Longest Common Subsequence as Private Search [*]

Mark Gondree[1] and Payman Mohassel[2]

[1]Department of Computer Science, Naval Postgraduate School
[2]Department of Computer Science, The University of Calgary

Abstract

At STOC 2006 and CRYPTO 2007, Beimel *et al.* introduced a set of privacy requirements for algorithms that solve search problems. In this paper, we consider the longest common subsequence (LCS) problem as a private search problem, where the task is to find a string of (or embedding corresponding to) an LCS. We show that deterministic selection strategies do not meet the privacy guarantees considered for private search problems and, in fact, may "leak" an amount of information proportional to the entire input.

We then put forth and investigate several privacy structures for the LCS problem and design new and efficient output sampling and equivalence protecting algorithms that provably meet the corresponding privacy notions. Along the way, we also provide output sampling and equivalence protecting algorithms for finite regular languages, which may be of independent interest.

1 Introduction

The sensitivity of patient data stored in large genomic databases makes their widespread use in research problematic. Working with this data while protecting a patient's privacy is recognized as a major challenge for the biomedical research community [8, 39, 42]. Rigorous definitions for privacy in this area, however, are still in development. In this paper, we investigate a problem of significant importance to genomic computation: the longest common subsequence (LCS) problem. We propose that the privacy notions from *private search* give a type of functional security, offering a strong definition for patient privacy while allowing specific questions posed by researchers to be answered. We show that these privacy notions can be realized for the LCS problem, efficiently and with perfect privacy.

To date, research in secure genomic computation has considered variants of the string alignment and LCS problems where the desired output is the edit-distance or the length of a subsequence [9, 21, 30, 41]. The alignment or subsequence itself, however, is often of equal or greater interest to genomic research. In these scenarios, the problem is no longer a function, for example there may be many longest common subsequences for any pair of strings (in fact, the number of solutions may be exponential in the length of the input). To remedy this, we must first fix some strategy to select an output. There is a growing body of literature — securely solving distributed constraint satisfaction problems [33, 36, 37, 45], combinatorial auction optimization [40, 44], selecting a stable bipartite

[*]This is the full version of an article [24] to appear in *WPES'09*.

matching [20], selecting an optimal k-means clustering [17], generating small decision trees [31] — where the selection strategy chosen is arbitrary, heuristic, or simply the one terminating earliest. However, the choice of output may serve as a kind of "covert channel" that leaks information about the inputs (unwittingly or maliciously). For tasks in genomic computation, this can mean an accidental compromise of patient privacy.

We design algorithms to perform this selection for the LCS problem, provably meeting the security definitions introduced for private search. This is not always possible. Firstly, private search algorithms are unlikely to exist for many problems, including approximations of 3SAT [11], k-means clustering and vertex cover [12]. Secondly, there are problems for which finding a solution is easy yet uniformly sampling a solution seems hard, *e.g.* stable bipartite matching [15]. Lastly, it is not always possible to re-use private search algorithms for related problems because the reduction typically distorts the solution space. The private search algorithm for shortest path [13], for example, cannot be used for LCS because all known reductions from LCS to shortest path have the property that multiple paths correspond to the same subsequence; thus, a random path does not correspond to a random subsequence.

We define and investigate privacy structures for two important variants of the LCS problem. For the first, a solution is some longest common subsequence string. For the latter, a solution is some "embedding," *i.e.* structural data showing how the subsequence was encoded in the original input. While these two variants are used interchangeably in the LCS literature and are often computed using the same set of techniques, we demonstrate that they have very different privacy requirements. We provide output sampling algorithms and equivalence protecting algorithms for both these variants. Additionally, we design private search algorithms for selecting a word in the language of some deterministic finite automaton, which may be of use in private search beyond our specific application (searching lexicons, searching a trie of game-playing strategies, pattern-recognition, etc).

Our main contribution is to design private search algorithms which are *efficient*. Towards this goal, we provide an efficient, generic technique to convert a dynamic programming algorithm into an output sampling algorithm. Unlike constructions from previous work, which perform a weighted selection by repeatedly counting the solution space, we use the dynamic programming paradigm to count the solution spaces of all possible sub-problems at the same cost of counting solutions to the original instance (and avoid counting repeatedly). We also describe and use an efficient reduction from the LCS string problem to the problem of words in the language of a DFA, where the reduction has the important property of preserving the structure of the original solution space (sometimes referred to as a parsimonious reduction).

In contrast, the literature on private search algorithms [11, 12] is overwhelmingly concerned with the existence or non-existence of polynomial-time algorithms for various problems, and less concerned with the efficiency of those algorithms. Our work can be seen as the beginning of a line of research on the efficiency of private search, and a new line of research for the literature on LCS efficiency.

Motivation Performing different computational tasks on large biological databases is becoming a more common practice in both public and private institutions. The FBI maintains a database of over four million DNA profiles of criminal offenders, crime scene evidence, and missing persons in its CODIS system [3], and uses the data for forensic studies and DNA-based identification. The United Kingdom's UK Biobank [7] and Quebec's CARTaGENE [2] projects each plan to collect genetic samples from 1 percent of their respective populations. deCODE Genetics [4], a biopharmaceutical company which studies genomic data for drug discovery and development, has collected the genotypic and medical data of over 50 percent of the population in Iceland. Similar endeavors seek to make these types of databases available for scientific study [6].

The genomic data stored in these databases may be extremely sensitive: an individual's DNA sequence reveals a great deal of information regarding that individual's health, background, and physical appearance [1, 5]. It has been shown that a sequence can be linked to the corresponding individual simply by recognizing the presence of certain markers [32]. In the United States, HIPAA's Privacy Rule [34] mandates that a patient's identity must be protected when their data (including genomic data) is shared; failure to assure this may result in legal action, fines, revocation of government funding, and imprisonment. The privacy notions considered in this paper may help establish the groundwork for investigators, clinical researchers, and institutional review boards when designing projects, defining rules for disclosure, and obtaining informed consent from participants.

2 Preliminaries

Let A and B be two strings over a finite alphabet Σ of size σ, with lengths $m = |A|$ and $n = |B|$ (without loss of generality, let $m \leq n$). A longest common subsequence (LCS) of A and B is a subsequence of both A and B such that no other common subsequence has greater length. To help us formally define those variants of the LCS problem that have received attention in the literature, we introduce the following notation.

For any $\ell \in \mathbb{Z}^+$, let the value function $v : \Sigma^\ell \times \{0,1\}^\ell \to \Sigma^*$ be defined such that for all $S \in \Sigma^{\ell-1}, \gamma \in \{0,1\}^{\ell-1}, b \in \{0,1\}, a \in \Sigma$ we have

$$v(S||a, \gamma||b) = \begin{cases} v(S,\gamma) & \text{if } b = 0 \\ v(S,\gamma)||a & \text{if } b = 1 \end{cases}$$

and let $v(\varepsilon, \varepsilon) = \varepsilon$.

Definition 1 (LCS Embeddings)**.** *Let the relation*
LCSe(A, B, α, β) *be true if* $\alpha \in \{0,1\}^m, \beta \in \{0,1\}^n$ *where* $v(A, \alpha) = v(B, \beta) = x$ *and there is no* $\hat{\alpha} \in \{0,1\}^m, \hat{\beta} \in \{0,1\}^n$ *where* $v(A, \hat{\alpha}) = v(B, \hat{\beta}) = \hat{x}$ *such that* $|\hat{x}| > |x|$.

If LCSe(A, B, α, β) is true, we call (α, β) an *embedding* of an LCS of A and B. An embedding is essentially a witness that $v(A, \alpha)$ is a valid subsequence of A. In some situations, however, we are only interested in the subsequence itself.

Definition 2 (LCS String)**.** *Let the relation*
LCS(A, B, x) *be true if there exists some* $\alpha \in \{0,1\}^m, \beta \in \{0,1\}^n$ *such that* *LCSe*(A, B, α, β) *and* $v(A, \alpha) = v(B, \beta) = x$.

Algorithms in the literature that solve the longest common subsequence problem return one or more of the following outputs: (i) the length $|x|$, where LCS(A, B, x); (ii) a string x, where LCS(A, B, x); (iii) an embedding (α, β) of an LCS, where LCSe(A, B, α, β). The literature has typically not differentiated strongly between algorithms which recover strings and those which recover embeddings, but we will show later that these problems have different security requirements.

The following dynamic programming algorithm solving the longest common subsequence problem was independently discovered by many researchers (in both computer science and biology). Let L be the

$(m+1) \times (n+1)$ matrix whose entries can be computed (row-by-row or column-by-column) using the following:

$$L[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ L[i-1,j-1]+1 & \text{if } A[i] = B[j] \\ \max(L[i-1,j], L[i,j-1]) & \text{otherwise} \end{cases}$$

Entry $L[m,n]$ holds the length of the LCS for A and B. When we consider the variant of the LCS problem which outputs only the length of the LCS, then the LCS problem is a function. However, when we consider the variants which output embeddings or strings (or both), then the problem is no longer a function and we must consider how to perform the selection. When selecting an LCS embedding, for example, the number of possible solutions is bounded by the following.

Claim 1 (Bound on Number of Embeddings [25])**.** *Let $E(n,m)$ be the maximum number of embeddings of an LCS into strings of length n and $m \leq n$. Then $E(n,m) \leq \frac{\phi^2\sqrt{5}}{2\pi}\phi^{2n} \approx .932(2.62)^n/n$, where ϕ is the golden ratio. As a consequence, $\lceil \log E(n,m) \rceil < 1.4n$.*

Note that because each string must have an embedding, $E(n,m)$ also provides an upper bound on the number of possible solutions when selecting an LCS string. There exist efficient dynamic programming algorithms to count the exact number of LCS embeddings or LCS strings for a particular problem instance (see Appendix B for such algorithms). Each counting algorithm requires $O(mn \log E(m,n))$ time and space. We use counting algorithms like these as the basis of our output sampling algorithms, later.

