BIOINFORMATICS APPLICATIONS NOTE Vol. 30 no. 22 2014, pages 3274–3275
doi:10.1093/bioinformatics/btu541

Sequence analysis Advance Access publication August 8, 2014

Fast construction of FM-index for long sequence reads

Heng Li
Medical Population Genetics Program, Broad Institute, 75 Ames Street, Cambridge, MA 02142, USA
Associate Editor: Michael Brudno

ABSTRACT

Summary: We present a new method to incrementally construct the FM-index for both short and long sequence reads, up to the size of a genome. It is the first algorithm that can build the index while implicitly sorting the sequences in the reverse (complement) lexicographical order without a separate sorting step. The implementation is among the fastest for indexing short reads and the only one that practically works for reads of averaged kilobases in length.

Availability and implementation: https://github.com/lh3/ropebwt2

Contact: hengli@broadinstitute.org

Received on June 2, 2014; revised on July 15, 2014; accepted on August 4, 2014

1 INTRODUCTION

FM-index plays an important role in DNA sequence alignment, *de novo* assembly (Simpson and Durbin, 2012) and compression (Cox *et al.*, 2012). Fast and lightweight construction of FM-index for a large dataset is the key to these applications. In this context, a few algorithms (Bauer *et al.*, 2013; Liu *et al.*, 2014) have been developed that substantially outperform earlier algorithms. However, they are only efficient for short reads. A fast and practical algorithm for long sequence reads is still lacking. This work aims to fill this gap.

2 METHODS

Let $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}, \mathtt{N}\}$ be the *alphabet* of DNA with a lexicographical order $\mathtt{A} < \mathtt{C} < \mathtt{G} < \mathtt{T} < \mathtt{N}$. Each element in Σ is called a *symbol* and a sequence of symbols called a *string* over Σ. Given a string P, $|P|$ is its length and $P[i]$ the symbol at position i. A sentinel \$ is smaller than all the other symbols. For simplicity, we let $P[-1] = P[|P|] = \$$. We also introduce $\tilde{P}$ as the reverse of P and $\overline{P}$ as the reverse complement of P.

Given a list of strings over Σ, $(P_i)_{0 \le i < m}$, let $T = P_0\$_0 \ldots P_{m-1}\$_{m-1}$ with $\$_0 < \cdots < \$_{m-1} < \mathtt{A} < \mathtt{C} < \mathtt{G} < \mathtt{T} < \mathtt{N}$. The *suffix array* of T is an integer array S such that $S(i)$, $0 \le i < |T|$, is the starting position of the i-th smallest suffix in the collection T. The *Burrows-Wheeler Transform*, or *BWT*, of T can be computed as $B[i] = T[S(i) - 1]$. For the description of the algorithm, we segment B into $B = B_\$ B_\mathtt{A} B_\mathtt{C} B_\mathtt{G} B_\mathtt{T} B_\mathtt{N}$, where $B_a[i] = B[i + C(a)]$ with $C(a) = |\{j : T[j] < a\}|$ being the array of accumulative counts. By the definition of suffix array and BWT, B_a consists of all the symbols with their next symbol in T being a.

The above defines BWT for an ordered list of strings. We next seek to define BWT for an unordered set of strings $\mathcal{C}$ by imposing an arbitrary sorting order on $\mathcal{C}$. We say list $(P_i)_i$ is in the *reverse lexicographical order* or *RLO*, if $\tilde{P}_i \le \tilde{P}_j$ for any $i < j$; say it is in the *reverse-complement lexicographical order* or *RCLO*, if $\overline{P}_i \le \overline{P}_j$ for any $i < j$. The *RLO-BWT* of $\mathcal{C}$, denoted by $B^{\mathrm{RLO}}(\mathcal{C})$, is constructed by sorting strings in $\mathcal{C}$ in RLO and then applying the procedure in the previous paragraph on the sorted list. *RCLO-BWT* $B^{\mathrm{RCLO}}(\mathcal{C})$ can be constructed in a similar way. In $B^{\mathrm{RCLO}}(\{P_i\}_i \cup \{\overline{P}_j\}_j)$, the k-th smallest sequence is the reverse complement of the k-th sequence in the FM-index. This property removes the necessity of keeping an extra array to link the rank and the position of a sequence in the FM-index, and thus helps to reduce the memory of some FM-index–based algorithms (Simpson and Durbin, 2012). For short reads, RLO/RCLO-BWT is also more compressible (Cox *et al.*, 2012).

As a preparation, we further define two string operations: $\mathrm{rank}(c, k; B)$ and $\mathrm{insert}(c, k; B)$, where $\mathrm{rank}(c, k; B) = |\{i < k : B[i] = c\}|$ gives the number of symbols c before the position k in B, and $\mathrm{insert}(c, k; B)$ inserts symbol c after k symbols in B with all the symbols after position k shifted to make room for c. We implemented the two operations by representing each B_c in a B+-tree in memory, where a leaf keeps a run-length encoded string and an internal node keeps the count of each symbol in the leaves descended from the node.

