

The SCP and Compressed Domain Analysis of Biological Sequences

Don Adjeroh and Feng Jianan

[don, feng]@csee.wvu.edu

Lane Department of Computer Science and Electrical Engineering

West Virginia University, Morgantown, West Virginia

Abstract.

We introduce the SCP - the sorted common prefix, and study some of its properties. Based on the internal representations used by a class of new compression schemes, we show how the SCP table can be constructed using an $O(u + |\Sigma|k_{\max})$ number of comparisons on average, and $O(u|\Sigma|)$ worst case, where u is the size of the sequence, $|\Sigma|$ is the number of symbols, and k_{\max} is the maximum SCP value. We describe how two applications of the SCP in biological sequence analysis. In particular, using the SCP, and the compressed representation of the sequence, we present an algorithm for finding all the h_{occ} canonical tandem arrays in the sequence in $O(u + h_{occ} + |\Sigma|k_{\max})$ time on average, and $O(h_{occ} + u|\Sigma|)$ worst case. Preliminary results on the statistics of the SCP for some DNA and protein sequences are included.

1. Introduction

The complete sequencing of the human genome and the genomes from various other model organisms represent an important milestone, with impacts beyond the direct realms of molecular biology. An important new challenge facing scientists, therefore, is how to make sense out of this huge mass data. With single gene sequences, we usually consider sequences with tens of thousands of base pairs. With whole genomes, we now have to deal with millions, or billions of base pairs. When we have a database of such genomes, as is typically the case, the problem becomes even more compounding. Thus, there is need to devise effective techniques for the management, organization, and distribution of this unprecedented mass of biological sequence data. This is in addition to the (now heightened) need for efficient and effective algorithms for the analysis, annotation, interpretation and visualization of biological sequences [Roose2001].

In this work, we make use of the internal representations used by a class of text compression algorithms (those that compress data by context sorting) to develop new data structures that are central to various pattern matching and analysis operations on biological sequences. We show that the data structures can indeed be used to develop new methods to rapidly analyze biological sequences, by looking at their compressed representations. In particular, we introduce the SCP – sorted common prefix as one such data structure, and show how the SCP could be used to for improved algorithms for anchor-based sequence alignment, and for locating tandem repeats.

Our approach to compressed domain biosequence analysis is based on new class of compression algorithms. The distinguishing property of the members of this class of algorithms is that they perform *compression by sorted contexts*. They compress the given data by sorting the context of the individual symbols. In the compressed form, these sorted contexts manifest in the form of important data structures that can be used for a speedy search of a pattern directly on the compressed text, without the usual process of decompression. Example algorithms in this class are the block-sorted compression schemes of Burrows and Wheeler (the Burrows-Wheeler Transform, BWT) [Burrows94w], the sort-based scheme of Yooko [Yooko97], and the permutation scheme of Arnavut and Magliveras [Arnavut97m].

Pattern matching plays a central role in different aspects of biological sequence analysis, and has been used in applications as diverse as short-gun sequencing, multiple sequence alignments, analysis of repetition structures [Volfovsky2001hs], searching for unique oligonucleotides [Delcher99kfpws], finding DNA-binding protein motifs, identifying single nucleotide polymorphisms (SNPs), etc. See [Gusfield97, Durbin98ekm] for more detailed accounts. Operating directly on the compressed data has the obvious advantage of significantly reducing the time required for such analysis. However, and more importantly, when the identified class of compression algorithms is used on biological sequences, the internal structures produced could expose some previously unknown or un-identified relationships within the sequences, or across genomes – when used in cross-genome comparisons. For instance, the sorted nature of the context will mean that, typically, areas having similar contexts in the sequence will be clustered together in the compressed form.

Significantly, the identified class of sorted-context compression algorithms have an interesting relationship with other known data structures such as suffix trees[McCreight76] and suffix arrays[Manber93], which have important applications in rapid pattern analysis], repeat finding, and multiple sequence alignment [Gussfield97]. The implication of this important relationship is that sequence analysis tasks involving these data structures can now be performed directly on the compressed data – when the compression is performed with a member of the identified class.

2. Background

We study a new class of problems – compressed domain analysis of biological sequences. Although related, this is different from the more usual case of predicting important relationships between homologous families based on the similarity in the compressed data sizes [Chen99kl, Grunbach94t, Loewenstern99y]. Here, we consider the internal representations of the data as used by the defined class of compression algorithms, and we perform analysis based on this representation. In this section, we describe the BWT algorithm. The BWT is the foundation for our sorted-context approach to compressed-domain sequence analysis.

2.1 Nature of biological sequences. From a computational viewpoint, a biological sequence can be viewed simply as a one-dimensional sequence of symbols, for instance, with an alphabet of 4 symbols for DNA, or RNA, and 20 symbols for proteins. Biological sequences typically contain different types of repetitions and other hidden regularities. Long runs of tandem repeats and of randomly interspersed repeats are prominent features of DNA sequences. The function of the repetitive non-coding areas are not completely understood. Repetition structures have been implicated in various diseases and genetic disorders. For instance, the triplet repeats $(CTG)_n/(CAG)_n$ have been associated with the Huntington's disease, while the hairpins formed in $(CGG)_n/(CCG)_n$ repeats have been linked to the Fragile-X mental retardation syndrome [Bat97ka]. While the exact function of each of the different genes so far identified is not completely known, an even less is known about the repeats, and this represents a major challenge.

2.2 Compression of biological sequences. Most successful methods for compressing biological sequences consider the different regularities or repetition structures that are inherent in these sequences. Some repetition structures that can be exploited here include simple (interspersed) repeats (SINES and LINES), tandem repeats, palindromes, complemented repeats, and complemented palindromes. In each case, we may need to consider both cases of exact and in-exact (approximate) repetitions. The key problem becomes how we can speedily identify these repetition structures, and how we can make use of their knowledge in compressing the sequence. Different specialized algorithms have thus been proposed for compressing biosequences, with varying degrees of success [Grumbach94t, Chen99kl, Lanctot2000ly, Apostolico2000l, Loewenstern99y]. The different algorithms differ in the type(s) of repetitions they exploit, and how they do this. A more closely related work is that reported in [Adjeroh2002zpb], where offline and online dictionary-based methods were described for compressing DNA sequences using the BWT.

2.3 The Burrows-Wheeler Transform

The BWT [Burrows94w] performs a permutation of the characters in the sequence, such that characters in lexically similar contexts will be near to each other. The important procedures in BWT-based compression/decompression are the forward and inverse BWT, and encoding of the permuted sequence.

2.3.1 The forward transform. Given an input sequence $T = t_1, t_2, \dots, t_u$, the forward BWT is composed of three steps: i) Form u permutations of T by cyclic rotations of the characters in T . The permutations form a $u \times u$ matrix \mathbf{M} , with each row in \mathbf{M} representing one permutation of T ; ii) Sort the rows of \mathbf{M} lexicographically to form another matrix \mathbf{M} . \mathbf{M} includes T as one of its rows; iii) Record L , the last column of the sorted permutation matrix \mathbf{M} , and id , the row number for the row in \mathbf{M} that corresponds to the original sequence T . The output of the BWT is the pair, (L, id) . Generally, the effect is that the contexts that are similar in T are made to be closer together in L . This similarity in nearby contexts can be exploited to achieve compression. As an example, suppose $T=ACTAGA$. Let F and L denote the array of *first* and *last* characters respectively. Then, $F=AAACGT$ and $L=GATAAC$. The output of the transformation will be the pair: $(L, id)=(GATAAC, 2)$ - indices are from 1 to u . The rotation matrices for the sequence $T=ACTAGA$ is given in Figure 1.

