# Indexing Similar DNA Sequences

Songbo Huang[1], T.W. Lam[1], W.K. Sung[2], S.L. Tam[1], and S.M. Yiu[1]

[1] Department of Computer Science, The University of Hong Kong, Hong Kong
{sbhuang,twlam,sltam,smyiu}@cs.hku.hk
[2] Department of Computer Science, National University of Singapore, Singapore
ksung@comp.nus.edu.sg

**Abstract.** To study the genetic variations of a species, one basic operation is to search for occurrences of patterns in a large number of very similar genomic sequences. To build an indexing data structure on the concatenation of all sequences may require a lot of memory. In this paper, we propose a new scheme to index highly similar sequences by taking advantage of the similarity among the sequences. To store $r$ sequences with $k$ common segments, our index requires only $O(n + N \log N)$ bits of memory, where $n$ is the total length of the common segments and $N$ is the total length of the distinct regions in all texts. The total length of all sequences is $rn + N$, and any scheme to store these sequences requires $\Omega(n + N)$ bits. Searching for a pattern $P$ of length $m$ takes $O(m + m \log N + m \log(rk) psc(P) + occ \log n)$, where $psc(P)$ is the number of prefixes of $P$ that appear as a suffix of some common segments and $occ$ is the number of occurrences of $P$ in all sequences. In practice, $rk \leq N$, and $psc(P)$ is usually a small constant. We have implemented our solution[1] and evaluated our solution using real DNA sequences. The experiments show that the memory requirement of our solution is much less than that required by BWT built on the concatenation of all sequences. When compared to the other existing solution (RLCSA), we use less memory with faster searching time.

## 1 Introduction

The study of genetic variations of a species often involves mining very similar genomic sequences. For example, when studying the association of SNPs (single nucleotide polymorphism) with a certain disease [3,1] in which the differences on a few characters in the genomes cause the disease, the same regions of individual genomes from different normal people and patients are extracted and compared. These different sequences are almost identical except on those SNPs. The length of each sequence can be from several million to several hundred million, and the number of copies can be up to a few hundreds.

When studying these similar sequences, a basic operation is to search the occurrences of different patterns. This seems to be straightforward as one can consider a given set of similar sequences as a single long sequence and exploit classical text indexes like suffix trees or even better, compressed indexes like BWT

[1] The software is available at http://i.cs.hku.hk/∼sbhuang/SimDNA/

B. Chen (Ed.): AAIM 2010, LNCS 6124, pp. 180–190, 2010.

(Burrows-Wheeler Transform) to perform very fast pattern searching [9, 10]. However, these indexes would demand much more memory than ordinary computers can support. Roughly speaking, a suffix tree requires more than 10 bytes per nucleotide, and BWT requires 0.5 to 1 byte per nucleotide (other compressed indexes like CSA [6] and FM-index [4, 5] have slightly higher memory requirement). Consider a case involving 250 sequences each of 200 million nucleotides. To index all these sequences, even BWT would require about 40 Gigabytes, far exceeding the capacity of a workstation. The problem of these naive solutions lies on that they do not take advantage of the high similarity of these sequences. In this paper we propose a new scheme to index highly similar sequences. It takes advantage of the similarity to obtain a very compact index, while allowing very efficient searching for any given patterns.

We first consider the following model of similarity of the input sequences. We assume that the positions in a sequence at which the symbols are different from other sequences are more or less the same for all sequences. One example is the SNP locations in a set of genes. Details are given as follows.

**Model 1:** Consider $r$ sequences $T_0, T_1, \ldots, T_{r-1}$, not necessarily of the same length. Assume that they have $k$ segments $C_0, C_1, \ldots, C_{k-1}$ in common. That is, each $T_i$ is equal to $R_{i,0}C_0R_{i,1}C_1 \ldots C_{k-1}R_{i,k}$, where $R_{i,j}$ $(0 \leq j \leq k)$ is a segment varies according to $T_i$. Note that some $R_{i,j}$ can be empty and, for any $i' \neq i$, $R_{i,j}$ and $R_{i',j}$ may not have the same length.

Below we use $n$ to denote the total length of all $C_j$'s, and use $N$ to denote the total length of all $R_{i,j}$'s over all $T_i$'s. Note that the total length of all $T_i$'s is exactly $rn + N$. Since the sequences are highly similar, the length of each $T_i$ is dominated by $n$, and $N << rn$. Furthermore, $N$ is usually larger than $rk$. The problem is to design a space-efficient index to store the sequences while allowing efficient search for any given pattern.