Algorithm 1 appends a string to an existing index by inserting each of its symbol from the end of P. It was first described by Chan *et al.* (2004). Algorithm 2 constructs RLO/RCLO-BWT in a similar manner to Algorithm 1 except that it inserts $P[i]$ to $[l, u)$, the suffix array interval of P's suffix starting at $i + 1$, and that BWT symbols in this interval are already sorted. This process implicitly applies a radix sort from the end of P, sorting it into the existing strings in the BWT in RLO/RCLO. Note that if we change line 1 to "$l \leftarrow u \leftarrow |\{i : B[i] = \$\}|$", Algorithm 2 will be turned into Algorithm 1. Recall that the BCR algorithm (Bauer *et al.*, 2013) is, to some extent, the multi-string version of Algorithm 1. Following similar reasoning, we can extend Algorithm 2 so as to insert multiple strings at the same time, which gives Algorithm 3. We use an array $A(j)$ to keep the state of the j-th sequence after inserting its d-long suffix. At line 2, $A(j).c$ is the previously inserted symbol and $[A(j).l, A(j).u)$ is the interval to which the new symbol is inserted. In implementation, we may speed up the sorting mode by inserting multiple symbols at line 3.

When B is represented by a balanced tree structure, the time complexity of all three algorithms is $O(n \log n)$, where n is the total number of symbols in the input. However, we will see later that for short strings, Algorithm 3 is substantially faster than the first two algorithms, due to the locality of memory accesses, the possibility of cached B+-tree update and the parallelization of the 'for' loop at line 1. These techniques are more effective for a larger batch of shorter strings.

Disregarding RLO/RCLO, Algorithm 3 is similar to BCR except that BCR keeps B in monolithic arrays. As a result, the time complexity of BCR is $O(nl)$, where l is the maximum length of reads, not scaling well to l.

Algorithm 1: Append one string

```
Input: A string $P$ and an existing BWT $B$ for $T$
Output: BWT for $TP\$$

Function InsertIO1$(B, P)$ begin
    $c \leftarrow \$; k \leftarrow |\{i : B[i] = \$\}|$
    for $i \leftarrow |P| - 1$ to $-1$ do
        $\mathrm{insert}(P[i], k; B_c)$
        $k \leftarrow \mathrm{rank}(P[i], k; B_c) + \sum_{a<c} |\{j : B_a[j] = P[i]\}|$
        $c \leftarrow P[i]$
    return $B$
```

Algorithm 2: Insert one string to RLO/RCLO-BWT

```
**Input**: $B^{\mathrm{RLO}}(\mathcal{C})$ (or $B^{\mathrm{RCLO}}(\mathcal{C})$) and a string $P$
**Output**: $B^{\mathrm{RLO}}(\mathcal{C} \cup \{P\})$ (or $B^{\mathrm{RCLO}}(\mathcal{C} \cup \{P\})$)

**Function** INSERTRLO1($B, P, is\_comp$) **begin**
1       $[l, u) \leftarrow [0, |\{i : B[i] = \$\}|)$
        **for** $i \leftarrow |P| - 1$ **to** $-1$ **do**
            $[l, u) \leftarrow$ INSERTAUX($B, P[i], l, u, P[i+1], is\_comp$)
        **return** $B$
**Function** INSERTAUX($B, c', l, u, c, is\_comp$) **begin**
        $k \leftarrow l$
        **if** $is\_comp$ *is* **true and** $c' \neq$ "N" **then**
            **for** $a = \$$ **or** $c' < a <$ "N" **do**
                $k \leftarrow k + [\mathrm{rank}(a, u; B_c) - \mathrm{rank}(a, l; B_c)]$
        **else**
            **for** $\$ \leq a < c'$ **do**
                $k \leftarrow k + [\mathrm{rank}(a, u; B_c) - \mathrm{rank}(a, l; B_c)]$
        $l' \leftarrow \mathrm{rank}(c', l; B_c)$; $u' \leftarrow \mathrm{rank}(c', u; B_c)$
        $\mathrm{insert}(c', k; B_c)$
        $m \leftarrow \sum_{a<c} |\{j : B_a[j] = c'\}|$
        **return** $[l' + m, u' + m)$
```

Algorithm 3: Insert multiple strings

```
**Input**: Existing BWT $B$ and a list of strings $\{P_k\}_k$
**Output**: Updated BWT $B$ with strings inserted in the specified order

**Function** INSERTMULTI($B, \{P_k\}_k, is\_sorted, is\_comp$) **begin**
        **for** $0 \leq j < |\{P_k\}_k|$ **do**
            $A(j).c \leftarrow \$$; $A(j).i \leftarrow j$
            **if** $is\_sorted$ *is* **true then**
                $[A(j).l, A(j).u) \leftarrow [0, |\{i : B[i] = \$\}|)$
            **else**
                $A(j).l \leftarrow A(j).u \leftarrow |\{i : B[i] = \$\}| + j$
        $d \leftarrow 0$
        **while** $|A| \neq 0$ **do**
1           *Stable sort array $A$ by $A(\cdot).c$*
2           **for** $0 \leq j < |A|$ **do**
                $c \leftarrow A(j).c$; $A(j).c \leftarrow P_{A(j).i}[|P_{A(j).i}| - 1 - d]$
                $[A(j).l, A(j).u)$
3                 $\leftarrow$ INSERTAUX($B, A(j).c, A(j).l, A(j).u, c, is\_comp$)
            *Remove $A(j)$ if $A(j).c = \$$*
            $d \leftarrow d + 1$
        **return** $B$
```

