our approach is probably most similar to prior work on computations on data streams (e.g., see [1, 19, 21]). In this model, data is presented in single stream, which arrives in an arbitrary order and is processed in an online, read-only fashion using a workspace of small size. Each time an item is considered, all the processing involving that item has to be completed before considering the next item. Henzinger *et al.* [19] introduce a version of this model that allows for a small number of passes over the data using a small workspace, but their approach still assumes that data is presented in a read-only fashion in an arbitrary order (although they do leave as an open problem whether allowing for alternative orderings can reduce the size of workspace in some cases). In addition, Feldman *et al.* [10] define the MUD model for describing MapReduce algorithms, which also involves scans and small local workspace, but in their model scans are over these workspaces rather than a large set of data.

In the context of graph drawing, Binucci *et al.* [4] describe a framework for drawing trees in the streaming model, where one draws trees using a single scan of the edges, using a framework that is similar to our approach but nonetheless has some important differences. Specifically, as in the traditional data streaming model, their approach only allows for a single scan of the edges of a tree in an order that is not under the control of the algorithm. In our case, the client can make multiple scans of her data and specify the ordering of the scan each time. In addition, in their model, once a node is placed it cannot be moved, whereas we allow for the client to make tentative assignments of coordinates in one scan that can be refined or changed in a future scan, since this more naturally fits the approach of cloud computing.

Our Results. To enable data-oblivious algorithms for graph drawing problems, we introduce *compressed-scanning*, an algorithmic design framework based on a series of scans. Our method is related to the massive, unordered, distributed (MUD) model [10] for efficient computation in the map-reduce framework. We assume that the server holds a set of n data items and the client has a small private workspace of size $O(\log n)$. The data items at the server are encrypted with a semantically secure (probabilistic) cipher so that it is hard for the server to determine whether two items are equal.

An algorithm for the compressed-scanning model consists of a sequence of rounds, where in each round the entire data set is scanned by the client. During the scan, each item is processed exactly once by the client: first the client downloads the item from the server into workspace; next, the client performs some internal-memory computation on the item and the content of the workspace; finally the item is written out to an output stream at the server. When a round is completed, the output stream is either confirmed as the algorithm's output or it is used as the input data set for the next round. The efficiency of such an algorithm is measured, therefore, by the number of rounds needed and the size of the local workspace that is required. Ideally, the number of rounds should be $O(1)$ and the workspace should be sublinear. As shown in Section 2 an algorithm designed in the compressed-scanning framework can be implemented in a data-oblivious way by randomly shuffling the items in between scans.

Using the compressed-scanning approach, we provide efficient data-oblivious algorithms for a number of classic graph drawing methods [7], including symmetric straight-line drawings and treemap [20] drawings of trees, dominance drawings of planar acyclic digraphs [8], and Δ-drawings of series-parallel graphs [3]. Our methods result in privacy-preserving graph drawing algorithms with a smaller overhead than could be achieved by applying general-purpose privacy-preserving techniques (e.g., see [14–17]).

2 Compressed-Scanning

In this section, we formally define the *compressed-scanning* model for designing client-server algorithms that can be efficiently implemented using a small workspace, W, at the client. We assume that the server holds an array, S, of n data elements.

Model. An algorithm for our model consists of a sequence of t rounds. A round involves accessing each of the elements of S exactly once in a read-compute-write operation. This operation consists of reading an element from the server into private workspace, using the element in some computation, and writing a new element to an output stream, O, at the server. When a round completes, either the output stream O and/or a set of values in W are confirmed as the output of the algorithm, or we assign $S = O$ and start the next round. Hence, the running time of an algorithm in our model is $O(tn)$.

This size of the workspace, W, is a parameter of our model, and is intended to be small (e.g., constant or $O(\log n)$). The name of our model is derived from the fact that each round scans the set S and computations are performed using a small, or "compressed", amount of workspace. Note that our compressed-scanning model generalizes the standard data streaming model.

Privacy Protection. Suppose we are given a compressed-scanning algorithm, A, which runs in t rounds using a workspace, W, and a data set, S, of size n. We can implement A in a privacy-preserving way as follows.