**Our contributions:** We develop a solution to solve the above problem by exploiting BWT and the suffix array data structures. Our solution requires $O(n + N \log N)$ bits of memory. To search a pattern $P$ of length $m$, our solution takes $O(m + m \log N + m(\log rk)psc(P) + occ \log n)$ time, where $psc(P)$ is the number of prefixes of $P$ that appear as a suffix of some common segments, and $occ$ is the total number of occurrences of $P$ in all sequences. In practice, $psc(P)$ is usually a small constant. It can also be shown that $psc(P)$ is upper bounded by $O(\log n)$ for random sequences. We implemented our solution. To store 250 versions of a sequence about 200M long, our solution only requires 2.7G memory. The memory requirement is only less than 7% of the memory required to store all 250 sequences using BWT. To search a pattern of length 500 in this collection of sequences, it takes less than 0.5ms, thus our solution is practical.

We also extend our solution to another model of similarity of the input sequences. In this model, every pair of sequences have a few positions with different nucleotides, but such positions vary from sequence to sequence. A typical example is a set of genes from closely related species. In this case, we arbitrarily take a sequence as a reference sequence, and the model is defined as follows.

**Model 2:** Consider $r$ sequences $T_0, T_1, \ldots, T_{r-1}$, all of the same length $n$. Each $T_i$ ($i \neq 0$) differs from $T_0$ in $x_i$ positions, where $x_i << n$. For $i \neq i'$, the positions at which $T_i$ is different from $T_0$ may not be the same as those of $T_{i'}$.

With minor modification, our solution can also be applied to handle Model 2 with similar space and time complexity. [12] provides a solution (referred as RLCSA) for Model 2. Their core idea is to make use of the *run-length encoding* [11] to further compress BWT by considering the maximal segments in BWT in which all symbols are the same (called *runs*). The key observation is that if we concatenate all $r$ sequences as a long sequence $T$, then construct BWT for $T$, the expected value for the number of runs $X_T$ in $T$ is bounded by $X + O(s \log L)$, where $X$ is the number of runs of BWT for $T_0$, $s$ is the sum of all $x_i$'s, and $L$ is the total length of all sequences. Based on this bound, using our notation, their space complexity is $O((n + N \log(rn + N)) \log[(rn + N)/(n + N \log(rn + N))])$, which is slightly worse than our solution. In practice, $X_T$ can be small. The searching time complexities of both our and their solutions depend on a factor which is related to the input data. We compare the two solutions based on real experiments in Section 4.

## 2 Preliminaries

We give a brief review of two indexing data structures, namely, suffix array and Burrows-Wheeler Transform (BWT) [2]. In the paper, we only consider DNA sequences which are strings of 4 symbols, {A, C, G, T}, only.

**Suffix array:** Given a text $T[0..n-1]$, we define the suffix array of $T$, denoted $SA[0..n-1]$, as follows. $SA[i] = j$ if the suffix $T[j..n-1]$ is lexicographically the $i$-th smallest suffix among all suffixes of $T$ (and we say that the *rank* of the suffix $T[j..n-1]$ is $i$). In other words, $SA$ stores the starting positions of all suffixes of $T$ in lexicographical order. For any pattern $P$, suppose $P$ appears in $T$. We define the *SA range* of $P$ with respect to $T$ as $[s, e]$ such that $s$ and $e$ are respectively the rank of the lexicographically-smallest and largest suffix of $T$ that contains $P$ as a prefix.

To find all occurrences of a pattern $P$ in $T$, we can first compute the SA range of $P$ (using $O(m \log n)$ time [7]), afterwards the occurrences of $P$ can be retrieved from the suffix array directly one by one in constant time. To store the suffix array of a text with $n$ characters, we need to store $n$ positions (more precisely, $n \log n$ bits of memory) in addition to the text. Suffix array can also be defined on a set of strings $D_0, D_1, \ldots, D_{q-1}$ as follows. $SA[i] = j$ if the string $D_j$ is lexicographically the $i$-th smallest among all given strings. The rank of $D_j$ is defined to be $i$. For any pattern $P$, suppose $P$ appears as a prefix of some $D_i$. The SA range of $P$ is defined to be $[s, e]$ where $s$ and $e$ are respectively the rank of the lexicographically-smallest and largest $D_i$ that has $P$ as a prefix.