2.3.2 The inverse transform. The BWT is reversible. It is quite striking that given only the (L, id) pair, the original sequence can be recovered exactly. The inverse transformation can be performed using the following steps [Burrows94w]: i) Sort L to produce F , the array of first characters; ii) Compute V , the **transformation vector** that provides a one-to-one mapping between the elements of L and F , such that $F[V[j]] = L[j]$. That is, for a given symbol $s \in \Sigma$, if $L[j]$ is the c -th occurrence of s in L , then $V[j]=i$, where $F[i]$ is the c -th occurrence of s in F ; iii) Generate the original sequence T , since the rows in \mathbf{M} are cyclic rotations of each other, the symbol $L[i]$ cyclically precedes the symbol $F[i]$ in T . That is, $L[V[j]]$ cyclically precedes $L[j]$ in T . For the example with $T=ACTAGA$, we will have $V = [5\ 1\ 6\ 2\ 3\ 4]$. Given V and L , we can generate the original text by iterating with V . This is captured by a simple algorithm: $T[u+1-i] = L[V^{i-1}[id]]$, $\forall i = 1, 2, \dots, u$, where $V^0[s] = s$, and $V^{i+1}[s] = V^i[s]$, $1 \leq s \leq u$. In

practically implementations, V is usually computed using an array of character counts, C , which contains the number of occurrence of each symbol in the text. With V , we can use the relationship between L , F , and V to avoid the sorting required to obtain F . Thus, we can compute F in linear time, $O(u)$.

2.3.3 BWT-based compression. Compression with the BWT is usually accomplished in a four-phase pipeline, viz. $input \rightarrow BWT \rightarrow MTF \rightarrow RLE \rightarrow VLC \rightarrow output$. Here, BWT stands for the forward BWT transform; MTF is move-to-front encoding [Bentley86stw] used to further transform L for better compression (this usually produces runs of the same symbol); RLE is run length encoding of the runs produced by the MTF; and VLC is variable length coding of the RLE output using entropy encoding methods, such as Huffman or arithmetic coding.

2.3.4 Auxiliary transformation arrays. In order to provide some form of random access to the transformed BWT output, we introduced auxiliary transformation vectors [Adjero2002mpbz, Bell2002pbma]. Define an array Hr computed from V by the following algorithm: $x:=id$; for $i:=1$ to u do $\{x:=V[x]; Hr[u+1-i]:=x\}$. Given the arrays F and Hr , the original text can be retrieved by applying the *inverse* transformation $T[i]=F[Hr[i]]$, $1 \leq i \leq u$. Another auxiliary array used is Hrs , defined as the index vector to Hr . That is, $T[Hrs[i]]=F[i]=F[Hr[Hrs[i]]]$. With the example, $T=ACTAGA$, $L=GATAAC$, $id=2$, $F=AAACGT$, $V=[5\ 1\ 6\ 2\ 3\ 4]$, $Hr=[2\ 4\ 6\ 3\ 5\ 1]$ and $Hrs=[6\ 1\ 4\ 2\ 3\ 5]$. Figure 2 provides an illustration of the auxiliary arrays and their relation to other vectors.

2.3.5 The BWT and repetition structures. The BWT is very closely related to the suffix tree and suffix array - two important data structures used in pattern analysis and in analysis of repetition structures. The major link is the fact the BWT provides a lexicographic sorting of the contexts as part of the permutation of the input sequence. The suffix tree [McCreight76] is a tree that contains all the suffixes of a given sequence, such that each leaf node in the tree corresponds to a suffix in the original sequence. A traversal of the node from the root to the leaf reads out the corresponding suffix. An internal node in the suffix tree with more than one child leaf node implies that the substring obtained by tracing from the root to the internal node is a repeated substring in the sequence. A suffix array [Mamber93] is obtained by sorting the suffixes (or leaf nodes in the tree). Thus, a suffix array can be seen as a lexicographically ordered list of the suffixes. Given the started positions of each substring in the sorted array of suffixes, the suffix array can be represented as a simple array of these indices. In Figure 3, we show the list of suffixes and the corresponding sorted suffixes for the example used above. The suffix tree and the suffix array are also included.

To see the relationship between the BWT and the suffix tree or suffix array, we have also included the final matrix of sorted rotations (M) from the BWT (described above). If we ignore the characters after the special symbol $\$$ in the final results from the BWT rotation and permutation procedures, the sorted suffixes correspond exactly to the results from the BWT. (We have used bold and uppercase letters to highlight these suffixes in the sorted matrix of rotations). Most importantly, the suffix array corresponds exactly to the auxiliary array, Hrs , defined previously! This relationship is important in our approach to the compressed domain analysis of repetition structures. By constructing the auxiliary arrays from the output of the BWT, we implicitly construct the suffix array from the original sequence. Further, with the availability of the two auxiliary arrays, Hr , and Hrs , and the array of first characters, F , we can access any part of the original sequence T , without complete decompression. Thus, we can perform an analysis of the repetition structures in the original sequences directly on the compressed representations.

We point out that suffix trees have also been used in other related work in finding repetitions in genome-scale biological sequences, for instance, [Volfovsky2001hs, Delcher99kfpws, Gusfield97]. Various other methods have also been proposed for finding repeats [Crochemore81, Main84l, Benson94w, Coward98d]. The methods however perform in the uncompressed domain, and hence do not exploit the data structures used in compression. The time and space requirement for suffix trees is quite significant, although it is linear in theory.

2.3.5 Compressed pattern matching and compressed domain biological sequence analysis

Given a text string T of size u , a search pattern P of size m , and an encoded string Z of size n (Z is the compressed representation of T), the question is: can we locate the occurrences of P in T without full decompression of Z , or with only a partial decompression? The compressed pattern-matching problem has attracted a reasonable amount of attention in the past few years. Compressed pattern matching methods have been proposed for the LZ-family of algorithms [Farach98t], where algorithms have been proposed that can search for a pattern in $O(n \log^2(\frac{u}{n}) + m)$ time.

In [Moura2000nzb], $O(n + m\sqrt{u})$ algorithms were proposed for searching Huffman-encoded files. An off-line search index has also been reported for the LZ78 algorithm [Karkkainen98u]. *Off-line* exact-pattern matching algorithms

[Ferragina2000m, Sadakane2000] have been proposed for searching directly on BWT-compressed data. Compressed pattern matching methods for the BWT have been reported in [Adjeroh2002mpbz, Bell2002pbma].

There are also several variants of the pattern-matching problem. An important one for biological sequence analysis is pattern matching with errors (or k -approximate matching), whereby up to k errors can be allowed in the match [Gusfield97, Navarro2001]. In [Karpinski97pr], the problem of **fully compressed pattern matching** was described, where both the text and the pattern are compressed, and the search is to be performed without any decompression. Fully compressed pattern matching will be important for biological sequences, where we might want to avoid decompression, since both the query and the database sequences could be quite large (for instance, in comparative genomics). While the image analysis and text retrieval communities have long realized the importance of operations directly in the compressed domain [see Bell2001am], surprisingly no such notions have been observed in the bio-sequence analysis community. Yet, pattern matching is a central procedure for various analysis tasks in biological sequences.