The first essential step in ensuring privacy is the encryption of the elements in S. From now on we assume that the input stream, S, is stored encrypted at the server and whenever we write elements to the output stream, O, we also encrypt them. We use semantically secure encryption [12], which takes as input the plaintext and a random value. Thus, if the same element is encrypted twice, the resulting ciphertexts are different. With semantically secure encryption the server will not be able to distinguish whether two data elements are equal or whether the output element of a read-compute-write operation is equal to the input element.

The next step in ensuring privacy is hiding the access pattern from the server. In other words, the accesses to S should be data independent in each round and one should not be able to correlate accesses between the rounds. Each round in our model consists of scanning S one element at a time, performing local computations using the value of this element, and possibly modifying it and writing it back. Even if we have nothing to output, we can always write a dummy element, for the sake of being oblivious. However, a single scan is not enough to perform complex computations over data. The computation in the next round usually relies on computations from previous rounds and may require rearrangement of the data to allow a sequential access of that round. This shuffle of the data can be carried out by sorting over one of element's fields.

We employ an oblivious sorting algorithm for the purpose of hiding the correlation of accesses between the rounds. As mentioned earlier, several oblivious sorting techniques have been developed. Each oblivious sorting algorithm B offers a tradeoff between the time it takes the client to sort n items, $\mathsf{sort}_B(n)$, and the size of the client's private workspace, $\mathsf{workspace}_B(n)$. In oblivious merge sort, either $\mathsf{sort}_B(n)$ is $O(n \log^2 n)$ and $\mathsf{workspace}_B(n)$ is constant or $\mathsf{sort}_B(n)$ is $O(n)$ and $\mathsf{workspace}_B(n)$ is $O(\sqrt{n})$ [14]. In oblivious randomized shell short [13], which succeeds with high probability, we have that $\mathsf{sort}_B(n)$ is $O(n \log n)$ and $\mathsf{workspace}_B(n)$ is constant. We can use one of the above methods depending on the tradeoff we are willing to take and from now on, we refer to the oblivious algorithm as a black box algorithm, B.

In conclusion, our simulation of algorithm A consists of t scans of S and a call to an oblivious sort procedure B between the rounds. Each round requires $O(\mathsf{sort}_B(n))$ time while fully hiding the pattern of access to the items in S. Thus, the simulation of A takes time $O(t\ \mathsf{sort}_B(n))$ and uses workspace of size proportional to that of A plus the space required between the rounds for sorting, $\mathsf{workspace}_B(n)$.

Definition 1. *A probabilistic algorithm A is* data-oblivious *if given two inputs of the same size, I_1 and I_2, the accesses that A makes to the memory for I_1 and I_2 have the same probability distribution.*

In other words, one cannot distinguish between I_1 and I_2 by just looking at their access patterns. For example, consider an algorithm that scans the elements of a sorted array and writes to the output stream, O, only distinct elements. This algorithm is not data-oblivious since, given inputs $(1, 1, 1, 2)$ and $(1, 2, 2, 2)$, the write accesses to O happen after a different number of read accesses are made to the input stream. A data-oblivious algorithm would write a value to O for every element it reads from the input: a dummy element if the same element as the previous one is read, and a real one, otherwise. One can then make a simple sorting pass over O to bring real items to the front of the list. A workspace of constant size is used to store the last read element.

Theorem 1. *Let A be an algorithm in the compressed-scanning model for an input of size n that uses a workspace of size* $\mathsf{workspace}_A(n)$*. Algorithm A can be simulated by a data-oblivious algorithm if the number of rounds and the number of elements written to the output stream at each round depend only on n. Also, the simulation uses a workspace of size* $O(\mathsf{workspace}_A(n) + \mathsf{workspace}_B(n))$ *and runs in time* $O(t\ \mathsf{sort}_B(n))$*, where t is the number of rounds of A and B is an oblivious sorting algorithm.*