**Burrows-Wheeler Transform (BWT):** Given a text $T[0..n-1]$, the BWT data structure, $BWT[0..n-1]$, is defined as $BWT[i] = T[j-1]$ where $j = SA[i]$ for $SA[i] \neq 0$, otherwise, set $BWT[i] =$ \$, where \$ is a special character not

in the alphabet $\Sigma$ and assumed to be lexicographically smaller than all other characters. That is, $BWT[i]$ stores the character immediately before the $i$-th smallest suffix. BWT requires only the same amount of memory as for storing the text. Using BWT and some auxiliary functions, we can compute the SA range of a given pattern of length $m$ in a backward manner (*backward search*) in $O(m)$ time [8, 9].

To retrieve the positions of an SA range, we only store part of the suffix array, called *sampled suffix array*. Intuitively, we store one SA value for every $\alpha$ entries for some constant $\alpha$. More precisely, we store the $SA[i]$ value for $i = k\alpha$ for $0 \leq k \leq \lceil \frac{n}{\alpha} \rceil$, i.e., we store the $SA[i]$ value if the rank of the suffix $T[SA[i]..n-1]$ is a multiple of $\alpha$. Retrieving the value for $SA[i]$ where $i$ is not a multiple of $\alpha$ can be done by searching repeatedly the BWT data structure [8].

## 3 Our Solution

For Model 1, recall that the input is a set of $r$ DNA sequences $T_0, T_1, \ldots, T_{r-1}$, each $T_i$ can be partitioned into $R_{i,0}C_0R_{i,1}C_1 \ldots C_{k-1}R_{i,k}$, where $C_j$ $(0 \leq j \leq k-1)$ is a common segment that all sequences agree, and $R_{i,j}$ $(0 \leq j \leq k)$ is called an R segment, which may be different for different $T_i$'s. Let $n$ be the total length of all common segments and $N$ be the total length of all R segments.

**Indexing data structure:** Let $C = C_0\$C_1\$ \ldots \$C_{k-1}\$$, where \$ is a new symbol and is lexicographically smaller than all other symbols. We store $C$ as an array of characters, together with another array $Start_C$ of the starting positions of each $C_i$ in $C$ which can be used to recover the text for any common segment (see Figure 1 for an example, (b) shows the common segments, $C$, $C^R$ together with $Start_C$). We construct a BWT index for $C^R$. Given a string $P$, we can search the BWT index using $P^R$ to determine the SA range of $P^R$, which captures all the occurrences of $P^R$ in $C^R$ (that is, all occurrences of $P$ in $C$).

Each $T_i = R_{i,0}C_0R_{i,1}C_1 \ldots C_{k-1}R_{i,k}$. We consider every suffix $d$ of $T_i$ that starts inside an $R$ segment and refer it as a *differentiating suffix*. Note that $N$ is the total number of *differentiating suffixes* for all $T_i$'s. We sort the *differentiating suffixes* of all $T_i$'s and construct a suffix array $SAR[0..N-1]$ such that $SAR[i]$ stores a reference to $i$-th lexicographically smallest differentiating suffix.

Before defining what is a reference, we need to show how each $T_i$ and its $R$ segments are represented. For each $T_i$, we only store its $R$ segments. The segments $R_{0,0}, \ldots, R_{0,k}, R_{1,0}, \ldots, R_{1,k}, \ldots, R_{r-1,0}, \ldots, R_{r-1,k}$ are stored sequentially in a character array $T[0..N-1]$. We assign a segment number to each $R_{i,j}$, which is its order in $T$. Precisely, the segment number of $R_{i,j}$ is $(k+1)i + j$. For example, $R_{0,0}$ is segment 0, $R_{0,1}$ segment 1, and $R_{1,0}$ segment $k+1$ (see Figure 1(c) for an example of all R segments). Note that a segment number $w$ can be used to identify to which $T_i$ this segment belongs, namely, $i = \lfloor w/(k+1) \rfloor$. We also construct an array $Start_T[0..r(k+1)-1]$ such that $Start_T[j]$ stores the starting position of segment $j$ in $T$. We are now ready to define a reference to a differentiating suffix. It is essentially a pair of integers, (segment number $w$, offset $o$), where $T[Start_T[w] + o]$ stores the first character of the differentiating