Typically, simple deterministic algorithms that "backtrack" through the LCS dynamic programming table are used to recover the string and/or embedding of an LCS for A and B. In the next section we discuss new, strong privacy requirements for the selection mechanism that motivate the use of alternate methods for backtracking.

Our algorithms satisfying these stronger security requirements require the ability to efficiently sample from specific discrete distributions. Given a finite set of variables $V = \{v_0, v_1, \ldots, v_\ell\}$ and their frequencies $\{a_0, a_1, \ldots, a_\ell\}$, a weighted coin toss selects a variable $v_i \in V$ with probability a_i/b where $b = \sum_{j \in [0,\ell]} a_j$. There are many procedures for flipping a weighted coin. One very simple procedure is the following: select an appropriate number of random bits and let z be the integer they represent; if $z > b$, then we fail; else, we output v_i where i is the smallest integer such that $z \leq \sum_{j \in [0,i]} a_j$. This procedure fails with probability $(2^{\lfloor \log b \rfloor} - 1)/2^{\lceil \log b \rceil} < 1/2$. The procedure uses $g_{wct}(\ell, b) := O(\ell \log b)$ time and space. Additionally, notice that we can perform N weighted coin flips using $kN \log N \lceil \log b \rceil$ random bits, where the probability of succeeding in all N coin flips is at least $1 - 1/2^k$.

3 LCS as a Search Problem

In this section, we define search problems, give two natural variants of the longest common subsequence problem formulated as search problems, and give some preliminary observations on their relationships.

Definition 3 (Privacy Structure [13])**.** *A search problem is an ensemble $\mathcal{P} = \{\mathcal{P}_n\}_{n \in \mathbb{N}}$ such that $\mathcal{P}_n : \{0,1\}^n \to 2^{\{0,1\}^{q(n)}}$ for some positive polynomial $q(n)$. For a search problem $\mathcal{P}$, the privacy structure $\equiv_{\mathcal{P}}$ is an equivalence relation on instances $x, y \in \{0,1\}^n$ such that $x \equiv_{\mathcal{P}} y$ iff $\mathcal{P}_n(x) = \mathcal{P}_n(y)$.*

Following Halevi *et al.* [27] and Beimel *et al.* [11], we say that an algorithm *leaks* at most k bits if it *refines* each equivalence class by dividing it to at most 2^k sub-classes.

Given two search problems $\mathcal{P}, \mathcal{R}$ defined over the same input space, we call their privacy structures *incomparable* if neither is a strict refinement of the other. In other words, if $\equiv_{\mathcal{P}}$ refines $\equiv_{\mathcal{R}}$ and $\equiv_{\mathcal{R}}$ refines $\equiv_{\mathcal{P}}$, then $\equiv_{\mathcal{P}}$ and $\equiv_{\mathcal{R}}$ are incomparable.

Definition 4 (LCS Embeddings). *Define LCS-e as the search problem for LCS embeddings, where for all $A \in \Sigma^m, B \in \Sigma^n$, we have*

$$\textit{LCS-e}(A,B) = \{(\alpha,\beta) : \mathrm{LCSe}(A,B,\alpha,\beta)\}$$

Let $\equiv_{\textit{LCS-e}}$ be the related privacy structure, such that if $\textit{LCS-e}(A,B) = \textit{LCS-e}(A',B')$ we say $(A,B) \equiv_{\textit{LCS-e}} (A',B')$. For example, $(atg, aga) \equiv_{\textit{LCS-e}} (agc, act)$.

Definition 5 (LCS Strings). *Define LCS-s as the search problem for LCS strings, where for all $A \in \Sigma^m, B \in \Sigma^n$, we have*

$$\textit{LCS-s}(A,B) = \{x : \mathrm{LCS}(A,B,x)\}$$

Let $\equiv_{\textit{LCS-s}}$ be the related privacy structure, such that if $\textit{LCS-s}(A,B) = \textit{LCS-s}(A',B')$ we say $(A,B) \equiv_{\textit{LCS-s}} (A',B')$. For example, $(atg, aga) \equiv_{\textit{LCS-s}} (cag, atg)$.

A few observations follow immediately from these definitions. First, privacy structures for LCS-s and LCS-e are incomparable. Second, there are inputs of length n for which natural deterministic selection strategies leak $\Theta(n)$ bits with respect to these privacy structures. Together, these observations motivate us to look at these variants as separate and independent problems, and to search for clever non-deterministic strategies which provably protect their privacy structures. See Appendix C for details on these observations.

4 LCS Output Sampling

In this section, we give algorithms for our LCS variants which protect their respective privacy structures on a single query, known as output sampling algorithms. An *output sampling algorithm* is a randomized algorithm whose outputs (i) are correct answers to the search problem, and (ii) form a distribution which is indistinguishable from the uniform distribution on answers to the search problem.

Definition 6 (Output Sampling [11, 13]). *Let $\mathcal{P} = \{\mathcal{P}_n\}_{n\in\mathbb{N}}$ be a search problem. An algorithm $\mathcal{A}$ is called an* output sampling algorithm *for $\mathcal{P}$ if (i) $\mathcal{A}$ is a deterministic polynomial time algorithm taking two inputs x, s_n where $|x| = n$ and $|s_n| = p(n)$ for some polynomial p; and (ii) for every string $x \in \{0,1\}^n$ the distribution $\mathcal{A}(x, s_n)$ is computationally indistinguishable from the uniform distribution on $\mathcal{P}(x)$.*

Our output sampling algorithms are based on efficient dynamic programming solutions for counting the number of outputs for each search problem. Below, we design an output sampling algorithm for the LCS-e search problem on inputs A and B. We use the LCS-e counting algorithm (see Appendix B) to compute the D matrix used by the algorithm. We assume that the L matrix is computed following the typical dynamic programming LCS algorithm.

Algorithm 1 Output sampling algorithm for LCS-e

```
1. $\alpha, \beta, s \leftarrow \epsilon$; $i \leftarrow m$; $j \leftarrow n$; $forceUp, forceLeft \leftarrow false$
2. while $i > 0$ and $j > 0$ do
3.    $W_1, W_2, W_3, W_4 \leftarrow 0$
4.    if $A[i] = B[j]$ then $W_1 \leftarrow D[i-1, j-1]$
5.    if $L[i-1, j-1] = L[i, j]$ then $W_2 \leftarrow D[i-1, j-1]$
6.    if $L[i-1, j] = L[i, j]$ then $W_3 \leftarrow D[i-1, j] - W_2$
7.    if $L[i, j-1] = L[i, j]$ then $W_4 \leftarrow D[i, j-1] - W_2$
8.    if $forceUp$ then sample $z \in \{1, 3\}$ where
         $\Pr[z = i] = W_i/(W_1 + W_3)$
9.    else if $forceLeft$ then sample $z \in \{1, 4\}$ where
         $\Pr[z = i] = W_i/(W_1 + W_4)$
10.   else sample $z \in \{1, 2, 3, 4\}$ where
         $\Pr[z = i] = W_i/(W_1 + W_2 + W_3 + W_4)$
11.   if $z = 1$ then
12.      $\alpha \leftarrow 1||\alpha$; $\beta \leftarrow 1||\beta$; $s \leftarrow A[i]||s$
13.      $forceUp, forceLeft \leftarrow false$
14.      $i \leftarrow i - 1,\ j \leftarrow j - 1$
15.   else if $z = 2$ then
16.      $\alpha \leftarrow 0||\alpha$; $\beta \leftarrow 0||\beta$
17.      $i \leftarrow i - 1,\ j \leftarrow j - 1$
18.   else if $z = 3$ then
19.      $\alpha \leftarrow 0||\alpha$; $forceUp \leftarrow true$
20.      $i \leftarrow i - 1$
21.   else if $z = 4$ then
22.      $\beta \leftarrow 0||\beta$; $forceLeft \leftarrow true$
23.      $j \leftarrow j - 1$
24. end while
25. return $(\alpha, \beta)$
```

With a few non-trivial modifications, the algorithm below can be transformed into an output sampling algorithm for the LCS-s search problem (see Appendix D).

Claim 2. *Algorithm 1 is an output sampling algorithm for the LCS-e search problem and uses $O(n \log E(m, n))$ time and $O(|D|)$ space.*

Proof. At each step of the backtracking algorithm, there are several options to choose from. In order to generate a uniformly random embedding, our algorithm needs to satisfy the following two properties. (i) At each step of the backtracking, every two possible backtracking options should lead to two disjoint sets of LCS embeddings. We call this the *non-overlapping* property. (ii) At each step, a backtracking option is chosen randomly, according to a distribution that is weighted proportional to the number of solutions that each option affords. We call this the *correctly-weighted* property.

We show that our algorithm has both of the above properties. Given that this is the case, based on a simple lemma which we introduce next, it is easy to see that our algorithm generates an LCS embedding uniformly at random.

Consider a tree T rooted at node r. The following simple algorithm shows how to traverse the tree and output one leaf of the tree uniformly at random.

1. Let $StartNode \leftarrow r$.

2. End if $StartNode$ does not have any children. Otherwise, randomly choose one of $StartNode$'s children, according to the distribution that is weighted by the number of that child's desendents.

3. Let $StartNode$ be equal to the chosen child. Go back to the first step.

Lemma 1. *Consider a tree T rooted at node r. The algorithm given above outputs a leaf of the tree* uniformly at random*.*

In our algorithm each pair (i, j) corresponds to an internal node in the tree, and each LCS embedding between strings A and B corresponds to a leaf. In each iteration of the while loop, we partition the set of all possible LCS embeddings between $A[1 \dots i]$ and $B[1 \dots j]$ into four disjoint groups: (i) embeddings that use $A[i]$ and $B[j]$; (ii) embeddings that use neither $A[i]$ nor $B[j]$; (3) embeddings that use $B[j]$ but not $A[i]$; (4) embeddings that use $A[i]$ but not $B[j]$. If *forceUp* is set, it means that $B[j]$ needs to be part of the embedding and, therefore, backtracking options that do not use $B[j]$ are not allowed.

