Pan-genome de Bruijn Graph using the Bidirectional FM-index: Supplementary Material

Lore Depuydt[1*], Luca Renders[1], Thomas Abeel[2,3] and Jan Fostier[1*]

[*]Correspondence:
Lore.Depuydt@UGent.be;
Jan.Fostier@UGent.be
[1]Department of Information Technology - IDLab, Ghent University - imec, Technologiepark 126, B-9052 Ghent (Zwijnaarde), Belgium
Full list of author information is available at the end of the article

1 Supplementary Methods

1.1 Bidirectional FM-index

In this section, we provide a brief overview of the bidirectional FM-index, which forms the base of our pan-genome graph representation.

The Burrows-Wheeler transform $\mathrm{BWT}[0..n[$ of a text T of length n is defined as $\mathrm{BWT}[i] = T[\mathrm{SA}[i]-1]$ if $\mathrm{SA}[i] > 0$ and $\mathrm{BWT}[i] = \$$ otherwise [1]. Here, SA denotes the suffix array, an array over integer values that indicate the starting positions of the suffixes of T in lexicographic order [2]. We build the suffix array using `divsufsort` (coded by Yuta Mori). There is an important relationship between the characters in the sorted permutation of T (called F) and the BWT of T (called L), namely the LF property. The LF property states that the ith occurrence of a particular character c in the BWT and the ith occurrence of c in F correspond to the same character in T. We thus need support for $\mathrm{Occ}(c, i)$ queries on the BWT that return the number of occurrences of character c in the prefix $\mathrm{BWT}[0..i[$. We realize this using $|\Sigma|$ bit vectors with constant-time rank support (`rank9` algorithm [3]). The LF property can then be computed in constant time as follows: $\mathrm{LF}[i] = \mathrm{C}(c)+\mathrm{Occ}(c, i)$, with $c = \mathrm{BWT}[i]$. Here, $\mathrm{C}(c)$ denotes the number of characters in T strictly smaller than c. These values are pre-computed and stored in a small array of size $|\Sigma|$. Collectively, the BWT, SA, bit vectors and C array are referred to as the (unidirectional) FM-index [4]. Often, only the suffix array entries for every s_{SA}th suffix are stored, s_{SA} being the suffix array sparseness factor. This sparse representation of the suffix array requires an auxiliary bit vector with rank support to indicate the presence or absence of index positions and to compute their offsets within the sparse suffix array. Every entry of the suffix array can be computed in $\mathrm{O}(s_{\mathrm{SA}})$ time using the LF property. This is one of the various design choices through the which time-space tradeoff can be controlled.

Exact pattern matching using the FM-index is performed by matching character by character, from right to left. Let $[b, e[$ denote the interval over the suffix array for which the corresponding suffixes have P as a prefix. The suffix array interval $[b', e'[=$ `extendBackward`$([b, e[, c)$ whose suffixes have cP as a prefix can then be computed by $b' = \mathrm{C}(c) + \mathrm{Occ}(c, b)$ and $e' = \mathrm{C}(c) + \mathrm{Occ}(c, e)$. Because $\mathrm{Occ}(c, i)$ queries can be performed in constant time, exact matching of a pattern P of size m takes $\mathrm{O}(m)$ time. The size of the obtained interval $[b, e[$ denotes the number of occurrences of P in T. The positions of the occurrences in T can be obtained using the suffix array.

A bidirectional FM-index is obtained by also storing BWT^r, the Burrows-Wheeler transform of T^r, the reverse of T. By keeping track of both the range $[b, e[$ over BWT as well as the range $[b^r, e^r[$ over BWT^r in a synchronized manner, one can extend a pattern P to either cP (`extendBackward`) or Pc (`extendForward`) in $\mathrm{O}(|\Sigma|)$ time [5].

Table S1 Overview of all components of our bidirectional FM-index, with their respective memory usage. For each component, we clarify the number of entries it contains, and the number of bits needed to store each entry. The number of entries and memory usage of each component is illustrated for the pan-genome of 10 human genomes ($s_{\mathrm{SA}} = 16$).

