A New Algorithm for "The LCS Problem" with Application in Compressing Genome Resequencing Data

Richard Beal, Tazin Afrin, Aliya Farheen, and Don Adjeroh
Lane Department of Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV, US
r.beal@computer.org, don@csee.wvu.edu

***Abstract*—The longest common subsequence (LCS) problem is a classical problem in computer science, and forms the basis of the current best-performing reference-based compression schemes for genome resequencing data. First, we present a new algorithm for the LCS problem. Then, we introduce an LCS-motivated reference-based compression scheme using the components of the LCS, rather than the LCS itself. For the *Homo sapiens* genome (original size 3,080,436,051 bytes), our proposed scheme compressed the genome to 5,267,656 bytes). This can be compared with the previous best results of 19,666,791 bytes (Wang and Zhang, 2011) and 17,971,030 bytes (Pinho, Pratas, and Garcia, 2011). Thus, our compression ratio is about 3.73 to 3.41 times better than those from the state-of-the-art reference-based compression algorithms.**

***Keywords*-longest common subsequence, LCS, longest previous factor, LPF, compression, biology, genome resequencing**

I. Introduction and Background

Measuring similarity between sequences, be it DNA, RNA, or protein sequences, is at the core of various problems in molecular biology. An important approach to this problem is computing the longest common subsequence (LCS) between two strings S_1 and S_2, i.e. the longest ordered list of symbols common between S_1 and S_2. For example, when $S_1 = \texttt{abba}$ and $S_2 = \texttt{abab}$, we have the following LCSs: `abb` and `aba`. The LCS has been used to study various string analysis problems [1], [2]. Biological applications of the LCS and similarity measurement are varied, from sequence alignment [3] in comparative genomics [4], to phylogenetic construction and analysis, to rapid search in huge biological sequences [5], to compression and efficient storage of the rapidly expanding genomic data sets [6], to re-sequencing a set of strings given a target string [7], which is important in efficient genome assembly.

Finding the LCS between the n-length S_1 and m-length S_2 is relatively easy, the real challenge is to do this in a time- and space-efficient manner. The LCS computation is a classical computer science problem with a dynamic programming solution on an m-by-n grid (see [1], [2]). The grid is populated and a trace back is used to compute the LCS in $O(mn)$ time and $O(mn)$ space. This trace back was proposed as a minimum cost path determination problem by Myers et al. [8] and Ukkonen [9]. Hunt and Szymanski [10] earlier used an essentially similar approach to solve the LCS problem in $(r+n)\log n$ time, with $n \ll m$, where r is the number of pairwise symbol matches. When two non-similar files are compared, we will have $r \ll mn$, or r in $O(n)$, leading to a practical $O(n \log n)$ time algorithm. However, for very similar files, we have $r \approx mn$, or an $O(mn \log n)$ algorithm. Space-efficient algorithms for the LCS problem has also been considered (see [11], [12]).

The LCS has been used in some recent algorithms to compress genome resequencing data [13], [14]. Compression of biological sequences is an important but difficult problem, which has been studied for decades by various authors (see [6], [5], [15]). Most of the earlier studies focused on lossless compression and generally exploited self-contained redundancies, without using a reference sequence. Lossy compression was proposed in [16], [17] for high throughput sequences admitting limited errors.

More recent methods ([14], [13]) have considered lossless compression of re-sequencing data by exploiting the significant redundancies between the genomes from related species, reporting compression ratios in the order of 80 to 18,000 without loss. The LCS is the hallmark of these reference-based approaches. In this work, we first introduce a new algorithm for the LCS problem, using suffix trees and shortest-path graph algorithms. Motivated by our LCS algorithm, we introduce an improved reference-based compression scheme for resequencing data using the longest previous factor (LPF) data structure [18], [19], [20].

II. Preliminaries

A string T is a sequence of symbols from some alphabet Σ. We append a terminal symbol $\$ \notin \Sigma$ to strings for completeness. A string or data structure D has length $|D|$, and its ith element is indexed by $D[i]$, where $1 \leq i \leq |D|$. A prefix of a string T is $T[1...i]$ and a suffix is $T[i...|T|]$, where $1 \leq i \leq |T|$. The suffix tree (ST) on the n-length T is a compact trie (with $O(n)$ nodes constructed in $O(n)$ time [2]) that represents all of the suffixes of T. Suffixes with common prefixes share nodes in the tree until the suffixes differentiate and ultimately, each suffix $T[i...n]$ will have its own leaf node to denote i. A generalized suffix tree (GST) is an ST for a set of strings. A substring of T is $T[i...j]$,

978-1-4673-6799-8/15/$31.00 ©2015 IEEE

where $1 \leq i \leq j \leq n$. The longest common subsequence is defined below in terms of length-1 common substrings.