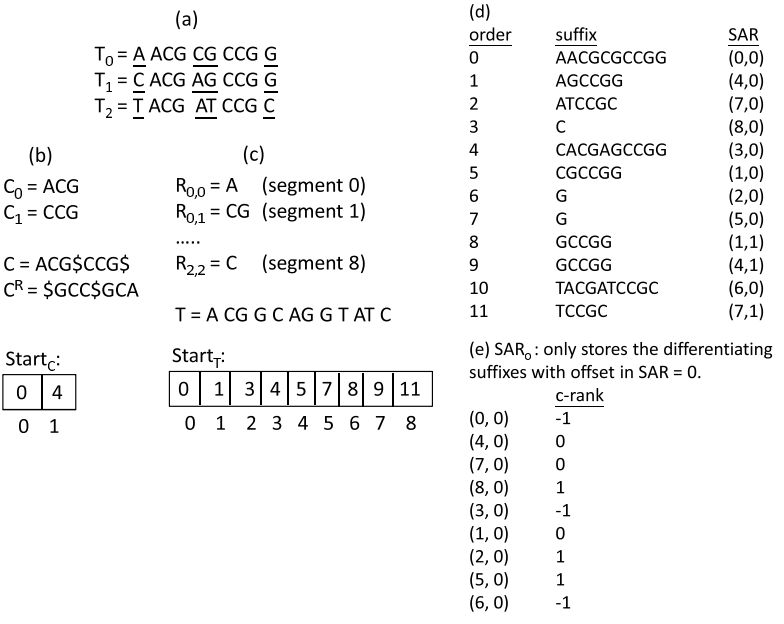


**Fig. 1.** An example for the indexing data structure. (a) shows the input sequences, the underlined segments are the R segments. (b) shows the common segments and content of $Start_C$. (c) shows the R segments and content of $Start_T$. (d) shows all differentiating suffixes in lexicographical order and also $SAR$, note that each entry in $SAR$ stores the pair (segment number, offset). (e) shows the example for $SAR_o$.

suffix. Given such a reference, we can recover every character of the suffix by referencing the corresponding segments from $T$ and $C$. Figure 1(d) shows an example of all differentiating suffixes in lexicographical order and the contents of $SAR$ for the same example.

From $SAR$, we construct a subarray $SAR_o$ that contains only those entries with offset zero. In other words, $SAR_o$ includes the differentiating suffixes starting from the first character of an R segment. To save space, we only need to store a segment number $w$ in each entry of $SAR_o$ as the offest is always zero. In addition, let $w$ correspond to the segment $R_{i,j}$, we store a number c-rank that is the rank of the suffix $\$C^R_{j-1}\$C^R_{j-2}\ldots\$C^R_0$ among all suffixes of $C^R$ (if $j = 0$, we set c-rank $= -1$). The latter is useful to determine if a pattern crosses the boundary between a common segment and an R segment of some $T_i$. See Figure 1(e) for an example for $SAR_o$.

**Space complexity:** Given $r$ similar sequences $T_0, T_1, \ldots, T_{r-1}$, each can be partitioned into $k$ common segments and $(k+1)$ $R$ segments. For $C^R$, we have a BWT index and the array $Start_C$ of $k$ entries. For $T$, we have the suffix arrays $SAR$ and $SAR_o$ and the array $Start_T$ for the starting positions of all R segments. The whole data structure requires $O(n + N \log rk + rk(\log n + \log N))$ bits.

**Searching algorithm:** Given a pattern $P$ of length $m$, if it occurs in any of the text $T_i$, there are three cases.
(1) $P$ is completely inside a common segment.
(2) $P$ is a prefix of a differentiating suffix of $T_i$ (i.e., $P$ starts in an R region).
(3) $P$ can be partitioned into non-empty substrings $P_1P_2$ where $P_1$ is a suffix of a common segment $C_j$ of $T_i$ and $P_2$ is a prefix of the following differentiating suffix in $T_i$.

For Case 1, we search the BWT index of $C^R$ for $P^R$. This takes $O(m + occ_1)$ time, where $occ_1$ is the number of Case 1 occurrences of $P$. For Case 2, we search $SAR$ for $P$. This takes $O(m \log N + occ_2)$ time, where $occ_2$ is the number of Case 2 occurrences of $P$.