Component	**Memory usage per entry [bits]**	**Number of entries**		**Total for 10 human genomes**
		General	**10 human genomes**	
Compacted text	3	n	$\sim$ 30 billion bp	10.60 GiB
Counts array C	64	256	256 characters	2 KiB
BWT	3	n	$\sim$ 30 billion bp	10.60 GiB
PrefixOcc	6.25	n	$\sim$ 30 billion bp	22.08 GiB
$\mathrm{PrefixOcc}^r$	6.25	n	$\sim$ 30 billion bp	22.08 GiB
Sparse SA	64	n/s_{SA}	$\sim$ 1.9 billion entries	14.13 GiB
SA bit vector	1.25	n	$\sim$ 30 billion bp	4.42 GiB
Total				83.89 GiB

By replacing the Occ data structure with a PrefixOcc data structure, this bound is improved to O(1) [6]. Note that both ranges always have the same width, which monotonically decreases when new characters are added to the occurrence (adding a character cannot lead to more occurrences in the reference text).

Table S1 details the components of the bidirectional FM-index that is used in our implementation. The compacted reference text uses 3 bits to distinguish 6 characters. Note that the original reference text T (8 bits per character), is necessary to build the representation. Once this process is finished, T can be removed from disk to spare memory. The counts array stores counts for each character in the extended ASCII alphabet, which is why 256 entries are stored. In practice, only 6 of these entries can have non-zero values (‘A’, ‘C’, ‘G’, ‘T’, ‘%’ and ‘\$’). BWT uses 3 bits per character, analogous to the compacted reference text. PrefixOcc, resp. $\mathrm{PrefixOcc}^r$, stores the locations of characters ‘A’, ‘C’, ‘G’, ‘T’, ‘%’ in BWT, resp. BWT^r, in 5 bit vectors (accounting for 5 bits per character). Additionally, each of these bit vectors must support constant-time rank operations, which leads to 0.25 bits overhead per character per bit vector (`rank9` algorithm [3]). This totals to 6.25 bits per character for tables PrefixOcc and $\mathrm{PrefixOcc}^r$. Note that we only need $\mathrm{PrefixOcc}^r$, not BWT^r itself. To guarantee $\mathrm{O}(s_{\mathrm{SA}})$ suffix array indexing, we store each s_{SA}th suffix in the sparse SA. To check whether a certain index in the SA corresponds to a s_{SA}th suffix or not, we need an additional bit vector indicating the stored entries. This bit vector also needs 0.25 additional bits per entry for constant-time rank support.

Finally, the longest common prefix array or the LCP array is also used for the construction of our pan-genome graph. The LCP array of string T, denoted by LCP, is an array of size $n + 1$ such that $\mathrm{LCP}[0] = -1$, $\mathrm{LCP}[n] = -1$ and $\mathrm{LCP}[i] = \mathrm{lcp}(T_{\mathrm{SA}[i-1]}, T_{\mathrm{SA}[i]})$ for $0 < i < n$, where $\mathrm{lcp}(u, v)$ denotes the length of the longest common prefix between two strings u and v [7].

2 Supplementary Results

2.1 The Effect of the Suffix Array Sparseness Factor on Memory Usage and APM Performance

Adjusting the suffix array sparseness factor can lead to significantly less memory usage, as the complete suffix array is by far the largest component of the data