Similarly, if $forceLeft$ is set, $A[i]$ needs to be part of the embedding and backtracking options that do not use $A[i]$ should be ignored. This procedure guarantees that we meet the *non-overlapping* property at each step.

Furthermore, the number of embeddings for each of the above four backtracking options is computed and stored in $W_1, \dots, W_4$, respectively. The W_i values are used to create the distribution for each coin toss and therefore satisfy the second property. This concludes the correctness argument.

The main loop iterates at most $m + n$ times. In each iteration we perform a single weighted coin toss, requiring $g_{wct}(4, E(n, m)) = O(\log E(n, m))$ time and space. Thus, the algorithm uses $O(|L| + |D| + (n + m) \log E(n, m)) = O(|D|)$ space and $O((n + m) \log E(n, m)) = O(n \log E(n, m))$ time. $\square$

Output sampling algorithms may be appropriate for some applications and useful as independent constructions. Notice, however, that multiple queries to an output sampling algorithm allow one to learn many outputs, possibly. Next, we design algorithms which meet a stronger notion of privacy, respecting the privacy structure while restricting what is learned across multiple queries.

5 LCS Equivalence Protecting

In this section, we give algorithms for our LCS variants which protect their respective privacy structures across multiple queries, known as equivalence protecting algorithms. An *equivalence protecting algorithm* cannot be efficiently distinguished from a randomly selected oracle that (i) provides correct answers to the search problem, and (ii) gives the same output for all inputs in the same equivalence class with respect to the problem's privacy structure.

Definition 7 (Equivalence Protecting [13]). *Let*
$\mathcal{P} = \{\mathcal{P}_n\}_{n \in \mathbb{N}}$ be a search problem and p be the polynomial such that $\mathcal{P}_n : \{0,1\}^n \to 2^{\{0,1\}^{p(n)}}$. We say that for any $n \in \mathbb{N}$, an oracle $O_n : \{0,1\}^n \to \{0,1\}^{p(n)}$ is private with respect to the privacy structure $\equiv_{\mathcal{P}}$ (alternately, we say it is private with respect to $\mathcal{P}$ or protects the equivalence relation $\equiv_{\mathcal{P}}$) if (i) for every $x \in \{0,1\}^n$ it holds that $O_n(x) \in \mathcal{P}_n(x)$; and, (ii) for every $x, y \in \{0,1\}^n$ it holds that $x \equiv_{\mathcal{P}} y$

implies $O_n(x) = O_n(y)$. An algorithm $\mathcal{A}(\cdot,\cdot)$ is an equivalence protecting algorithm *for $\equiv_{\mathcal{P}}$, if for every polynomial time oracle machine D, for every polynomial q, and for any sufficiently large n*

$$|Pr[D^{O_n}(1^n) = 1] - Pr[D^{\mathcal{A}(\cdot,s_n)}(1^n) = 1]| < 1/q(n)$$

where the first probability is over the uniform distribution over oracles O_n that are private with respect to $\mathcal{P}$, and the second probability is uniform over the choices of the seed s_n for the algorithm $\mathcal{A}$.

In summary, an equivalence protecting algorithm provides solutions that look random and behaves consistently on equivalent inputs (and therefore does not leak further information when executed repeatedly on equivalent inputs).

5.1 Equivalence Protecting: LCS Embedding

In this section, we design an equivalence protecting algorithm for the LCS-e search problem. We do so by first building a canonical representative algorithm for $\mathcal{P}$. Briefly, a *canonical representative algorithm* is a randomized algorithm which, for all inputs in the same equivalence class, gives some "canonical" output, which itself is a member of the equivalence class.

Definition 8 (Canonical Representative [13])**.** *Let $\mathcal{P} = \{\mathcal{P}_n\}_{n\in\mathbb{N}}$ be a search problem. A randomized algorithm $\mathcal{A}$ is called a* canonical representative algorithm *for $\mathcal{P}$ if (i) for every $x \in \{0,1\}^n$ it holds that $x \equiv_{\mathcal{P}} \mathcal{A}(x)$; and (ii) for every $x, y \in \{0,1\}^n$, it holds that $\mathcal{A}(x) = \mathcal{A}(y)$ iff $x \equiv_{\mathcal{P}} y$.*

We then make use of a generic construction, due to Beimel *et al.* [13], which reduces the problem of designing an equivalence protecting algorithm for $\mathcal{P}$ to that of designing an output sampling algorithm and canonical representative algorithm for $\mathcal{P}$.

Theorem 2 (Generic Construction [13])**.** *Let $\mathcal{P}$ be a search problem and $F = \{F_n\}_{n\in\mathbb{N}}$ be a family of pseudorandom functions. Suppose $\mathcal{P}$ has (i) an efficient output sampling algorithm $\mathcal{A}_{rand}$; and (ii) an efficient canonical representative algorithm $\mathcal{A}_{rep}$. Then there exists an efficient equivalence protecting algorithm for $\mathcal{P}$. The algorithm's running time is bounded by exactly one invocation of $\mathcal{A}_{rand}$, $\mathcal{A}_{rep}$, and the pseudorandom function.*

We refer the reader to Beimel *et al.* [13] for details on the construction. At a high level, however, the construction proceeds as follows: on input x, use the canonical representative algorithm to generate an instance x_{rep}; run the output sampling algorithm on x_{rep} using a source of randomness that is deterministically based on x_{rep}, *i.e.* let the algorithm's seed be $F_n(x_{rep})$.

Claim 3. *There exists an equivalence protecting algorithm for the LCS-e search problem, using $O(mn + n \log E(m,n))$ time and $O(|D|)$ space.*

The construction automatically follows from Theorem 2, our output sampling algorithm (Algorithm 1), and our canonical representative algorithm (Algorithm 2), described next.

One straight-forward strategy for computing (C, D) given the set $rep_{(A,B)}$ (*i.e.*, for Algorithm 2, Step 16) is provided in Algorithm 3. Without loss of generality we assume that $|\Sigma| \geq n$. If this is not the case, we can simply add dummy characters to the alphabet to satisfy this requirement. Let σ_k denote the lexicographically k-th character in the alphabet.

Algorithm 2 Canonical representative algorithm for LCS-e

```
1.  for $j \leftarrow n$ to 0, $i \leftarrow m$ to 0 do
2.    $rep_{(A,B)} \leftarrow \emptyset$
3.    if $i = m$ or $j = n$
4.      then $Reach[i,j] \leftarrow True$
5.    else
6.      $Reach[i,j] \leftarrow False$
7.    if $L[i,j] = L[i+1,j+1] - 1$ and $A[i+1] = B[j+1]$
8.      then $Reach[i,j] \leftarrow True$
9.    else if $L[i,j] = L[i+1,j]$ and $Reach[i+1,j]$
10.     then $Reach[i,j] \leftarrow True$
11.   else if $L[i,j] = L[i,j+1]$ and $Reach[i,j+1]$
12.     then $Reach[i,j] \leftarrow True$
13.   if $Reach[i,j]$ and $A[i] = B[j]$
14.     then $rep_{(A,B)} \leftarrow rep_{(A,B)} \cup (i,j)$
15. end for
16. Deterministically compute a pair of strings $(C,D)$
      such that $rep_{(C,D)} = rep_{(A,B)}$
17. return $(C,D)$
```

Algorithm 3 A selection mechanism for Algorithm 2

```
1.  Let $k \leftarrow 0$
2.  Sort $rep_{(A,B)} = \{(i,j)\}$ by $i$
3.  while $rep_{(A,B)} \neq \emptyset$ do
4.    Let $k \leftarrow k+1$
5.    Let $(i,j)$ be the first pair in $rep_{(A,B)}$
6.    if $C[i]$ is defined then set $D[j] \leftarrow C[i]$
7.    else if $D[j]$ is defined then set $C[i] \leftarrow D[j]$
8.    else set $C[i], D[j] \leftarrow \sigma_k$
9.    Remove $(i,j)$ from $rep_{(A,B)}$
10. end while
11. for all $i' \leq m$
12.   if $C[i']$ is not yet defined then set $C[i'] \leftarrow \sigma_{k+1}$
13. for all $i' \leq n$
14.   if $D[i']$ is not yet defined then set $D[i'] \leftarrow \sigma_{k+2}$
15. output $(C,D)$
```

Claim 4. *Algorithm 2 is a canonical representative algorithm for the LCS-e search problem and uses $O(mn)$ time and space. For this algorithm, we assume that entries of the L matrix are computed using the normal dynamic programming LCS algorithm.*

At a high level, the algorithm works by collecting any (i, j) for which there is at least one LCS of A and B where $A[i]$ and $B[j]$ are matched. Particularly, by Step 12, if $Reach[i, j] = True$ then a subsequence common to $A[1 \dots i]$ and $B[1 \dots j]$ is the prefix of an LCS for A and B. In other words, $L[m, n]$ is reachable from $L[i, j]$. Step 13 verifies, based on any of the previous conditions, that at least one LCS uses $A[i]$ and $B[j]$. This collects all the "important" structure of A and B. In the proof, we show that any two inputs are in the same equivalence class if and only if the algorithm generates the same *rep* set for them. Using the *rep* set, we deterministically output a pair of strings (C, D) such that $rep_{(C,D)} = rep$. Thus, for any two equivalent inputs, the algorithm generates the same *rep* set and outputs the same string (C, D), which is itself a member of the equivalence class.

Proof. Denote Algorithm 2 by $\mathcal{A}$. We show that for any pair of inputs $(A, B), (A', B')$, we have $(A, B) \equiv_{\text{LCS-e}} (A', B')$ iff $\mathcal{A}(A, B) = \mathcal{A}(A', B')$. This will suffice to show that $\mathcal{A}$ is a canonical representative algorithm for the LCS-e search problem.