3 RESULTS AND DISCUSSION

We implemented the algorithm in ropeBWT2 and evaluated its performance together with BEETL (http://bit.ly/beetlGH), the original on-disk implementation of BCR and BCRext, ropeBWT-BCR (https://github.com/lh3/ropebwt), an in-memory reimplementation of BCR by us, and NVBio (http://bit.ly/nvbioio), a GPU-based algorithm inspired by CX1 (Liu *et al.*, 2014). Table 1 shows that for ~100 bp reads, ropeBWT2 has comparable performance to others. For the ~875 bp Venter dataset, NVBio aborted due to insufficient memory under various settings. We did not apply BCR because it is not designed for long reads of unequal lengths. Only ropeBWT2 works with this data set and the even longer moleculo reads.

Table 1. Performance of BWT construction

Data[a]	Algorithm	RCLO	Real	CPU%	RAM[b] (GB)	Comments
worm	nvbio	–	316 s	138	12.9	See note[c]
worm	ropebwt-bcr	–	480 s	223	2.2	-btORf
worm	Algorithm 3	Yes	506 s	250	10.5	-brRm10g
worm	Algorithm 3	No	647 s	249	11.8	-bRm10g
worm	beetl-bcr	–	965 s	259	1.8	RAM disk[d]
worm	beetl-bcr	–	2092 s	122	1.8	Network[e]
worm	Algorithm 1	–	5125 s	100	2.5	-bRm0
worm	beetl-bcrext	–	5900 s	48	0.1	Network[e]
12 878	ropebwt-bcr	–	3.3 h	210	39.3	-btORf
12 878	nvbio	–	4.1 h	471	63.8	See note[f]
12 878	Algorithm 3	Yes	5.0 h	261	34.0	-brRm10g
12 878	Algorithm 3	No	5.1 h	248	60.9	-bRm10g
12 878	beetl-bcr	–	11.2 h	131	31.6	Network[e]
Venter	Algorithm 3	Yes	1.4 h	274	22.2	-brRm10g
Venter	Algorithm 3	No	1.5 h	274	22.8	-bRm10g
mol	Algorithm 3	No	6.8 h	285	20.0	-bRm10g

[a]Datasets—*worm*: 66M × 100 bp *Caenorhabditis elegans* reads from SRR065390; *12878*: 1206M × 101 bp human reads for sample NA12878 (Depristo *et al.*, 2011). *Venter*: 32M × 875 bp (in average) human reads by Sanger sequencing (Levy *et al.* 2007; http://bit.ly/levy2007); *mol*: 23M × 4026 bp (in average) human reads by Illumina's Moleculo sequencing (http://bit.ly/mol12878).
[b]Hardware—CPU: 48 cores of Xeon E5-2697v2 at 2.70 GHz; GPU: one Nvidia Tesla K40; RAM: 128 GB; Storage: Isilon IQ 72000x and X400 over network. CPU time, wall-clock time and peak memory are measured by GNU time.
[c]Run with option '-R -cpu-mem 4096 -gpu-mem 4096'. NVBio uses more CPU and GPU RAM than the specified.
[d]Results and temporary files created on in-RAM virtual disk '/dev/shm'.
[e]Results and temporary files created on Isilon's network file system.
[f]Run with option '-R -cpu-mem 48000 -gpu-mem 4096'.

Funding: NHGRI U54HG003037; NIH GM100233.

Conflict of Interest: none declared.

REFERENCES

Bauer,M.J. *et al.* (2013) Lightweight algorithms for constructing and inverting the BWT of string collections. *Theor. Comput. Sci.*, **483**, 134–148.

Chan,H.-L. *et al.* (2004) Compressed index for a dynamic collection of texts. In: Sahinalp,S.C. Muthukrishnan,S. and Dogrusöz,U. (eds) *CPM, Volume 3109 of Lecture Notes in Computer Science*. Springer, Berlin Heidelberg, pp. 445–456.

Cox,A.J. *et al.* (2012) Large-scale compression of genomic sequence databases with the burrows-wheeler transform. *Bioinformatics*, **28**, 1415–1419.

Depristo,M.A. *et al.* (2011) A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat. Genet.*, **43**, 491–498.

Levy,S. *et al.* (2007) The diploid genome sequence of an individual human. *PLoS Biol.*, **5**, e254.

Liu,C.-M. *et al.* (2014) GPU-accelerated BWT construction for large collection of short reads. arXiv:1401.7457.

Simpson,J.T. and Durbin,R. (2012) Efficient de novo assembly of large genomes using compressed data structures. *Genome Res.*, **22**, 549–556.