Dynamic Alignment-free and Reference-free Read Compression

Guillaume Holley[1,2], Roland Wittler[1,2], Jens Stoye[1], and Faraz Hach[3,4,5]

[1] Genome Informatics, Faculty of Technology and Center for Biotechnology, Bielefeld University, Germany
[2] International Research Training Group 1906 "Computational Methods for the Analysis of the Diversity and Dynamics of Genomes", Bielefeld University, Germany
[3] School of Computing Science, Simon Fraser University, Burnaby, Canada
[4] Department of Urologic Sciences, University of British Columbia , Vancouver, Canada
[5] Vancouver Prostate Centre, Vancouver, Canada

Abstract. The advent of High Throughput Sequencing (HTS) technologies raises a major concern about storage and transmission of data produced by these technologies. In particular, large-scale sequencing projects generate an unprecedented volume of genomic sequences ranging from tens to several thousands of genomes per species. These collections contain highly similar and redundant sequences, also known as pan-genomes. The ideal way to represent and transfer pan-genomes is through compression. A number of HTS-specific compression tools have been developed to reduce the storage and communication costs of HTS data, yet none of them is designed to process a pan-genome. In this paper, we present DARRC, a new alignment-free and reference-free compression method. It addresses the problem of pan-genome compression by encoding the sequences of a pan-genome as a guided de Bruijn graph. The novelty of this method is its ability to incrementally update DARRC archives with new genome sequences without full decompression of the archive. DARRC can compress both single-end and paired-end read sequences of any length using all symbols of the IUPAC nucleotide code. On a large *P. aeruginosa* dataset, our method outperforms all other tested tools. It provides a 30% compression ratio improvement in single-end mode compared to the best performing state-of-the-art HTS-specific compression method in our experiments.
Availability. DARRC is available at `https://bitbucket.org/GuillaumeHolley/darrc/`

1 Introduction

Motivation. High Throughput Sequencing (HTS) technologies are constantly improving and making sequencing of genomes more affordable. The second generation of HTS technologies was introduced to the sequencing market in 2007, enabling higher throughput and drastically reducing the cost of sequencing per genome [20]. As a result, the number

of sequenced genomes is growing exponentially [19], making storage and access to these data a problem of main importance. For example, the Sequence Read Archive (SRA) public database was endangered in 2011 because of budgetary constraints [30]. In order to reduce storage and transmission costs, raw sequencing data are often compressed using general purpose compression tools such as gzip (based on Lempel-Ziv-77 [36]) or bzip (based on the Burrows-Wheeler Transform [4]). Although these classic tools compressed most of the public data, they are not optimized for HTS compression [8, 10, 13, 15, 21]. In FASTQ format, each record has three major components: (i) unique identifier, (ii) read sequence and (iii) quality scores. A large variety of HTS-specific compression tools were proposed [1, 3, 11, 12, 17, 18, 22, 25–27] to compress either FASTQ files or only the read sequences. While these tools are very efficient, they are not adapted to the context of large-scale sequencing projects that produce tens to several thousands of genomes per species. A *pan-genome*, a set of genomes belonging to different strains of the same species, is characterized by a high degree of similarity and redundancy between the genomes [31]. All HTS-specific compression tools can only consider redundancy and similarity within a single genome and not in a collection of genomes. Furthermore, large-scale sequencing projects such as the 1000 Genomes Project [7] may take years to complete, making the compression of continually growing pan-genomes a challenging process.

Existing approaches. HTS-specific compression tools are divided into two categories: *reference-based* and *de novo*. Reference-based methods generally provide high compression ratio by encoding similarities between the read sequences (*reads*) and a reference sequence (*reference*) usually by mapping the reads to the reference. These tools require that the reference used for compression is provided with the compressed archive for decompression, adding extra storage and transmission costs. Note that only a small fraction of sequenced species that are accessible in public databases have such a reference available. On the other hand, *de novo* compression tools perform similarity search within a set of reads in order to exploit its redundancy. BARCODE [26] is a reference-based method that makes use of cascading Bloom filters [29] to compress reads. It inserts reads perfectly matching to a reference into a Bloom filter [2] that can generate false positives. To reduce the number of false positives, BARCODE subsequently inserts them into cascading Bloom filters to tell apart false positives from true positives. Kpath [18] constructs a de Bruijn graph from the reference and encodes each read as a path within the graph. The paths within the

