Relative FM-indexes

Djamal Bellazzougui[1,2,⋆], Travis Gagie[1,2,⋆⋆], Simon Gog[3,⋆⋆⋆], Giovanni Manzini[4], and Jouni Sirén[5,†]

[1] University of Helsinki
[2] Helsinki Institute for Information Technology
[3] Karlsruhe Institute of Technology
[4] University of Eastern Piedmont
[5] University of Chile

Abstract. Intuitively, if two strings S_1 and S_2 are sufficiently similar and we already have an FM-index for S_1 then, by storing a little extra information, we should be able to reuse parts of that index in an FM-index for S_2. We formalize this intuition and show that it can lead to significant space savings in practice, as well as to some interesting theoretical problems.

1 Introduction

FM-indexes [4] are core components in most modern DNA aligners (e.g., [8–10]) and have thus played an important role in the genomics revolution. Medical researchers are now producing databases of hundreds or even thousands of human genomes, so bioinformatics researchers are working to improve FM-indexes' compression of sets of nearly duplicate strings. As far as we know, however, the solutions proposed so far (e.g., [3, 11]) index the concatenation of the genomes, so we can search the whole database easily but searching only in one specified genome is more difficult. In this paper we consider how to index each of the genomes individually while still using reasonable space and query time.

Our intuition is that if two strings S_1 and S_2 are sufficiently similar and we already have an FM-index for S_1 then, by storing a little extra information, we should be able to reuse parts of that index in an FM-index for S_2. More specifically, it seems S_1's and S_2's Burrows-Wheelers Transforms [2] (BWTs) should also be fairly similar. Since BWTs are the main component of FM-indexes, it is natural to try to take advantage of such similarity to build an index for S_2 that "reuses" information already available in S_1's FM-index.

Among the many possible similarities one can find and exploit in the BWTs, in this paper we consider the longest common subsequence (LCS). The BWT

⋆ Partially supported by Academy of Finland under grant 250345 (CoECGR).
⋆⋆ Funded by Academy of Finland grant 268324.
⋆⋆⋆ This work was carried out while the third author was employed at the University of Melbourne, supported by ARC Grant DP110101743.
† Funded by the Jenny and Antti Wihuri Foundation, Finland, and Fondecyt Grant 1-140796, Chile.

sorts the characters of a string into the lexicographic order of the suffixes following those characters. For example, if

$$S_1 = \mathsf{GCACTTAGAGGTCAGT}, \qquad S_2 = \mathsf{GCACTAGACGTCAGT};$$

then

$$\mathsf{BWT}(S_1) = \mathsf{TCTGCGTAAAAGGTGC}, \qquad \mathsf{BWT}(S_2) = \mathsf{TGCTCGTAAAACGCG};$$

whose LCS $\mathsf{TCTCGTAAAAGG}$ is nearly as long as either BWT.

We introduce the concept of *BW-distance* $\mathsf{BWD}(S_1, S_2)$ between S_1 and S_2 defined as $|S_1| + |S_2| - 2|\mathsf{LCS}(\mathsf{BWT}(S_1), \mathsf{BWT}(S_2))|$. Note that this coincides with the edit distance between $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ when only insertions and deletions are allowed. We prove that, if we are willing to tolerate a slight increase in query times, we can build an index for S_2 using an unmodified FM-index for S_1 and additional data structures whose total space in words is asymptotically bounded by $\mathsf{BWD}(S_1, S_2)$ (Theorem 1).

This first result is the starting point for our investigation as it generates many challenging issues. First, since we are interested in indexing whole genomes, we observe that finding the LCS of strings whose length is of the order of billions is outside the capabilities of most computers. Thus, in Section 3.1 we show how to approximate the LCS of two BWTs, using combinatorial properties of the BWT to align the sequences. In the same section we also discuss and test several practical alternatives for building the index for S_2 given the one for S_1 and we analyze their time/space trade-offs.

If we need an index not only for counting queries but also for locating and extracting, we must enrich it with suffix array (SA) samples. Such samples usually take significantly less space than the main index. However, we may still want to take advantage of the similarities between S_1 and S_2 to "reuse" SA samples from S_1 for S_2's index. In Section 4 we show that this is indeed possible if, instead of considering the LCS between the BWTs, we use a common subsequence with the additional constraint of being *BWT-invariant* (Theorem 2). This result motivates the problem of finding the longest BWT-invariant subsequence, which unfortunately turns out to be NP-hard (Theorem 3). We therefore devise a heuristic to find a "long" BWT-invariant subsequence in $\mathcal{O}(|S_1| \log |S_1|)$ time.

We have tested our approach in practice by building an FM-index for the genomes of two human individuals, "reusing" an FM-index of the human reference genome. The reference genome is 3096 million base pairs, the individual genomes are 3002 million and 3036 million base pairs, and we found common subsequences of 2935 million and 2992 million base pairs, respectively. Our index is 3.8–5.0 times or 2.2–2.9 times smaller than a standard implementation of a stand-alone FM-index, depending on the encoding of the stand-alone index. On the other hand, queries to our index take about 11 times or 1.9 times longer, respectively. Since our index is compressed relative to the underlying index for the reference, we call it a relative FM-index.

