2016

A New Algorithm for “the LCS problem” with Application in Compressing Genome Resequencing Data

Richard Beal
West Virginia University

Tazin Afrin
West Virginia University

Aliya Farheen
West Virginia University

Donald Adjeroh
West Virginia University

The Author(s). *BMC Genomics* 2016, **17**(Suppl 4):544
DOI 10.1186/s12864-016-2793-0

BMC Genomics

RESEARCH Open Access

A new algorithm for "the *LCS* problem" with application in compressing genome resequencing data

Richard Beal*, Tazin Afrin, Aliya Farheen and Donald Adjeroh

From IEEE International Conference on Bioinformatics and Biomedicine 2015
Washington, DC, USA. 9-12 November 2015

Abstract

Background: 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.

Methods: First, we present a new algorithm for the LCS problem. Using the generalized suffix tree, we identify the common substrings shared between the two input sequences. Using the maximal common substrings, we construct a directed acyclic graph (DAG), based on which we determine the LCS as the longest path in the DAG. Then, we introduce an LCS-motivated reference-based compression scheme using the components of the LCS, rather than the LCS itself.

Results: Our basic scheme compressed the *Homo sapiens* genome (with an original size of 3,080,436,051 bytes) to 15,460,478 bytes. An improvement on the basic method further reduced this to 8,556,708 bytes, or an overall compression ratio of 360. This can be compared to the previous state-of-the-art compression ratios of 157 (Wang and Zhang, 2011) and 171 (Pinho, Pratas, and Garcia, 2011).

Conclusion: We propose a new algorithm to address the longest common subsequence problem. Motivated by our LCS algorithm, we introduce a new reference-based compression scheme for genome resequencing data. Comparative results against state-of-the-art reference-based compression algorithms demonstrate the performance of the proposed method.

Keywords: Longest common subsequence, LCS, Longest previous factor, LPF, Compression, Biology, Genome resequencing

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 *LCS*s: `abb` and `aba`. The *LCS* has been used to study various areas (see [2, 3]), such as text analysis, pattern recognition, file comparison, efficient tree matching [4], etc. Biological applications of the *LCS* and similarity measurement are varied, from sequence alignment [5] in comparative genomics [6], to phylogenetic construction and analysis, to rapid search in huge biological sequences [7], to compression and efficient storage of the rapidly expanding genomic data sets [8, 9], to re-sequencing a set of strings given a target string [10], an important step in efficient genome assembly.

The basic approach to compute the *LCS*, between the n-length S_1 and m-length S_2, is via dynamic programming.

*Correspondence: r.beal@computer.org
A preliminary version of this paper was presented at IEEE BIBM'15, see [1].
Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV, USA

Using *LCS* to denote the dynamic programming (DP) table, the basic formulation is as follows, given $0 \leq i \leq n$ and $0 \leq j \leq m$:

$$LCS(i,j) = \begin{cases} 0, \textbf{if } i = 0 \ \vee \ j = 0 \\ 1 + LCS(i-1, j-1), \ \textbf{if } S_1[i] = S_2[j] \\ \max\left\{LCS(i, j-1), LCS(i-1, j)\right\}, \\ \qquad \textbf{if } S_1[i] \neq S_2[j] \end{cases}$$

The above computes the length of the *LCS* in the last position of the table ($LCS(n,m)$). As with the edit distance computation, the actual string forming the *LCS* can be obtained by using a trace back on the DP table. This requires $O(nm)$ time and $O(nm)$ space. The *LCS* matrix has some interesting properties: the entries in any row or in any column are monotonically increasing, and between any two consecutive entries in any row or column, the difference is either 0 or 1. An example *LCS* matrix and trace are shown in Fig. 1.

Alternatively, we can formulate the problem as a two-dimensional grid, where the goal is to find the minimal cost (or maximal cost, depending on the formulation) path, from the start position on the grid (typically, (0,0)), to the end position (n,m). Myers et al. [11] and Ukkonen [12] used this idea to propose a minimum cost path determination problem on the grid, where the path takes a diagonal line from $(i-1, j-1)$ to (i,j) if $S_1[i] = S_2[j]$ with cost 0, and takes a horizontal or vertical line with a cost of 1, corresponding respectively to insert or delete operations. Hunt and Szymanski [13] 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 ($S_1[i] = S_2[j]$). When two non-similar files are compared, we will have $r \ll nm$, or r in $O(n)$, leading to a practical $O(n \log n)$ time algorithm. However, for very similar files, we have $r \approx nm$, or an $O(nm \log n)$ algorithm. This worst-case occurs, for instance, when $S_1 = a^n$ and $S_2 = a^m$.

Hirschberg [14] proposed space-efficient approaches to compute the *LCS* using DP in $O(nm)$ time and $O(n+m)$ space, rather than $O(nm)$. More recently, Yang et al. [15] used the observation on monotonically increasing values in the *LCS* table to identify the "corner points", where the values on the diagonals change from one row to the next. The corners define a more sparse 2D grid, based on which they determine the *LCS*.

	j	0	1	2	3	4	5	6	7	8
i		∅	**A**	**G**	**G**	**T**	**C**	**G**	**T**	**A**
0	∅	0	0	0	0	0	0	0	0	0
1	**A**	0	1	1	1	1	1	1	1	1
2	**A**	0	1	1	1	1	1	1	1	2
3	**C**	0	1	1	1	1	2	2	2	2
4	**C**	0	1	1	1	1	2	2	2	2
5	**T**	0	1	1	1	2	2	2	3	3
6	**T**	0	1	1	1	2	2	2	3	3
7	**A**	0	1	1	1	2	2	2	3	4
8	**A**	0	1	1	1	2	2	2	3	4

Fig. 1 *LCS* dynamic programming table for $S_1 = AACCTTAA$ and $S_2 = AGGTCGTA$. A sample *LCS* trace (*ACTA*) is highlighted

A generalization of the *LCS* problem is to find the *LCS* for a set of two or more sequences. This is the multiple longest common subsequence problem, which is known to be NP-hard for an arbitrary number of sequences [16]. Another interesting view of the *LCS* problem is in terms of the longest increasing subsequence (*LIS*) problem, suggested earlier in [17–19], and described in detail in [2]. The *LIS* approach also solves the *LCS* problem in $O(r \log n)$ time (where $m \leq n$). In most practical scenarios, $r < nm$.