Proof. (*Sketch*) Each round is simulated by reading elements from S, writing elements to O, and reshuffling the next input set. Accesses to locations in S are made in a sequential order. This ensures that accesses to S in a single round are data-oblivious. Write accesses to O are also data-oblivious, since they happen on every access to S. After every round, the input sequence is reshuffled (data-obliviously); hence, one cannot correlate accesses between rounds as well. Thus, accesses to S and O depend only on size of S while the number of rounds is fixed by the algorithm regardless of S. □

In the next section we describe graph drawing algorithms that fit the compressed-scanning model and, hence, can be implemented in a data-oblivious manner. [1]

3 Graph Drawing Algorithms

Most existing graph drawing algorithms are designed without privacy concerns in mind; hence, if they are run in a cloud-computing environment, they can reveal potentially sensitive information from their access patterns. For example, a recursive binary-tree drawing algorithm implemented in the standard way can reveal the depth of the tree from the access patterns used for the recursion stack, even if all the nodes in the tree are encrypted. In this section, we present several graph drawing algorithms modified to fit the compressed-scanning model. We modify the representation of the graph so that we never access the same location more than once in the same round. For example, consider a tree represented with a set of nodes and pointers from each node to its children and a parent. Traversing the tree in this case involves accessing an internal node several times depending on its degree, which reveals information about the tree.

Euler Tours in the Compressed-Scanning Model. Traversing a tree in the compressed-scanning model requires that we access each memory location exactly once; hence, we need to reorganize how we normally perform data accesses, since, for example, we cannot access a parent again when coming from its left child after we have already visited it. Given small private storage, W, we cannot store previously accessed nodes. Thus, we need a representation of a tree that allows for a traversal where elements are accessed only once. For this purpose, we construct an Euler tour over a tree that is based on duplicating edges and defines a left to right traversal of a tree. Each copy of an edge

[1] An extended version of this paper, including pseudocode and figures, appears in [18].

contains a pointer to a copy of the next edge in the tour so we can go to the next edge without using recursion and visiting each edge of the tour only once.

For an ordered tree, $T = (V, E)$, we store an Euler tour as a set of items, C, where $|C| = 2|E|$. Each item represents an edge of the tour and stores information related to the tree, e.g., parent, child node names, and the order of the child among all its siblings. Additionally, it stores information related to the actual cycle of the Euler tour: (a) tag: a unique tag associated with this item, $0 \leq \mathsf{tag} < 2|E|$. This is used to locate and sort the items. (b) direction: up or down. This indicates which direction in the tree we are following. (c) next: tag of the next edge in the cycle.

We assume that $\mathsf{tag} = 0$ for the leftmost edge of the root of T. Suppose we shuffle the items in C using the tag field. Then a traversal of C is a simple scan of the memory and is data-oblivious revealing only the number of edges and nodes in the tree.

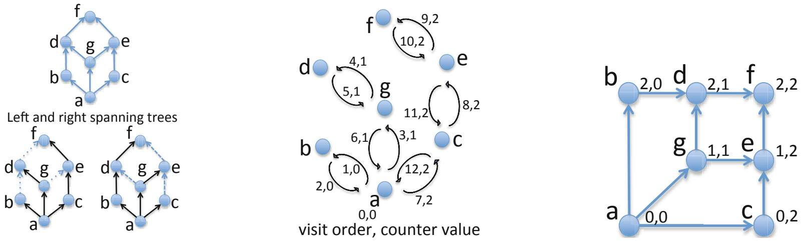
Computation over Euler Tour Representations. Many graph drawing algorithms collect information from a tree representation of the graph to determine the layout. We now show how one can use an Euler tour representation of a rooted tree to compute for each node of the tree, the size (number of nodes) of its subtree in a data-oblivious manner.