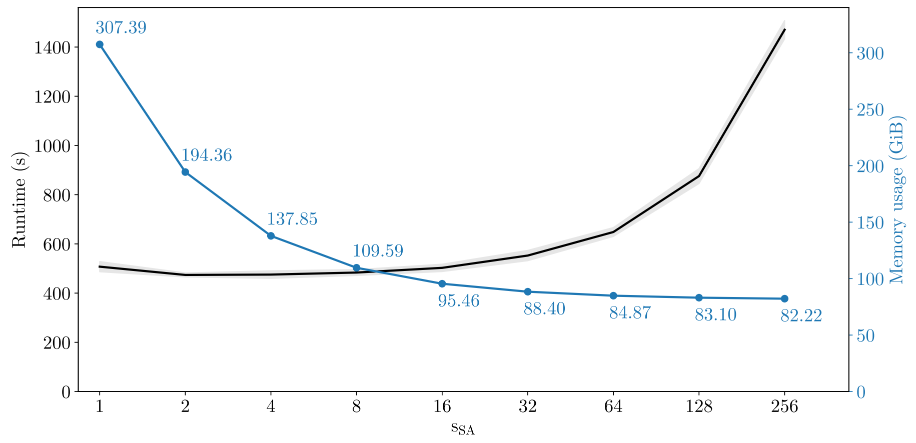

Figure S1 Black: average runtime over 10 experiments for mapping 100 000 Illumina reads of length 101 bp as well as their reverse complement to the pan-genome graph of 10 human genomes, as a function of the suffix array sparseness factor s_{SA} (1 to 256). We perform approximate pattern matching with a maximum allowed number of errors of $K = 4$, using the search scheme proposed by Kucherov et al.. The pan-genome is built for $k = 25$ and $s_{cp} = 128$. The 95% confidence interval for the runtime is also indicated. Blue: the memory usage of the complete representation of the ccdBG for the pan-genome of 10 human genomes (including the underlying bidirectional FM-index), as a function of the suffix array sparseness factor s_{SA}.

structure. Fig. S1 illustrates the time-space tradeoff that arises when s_{SA} is altered. It shows the runtime of the APM procedure for matching 100 000 Illumina reads to the pan-genome graph of 10 human genomes as well as its memory usage, as a function of s_{SA}.

For $s_{\mathrm{SA}} = 1$, the suffix array is stored at full capacity and the total memory usage is 307.39 GiB. When s_{SA} is increased, the memory usage of SA decreases as less of its entries are explicitly stored. For an infinitely large value of s_{SA}, the data structure comprises 81.33 GiB as the memory usage of SA approaches 0 GiB. The performance of the APM procedure decreases as s_{SA} increases, since the SA entries that are not explicitly stored must be recalculated on the fly. We see that the runtime increases exponentially with s_{SA}. Note however that for very small values of s_{SA} (1, 2, 4, 8), the APM performance is barely affected. In fact, the APM performance is better at $s_{\mathrm{SA}} = 2$ than at $s_{\mathrm{SA}} = 1$, presumably due to the high RAM requirements.

Determining the right value for s_{SA} is different in every scenario: large pan-genomes require a higher suffix array sparseness factor, whilst smaller datasets benefit from a more complete suffix array. Still, choosing $s_{\mathrm{SA}} = 16$ or $s_{\mathrm{SA}} = 32$ generally leads to a good balance.

2.2 Cytoscape Visualization

In Fig. 5 in the manuscript, we illustrate the visualization of a subgraph of the pan-genome of 341 *M. tuberculosis* strains in a curated form:

- the first $k - 1$ overlapping characters were omitted from each node;
- node identifiers were replaced by a smaller set of character identifiers;
- irrelevant nodes were purged from the graph;
- the multiplicity of the edges was added to the figure explicitly.