The *LCS* has been used in some recent algorithms to compress genome resequencing data [20, 21]. Compression of biological sequences is an important and difficult problem, which has been studied for decades by various authors [22–24]. See [9, 25, 26] for recent surveys. Most of the earlier studies focused on lossless compression because it was believed that biological sequences should not admit any data loss, since that would impact later use of the compressed data. The earlier methods also generally exploited self-contained redundancies, without using a reference sequence. The advent of high-throughput next generation sequencing, with massive datasets that are easily generated for one experiment, have challenged both compression paradigms.

Lossy compression of high-throughput sequences admitting limited errors have been proposed in [27, 28] for significant compression. With the compilation of several reference genomes for different species, more recent methods have considered lossless compression of re-sequencing data by exploiting the significant redundancies between the genomes of related species. This observation is the basis of various recently proposed methods for reference-based lossless compression [20, 21], whereby some available standard reference genome is used as the dictionary. Compression ratios in the order of 80 to 18,000 without loss have been reported [20, 21]. 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 [29–31].

Methods

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 $T[1 \ldots i]$ and a suffix is $T[i \ldots |T|]$, 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 [3]) 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 $T[i \ldots n]$ 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 $T[i \ldots j]$, 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 $\mathcal{M} = \{m_1, \ldots, m_M\}$, 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 $1 \leq i < M$.

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.

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.

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 *id*s 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 $dag.gr[N.id][1 \ldots dag.sz[N.id]]$. By setting $dag.sz = \{0, \ldots, 0\}$, 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.

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 *tbl*1[][]. A fundamental data structure to support the `getPrnts` function is the *tbl*1, defined below.

Definition 3. **Max Table w.r.t.** p_1 (*tbl*1): 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 $tbl1[i][\mathrm{j}]$ stores the "closest" MATCH a with respect to the p_1 values (i.e. we maximize $a.p1$ before $a.p2$). To construct *tbl*1, we first declare 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 *tbl*1 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 *tbl*1 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 *tbl*2[][]. The *tbl*2 is the same as *tbl*1 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 (*tbl*2): 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 *tbl*2 is the same as *tbl*1, except that in the final horizontal scan, we compare $tbl2[i][v].p2$ and $tbl2[i][v-1].p2$.

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[][];
5         HashMap<int,int> p1,p2 }
6  DAG constructDAG(Set<MATCH> A){
7      int num=A.size(), sz[num]={0,...,0}
8      PREV pv=constructPREV(A); MATCH a, a2
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),
3                    pv.p2.get(L.p2),wrtS1)
4  }
5
6  MATCH getDPrnt(PREV pv, int i, int j,
7                 bool wrtS1){
8    if(i≤1 ∨ j≤1) return null
9    if(wrtS1) return pv.tbl1[i-1][j-1]
10   else return pv.tbl2[i-1][j-1]
11 }
```

Algorithm 3. Get all parents for c.

```
1  Set<MATCH> getPrnts(PREV pv, MATCH c) {
2    Set<MATCH> P
3    int i=pv.p1.get(c.p1), j=pv.p2.get(c.p2)
4    int q, i1, I1, i2, I2, j1, J1, j2, J2
5    MATCH y, dd1, dd2
6    MATCH d1=getDPrnt(pv,c,true),
7          d2=getDPrnt(pv,c,false)
8    if(d1=null ∧ d1=d2) P.add(d1)
9    else if(d1=null){
10     P.add(d1), P.add(d2)
11     i1=d2.p1, I1=pv.p1.get(i1)
12     i2=d1.p1, I2=pv.p1.get(i2)
13     j1=d1.p2, J1=pv.p2.get(j1)
14     j2=d2.p2, J2=pv.p2.get(j2)
15     for(q=I1+1 to I2) {
16       dd1=getDPrnt(pv,q,j,true)
17       dd2=getDPrnt(pv,q,j,false)
18       if(valid(dd1,i1,i2,j1,j2)) P.add(dd1)
19       if(valid(dd2,i1,i2,j1,j2)) P.add(dd2)
20     }for(q=J1+1 to J2) {
21       dd1=getDPrnt(pv,i,q,true)
22       dd2=getDPrnt(pv,i,q,false)
23       if(valid(dd1,i1,i2,j1,j2)) P.add(dd1)
24       if(valid(dd2,i1,i2,j1,j2)) P.add(dd2)
25     }for each y in P {
26       dd1=getDPrnt(pv,y,true)
27       dd2=getDPrnt(pv,y,false)
28       if(P.contains(dd1)) P.remove(dd1)
29       if(P.contains(dd2)) P.remove(dd2)
30     }
31   }return P
32 }
33
34 bool valid(MATCH m, int i1, int i2,
35            int j1, int j2){
36   return (m≠null ∧ i1<m.p1<i1 ∧ j1<m.p2<j2)
37 }
```

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.

`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, $P = \emptyset$.

Algorithm 4. Compression of target T (given reference R, parameter k, and LPF and POS arrays on $Z = R \circ T$ with $z = |Z|$) into $symbols$ file (with filename $symbols\_fn$) and $triples$ file (with filename $triples\_fn$).