3 The SCP Data Structure

Suppose, we pre-compute the longest-common-prefix (LCP) between *all* pairs of the sorted suffixes from a sequence, T . Using the relationship between the BWT and the suffix tree, in addition to the auxiliary arrays previously described, we can obtain these sorted suffixes directly from the compressed sequence. We store this information in a table. Since our table is based on the sorted suffixes, we call this the **sorted common prefix (SCP)**. Although the SCP and traditional LCP store the same basic information, the structure of the SCP is completely. Also, the sorted nature of the suffixes for the SCP will have implications in the computation this table, and its diverse uses. Given the i -th and j -th sorted suffixes, if $SCP(i,j)=k$, it means that the first k positions in the i -th suffix and the j -th suffix are exact matches (i.e. $SUFFIX_i[1..k] = SUFFIX_j[1..k]$ and $SUFFIX_i[k+1] \neq SUFFIX_j[k+1]$). This kind of computation is often used to speed up offline pattern matching. The SCP is a centerpiece in our approach. Although motivated by the problems in pattern matching, we will use it to study various problems in compressed domain analysis of biological sequences.

3.1 Nature of the SCP data structure.

An example SCP table (and the corresponding LCP) for a short sequence $T = \text{ACTAGACTA}\$$ is given in Figure 4. The figure shows a very important difference between the LCP and the SCP. The LCP provides no apparent structure. The SCP provides a cluster of similar areas in the sequence. Compared with the LCP, the structural nature of the SCP provides more explicit information about the nature of the original sequence. More importantly, this structure can be used in efficient applications of the SCP, and its construction. We observe the following properties from the SCP structure. Let i, j, k , be indices for sorted suffixes, such that $i < j$, and $j < k$. Then, $SCP(i, j) \geq SCP(i, k), \forall k > j$. Further, if $SCP(i, j) \geq 0$ and $SCP(i, k) = 0$, then $\forall k > j, SCP(j, k) = 0$. More generally, let $x = SCP(i, k)$. Then $\forall k > j$,

$$\begin{aligned} SCP(j, k) &= SCP(i, k) + SCP(SUFFIX_j[x+1, \dots, u], SUFFIX_k[x+1, \dots, u]) \\ &= x + SCP(SUFFIX_j[x+1, \dots, u], SUFFIX_k[x+1, \dots, u]). \end{aligned}$$

Obviously, the SCP is symmetric, $SCP(j,k)=SCP(k,j)$. Also, the SCP is sparse in general, although not always. For instance, a sequence with $S = \text{AAAAA}$ will produce a full SCP table, where the row entries are of the form **1 1 1 1; 2 2 2; 3 3; 4**. Using the SCP, we define a measure of the "*self-similarity*" of the sequence (or the "fullness" of the SCP) by simply counting the sum of the entries (above the top-left to bottom-right diagonal). For a sequence of size u , with a full SCP, we have $Sim(u) = \frac{1}{6}u(u+1)(u-1)$. Then, for any given sequence T of length u , we define its self-similarity as: $Sim(T) = \frac{1}{Sim(u)} \sum_{i,j} SCP(i, j)$. Hence, for the full SCP, the self-similarity of the corresponding

sequences will be 1. A sequence with low self-similarity will be difficult to compress, but the SCP can be more compactly represented. The SCP has a lot of interesting properties. We will be introducing some of these, as we need them.

4. Computing the SCP

Because of the sorting involved, computing the SCP could be quite a daunting process. A naïve approach could run in as much as $O(u^3)$, after an $O(u^2 \log u)$ time for sorting the suffixes. However, as noted earlier, with the BWT, only $O(u)$ time is required to produce the sorted indices and the required auxiliary arrays. Below, we describe several

algorithms for computing *all* the SCP values. We start a simple algorithm to compute the nearest sorted common prefixes (or nearest sorted common ancestors) in linear time based on binary search.

4.1 Computing the nearest sorted common prefix.

In [Mamber93m], the suffix array was described as a method for fast pattern matching. To improve the worst case running time of their algorithm, they computed the common prefix for each adjacent pairs sorted suffices in linear time, and stored these in an $O(u)$ array called *lcp*. Thus, the *lcp* corresponds to the $(j, j+1)$ diagonal entries in our SCP¹. We refer to these as the *nearest sorted common prefix* (or nSCP). Below, we give a simpler linear-time algorithm for computing the nSCP.

We assume that suffixes are sorted in ascending order. That is, $SUFFIX_i \leq SUFFIX_j \forall i < j$. To compute the nSCP, we start by computing the SCP values for the last row, $SCP(u, x), \forall x$. This is done recursively, by jumping into the middle of the sorted array of suffixes, and performing suffix match between the suffixes at mid-point and the previous endpoint. Then, we use the SCP rules to determine the intermediate SCP values. Without loss of generality, we assume that $u = 2^v$, for some integer v . Also, since the worst case number of comparison will occur when we have a periodic input string $T = aaa...a = a^u$ {i.e. a repeated u times}, we use this to explain the algorithm. And later, we relate this to the general case when $T \neq a^u$. We use a function *suffixmatch*() which compares two suffixes and returns the number of matching prefixes between the two. Basically, *suffixmatch*(A, B) is defined as:

suffixmatch(A, B) {count =0; for (k=1; k < u; k++) if (A[k] == B[k]) count++; else break; return count; }

Thus, to compute $SCP(u, v)$, for all v , we first compute the SCP values at $v = \frac{u}{2}$ and $v = u - \frac{u}{2}$, $i = 1, \dots, \log u - 1$. We start with $SCP(u, \frac{u}{2}) = \text{suffixmatch}(SUFFIX_u, SUFFIX_u/2)$. This divides the SCP table into two halves – see Figure 5. We will still need to compute the remaining $SCP(u, v)$ for $v < \frac{u}{2}$ (upper half) and for $v > \frac{u}{2}$ (lower half). Consider the upper half. To compute $SCP(u, \frac{u}{4})$, we first compute

$SCP(\frac{u}{2}, \frac{u}{4})$ and use the properties of the SCP to determine $SCP(u, \frac{u}{4})$, viz:

$SCP(u, \frac{u}{4}) = \min\{ SCP(u, \frac{u}{2}), SCP(\frac{u}{2}, \frac{u}{4}) \}$. Similarly, we compute $SCP(u, \frac{u}{2^i})$, $i = 1, 2, \dots, \log u$, using the values for $SCP(u, \frac{u}{2^{i-1}})$, and some additional (maximum of $\frac{u}{2^i}$) comparisons. In the process we have the values for $SCP(\frac{u}{2^i}, \frac{u}{2^k}), \forall k < i$ as a by-product. We apply the same procedure to the lower half to determine the values for $SCP(u, u - \frac{u}{2^i}), i = 2, \dots, \log u - 1$. Similar to the upper half, this also yields the $SCP(u - \frac{u}{2^i}, u - \frac{u}{2^k}), \forall i < k$ as byproducts.