Definition 1: **Longest common subsequence** (LCS): For the n-length S_1 and m-length S_2, the LCS between S_1 and S_2 is the length of the longest sequence of pairs $\mathcal{M} = \{m_1, ..., m_M\}$, where $m_i = (u, v)$ such that $S_1[m_h.u] = S_2[m_h.v]$ for $1 \leq h \leq M$ and $m_i.u < m_{i+1}.u \ \wedge \ m_i.v < m_{i+1}.v$ for $1 \leq i < M$.

III. LCS Algorithm

Below, we compute the LCS between S_1 and S_2 in the following way. (i) We use the GST to compute the common substrings (CSSs) shared between S_1 and S_2. (ii) We use the CSSs to construct a directed acyclic graph (DAG) of maximal CSSs. (iii) We compute LCS by finding the longest path in the DAG. Steps (i) and (iii) are standard tasks. For step (ii), we develop new algorithms and data structures.

A. Computing the CSSs

We now briefly describe finding the common substrings (CSSs) between S_1 and S_2. In our LCS algorithm, for simplicity of discussion, we will only use CSSs of length-1.

Let $\mathcal{A} = \emptyset$. Compute the GST on $S_1\$_1 \circ S_2\$_2$, for terminals $\{\$_1, \$_2\}$. Consider a preorder traversal of the GST. When at depth-1 for a node N, let $\mathcal{S} = \emptyset$. During the preorder traversal from N, we collect in $\mathcal{S}$ all of the suffix index leaves descending from N, which represent the suffixes that share the same first symbol. Let $\mathcal{S}_1 = \mathcal{S}_2 = \emptyset$. For $s \in \mathcal{S}$, if $s \leq |S_1|$, then store s in $\mathcal{S}_1$. Otherwise, store s in $\mathcal{S}_2$. We represent all of our length-1 matches in the following structure: MATCH $\{id, p1, p2\}$. The id is a unique number for the MATCH, and p_1 and p_2 are respectively the positions in S_1 and S_2 where the CSS exists. Let $id = 2$. Now, for each $s_1 \in \mathcal{S}_1$, we create a new MATCH $m = (id{+}{+}, s_1, s_2)$ for each $s_2 \in \mathcal{S}_2$. Store each m in $\mathcal{A}$.

The running time is clearly the maximum of the GST construction and the number of length-1 CSSs.

Lemma 2: Say n=$|S_1|$ and m=$|S_2|$, then computing the η CSSs of length-1 between S_1 and S_2 requires $O(\max\{n + m, \eta\})$ time.

B. DAG Construction

Given all of the MATCHes found in $\mathcal{A}$, our task now is to construct the DAG for $\mathcal{A}$. For all paths of the DAG to start and end at a common node, we make MATCHes S and E to respectively precede and succeed the MATCHes in $\mathcal{A}$. (Let S have $id = 1$ and E have $id = |\mathcal{A} + 2|$ and then store S and E in $\mathcal{A}$.) The goal of the DAG is to represent all maximal CSSs between S_1 and S_2 as paths from S to E. We will later find the LCS, the longest such path.

In the DAG, the nodes will be the MATCH ids and the edges between MATCHes, say m_1 and m_2, represent that $S_1[m_1.p1] = S_2[m_1.p2]$ is chosen in the maximal common subsequence followed by $S_1[m_2.p1] = S_2[m_2.p2]$. The DAG is acyclic because, by Definition 1, the LCS is a list of ordered MATCHes. Since we cannot choose $m_i \in \mathcal{M}$ and then $m_h \in \mathcal{M}$ with $h < i$, then no cycle can exist.

Our DAG construction, displayed in Algorithm 1, operates in the following way. We initialize the DAG dag by first declaring $dag.gr$ of size $|\mathcal{A}|$, since gr will represent all of the nodes. All outgoing edges for say the node $N \in \mathcal{A}$ are represented by $dag.gr[N.id][1...dag.sz[N.id]]$. By setting $dag.sz = \{0, ..., 0\}$, we clear the edges in our dag. Now, setting these edges is the main task of our algorithm.

We can easily construct the edges by assuming that there exists a data structure PREV pv that can tell us the set of parents for each node $a \in \mathcal{A}$. That is, we can call `getPrnts`(pv, L) to get the set of nodes P that *directly precede* MATCH $L \in \mathcal{A}$ in the final dag. By "directly precede", we mean that in the final dag, there is connection from each $p \in P$ to a, i.e. each p is in *series* with a, meaning that both p AND a are chosen in a maximal CSS. Further, no $p, p2 \in P$ can be in series with one another, and rather, they are in *parallel* with one another, meaning that either p OR $p2$ is chosen in a maximal common subsequence.

With P, we can build an edge from $a2 \in P$ to a by first allocating a new space in $dag.gr[a2.id]$ by incrementing $dag.sz[a2.id]$ and then making a directed edge from parent to child, i.e. $dag.gr[a2.id][dag.sz[a2.id]] = a.id$. After computing the incoming edges for each node $a \in \mathcal{A}$, the dag construction is complete.