```
1  void compress(String T,String R,int k,
2                int LPF[z],int POS[z],
3                String symbols_fn,
4                String triples_fn){
5    int i,posT,posZ,len,rlen=|R|
6    File* symbols=open(symbols_fn,WRITE)
7    File* triples=open(triples_fn,WRITE)
8    for i=rlen+1 to z {
9      posT=i-rlen
10     len=LPF[i]
11     if(len<k){
12       write_char(symbols,T[posT])
13     }else{
14       write_int(triples,posT)
15       posZ=POS[i]
16       write_int(triples,posZ)
17       write_int(triples,len)
18       i=i+len-1
19     }
20   }close(symbols)
21   close(triples)
22 }
```

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 $m1.p2 \in \{1, 2, \ldots, d1.p2 - 1\}$. Second, we need to consider the MATCHes, say $m2$, with the same endpoint $m2.p2 = d1.p2$ and $m2.p2 \in \{1, 2, \ldots, d1.p1 - 1\}$. 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

Algorithm 5. Decompressing target T using reference R, $symbols$ file (with filename $symbols\_fn$), and $triples$ file (with filename $triples\_fn$).

```
1  String decompress(String R,String symbols_fn,
2                    String triples_fn){
3    String T
4    int s=1,t=1,i=1,j,begins,len,loc,rlen=|R|
5    int slen=num_bytes(symbols_fn)/sizeof(int)
6    int tlen=num_bytes(triples_fn)/sizeof(char)
7    File* symbols=open(symbols_fn,READ)
8    File* triples=open(triples_fn,READ)
9    bool more=true
10   while(more){
11     loc=peek_at_int(triples)
12     if(t≤tlen ∧ i=loc){
13       get_next_int(triples) // discard
14       begins=get_next_int(triples)
15       len=get_next_int(triples)
16       t+=3
17       for j=begins to begins+len−1 {
18         i++
19         if(j>rlen) T+=T[j−rlen]
20         else T+=R[j]
21       }
22     }else if(s≤slen){
23       i++
24       s++
25       T+=get_next_char(symbols)
26     }else more=false
27   }close(symbols)
28   close(triples)
29   return T
30 }
```

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 $w = \{i1+1, i1+2, \ldots, i2-1\}$ and $x = \{j1+1, j1+2, \ldots, j2-1\}$. 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 $dd2 \in P$. □

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 [32].

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 $S_2[s-\Delta \ldots s+\Delta]$ 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 [32], which cannot require more time than that needed to build the *dag* (see Theorem 10).

Summary

Overall, (i) and (iii) do not add to the complexity of (ii). Given the above, the overall running time is as follows.

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.

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 the *target* and D as the *reference*. This idea is used to compress resequencing data in [20, 21], 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 CSSs are likely to lead to improved compression, we use the longest previous factor (*LPF*), an important data structure in text compression [33]. 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 $\beta = Z[i \ldots |Z|]$ we have $\alpha = Z[h \ldots |Z|]$ such that (1) CSSs $\alpha[1 \ldots k] = \beta[1 \ldots k]$ 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 $POS[i] = h$ [29]. (Note that *LPF* and *POS* are constructed in linear time [29–31].) 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 [29], 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 $Z[1 \ldots i - 1]$, and $l = LPF[i]$ is the length of the CSS. We write three long (say 4-byte integer) 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

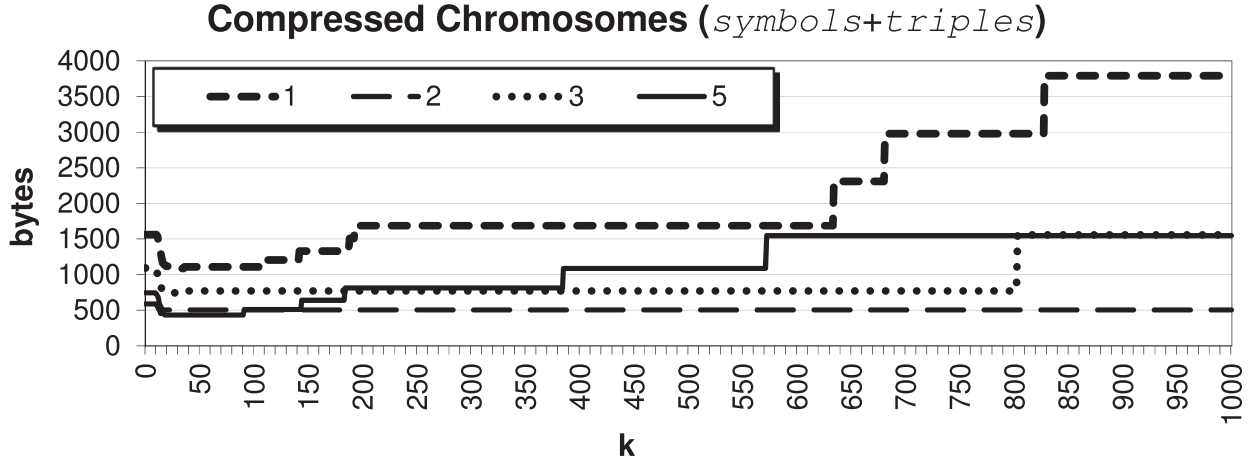

Fig. 2 Total bytes needed by our algorithm to compress the *Arabidopsis thaliana* genome, i.e. file size sum of *symbols* and *triples*

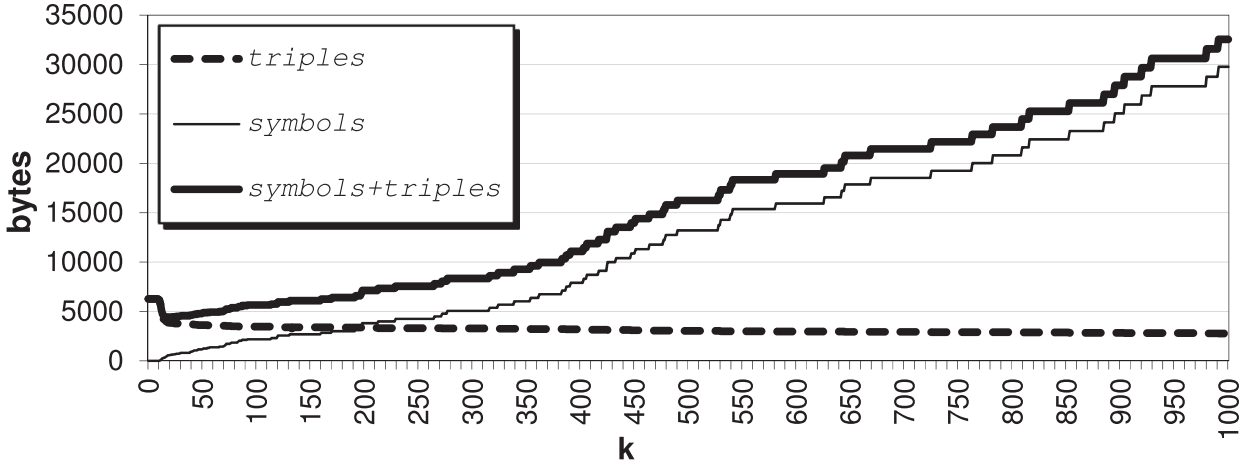

Fig. 3 Compressing the *Arabidopsis thaliana* genome Chromosome 4

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 $T[i \ldots i + w_3 - 1] = Z[w_2 \ldots w_2 + w_3 - 1]$. Since we only have access to R and $T[1 \ldots i - 1]$, then we pick up each symbol of $Z[w_2 \ldots w_2 + w_3 - 1]$ 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 (†).

The compression and decompression algorithms are detailed in Algorithms 4 and 5, respectively.

Results and discussion