First, we show the forward direction. If $(A, B) \equiv_{\text{LCS-e}} (A', B')$, then the length of any LCS for the pairs is the same. It suffices to show that every $(i, j) \in rep_{(A,B)}$ is also in $rep_{(A',B')}$. By construction, if $(i, j) \in rep_{(A,B)}$ then there exists at least one LCS embedding (α, β) where $A[i]$ matches $B[j]$. Since (A, B) and (A', B') are in the same equivalence class, A' and B' have the same LCS embedding (α, β) and therefore $(i, j) \in rep_{(A',B')}$. This shows that $rep_{(A,B)} = rep_{(A',B')}$. Because our choice of string is deterministic and based on $rep_{(A,B)}$, this shows $\mathcal{A}(A, B) = \mathcal{A}(A', B')$.

Second, we show the reverse direction. Let $(C, D) = \mathcal{A}(A, B)$. Recall (C, D) was chosen by the algorithm so that $rep_{(C,D)} = rep_{(A,B)}$. Following from our assumption that $\mathcal{A}(A, B) = \mathcal{A}(A', B')$, we now have that $rep_{(A,B)} = rep_{(C,D)} = rep_{(A',B')}$. We prove our claim by showing that, under this assumption, any LCS embedding for (A, B) is an LCS embedding for (A', B') and vice versa. Consider an arbitrary embedding (α, β) corresponding to an LCS of length ℓ for (A, B). The embedding (α, β) corresponds to a sequence of pairs $(i_1, j_1), \dots, (i_\ell, j_\ell)$ in $rep_{(A,B)}$ that is monotonically increasing, *i.e.* for any $(i_k, j_{k'})$ in the sequence we have $i_k < i_{k+1}$ and $j_{k'} < j_{k'+1}$. This sequence of pairs also exists in $rep_{(A',B')}$, since $rep_{(A,B)} = rep_{(A',B')}$. By our construction for the set $rep_{(A',B')}$, this means (α, β) is an embedding of a common subsequence for (A', B'). We now need to show that this common subsequence is of maximal length and therefore is an LCS for (A', B'). It suffices to argue that no monotonically increasing sequence of length $\ell' > \ell$ in $rep_{(A',B')}$ exists. If this were true, any such sequence would correspond to a common subsequence of length $\ell' > \ell$ for (A, B) too, and would contradict our assumption that (α, β) was an embedding of an LCS of length ℓ for (A, B). Therefore, (α, β) is also an LCS embedding for (A', B'). Symmetrically, one can show that any LCS embedding for (A', B') is also an LCS embedding for (A, B). This concludes our claim that $(A, B) \equiv_{\text{LCS-e}} (A', B')$.

The main loop of $\mathcal{A}$ iterates mn times. In each iteration, the set $rep_{(A,B)}$ increases by at most one. Thus, the algorithm uses $|L| + mn = O(mn)$ space and $O(mn)$ time. □

5.2 Equivalence Protecting: LCS String

Here, we design an equivalence protecting algorithm for the LCS-s search problem. Unlike before, we do not build a canonical representative algorithm for the LCS-s search problem. Instead, we take advantage of the fact that LCS-s(A, B) is a finite language (and therefore regular) and we construct an efficient acyclic deterministic finite automaton (DFA) whose language is precisely LCS-s(A, B). Essentially, we reduce the problem to that of finding an equivalence protecting algorithm for words in the language of an acyclic DFA. We give our equivalence protecting algorithm for words in the language of an acyclic DFA in Section 5.3. Our reduction works because the solution space is not distorted by the transformation; it is efficient because the DFA we produce is small.

Algorithm 4 Equivalence protecting algorithm for LCS-s

```
1. Build an acyclic DFA whose language is LCS-s$(A, B)$
    (a) Build $M'_{AB}$ such that
            $\mathcal{L}(M'_{AB}) = \{w : w \text{ is a subsequence of } A \text{ or } B\}$
    (b) Prune $M'_{AB}$ to build $M_{AB}$ such that
            $\mathcal{L}(M_{AB}) = \text{LCS-s}(A, B)$
2. Use an equivalence protecting algorithm to select a word from the language of $M_{AB}$
```

Claim 5. *Algorithm 4 is an equivalence protecting algorithm for the LCS-s search problem, running in $O(nm + n \log E(n, m))$ time and space (for alphabets of fixed size) or $O(n^2 \log E(n, m))$ time and space (for unbounded alphabets).*

Proof. The ability to use input (A, B) to efficiently construct an acyclic DFA M_{AB} where $\mathcal{L}(M_{AB}) =$ LCS-s(A, B) follows from Claim 6.

If inputs $(A, B), (A', B')$ are in the same equivalence class with respect to the privacy structure for LCS-s, then M_{AB} and $M_{A'B'}$ are in the same equivalence class with respect to the privacy structure for the language of a DFA. This follows immediately from the fact that $w \in \mathcal{L}(M_{AB})$ iff $w \in$ LCS-s(A, B). Thus, an equivalence protecting algorithm for the language of acyclic DFAs on input M_{AB} is an equivalence protecting algorithm for LCS-s on input (A, B). The existence of an equivalence protecting algorithm for the language of an acyclic DFA follows from Claim 9.

The efficiency of Algorithm 4 follows from the efficiency of Claim 6, from the efficiency of Claim 9, and from the fact that $|\mathcal{L}(M_{AB})| = |\text{LCS-s}(A, B)| \le E(n, m)$. □

The transformation we use in Step 1 of Algorithm 4 to convert input (A, B) into a DFA is similar to the algorithm of Baeza-Yates [10] used to solve the LCS problem. This algorithm builds a directed acyclic subsequence graph [19] (DASG) for strings A and B. A DASG for a string A is a DFA that accepts the language of all subsequences in A (for an example, see Figure 1).

A DASG is analogous to a directed acyclic word graph (DAWG), but while a DAWG recognizes all possible $O(n^2)$ subwords of A using $O(n)$ space, a DASG recognizes all possible 2^n subsequences of A using $O(n)$ space. We do not require our DFA to be this space-efficient (in particular, the canonical representative algorithm for finite regular languages we use will not preserve this space efficiency) and instead use a variation of Baeza-Yates' construction to build a DASG for A and B using $O(nm)$ time and $O(n^2)$ space.

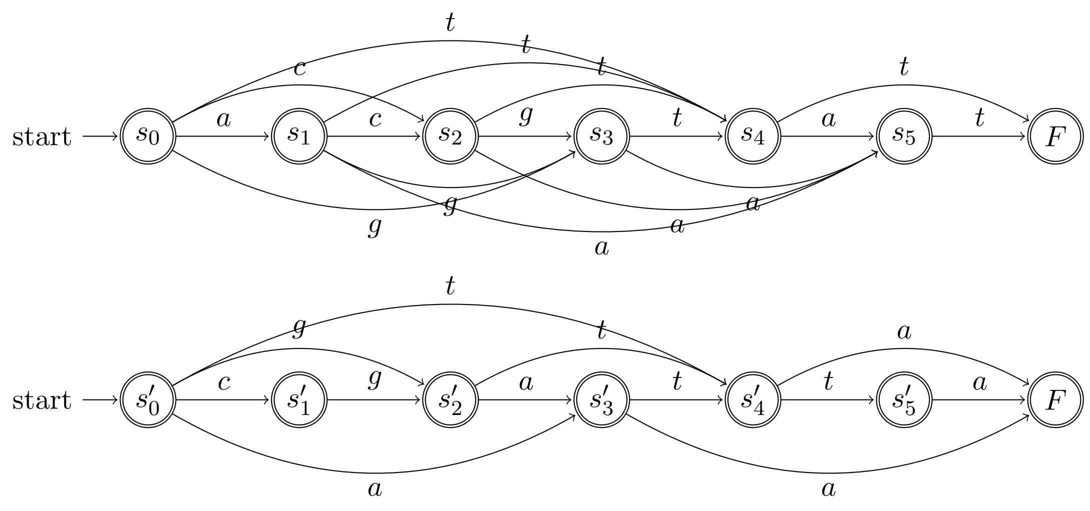

Figure 1: The DASG for input "acgtat" (above) and "cgatta" (below).

Claim 6. *On input (A, B) we can make a DFA M where $\mathcal{L}(M) = \{w : LCS(A, B, w)\}$. The DFA M has $O(n+m)$ states and $O(n^2)$ transitions. The algorithm producing M uses $O(nm)$ time and $O(n^2)$ space.*

Proof. On input (A, B) we can build a DASG M'_{AB} using $O(nm)$ time and $O(n^2)$ space. The DASG M'_{AB} has $O(n + m)$ states and $O(n^2)$ transitions, and accepts the language $\mathcal{L}(M'_{AB}) = \{w : w \text{ is a subsequence of } A \text{ or } B\}$. Additionally, each transition in M'_{AB} is labeled if the transition corresponds to part of a subsequence common to both A and B. The construction is given by Baeza-Yates [10, §5], using suffix-trees and table accesses. This DASG can be pruned in a straight-forward manner, to produce a machine M_{AB} where $\mathcal{L}(M_{AB}) = \{w : \text{LCS}(A, B, w)\}$. The pruning procedure takes time and space linear in the size of M'_{AB}, and is described in-depth in Appendix E. ☐

5.3 Equivalence Protecting: Finite Languages

Here, we design an equivalence protecting algorithm for words in the language of an acyclic DFA. We use the following notation. A deterministic finite automaton (DFA) $M = (S, \Sigma, T, s_0, A)$ is a 5-tuple where: S is a set of states; Σ is a finite alphabet; s_0 is the initial state; $A \subseteq S$ is the set of accepting states; $T : S \times \Sigma \to S$ is a function defining the transitions of the automata. Given any DFA $M = (S, \Sigma, T, s_0, A)$, define its (directed) graph $G_M = (S, E)$ with $E = \{(s, s') : \exists (s, \sigma, s') \in T\}$. A DFA is *acyclic* if its graph is acyclic.

First, we define the privacy structure for the finite language search problem. Note that while any finite language can be represented by an acyclic DFA, the size of the DFA may not be small. Our algorithms are efficient for languages whose acyclic DFA representations are small. As we have shown, this is the case for the language LCS-s.