Case 3 is the only non-trivial case, detailed as follows. Consider a particular partition $(P_1, P_2)$ of $P$. First, we search the BWT index of $C^R$ for the presence of $(P_1\$)^R$. Let $LR_1$ be the resulting SA range. If $LR_1$ is non-empty, $P_1$ appears as a suffix of some common segment. We then search the suffix array $SAR_o$ for $P_2$ and let $LR_2$ be the resulting SA range. For each $x$ in $LR_2$, suppose $SAR[x] = (w, \text{c-rank})$. Let $i = \lfloor w/(k+1) \rfloor$ and $t = w \mod (k+1)$. By definition, $P_2$ has an occurrence in $T_i$, starting from the first character of $R_{i,t}$. Thus, if there is a corresponding occurrence of $P_1$ at the end of the common segment $C_{t-1}$, we find a valid occurrence of $P$. We can make use of the c-rank stored in $SAR_o$ to perform the above checking in constant time. By definition, the c-rank value stored in $SAR[x]$ is the rank of the suffix $\$C^R_{t-1}\$C^R_{t-2} \ldots \$C^R_0$ with respect to all suffixes of $C^R$. Thus, if c-rank in within $LR_1$, $P_1$ must appear as a suffix of $C_{t-1}$. This is summarized in the following lemma.

**Lemma 1.** *Suppose $P$ is partitioned into $P_1P_2$ as described above. For any $x$ in $LR_2$, let $SAR[x] = (w, \text{c-rank})$. Let $i = \lfloor w/(k+1) \rfloor$ and $t = w \mod (k+1)$. Then, if c-rank $\in LR_1$, $P$ has an occurrence in $T_i$, precisely, the concatenation of the last $|P_1|$ characters of $C_{t-1}$ and the first $|P_2|$ characters of the differentiating suffix $R_{i,t}C_tR_{i,t+1} \ldots R_{i,k}$.*

Note that $C^R$, $Start_C$, $T$, and $Start_T$ are required for searching the suffix arrays $SAR$ and $SAR_o$. We put together the above ideas in Algorithm 1, which forms the basics of the searching algorithm of Case 3. Note that BWT supports backward searching. With the BWT index of $C^R$, we can compute the SA ranges for $P[0], P[0..1]^R, P[0..2]^R, \ldots P[0..m-1]^R$ incrementally using $O(m)$ time; of course, we can terminate the search as soon as an empty SA range is found. Furthermore, from the SA range of $P[0..i]^R$, we can compute the SA range of $(P[0..i]\$)^R$ in $O(1)$ time using the auxiliary functions for BWT.

**Time complexity:** Below we denote $psc(P)$ to be the number of prefixes of $P$ which are suffixes of some common segments. In an iteration where $(P[0..j]\$)^R$ is found to have a non-empty $LR_1$ (i.e., $P[0..j]$ is a suffix of some common segment), the corresponding $LR_2$ may contain up to $rk$ candidates and the verification may take $O(rk)$ time. The overall time required by Case 3 searching is $O(m + psc(P)(m \log(rk) + rk) + occ_3)$ time where $occ_3$ is the number of Case 3 occurrences of $P$.

**Algorithm 1.** *Case_3_Search*($P[0..m-1]$)

**Require:** $|P| > 0$
**Ensure:** All Case 3 occurrences of $P$ in $T_0, \ldots, T_{r-1}$

```
1: for j = 0 to m − 2 do
2:    Using BWT of C^R, find SA range LR_1 of (P[0..j]$)^R.
3:    if LR_1 is non-empty then
4:       Find SA range LR_2 of P[j + 1..m − 1] using SAR_o.
5:       for each x ∈ LR_2 do
6:          Let SAR_o[x] = (w, c-rank ).
7:          if c-rank ∈ LR_1 then
8:             Report the corresponding occurrence of P.
9:          end if
10:      end for
11:   end if
12: end for
```

The problem of Algorithm 1 is that the time required in each iteration depends on the size of $LR_2$, yet it is possible that no entries in $LR_2$ could form a valid occurrence of $P$. To speed up the checking whether the c-ranks of the entries captured by $LR_2$ fall in $LR_1$, we construct a 2-dimensional range search index [13] of the c-ranks stored in $SAR_o$. Note that the value of each c-rank is in the range $[1..n]$. Then given $LR_1$ and $LR_2$, we can find the existence of any c-ranks specified by $LR_2$ fall in the range $LR_1$ in $O(\log n)$ time, and each occurrence can be retrieved in $O(\log n)$ time. The time complexity of Case 3 becomes $O(m + psc(P)(m \log(rk)) + occ_3 \log n)$ time. The range search index requires $O(rk \log n)$ bits. The overall result is summarized in the following theorem.