2 Review of the FM-index structure

The core component of an FM-index for a string $S[1..n]$ is a data structure supporting rank queries on the Burrows-Wheeler Transform $\mathsf{BWT}(S)$ of S. This transform permutes the characters in S such that $S[i]$ comes before $S[j]$ in $\mathsf{BWT}(S)$ if $S[i+1..n]$ is lexicographically less than $S[j+1..n]$.

If the lexicographic range of suffixes of S starting with β is $[i..j]$, then the range of suffixes starting with $a\beta$ is

$$\left[\mathsf{BWT}(S).\mathsf{rank}_a(i-1) + 1 + \sum_{a' \prec a} S.\mathsf{rank}_{a'}(n).. \mathsf{BWT}(S).\mathsf{rank}_a(j) + \sum_{a' \prec a} S.\mathsf{rank}_{a'}(n)\right]$$

It follows that, if we have precomputed an array storing $\sum_{a' \prec a} S.\mathsf{rank}_{a'}(n)$ for each distinct character a (i.e., the number of characters in S less than a), then we can find the range of suffixes starting with a pattern $P[1..m]$ — and, thus, count its occurrences — using $\mathcal{O}(m)$ rank queries.

If the position of $S[i]$ in $\mathsf{BWT}(S)$ is j, then the position of $S[i-1]$ is

$$\mathsf{BWT}(S).\mathsf{rank}_{S[i]}(j) + \sum_{a \prec S[i]} \mathsf{BWT}(S).\mathsf{rank}_a(n)\,.$$

It follows that, if we have also precomputed a dictionary storing the position of every rth character of S in $\mathsf{BWT}(S)$ with its position in S as satellite information, then we can find a character's position in S from its position in $\mathsf{BWT}(S)$ using $\mathcal{O}(r)$ rank and membership queries. Therefore, once we know the lexicographic range of suffixes starting with P, we can locate each of its occurrences using $\mathcal{O}(r)$ rank queries.

Finally, if we have also precomputed an array storing the position of every rth character of S in $\mathsf{BWT}(S)$, in order of appearance in S, then given i and j, we can extract $S[i..j]$ using $\mathcal{O}(r + j - i)$ rank queries.

3 BW-distance and relative FM-indices

Given two strings $S_1[1..n_1]$ and $S_2[1..n_2]$ we define the *BW-distance* $\mathsf{BWD}(S_1, S_2)$ between S_1 and S_2 as

$$\mathsf{BWD}(S_1, S_2) = n_1 + n_2 - 2|\mathsf{LCS}(\mathsf{BWT}(S_1), \mathsf{BWT}(S_2))|. \tag{1}$$

Note that the BW-distance is nothing but the edit distance between $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ when only insertions and deletions are allowed [13] (also known as the shortest edit script or indel distance), and is thus at most twice their normal edit distance. We now show how to support counting queries on S_2 using an FM-index for S_1 and some auxiliary data structures taking $\mathcal{O}(\mathsf{BWD}(S_1, S_2))$

words of space. Specifically, we consider how we can support rank queries on $\mathsf{BWT}(S_2)$ and partial-sum queries on the distinct characters' frequencies.

Let C denote a LCS of $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ with $|C| = m$. Let $C = c_1 \cdots c_m$, and for $i = 1, \ldots, m$, let α_i (resp. β_i) be the position of c_i in $\mathsf{BWT}(S_1)$ (resp. $\mathsf{BWT}(S_s)$) with $\alpha_1 < \cdots < \alpha_m$ (resp. $\beta_1 < \cdots < \beta_m$). Define

- bitvector $B_1[1..n_1]$ with 0s in positions $\alpha_1, \ldots, \alpha_m$,
- bitvector $B_2[1..n_2]$ with 0s in positions of $\beta_1, \ldots, \beta_m$,
- subsequence D_1 of $\mathsf{BWT}(S_1)$ marked by 1s in B_1; D_1 is the complement of C in $\mathsf{BWT}(S_1)$,
- subsequence D_2 of $\mathsf{BWT}(S_2)$ marked by 1s in B_2; D_2 is the complement of C in $\mathsf{BWT}(S_2)$.

We claim that if we can support fast rank queries on $\mathsf{BWT}(S_1)$, B_1, B_2, D_1 and D_2 and fast select_0 queries on B_1, then we can support fast rank queries on $\mathsf{BWT}(S_2)$. To see why, notice that

$$\begin{aligned}\mathsf{BWT}(S_2).\mathsf{rank}_X(i) = {} & C.\mathsf{rank}_X(B_2.\mathsf{rank}_0(i)) \\ & + D_2.\mathsf{rank}_X(B_2.\mathsf{rank}_1(i))\end{aligned}$$

and, by the same reasoning,

$$\begin{aligned}C.\mathsf{rank}_X(j) = {} & \mathsf{BWT}(S_1).\mathsf{rank}_X(B_1.\mathsf{select}_0(j)) \\ & - D_1.\mathsf{rank}_X(B_1.\mathsf{rank}_1(B_1.\mathsf{select}_0(j)))\,.\end{aligned}$$

Therefore,

$$\begin{aligned}\mathsf{BWT}(S_2).\mathsf{rank}_X(i) = {} & \mathsf{BWT}(S_1).\mathsf{rank}_X(k) \\ & - D_1.\mathsf{rank}_X(B_1.\mathsf{rank}_1(k)) \\ & + D_2.\mathsf{rank}_X(B_2.\mathsf{rank}_1(i))\end{aligned}$$

where $k = B_1.\mathsf{select}_0(B_2.\mathsf{rank}_0(i))$.