For this computation, we add a new field subsize for every edge in the Euler tour C. The algorithm maintains in local memory, W, a variable, total_subsize, initially set to 0. Edges in C are traversed as described in the previous section. However, every time we now read an edge, i, we update i.subsize with the value stored at total_subsize and write it back. When we are going up, i.e., $i.\mathsf{direction} = \mathsf{up}$, total_subsize is incremented by 1. Once the traversal finishes, we observe that for every two items, i and i', that represent a traversal of the same edge, i.e., $i.\mathsf{parent} = i'.\mathsf{parent}$, $i.\mathsf{child} = i'.\mathsf{child}$, $i.\mathsf{direction} = down$ and $i'.\mathsf{direction} = up$, the value $(i'.\mathsf{subsize} - i.\mathsf{subsize})$ is the size of the subtree rooted at i.child and the final value of total_subsize is subsize of the root. However, we need to associate nodes of the tree T with these values in the compressed-scanning model as well. For this purpose, we obliviously sort the values in C using the fields, parent and child, to bring items that correspond to the same edge next to each other. We then simply scan the resulting sorted list and after reading a pair of items, i and i', output a pair $(i.\mathsf{child}, i'.\mathsf{subsize} - i.\mathsf{subsize})$.

The above computation consists of two rounds: the first round reads one item of C at a time, modifies it and writes it back. The second round starts after the sorting is complete, where items are read one at a time and a new item is written to the output after every two reads. We can compute the depth of each node using a similar technique.

Drawing of Planar Acyclic Digraphs. We adopt an algorithm for dominance drawings of planar acyclic st-digraphs from [8]. To find the x-coordinate of each node, one builds a spanning tree based on leftmost incoming edges of the nodes and then traverses this tree from left to right, numbering each node in this order. The resulting numbering of each node is its x-coordinate. The algorithm to determine the y-coordinates uses the rightmost spanning tree.

We assume that the graph, G, is given as a set of edges, E, where $e \in E$ is an edge directed from node a to b storing indegree, the number of incoming edges to b, and child_num, the order of a among all incoming edges to b; the leftmost edge has order 0.



(a) (b) (c)

Fig. 1. (a) A planar acyclic st-digraph with its left and right spanning trees. (b) The order of the visit to each edge of Euler tour of the left spanning tree and the counter of x coordinate for child nodes, e.g., edge a-g is visited third and g is assigned x coordinate of 1. (c) The final drawing.

Following the original algorithm, we show how one can construct a spanning tree and number the nodes to get the final drawing. Our first task is to augment each edge with information about a spanning tree of G. We augment e with additional fields, left_spanning and right_spanning, which are set to true or false depending on which spanning tree e belongs to. In the compressed-scanning model, one simply accesses e, sets e.left_spanning to true if e.indegree equals e.child_num or e.right_spanning to true if e.child_num is 1, and writes e back.

Given annotated edges, we construct an Euler tour over each spanning tree. Note that given that the number of nodes in G is revealed, we do not need to hide the number of edges in either of the spanning trees. For ease of explanation, we say that we traverse an edge down when we follow an edge of the spanning tree in its direction in G. The left spanning tree is traversed starting with the leftmost outgoing edge of the root, and rightmost outgoing edge for the right tree. We are now ready to make a tour traversal and assign coordinates to the nodes. We adopt a compressed version of the algorithm that minimizes the area of the drawing and start with traversal of the left tree. In private memory, a counter for x-coordinates is maintained, set to 0. Initially, we output $(\text{source}, 0, x)$. For every edge e that has direction $=$ down, and e.indegree > 1 or e is the first traversed edge of a, we output $(e.b, \text{counter}, x)$. If e has down direction but is not the first edge of a traversed (in Euler tour this corresponds to remembering the latest visited edge) or is the only incoming edge to b, then we increment the counter by 1 and output $(e.b, \text{counter}, x)$. If e.direction is set to up, then we output $(\text{dummy}, 0, x)$. The algorithm for computing y-coordinates is similar and outputs values with $(e.\text{parent}, \text{counter}, y)$. Note that access pattern of reads and writes is always the same: read an edge of the Euler tour and output a tuple of three values.

The output of the above procedure contains tuples of real and dummy values. We can remove dummy values and bring x, y coordinates of each node together by obliviously sorting tuples by the first field (node name) such that string dummy is always greater than any real node name. The resulting list contains all dummy tuples at the end. Also, each node has its x- and y-coordinates adjacent. See Figure 1 for an example.