**Theorem 1.** *Given $r$ similar sequences $T_0, T_1, \ldots, T_{r-1}$, each can be partitioned into $k$ common segments and $(k+1)$ $R$ segments. Let $n$ be the total length of the common segments, and let $N$ be the total length of the $R$ segments. We can build an indexing data structure using $O(n + N \log rk + rk(\log n + \log N))$ bits such that locating the occurrences of a pattern $P$ of length $m$ in the sequences can be done in $O(m + m \log N + psc(P)(m \log(rk)) + occ \log n)$ time, where $occ$ is the total number of occurrences of $P$.*

**Extension to Model 2:** Recall that in Model 2, we are given a set of $r$ DNA sequences $T_0, T_1, \ldots, T_{r-1}$, all with the *same length* $n$, each $T_i$ $(i \neq 0)$ differs from $T_0$ in $x_i$ positions. Let $N = \sum x_i$. We describe the version without using range search index. Modifying it to use range search index is straightforward.

We redefine a differentiating suffix as a suffix in $T_i$ such that this suffix starts at one of the $x_i$ positions. There are $N$ differentiating suffixes. Similar to Model 1, we store $T_0^R$ as an array of characters and construct a BWT index for $T_0^R$. Note that we do not have an array similar to $Start_C$. Then, construct a suffix array $SAR$ for all differentiating suffixes. We can define an R segment as a maximal region of characters in $T_i$ which differ from the corresponding characters in $T_0$ and label the R segments from 0 to $s-1$ where $s$ is the total number of R

segments. We store $T = R_0 R_1 \dots R_{s-1}$ as an array of characters. To recover each differentiating suffix, we need an array $Start_T$ to store the starting position of each segment in $T$ and also the starting position of this segment in its original sequence $T_i$. So, $SAR$ will store a pair of integers (segment number $w$, offset $o$) in order to reference a differentiating suffix.

From $SAR$, we construct $SAR_o$ that contains only those entries with offset zero. We construct an array $B'$ as follows. Let $d = T_i[j..n-1]$ be a differentiating suffix of $SAR_o$ and rank of $d$ in $SAR_o$ is $x$. Then, consider $T_0[0..j-1]$ and let rank of $T_0[0..j-1]^R$ with respect to $T_0^R$ be $y$. Then, we set $B'[x] = y$.

The searching algorithm is very similar. Given a pattern $P$, search $P$ in $SAR$ to locate all occurrences of $P$ which are completely inside a differentiating suffix. Search $P^R$ in $T_0^R$ using BWT to locate all occurrences of $P$ in $T_0$. The additional checking we need here is for each occurrence of $P$ in $T_0$, we check if it also occurs in the same position in each of $T_i$ for all $i > 0$. The last case is to partition $P$ into $P_1 = P[0..j]$ ($j < m-2$) and $P_2 = P[j+1..m-1]$. Then, search $P_1^R$ using BWT. Let the SA range returned be $LR_1$. Search $P_2$ using $SAR_o$ and let the SA range returned be $LR_2$. For each $x$ in $LR_2$, check if $B'[x] \in LR_1$. If yes, an occurrence of $P$ is found. The space complexity is $O(n + N \log N + s(\log n + \log N))$ bits while the time complexity is $O(m + m \log N + psc'(P)(m \log s + s) + occ)$ where $psc'(P)$ is the number of prefix of $P$ that occurs in $T_0$.