Definition 9 (Lang Privacy Structure)**.** *For any acyclic DFA $M = (S, \Sigma, T, s_0, A)$, let Lang be the search problem for finite (regular) languages, where $Lang(M) = \{w : w \in \mathcal{L}(M)\}$. Let $\equiv_{Lang}$ be the related privacy structure, such that $M \equiv_{Lang} M'$ iff $\mathcal{L}(M) = \mathcal{L}(M')$.*

We will incrementally build our equivalence protecting algorithm for Lang, following the generic construction from Theorem 2. An algorithm $\mathcal{A}$ is a canonical representative algorithm for the Lang search problem if (i) $M \equiv_{\text{Lang}} \mathcal{A}(M)$ and (ii) for every M, M' it holds that $\mathcal{A}(M) = \mathcal{A}(M')$ iff $M \equiv_{\text{Lang}} M'$.

Claim 7. *There is a canonical representative algorithm for the Lang search problem such that, given an acyclic DFA M with $O(n)$ states and $O(n^2)$ transitions, the algorithm constructs the canonical acyclic DFA M_{rep} with $O(n)$ states and $O(n^2)$ transitions and uses $O(n)$ time and $O(n^2)$ space.*

Proof. An acyclic DFA M can be used to produce the minimal-state DFA M_{rep} in $O(|\Sigma||S|)$ time [35]. The resulting DFA will necessarily also be acyclic. M_{rep} is a canonical representative algorithm for the Lang search problem because any two DFAs M, M' accepting the same language will produce the same DFA M_{rep} when minimized. This follows from the classic Myhill-Nerode theorem on the uniqueness (up to isomorphism) of the minimal-state DFA accepting the language $\mathcal{L}(M)$. □

Claim 8. *Given a minimal-state acyclic DFA M with $|S|$ states, there is an output sampling algorithm for the Lang search problem taking $O(|S| \log |\mathcal{L}(M)|)$ time and space (for fixed alphabets) or $O(|S|^2 \log |\mathcal{L}(M)|)$ time and space (for unbounded alphabets).*

The algorithm for this claim is given in Algorithm 5. As before, we prove the claim by showing the algorithm has the non-overlapping and correctly-weighted properties. Unlike before, the algorithm and proof are much simpler, due to the fact that every path through the graph corresponds to a unique output (else, we reach a contradiction with the DFA being minimal).

Algorithm 5 Output sampling algorithm for Lang

```
 1. for all $s \in A$ do $strings(s) \leftarrow 1$
 2. for all $s \in S \setminus A$ do
 3.    let $num(s,x)$ be the number of transitions of
          the form $(s,\sigma,x) \in T$
 4.    let $strings(s) \leftarrow \sum_{x \in S} num(s,x) strings(x)$
 5. $s \leftarrow s_0$, $ans \leftarrow \epsilon$
 6. loop
 7.    $R \leftarrow \emptyset$
 8.    for all $x$ such that $\exists(s,\sigma,x) \in T$ do $R \leftarrow \{x\} \cup R$
 9.    for all $i \in R$ do $t_i \leftarrow num(s,i) strings(i)$
10.    if $s \in A$ then $R \leftarrow s \cup R$ and $t_s \leftarrow 1$ end if
11.    $t \leftarrow \sum_{i \in R} t_i$
12.    sample $z \in R$ where $Pr[z = i] = t_i/t$
13.    if $z = s$ then
14.       return $ans$
15.    else
16.       choose a random transition of the form $(s,\sigma,z) \in T$
17.       $ans \leftarrow ans||\sigma$
18.       $s \leftarrow z$
19.    end if
20. end loop
```

Proof. We prove the claim by showing Algorithm 5 has the non-overlapping and correctly-weighted properties.

In the algorithm, the labeling procedure is straight-forward since the automaton is finite and acyclic. At the end of the labeling procedure, the variable *strings*(s) represents the number of paths from s to some accepting state, because every transition leads to a state that is within the reach of an accepting state (M does not have any non-accepting states with no outgoing transitions, since these "dead states" could be removed thus contradicting the minimality of M). Each path from from s to some accepting state corresponds to a unique string, because each transition out of s is labeled with a unique character (recall, M is a DFA). Thus the algorithm has the non-overlapping property.

In the algorithm, for $i \neq s$, t_i is the number of strings that can be produced moving to state i. When s is an accepting state then we can consider returning the current string by letting $t_s = 1$. Otherwise, t_i is the number of strings produced along paths from s to some accepting state along a path passing through state i. If we select $i \neq s$, then we select a random transition to move to state i. Thus, the algorithm has the correctly-weighted property.

Note that the values *strings*(s) and t_i are never larger than the size of the language $|\mathcal{L}(M)|$. Thus, the length of the labels and the length of the numbers used for the weighted coin toss are both bound by $O(\log |\mathcal{L}(M)|)$. Thus, the labeling step takes $|T| \log |\mathcal{L}(M)|$ time and space, where $|T| = \min(|S||\Sigma|, |S|^2)$. Each iteration, the size of the set R is bound by $\min(|\Sigma|, |S|)$. Each iteration, the weighted coin toss takes $g_{wct}(|R|, |\mathcal{L}(M)|)$ time and space. The loop iterates at most $|S|$ times, because the automaton is acyclic and the longest word in its language has length $|S|$.

Thus, for alphabets of fixed size, Algorithm 5 runs in $O(|S| \log |\mathcal{L}(M)|)$ time and space. For unbounded alphabets, Algorithm 5 runs in $O(|S|^2 \log |\mathcal{L}(M)|)$ time and space. □

Claim 9. *There is an equivalence protecting algorithm for the Lang search problem such that, given an acyclic DFA $M = (S, \Sigma, T, s_0, A)$, the algorithm uses $O(|S| \log |\mathcal{L}(M)|)$ time and space (for fixed alphabets) or $O(|S|^2 \log |\mathcal{L}(M)|)$ time and space (for unbounded alphabets).*

This follows immediately from the generic construction of Theorem 2, the existence of an efficient canonical representative algorithm for this search problem (Claim 7), and the existence of an efficient output sampling algorithm for this search problem (Claim 8).

6 Conclusion

We have introduced variants of the longest common subsequence problem, a classic problem for both computer scientists and biologists, and considered each as a *private search problem.* For each we defined a strong security requirement and explored the definitions and the relationships of these privacy structures. We presented efficient private search algorithms for each variant and for the search problem of words in the language of a DFA, which may be of independent interest. The time and space required by each of our algorithms is asymptotically the same as (or better than) that of the algorithms used to count the size of the problem's output set on a given input. All our algorithms can be securely implemented in a variety of client-server or distributed settings using existing, general secure multiparty computation protocols, *e.g.* [14, 18, 23, 43].

Generating uniformly random solutions to problems is a very natural notion and, as such, a large body of research has been devoted to designing polynomial-time output sampling algorithms for a variety of combinatorial problems, such as perfect bipartite matchings [28] and spanning trees [16] (for some survey work and more examples, see [22, 29, 38]). Further complementing this line of research, we hope our work may motivate more efficient sampling algorithms for our problems, perhaps by relaxing our objectives and sampling from the solution space computationally close to uniform, or perhaps by finding more efficient counting algorithms (on which our sampling algorithms rely).

In addition to the definitions considered in this paper, Beimel *et al.* [13], propose an alternative notion for private search algorithms, called resemblance preserving algorithms. We leave it as an open problem to design *efficient* resemblance preserving algorithms for the LCS-e and the LCS-s problems.

While the literature on private genomic computation is itself still evolving, we hope our work motivates further research in this area: research on the relationship between the strong definitions of privacy presented here and the types of privacy being considered in biomedical research; research on bounding the expected value of $E(n, m)$ for realistic genomic data; and research on other private search problems of interest to genomic research.

References

[1] Are guarantees of genome anonymity realistic? `http://arep.med.harvard.edu/PGP/Anon.htm`, June 2009.

[2] CARTaGENE: A genetic map of quebec. `http://www.cartagene.qc.ca/`, June 2009.

[3] CODIS: Combined DNA index system. `http://www.fbi.gov/hq/lab/html/codis1.htm`, June 2009.

[4] deCODE genetics. `http://www.decodegenetics.com/`, June 2009.

[5] The genomic privacy project. `http://privacy.cs.cmu.edu/dataprivacy/projects/genetic/`, June 2009.

[6] HapMap: International HapMap project. `http://www.hapmap.org/`, June 2009.

[7] UK biobank. `http://www.ukbiobank.ac.uk/`, June 2009.

[8] Russ B. Altman and Teri E. Klein. Challenges for biomedical informatics and pharmacogenenomics. *Annual Review of Pharmacology and Toxicology*, 42:113–133, 2002.

[9] Mikhail J. Atallah, Florian Kerschbaum, and Wenliang Du. Secure and private sequence comparisons. In *Proceedings of the 2003 ACM Workshop on Privacy in the electronic society (WPES 2003)*, pages 39–44, 2003.

[10] Ricardo A. Baeza-Yates. Searching subsequences. *Theoretical Computer Science*, 78:363–376, 1991.

[11] Amos Beimel, Paz Carmi, Kobbi Nissim, and Enav Weinreb. Private approximation of search problems. In *Proceedings of the 38th ACM Symposium on Theory of Computing*, pages 119–128, 2006.

[12] Amos Beimel, Renen Hallak, and Kobbi Nissim. Private approximation of clustering and vertex cover. In *Proceedings of the Theory of Cryptography Conference*, pages 383–403, 2007.

[13] Amos Beimel, Tal Malkin, Kobbi Nissim, and Enav Weinreb. How should we solve search problems privately? In *Advances in Cryptology – Proceedings of CRYPTO 2007*, pages 31–49, 2007.

[14] Michal Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant disributed computation. In *STOC '88: Proceedings of the 20th annual ACM Symposium on Theory of Computing*, pages 1–10, 1988.