1) PREV Data Structure: The simplicity of the DAG construction is due to the PREV pv, detailed here. The pv is composed of four attributes.

HashMap<int,int> $p1$. Suppose that all $a.p1$ values (for $a \in \mathcal{A}$) are placed on an integer number line. It is very unlikely that all $a.p1$ values will be consecutive and so, there will be unused numbers (gaps) between adjacent values. Since we later declare matrices on the MATCH $p1$ (and $p2$) values, these gaps will be wasteful. With a scan of the $a.p1$ values (say using a Set), we can rename them consecutively without gaps; these renamed values are found by accessing HashMap<int,int> $p1$ with the original $a.p1$ value.

HashMap<int,int> $p2$. This is the same as the aforementioned $p1$, but with respect to the $a.p2$ values.

MATCH $tbl1$**[][]**. A fundamental data structure to support the `getPrnts` function is the $tbl1$, defined below.

Definition 3: **Max Table w.r.t.** p_1 ($tbl1$): Given the set of all MATCH values $\mathcal{A}$ and PREV pv on $\mathcal{A}$ (with $pv.p1$ and $pv.p2$), the $tbl1[|pv.p1|][|pv.p2|]$ is defined such that each $tbl1[i][j]$ is the $a \in \mathcal{A}$ with the **maximum** $pv.p1.get(a.p1) \leq i$, where $pv.p2.get(a.p2) \leq j$. In the case that multiple such a exist, $tbl1[i][j]$ is the a with the **rightmost** $pv.p2.get(a.p2) \leq j$. If no such a exists, $tbl1[i][j] = null$.

In other words, the $tbl1[i][j]$ stores the "closest" MATCH a with respect to the p_1 values (i.e. we maximize $a.p1$ before $a.p2$). To construct $tbl1$, we first declare

Algorithm 1. Construct the DAG.

```
1  MATCH { int id,p1,p2 }
2  DRCTPRNTS { MATCH m1,m2 }
3  DAG { int gr[][],sz[] }
4  PREV { MATCH tbl1 [][],tbl2 [][]; HashMap<int,int> p1,p2 }
5
6  DAG constructDAG(Set<MATCH> A){
7    int num=A.size(), sz[num]={0,..,0}; MATCH a, a2
8    PREV pv=constructPREV(A)
9    DAG dag={ new int[num][], sz }
10   for each a in A {
11     Set<MATCH> P=getPrnts(pv,L)
12     for each a2 in P {
13       dag.sz[a2.id]++
14       dag.gr[a2.id][dag.sz[a2.id]]=a.id
15     }
16   }return dag
17 }
```

Algorithm 2. Get the direct parent w.r.t. $p1$ or $p2$.

```
1  MATCH getDPrnt(PREV pv, MATCH L, bool wrtS1){
2    return getDPrnt(pv,pv.p1.get(L.p1),pv.p2.get(L.p2),wrtS1)
3  }
4
5  MATCH getDPrnt(PREV pv, int i, int j, bool wrtS1){
6    if(i≤1 ∨ j≤1) return null
7    if(wrtS1) return pv.tbl1[i-1][j-1]
8    else return pv.tbl2[i-1][j-1]
9  }
```

Algorithm 3. Get all parents for c.

```
1  Set<MATCH> getPrnts(PREV pv, MATCH c) {
2    Set<MATCH> P;   int i=pv.p1.get(c.p1), j=pv.p2.get(c.p2)
3    int q, i1, I1, i2, I2, j1, J1, j2, J2
4    MATCH y, dd1, dd2
5    MATCH d1=getDPrnt(pv,c,true), d2=getDPrnt(pv,c,false)
6    if(d1=null ∧ d1=d2) P.add(d1)
7    else if(d1=null){
8      P.add(d1), P.add(d2)
9      i1=d2.p1, I1=pv.p1.get(i1), i2=d1.p1, I2=pv.p1.get(i2)
10     j1=d1.p2, J1=pv.p2.get(j1), j2=d2.p2, J2=pv.p2.get(j2)
11     for(q=I1+1 to I2) {
12       dd1=getDPrnt(pv,q,j,true), dd2=getDPrnt(pv,q,j,false)
13       if(valid(dd1,i1,i2,j1,j2)) P.add(dd1)
14       if(valid(dd2,i1,i2,j1,j2)) P.add(dd2)
15     }for(q=J1+1 to J2) {
16       dd1=getDPrnt(pv,i,q,true), dd2=getDPrnt(pv,i,q,false)
17       if(valid(dd1,i1,i2,j1,j2)) P.add(dd1)
18       if(valid(dd2,i1,i2,j1,j2)) P.add(dd2)
19     }for each y in P {
20       dd1=getDPrnt(pv,y,true), dd2=getDPrnt(pv,y,false)
21       if(P.contains(dd1)) P.remove(dd1)
22       if(P.contains(dd2)) P.remove(dd2)
23     }
24   }return P
25 }
26
27 bool valid(MATCH m, int i1, int i2, int j1, int j2){
28   return (m≠null ∧ i1<m.p1<i1 ∧ j1<m.p2<j2)
29 }
```