To compute $SCP(u, v)$ for the remaining values of v , (i.e. $v \neq \frac{u}{2^i}$ and $v \neq u - \frac{u}{2^i}, i = 2, \dots, \log u$), we apply the procedure recursively to each of the partitions produced at the previous step, and then determine the SCP value using the properties of the SCP was done previously. We continue this process of binary partitioning until we get to the base case when the partition contains neighboring suffixes, $SUFFIX_k \leq SUFFIX_{k+1}$ say. Thus, at the final iteration, the suffix match at each small partition will produce the nearest sorted common prefix for the pair. In the process, we have computed the values for $SCP(u, v), \forall v$ (our initial objective), and for $SCP(\frac{u}{2^i}, \frac{u}{2^k}), \forall k < i$, and $SCP(u - \frac{u}{2^i}, u - \frac{u}{2^k}), \forall i < k$.

Definition: Define a *b-binary path* (*b-part*, for short) as a traversal that starts from the midpoint a side (different from the hypotenuse) of a right-angled triangle of base b , and visits the midpoints of successive divisions of the hypotenuse, in the manner shown in Figure 5. Thus, the length of a *b-path* is always equal to b .

Analysis. Consider the u -path (A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow ... \rightarrow Z) in Figure 5, and the lower half of the triangle BXZ. For the lower half, we need to consider one $\frac{u}{4}$ -block, one $\frac{u}{8}$ -block, ..., etc, for a total of $\log u$ smaller blocks,

¹ We use LCP to represent the table of common prefixes between each pair of suffixes in the original sequence. That is, the suffixes are not sorted. The *lcp* as was used in [Mamber93m] represents only the common prefix between each adjacent pair of *sorted* suffixes.

each half the size of the previous block. The number of comparisons required for the u -path and to determine all the values for $SCP(u, j)$ in this lower half is given by:

$c(u) = (u - 1) + (\frac{u}{2} - 1) + (\frac{u}{4} - 1) + (\frac{u}{8} - 1) + \dots + 1 \leq 2u - \log u$. For the upper half, (i.e. upper triangle YAB), we start out to recursively compute the value for $SCP(\frac{u}{2}, k), \forall k = 1, 2, \dots, \frac{u}{2} - 1$, as we did for the u -block. Then,

using SCP rules, we determine the required $SCP(u, j)$ using $SCP(\frac{u}{2}, j)$, for j in the upper half. We will have a total of $\log u$ number of upper halves. Thus, the total time will be:

$$Cost(u) = c(u) + c(\frac{u}{2}) + c(\frac{u}{4}) + \dots + 1 \leq 3u + \frac{1}{2}(\log u - \log u \log u) = O(u)$$

We have assumed above that the $T = a^u$, i.e. $\Sigma = \{a\}$, or $|\Sigma| = 1$. For the average case performance, we can assume that we have an alphabet with multiple symbols, and that $T \neq a^u$. Here, if we assume an equi-probable alphabet, the height of the right-angled triangle in the b -path becomes $\frac{u}{|\Sigma|}$, instead of u .

Definition: Given the array of sorted suffixes, the suffixes that start with the same symbol form a partition of the array. We call such a partition a **s-partition**, where **s** is the starting symbol. In a similar manner, the SCP could be partitioned into $|\Sigma|$ partitions, whereby members of each partition have a common first symbol.

Using the analysis above, the cost of one **s-partition** becomes $Cost(\mathbf{s-partition}) \leq 2(u - |\Sigma|) - \frac{1}{2}(\log u \log \frac{u}{|\Sigma|}) = O(u - |\Sigma|)$. For all the $|\Sigma|$ **s-partitions**, we will have a total cost of: $Cost(u) = |\Sigma| \left(u - |\Sigma| - \log u \log \frac{u}{|\Sigma|} \right) \leq |\Sigma| (u - |\Sigma|)$. When we assume that $|\Sigma|$ is constant we will have an $O(u)$ overall cost. We summarize the above with the following theorem:

Theorem 1: Given a string $T = t_1 t_2 \dots t_u$, and an equi-probable symbol alphabet Σ , the nearest sorted common prefix (nSCP) can be computed in $O(|\Sigma|(u - |\Sigma|))$ time. For fixed $|\Sigma|$, the nSCP can be computed in $O(u)$ time.

4.2 Computing the SCP

4.2.1 Algorithm A. Given the nearest sorted common prefixes, it becomes easy to compute the value of $SCP(j, k)$ for a given (j, k) pair. Let $j < k$. Using the properties of the SCP, we can compute the SCP values from the nSCP as follows: $SCP(j, k) = \min\{SCP(j, j+1), SCP(j+1, j+2), \dots, SCP(k-1, k)\}$. For a fixed j , we can compute $SCP(j, k) \forall k = j+2, j+3, \dots, u$ by recording the minimum values when we consider each pair of neighboring suffixes, and re-use it at the next step, viz:

$$\begin{aligned} mnscp &= SCP(j, j+1) \\ \text{for } (k = j+1; k < u; k++) \\ &\{ mnscp = \min\{mnscp, SCP(k, k+1)\}; SCP(j, k+1) = mnscp \} \end{aligned}$$

This requires $O(u)$ lookups and $O(u)$ number of $\min()$ operations, for a total of $O(u^2)$ time to compute all the SCP values. This can be reduced to $O(u \mathbf{k}_{\max})$, where \mathbf{k}_{\max} is the maximum nSCP value. This is based on the SCP rule: $SCP(i, k) \leq SCP(i, j), \forall i < j < k$, and hence $SCP(i, k) = 0$ if $SCP(i, j) = 0$.

4.2.2 Algorithm B. For most files, many of the SCP values will be zero. Based on the SCP properties, with a zero in the SCP, we can make a number of deductions about some of the other SCP values. Consider one **s-partition** in the SCP. First, starting from the last suffix in the sorted array of suffixes, we use binary search to find the end of the **s-partition**. Let j be the start of the **s-partition**, and $k+1$ be the first position having zero SCP with j . That is, $SCP(j, k) \neq 0$ and $SCP(j, k+1) = SCP(j, k+2) = \dots = SCP(j, u) = 0$. Then, compute the SCP values $SCP(j, k)$ by starting the suffix match from position $k, k-1, \dots, j+1$, using the relation:

$SCP(j, k) = \text{suffixmatch}(SUFFIX_j, SUFFIX_k)$. At each step in the SCP computation, we record c_p , the position of mismatch between $SUFFIX_j$ and $SUFFIX_k$:

$$c_p = SCP(j, k) = \text{suffixmatch}(SUFFIX_j, SUFFIX_k).$$

To compute $SCP(j, k-1)$, we start the match from position c_p in the suffixes at positions j and $k-1$, viz:

$$SCP(j, k-1) = \text{suffixmatch}(SUFFIX_j[c_p..u], SUFFIX_k[c_p..u]).$$

Analysis. Let $K_{\mathcal{S}_{\max}}$ be the maximum SCP (same as the maximum nSCP) in the \mathcal{S} -partition. The first step of finding the bounds on the \mathcal{S} -partition can be done in $O(\log \frac{u}{|\Sigma|})$ number of comparisons. With the mismatch point already recorded, the maximum number of comparisons at any step k will be $K_{\mathcal{S}_{\max}}$. Thus, the overall time will be in $O(\log \frac{u}{|\Sigma|} + \frac{u}{|\Sigma|} K_{\mathcal{S}_{\max}})$. However, based on the auxiliary arrays from the BWT (see Section 2), the $O(\log \frac{u}{|\Sigma|})$ step of finding the beginning and end of a \mathcal{S} -partition will not be needed. We can get this with two lookups using the \mathbf{C} array. Thus, the actual time required for one \mathcal{S} -partition will be in $O(\frac{u}{|\Sigma|} K_{\mathcal{S}_{\max}})$. For all the $|\Sigma|$ partitions, and assuming equi-probable symbols, we will require an overall time in $O(\frac{u}{|\Sigma|} (K_{1\max} + K_{2\max} + \dots + K_{|\Sigma|\max}))$, or $O(u\mathbf{k}_{\max})$, where

$$\mathbf{k}_{\max} = \max \{K_{1\max} + K_{2\max} + \dots + K_{|\Sigma|\max}\}. \text{ The worst case time will still be } O(u^2).$$

For random sequences, the theoretical expected value of \mathbf{k}_{\max} is given by [Karlin83gotk] $\mathbf{k}_{\max} = 2 \frac{\log u}{\log |\Sigma|} + O(1)$. For most sequences tested (see section on preliminary results), the practical of \mathbf{k}_{\max} are quite far from the theoretical value, again questioning the view that biological sequences are random.