[15] N. Bhatnagar, S. Greenberg, and D. Randall. Sampling stable marriages: why spouse-swapping won't work. *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1223–1232, 2008.

[16] A. Broder. Generating random spanning trees. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 442–447, 1989.

[17] Paul Bunn and Rafail Ostrovsky. Secure two-party k-means clustering. In *Proceedings of the 14th ACM conference on Computer and communications security (CCS)*, pages 486–497, 2007.

[18] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols. *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[19] Maxime Crochemore, Bořivoj Melichar, and Zdeněk Troníček. Dicrected acyclic subsequence graph – overview. *Journal of Discrete Algorithms*, 1(3–4):255–280, 2003.

[20] Matthew Franklin, Mark Gondree, and Payman Mohassel. Improved efficiency for private stable matching. In *The Cryptographer's Track at the RSA Conference (CT-RSA)*, pages 163–177, 2007.

[21] Matthew Franklin, Mark Gondree, and Payman Mohassel. Communication-efficient private protocols for longest common subsequence. In *The Cryptographer's Track at the RSA Conference (CT-RSA)*, pages 265–278, 2009.

[22] Vibhav Gogate and Rina Dechter. A new algorithm for sampling CSP solutions uniformly at random. In *Principles and Practice of Constraint Programming - CP 2006*, pages 711–715, 2006.

[23] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game or a completeness theorem for protocols with honest majority. In *STOC '87: Proceedings of the 19th annual ACM Symposium on Theory of Computing*, pages 218–229, 1987.

[24] Mark Gondree and Payman Mohassel. Longest common subsequence as private search. In *Proceedings of the 2009 ACM Workshop on Privacy in the electronic society (WPES 2009)*, 2009.

[25] Ronald I. Greenberg. Bounds on the number of longest common subsequences. arXiv:cs/0301030v2, Aug 2003. `http://arxiv.org/abs/cs/0301030v2`.

[26] Ronald I. Greenberg. Computing the number of longest common subsequences. arXiv:cs/0301034v1, Jan 2003. `http://arxiv.org/abs/cs/0301034v1`.

[27] Shai Halevi, Robert Krauthgamer, Eyal Kushilevitz, and Kobbi Nissim. Private approximation of NP-hard functions. In *Proceedings of the 33rd ACM Symposium on Theory of Computing*, pages 550–559, 2001.

[28] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *Journal of the ACM*, 51(4):671–697, 2004.

[29] Mark Jerrum, Leslie G. Valiant, and Vijay V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science*, 43:169–188, 1986.

[30] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards practical privacy for genomic computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[31] Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In *Advances in Cryptology – Proceedings of CRYPTO 2000*, pages 36–54, 2000.

[32] Bradley Malin and Latanya Sweeney. Re-identification of DNA through an automated linkage process. In *Proceedings/AMIA Annual Symposium*, pages 423–427, 2001.

[33] Kobbi Nissim and Roie Zivan. Secure DisCSP protocols - from centralized towards distributed solutions. In *Proceedings of the 6th Workshop on Distributed Constraint Reasoning (DCR-05)*, 2005.

[34] Department of Health and Human Services. 45 CFR (Code of Federal Regulations), parts 160–164. Standards for privacy of individually identifiable health information, final rule. Federal Register: 67 (157): 53182-53273, August 12 2002.

[35] Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92:181–189, 1990.

[36] Marius-Calin Silaghi. Solving a distributed CSP with cryptographic multi-party computations, without revealing constraints and without involving trusted servers. In *Proceedings of the 4th Workshop on Distributed Constraint Reasoning (DCR-03)*, 2003.

[37] Marius-Calin Silaghi and Debasis Mitra. Distributed constraint satisfaction and optimization with privacy enforcement. In *Proceedings of the 3rd International Conference on Intelligence Agent Technology*, pages 531–535, 2004.

[38] Alistair Sinclair. *Algorithms for Random Generation and Counting: A Markov Chain Approach*. 1993.

[39] Frank Stajano, Lucia Bianchi, Pietro Liò, and Douwe Korff. Forensic genomics: kin privacy, driftnets and other open questions. In *Proceedings of the 2008 ACM Workshop on Privacy in the electronic society (WPES 2008)*, pages 15–22, 2008.

[40] Koutarou Suzuki and Makoto Yokoo. Secure combinatorial auctions by dynamic programming with polynomial secret sharing. In *Financial Cryptography 2002*, pages 44–56, 2003.

[41] Doug Szajda, Michael Pohl, Jason Owen, and Barry G. Lawson. Toward a practical data privacy scheme for a distributed implementation of the Smith-Waterman genome sequence comparison algorithm. In *Proceedings of the 2006 ISOC Network and Distributed System Security Symposium (NDSS 2006)*, pages 253–265, 2006.

[42] Laszlo T. Vaszar, Mildred K. Cho, and Thomas A. Raffin. Privacy issues in personalized medicine. *Pharmacogenomics*, 4(2):107–112, 2003.

[43] Andrew C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 162–167, 1986.

[44] Makoto Yokoo and Koutarou Suzuki. Secure multi-agent dynamic programming based on homomorphic encryption and its application to combinatorial auctions. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 112–119, 2002.

[45] Makoto Yokoo, Koutarou Suzuki, and Katsutoshi Hirayama. Secure distributed constraint satisfaction: Reaching agreement without revealing private information. *Artificial Intelligence Journal*, 161(1–2):229–246, 2005.

A Definitions

Definitions 10–14 are adapted from definitions provided by Beimel *et al.* [11, 13].

Definition 10 (Seeded Algorithms). *A seeded algorithm $\mathcal{A}$ is a deterministic polynomial time algorithm taking two inputs x, s_n where $|x| = n$ and $|s_n| = p(n)$ for some polynomial p. The distribution induced by a seeded algorithm on an input x is the distribution on outcomes $\mathcal{A}(x, s_n)$ where s_n is chosen uniformly at random from $\{0,1\}^{p(|x|)}$.*

Definition 11 (Output Sampling Algorithm). *Let $\mathcal{P} = \{\mathcal{P}_n\}_{n\in\mathbb{N}}$ be a search problem. A seeded algorithm $\mathcal{A}$ is called an* output sampling algorithm *for $\mathcal{P}$ if for every pair of strings $x \in \{0,1\}^n$ the distribution $\mathcal{A}(x, s_n)$ is computationally indistinguishable from the uniform distribution on $\mathcal{P}(x)$.*

Definition 12 (Private Oracle). *Let $\mathcal{P} = \{\mathcal{P}_n\}_{n\in\mathbb{N}}$ be a search problem and p be the polynomial such that $\mathcal{P}_n : \{0,1\}^n \to 2^{\{0,1\}^{p(n)}}$. We say that for a given $n \in \mathbb{N}$, an oracle $O_n : \{0,1\}^n \to \{0,1\}^{p(n)}$ is private with respect to the privacy structure $\equiv_{\mathcal{P}}$ if (i) for every $x \in \{0,1\}^n$ it holds that $O_n(x) \in \mathcal{P}_n(x)$ (that is, O_n returns correct answers); (ii) for every $x, y \in \{0,1\}^n$ it holds that $x \equiv_{\mathcal{P}} y$ implies $O_n(x) = O_n(y)$. We say such an oracle is* private *with respect to $\mathcal{P}$, or* protects the equivalence relation $\equiv_{\mathcal{P}}$.

Definition 13 (Equivalence Protecting Algorithm). *Let $\mathcal{P} = \{\mathcal{P}_n\}_{n\in\mathbb{N}}$ be a search problem. An algorithm $\mathcal{A}(\cdot,\cdot)$ is an* equivalence protecting algorithm *for $\equiv_{\mathcal{P}}$, if for every polynomial time oracle machine D, for every polynomial p, and for any sufficiently large n,*

$$|Pr[D^{O_n}(1^n) = 1] - Pr[D^{\mathcal{A}(\cdot,s_n)}(1^n) = 1]| < 1/p(n)$$

where the first probability is over the uniform distribution over oracles O_n that are private with respect to $\mathcal{P}$, and the second probability is uniform over the choices of the seed s_n for the algorithm $\mathcal{A}$.

Definition 14 (Canonical Representative Algorithm). *Let $\mathcal{P} = \{\mathcal{P}_n\}_{n\in\mathbb{N}}$ be a search problem. A randomized algorithm $\mathcal{A}$ is called a* canonical representative algorithm *for* $\mathbf{P}$ *if (i) for every $x \in \{0,1\}^n$ it holds that $x \equiv_{\mathcal{P}} \mathcal{A}(x)$; and (ii) for every $x, y \in \{0,1\}^n$, it holds that $\mathcal{A}(x) = \mathcal{A}(y)$ iff $x \equiv_{\mathcal{P}} y$.*

Definition 15 (DFA). *A deterministic finite automaton (DFA) $M = (S, \Sigma, T, s_0, A)$ is a 5-tuple where: S is a set of states; Σ is a finite alphabet; s_0 is the initial state; $A \subseteq S$ is the set of accepting states; $T : S \times \Sigma \to S$ is a function defining the transitions of the automata.*

From state s on input $\sigma \in \Sigma$, let $\langle s, \sigma\rangle = T(s, \sigma)$ be the state reached from the transition. Let the notation be transitive: if $w = \sigma_1\sigma_2 \ldots \sigma_\ell$ is a word, then let $\langle s, w\rangle$ be the state reached by transitioning from s given the ordered inputs $\sigma_1, \sigma_2, \ldots, \sigma_\ell$. A word w is accepted by the automaton if $\langle s_0, w\rangle \in A$. Define the automaton's language as $\mathcal{L}(M) = \{w : \langle s_0, w\rangle \in A\}$.

Definition 16 (Acyclic DFA). *Given a DFA $M = (S, \Sigma, T, s_0, A)$, define its (directed) graph $G_M = (S, E)$ with $E = \{(s, s') : \exists(s, \sigma, s') \in T\}$. A DFA is* acyclic *if its graph is acyclic. The language of an acyclic DFA is finite and composed entirely of finite-length words.*

Any regular language is accepted by some DFA, and any finite language is accepted by some acyclic DFA.