Treemap Drawings. Treemaps are a representation designed for human visualization of complex tree structures, where arbitrary trees are shown with a 2-d space-filling area. Here, we present how one can draw a treemap using an algorithm from [20] adapted to the compressed-scanning model. The original algorithm takes a rectangle area and splits it vertically into two sections. The area of the first section is enough to fit the first child, child_1, of the root and the rest is enough to fit the rest of its children. The next step is to divide the first section among children of child_1 but this time splitting the area horizontally. The algorithm continues in the same manner for all decedents of child_1. Once finished, it proceeds to splitting the second section between second child of the root, child_2, and the rest of root's children.

Input: A tree, T, where each leaf also contains a value area and the size of a rectangle area, $w \times h$, where T should be drawn. We build an Euler tour, C, from T and add two fields parent_area and child_area to each edge in C.

Output: Each node is labeled with (x, y) coordinates of the top-left corner, P, and bottom-right corner, Q, of the rectangle area where the node should be placed in.

Data-oblivious algorithm: We first run a procedure similar to the one for computing subsize, to assign area values to inner nodes of T. The original algorithm labels the nodes with values P and Q via pre-order traversal of T. The algorithm we propose here first goes down the leftmost subtree computing values P, Q and labeling the nodes on the way. In private memory, it maintains only one copy of the last two assigned values of P and Q, prevP and prevQ. It then goes up the tree "undoing" all the computations made to prevP and prevQ. We do it in such a way that when going up and reaching some node, we recover its P and Q values as they were before we visited any of its children or other nodes in its subgraph. This algorithm fits the traversal of Euler tour C of the tree T. When going down the tree, we read each item i of tour C and output P, Q values corresponding to i.child. However, when going up we cannot retrieve earlier written P, Q values, since this will not be data-oblivious and we reveal that we are going up, which consequently reveals the depth of the tree. This is where "undoing" computations when going up on prevP and prevQ helps. This is possible since the information used to compute P and Q is stored twice in C: once for edge with direction set to down and once for up. Figure 2 shows an execution of the algorithm on a small tree.

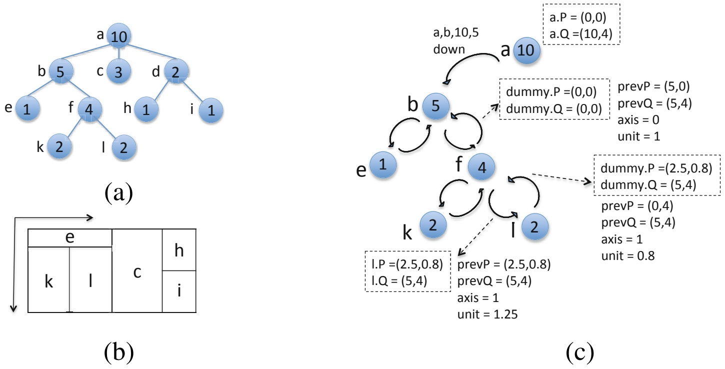


Fig. 2. Treemap graph drawing. (a) The original graph. (b) The final drawing. (c) Execution of an oblivious treemap drawing algorithm on the graph in (a) on a 10×4 rectangle area. The values in dashed rectangles are written for every edge and are never accessed. Variables prevP, prevQ, axis and unit are kept in memory.

Series-Parallel Graphs. A series-parallel (SP) graph is a directed acyclic graph that can be decomposed recursively into a combination of series-parallel digraphs. The base case of such a graph is a simple directed edge. A series composition consists of two series-parallel graphs G_1 and G_2 where the sink of G_1 is identified with the source of G_2. A parallel composition of two series-parallel graphs G_1 and G_2 is the digraph where source of G_1 is identified with the source of G_2 and similar for their sink nodes. For example, consider the series-parallel digraph shown in Figure 3a.

An SP graph G can be represented with a binary tree (SPQ tree) with three types of nodes, S, P and Q. Q nodes are leaves of the tree and correspond to individual edges of G. An internal node is of type P if it is a parallel composition of the children digraphs. If a node corresponds to a series composition it is called S node. Here, we use a right-pushed embedding of G such that a transitive edge in parallel composition is always embedded on the right. (Figure 3b shows the SPQ tree of the graph of Figure 3a.)