graph are then encoded via arithmetic coding [33]. The beginnings of such paths are stored separately in a trie and encoded with LZ-77. QUIP [17] uses a lossless compression algorithm based on adaptive arithmetic encoding of the identifier, read and quality score streams of the FASTQ format. A reference sequence and a sequence alignment of the reads can be used to improve compression of the reads. QUIP can also perform assembly-based compression. Similar methods are used in FASTQZ and FQZCOMP [3]. SCALCE [12] uses *core substrings* as a measure of similarity in order to cluster similar reads together. Those core substrings are generated via Locally Consistent Parsing (LCP) [28]. SCALCE compresses the reads in each cluster with gzip. ORCOM [11] re-orders reads by similarity as well: it creates clusters of reads that share the same minimizer [24], i.e. the lexicographically smallest p-mer of each read with p usually between 8 to 15. Reads of the same cluster are then merged and compressed. Similar to ORCOM, Mince [22] uses the minimizer approach for clustering. For each read to process, a set of candidate clusters is first established from the k-mers it is composed of. The read is then assigned to the candidate cluster that maximizes the number of q-mers they share. If the read has no candidate cluster, it is assigned to a new cluster corresponding to its minimizer of length k. DSRC 2 [25] compresses the different streams of FASTQ files with different methods: arithmetic coding, Huffman coding [16], as well as 2 bits per base in the case of the DNA sequence stream. Finally, LEON [1] encodes the reads as paths of a de Bruijn graph represented with a Bloom filter. The de Bruijn graph is built from *solid* k-mers of the reads, i.e. k-mers occurring multiple times in the reads. A read is anchored in the graph if it contains at least one solid k-mer and encoded as a list of graph bifurcations from this anchor.

Contributions. In this paper, we present a new alignment-free and reference-free method, DARRC, that compresses the sequencing reads dynamically. The main contribution of this work is the guided de Bruijn graph (gdBG) which allows a unique traversal to reconstruct the reads it was build from. The gdBG is indexed using a colored de Bruijn graph succinct data structure, the Bloom Filter Trie (BFT) [14] that enables the update of the gdBG with reads of other similar genomes. Additional methods are presented to optimize the encoding of the reads. On a large *P. aeruginosa* dataset, DARRC outperforms all other tested tools. It provides a 30% compression ratio improvement in single-end mode compared to the best performing state-of-the-art HTS-specific compression method in our experiments.

2 Methods

A *string* x is a sequence of symbols drawn from a finite, non-empty set, called the *alphabet* $\mathcal{A}$. Its *length* is denoted by $|x|$. Strings are concatenated by juxtaposition. If $x = ps$ for (potentially empty) strings p and s, then p is a *prefix* and s is a *suffix* of x. The symbol at position i is denoted by $x[i]$, the suffix starting at position i by $x(i)$, the substring starting at position i and having length l by $x(i, l)$.

2.1 The de Bruijn graph

A de Bruijn graph (dBG) is a directed graph $G = (V_G, E_G)$ in which each vertex $v \in V_G$ represents a k-mer, a string of length k over $\mathcal{A}$. A directed edge $e \in E_G$ from vertex v to vertex v' representing k-mers x and x', respectively, exists if and only if $x(2, k-1) = x'(1, k-1)$. Each k-mer x has $|\mathcal{A}|$ possible successors $x(2, k-1)c$ and $|\mathcal{A}|$ possible predecessors $cx(1, k-1)$ with $c \in \mathcal{A}$. A colored de Bruijn graph (cdBG) is a dBG $G = (V_G, E_G, C_G)$ in which C_G is a set of colors such that each $v \in V_G$ contains a subset of C_G. A lightweight representation of dBGs and cdBGs does not store edges since they are implicitly given by vertices overlapping on $k-1$ symbols. However, implicit edges can falsely connect vertices that share an overlap of $k-1$ but do not overlap in the sequences the graph was built from.

The dBG is a long-studied abstract data structure used in computational biology. It is particularly useful for the problem of *read assembly* [6] in which the goal is to reconstruct a genome as a single sequence from a set of reads. For this purpose, it is necessary to find a Hamiltonian cycle in the graph, a path starting and ending on the same vertex that visits each vertex exactly once. Although heuristics exist to extract Hamiltonian cycles from a graph, the read assembly problem is yet to be solved because a Hamiltonian cycle is only one possible reconstruction of the original genome the graph was built from.

2.2 The guided de Bruijn graph

The read assembly problem shows that different traversals of dBGs are possible. In the worst case, the number of possible paths between two vertices in a graph is infinite if the graph is cyclic, and exponential otherwise. Given a dBG built from a sequence and a starting vertex for the traversal, the dBG must be supplemented with information to guide its traversal in order to reconstruct the sequence it was built from.