B LCS Counting Algorithms

In this section, we summarize two basic dynamic programming algorithms due to Greenberg [26] for counting the number of solutions to the LCS-e and LCS-s search problems. We use these as a foundation for building our output sampling algorithms. For each counting algorithm, we assume the typical LCS dynamic programming matrix L has already been populated.

Claim 10. *Algorithm 6 correctly counts the number of solutions to LCS-e or LCS-s. Each requires $O(mn \log E(m,n))$ time and space.*

Algorithm 6 Count LCS Embeddings and Strings [26].

To count solutions in LCS-e, use with $\boxed{\text{boxed code}}$.

To count solutions in LCS-s, use with $\fcolorbox{black}{lightgray}{\text{shaded code}}$.

```
for $j \leftarrow 0$ to $n$, $i \leftarrow 0$ to $m$ do
  if $i = 0$ or $j = 0$ then
    $D[i,j] \leftarrow 1$
  else
    $D[i,j] \leftarrow 0$
    if $A[i] = B[j]$ then
      $D[i,j] \leftarrow D[i-1,j-1]$
    $\boxed{\textbf{end if}}$
    $\fcolorbox{black}{lightgray}{\textbf{else}}$
      if $L[i-1,j] = L[i,j]$ then
        $D[i,j] \leftarrow D[i,j] + D[i-1,j]$
      end if
      if $L[i,j-1] = L[i,j]$ then
        $D[i,j] \leftarrow D[i,j] + D[i,j-1]$
      end if
      if $L[i-1,j-1] = L[i,j]$ then
        $D[i,j] \leftarrow D[i,j] - D[i-1,j-1]$
      end if
    $\fcolorbox{black}{lightgray}{\textbf{end if}}$
  end if
end for
return $D[m,n]$
```

C Privacy Structure Relationships

Claim 11. *Privacy structures $\equiv_{LCS\text{-}s}$ and $\equiv_{LCS\text{-}e}$ are incomparable.*

Proof. The following proves the claim: (i) $(cat, ct) \equiv_{\text{LCS-e}} (gat, gt)$ but $(cat, ct) \not\equiv_{\text{LCS-s}} (gat, gt)$; (ii) $(cat, ct) \equiv_{\text{LCS-s}} (ctg, ct)$ but $(cat, ct) \not\equiv_{\text{LCS-e}} (ctg, ct)$. □

Claim 12. *Some natural deterministic selection strategies which output strings (embeddings) on inputs of length n may leak $\Theta(n)$ bits with respect to the privacy structure $\equiv_{LCS\text{-}s}$ ($\equiv_{LCS\text{-}e}$).*

Proof. Let $n = 3\ell$ and $|\Sigma| \geq n$. Consider the following grammars:

$$\begin{aligned} G &::= (\text{“}abc\text{”}|\text{“}bca\text{”})||(\text{“}def\text{”}|\text{“}efd\text{”})||(\text{“}ghi\text{”}|\text{“}hig\text{”})\ldots \\ X &::= (\text{“}a\text{”}|\text{“}b\text{”}|\text{“}c\text{”})||(\text{“}d\text{”}|\text{“}e\text{”}|\text{“}f\text{”})||(\text{“}g\text{”}|\text{“}h\text{”}|\text{“}i\text{”})\ldots \end{aligned}$$

For each $A \in \mathcal{L}(G)$ there exists a B such that for all $x \in \mathcal{L}(X)$ we have $\text{LCS}(A, B, x)$, and for all $x' \notin \mathcal{L}(X)$ it is not true that $\text{LCS}(A, B, x')$. Specifically for $A = abc\ldots$ we choose $B = cba\ldots$, and for $A = bca\ldots$ we choose $B = acb\ldots$. Let $S = \{(A, B) : A \in \mathcal{L}(G)$ and B is constructed in this manner$\}$.

For all $(A, B), (A', B') \in S$, we have

$$(A, B) \equiv_{\text{LCS-s}} (A', B') \text{ and } (A, B) \equiv_{\text{LCS-e}} (A', B')$$

Now, consider any deterministic selection strategy f_{lcs} that recovers a longest common sequence string. Of the longest common subsequences, it must return one. Each unique LCS embedded in input A corresponds to a unique string. Without loss of generality, say the deterministic algorithm $f_{lcs}(A, B)$ returns the LCS with the “earliest” embedding in A. For example, our strategy yields $f_{lcs}(abc, cba) = a$ while $f_{lcs}(bca, acb) = b$.

Thus, $\{f_{lcs}(A, B) : (A, B) \in S\}$ is a set of size 2^ℓ. Therefore, f_{lcs} partitions the equivalence class into 2^ℓ sub-classes. Thus, this deterministic backtracking algorithm leaks $\ell = \Theta(n)$ bits relative to $\equiv_{\text{LCS-s}}$.

To show the same result for embeddings, let $f_{lcs}(A, B)$ be the deterministic selection strategy that returns the embedding of a longest common subsequence. Consider the strategy that, at each step, chooses the lexicographically first character. For example, $f_{lcs}(gcta, cagt)$ returns the embedding of ca. Again, the set $\{f_{lcs}(A, B) : (A, B) \in S\}$ contains 2^ℓ unique embeddings (all embed the string $adgj\ldots$), thus dividing the equivalence class into 2^ℓ sub-classes. □

Definition 17 (Embeddings with Strings)**.** *Let $LCSes(A, B, \alpha, \beta, x)$ be true if $LCS(A, B, x)$, $LCS\text{-}e(A, B, \alpha, \beta)$ and $x = v(A, \alpha)$, and false otherwise. Define LCS-es as the search problem for LCS embeddings with strings, where for all $A \in \Sigma^m, B \in \Sigma^n$, we have*

$$LCS\text{-}es(A, B) = \{(\alpha, \beta, x) : \text{LCSes}(A, B, \alpha, \beta, x)\}$$

Let $\equiv_{LCS\text{-}es}$ be the related privacy structure, where if $LCS\text{-}es(A, B) = LCS\text{-}es(A', B')$ we say $(A, B) \equiv_{LCS\text{-}es} (A', B')$. For example, $(atg, aga) \equiv_{LCS\text{-}es} (acg, agt)$.

As one might expect, $\equiv_{\text{LCS-es}}$ refines both $\equiv_{\text{LCS-s}}$ and $\equiv_{\text{LCS-e}}$. In fact, as we show next, the structure $\equiv_{\text{LCS-es}}$ leaks $\Theta(n)$ bits relative to both $\equiv_{\text{LCS-s}}$ and $\equiv_{\text{LCS-e}}$.

Claim 13. *Privacy structure $\equiv_{LCS\text{-}es}$ leaks $\Theta(n)$ bits relative to privacy structure $\equiv_{LCS\text{-}s}$.*

Proof. It it clear that $\equiv_{\text{LCS-es}}$ refines $\equiv_{\text{LCS-s}}$. Consider the set S from before. For $(A, B), (A', B') \in S$, the strings are structurally the same but the embeddings yield different values. Specifically, $(abc\dots, cba\dots)$ and $(bca\dots, acb\dots)$ both have embedding $(100\dots, 001\dots)$ but in one case the embedding's value is $a\dots$ and in the other it is $b\dots$. In fact, this set is divided by $\equiv_{\text{LCS-es}}$ into 2^ℓ sub-classes. Thus, $\equiv_{\text{LCS-es}}$ leaks $\Theta(n)$ bits relative to $\equiv_{\text{LCS-s}}$. $\square$

Claim 14. *Privacy structure $\equiv_{LCS\text{-}es}$ leaks $\Theta(n)$ bits relative to privacy structure $\equiv_{LCS\text{-}e}$.*

Proof. It is clear that $\equiv_{\text{LCS-es}}$ refines $\equiv_{\text{LCS-e}}$. Consider the set S from before. Let A^π, B^π be the strings formed from A by permuting the characters in Σ according to π. For all $(A, B) \in S$, we have

$$(A^\pi, B^\pi) \equiv_{\text{LCS-e}} (A^{\pi'}, B^{\pi'}) \text{ and } (A^\pi, B^\pi) \not\equiv_{\text{LCS-es}} (A^{\pi'}, B^{\pi'})$$

Permuting the alphabet will not change the embeddings (the strings are, structurally, the same as before), but it will change the values of the strings at those embeddings. When $|\Sigma| = n = 3\ell$, this divides $\equiv_{\text{LCS-e}}$ into $\binom{n}{\ell}$ sub-classes. Since $(3e)^\ell \leq \binom{n}{\ell} \leq 3^\ell$, $\equiv_{\text{LCS-es}}$ leaks $\Theta(n)$ bits relative to $\equiv_{\text{LCS-e}}$. $\square$

D Output Sampling: LCS String

For comparison purposes, we give the output sampling algorithms for both the LCS-e and LCS-s search problems in Algorithm 7. For each variant, we assume that the L matrix and the D matrix are already computed using the normal LCS dynamic programming algorithm and the appropriate variant of the counting algorithm (see Algorithm 6), respectively.