Input: SPQ tree from a right-pushed embedding of SP digraph G and nodes that are annotated as S, P or Q. We convert this tree into an Euler tour with addition of parent and child node type: parent_spq_type and child_spq_type which are either S, P or Q.

Data-oblivious algorithm: We adopt the Δ-drawing algorithm from [3]. The algorithm makes several computations over the tree to annotate the nodes of the SPQ tree with values b, b' and (x, y). Here b is the size of the bounding triangle of a node and b' stores a distance between parallel drawings to make sure they do not intersect (see [18]). Value b can easily be computed in the same manner as we computed the subgraph size earlier in this section. Value b' of the left child is added only for parents of P nodes. When an Euler tour is going up the tree we can always check the value of parent_spq_type to know if b' of the left subgraph should be carried to the right one. Coordinates (x, y) for each node are computed from a small modification of the Euler tour: the left child needs know value $b(\triangle(G_2))$ and right child needs to know $b'(\triangle(G_1))$. It is easy to do this by always reading the next edge and remembering the last edge.

Given that we know the coordinates of each triangle, we can now assign coordinates for individual nodes. Recall that every leaf node of SPQ tree is associated with an edge while an internal node is either a DAG or a path of edges in the subtree rooted at this node. Hence, we can associate each internal node of SPQ tree, and edges in the corresponding Euler tour, with two nodes of the series-parallel graph that correspond to the source and the sink of the underlying subgraphs. Given a parent node of SPQ tree and source and sink nodes of its children, c_1 and c_2, if c_1^{sink} and c_2^{source} are equal then node c_1^{sink} is placed at $(c_1.x, c_1.y + c_1.b)$. Otherwise, we output a dummy.

Drawing Trees with Bounding Rectangles. In this section, we present an algorithm that draws a binary tree T using a bounding rectangle approach from [6], adapted to the compress-scanning model. This algorithm is slightly different from the approaches we took in previous algorithms and involves a more complex way of converting it to fit data-oblivious mode. The original algorithm recursively assigns bounding rectangles to nodes of the tree. A leaf node is assigned a rectangle of size 2×1, while an internal node is assigned a rectangle that fits the bounding rectangles of its children. Each rectangle is represented by parameters width, height, and refpoint (left top corner). For a leaf, width $= 2$. For an internal node, width is the sum of the widths of its

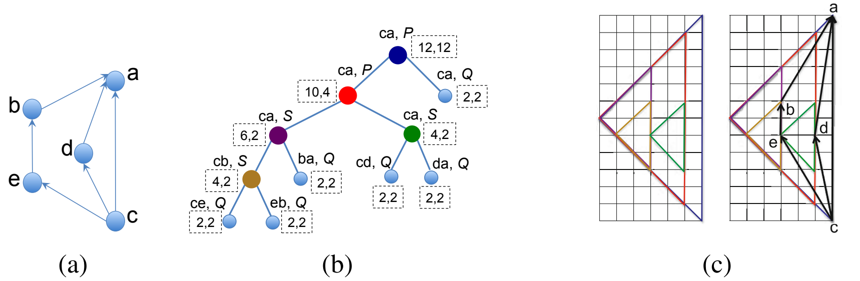


Fig. 3. (a) A series-parallel graph. (b) SPQ tree representation annotated with values b and b' (dashed rectangles). (c) The final drawing.

children. The height of the rectangle is defined as $1 + \max_i \mathsf{child}_i.\mathsf{height}$. The bounding rectangle of the root has refpoint $= (0, \mathsf{tree\_height})$. The refpoint of the ith child of node p is $(p.\mathsf{refpoint}.x + \sum_{j<i} \mathsf{child}_j.\mathsf{width}, p.\mathsf{height})$. A leaf node l is placed at point $(x, y) = (l.\mathsf{refpoint}.x + l.\mathsf{width}/2, l.\mathsf{refpoint}.y)$. An internal node is placed between its children, hence, a node l with children child_i $(i = 1, 2)$ is placed at point $(\sum \mathsf{child}_i.x/2, l.\mathsf{refpoint}.y)$.