Definition 1. *Given a de Bruijn graph G built from a sequence S, a* partition *$part(G, S)$ is a subgraph G' of G such that G' is a path graph that reconstructs a subsequence of S.*

A guided de Bruijn graph (gdBG) built from a sequence S is a cdBG $G = (V_G, E_G, P_G)$ in which the set of colors, now denoted as P_G, represents partitions guiding the traversal of G to reconstruct S. Self-overlapping k-mers, for which the prefix of length $k-1$ is equal to the suffix of length $k-1$, require a special treatment to avoid looping on themselves within the same partition. Algorithm 1 creates a gdBG G from a sequence S using vertices of length k. It returns all information necessary to reconstruct S: the gdBG encoding S and the $k-1$ length prefix of the first k-mer of S starting the graph traversal for decoding. Note that self-overlapping k-mers terminate their partition such that the next inserted k-mers start a new one (line 9). The algorithm requires $\mathcal{O}(|S|)$ time and $\mathcal{O}(|G|)$ space where $|G| = |V_G| + |P_G|$ if the gdBG uses an implicit representation of edges.

Algorithm 1 Encode(S, k)

```
1: p ← 1                                                  ▷ partition index
2: G ← the empty graph
3: for i ← 1, ..., |S| − k + 1 do
4:     x ← S(i, k)
5:     Y ← {y | y successor of x in G with p ∈ G[y]}
6:     if Y ≠ ∅ then p ← p + 1
7:     if x ∈ G then G[x].add(p)                          ▷ add p to vertex x in G
8:     else G.add(x, p)                                   ▷ insert vertex x with p in G
9:     if x(2, k − 1) = x(1, k − 1) then p ← p + 1
10: return (G, S(1, k − 1))
```

Algorithm 2 decodes a sequence S from a gdBG G using vertices of length k starting with k-mer prefix x. Algorithm 1 guarantees that for any k-mer and one of its partitions, this k-mer can only have zero or one successor in the graph with the same partition. Therefore, Algorithm 2 traverses the graph by searching, for each traversed vertex, the successor with the same partition. If it is not found, the partition index is incremented and the traversal continues. As for Algorithm 1, the algorithm requires $\mathcal{O}(|S|)$ time and $\mathcal{O}(|G|)$ space.

Figure 1 represents a simple cyclic dBG built from a sequence containing a repetition. An infinite number of sequences could be extracted

Algorithm 2 Decode(G, x, k)

```
1: p ← 1
2: z ← k-mer y in G with y(1, k − 1) = x and p ∈ G[y]
3: x ← z
4: S ← z
5: Z ← {z}
6: if z(2, k − 1) = z(1, k − 1) then p ← p + 1
7: while Z ≠ ∅ and p ∈ P_G do
8:     Z ← {z | z successor of x in G with p ∈ G[z]}
9:     if Z contains exactly one element z then
10:        S ← Sz[k]
11:        x ← z
12:        if x(2, k − 1) = x(1, k − 1) then p ← p + 1
13:    else
14:        p ← p + 1
15:        Z ← {x}
16: return S
```

from the dBG because of the cycle. However, by augmenting the dBG with partitions, Algorithm 2 will traverse the cycle only once during the reconstruction of the sequence. Indeed, when Algorithm 1 tries to insert k-mer *agt* with partition 1, a successor with the same partition is found. Therefore, k-mer *agt* is inserted with partition 2 such that the cycle is not contained in one partition.

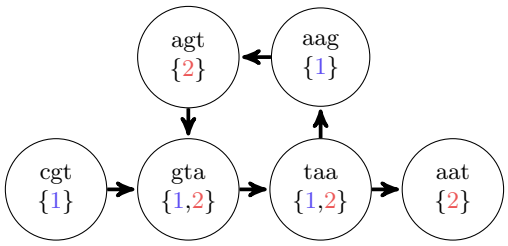

Fig. 1. The guided de Bruijn graph of sequence S = cgtaagtaat as constructed by Algorithm 1 with $k = 3$.

An important property of gdBGs using implicit edges is that no false implicit edge can be traversed during the decoding.

Proposition 1. *Let G be a gdBG built from a sequence S using an implicit representation of edges. An edge between vertices v and v' corresponding to k-mers x and x' respectively, such that $x(2, k-1) = x'(1, k-1)$*

but $xx'[k]$ is not a substring of S is called a false implicit edge. Algorithm 2 does not consider any false implicit edge when traversing G to reconstruct S.

Proof. If a false implicit edge connects vertices not sharing a partition, Algorithm 2 will not consider this edge as only successors with the same partition are traversed. If a false implicit edge connects vertices v and v' which share a partition, the edge out-degree of v is at least 2 and the edge in-degree of v' is at least 2: one true implicit edge each and at least one false implicit edge each. As these vertices are branching, Definition 1 guarantees that v and v' are not in the same partition. □

Algorithm 1 does not distinguish true implicit edges from false implicit edges, ensuring that Definition 1 is always respected during the encoding.