the table, $tbl1[|pv.p1|][|pv.p2|]$ and initialize all elements $tbl1[i][j] = null$, signifying that no MATCHes are found. Next, we insert each $a \in \mathcal{A}$ into the list by setting $tbl1[pv.p1.get(a.p1)][pv.p2.get(a.p2)] = a$. Now, each $tbl1[i][j] = null$ needs to be set as the rightmost MATCH m with the maximum $m.p1$ in the subtable $tbl1[1...i][1...j]$. This is easily computed by first moving vertically in $tbl1$ and setting $tbl1[i][j] = tbl1[i-1][j]$ if $tbl1[i][j] = null$ to propagate the maximum values vertically. Finally, we need to move horizontally in $tbl1$ and store in $tbl1[i][j]$ the rightmost $tbl1[i][v]$ $(1 \leq v \leq j)$ with the maximum $tbl1[i][v].p1$. This is done by a left-to-right scan of each row, comparing the adjacent elements, and setting $tbl1[i][v] = tbl1[i][v-1]$ if $tbl1[i][v-1].p1 > tbl1[i][v].p1$.

MATCH $tbl2$**[][]**. The $tbl2$ is the same as $tbl1$ except that we define "closest" to mean that the $a.p2$ value is maximized before the $a.p1$.

Definition 4: **Max Table w.r.t.** p_2 ($tbl2$): Given the set of all MATCH values $\mathcal{A}$ and PREV pv on $\mathcal{A}$ (with $pv.p1$ and $pv.p2$), the $tbl2[|pv.p1|][|pv.p2|]$ is defined such that each $tbl2[i][j]$ is the $a \in \mathcal{A}$ with the **maximum** $pv.p2.get(a.p2) \leq j$, where $pv.p1.get(a.p1) \leq i$. In the case that multiple such a exist, $tbl2[i][j]$ is the a with the **rightmost** $pv.p1.get(a.p1) \leq i$. If no such a exists, $tbl2[i][j] = null$.

The construction of $tbl2$ is the same as $tbl1$, except that in the final horizontal scan, we compare $tbl2[i][v].p2$ and $tbl2[i][v-1].p2$.

In terms of construction time, if we assume that adding and accessing HashMap entries are constant time operations, and the Set is implemented with a HashMap, then the PREV pv on $\mathcal{A}$ from the n-length S_1 and m-length S_2 is constructed in $O(|pv.p1| \times |pv.p2|)$ time. While $pv.p1$ and $pv.p2$ eliminate the gaps between the respective $p1$ and $p2$ values of $\mathcal{A}$, we have $|pv.p1| \in O(n)$ and $|pv.p2| \in O(m)$ in the very worst case.

Theorem 5: Given the n-length S_1 and m-length S_2, and the set of all MATCHes $\mathcal{A}$, PREV pv on $\mathcal{A}$ is constructed in $O(nm)$ time.

2) `getPrnts` *Function:* Given the PREV pv data structure on all MATCHes $\mathcal{A}$, we call `getPrnts`(pv, L) in line 11 of `constructDAG` to retrieve the set of parent MATCHes P of the MATCH $L \in \mathcal{A}$. Recall that these parents P of the MATCH L are all MATCHes that directly precede L in the DAG, i.e. each $p \in P$ is in series with L and no $p, p2 \in P$ are in series with one another. Using pv, we can compute, for any MATCH $c \in \mathcal{A}$, two *direct parents* that are closest to c with respect to the $p1$ and $p2$ values.

Definition 6: **Direct Parents**: Given the PREV pv on the MATCHes in $\mathcal{A}$ between the n-length S_1 and the m-length S_2, and a MATCH $c \in \mathcal{A}$, let $i = pv.p1.get(c.p1)$ and $j = pv.p2.get(c.p2)$. The *direct parent of c w.r.t.* $p1$ is:

$$d1 = \begin{cases} null, \text{ if } i \leq 1 \vee j \leq 1 \vee i > |pv.p1| \vee j > |pv.p2| \\ pv.tbl1[i-1][j-1], \text{ otherwise} \end{cases}$$

The *direct parent of c w.r.t.* $p2$ is:

$$d2 = \begin{cases} null, \text{ if } i \leq 1 \vee j \leq 1 \vee i > |pv.p1| \vee j > |pv.p2| \\ pv.tbl2[i-1][j-1], \text{ otherwise} \end{cases}$$

The first `getDPrnt` in Algorithm 2 implements Definition 6 to return the direct parents for any MATCH say $L \in A$. In cases where we want to find the direct parent for a MATCH at a certain location in the $pv.tbl1$ or $pv.tbl2$, say $pv.tbl1[i][j]$ or $pv.tbl2[i][j]$, we overload `getDPrnt`.