We implemented the previously described compression scheme, selected and fixed parameter k, and ran our program to compress various DNA corpora. In this section, we describe the selection of k and our final results.

Choosing parameter k

Recall that the parameter k is a type of threshold used by our compression scheme to determine whether it is more beneficial to encode a symbol verbatim (that is, 1 byte) or encode a CSS as a triple (that is, 12 bytes). Specifically, our compression algorithm works on the *LPF* (which represents the CSSs of the n-length T) in a left-to-right fashion, selecting the leftmost CSS, say $T[i \ldots i + l - 1]$ of length-($LPF[i] = l$), and determining whether to encode that CSS as a triple [and then consider the next CSS ($T[i + l \ldots i + l + LPF[i + l] - 1]$ of length-$LPF[i + l]$)], or encode the first symbol ($T[i]$) [and then consider the next CSS ($T[i + 1 \ldots i + LPF[i + 1]]$ of length-$LPF[i + 1]$)].

Obviously, it is better to encode a length-($l = 1$) CSS with a 1-byte symbol, rather than a 12-byte triple. It is clearly the case that for any CSS length $1 \leq l < 12$, it is better to encode the first symbol with 1-byte and take a *chance* that the next CSS to the right will be significantly

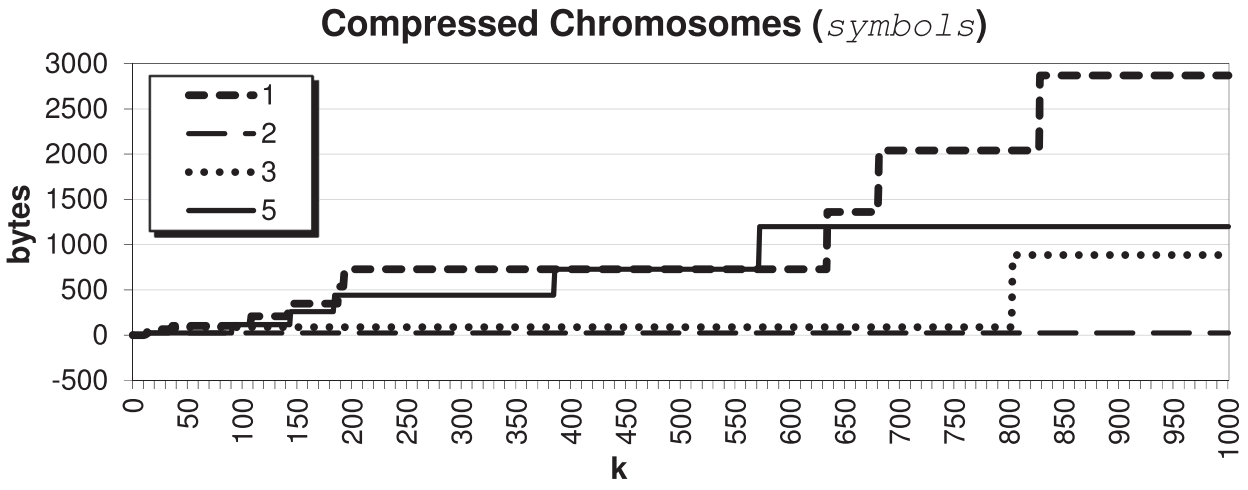

Fig. 4 Size of the *symbols* file when compressing the *Arabidopsis thaliana* genome

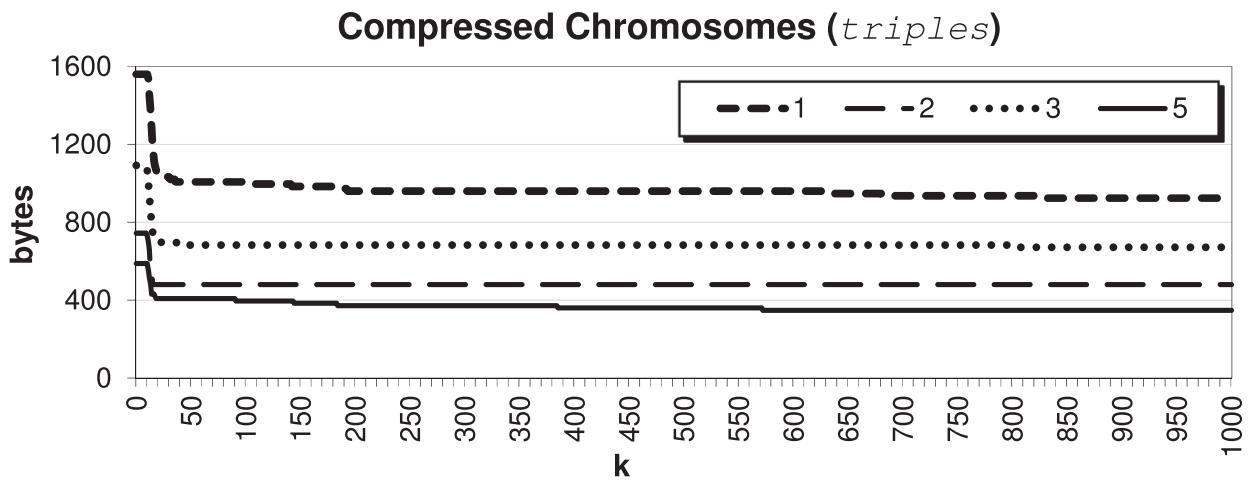

Fig. 5 Size of the *triples* file when compressing the *Arabidopsis thaliana* genome

larger. Why can we afford to take this *chance*? One *LPF* property, which also allows for an efficient construction of the data structure (see [29]), is that $LPF[i+1] \geq LPF[i]-1$. That is, if we pass up on encoding the CSS at i of length-($LPF[i] = l$) as a triple, we can encode $T[i]$ as a symbol and (1) are guaranteed that there is at least a length-$(l-1)$ CSS with a prefix of $T[i+1 \ldots n]$ and (2) the longest CSS common to a prefix of $T[i+1 \ldots n]$ is of length-$LPF[i+1]$, maybe even larger than $LPF[i]$. Clearly, we want to encode most CSSs as triples to take advantage of the concise triple representation. Now, the question becomes: how large should we set k, such that we can afford to take a risk passing up length-($l < k$) CSSs in hopes of finding even larger CSSs better suited as triples?

For this paper, we decided to select k by studying the impact of the parameter on our compressed results for the *Arabidopsis thaliana* genome, using target TAIR9 and reference TAIR8. The compression results for various k are shown in Fig. 2; since chromosome 4 does not compress as well as the others, we show it separately in Fig. 3 for improved visualization. For very small $k < 12$, we have a result that basically encodes with triples only; when $k = 1$, we are exclusively encoding CSSs as triples. We see that when k is roughly between 12 and 35, we are encouraging the algorithm to pass up encoding smaller CSSs as triples, which leads to the best compression result. The results stay competitive until say $k \geq 100$, where the algorithm becomes *too optimistic* and passes up the opportunity to encode smaller CSSs as triples in hopes that larger CSSs will exist. Further, we see from Fig. 4 that as k becomes large, it indeed becomes very expensive to pass up encoding these CSSs as triples. Also, we see from Fig. 5 that beyond say $k = 20$, there is minimal compression savings. Thus, we want to balance the expensive *symbols* files with the space-savings from the *triples* files.

In Table 1, we show the best compression results and k for the *Arabidopsis thaliana* genome. Unless otherwise specified, our experiments below fix parameter k as 31, since it is the optimal k common to 4-of-5 of the *Arabidopsis thaliana* chromosomes and gives a competitive result for the remaining chromosome. This result follows intuition because k should be at least 11 and not too large, so that we can consider CSSs that are worthy of encoding.

Compression results

Like [20, 21], we compress the *Arabidopsis thaliana* genome chromosomes in TAIR9 (target) with respect to TAIR8 (reference). In Table 2, 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 using compression schemes from 7-zip, with $\mathbb{L}$ and $\mathbb{P}$ respectively representing `lzma2` and `ppmd`, to achieve even better compression.