Algorithm 7 Output sampling algorithms for LCS variants

```
For LCS-e, use with $\boxed{\text{boxed code}}$.
For LCS-s, use with $\colorbox{lightgray}{shaded code}$.
 1. $\alpha, \beta, s \leftarrow \epsilon$; $i \leftarrow m$; $j \leftarrow n$; $forceUp, forceLeft \leftarrow false$
 2. while $i > 0$ and $j > 0$ do
 3.    $W_1, W_2, W_3, W_4 \leftarrow 0$
 4.    $\boxed{\textbf{if } A[i] = B[j] \textbf{ then } W_1 \leftarrow D[i-1, j-1]}$
 5.    if $L[i-1, j-1] = L[i, j]$ then $W_2 \leftarrow D[i-1, j-1]$
 6.    if $L[i-1, j] = L[i, j]$ then $W_3 \leftarrow D[i-1, j] - W_2$
 7.    if $L[i, j-1] = L[i, j]$ then $W_4 \leftarrow D[i, j-1] - W_2$
 8.    if $forceUp$ then sample $z \in \{1, 3\}$ where
         $\Pr[z = i] = W_i/(W_1 + W_3)$
 9.    else if $forceLeft$ then sample $z \in \{1, 4\}$ where
         $\Pr[z = i] = W_i/(W_1 + W_4)$
10.    else sample $z \in \{1, 2, 3, 4\}$ where
         $\Pr[z = i] = W_i/(W_1 + W_2 + W_3 + W_4)$
11.    if $z = 1$ $\colorbox{lightgray}{or $A[i] = B[j]$}$ then
12.       $\alpha \leftarrow 1||\alpha$; $\beta \leftarrow 1||\beta$; $s \leftarrow A[i]||s$
13.       $forceUp, forceLeft \leftarrow false$
14.       $i \leftarrow i - 1,\ j \leftarrow j - 1$
15.    else if $z = 2$ then
16.       $\alpha \leftarrow 0||\alpha$; $\beta \leftarrow 0||\beta$
17.       $i \leftarrow i - 1,\ j \leftarrow j - 1$
18.    else if $z = 3$ then
19.       $\alpha \leftarrow 0||\alpha$; $forceUp \leftarrow true$
20.       $i \leftarrow i - 1$
21.    else if $z = 4$ then
22.       $\beta \leftarrow 0||\beta$; $forceLeft \leftarrow true$
23.       $j \leftarrow j - 1$
24.    end if
25. end while
26. $\boxed{\textbf{return } (\alpha, \beta)}$ $\colorbox{lightgray}{\textbf{return} $s$}$
```

Claim 15. *There exists an output sampling algorithm for the LCS-s search problem using $O(n \log E(m, n))$ time and $O(|D|)$ space(see Algorithm 7).*

Proof. The efficiency claim follows the proof of Claim 2, due to the closeness in operation of the two variants. The correctness claim mostly follows from the same argument used in Claim 2, with the following differences. In the counting algorithm for LCS-s, there are two cases when determining contributions to $D[i,j]$: when $A[i] = B[j]$ and when $A[i] \neq B[j]$. In the first case, the only contribution to $D[i,j]$ is from $D[i-1,j-1]$ which corresponds to the scenario where the characters are matched; thus, we make this choice deterministically in the algorithm. When $A[i] \neq B[j]$, we proceed with the coin tossing as usual, weighing each choice by how much it has contributed to the total $D[i,j]$; notice the value $D[i-1,j-1]$ does not contribute to $D[i,j]$ when $A[i] \neq B[j]$, and this is reflected in the fact that $W_1 = 0$ and some of the remaining coin tosses become trivial. $\square$

E Pruning procedure for DASG

We can prune the DASG M'_{AB} to produce the acyclic DFA M_{AB}, the automaton which accepts the language $\mathcal{L}(M_{AB}) = \{w : \text{LCS}(A, B, w)\}$. The pruning procedure is given below. For an example of the procedure, see Figures 2–5.

1. For any state s, we call state s' reachable from s if there is a transition from s to s' that is labeled as being common to A and B.
2. Use depth-first search to label the DFA $M'_{AB} = (S', \Sigma, T', s_0, A')$, starting at $s = s_0$:
   (a) For any state s' reachable from s,
   set $\mathit{label}(s) = 1 + \max\{\mathit{label}(s')\}$.
   (b) If there are no states reachable from s,
   set $\mathit{label}(s) = 0$.
3. For all $s \in A'$, if $\mathit{label}(s) = 0$, then add s to A.
4. For all $(s_i, \sigma, s_j) \in T'$, if $\mathit{label}(s_i) = 1 + \mathit{label}(s_j)$ then add (s_i, σ, s_j) to T
5. Return $M_{AB} = (S', \Sigma, T, s_0, A)$.

From the construction, because we have only pruned transitions, it is clear that M_{AB} is also acyclic and $\mathcal{L}(M_{AB}) \subseteq \mathcal{L}(M'_{AB})$. To show that $\mathcal{L}(M_{AB}) = \{w : \text{LCS}(A, B, w)\}$, it suffices to show that any subsequence whose length is not maximal is not accepted by M_{AB}. If the length of the LCS for A and B is ℓ, then $\mathit{label}(s_0) = \ell$. Any word of length $\ell' < \ell$ needs to reach an accepting state s (where $\mathit{label}(s) = 0$) using ℓ' transitions. To so do would mean using a transition from s_i to s_j where $\mathit{label}(s_i) > 1 + \mathit{label}(s_j)$. By construction, no such transition exists in M_{AB}. Thus the pruned DFA only accepts words of length ℓ. This suffices to show that the pruned DFA accepts only common subsequences of length ℓ, *i.e.* longest common subsequences. This size of M_{AB} is, in the worst case, the same as the size of M'_{AB}.

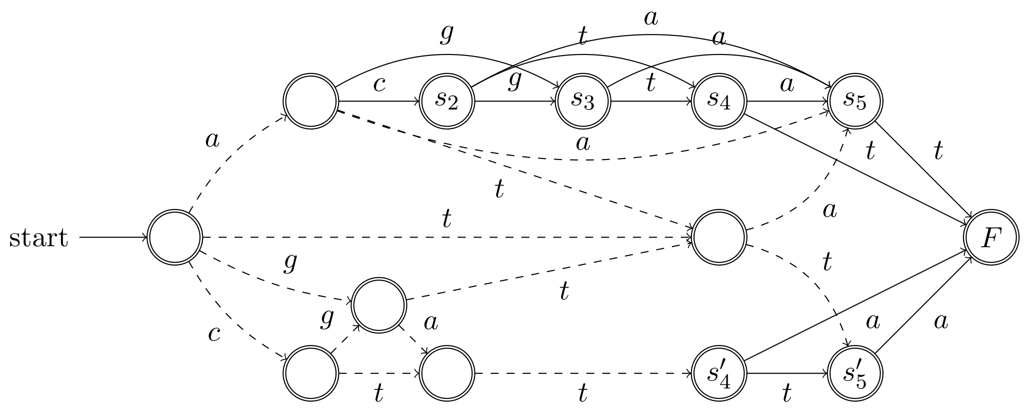

Figure 2: M'_{AB}, generated by the construction of Baeza-Yates [10]. States inherited from a string's DASG (see Figure 1) have been left labeled; dotted lines represent shared subsequences.

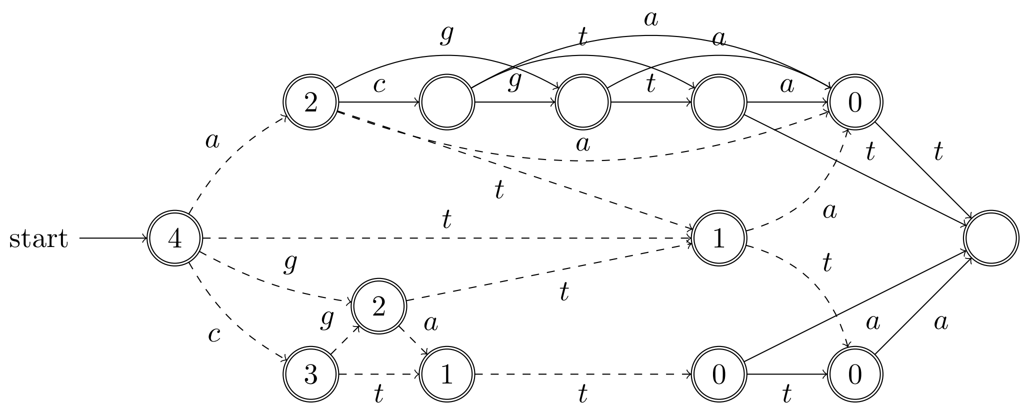

Figure 3: M'_{AB}, labeled according to the pruning procedure.

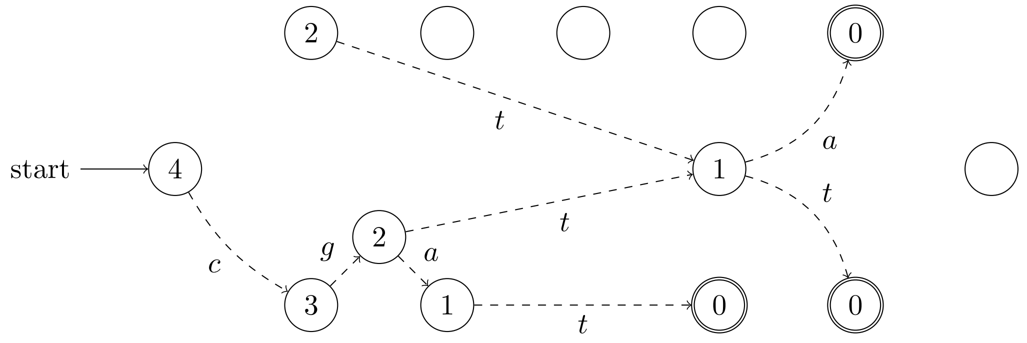

Figure 4: M_{AB}, after the pruning procedure.

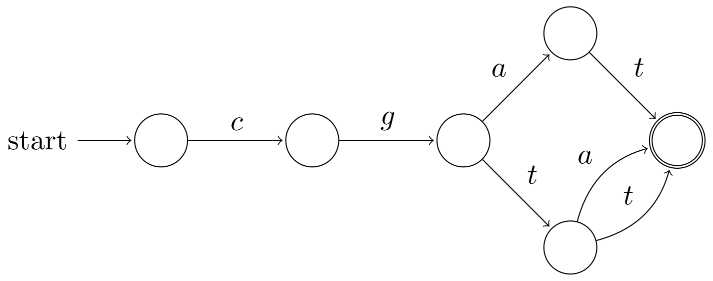

Figure 5: $M_{[AB]}$, the canonical representative automaton (*i.e.*, state-minimized canonical DFA) for M_{AB} from Figure 4.