## 4 Evaluation

We first compare the memory consumption of our solution[2] based on Model 1 with the following. Concatenate all sequences to a long sequence and apply BWT directly on the long resulting sequence. We generate the texts as follow. We use two chromosomes (Chromosome Y of length 25M and Chromosome 1 of length 217M) and download the positions of SNPs for these chromosomes from NCBI[3]. For Chromosome Y, 0.1% positions are SNPs. For Chromosome 1, 0.5% positions are SNPs. Most of these SNPs are not consecutive, i.e., most of the $R$ segments are of length 1. The number of $R$ segments ($k+1$) are very similar to the number of SNPs in both cases (23,677 and 980,618 for Chromosome Y and 1 respectively). We construct four test cases. Tests 1 and 2 use Chromosome Y as Tests 3 and 4 use Chromosome 1. For Tests 1 and 3, we generate 50 texts while for Tests 2 and 4, we generate 250 texts. For each text, for each position of SNPs, we randomly generate a nucleotide. For the patterns, we randomly select them from the texts with length varying from 50 to 500. For each length, we repeat the experiment 100 times and obtain the average searching time. All experiments were conducted in a personal computer with 8G memory and a dual core 2.66GHz CPU.

For the BWT data structure for $C^R$, we use 0.75bytes per character which store 1/8 sampled SA for all our experiments. Table 1 shows the memory

[2] There are a few implementation tricks we used to speed up the searching process. Details will be given in the full paper.

[3] ftp://ftp.ncbi.nih.gov/snp/organisms/human_9606/chr_rpts

consumed by our data structure and the memory required by using BWT to store all texts. We can see that the amount of memory required by our solution is about 2-9% of that required by using BWT. In fact, the smaller the amount of SNPs, the more memory our scheme can save. For the searching performance, Table 2 shows the searching time for different pattern lengths in both Test 2 and Test 4. The searching time for Test 1 is similar to Test 2 and that for Test 3 is similar to Test 4. Note that the searching time for Test 2 is longer than that for Test 4. The reason is because of the higher percentage of SNPs in Test 4. The common segments are shorter and the searching will stop earlier (refer to Step 3 of Algorithm 1). But then even for the slower case, searching a pattern of length 500 can still be done in less than 0.5ms which is reasonably fast. This shows that our solution is practical.

**Table 1.** Memory consumption of our solution

| Chromosome | Text length | No. of Texts ($r$) | No. of SNPs | No. of common segments ($k$) | Total length of R segments ($N$) | Memory Ours | Memory BWT |
|---|---|---|---|---|---|---|---|
| (Test 1) Y | 25M | 50 | 0.0247M (0.1%) | 0.0237M | 1.2M | 35M | 958M |
| (Test 2) Y | 25M | 250 | 0.0247M (0.1%) | 0.0237M | 5.9M | 61M | 4.8G |
| (Test 3) 1 | 217M | 50 | 1.06M (0.5%) | 0.98M | 51M | 716M | 8.3G |
| (Test 4) 1 | 217M | 250 | 1.06M (0.5%) | 0.98M | 253M | 2.7G | 41.6G |

**Table 2.** Searching performance (in milliseconds) for Test 2 (250 copies of Chromosome Y (25M)) and Test 4 (250 copies of Chromosome 1 (217M))

| | Ave. Searching Time ($\times 10^{-3}$ seconds) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Pattern length | 50 | 100 | 150 | 200 | 250 | 300 | 350 | 400 | 450 | 500 |
| Test 2 | 0.083 | 0.127 | 0.170 | 0.211 | 0.252 | 0.295 | 0.334 | 0.372 | 0.410 | 0.450 |
| Test 4 | 0.079 | 0.115 | 0.145 | 0.169 | 0.192 | 0.204 | 0.219 | 0.231 | 0.241 | 0.251 |

**Table 3.** Comparison of our solution with RLCSA (RLCSA v.1 requires less memory with longer searching time; RLCSA v.2 can search faster but requires more memory. The last column shows the ratio of our average searching time over theirs based on RLCSA v.2).

| No. of Texts | Memory Comparison in M | | | | | Searching Time |
|---|---|---|---|---|---|---|
| | Ours (A) | RLCSA v.1 (B) | (A)/(B) | RLCSA v.2 (C) | (A)/(C) | (Ours/RLCSA v.2) |
| 25 | 102 | 113 | 90.3% | 156 | 65.4% | 13.2% |
| 50 | 152 | 197 | 77.2% | 277 | 54.9% | 15.7% |
| 75 | 203 | 277 | 73.3% | 396 | 51.3% | 17.6% |
| 100 | 253 | 362 | 69.9% | 514 | 49.2% | 21.5% |
| 125 | 304 | 445 | 68.3% | 623 | 48.8% | 23.2% |