In Table 3, we show results for compressing the genome *Oryza sativa* using the target TIGR6.0 and reference TIGR5.0. After compressing our algorithm's output with `lzma2` or `ppmd`, our results are better than both GRS

Table 1 *Arabidopsis thaliana* genome: Optimal k for compressing chromosome U into the smallest C (in bytes)

U	k	$\|C\|$
1	31–35	1086
2	16–1578	504
3	24–39	746
4	18	4418
5	19–91	433

Table 2 *Arabidopsis thaliana* genome: Results (in bytes) for compressing chromosome *U* into *C*

U	$\|U\|$	Our Scheme			GRS	GReEn
		$\|C\|$	$\|\mathbb{L}(C)\|$	$\|\mathbb{P}(C)\|$	[20]	[21]
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

Bold signifies the best result

[20] and GReEn [21]. Note that for each of the chromosomes 6, 9, and 12, our algorithm's output is 12 bytes, better than that of GRS [20] (14 bytes) and GReEn [21] (482 bytes, 366 bytes, and 429 bytes respectively). When we compress our result with `lzma2` or `ppmd`, the result is bloated since more bytes are needed. So, we can further improve the overall result by not compressing chromosomes 6, 9, and 12, and further, selecting the best such compression scheme for each individual chromosome. We acknowledge that additional bits would need to be encoded to determine which compression scheme was selected.

In Table 4, we show the compression results for the *Homo sapiens* genome, using KOREF_20090224 as the target and KOREF_20090131 as the reference. After compressing our computed *symbols* and *triples* files with `lzma2`, we see that most all of our results are better than GRS and GReEn. Recall in previous experiments that sometimes secondary compression with 7-zip does not improve the initial compression achieved by our proposed algorithm. For this genome, we exercise the flexibility of our compression framework. In Table 4, (*) indicates that the *M* chromosome was not further compressed with `lzma2` due to the aforementioned reason. To indicate that *M* was not compressed, we will simply encode a length-25 bitstring (say 4-byte) header to specify whether or not the `lzma2` was applied. There is no need to encode *k* in the header since it is a fixed value. Thus, the overall compressed files require 15,460,478 bytes, which is only slightly better than GRS and GReEn.

To improve this result, we exploit the difference between the *Homo sapiens* genome and those discussed earlier. That is, the *Homo sapiens* genome uses the extended alphabet {A, C, G, K, M, R, S, T, W, Y, a, c, g, k, m, n, r, s, t, w, y}. The observation is that, the alphabet size decreases roughly in half by converting to one character-case. Such a significant reduction in alphabet size will yield more significant redundancies identified by our compression algorithm. Our new *decomposition* method will *decompose* each chromosome into two parts: (1) the *payload* (ρ), representing the chromosome in one character-case, and (2) the *character-case bitstring* (α), in which each bit records whether the corresponding position in the target was an upper-case character. Next, we use our previously proposed algorithm to compress ρ into C^{ρ} and α into C^{α}.

Table 5 shows compression via decomposition for the *Homo sapiens* genome. Note that the $|C^{\rho}|$, i.e. compressed payload, column corresponds to the results reported in our initial work [1]. We observe that in various scenarios, the character-case of the alphabet

Table 3 *Oryza sativa* genome: Results (in bytes) for compressing chromosome *U* into *C*

U	$\|U\|$	Our Scheme			GRS	GReEn
		$\|C\|$	$\|\mathbb{L}(C)\|$	$\|\mathbb{P}(C)\|$	[20]	[21]
1	43 268 879	15 207	4 735	**4 551**	1 502 040	4 972
2	35 930 381	4 645	1 649	1 517	**1 409**	1 906
3	36 406 689	54 234	15 693	**15 556**	47 764	17 890
4	35 278 225	21 474	6 636	**6 432**	36 145	6 750
5	29 894 789	17 030	5 431	**5 359**	6 177	5 539
6	31 246 789	**12**	146	141	14	482
7	29 696 629	5 899	2 064	**1 972**	4 067	2 448
8	28 439 308	23 126	**8 794**	10 115	118 246	9 507
9	23 011 239	**12**	146	141	14	366
10	23 134 759	175 228	**49 713**	50 277	788 542	60 449
11	28 512 666	41 407	**13 006**	13 351	2 397 470	14 797
12	27 497 214	**12**	146	141	14	429
Sum	372 317 567	358 286	**108 159**	109 553	4 901 902	125 535

Bold signifies the best result