For example, for the strings

$$S_1 = \mathsf{GCACTTAGAGGTCAGT}, \qquad S_2 = \mathsf{GCACTAGACGTCAGT}$$

given as an example in Section 1,

$$\mathsf{BWT}(S_1) = \mathsf{TCTGCGTAAAAGGTGC}, \qquad \mathsf{BWT}(S_2) = \mathsf{TGCTCGTAAAACGCG};$$

and $\mathsf{LCS}(\mathsf{BWT}(S_1), \mathsf{BWT}(S_2)) = \mathsf{TCTCGTAAAAGG}$ so

$$\begin{aligned}B_1 &= 0001000000000111 & D_1 &= \mathsf{GTGC} \\ B_2 &= 0100000000001010 & D_2 &= \mathsf{GCC}\,.\end{aligned}$$

Suppose we want to compute $\mathsf{BWT}(S_2).\mathsf{rank}_\mathsf{C}(13)$. Since $B_1.\mathsf{select}_0(B_2.\mathsf{rank}_0(13)) = 12$,

$$\begin{aligned}\mathsf{BWT}(S_2).\mathsf{rank}_\mathsf{C}(13) = {} & \mathsf{BWT}(S_1).\mathsf{rank}_\mathsf{C}(12) \;-\; D_1.\mathsf{rank}_\mathsf{C}(B_1.\mathsf{rank}_1(12)) \\ & + D_2.\mathsf{rank}_\mathsf{C}(B_2.\mathsf{rank}_1(13)) \;=\; 3.\end{aligned}$$

Observing that the number of 1s in B_1 and B_2 is $\mathcal{O}(\max(n_1, n_2) - m) = \mathcal{O}(\mathsf{BWD}(S_1, S_2))$, we can store data structures for B_1, B_2, D_1 and D_2 in $\mathcal{O}(\mathsf{BWD}(S_1, S_2))$ space such that the desired rank/select queries take $\mathcal{O}(\log \mathsf{BWD}(S_1, S_2))$ time.

The only other component required for an FM-index for S_2 for counting, is a data structure for computing $\sum_{a' \prec a} S_2.\mathsf{rank}_{a'}(n)$ for each character a. Notice that $\mathsf{BWD}(S_1, S_2)$ is at least the number of distinct characters whose frequencies in S_1 and S_2 differ. It follows that in $\mathcal{O}(\mathsf{BWD}(S_1, S_2))$ space we can store

- a $\mathcal{O}(\log \mathsf{BWD}(S_1, S_2))$-time predecessor data structure storing those distinct characters,
- an array storing $\sum_{a' \prec a} S_2.\mathsf{rank}_{a'}(n_2)$ for each such distinct character a.

For any distinct character b, we can find the preceding distinct character a whose frequencies in S_1 and S_2 differ and compute

$$\sum_{a' \prec b} S_2.\mathsf{rank}_{a'}(n_2) = \sum_{a' \prec b} S_1.\mathsf{rank}_{a'}(n_1) - \sum_{a' \prec a} S_1.\mathsf{rank}_{a'}(n_1) + \sum_{a' \prec a} S_2.\mathsf{rank}_{a'}(n_2)$$

using $\mathcal{O}(\log \mathsf{BWD}(S_1, S_2))$ time. Summing up:

Theorem 1. *If we have an FM-index for S_1, we can store a relative FM-index for S_2 using $\mathcal{O}(\mathsf{BWD}(S_1, S_2))$ words of extra space. Counting queries on the relative FM-index take time an $\mathcal{O}(\log \mathsf{BWD}(S_1, S_2))$ factor larger than on S_1.*

3.1 A practical implementation

A longest common sequence of $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ can be computed in $\mathcal{O}(n_1 n_2/w)$ time, where w is the word size [12]. Since we are mainly interested in strings with a small BW-distance, a better alternative could be the algorithms whose running times are bounded by the number of differences between the input sequences (see eg [7, 13]). Unfortunately none of these algorithms is really practical when working with such very large files as the complete genomes we considered in our tests. Hence, to make the construction of a relative FM-index practical, we approximate the LCS of the two Burrows-Wheeler transforms, using the combinatorial properties of the BWT to align the sequences.