Furthermore, partitions allow to apply the following generalized definition of edges in dBGs to gdBGs:

Definition 2. *In a de Bruijn graph, a directed edge from vertex v to vertex v' representing k-mers x and x', respectively, exists if and only if $x(l+1, k-l) = x'(1, k-l)$ with $l \geq 1$.*

For a sequence S to encode in a gdBG and $l > 1$, $\left\lfloor \frac{|S|-k+1}{l} \right\rfloor$ k-mers will be inserted instead of $|S| - k + 1$. However, the graph can contain more partitions as each vertex has now $|\mathcal{A}|^l$ possible successors and predecessors. Figure 2 gives the gdBG encoding the same sequence as in Figure 1 using a k-mer overlap of $k-2$ instead of $k-1$. The resulting gdBG contains only half the number of vertices than the one in Figure 1.

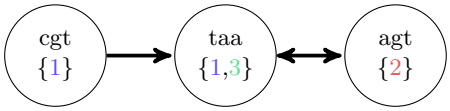

Fig. 2. The guided de Bruijn graph of sequence S = cgtaagtaat using 3-mers overlapping on $k - l = 1$. The last symbol of S is not encoded in the gdBG as it cannot be part of a k-mer.

3 Compression

Section 2 presented methods to encode a sequence as a gdBG and to decode it. In this section, we describe how to use this methodology to compress reads. To improve compression efficiency, we preprocess the reads.

3.1 Read clustering and merging

A simple form of read assembly extended from ORCOM [11] is performed to reduce the input data. It clusters reads according to their minimizer, then merges reads sharing an overlap within each cluster and finally merges reads sharing an overlap but originating from different clusters. These three steps are described in the following.

Clustering. The minimizer [24] of a read r is the lexicographically smallest of its p-mers with $p \ll |r|$. The canonical minimizer of r is the lexicographically smallest minimizer of r and its reverse-complement $\overline{r}$. The following method is based on the simple assumption that reads sharing a minimizer are likely to share a longer overlap and therefore be similar. Thus, the canonical minimizer m is computed for each read r such that r or $\overline{r}$ is assigned to its cluster m.

Intra-cluster merging. Within each cluster, the reads are sorted by decreasing position of their minimizer, in which reads sharing the same minimizer position are sorted lexicographically. For each read r and its minimizer m at position p_m, all reads r' with minimizers at positions $p'_m \leq p_m$ are considered for merging, in decreasing order of positions p'_m to maximize the overlap lengths. To merge reads r and r', they are first anchored at the position of their minimizers such that they overlap on $o = p'_m + \min(|r| - p_m, |r'| - p'_m)$ symbols. Reads are merged into a *super read* [35] if $r(p_m - p'_m, o) = r'(1, o)$ with at most d mismatches. The same process is applied to the created super read in order to merge it with the remaining reads of the cluster. For each super read, we encode all of its read meta data in separate streams: position, length, reverse-complement information and mismatches.

Inter-cluster merging. As an extension of the previous steps used by ORCOM, we additionally perform a process similar to the intra-cluster read merging described previously to merge super reads from multiple clusters. For each super read sr and its minimizer m at position p_m, a new minimizer m' is computed in $sr(p_m + 1)$ and $\overline{sr}$. All super reads of cluster m' are considered for a merging with sr or $\overline{sr}$. Merging two or more super reads creates a *spanning super read* (SSR). The same process is applied to the created SSR until no super reads can merge with it.

Paired-end reads. Each mate of a pair is considered as a single read that is clustered and merged using the previously described methods. However, the clustering and merging steps keep track of the position of the mates in the SSRs. This information is used afterwards to store in each read meta data whether the read is the first mate of its pair. In such case, the position of its corresponding mate in the SSRs is stored as well.

3.2 Spanning Super Read encoding

Encoding a set of SSRs using a gdBG requires to extract k-mers from the SSRs. If edges represent overlaps of length $k-1$, all k-mers of the SSRs are extracted. If edges represent overlaps of length $k-l$ with $l > 1$, k-mers are extracted every l positions. As a consequence, similar SSRs can have different sets of k-mers. An example is given in Figure 3, in which two similar SSRs, ssr_1 and ssr_2, do not share any k-mers because they are extracted every $l = 2$ positions from the first position of each SSR. By shifting the k-mer extraction start position by one position in the second SSR, as shown with $ssr_2{}'$, two extracted k-mers are shared with the first SSR.