We also compare our solution with RLCSA for Model 2. We use simulated data with text length 25M, mutation rate 1%. We compare their performance using different number of texts. For the patterns, we follow [12] and use shorter patterns with length 10 to 50. For each pattern length, we randomly retrieve

1000 patterns from the texts and take the average searching time. RLCSA has different settings, to compare memory consumption, we use the option which uses the smallest amount of memory (with slower searching time). When comparing the searching time, we use their option which can search faster but use more memory. The results are shown in Table 3. The results show that we use less memory and can search faster than RLCSA.

## 5 Discussion and Conclusions

Recall that the searching time of our solution depends on $psc(P)$, the number of prefixes of $P$ that is the suffix of some common segment $C_i$. If the sequences are random texts, we can show that $psc(P)$ is upper bounded by $O(\log n)$ as follows. Let $P[0..c^*]$ be the longest prefix in $P$ such that it is a suffix of some common segment, say $C_i[s - c^* + 1..s]$. It is obvious that $psc(P) = psc(C_i[s - c^* + 1..s])$. Then, for any $P$, $psc(P) \leq max_{S' \in \Delta} psc(S')$ where $\Delta$ is the set of all suffixes over all common segments. Let $Y = max_{S' \in \Delta} psc(S')$. To bound $Y$, we consider a generalized suffix tree $G$ for all common segments. For any $S' \in \Delta$, let $u$ be the node representing $S'$ in $G$, $psc(S') \leq$ node depth of $u$ which is upper bounded by $O(\log n)$ for random texts [14].

We can further reduce the space complexity of our solution to $O(n + N + rk \log(rk))$ bits, closer to the lower bound of $O(n+N)$ bits with a slightly increase in searching time. Details will be shown in the full paper. For the evaluation of our solution, we also tried different mutation rates, the performance is consistent with the ones shown in the paper. We are now investigating how to extend the scheme for approximate pattern matching.

**Acknowledgements.** The project is partially supported by the Seed Funding Programme for Basic Research of the University of Hong Kong (200811159089). We would also like to thank the authors of [12] for providing us the programs of RLCSA.

## References

1. Briniza, D., He, J., Zelikovsky, A.: Combinatorial search methods for multi-SNP disease association. In: EMBS, pp. 5802–5805 (2006)
2. Burrows, M., Wheeler, D.J.: A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, California (1994)
3. Emahazion, T., Feuk, L., Jobs, M., Sawyer, S.L., Fredman, D., Clair, D.S., Prince, J.A., Brookes, A.J.: SNP association studies in Alzheimer's disease highlight problems for complex disease analysis. Trends in Genetics 17(7), 407–413 (2001)
4. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: FOCS, pp. 390–398 (2000)
5. Ferragina, P., Manzini, G.: An experimental study of an opportunistic index. In: SODA, pp. 269–278 (2001)
6. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In: STOC, pp. 397–406 (2000)

7. Gusfield, D.: Algorithms on strings, trees, and sequences. Cambridge University Press, Cambridge (1997)
8. Kao, M.-Y. (ed.): Encyclopedia of Algorithms. Springer, Heidelberg (2008)
9. Lam, T.W., Sung, W.K., Tam, S.L., Wong, C.K., Yiu, S.M.: Compressed indexing and local alignment of DNA. Bioinformatics 24(6), 791–797 (2008)
10. Lippert, R.A.: Space-efficient whole genome comparisons with Burrows-Wheeler transforms. Journal of Computational Biology 12(4), 407–415 (2005)
11. Mäkinen, V., Navarro, G.: Succinct suffix arrays based on run-length encoding. Nordic Journal of Computing 12(1), 40–66 (2005)
12. Mäkinen, V., Navarro, G., Sirén, J., Välimäki, N.: Storage and retrieval of individual genomes. In: Batzoglou, S. (ed.) RECOMB 2009. LNCS, vol. 5541, pp. 121–137. Springer, Heidelberg (2009)
13. Nekrich, Y.: Orthogonal range searching in linear and almost-linear space. Computational Geometry: Theory and Applications 42(4), 342–351 (2009)
14. Szpankowski, W.: Probabilistic analysis of generalized suffix trees. In: CPM, pp. 1–14 (1992)