Let S_1 be a random string of length n over alphabet Σ of size σ, and let string S_2 differ from it by s insertions, deletions, and substitutions. In the expected case, the lexicographic rank of each suffix of S_1 is determined by a prefix of length $\mathcal{O}(\log_\sigma n)$ of that suffix. Thus, the s edit operations are expected to affect the relative lexicographic order of $\mathcal{O}(s \log_\sigma n)$ suffixes [11], possibly causing the characters immediately preceding those suffixes to appear in different positions in $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$. The edits can also change the characters immediately preceding at most s suffixes. If we remove the characters preceding the affected suffixes from $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$, we have a common subsequence of length $n - O(s \log_\sigma n)$ in the expected case.

Assume that we have partitioned the BWTs according to the first k characters of the suffixes, for $k \geq 0$. For all $x \in \Sigma^k$, let $\mathsf{BWT}_x(S_1)$ and $\mathsf{BWT}_x(S_2)$ be

the substrings of the BWTs corresponding to the suffixes starting with x. If we remove the suffixes affected by the edit operations, as well as the suffixes where string x covers an edit, we have a common subsequence BWT'_x of $\mathsf{BWT}_x(S_1)$ and $\mathsf{BWT}_x(S_2)$. If we concatenate the sequences BWT'_x for all x, we get a common subsequence of $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ of length $n - O(s(k + \log_\sigma n))$ in the expected case. This suggests that we can find a long common subsequence of $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ by partitioning the BWTs, finding an LCS for each partition, and concatenating the results.

In practice, we partition the BWTs by variable-length strings. We use backward searching on the BWTs to traverse the suffix trees of S_1 and S_2, selecting a partition when either the length of $\mathsf{BWT}_x(S_1)$ or $\mathsf{BWT}_x(S_2)$ is at most 1024, or the length of the pattern x reaches 32. For each partition, we use the greedy LCS algorithm [13] to find the longest common subsequence of that partition. To avoid hard cases, we stop the greedy algorithm if it would need diagonals beyond ± 50000, and match only the most common characters for that partition. We also predict in advance the common cases where this happens (the difference of the lengths of $\mathsf{BWT}_x(S_1)$ and $\mathsf{BWT}_x(S_2)$ is over 50000, or $x = \mathsf{N}^{32}$ for DNA sequences, where N is the "any base" symbol), and match the most common characters in that partition directly.

We implemented the counting structure of the relative FM-index using the SDSL library [5], and compared its performance to a regular FM-index.[6] To encode the BWTs and sequences D_1 and D_2, we used Huffman-shaped wavelet trees with either plain or entropy-compressed (RRR) [15] bitvectors. For marking the positions of the LCS in $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$, we used either entropy-compressed or sparse [14] bitvectors.

The implementation was written in C++ and compiled on g++ version 4.7.3. We used a system with 32 gigabyes of memory and two quad-core 2.53 GHz Intel Xeon E5540 processors, running Ubuntu 12.04 with Linux kernel 3.2.0. Only one CPU core was used in the experiments.

For our experiments, we used the 1000 Genomes Project assembly of the human reference genome as sequence S_1.[7] As sequence S_2, we used the genome of a Han Chinese male from the YanHuang project[8], and the genome of the 1000 Genomes Project individual NA12878 (Utah female, maternal haplotype) [16]. The properties of the datasets can be seen in Table 1. As our pattern set, we used 3.68 million reads of length 108 from the 1000 Genomes Project individual HG00122 (British female). The results of the experiments can be seen in Table 2.

The fastest variant of the relative FM-index uses plain bitvectors in the wavelet trees and RRR bitvectors for the LCS. It is 3.8–5.0 times smaller and 11 times slower than a regular FM-index using plain bitvectors, and 2.2–2.9 times smaller and 1.9 times slower than a regular index using RRR bitvectors. Switching to compressed bitvectors in the wavelet trees does not yield a good trade-off. Using sparse bitvectors for the LCS is a slightly better option on the

[6] The implementation is available at http://jltsiren.kapsi.fi/relative-fm

[7] GRCh37, ftp://ftp-trace.ncbi.nih.gov/1000genomes/ftp/technical/reference/

[8] ftp://public.genomics.org.cn/BGI/yanhuang/fa/

Table 1. Properties of the datasets. The length of the sequence and the common subsequence of the BWTs; the number of matching patterns and the total number of occurrences for those patterns.

Dataset	Length	LCS	Matches	Occurrences
Reference	3096M	–	–	–
YanHuang	3002M	2935M	1.14M	5.49M
NA12878	3036M	2992M	1.21M	5.67M

Table 2. Experiments with human genomes. Dataset, bitvector used in the wavelet trees (WT) and for the LCS; time and space requirements for building the relative FM-index; time required for counting queries and index size for a regular FM-index and a relative FM-index. The query times are averages over five runs.