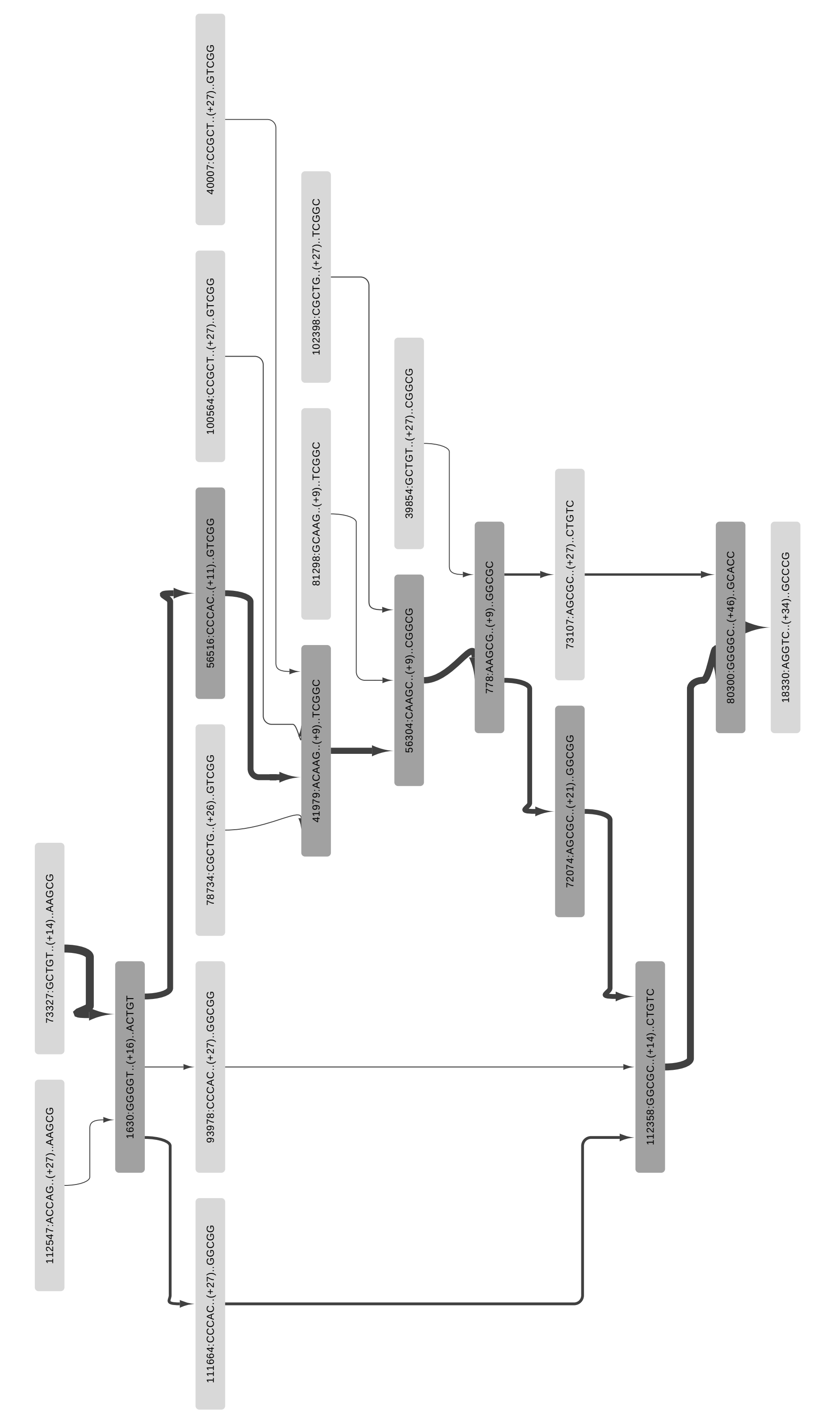
40007:CCGCT..(+27)..GTCGG
100564:CCGCT..(+27)..GTCGG
56516:CCCAC..(+11)..GTCGG
78734:CGCTG..(+26)..GTCGG
93978:CCCAC..(+27)..GGCGG
111664:CCCAC..(+27)..GGCGG
102398:CGCTG..(+27)..TCGGC
81298:GCAAG..(+9)..TCGGC
41979:ACAAG..(+9)..TCGGC
39854:GCTGT..(+27)..CGGCG
56304:CAAGC..(+9)..CGGCG
778:AAGCG..(+9)..GGCGC
73107:AGCGC..(+27)..CTGTC
72074:AGCGC..(+21)..GGCGG
112358:GGCGC..(+14)..CTGTC
80300:GGGGC..(+46)..GCACC
18330:AGGTC..(+34)..GCCCG
73327:GCTGT..(+14)..AAGCG
112547:ACCAG..(+27)..AAGCG
1630:GGGGT..(+16)..ACTGT

Figure S2 Cytoscape visualization of a subgraph of the pan-genome ccdBG of 341 *M. tuberculosis* strains ($k = 19$), corresponding to the end of the RRDR region of gene *rpoB*. Dark nodes represent the node path corresponding to the sequence for which visualization was requested.