Table 4 *Homo sapiens* genome: Results (in bytes) for compressing chromosome *U* into *C*

U	$\|U\|$	Our Scheme		GRS	GReEn
		$\|C\|$	$\|\mathbb{L}(C)\|$	[20]	[21]
1	247 249 719	2 836 652	**1 082 859**	1 336 626	1 225 767
2	242 951 149	2 871 186	**1 050 170**	1 354 059	1 272 105
3	199 501 827	2 115 410	**790 444**	1 011 124	971 527
4	191 273 063	2 398 432	**910 898**	1 139 225	1 074 357
5	180 857 866	2 064 874	**764 458**	988 070	947 378
6	170 899 992	1 902 067	**710 355**	906 116	865 448
7	158 821 424	2 326 721	**844 194**	1 096 646	998 482
8	146 274 826	1 617 884	**617 996**	764 313	729 362
9	140 273 252	1 877 509	**704 205**	864 222	773 716
10	135 374 737	1 623 010	**617 633**	768 364	717 305
11	134 452 384	1 586 558	**604 901**	755 708	716 301
12	132 349 534	1 476 523	**566 997**	702 040	668 455
13	114 142 980	1 100 576	**399 527**	520 598	490 888
14	106 368 585	1 026 227	**377 695**	484 791	451 018
15	100 338 915	1 055 663	**398 720**	496 215	453 301
16	88 827 254	1 225 378	**443 009**	567 989	510 254
17	78 774 742	1 081 739	**396 371**	505 979	464 324
18	76 117 153	865 138	**320 361**	408 529	378 420
19	63 811 651	862 129	**320 789**	399 807	369 388
20	62 435 964	605 179	**229 418**	282 628	266 562
21	46 944 323	488 340	**180 096**	226 549	203 036
22	49 691 432	568 734	**205 244**	262 443	230 049
X	154 913 754	7 525 925	**2 494 884**	3 231 776	2 712 153
Y	57 772 954	1 343 260	**429 099**	592 791	481 307
M	16 571	151	151(*)	183	**127**
Sum	3 080 436 051	42 445 265	**15 460 474**	19 666 791	17 971 030

Bold signifies the best result

symbol is not significant. For example, the IUB/IUPAC amino acid and nucleic acid codes use only upper-case letters (see http://www.bioinformatics.org/sms/iupac.html). Also, some environments and formats (such as FASTA) do not distinguish between lower-case and upper-case. According to the NCBI website for BLAST input formats (see http://blast.ncbi.nlm.nih.gov/blastcgihelp.shtml): "Sequences [in FASTA format] are expected to be represented in the standard IUB/IUPAC amino acid and nucleic acid codes, with these exceptions: lower-case letters are accepted and are mapped into upper-case; ..." Further, some programs/environments use character cases for improved visualization, as is the case with the USC Genome Browser, which uses lower-case to show repeats from RepeatMasker and Tandem Repeats Finder (ftp://hgdownload.cse.ucsc.edu/goldenPath/hg38/chromosomes/README.txt).

Also, we see that further compressing the payload with `lzma2` more than doubles the compression ratio. Interestingly, the payload (ρ) compresses much better than the character-case bitstring (α). Nonetheless, the compression via decomposition (in Table 5) yields a compression ratio of 360, a significant improvement over the 199 compression ratio when compressing the genome's characters in their native character-case (in Table 4). As described earlier, we do not further compress chromosome *M* after initial coding for the symbols and triplets, and thus encode only a 4-byte header to remember this decision, given that the payload and character-case bitstring *k* values are fixed. Thus, 8,556,708 bytes are needed, which is an improvement over GRS and GReEn.

Theoretically, our compression scheme requires time linear in the length of the uncompressed text, since we perform one scan of the *LPF*, which is constructed in linear time via the suffix array *SA* [29]. For the *Arabidopsis*

Table 5 *Homo sapiens* genome: Results (in bytes) for compressing chromosome U via decomposition, i.e. compressing the payload (ρ) into C^{ρ} and compressing the character-case bitstring α into C^{α}

U	$\lvert U\rvert$	Our Scheme					GRS	GReEn
		$\lvert C^{\rho}\rvert$	$\lvert\mathbb{L}(C^{\rho})\rvert$	$\lvert C^{\alpha}\rvert$	$\lvert\mathbb{L}(C^{\alpha})\rvert$	$\lvert\mathbb{L}(C^{\rho})\rvert + \lvert\mathbb{L}(C^{\alpha})\rvert$	[20]	[21]
1	247 249 719	381 577	161 319	755 092	447 919	**609 238**	1 336 626	1 225 767
2	242 951 149	356 526	153 805	756 823	452 338	**606 143**	1 354 059	1 272 105
3	199 501 827	284 096	119 348	553 835	343 213	**462 561**	1 011 124	971 527
4	191 273 063	330 381	137 301	619 981	383 882	**521 183**	1 139 225	1 074 357
5	180 857 866	259 922	109 768	550 876	331 063	**440 831**	988 070	947 378
6	170 899 992	265 222	110 544	508 662	310 029	**420 573**	906 116	865 448