			Construction		Regular		Relative	
Dataset	WT	LCS	Time [s]	Space [MB]	Time [s]	Size [MB]	Time [s]	Size [MB]
YanHuang	Plain	RRR	708	9124	56.45	1090	621.47	288
YanHuang	Plain	Sparse	711	9124	56.45	1090	1162.47	290
YanHuang	RRR	RRR	5898	7823	328.86	628	1637.44	256
YanHuang	RRR	Sparse	5882	7823	328.86	628	1994.89	257
NA12878	Plain	RRR	589	9124	57.31	1090	619.81	218
NA12878	Plain	Sparse	575	9124	57.31	1090	1058.75	199
NA12878	RRR	RRR	5454	7823	325.49	636	1614.56	192
NA12878	RRR	Sparse	5412	7823	325.49	636	1921.92	173

NA12878 dataset, making the relative index 1.1 times smaller and 1.7 times slower. On the YanHuang dataset, sparse bitvectors require more space than RRR bitvectors, because the sequence is too different from the reference.

Bitvectors B_1 and B_2 take 70% to 85% of the total size of the relative index, so improving their compression may be the best way to make the index smaller. Hybrid bitvectors using different encodings for different parts of the bitvector [6] could be one option, but the existing implementation does not work with vectors longer than 2^{31} bits. It should be noted that the size difference between NA12878 and the reference is mostly due to the inclusion of chromosome Y (59 million base pairs) in the reference. Therefore we can expect the relative FM-index to work significantly better with male genomes than female genomes.

Building a relative FM-index out of regular FM-indexes for two human genomes takes 10–12 minutes. Using RRR bitvectors for the wavelet trees increases this to 90–98 minutes, as extracting substrings from the wavelet trees becomes the bottleneck. Decompressing the regular FM-index of S_2 from the regular index of S_1 and the relative index of S_2 should be even faster. As a comparison, building BWT for a human genome takes 19–20 minutes and 25–26

gigabytes of memory using libdivsufsort 2.0.1[9], depending on the sequence. The space usage of relative FM-index construction has not been optimized, and it can probably be improved significantly.

4 Relative FM-indices supporting locating and extracting

As mentioned in Section 2, an FM-index for S_1 usually has an SA sample that takes an only slightly sublinear number of bits. This sample has two parts: the first consists of a bitvector R with 1s marking the positions in $\mathsf{BWT}(S_1)$ of every rth character in S_1, and an array A storing a mapping from the ranks of those characters' positions in $\mathsf{BWT}(S_1)$ to their positions in S_1; the second is an array storing a mapping from the ranks of those characters' positions in S to their positions in $\mathsf{BWT}(S_1)$. With these, given the position of a sampled character in $\mathsf{BWT}(S_1)$, we can find its position in S_1, and vice versa.

These parts are used for locating and extracting queries, respectively, and the worst-case query times are proportional to r. On the other hand, the size of the sample in words is proportional to the length of S divided by r. For details on how the sample works, we direct the reader to the full description of FM-indexes [4]. We note only that if we sample irregularly, then the worst-case query times for locating and extracting are proportional to the maximum distance in S between two consecutive sampled characters. We leave consideration of extracting for the full version of the paper — it is nearly symmetric to locating — so we do not discuss the second part of the sample here.

Let $G = S_1[i_1] \cdots S_1[i_\ell]$ denote a length-ℓ common subsequence of S_1 and S_2 (not their BWTs). That is, we have $i_1 < \cdots < i_\ell$ and there exists $j_1 < \cdots < j_\ell$ such that

$$S_1[i_1] = S_2[j_1], \ldots, S_1[i_\ell] = S_2[j_\ell].$$

Since there is a one-to-one correspondence between the characters in a text and in its BWT, we can define the indexes $v_1, \ldots, v_\ell$ (resp. $w_1, \ldots, w_\ell$) such that for $k = 1, \ldots, \ell$, $\mathsf{BWT}(S_1)[v_k]$ is the character corresponding to $S_1[i_k]$ (resp. $\mathsf{BWT}(S_2)[w_k]$ is the character corresponding to $S_2[j_k]$). We say that the common subsequence G is *BWT-invariant* if there exists a permutation $\pi : \{1, \ldots, \ell\} \to \{1, \ldots, \ell\}$ such that we have simultaneously

$$v_{\pi(1)} < v_{\pi(2)} < \cdots < v_{\pi(\ell)}, \quad \text{and} \quad w_{\pi(1)} < w_{\pi(2)} < \cdots < w_{\pi(\ell)}. \tag{2}$$

In other words, when we go from the texts to the BWTs the elements of G are permuted in the same way in S_1 and S_2.

An immediate consequence of (2) is that the sequence

$$G' = \mathsf{BWT}(S_1)[v_{\pi(1)}]\,\mathsf{BWT}(S_1)[v_{\pi(2)}] \cdots \mathsf{BWT}(S_1)[v_{\pi(\ell)}]$$

[9] https://code.google.com/p/libdivsufsort/

is a common subsequence of $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$. We can therefore generalize (1) and define

$$\mathsf{BWD}_G(S_1, S_2) = \max(n_1, n_2) - |G|$$

and repeat the construction of Theorem 1 with BWD replaced by BWD_G. However, since G is BWT-invariant it is now possible to reuse the the SA samples from S_1 relative to positions in G for the string S_2 provided that we have

- bitvector $M_1[1..n_1]$ with 0s in positions $i_1, \ldots, i_\ell$, supporting fast rank queries,
- bitvector $M_2[1..n_2]$ with 0s in positions $j_1, \ldots, j_\ell$, supporting fast select_0 queries.