The direct parents computation (`getDPrnt`) is the cornerstone of the `getPrnts` function. The following lemma, implemented in Algorithm 3, proves that the direct parents of c can be used to determine all parents of c.

Lemma 7: Given $\mathcal{A}$, the MATCHes between S_1 and S_2, and a MATCH $c \in \mathcal{A}$, the two direct parents of c can be used to compute the set P with all parents of c.

Proof: Let $d1$ and $d2$ be the direct parents of c (Definition 6). By Definition 3, $d1$ is a direct parent because it directly precedes c with the maximum $p1$ and the rightmost $p2$ value. Similarly by Definition 4, $d2$ is a direct parent of c because it directly precedes c with the maximum $p2$ and the rightmost $p1$ value. To find the remaining parents of c, we now find other MATCHes that precede c, which are also parallel with $d1$ and $d2$. There are three cases.

Case (a). When $d1 = null$, then also $d2 = null$ since there cannot be another MATCH preceding c. Thus, $P = \emptyset$.

Case (b). When $d1 = d2$, the nearest parents to c are the same MATCH. There are only two types of MATCHes that are parallel with $d1$. First, we need to consider all MATCHes, say $m1$, with the same endpoint $m1.p1 = d1.p1$ and $m1.p2 \in \{1, 2, ..., d1.p2 - 1\}$. Second, we need to consider the MATCHes, say $m2$, with the same endpoint $m2.p2 = d1.p2$ and $m2.p2 \in \{1, 2, ..., d1.p1 - 1\}$. In the LCS computation, suppose that we chose, w.l.o.g., $m1$ (with $m1.p2 = d1.p2 - 2$) instead of $d1$. Then, we cannot choose a MATCH $m3$ with $m3.p1 < d1.p1$ and $m3.p2 = d1.p2 - 1$. So, having any $m1$ or $m2$ parallel to $d1$ will only lead to suboptimal CSSs. Thus, only $P = \{d1\}$ is a parent of c.

Case (c). Otherwise, $d1 \neq d2$ and we have two different direct parents of c. Set $P = \{d1, d2\}$. Let us collect the endpoints of $d1$ and $d2$: $i1 = d2.p1$, $i2 = d1.p1$, $j1 = d1.p2$, and $j2 = d2.p2$. What MATCH, say $m3$, is parallel to $d1$ and $d2$? By Definition 6, there cannot be any MATCH $m3$ directly preceding c with endpoints after $i2$ or $j2$. By (b), we do not need to consider other MATCHes with endpoints on either $d1$ or $d2$. So, all the *possible* MATCHes parallel to $d1$ and $d2$ are those with $(m3.p1 \in w \wedge m3.p2 \in x)$, where $w = \{i1 + 1, i1 + 2, ..., i2 - 1\}$ and $x = \{j1 + 1, j1 + 2, ..., j2 - 1\}$. To find such $m3$, we only need to find direct parents (by (b)), say $dd1$ and $dd2$, for a theoretical MATCH m with $(m.p1 \in w \wedge m.p2 = j) \vee (m.p1 = i \wedge m.p2 \in x)$. Then, when we have $i1 < dd1.p1 < i2$ and $j1 < dd1.p2 < j2$, this is a possible MATCH parallel with $d1$ and $d2$, which is also a possible parent of c, so we add $dd1$ to P. We do the same process for $dd2$.

Since we computed all the *possible* parents in P, additional processing on P is needed to ensure that no pair of MATCHes in P are in series; if any are in series, delete the MATCH furthest from c. With the pv and `getDPrnt`, this task is simple. We simply check the direct parents (say $dd1$ and $dd2$) for each $y \in P$, and remove $dd1$ if $dd1 \in P$ and remove $dd2$ if $dd2 \in P$. ■

C. Computing the LCS

Since our dag has a single source S (and all paths end at E), the longest path between S and E, i.e. the LCS, is computed by giving all edges a weight of -1 and finding the shortest path from S to E via a topological sort [21].

D. Complexity Analysis

Our LCS algorithm: (i) finds the length-1 CSSs, (ii) computes the DAG on the CSSs, and (iii) reports the longest DAG path. Here, we analyze the overall time complexity.

Step (i). First, we find (and store in $\mathcal{A}$) the η length-1 CSSs in $O(\max\{n + m, \eta\})$ time by Lemma 2.