Data-oblivious algorithm: An Euler tour over T allows to compute the width, level and refpoint values of the nodes. Computing the coordinates of an internal node requires knowing the coordinates of its children, and thus can be done only after the subtrees of both children are processed. If we use an Euler tour traversal, we need to store the coordinates of the points in the left subtree while processing the right subtree. We cannot store these coordinates in the private workspace since in the worst case their number is linear in the size of the tree. Indeed, in our previously described methods, we only store a constant number of values when traversing a tour. Therefore, in this section we propose a different technique that is based on a dashed-solid representation. This representation allows us to store only $O(\log n)$ coordinates in the worst case, which fits our compressed-scanning model.

In the *dashed-solid* representation of a tree [22], an edge parent-child_i is said to be solid if $\mathsf{parent.subsize}/2 < \mathsf{child}_i.\mathsf{subsize}$ and dashed otherwise. If the children have the same subsize, the right edge is solid and the left one is dashed. Thus, a parent node has a solid edge to only one of its children whose subtree has size equal or larger than that of the sibling. The main property of the dashed-solid assignment is that any path of the tree has $O(\log n)$ dashed edges. The dashed-solid representation can be computed from the subsize values using another Euler tour traversal.

Given a dashed-solid representation, we compute the (x, y) coordinates by creating a tour around the tree where edges are accessed in a specific order. First, we go down the path of solid edges starting at the root. When a leaf is reached, we go back up until a node with a dashed edge is reached. We then recursively traverse the subtree connected to the dashed edge. To construct this traversal one needs to store with every node which one of its children is solid. The coordinates are computed as follows. We follow a solid edge path until a leaf l is reached and then the leaf node is assigned to coordinates $(l.\mathsf{refpoint}.x + l.\mathsf{width}/2, 0)$. We store these coordinates in variable s in private memory. When going up, if the parent node p does not have any other children, then we assign it to $(s.x, s.y)$ and continue traversing up the tree. If instead node p has a dashed edge to child c, then we recursively traverse the subtree of c, which results in the computation of the coordinates of c, denoted d. Once this traversal is finished, node p is assigned to coordinates $f = ((s.x + d.x)/2, 1 + \max(s.y, d.y))$. We now set $s = f$ and keep going up the solid path. Note that in the recursive traversal of the subtree of node c, we will store additional coordinates in private memory. Since a root-to-leaf path in the tree has no more than $\log n$ dashed edges, a private workspace of size $O(\log n)$ is enough to store all the coordinates needed by the traversal. We note that this algorithm can be extended to arbitrary trees if we represent the dashed edges of a node as a balanced binary tree (see [6] for details).

Summary. We have given drawing algorithms in the compressed-scanning model that consist of a constant number of Euler tours. In Section 3, we have shown that an Euler tour can be implemented with a single-round compressed scan, where, from the server's perspective, the items associated with the edges of the tour are accessed sequentially. Thus, the following theorem is a consequence of Theorem 1.

Theorem 2. *The drawing algorithms described in this section are data-oblivious according to Definition 1 and run in time $O(\mathsf{sort}_B(n))$ where n is the size of the input graph/tree and B is the oblivious sorting algorithm used between the rounds. Also, the private workspace has size $O(\log n + \mathsf{workspace}_B(n))$ for the bounding-rectangle tree-drawing algorithm and has size $O(\mathsf{workspace}_B(n))$ for the other algorithms.*

Our algorithms hide the combinatorial structure and layout of the graphs, while the number of edges and vertices is revealed. One can achieve even stronger privacy if dummy edges and nodes are added.

4 Conclusions and Open Problems

We introduce the compressed-scanning technique for designing data-oblivious algorithms in a cloud-computing environment. We show how to use this technique to develop data-oblivious variations of several classic graph drawing algorithms. Open problems include finding other applications of this technique and developing alternative data-oblivious approaches for graph drawing. For example, it is not known how to compute in a data-oblivious way st orientations and st-numberings, used for visibility representations of planar graphs [23], or canonical orderings [11], used for planar straight-line drawings.