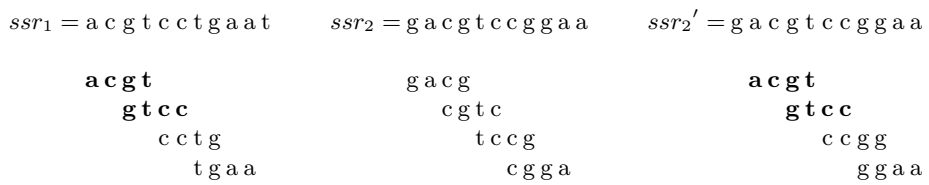

Fig. 3. Extraction of 4-mers overlapping on $k-l=2$ from two similar SSRs, ssr_1 and ssr_2.

In order to keep the growth of the gdBG small when inserting a new SSR, we determine the k-mer extraction start position, called *start position* in the following, that maximizes the number of k-mers already stored in the gdBG. To this end, we maintain in memory a k-mer index recording all k-mers extracted. As the cost in time and memory of such an index is prohibitive, we use a Bloom filter instead.

Introduced by Bloom, a *Bloom filter* (BF) [2] records the presence of elements in a set. Based on the hash table principle, look-up and insertion times are constant. The BF is composed of an array B of b bits, initialized with 0s, in which the presence of n elements is recorded. A set of f hash functions $h_1, ..., h_f$ is used, such that for an element e, $h_i(e) : e \rightarrow \{1, .., b\}$.

Inserting an element into B and testing for its presence are then

$$\mathsf{Insert}(e, B) : B[h_i(e)] \leftarrow 1 \text{ for all } i = 1, ..., f$$

and

$$\mathsf{MayContain}(e, B) : \bigwedge_{i=1}^{f} B[h_i(e)],$$

respectively, in which $\bigwedge$ is the logical conjunction operator. The BF does not generate false negatives but may generate false positives, as MayContain can report the presence of elements which are not present but a result of independent insertions.

We propose a greedy approach making use of the BF to iteratively detect for each SSR of a set its optimal start position and updating the BF with all novel k-mers. The optimal start position of an SSR is a position from 1 to l maximizing the number of k-mers extracted that are already present in the BF compared to the other possible start positions. Once the optimal start position of an SSR is determined, the BF is updated with the k-mers extracted and the next SSR of the set is processed. To encode all SSRs completely, this approach does not only returns the k-mers to insert into a gdBG, because these do not necessarily cover the entire SSRs. It also returns the *head* and *tail* of each SSR, which are the uncovered prefix and suffix, respectively, not encoded in the gdBG. Additionally, to provide an entry point into the gdBG for the decoding, it returns the *starting overlap* of each SSR, which is the $k - l$ length prefix of the first k-mer. More precisely, we denote by x and y the first and last k-mers extracted, respectively, from an SSR ssr with pos_x and pos_y as their respective occurrence positions in ssr. Then, the head of ssr is the prefix $ssr(1, pos_x - 1)$, the tail of ssr is the suffix $ssr(pos_y + k)$, and the starting overlap of ssr is $ssr(pos_x, k - l)$. SSR heads, tails and starting overlaps are encoded in separate streams and compressed separately from the gdBG.

3.3 Partition encoding

Encoding. Partition sets associated with k-mers in gdBGs are represented as lists of sorted integers. A naive way to store a partition set is to use a fixed number of bytes for each partition. For example, 4 bytes is a standard size for integers on current computer architectures. In order to decrease the memory footprint while keeping the lists indexed, partitions are first delta encoded by storing the difference between each integer and

its predecessor in the list (or 0 if the integer is in first position). The resulting values are called *deltas*. However, it only decreases the minimum number of bits necessary to encode the partitions but not their final representation. Consequently, deltas are Vbyte encoded [32]: each byte used to encode a delta has one bit indicating whether the byte starts a new delta or not, allowing to remove unnecessary bytes from each delta. Thus, partitions use a variable number of bytes proportional to the minimum number of bits necessary to encode their deltas.

Recycling. As a small delta produces a small encoding, partition integers are recycled instead of naively using the next higher integer for every new partition as, for the sake of convenience, described in Algorithm 1. Partition sets a and b can share the same partition integer if they are not neighbors in the graph, i.e., no k-mer suffix or prefix of a overlaps a k-mer prefix or suffix of b, for suffixes and prefixes of length $k - l$. A trivial example is provided in Figure 4 in which k-mer *cttc* uses the same partition integer as k-mer *acgt* because they are not neighbors in the graph.