Step (ii). We then construct the DAG dag on these $a \in \mathcal{A}$ with `constructDAG`. In `constructDAG`, we initially compute the newly proposed PREV pv data structure in $O(nm)$ time by Theorem 5. After constructing pv, the `computeDAG` iterates through each $a \in \mathcal{A}$ and creates an incoming edge between the parents of a and a. So, `computeDAG` executes in time $O(\max\{nm, \eta \times t_{\texttt{getPrnts}}\})$, where $t_{\texttt{getPrnts}}$ is the time of `getPrnts`. The `getPrnts` running time is in $O((i2 - i1) + (j2 - j1))$, with respect to the local variables $i1$, $i2$, $j1$, and $j2$. However, it may be the case that $i1 = j1 = 1$, $i2 = n$, and $j2 = m$, and so $O(n + m)$ time is required by `getPrnts`. Below we formalize the worst case result and the case for average strings from a uniform distribution.

Lemma 8: For the n-length S_1 and the m-length S_2, the `getPrnts` function requires $O(n + m)$ time.

Lemma 9: For average case strings S_1 and S_2 with symbols uniformly drawn from alphabet Σ, the `getPrnts` function requires $O(|\Sigma|)$ time.

Proof: Since $d1$ and $d2$ are the direct parents of c (see Definitions 3, 4, and 6), and since the uniformness of S_1 and S_2 means that for any symbol say $S_1[s]$ we can find every $\sigma \in \Sigma$ in $S_2[s - \Delta ... s + \Delta]$ with $\Delta \in O(|\Sigma|)$, then $(i2 - i1) \in O(|\Sigma|)$ and $(j2 - j1) \in O(|\Sigma|)$. ■

So, the overall `constructDAG` time follows.

Theorem 10: Given $\mathcal{A}$, the length-1 MATCHes in the n-length S_1 and the m-length S_2, the `constructDAG` requires $O(\max\{nm, \eta \times \max\{n, m\}\})$ time in the worst case and $O(\max\{nm, \eta \times |\Sigma|\})$ on average.

Step (iii). We find the LCS with a topological sort in time linear to the dag size [21], which cannot require more time than that needed to build the dag (see Theorem 10).

Overall, (i) and (iii) do not add to the complexity of (ii).

Theorem 11: The LCS between the n-length S_1 and the m-length S_2 can be computed in $O(\max\{nm, \eta \times \max\{n, m\}\})$ time in the worst case and $O(\max\{nm, \eta \times |\Sigma|\})$ on average.

IV. Compressing Resequencing Data

When data is released, modified, and re-released over a period of time, a large amount of commonality exists between these releases. Rather than maintaining all uncompressed versions of the data, it is possible to keep one uncompressed version, say D, and compress all future versions D_i with respect to D. We refer to D_i as

Table I
Arabidopsis thaliana GENOME: RESULTS (IN BYTES) FOR COMPRESSING CHROMOSOME U INTO C, WHERE $\mathbb{L}$ AND $\mathbb{P}$ RESPECTIVELY REPRESENT LZMA2 AND PPMD FROM 7-ZIP.

U	$\|U\|$	Our Scheme			GRS	GReEn
		$\|C\|$	$\|\mathbb{L}(C)\|$	$\|\mathbb{P}(C)\|$	[13]	[14]
1	30 427 671	1 086	963	1 037	**715**	1 551
2	19 698 289	504	584	605	**385**	937
3	23 459 830	**746**	759	803	2 989	1 097
4	18 585 056	4 555	2 507	3 156	**1 951**	2 356
5	26 975 502	**433**	502	520	604	618
Sum	119 146 348	7 324	**5 315**	6 121	6 644	6 559

Table II
Homo sapiens GENOME: RESULTS (IN BYTES) FOR COMPRESSING CHROMOSOME U INTO C.

U	$\|U\|$	Our Scheme	GRS	GReEn
		$\|C\|$	[13]	[14]
1	247 249 719	**381 577**	1 336 626	1 225 767
2	242 951 149	**356 526**	1 354 059	1 272 105
3	199 501 827	**284 096**	1 011 124	971 527
4	191 273 063	**330 381**	1 139 225	1 074 357
5	180 857 866	**259 922**	988 070	947 378
6	170 899 992	**265 222**	906 116	865 448
7	158 821 424	**292 797**	1 096 646	998 482
8	146 274 826	**222 972**	764 313	729 362
9	140 273 252	**309 512**	864 222	773 716
10	135 374 737	**245 264**	768 364	717 305
11	134 452 384	**222 735**	755 708	716 301
12	132 349 534	**214 123**	702 040	668 455
13	114 142 980	**148 938**	520 598	490 888
14	106 368 585	**141 128**	484 791	451 018
15	100 338 915	**138 219**	496 215	453 301
16	88 827 254	**151 606**	567 989	510 254
17	78 774 742	**136 168**	505 979	464 324
18	76 117 153	**113 469**	408 529	378 420
19	63 811 651	**130 468**	399 807	369 388
20	62 435 964	**94 273**	282 628	266 562
21	46 944 323	**71 121**	226 549	203 036
22	49 691 432	**81 329**	262 443	230 049
M	16 571	**64**	183	127
X	154 913 754	**523 282**	3 231 776	2 712 153
Y	57 772 954	**152 464**	592 791	481 307
Sum	3 080 436 051	**5 267 656**	19 666 791	17 971 030