4.2.3 Algorithm C. We improve time complexity of **Algorithm B**, at a cost of an $O(u)$ extra space. Suppose rather than recording single c_p values for each mismatch point, we record an array of c_p values corresponding to the different steps of the iteration on k . That is, while computing $SCP(j, k)$, $\forall k > j$, we record the corresponding mismatch points in an array. We call this array **EP** – match end points. That is, $EP[k]$ = mismatch point between $SUFFIX_j$ and $SUFFIX_k$, which is available after we know $SCP(j, k)$. (j is that starting point of the \mathcal{S} -partition).

The key observation is that, after computing $SCP(j, k) \forall k > j$, these previous mismatch points can be used at the next iteration on j – i.e. in computing $SCP(j+1, k)$, $\forall k > j+1$. From the SCP properties, we know that $SCP(i, k) \leq SCP(i, j)$, $\forall i < j < k$. Thus, to perform the suffix matches for $SCP(j+1, k)$, again instead starting at position 1 in $SUFFIX_j+1$ and $SUFFIX_k$, we use the stored array of mismatch points EP, and start matching from positions $SUFFIX_j+1[EP[k]]$ and $SUFFIX_k[EP[k]]$. Thus, $SCP(j+1, k)$ is computed as follows:

compute scp:

$$SCP(j+1, k) = EP[k] + \text{suffixmatch}(SUFFIX_j+1[EP[k].u], SUFFIX_k[EP[k].u])$$

update EP: $EP[k] = SCP(j+1, k)$

Analysis. Unlike in **Algorithm B**, we do not need to perform a potential $O(K_{\mathcal{S}_{\max}})$ comparisons for *each* $SCP(j, k)$, $\forall k > j$ and fixed j . Now, we need the $O(K_{\mathcal{S}_{\max}})$ comparisons for the entire \mathcal{S} -partition, not just one row in the \mathcal{S} -partition. This reduces the overall time complexity for the algorithm to:

$O(\frac{u}{|\Sigma|} + K_{1\max}) + (\frac{u}{|\Sigma|} + K_{2\max}) + \dots + (\frac{u}{|\Sigma|} + K_{|\Sigma|\max})$, or $O(u + |\Sigma|\mathbf{k}_{\max})$. This improvement comes at an extra space requirement of $O(u)$, which can be reduced to $O(\mathbf{k}_{\max})$, since we can re-use the memory space after we finish with each partition. We summarize the results above in the following theorem:

Theorem 2. Given a string $T = t_1 t_2 \dots t_u$, and an equi-probable symbol alphabet Σ , the table of sorted common prefixes (SCP) can be computed using number of comparisons in $O(u + |\Sigma|\mathbf{k}_{\max})$ on average, and in $O(u|\Sigma|)$ worst

case, with an extra space in $O(k_{\max})$ on average, and $O(u)$ worst case, where k_{\max} is the maximum SCP value. For a fixed alphabet size, the SCP can be computed using an $O(u)$ number of comparisons.

5. Applications

One major motivation for our compressed domain analysis of biological sequences is the expected improvement in speed, since algorithms that operate in the compressed domain will inherently manipulate a smaller amount of data. A potentially more important motivation is the ability of some compression algorithms to more easily expose certain hidden structures in the original biological sequence. The SCP is a natural data structure for analysis of compressed biological sequences, when the compression is based on the BWT. We describe two uses of the SCP in analysis of biological sequences, with only partial decompression of the compressed sequence. Partial decompression is needed to obtain the auxiliary arrays, based on which we compute the SCP table.

5.1. Anchor points in multiple sequence alignment

Given k strings ($k > 2$), $TS = \{T_1, T_2, \dots, T_k\}$, the global multiple alignment is obtained by inserting spaces at chosen positions in the k strings, such that the resulting strings all have the same length, say l . Thus, the strings can be arranged as a $k \times l$ array of characters. The value of the alignment is given by: $V(T_1, T_2, \dots, T_k) = \sum_{i=1}^l v(T_1'(i), T_2'(i), \dots, T_k'(i))$, where $v(T_1'(i), T_2'(i), \dots, T_k'(i))$ denotes alignment score, and $T_j'(i)$ denotes the j -th symbol in resulting sequence T_j' . The problem is to construct multiple alignments with a minimal value. Multiple alignments often require local alignments between the sequences. Efficient computation of multiple alignments is a difficult but important problem in biosequence analysis, and an optimal solution it is known to be an NP-complete problem [dasGupta98w]. Various approximations have been proposed, such as those based on hidden Markov models [Durbin98ekm], consensus subsequences, sum-of-pairs alignments, center-star alignments [Gusfield97]. A popular approach to multiple sequence alignments involves the use of *anchor points* to hone-in on the areas of the sequences that are more likely to lead to an alignment.

Definition. Let $v = SCP(i, j)$. Any position (x, y) in the SCP such that $|i - x| < v$ or $|j - y| < v$ is said to be within the *sphere of influence* of $SCP(i, j)$, and hence can be affected by it.

We use the notion of sphere of influence to indicate the range of substrings that could be affected by the value at a given cell in the SCP. Let $TS = \{T_1, T_2, \dots, T_n\}$ be the set of sequences to be aligned. Since similar areas in the sequence will be clustered together in the SCP, there is a simple way to use the SCP in multiple sequence alignments. To locate the required anchor points, we construct the SCP for the combined sequence: $T = T_1 \$_1 T_2 \$_2 \dots T_n \$_n$, and look for the cluster centers in the SCP. An example SCP table for combined multiple sequences is given in Figure 6 for three sequences: $T_1 = \text{ACCGTT}$; $T_2 = \text{ACGTC}$; $T_3 = \text{CACGTCT}$.

If we view the clusters in the SCP as signals for the potential candidate areas for alignment, we can use them as the anchor points, and hence compute the alignments faster. The problem of determining anchor points thus becomes that of identifying the cluster centers in the SCP and their respective sphere of influence. The simplest way to do this will be to record the positions with the maximum entry in each cluster (with consideration of their individual sphere of influence). This can be done in $O(u)$ time by simply scanning along the diagonal making up the nSCP. Based on the sphere of influence, we can indeed compute the number (or combination of) clusters that must be included in any alignment.