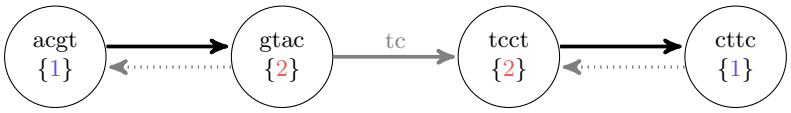

Fig. 4. The guided de Bruijn graph of SSRs ssr_1 = acgtac and ssr_2 = tccttc using 4-mers ($l = 2$). Dotted gray edges are implicit edges. The solid gray edge exists by using the starting overlap of ssr_2 after the traversal of ssr_1, as described in Section 3.2.

As there can be a large number of partitions in the graph, verifying the connectivity of one partition to all other partitions is often impractical. We propose instead a heuristic that verifies the connectivity only to the last t partitions inserted, t being a user-defined threshold, such that these t partitions are the only candidates for recycling. Using partition recycling requires to save the partitions traversal order which cannot be incremental anymore as shown in Algorithms 1 and 2.

3.4 Meta data and gdBG compression

Steps described previously generate meta data specific to one input file such as read lengths and positions in SSRs. These meta data are first encoded in separate streams and are then compressed using an LZ-type

algorithm, LZMA [23]. After all k-mers and partitions are inserted in the gdBG, the latter is written to disk. As it must be loaded in memory for every update and decompression, the gdBG is compressed with Zstd [5], a compression method based on Huffman coding and Asymmetric Numeral Systems [9] that favors compression and decompression speed over compression ratio.

4 Update and decompression

In order to update a compressed archive with a new input file, only the gdBG previously created is decompressed and loaded in memory, as meta data are not used for the update. A fast procedure iterates over all k-mers of the gdBG and inserts them into the BF proposed in Section 3.2 instead of starting with an empty BF in order to optimize the choice of the k-mer extraction start positions in the SSRs. The gdBG is then updated with the new k-mers and partitions. The starting partition index is greater than the partition indexes already present in the gdBG, ensuring that each input file is encoded with a unique set of partitions.

Decompressing a read file starts with decompressing its meta data and the gdBG it is encoded in. The gdBG is then loaded in memory and Algorithm 2 is used to traverse the gdBG, but only following those partitions that are specific to the read file to decompress. This way, single files can be decompressed separately. As Algorithm 2 decodes SSRs, meta data are used afterwards to extract the actual reads. If reads are paired-end, meta data are also used to reorganize them such that corresponding mates of the same pair are together in the decompressed file.

5 Results

DARRC is implemented in C and uses the Bloom Filter Trie (BFT) library [14] for its gdBG. The BFT provides time and space efficient functionalities that are required by DARRC. These functionalities include: (i) the ability to update the BFT with new k-mers and colors without recomputing the index, (ii) k-mers extraction from the BFT and (iii) prefix search over the set of k-mers within the BFT. The software is available at `https://bitbucket.org/GuillaumeHolley/darrc/`. We compared DARRC to three state-of-the-art *de novo* DNA sequence compression tools: ORCOM [11], LEON [1] and Mince [22]. DARRC was also compared to the same LZ-type algorithm used to compress its meta data, LZMA [23]. Experiments were carried out on a server with 378 GB

of RAM and two 8-core Intel Xeon E5-2630 v3 2.4 GHz processors. All input files were placed on mechanical hard drives. Compressed archives and decompressed files, during compression and decompression respectively, together with temporary files such as read clusters were written to a RAM-based partition when the tools allowed to specify an output directory. As the current version of DARRC does not take advantage of parallelism, all software were run using a single thread, except Mince which requires a minimum of four threads. All *de novo* DNA sequence compression tools were run using their default parameters. LZMA was run with the same compression level as the one used to compress DARRC meta data. DARRC default parameters are minimizers of length 9 for the clustering, 5 mismatches allowed per read merging and 36-mers overlapping on 11 symbols for the gdBG. ORCOM, LEON, Mince and LZMA compressed all files in separate archives while DARRC updated the same archive iteratively with the files to compress: each iteration decompressed and reloaded the necessary data from the data written to disk in the previous iteration. The dataset used for the experiment consists of 473 clinical isolates of *Pseudomonas aeruginosa* sampled from 34 patients (NCBI BioProject PRJEB5438), resulting in 338.61 Gbp of high coverage sequences. Reads are 100 bp paired-end reads generated by Illumina HiSeq 2000. Pair mates were placed in different files for every isolate. The experiment was run in single-end mode and paired-end mode for all tools such that in the single-end mode, every mate file is considered as a single-end read file. The appropriate single-end and paired-end modes were used for DARRC and Mince. The mates were concatenated for the paired-end experiment of ORCOM as the tool neither preserves the order of the reads nor stores the paired-end information. LEON and LZMA do not have an explicit paired-end mode but keep the original order of the reads, thus for the paired-end experiment of LEON and LZMA, the mate files of every isolate were concatenated.