7	158 821 424	292 797	121 289	611 475	355 616	**476 905**	1 096 646	998 482
8	146 274 826	222 972	93 378	434 420	261 455	**354 833**	764 313	729 362
9	140 273 252	309 512	132 957	493 024	276 468	**409 425**	864 222	773 716
10	135 374 737	245 264	103 115	436 272	257 895	**361 010**	768 364	717 305
11	134 452 384	222 735	92 471	423 687	254 637	**347 108**	755 708	716 301
12	132 349 534	214 123	88 447	393 764	239 811	**328 258**	702 040	668 455
13	114 142 980	148 938	62 730	301 116	183 038	**245 768**	520 598	490 888
14	106 368 585	141 128	57 354	286 839	170 916	**228 270**	484 791	451 018
15	100 338 915	138 219	58 777	302 957	173 600	**232 377**	496 215	453 301
16	88 827 254	151 606	62 779	346 282	191 190	**253 969**	567 989	510 254
17	78 774 742	136 168	57 030	301 837	171 680	**228 710**	505 979	464 324
18	76 117 153	113 469	47 122	241 437	140 909	**188 031**	408 529	378 420
19	63 811 651	130 468	53 531	230 673	134 701	**188 232**	399 807	369 388
20	62 435 964	94 273	38 689	169 584	99 796	**138 485**	282 628	266 562
21	46 944 323	71 121	28 744	141 387	79 835	**108 579**	226 549	203 036
22	49 691 432	81 329	33 663	164 026	89 961	**123 624**	262 443	230 049
X	154 913 754	523 282	196 868	1 533 249	875 026	**1 071 894**	3 231 776	2 712 153
Y	57 772 954	152 464	57 002	300 287	153 582	**210 584**	592 791	481 307
M	16 571	64	64(*)	49	49(*)	**113**	183	127
Sum	3 080 436 051	5 267 656	2 178 095	10 857 634	6 378 609	**8 556 704**	19 666 791	17 971 030

Bold signifies the best result

thaliana and *Oryza sativa* genomes, we ran our programs on a laptop; for the *Homo sapiens* genome, we ran our programs in an AWS EC2 m4.4xlarge environment. Consider, for example, the larger chromosomes of the *Homo sapiens* genome. For a payload (ρ), the *SA* construction required 2,376 sec and the *LPF* construction required 399 sec. Note that depending on the application, the *SA* and *LPF* may already be available. Given the *LPF*, our compression algorithm completed in less than one second. Decompression is also fast, and lightweight, since no data structures are required as parameters. Our future plan includes using more efficient *SA* and *LPF* constructions.

Conclusions

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 5315 bytes, an improvement over the best performing state-of-the-art methods (6644 bytes [20] and 6559 bytes [21]). For the *Oryza sativa* genome (originally 372,317,567 bytes), our scheme compressed the genome to 108,159 bytes, an improvement over the 4,901,902 bytes in [20] and the 125,535 bytes in [21]. We also experimented with the *Homo sapiens* genome (originally 3,080,436,051 bytes), which was compressed to 19,666,791 bytes and 17,971,030 bytes in [20] and [21], respectively. By applying our scheme via a decomposition approach, we compress the genome to 8,556,708 bytes, and if alphabet character-case is not significant, we compress the genome to 2,178,095 bytes. Further improvement can be obtained by choosing the k parameter for each specific chromosome, or each specific species.

Declarations
This article has been published as part of *BMC Genomics* Vol 17 Suppl 4 2016: Selected articles from the IEEE International Conference on Bioinformatics and Biomedicine 2015: genomics. The full contents of the supplement are available online at http://bmcgenomics.biomedcentral.com/articles/supplements/volume-17-supplement-4.

Funding
This work was supported in part by grants from the US National Science Foundation, #IIS-1552860, #IIS-1236983.

Authors' contributions
All authors contributed to the core elements of this work. DA initiated the project. RB, TA, and DA developed the LCS algorithm. DA and RB developed the compression algorithm. RB implemented the *LPF* and compression methods. AF and RB performed the experiments. DA coordinated the overall project. RB and DA prepared the final manuscript. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Published: 18 August 2016

References

1. Beal R, Afrin T, Farheen A, Adjeroh D. A new algorithm for 'the LCS problem' with application in compressing genome resequencing data. In: Bioinformatics and Biomedicine (BIBM), 2015 International, IEEE, Conference on; 2015. p. 69–74.
2. Gusfield D. Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. New York, NY: Cambridge University Press; 1997.