5.2. Tandem Repeats.

The SCP is a natural data structure for the analysis of regularities in a sequence. It obvious from the tables and examples that, whenever $SCP(i, j) \geq 2$, automatically we know that there is a repeat, and the position, length and exact substring that make up the repeat are already available to us. Thus, analysis of repetition structures could be turned into a simple problem of looking up the SCP values in a table. By a careful consideration of the relative positions of the repeats, it is possible to identify different types of repeats such as tandem repeats or interspersed repeats. Palindromes and complemented repeats can be identified in a similar manner by constructing the SCP for the original sequence and its reverse (or complemented) version. Further, the availability of the suffix arrays, (in the compressed domain via the auxiliary arrays) imply that repetition analysis could be much faster when done with the BWT compressed sequence. Here, we show how exact tandem repeats can be found using the SCP, with an improved

time complexity. Previous non-compressed domain methods solve this problem in $O(u \log u)$ time. Below, we use the SCP data structure to provide an $O(u)$ algorithm.

5.2.1 Definitions and Notations. Following[Gusfield97], we use the notion of maximality to describe repetitions.

Definition. A periodic string \mathbf{g} is called a **tandem repeat** if $\mathbf{g} = \mathbf{b}^2$, for some string \mathbf{b} . That is, \mathbf{g} can be written as $\mathbf{g} = \mathbf{bb}$. A tandem repeat \mathbf{g} is called a **maximal tandem repeat** if \mathbf{g} cannot be extended to its immediate right or left.

The tandem repeat is a special case of the more general **tandem array**. \mathbf{g} is called a tandem array if $\mathbf{g} = \mathbf{b}^l$, $l \geq 2$. For any l , we can find some integer k , such that $l = 2k$ (or $l = 2k + 1$ when l is odd). Hence the problem of finding the tandem arrays in a sequence can be solved by finding the tandem repeats, and vice versa. For a given sequence T , the tandem arrays can be represented as a triple: $\langle t_{\mathbf{b}}, \mathbf{b}, l \rangle$ (or more compactly as $\langle t_{\mathbf{b}}, p, l \rangle$) where $t_{\mathbf{b}}$ is the start position in T , $p = |\mathbf{b}|$. For a given tandem repeat, there could be many possible maximal tandem repeats to describe the repeat. For example, the string $\mathbf{g} = acacacacacac$ could be described using any of the following: $(ac)^6$, $(ac)^6$, or $(ac)^6$.

Definition. A string \mathbf{b} is **primitive** if \mathbf{b} is not a tandem repeat (or tandem array). That is, \mathbf{b} is not periodic, and hence, there is no other string \mathbf{l} , such that $\mathbf{b} = \mathbf{l}^l$, for some $l > 1$. A tandem array $\mathbf{g} = \mathbf{b}^l$ is called a **primitive tandem array** if \mathbf{b} is primitive.

Definition. A tandem array $\mathbf{g} = \mathbf{b}^l$ is called **canonical** or **supermaximal** if \mathbf{g} is a primitive tandem array and also maximal. That is, \mathbf{b} is primitive, and \mathbf{b}^l is maximal. The triple $\langle t_{\mathbf{b}}, \mathbf{b}, l \rangle$ (or $\langle t_{\mathbf{b}}, p, l \rangle$) is called a **pm-triple** if $\mathbf{g} = \mathbf{b}^l$ is a supermaximal tandem array.

Examples. In the previous example with $T = acacacacacac$, ac is primitive, and hence the primitive tandem array $(ac)^6$ most succinctly represents T . ca is also primitive in T . The pm-triples will be: $\langle 1, ac, 6 \rangle$ and $\langle 2, ca, 5 \rangle$. For $T = acaacaacaacaacaaca$, we have three primitive strings aca , aac , and caa , with the corresponding supermaximal tandem arrays $(aca)^6$, $(aac)^5$, and $(caa)^5$ respectively. The pm-triples will be: $\langle 1, aca, 6 \rangle$, $\langle 3, aac, 5 \rangle$, $\langle 2, caa, 5 \rangle$.

The problem. The key problem therefore is to find all the canonical (supermaximal) tandem repeats (and their corresponding pm-triples). We observe that, for each primitive string \mathbf{b} , we can have just one supermaximal tandem array. And given the primitive string \mathbf{b} , and l , all that remains will be to find the starting position of \mathbf{b} in T to produce the pm-triple. Thus, the real problem is to find the primitive string \mathbf{b} , its starting positions in *distinct* maximal tandem arrays in T , and the number of occurrences for each starting position.

5.2.2 Geometry for the SCP

Notation: Define the **d -diagonal** as the diagonal in the SCP table, with entries (j, k) , such that $|j-k|=d$. Thus, the 0-diagonal corresponds the trivial case of SCP(k, k). The 1-diagonal corresponds to the SCP values at positions $(k, k+1)$, $k=1, \dots, u-1$. This is simply the nSCP, the nearest sorted common prefix.

To appreciate the geometry in the SCP and its significance, consider the LCP and SCP tables for the periodic sequence $T = acaacaacaacaacaaca$. Although we can observe the periodicity in the LCP table, it is more difficult to discern. With the SCP, the periodic nature of the string becomes very obvious. We observe that the repeating regions in the original string form different clusters in the SCP. In fact, by considering just the 1-diagonal, we can identify all the regions that could contain a tandem repeat. This can easily be done by simply scanning the 1-diagonal (the nSCP) and then considering the differences between adjacent entries along the 1-diagonal – an $O(u)$ time operation. We, however, will need further information to verify that the areas identified are true tandem arrays.

The areas that contain tandem arrays form a specific structure in the SCP. For each primitive repeating substring, we can construct an isosceles triangle, such that points along the sides correspond to SCP values in the area. We call this triangular structure the *p-periodic (triangular) neighborhood* in the SCP. By analyzing this triangular structure, we can provide specific information about the repeat. Two examples are given in Figure 7. We observe the following about the triangle formed from the potential repeat region. The entries along each side of the triangle are always sorted – but in no particular order. If \mathbf{b} is the primitive string, the differences along **two** sides of the triangle gives $p = |\mathbf{b}|$, the length of the primitive string. The values making up the third side of the triangle must be all the same. The number of SCP values along any side of the triangle gives l , the number of repeated primitives. Thus, given the p -periodic neighborhood, we can construct a **difference triangle**, and then check if the values along the sides of the difference triangle satisfy the expected constraints ..

To determine $t_{\mathbf{b}}$, we first obtain start index of the repeat in the F , the array of first characters, and then use the auxiliary arrays to determine the position in T .

Let N_p be the triangular neighborhood defined by the repeating string of size p in the SCP. Let idx be the start index in the F -array. Then, $idx = \arg \max_{j \in N_p} \{SCP(j, j+1) \text{ s.t. } SCP(j, j+1) \leq p\}$. This can **only** be the j value in the

first or the last $(j, j+1)$ position in the side of the triangle that corresponds to the 1-diagonal. We then compute the position in the original sequence T as follows: $t_{\mathbf{b}} = Hrs[idx]$. Thus, we can find the triple $\langle t_{\mathbf{b}}, \mathbf{b}, l \rangle$ in T , via the compressed version of T . Given the pm-triple, we can compute the maximal tandem repeats (or other maximal tandem arrays very easily).

The complete algorithm to compute the pm-triple is given in the appendix.