Compression ratios in paired-end mode and single-end mode are shown in Figure 5. DARRC clearly outperforms all the other tested tools in both modes. In paired-end mode and single-end mode, DARRC uses about 0.261 bits per base and 0.204 bits per base, corresponding to a 57 % and 30 % compression ratio improvement compared to the second best results, respectively. The paired-end compression ratio of ORCOM compared to its single-end compression ratio shows that the tool is not adapted to paired-end read compression. The gdBG represents about 10 % and 13 % of the data written to disk by DARRC in paired-end mode and single-end mode, respectively.

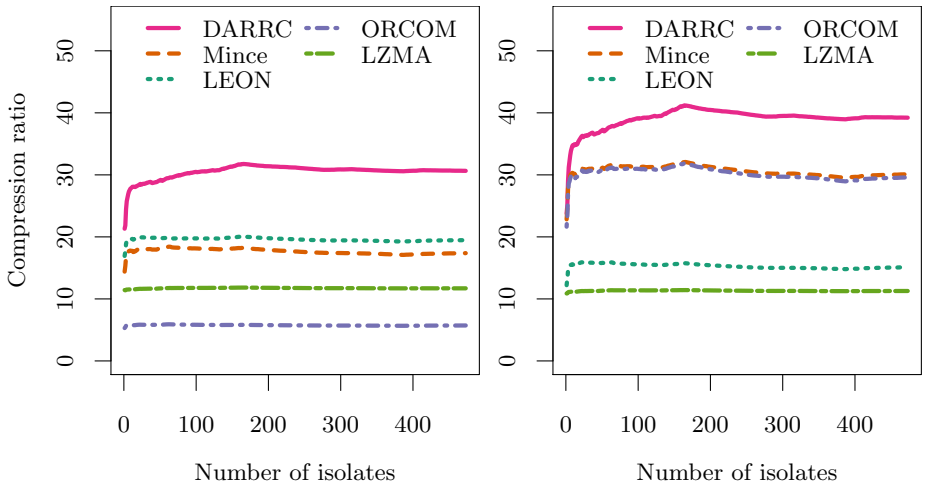

Fig. 5. Compression ratios in paired-end mode (left) and single-end mode (right).

DARRC compressed more than two times faster than LZMA but used the most time to decompress, as shown in Figures 6 and 7, respectively. DARRC's compression time overhead is explained by the fact that at each iteration, the gdBG must be decompressed, loaded in memory and updated with new k-mers and partitions.

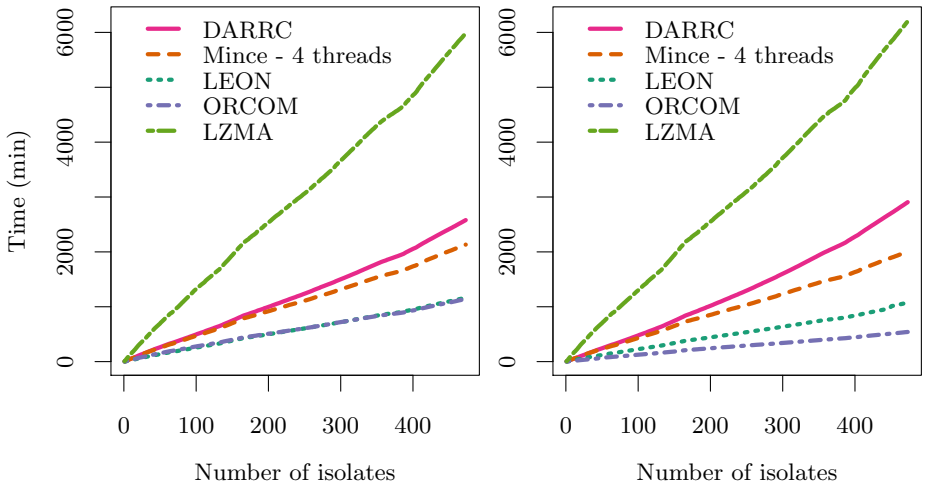

Fig. 6. Compression times in paired-end mode (left) and single-end mode (right).