3. Adjeroh D, Bell T, Mukherjee A. The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching, 1st ed. New York, NY: Springer; 2008.
4. Lin Z, Wang H, McClean S. A multidimensional sequence approach to measuring tree similarity. IEEE Trans Knowl Data Eng. 2012;24(2):197–208.
5. Smith TF, Waterman MS. Identification of common molecular subsequences. J Mol Biol. 1981;147:195–7.
6. Aach J, Bulyk M, Church G, Comander J, Derti A, Shendure J. Computational comparison of two draft sequences of the human genome. Nature. 2001;26(1):5–14.
7. Wandelt S, Leser U. FRESCO: Referential compression of highly similar sequences. IEEE/ACM Trans Comput Biol Bioinform. 2013;10(5):1275–88.
8. Wandelt S, Starlinger J, Bux M, Leser U. RCSI: Scalable similarity search in thousand(s) of genomes. Proc VLDB Endow. 2013;6(13):1534–45.
9. Giancarlo R, Scaturro D, Utro F. Textual data compression in computational biology: Algorithmic techniques. Comput Sci Rev. 2012;6(1):1–25.
10. Kuo C-E, Wang Y-L, Liu J-J, Ko M-T. Resequencing a set of strings based on a target string. Algorithmica. 2015;72(2):430–49.
11. Myers EW. An O(ND) difference algorithm and its variations. Algorithmica. 1986;1(2):251–66.
12. Ukkonen E. Algorithms for approximate string matching. Inform Control. 1985;64:100–18.
13. Hunt JW, Szymanski TG. A fast algorithm for computing longest subsequences. Commun ACM. 1977;20(5):350–3.
14. Hirschberg DS. A linear space algorithm for computing maximal common subsequences. Commun ACM. 1975;18(6):341–3.
15. Yang J, Xu Y, Shang Y, Chen G. A space-bounded anytime algorithm for the multiple longest common subsequence problem. IEEE Trans Knowl Data Eng. 2014;26(11):2599–609.
16. Maier D. The complexity of some problems on subsequences and supersequences. J ACM. 1978;25(2):322–36.
17. Apostolico A, Giancarlo R. The Boyer-Moore-Galil string searching strategies revisited. SIAM J Comput. 1986;15(1):98–105.
18. Jacobson G, Vo K-P. Heaviest increasing common subsequence problems. In: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching, ser. CPM '92. London: Springer-Verlag; 1992. p. 52–66.
19. Pevzner PA, Waterman MS. A fast filtration algorithm for the substring matching problem. LNCS 684 Comb Pattern Matching. 1993;684:197–214.
20. Wang C, Zhang D. A novel compression tool for efficient storage of genome resequencing data. Nucleic Acids Res. 2011;39(7):e45.
21. Pinho AJ, Pratas D, Garcia SP. GReEn: A tool for efficient compression of genome resequencing data. Nucleic Acids Res. 2012;40(4):e27.
22. Nevill-Manning CG, Witten IH. Protein is incompressible. In: Proceedings of the Conference on Data Compression, ser. DCC '99. Washington: IEEE Computer; 1999. p. 257.
23. Adjeroh D, Nan F. On compressibility of protein sequences. In: DCC.IEEE Computer Society. IEEE; 2006. p. 422–34.
24. Coxm AJ, Bauer MJ, Jakobi T, Rosone G. Large-scale compression of genomic sequence databases with the Burrows-Wheeler Transform. Bioinformatics. 2012;28(11):1415–9.
25. Giancarlo R, Scaturro D, Utro F. Textual data compression in computational biology: A synopsis. Bioinformatics. 2009;25(13):1575–86.
26. Wandelt S, Bux M, Leser U. Trends in genome compression. Curr Bioinform. 2014;9(3):315–26.
27. Fritz M, Leinonen R, Cochrane G, Birney E. Efficient storage of high throughput DNA sequencing data using reference-based compression. Genome Res. 2011;21:734–40.
28. Hach F, Numanagic I, Alkan C, Sahinalp SC. SCALCE: boosting sequence compression algorithms using locally consistent encoding. Bioinformatics. 2012;28(23):3051–7.
29. Crochemore M, Ilie L. Computing longest previous factor in linear time and applications. Inf Process Lett. 2008;106(2):75–80.
30. Beal R, Adjeroh D. Parameterized longest previous factor. Theor Comput Sci. 2012;437:21–34.
31. Beal R, Adjeroh D. Variations of the parameterized longest previous factor. J Discret Algorithm. 2012;16:129–50.
32. Cormen TH, Stein C, Rivest RL, Leiserson CE. Introduction to Algorithms, 2nd ed. Cambridge, Massachusetts: The MIT Press; 2001.
33. Crochemore M, Ilie L, Smyth WF. A simple algorithm for computing the Lempel Ziv factorization. In: Proceedings of the Data Compression Conference, ser. DCC '08. Washington: IEEE Computer Society; 2008. p. 482–8.