the *target* and D as the *reference*. This idea is used to compress resequencing data in [13], [14], primarily using the LCS. The LCS, however, has two core problems with respect to compression. For very similar sequences, the LCS computation time is almost quadratic, or worse, potentially leading to long compression time. Secondly, the LCS may not always lead to the best compression, especially when some CSS components are very short.

Rather than focusing on the LCS, we consider the maximal CSSs that make up the common subsequences. To intelligently choose which of the CSS's are likely to lead to improved compression, we use the longest previous factor (LPF). Consider compressing the target T with respect to the reference R; let $Z = R \circ T$. Suppose we choose exactly $|T|$ maximal length CSSs, specifically, for $\beta = Z[i...|Z|]$ we have $\alpha = Z[h...|Z|]$ such that (1) CSSs $\alpha[1...k] = \beta[1...k]$ and (2) this is the maximal k for $h < i$, where $|R| + 1 \leq i \leq |Z|$. These ks are computed in the LPF data structure on Z at $LPF[i] = k$ and the position of this CSS is at $POS[i] = h$ [18]. (Note that LPF and POS are constructed in linear time [18], [19], [20].) The requirement that $h < i$ suits dictionary compression and compressing resequencing data because the CSS beginning at i is compressed by referencing the same CSS at h, occurring earlier in target T or anywhere in the reference R. Our idea is to use the LPF and POS to *represent* or *encode* CSSs that make up the target T with tuples. We will then compress these tuples with standard compression schemes.

Our Compression Scheme. We now propose a reference-based compression scheme which scans the LPF and POS on Z in a left-to-right fashion to compress T with respect to R. This scheme is similar to the LZ factorization [18], but differs in how we will encode the CSSs. Our contribution here is (1) using two files to compress T, (2) only encoding CSSs with length at least k, and (3) further compressing the resulting files with standard compression schemes.

Initially, the two output files, $triples$ and $symbols$, are empty. Let $i = |R| + 1$.

(*) If $LPF[i] < k$, we simply encode the symbol; append the (say 1-byte) char $T[i - |R|]$ to $symbols$ and increment i. Otherwise $LPF[i] \geq k$, so we will encode this CSS with the triple (pT, pZ, l), where $pT = i - |R|$ is the starting position of the CSS in T, $pZ = POS[i]$ is the starting position of the CSS in $Z[1...i-1]$, and $l = LPF[i]$ is the length of the CSS. We write three long (say 4-byte) words pT, pZ, and l to $triples$. Since the triple encodes an l-length CSS, we set $i = i + l$ to consider compressing the suffix following the currently encoded CSS. Lastly, if $i \leq |Z|$, continue to (*).

The resulting files $triples$ and $symbols$ are binary sequences that can be further compressed with standard compression schemes (so, decompression will start by first reversing this process). The purpose of the k and the two files (one with byte symbols and one with long triples) is to introduce flexibility into the system and encode CSSs with triples (12 bytes) only when beneficial and otherwise, encode a symbol with a byte. For convenience, our implementation encodes each symbol with a byte, but we acknowledge that it is possible to work at the bit-level for small alphabets.

The decompression is also a left-to-right scan. Let $i = 1$ and point to the beginning of $triples$ and $symbols$.

(†) Consider the current long word w_1 in $triples$. According to the triple encoding, this will be the position of the CSS in T. If $i = w_1$, then we pick up the next two long words w_2 and w_3 in $triples$. We now know $T[i...i+w_3-1] = Z[w_2...w_2+w_3-1]$. Since we only have access to R and $T[1...i-1]$, then we pick up each symbol of $Z[w_2...w_2+w_3-1]$ by picking up $R[j]$ if $j \leq |R|$ and picking up $T[j-|R|]$ otherwise, for $w_2 \leq j \leq w_2+w_3-1$. We next will consider $i = i+w_3$. Else $i \neq w_1$, so we pick up the next char c in $symbols$ since $T[i] = c$; we next consider i++. If $i \leq |T|$, go to (†).

Compression Results. We implemented the aforementioned compression scheme and ran our program to com-

press, like [13], [14], the *Arabidopsis thaliana* genome chromosomes in TAIR9 (target) with respect to TAIR8 (reference). For chromosome 1, we found that $k = 31$ performs best; we used this same k for all chromosomes. In Table I, we display the compression results. We see that all of our results are competitive with the GRS and GReEn systems, except for chromosome 4, which has the smallest average CSS length of about 326K. Nonetheless, we are able to further compress our results with compression schemes in 7-zip to achieve better compression than GRS and GReEn.