Analysis. All that we really need in the p -periodic neighborhood are the sides of the triangle, not its internals. Thus, after we determine the 1-diagonal for a potential periodic repeat, we already know the length of all the sides of the triangle. Hence, we can check them directly to know if they meet the expected structure for a tandem repeat neighborhood. In the worst case, the side of each p -periodic triangular structure will be $K_{\mathcal{S}_{\max}}$, the maximum SCP value in the respective \mathcal{S} partition. When we have smaller regions, the total sum of their sizes will still be $O(K_{\mathcal{S}_{\max}})$. Thus, the overall time required to construct all the triangular structures for the sequence will be in $O(\mathbf{k}_{\max})$ on average, or $O(u)$ worst case. We notice that the pruning stage only requires us to find the minimum SCP value along the side of the triangle that represents the 1-diagonal. But the SCP values along the 1-diagonal are sorted (though in no specific order). Thus pruning can be done in $O(\log |N_p|)$ time, using binary search. The same applies to finding the maximum SCP values to determine idx . The remaining computations in step 5 of the algorithm each requires constant time, for a given p -periodic triangular neighborhood. Thus, the total time required by the algorithm will be in $O(u)$.

Lemma 1. Given a sequence $T = t_1 t_2 \dots t_u$, and its SCP table, a pm-triple $\langle t_{\mathbf{b}}, \mathbf{b}, l \rangle$ (i.e. a supermaximal tandem array) can be found for each primitive string \mathbf{b} in T in $O(u)$ time.

Theorem 3. Given a string $T = t_1 t_2 \dots t_u$, and an equi-probable symbol alphabet Σ , a pm-triple $\langle t_{\mathbf{b}}, \mathbf{b}, l \rangle$ can be found for each primitive string \mathbf{b} in $O(u + |\Sigma| \mathbf{k}_{\max})$ on average, and in $O(u |\Sigma|)$ worst case, using the BWT-compressed representation of T . For a fixed alphabet size, the tandem arrays can be found in $O(u)$ time worst case.

The theorem follows from Lemma 1 and Theorem 2.

5.2.3 Case of impostors and multiple occurrences. The question one may ask is what happens when we have impostors – i.e. repeat structures that are truncated by some other patterns. This is akin to the situation where the same primitive pattern \mathbf{b} occurs as in multiple distinct supermaximal tandem arrays in the sequence. Hence, these will have different starting positions $t_{\mathbf{b}}$, and potentially different values of l at each starting position.

Consider the two sequences $T_1 = tacacacact$, and $T_2 = tacacacacttacacacag$. Their SCP tables are given in Figure 8, where the repeating patterns and their p -periodic neighborhoods have been marked. Again, there is a definite structure to the p -periodic neighborhoods. Conceptually, the distinct occurrences manifest as similar (but shifted) copies of the same basic structure. The only difference now is in the size of the triangle, which gives us the l parameter, and the starting point of each copy. Again, the difference triangle can be used to determine the parameters of the repeats. Now, there is also some periodicity along the non-zero sides of the difference triangle. The period here gives the number of copies. It is simple to determine the starting position and the parameters for each distinct supermaximal tandem array. We leave the details for brevity.

Each copy of the structure will require some analysis to verify if it is a tandem repeat, and hence determine its parameters. Thus, the total time required by the algorithm will now depend on the number of duplicate copies of the p -periodic triangular structure. However, Steps 1 to 4 of the algorithm can be done just once for each basic primitive, say \mathbf{b} , since all its distinct supermaximals will lie around one big triangular neighborhood. See the SCP tables in Figure 8. Thus, only Step 5 will require a consideration for each distinct occurrence. Hence, the overall time will be in $O(u + \mathbf{h}_{occ})$, where \mathbf{h}_{occ} is the total number of distinct tandem arrays (with possibly the same base primitive string). The following theorem follows from the above analysis:

Theorem 4. Given a string $T = t_1 t_2 \dots t_u$, and an equi-probable symbol alphabet Σ , all the \mathbf{h}_{occ} pm-triples (supermaximal tandem arrays) in T can be found in $O(u + \mathbf{h}_{occ} + |\Sigma| \mathbf{k}_{max})$ on average, and in $O(\mathbf{h}_{occ} + u |\Sigma|)$ worst case, using the BWT-compressed representation of T . For a fixed alphabet size, the tandem arrays can be found in $O(u + \mathbf{h}_{occ})$ time worst case.

6. Preliminary results

We performed a preliminary experiment, mainly to see the nature of the SCP for different sequences, and to check the performance of the various algorithms proposed for computing the SCP. The DNA files are the same used in a previous work [Adjeroh2002zapb], and have been used by various authors in evaluating DNA compression algorithms. The protein sequences are taken from the protein corpus used in [Nevill-Manning99w]. The results are shown in Table 1. The maximum SCP values are much larger than the theoretical expectation, although the maximum are generally orders of magnitude less than the size of the file. It gives an idea on how difficult it will be to compress the files, and also on how long it could take to compute the SCP. In terms of computation time, expectedly, Algorithm C performed better than the other algorithms, and Algorithm B was consistently better than Algorithm A, although they have the same theoretical complexity.

7. Discussion

We have introduced the SCP data structure as a special structure with particular significance in biological sequence analysis. Based on with the SCP, we described an $O(u + \mathbf{h}_{occ} + |\Sigma| \mathbf{k}_{max})$ algorithm for computing the finding all the exact tandem arrays in a sequence. This can be compared with the previous $O(u \log u)$ algorithms [Gusfield97, Lorenz84m]. The key to our approach is use of auxiliary arrays derived from the compressed representations used by the BWT. This was made possible the special nature of the auxiliary arrays computed from the BWT compressed sequence, and the clustering property induced by the sorting transformation performed by the BWT during compression. Thus, sequence analysis can be performed directly on the compressed biological sequences, while leaving the sequence compressed for as much as possible.

Clearly, there are still a number of issues in the use of the SCP. A compact representation of the SCP is therefore mandatory for its practical deployment. From the sparse nature of the SCP, it should be easy to exploit this to store the SCP in a space-efficient manner. A complete characterization of the SCP poses an interesting problem, and which could lead to improved algorithms in a number of application areas. Other interesting issues could be on how the SCP could be used for approximate tandem repeats, its use in biological sequence compression, cross-genome analysis, accurate entropy estimates and in studying relatedness between sequences.