References

1. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proc. Symp. on Principles of Database Systems, pp. 1–16 (2002)
2. Barghouti, N., Mocenigo, J., Lee, W.: Grappa: A GRAPh PAckage in Java. In: Di Battista, G. (ed.) GD 1997. LNCS, vol. 1353, pp. 336–343. Springer, Heidelberg (1997)
3. Bertolazzi, P., Cohen, R.F., Di Battista, G., Tamassia, R., Tollis, I.G.: How to draw a series-parallel digraph. Internat. J. Comput. Geom. Appl. 4, 385–402 (1994)

4. Binucci, C., Brandes, U., Di Battista, G., Didimo, W., Gaertler, M., Palladino, P., Patrignani, M., Symvonis, A., Zweig, K.: Drawing Trees in a Streaming Model. In: Eppstein, D., Gansner, E.R. (eds.) GD 2009. LNCS, vol. 5849, pp. 292–303. Springer, Heidelberg (2010)
5. Bridgeman, S., Garg, A., Tamassia, R.: A graph drawing and translation service on the World Wide Web. Int. J. Comp. Geom. Appl. 9(4-5), 419–446 (1999)
6. Cohen, R.F., Di Battista, G., Tamassia, R., Tollis, I.G.: Dynamic graph drawings: Trees, series-parallel digraphs, and planar ST-digraphs. SIAM J. Comput. 24(5), 970–1001 (1995)
7. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing. Prentice Hall, Upper Saddle River (1999)
8. Di Battista, G., Tamassia, R., Tollis, I.G.: Area requirement and symmetry display of planar upward drawings. Discrete Comput. Geom. 7(4), 381–401 (1992)
9. Eppstein, D., Goodrich, M.T., Tamassia, R.: Privacy-preserving data-oblivious geometric algorithms for geographic data. In: 18th ACM Adv. in Geographic Information Systems, ACM GIS, pp. 13–22 (2010), http://doi.acm.org/10.1145/1869790.1869796
10. Feldman, J., Muthukrishnan, S., Sidiropoulos, A., Stein, C., Svitkina, Z.: On distributing symmetric streaming computations. ACM Trans. Algorithms 6(4), 66:1–66:19 (2010), http://doi.acm.org/10.1145/1824777.1824786
11. de Fraysseix, H., Pach, J., Pollack, R.: How to draw a planar graph on a grid. Combinatorica 10(1), 41–51 (1990)
12. Goldreich, O.: Foundations of Cryptography, vol. II. Cambridge University Press (2004)
13. Goodrich, M.T.: Randomized Shellsort: A simple oblivious sorting algorithm. In: Symposium on Discrete Algorithms, SODA, pp. 1–16 (2010)
14. Goodrich, M.T., Mitzenmacher, M.: Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011)
15. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious RAM simulation with efficient worst-case access overhead. In: Proc. ACM Workshop on Cloud Computing Security, CCSW, pp. 95–100 (2011)
16. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Practical oblivious storage. In: Proc. ACM Conference on Data and Application Security and Privacy, CODASPY (2012)
17. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless oblivious RAM simulation. In: Proc. ACM-SIAM Symp. on Discrete Algorithms, SODA (2012)
18. Goodrich, M.T., Ohrimenko, O., Tamassia, R.: Data-oblivious graph drawing model and algorithms. CoRR abs/1209.0756 (2012)
19. Henzinger, M.R., Raghavan, P., Rajagopalan, S.: Computing on data streams. In: External Memory Algorithms. Discrete Mathematics and Theoretical Computer Science, vol. 50, pp. 107–118. AMS (1999)
20. Johnson, B., Shneiderman, B.: Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In: IEEE Visualization, pp. 284–291 (1991)
21. Muthukrishnan, S.: Data Streams: Algorithms and Applications. In: Foundations and Trends in Theoretical Computer Science, vol. 1. Now Publishers (2005)
22. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. Journal of Computer and System Sciences 26(3), 362–381 (1983)
23. Tamassia, R., Tollis, I.G.: A unified approach to visibility representations of planar graphs. Discrete Comput. Geom. 1(4), 321–341 (1986)