Summing up, we have (proof idea in the appendix):

Theorem 2. *For any BWT-invariant subsequence G, if we already have an FM-index for S_1, then we can store $\mathcal{O}(\mathsf{BWD}_G(S_1, S_2))$ extra words such that the time bounds for locating and extracting queries on S_2 are an $\mathcal{O}(\log \mathsf{BWD}_G(S_1, S_2))$ factor larger than on S_1.*

In view of the above theorem, it is certainly desirable to find the longest common subsequence of S_1 and S_2 which is BWT-invariant. Unfortunately, this problem is NP-hard as shown by the following result.

Theorem 3. *It is NP-complete to determine whether there is an LCS of S_1 and S_2 which is BWT-invariant, even when the strings are over a ternary alphabet.*

Proof. Clearly we can check in polynomial time whether a given subsequence of S_1 and S_2 has this property, so the problem is in NP. To show that it is NP-complete, we reduce from the NP-complete problem of permutation pattern matching [1], for which we are given two permutations π_1 and π_2 over n and $m \leq n$ elements, respectively, and asked to determine whether there is a subsequence of π_1 of length m such that the relative order of the elements in that subsequence is the same as the relative order of the elements in π_2. For example, if $\pi_1 = 6, 3, 2, 1, 4, 5$ and $\pi_2 = 4, 2, 1, 3$, then $6, 2, 1, 4$ is such a subsequence. Specifically, we set

$$S_1 = \mathsf{AB}^{\pi_1[1]}\mathsf{AB}^{\pi_1[2]} \cdots \mathsf{AB}^{\pi_1[n]}, \qquad S_2 = \mathsf{AC}^{\pi_2[1]}\mathsf{AC}^{\pi_2[2]} \cdots \mathsf{AC}^{\pi_2[m]},$$

so the unique LCS of S_1 and S_2 is A^m. For our example,

$$S_1 = \mathsf{AB}^6\mathsf{AB}^3\mathsf{AB}^2\mathsf{ABAB}^5 = \mathsf{ABBBBBBABBBABBABABBBBB}$$
$$S_2 = \mathsf{AC}^4\mathsf{AC}^2\mathsf{ACAC}^3 = \mathsf{ACCCCACCACACCC}.$$

The BWT sorts the m copies of A in S_2 according to π_2 and sorts any subsequence of m copies of A in S_1 according to the corresponding subsequence of π_1. Therefore, there is an LCS of S_1 and S_2 such that the relative order of its characters is $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ is the same, if and only if there is a subsequence of π_1 of length m such that the relative order of the elements in that subsequence is the same as the relative order of the elements in π_2. □

In view of the above result, for large inputs we cannot expect to find the longest possible BWT-invariant subsequence, so, as for the LCS, we have devised the following fast heuristic for computing a "long" BWT-invariant subsequence.

We first compute the suffix array SA_{12} for the concatenation $S_1\#S_2$ and we use it to define the array A of size $n_1 \times 2$ as follows

- $A[i][1] = j$ iff $S_1[i] = S_2[j]$ and suffix $S_2[j+1, n_2]$ immediately follows suffix $S_1[i+1, n_1]$ in SA_{12}. If no such j exists $A[i][1]$ is undefined.
- $A[i][2] = j$ iff $S_1[i] = S_2[j]$ and suffix $S_2[j+1, n_2]$ is the lexicographically largest suffix of S_2 preceding suffix $S_1[i+1, n_1]$ in SA_{12}. If no such j exists $A[i][2]$ is undefined.

Next, we compute the longest subsequence $1 \le i_1 < i_2 < \cdots < i_\ell \le n_1$ such that there exist $b_1, \ldots, b_\ell$, with $b_k \in \{1, 2\}$ and the sequence

$$A[i_1][b_1] < A[i_2][b_2] < \cdots < A[i_\ell][b_\ell]$$

is the longest possible (every $A[i_k][b_k]$ must be defined). The values $i_1, \ldots, i_\ell$ and $b_1, \ldots, b_\ell$ can be computed in $\mathcal{O}(n_1 \log n_1)$ time using a straightforward modification of the dynamic programming algorithm for the longest increasing subsequence. Setting, for $k = 1, \ldots, \ell$, $j_k = A[i_k][b_k]$ we get that

$$G \;=\; S_1[i_1]S_1[i_2]\cdots S_1[i_\ell] \;=\; S_2[j_1]S_2[j_2]\cdots S_2[j_\ell]$$

is a common subsequence of S_1 and S_2.