In this section, we include the original subgraph as it is visualized by Cytoscape in Figure S2, which was created using the following command:

```
$ /nexus/build/visualizeRead -e 0 -d 1 -b -o outputFile
    ↪ MTuberculosisPanGenome 19
    ↪ ACCCACAAGCGCCGACTGTCGGCGCTGGGGCCCGGCGGTCTGTCA
```

The inputted DNA sequence corresponds to the last three codons from the RRDR region (“TCGGCGCTG”), padded with 18 nucleotides from the reference strain on either side, in order to visualize all nodes that contain a part of these three codons. The dark gray nodes in Fig. S2 correspond to the reference path in the manuscript’s Fig. 5 (i.e., node path ADEFGHIK). Furthermore, nodes 111664, 93978 and 73107 in Fig. S2 correspond to respectively nodes B, C and J in Fig. 5 (manuscript), representing RRDR mutations S450L, S450W and L452P.

3 Reproducing Results

Following software versions were used for benchmarking:

- Nexus v1.1.0:
  `https://github.com/biointec/nexus/releases/tag/v1.1.0`
- Beller and Ohlebusch’s A4 [7]:
  `https://www.uni-ulm.de/in/theo/research/seqana.html`
- deBGA [8] commit c2dbf6d6bb9bb27a0230b2d4022ec3e05efa46c9:
  `https://github.com/HongzheGuo/deBGA/tree/c2dbf6d`
- Pufferfish [9] (and PuffAligner [10]) v1.8.0:
  `https://github.com/COMBINE-lab/pufferfish/releases/tag/salmon-v1.8.0`

3.1 Obtaining the Reads

We sampled 100 000 reads from an Illumina experiment dataset (`ftp://ftp.sra.ebi.ac.uk/vol1/fastq/ERR194/ERR194147/ERR194147_1.fastq.gz`), which only contain ‘A‘, ‘C‘, ‘G‘ and ‘T‘ characters. The sampled dataset we used for the results in the paper is available on our GitHub page: `https://github.com/biointec/nexus/releases/download/v1.0.0/sampled_illumina_reads.fastq`. From now on, we refer to them as `reads.fastq`.

3.2 Commands for Building the Indexes (cf. Table 9)

deBGA construction:

```
$ ./deBGA index -k 25 10HumanGenomes.fa deBGA/10HumanGenomes
```

Pufferfish construction:

```
$ ./pufferfish index -r 10HumanGenomes.fa -o pufferfish/ --tmpdir
    ↪ pufferfishtemp/ -k 25 --noClip --filt-size 40 --threads 1
```

A4 construction:

```
$ ./a4.x construct --inputfile=10HumanGenomes.fa --outputfile=a4
    ↪ /10HumanGenomes --kfile=kfile.txt
```

Where kfile.txt contains the value 25.
Nexus construction:

```
$ ./nexusBuild -s 16 -c 128 -p 10HumanGenomes 25
```

Where input file 10HumanGenomes.txt contains a preprocessed version of corresponding 10HumanGenomes.fa. From each strain we removed all N's and substituted them with a random nucleotide. Then we removed the first line, concatenated the chromosomes, removed newline characters, and added a separation character to the end of file for each strain. Next, the strains can be concatenated into one pan-genome. An example of preprocessing a pan-genome of 3 strains is as follows:

```
$ tail -n +2 genome1.fasta.non | sed "s/>.*//g" | cat - percent |
    ↪ tr -d '\n' > genome1.txt
$ tail -n +2 genome2.fasta.non | sed "s/>.*//g" | cat - percent |
    ↪ tr -d '\n' > genome2.txt
$ tail -n +2 genome3.fasta.non | sed "s/>.*//g" | cat - dollar |
    ↪ tr -d '\n' > genome3.txt
$ cat genome1.txt genome2.txt genome3.txt > pangenome.txt
```