In Table II, we show results for the *Homo sapiens* genome compression (with $k = 31$), using KOREF_20090224 as the target and KOREF_20090131 as the reference. All of our results are better than GRS and GReEn. Note that these results can be further improved by applying 7-zip as in Table I. Theoretically, our compression scheme requires time linear in the uncompressed text length, since we perform one scan of the LPF, which is constructed in linear time via the suffix array SA [18]. We ran our programs in an AWS EC2 m4.4xlarge environment. For the larger chromosomes from the *Homo sapiens* genome, the SA construction required 2,376 seconds and the LPF construction required 399 seconds. Depending on the application, the SA and LPF may already be available. Given the LPF, our compression and decompression algorithms completed in less than one second. Our future plan includes using more efficient SA and LPF constructions.

V. Conclusion

We proposed a new algorithm to compute the LCS. Motivated by our algorithm, we introduced a new reference-based compression scheme for genome resequencing data using the LPF. For the *Arabidopsis thaliana* genome (originally 119,146,348 bytes), our scheme compressed the genome to 5,315 bytes, an improvement over the best performing state-of-the-art methods (6,644 bytes [13] and 6,559 bytes [14]). For the *Homo sapiens* genome (originally 3,080,436,051 bytes), our scheme compressed the genome to 5,267,656 bytes, an improvement over the 19,666,791 bytes and 17,971,030 bytes achieved in [13] and [14], respectively.

References

[1] D. Gusfield, *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[2] D. Adjeroh, T. Bell, and A. Mukherjee, *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*, 1st ed. Springer, 2008.

[3] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195–197, 1981.

[4] J. Aach, M. Bulyk, G. Church, J. Comander, A. Derti, and J. Shendure, "Computational comparison of two draft sequences of the human genome," *Nature*, vol. 26, no. 1, pp. 5–14, 2001.

[5] S. Wandelt and U. Leser, "Fresco: Referential compression of highly similar sequences," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 10, no. 5, pp. 1275–1288, Sep. 2013.

[6] R. Giancarlo, D. Scaturro, and F. Utro, "Textual data compression in computational biology: Algorithmic techniques," *Computer Science Review*, vol. 6, no. 1, pp. 1–25, 2012.

[7] C.-E. Kuo, Y.-L. Wang, J.-J. Liu, and M.-T. Ko, "Resequencing a set of strings based on a target string," *Algorithmica*, vol. 72, no. 2, pp. 430–449, Jun. 2015.

[8] E. W. Myers, "An $O(ND)$ difference algorithm and its variations." *Algorithmica*, vol. 1, no. 2, pp. 251–266, 1986.

[9] E. Ukkonen, "Algorithms for approximate string matching," *Inform and Control*, vol. 64, pp. 100–118, 1985.

[10] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest subsequences." *Commun. ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[11] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Commun. ACM*, vol. 18, no. 6, pp. 341–343, Jun. 1975.

[12] J. Yang, Y. Xu, Y. Shang, and G. Chen, "A space-bounded anytime algorithm for the multiple longest common subsequence problem," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 11, pp. 2599–2609, 2014.

[13] C. Wang and D. Zhang, "A novel compression tool for efficient storage of genome resequencing data," *Nucleic Acids Res.*, vol. 39, no. 4, 2011.

[14] A. J. Pinho, D. Pratas, and S. P. Garcia, "GReEn: A tool for efficient compression of genome resequencing data," *Nucleic Acids Research*, vol. 40, no. 4, 2012.

[15] D. Adjeroh and F. Nan, "On compressibility of protein sequences." in *DCC*. IEEE Computer Society, 2006, pp. 422–434.

[16] M. Fritz, R. Leinonen, G. Cochrane, and E. Birney, "Efficient storage of high throughput DNA sequencing data using reference-based compression," *Genome Research*, vol. 21, pp. 734–40, 2011.

[17] F. Hach, I. Numanagic, C. Alkan, and S. C. Sahinalp, "Scalce: boosting sequence compression algorithms using locally consistent encoding." *Bioinformatics*, vol. 28, no. 23, pp. 3051–3057, 2012.

[18] M. Crochemore and L. Ilie, "Computing longest previous factor in linear time and applications," *Information Processing Letters*, vol. 106, no. 2, pp. 75 – 80, 2008.

[19] R. Beal and D. Adjeroh, "Parameterized longest previous factor," *Theoretical Computer Science*, vol. 437, pp. 21 – 34, 2012.

[20] ——, "Variations of the parameterized longest previous factor," *Journal of Discrete Algorithms*, vol. 16, pp. 129 – 150, 2012.

[21] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill, 2001.