Lemma 1. *The subsequence G is BWT-invariant.*

Proof. Let $v_1, \ldots, v_\ell$ (resp. $w_1, \ldots, w_\ell$) such that for $k = 1, \ldots, \ell$, $\mathsf{BWT}(S_1)[v_k]$ is the character corresponding to $S_1[i_k]$ (resp. $\mathsf{BWT}(S_2)[w_k]$ corresponds to $S_2[j_k]$). It suffices to prove that for any pair h, k, with $1 \le h, k \le \ell$, the inequality $v_h < v_k$ implies $w_h < w_k$. Let $\prec$ denote the lexicographic order. By construction, and by the properties of the BWT, we have $v_h < v_k$ iff the suffix $S_1[i_h + 1, n_1] \prec S_1[i_k + 1, n_1]$ and we must prove that this implies $S_2[j_h + 1, n_2] \prec S_2[j_k + 1, n_2]$.

Since $j_h = A[i_h][b_h]$ and $j_k = A[i_k][b_k]$, the proof follows considering the four possible cases: $b_h, b_k \in \{1, 2\}$. We consider the case $b_h = 1$, $b_k = 2$ leaving the others to the reader. If $j_h = A[i_h][1]$ and $j_k = A[i_k][2]$ then $S_2[j_h + 1, n_2]$ immediately follows $S_1[i_h + 1, n_1]$ in SA_{12}. At same time $S_2[j_k + 1, n_2]$ precedes $S_1[i_k + 1, n_1]$ but there are no other suffixes from S_2 between them. Since $j_h \neq j_k$ the only possible ordering of the suffixes in SA_{12} is

$$S_1[i_h + 1, n_1] \;\prec\; S_2[j_h + 1, n_2] \;\prec\; S_2[j_k + 1, n_2] \;\prec\; S_1[i_k + 1, n_1]$$

implying $S_2[j_h + 1, n_2] \prec S_2[j_k + 1, n_2]$ as claimed. □

To evaluate whether the subsequence G derived from the above procedure is still able to capture the similarity between S_1 and S_2, we have compared the length of G with the LCS length for pairs of *S.cerevisiae* genomes from the

Table 3. Comparison between $|G|$ and $|\mathsf{LCS}|$. The normalizing factor n is the length of sequence 273614N.

	322134S	378604X	BC187	DBVPG1106
$\|\mathsf{LCS}\|/n$	0.9341	0.9669	0.9521	0.9590
$\|G\|/n$	0.8694	0.8655	0.8798	0.8800

Saccharomyces Genome Resequencing Project.[10] In particular we compared the 273614N sequence with sequences 322134S, 378604X, BC187, and DBVPG1106. For each sequence we report in Table 3 the ratio between the length of G and $\mathsf{LCS}(\mathsf{BWT}(S_1), \mathsf{BWT}(S_2))$ and the length of sequence 273614N (roughly 11.9 MB). We see that in all cases more than 85% of BWT positions are in G which roughly indicates that more than 85% of the SA samples from 273614N could be reused as SA samples for the other sequences.

5 Conclusions

In this paper we have considered the problem of building an index for a string S_2 given an FM-index for a similar string S_1. We have shown how to build such a "relative" index using space bounded by the BW-distance between S_1 and S_2. The BW-distance is simply the edit distance between $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ when only insertions and deletions are allowed. We have also introduced the notion of BWT-invariant subsequence and shown that it can be used to determine a set of S_1 suffix array samples that can be easily "reused" for an index for S_2.

We have tested our approach by building a relative index for a Han Chinese individual and a 1000 Genomes Project individual with respect to an FM-index of the human reference genome. We leave as a future work the development of these ideas and the complete implementation of a relative FM-index supporting locating and extracting. We also leave as future work proving bounds on the BW-distance and the length of the longest BWT-invariant subsequence in terms of the edit distance of the strings.

References

1. P. Bose, J. F. Buss, and A. Lubiw. Pattern matching for permutations. *Inf. Process. Lett.*, 65(5):277–283, 1998.
2. M. Burrows and D. J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
3. H. Ferrada, T. Gagie, T. Hirvola, and S. J. Puglisi. Hybrid indexes for repetitive datasets. *Phil. Trans. Royal Society A*, 372(2016), 2014.
4. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[10] https://www.sanger.ac.uk/research/projects/genomeinformatics/sgrp.html

5. S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *Proc. 13th International Symposium on Experimental Algorithms (SEA 2014)*, 2014. To appear.
6. J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proc. 2014 IEEE Data Compression Conference (DCC 2014)*, 2014. To appear.
7. G. M. Landau, U. Vishkin, and R. Nussinov. An efficient string matching algorithm with k differences for nucleotide and amino acid sequences. *Nucleic Acids Research*, 14(1):31–46, 1986.
8. B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10:R25, 2009.
9. H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.
10. R. Li, C. Yu, Y. Li, T.-W. Lam, S.-M. Yiu, K. Kristiansen, and J. Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.
11. V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
12. E. W. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.
13. Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
14. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, pages 60–70. SIAM, 2007.
15. R. Raman, V. Raman, and S. Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4):43, 2007.
16. Joel Rozowsky, Alexej Abyzov, Jing Wang, Pedro Alves, Debasish Raha, Arif Harmanci, Jing Leng, Robert Bjornson, Yong Kong, Naoki Kitabayashi, Nitin Bhardwaj, Mark Rubin, Michael Snyder, and Mark Gerstein. AlleleSeq: analysis of allelespecific expression and binding in a network framework. *Molecular Systems Biology*, 7:522, 2011.