Where genome1.fasta.non, genome2.fasta.non and genome3.fasta.non are the result of the substitution of the N's for the input strains and pangenome.txt is the reference pan-genome used to build the index. genome1.txt, genome2.txt and genome3.txt are intermediate files which can be removed once the pan-genome is obtained. dollar and percent are text files containing a single character '\$' and '%', respectively. These files should be present in your directory before executing the above commands.

Future work contains implementing this preprocessing into Nexus, such that the end user can simply input the .fasta/.fa file.

3.3 Command for Building All Nexus Indexes Used for Benchmarking

```
$ ./nexusBuild -s 1 -s 2 -s 4 -s 8 -s 16 -s 32 -s 64 -s 128 -s 256
    ↪  -c 8 -c 16 -c 32 -c 64 -c 128 -c 256 -c 512 -c 1024 -c 2048
    ↪  -c none -p 10HumanGenomes 25,50,75
```

3.4 Commands for Reproducing Table 10

For A4:

```
$ ./a4.x find_pattern --graphfile=10HumanGenomes.k25.bin --
    ↪ patternfile=reads.txt
```

Where `reads.txt` is a preprocessed text file containing only the DNA sequences contained in `reads.fastq`, no other information. Additionally, `reads.txt` also contains the reverse complement of all reads, as A4 does not match the reverse complements automatically.
For Nexus:

```
$ ./nexus -e <ED> -s 16 -c 128 -ss custom ../search_schemes/kuch_k
    ↪ +1_adapted/ 10HumanGenomes 25 reads.fastq
```

With `ED` ∈ {`0,1,2,3,4`}.

3.5 Commands for Reproducing Table 11

For PuffAligner:

```
$ ./pufferfish align -i pufferfish/ --read reads.fastq -t 1 -o
    ↪ output.sam --verbose --genomicReads
```

Where the `pufferfish/` directory contains the index.
For A4 and Nexus: see commands above.

3.6 Commands for Reproducing Figure 4

```
$ ./nexus -e 4 -s 16 -c <scp> -ss custom ../search_schemes/kuch_k
    ↪ +1_adapted/ 10HumanGenomes <k> reads.fastq
```

With `scp` ∈ {`8, 16, 32, 64, 128, 256, 512, 1024, 2048, none`} and `k` ∈ {`25,50,75`}.

3.7 Commands for Reproducing Figure S1

```
$ ./nexus -e 4 -s <sSA> -c 128 -ss custom ../search_schemes/kuch_k
    ↪ +1_adapted/ 10HumanGenomes 25 reads.fastq
```

With `sSA` ∈ {`1, 2, 4, 8, 16, 32, 64, 128, 256`}.

3.8 Case Study

3.8.1 Building the index

The preprocessed text file `MTuberculosisPanGenome.txt` containing the 341 concatenated *M. tuberculosis* strains can be found on our GitHub page: `https://github.com/biointec/nexus/releases/download/v1.1.0/MTuberculosisPanGenome.txt`.

Additionally, an annotation file is provide that contains the identifiers of these strains in the order in which they appear in the concatenated reference file: `https://github.com/biointec/nexus/releases/download/v1.1.0/MTuberculosisPanGenome.annotation.txt`.