References

- [Adjeroh2001mtpz] D.A. Adjeroh, A. Mukherjee, T.C. Bell, M. Powell, and N. Zhang, "Pattern matching in BWT-transformed text", *Proceedings, IEEE Data Compression Conference*, Snowbird, Utah, April 2 - 4, 2002.
 [Adjeroh2002zmzpb] Adjeroh D.A., Zhang Y, Mukherjee A., Zhang N., Powell M., and Bell, T.C., "DNA sequence compression using the Burrows-Wheeler Transform", *Proc., IEEE Bioinformatics Conference*, August 2003, pp. 303-313, 2002

- [Arnavut97m] Z. Arnavut and S. S. Magliveras, "Lexical permutation sorting algorithm", *The Computer J.*, **40**(5), 292-295, 1997.
- [Apostolico2000] Apostolico A and Lonardi S, "Offline compression by greedy textual substitution", *Proceedings of the IEEE*, **88** (11), 1733--1744, 2000
- [Bat97ka] Bat O, Kimmel M., and Axelrod D. E, "Computer simulation of expansions of DNA triplet repeats in the Fragile X Syndrome and Huntington's disease", *Journal of Theoretical Biology*, 188, 53-67., 1997,
- [Bell2001am] T.C. Bell, D.A. Adjeroh and A. Mukherjee, "Pattern matching in compressed text and images", May 2001,
- [Bell2001pma] Tim Bell, Matt Powell, Amar Mukherjee and Don Adjeroh "Searching BWT Compressed Text with the Boyer-Moore Algorithm and Binary Search", *Proceedings, IEEE Data Compression Conference*, Snowbird, Utah, April 2 - 4, 2002.
- [Benson94w] G. Benson and M.S. Waterman, "A method for fast database search for all k -nucleotide repeats", *Nucleic Acid Research*, 22, 4828-4836, 1994.
- [Burrows94w] M. Burrows and D.J. Wheeler, "A block-sorting lossless data compression algorithm", *Technical Report*, Digital Equipment Corporation, Palo Alto, CA, 1994.
- [Chen99kl] Chen X., Kwong S. and Li M, "A compression algorithm for DNA sequences and its applications in genome comparison", In *Proceedings, 10th Workshop on Genome Informatics (GIW'99)*, pp. 52--61, 1999.
- [Coward98d] E. Coward and F. Drablos, "Detecting periodic patterns in biological sequences", *Bioinformatics*, 14(6), 498-507, 1998.
- [Crochmore81] Crochmore M, "An optimal algorithm for computing repetitions in a word", *Information Processing Letters*, 12, 244-250.
- [DasGupta98l] DasGupta B, and Wang L., "Selected topics in computational biology", Department of Computer Science, Rutgers University, Camden, NJ, 1998
- [Delcher99kfpws] Delcher, A. L., Kasif, S., Fleischmann, R. D., Peterson, J., White, O., and Salzberg S. L, "Alignment of whole genomes". *Nucleic Acids Research*, **27**(11), 2369--2376, 1999.
- [Durbin98ekm] Durbin, R., Eddy, S., Krogh, A., & Mitchison, G, *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*, Cambridge University Press, (1998).
- [Farach98t] M. Farach and M. Thorup, "String matching in Lempel-Ziv compressed strings", *Algorithmica*, **20**: 388-404.
- [Ferragina2000m], P. Ferragina and G. Manzini, "Opportunistic Data Structures with applications", *Proc., 41st IEEE FOCS*, 2000.
- [Grumbach94t] Grumbach S and Tahi F., "A new challenge for compression algorithms: genetic sequences", *Information Processing & Management*, **30**: 875-886, 1994.
- [Gusfield97] D. Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, Cambridge University Press, Cambridge, UK, 1997.
- [Karlín83gotk] S. Karlín, G. Ghandour, F. Ost, S. Tavaré and L.J. Korn, "New approaches for computer analysis of nucleic acid sequences", *Proceedings, National Academy of Sciences USA*, 80, pp. 5660-5664, Sept. 1983.
- [Karpinski95pr] M. Karpinski, W. Plandowski and W. Rytter, "The fully compressed string matching for Lempel-Ziv encoding", *Technical Report*, (85132-CS), Department of Computer Science, University of Bonn", April, 1995.
- [Lancot2000ly] Lancot J. K., Li M. and Yang E., "Estimating DNA sequence entropy", *Proceedings, 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA2000)*, pp. 409-418, 2000.
- [Loewenstern99y] Loewenstern, D. and Yainilos, P., "Significantly lower entropy estimates for natural DNA sequences", *Proceedings, IEEE Data Compression Conference*, Snowbird, UT, pp. 151-161, 1997.
- [Main84l] M. Main and R. Lorenz, "An $O(n \log n)$ algorithm for finding all repeats in a string", *J. Algorithms*, 5, 422-432, 1984.
- [Manber93m], U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches", *SIAM Journal of Computing*, **22**(5), 935-948, 1993.
- [McCreight76] E. M. McCreight, "A space-economical suffix tree construction algorithm", *J. ACM*, **23**(2), 262-272, 1976.
- [Moura2000nzb], E. S. Moura, G. Navarro, N. Ziviani and R. Baeza-Yates, "Fast and flexible word searching on compressed text", *ACM Transactions on Information Systems*, **18**(2), 113-139, 2000.
- [nManning99w] Nevill-Manning C.G and Witten I.H. "Protein is incompressible", *Proc. IEEE Data Compression Conference*, 1999
- [Roos2001] Roos D.S. "Bioinformatics -- trying to swim in a sea of data", *Science*, 291 (5507): 1260-1261, 2001
- [Sadakane2000], K. Sadakane, "Compressed text databases with efficient query algorithms based on the compressed suffix array", *Proceedings, ISAAC'2000*.
- [Volfovsky2001hs] Volfovsky N. Hass B.J., Salzberg S, "A clustering method for repeat analysis in DNA sequences", *Genome Biology*, **2**(8), 2001
- [Yokoo97], H. Yokoo, "Data compression using a sort-based context similarity measure", *The Computer J.*, **40**(2/3), 94-102, 1997.

Appendix Algorithm for tandem arrays

The following algorithm produces the parameters for the pm-triple.

Step 1. Identify potential repeat areas using the 1-diagonal:

Step 2. Define p -periodic triangular neighborhood for tandem repeat:

Let $p = \pm 1, \pm 2, \dots, \pm \frac{u}{2}$

$N'_p = \{(j, k)\}$ such that

$SCP(j, k) - SCP(j+1, k+1) = p$	side along 1-diagonal
AND $\forall(j, k), j-k \neq 1$	
$SCP(j, k) - SCP(j+1, k) = p$	side with non-zero differences
OR $SCP(j, k) - SCP(j, k+1) = p$	side with non-zero differences
AND $\forall(j, k), j-k \neq 1$	
$SCP(j, k) - SCP(j+1, k) = 0$	side with zero differences
OR $SCP(j, k) - SCP(j, k+1) = 0$	side with zero differences

Step 3. If $N'_p = NULL$, then no tandem array in this region. Consider another region.

Else,

N'_p is a p -periodic region. Compute parameters for the tandem array:

Step 4. Prune N'_p such that the positions along the sides of the triangle all have: $SCP(i, j) \geq p$.

$N_p \leftarrow (i, j) \in N'_p$ such that $SCP(i, j) \geq p$.

Step 5. /* compute parameters for the pm-triple */

Compute length (period) of primitive pattern \mathbf{b} : $p = \text{abslute}(p)$

Compute the idx : the index of the starting position of \mathbf{b} in the F array:

$$idx = \arg \max_{j \in N_p} \{SCP(j, j+1) \text{ s.t. } SCP(j, j+1) \leq p\},$$

Compute $t_{\mathbf{b}}$, the starting position in T using the auxiliary arrays: $t_{\mathbf{b}} = Hrs[idx]$.

Compute l = number of points on any side of the triangle.

Determine the symbol sequence in \mathbf{b} : $\mathbf{b} = T[t_{\mathbf{b}}..t_{\mathbf{b}} + p]$.

Using the auxiliary arrays from compressed sequence, this is given by

$$\mathbf{b} = F[Hrs[t_{\mathbf{b}}]]..F[Hrs[t_{\mathbf{b}} + p]].$$

Step 6. Repeat the above for all the potential repeat-regions identified in step 1.