Appendix: Reusing an SA Sample

Consider the strings S_1, S_2 used as example in Section 3, and the corresponding LCS $C = \mathsf{LCS}(\mathsf{BWT}(S_1), \mathsf{BWT}(S_2))$ and bitvectors B_1 and B_2. The characters of $\mathsf{BWT}(S_1)[1..16]$ and $\mathsf{BWT}(S_2)[1..15]$ are mapped to their positions by the BWT from

$$S_1[16, 2, 6, 8, 13, 1, 12, 3, 7, 9, 14, 10, 15, 5, 11, 4]$$
$$S_2[15, 7, 2, 5, 12, 1, 11, 8, 3, 6, 13, 9, 14, 4, 10]$$

respectively. Notice the lists of indices are just the SAs of $S_1\$$ and $S_2\$$ with each value decremented by one. Therefore, if $r = 3$ then

$$R = 1000110010010001, \qquad A[1..6] = [16, 13, 1, 7, 10, 4]$$

(see beginning of Section 4 for the definition of R and A).

Comparing R and $B_1 = 0001000000000111$ we see that the sampled characters $\mathsf{BWT}(S_1)[1, 5, 6, 9, 12]$ that are in C, are C's 1st, 4th, 5th, 8th and 11th characters. From $B_2 = 010000000001010$ we see that the 1st, 4th, 5th, 8th and 11th characters in C in $\mathsf{BWT}(S_2)$ are $\mathsf{BWT}(S_2)[1, 5, 6, 9, 13]$, which are mapped to their positions by the BWT from $S_2[15, 12, 1, 3, 14]$.

The relative order $5, 3, 1, 2, 4$ of the positions $15, 12, 1, 3, 14$ in S_2 of these characters, is *almost* the same as the relative order $5, 4, 1, 2, 3$ of the positions $16, 13, 1, 7, 10$ in S_1 of the sampled characters in $\mathsf{BWT}(S_1)$ that are in C.

We can get rid of the "almost" if instead of C we consider a subsequence G' derived from a BWT-invariant sequence G. For example, we can choose

$$\begin{aligned} G' &= \mathsf{TCTCGTAAAGG} \\ B'_1 &= 0001000001010101 & B'_2 &= 010000010001010 \\ D'_1 &= \mathsf{GAGTC} & D'_2 &= \mathsf{GACC} \end{aligned}$$

Clearly G' is not an LCS of $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ and, thus, our data structures for supporting rank in $\mathsf{BWT}(S_2)$ are slightly larger. However, now the characters in $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ that are in G', are mapped to their positions by the BWT from

$$S_1[16, 2, 6, 13, 1, 12, 3, 7, 14, 15, 11], \qquad S_2[15, 2, 5, 12, 1, 11, 3, 6, 13, 14, 10]$$

and the relative order $11, 2, 4, 8, 1, 7, 3, 5, 9, 10, 6$ of the indices in those two lists is *exactly* the same. Now suppose we store yet another pair of bitvectors

$$M_1 = 0001100111000000, \qquad M_2 = 000100111000000$$

with 1s marking the positions in S_1 and S_2 of characters that are not mapped into G' in $\mathsf{BWT}(S_1)$ and $\mathsf{BWT}(S_2)$ (that is, the characters that are not in the BWT-invariant subsequence G). We claim that if we can support fast rank queries on B'_2, R and M_1, fast access to A and fast select_0 queries on B'_1 and M_2, then we can support fast access to a (possibly irregular) SA sample for S_2 with as many sampled characters as there are in G' in $\mathsf{BWT}(S_1)$. More specifically, if $\mathsf{BWT}(S_2)[i]$ is in G' and $R[B'_1.\mathsf{select}_0(B'_2.\mathsf{rank}_0(i))] = 1$ — meaning the corresponding character in G' in $\mathsf{BWT}(S_1)$ is sampled — then $\mathsf{BWT}(S_2)[i]$ is mapped to its position by the BWT from

$$S_2\left[M_2.\mathsf{select}_0\left(M_1.\mathsf{rank}_0\left(A\left[R.\mathsf{rank}_1\left(B'_1.\mathsf{select}_0\left(B'_2.\mathsf{rank}_0(i)\right)\right)\right]\right)\right)\right].$$

We leave a detailed explanation to the full version of this paper. We note, however, that this approach works for any sample rate r, and even if the SA sample for S_1 is irregular itself.

In our example, since $\mathsf{BWT}(S_2)[10]$ is in G', $B'_1.\mathsf{select}_0(B'_2.\mathsf{rank}_0(10)) = 9$ and $R[9] = 1$, we know $\mathsf{BWT}(S_2)[10]$ is mapped to its position by the BWT from position $M_2.\mathsf{select}_0\left(M_1.\mathsf{rank}_0\left(A[R.\mathsf{rank}_1(9)]\right)\right) = 6$ in S_2.