Using the reference text file, the index can be built as follows:

```
$ ./nexusBuild -s 16 -c 128 -p MTuberculosisPanGenome 19
```

3.8.2 Reproducing the Case Study

To reproduce the case study, switch to the corresponding branch on GitHub: https://github.com/biointec/nexus/tree/CaseStudy/casestudy. Compile the code. Run the following command **inside the folder containing the index**.

```
$ ./casestudy
```

This executable searches for compensatory mutations as is described in the manuscript. Comments are provided in the `casestudy.cpp` script to outline the process. The case study results in a file `MTuberculosisPanGenome_Compensatory.tsv`, which contains the 14 candidate putative compensatory mutations that were also discussed in Table 13 in the manuscript. This file contains 5 fields:

- The RRDR mutation to which the candidate putative compensatory mutation corresponds;
- The identifier of the node in which the candidate putative compensatory mutation is found;
- The number of strains that carry the candidate putative compensatory mutation;
- The position of the candidate putative compensatory mutation with respect to the reference strain;
- The length of the node containing the candidate putative compensatory mutation.

Note again that the pipeline implemented here is more of an ad hoc solution to the problem of finding candidate compensatory mutations corresponding to mutations in the RRDR region of *rpoB*, rather than a general pipeline to be readily applied to other problems. For this reason, we provide the functionality in a `.cpp` file. If there were any interest to extend these ideas into a more general pipeline, possibly written in a more accessible script, do not hesitate to contact the authors of the manuscript.

Author details
[1]Department of Information Technology - IDLab, Ghent University - imec, Technologiepark 126, B-9052 Ghent (Zwijnaarde), Belgium. [2]Delft Bioinformatics Lab, Delft University of Technology, 2628 XE Delft, Netherlands. [3]Infectious Disease and Microbiome Program, Broad Institute of MIT and Harvard, MA 02142 Cambridge, USA.

References

1. Burrows, M., Wheeler, D.: A Block-Sorting Lossless Data Compression Algorithm. Research Report 124, Digital Equipment Corporation Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301 (May 1994)
2. Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing **22**(5), 935–948 (1993). doi:10.1137/0222058
3. Vigna, S.: Broadword Implementation of Rank/Select Queries. In: McGeoch, C.C. (ed.) Experimental Algorithms, pp. 154–168. Springer, Berlin, Heidelberg (2008). doi:10.1007/978-3-540-68552-4_12
4. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings 41st Annual Symposium on Foundations of Computer Science, pp. 390–398 (2000). doi:10.1109/SFCS.2000.892127
5. Lam, T.W., Li, R., Tam, A., Wong, S., Wu, E., Yiu, S.M.: High Throughput Short Read Alignment via Bi-directional BWT. In: 2009 IEEE International Conference on Bioinformatics and Biomedicine, pp. 31–36 (2009). doi:10.1109/BIBM.2009.42
6. Pockrandt, C., Ehrhardt, M., Reinert, K.: EPR-Dictionaries: A Practical and Fast Data Structure for Constant Time Searches in Unidirectional and Bidirectional FM Indices. In: Sahinalp, S.C. (ed.) Research in Computational Molecular Biology, pp. 190–206. Springer, Cham (2017). doi:10.1007/978-3-319-56970-3_12
7. Beller, T., Ohlebusch, E.: A representation of a compressed de Bruijn graph for pan-genome analysis that enables search. Algorithms for Molecular Biology **11**(1), 20 (2016). doi:10.1186/s13015-016-0083-7
8. Liu, B., Guo, H., Brudno, M., Wang, Y.: deBGA: read alignment with de Bruijn graph-based seed and extension. Bioinformatics **32**(21), 3224–3232 (2016). doi:10.1093/bioinformatics/btw371

9. Almodaresi, F., Sarkar, H., Srivastava, A., Patro, R.: A space and time-efficient index for the compacted colored de Bruijn graph. Bioinformatics **34**(13), 169–177 (2018). doi:10.1093/bioinformatics/bty292
10. Almodaresi, F., Zakeri, M., Patro, R.: PuffAligner: a fast, efficient and accurate aligner based on the Pufferfish index. Bioinformatics **37**(22), 4048–4055 (2021). doi:10.1093/bioinformatics/btab408