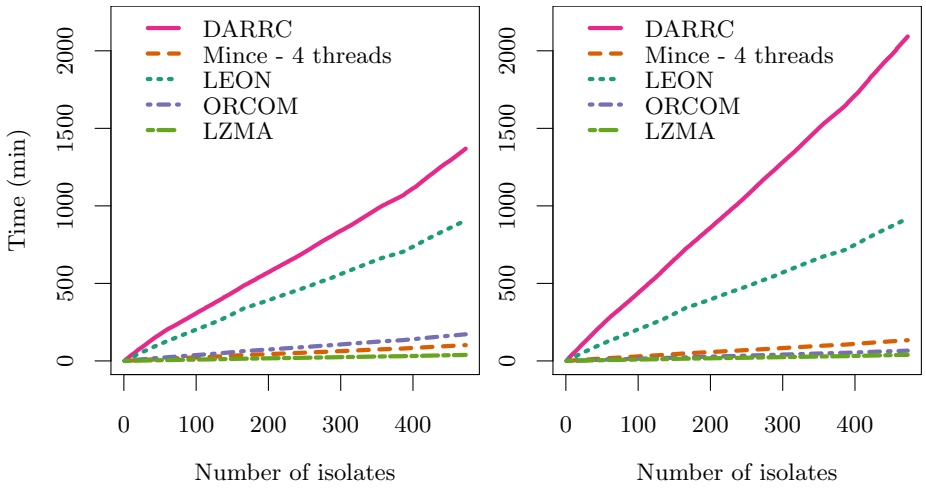

Fig. 7. Decompression times in paired-end mode (left) and single-end mode (right).

All tools performed compression and decompression using a maximum of four gigabytes of main memory, an amount nowadays available on most desktop computers and laptops. Even by updating the same archive iteratively, DARRC compression used less than two gigabytes of main memory.

6 Conclusions and future work

We presented DARRC, a dynamic alignment-free and reference-free read compression method that can incrementally update compressed archives with new genome sequences without full decompression of the archives. DARRC uses a new abstract data structure, the guided de Bruijn graph, that allows a unique traversal of the de Bruijn graph to reconstruct the sequences it is built from. We showed that, on a large pan-genome dataset, our method outperforms several state-of-the-art DNA sequence compression methods and a general purpose compression tool regarding the compression ratio while achieving reasonable running time and main memory usage. Furthermore, we showed that the compression ratio of DARRC is attractive even with only few files compressed. Future work concerns the parallelization of the software, particularly the read clustering and merging phase which offers a lot of potential for multi-threading. Additionally, a logical evolution of DARRC is the introduction of a pattern matching functionality within the compressed data as in [34], leading to

large scale complex methods such as read alignment and variant calling using multiple genomes.

Acknowledgments. This research is funded by the International DFG Research Training Group GRK 1906/1 for GH and RW, the NSERC Discovery Frontiers grant on "Cancer Genome Collaboratory" to FH.

References

1. Gaetan Benoit, Claire Lemaitre, Dominique Lavenier, Erwan Drezen, Thibault Dayris, Raluca Uricaru, and Guillaume Rizk. Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC Bioinformatics*, 16:288, 2015.
2. Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Comm. ACM*, 13(7):422–426, 1970.
3. James K Bonfield and Matthew V Mahoney. Compression of FASTQ and SAM format sequencing data. *PloS One*, 8(3):e59190, 2013.
4. Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. *Digital SRC Research Report 124*, 1994.
5. Yann Collet. ZSTD. `https://github.com/facebook/zstd`. [Online: 20-Dec-2016].
6. Phillip EC Compeau, Pavel A Pevzner, and Glenn Tesler. How to apply de Bruijn graphs to genome assembly. *Nat. Biotechnol.*, 29(11):987–991, 2011.
7. 1000 Genomes Project Consortium. A global reference for human genetic variation. *Nature*, 526(7571):68–74, 2015.
8. Sebastian Deorowicz and Szymon Grabowski. Data compression for sequencing data. *Algorithms Mol. Biol.*, 8:25, 2013.
9. Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv:1311.2540*.
10. Raffaele Giancarlo, Simona E Rombo, and Filippo Utro. Compressive biological sequence analysis and archival in the era of high-throughput sequencing technologies. *Brief. Bioinform.*, 15(3):390–406, 2014.
11. Szymon Grabowski, Sebastian Deorowicz, and Lukasz Roguski. Disk-based compression of data from genome sequencing. *Bioinformatics*, 31(9):1389–1395, 2014.
12. Faraz Hach, Ibrahim Numanagić, Can Alkan, and S Cenk Sahinalp. SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28(23):3051–3057, 2012.
13. Richard CG Holland and Nick Lynch. Sequence squeeze: an open contest for sequence compression. *GigaScience*, 2(1):5, 2013.
14. Guillaume Holley, Roland Wittler, and Jens Stoye. Bloom Filter Trie: an alignment-free and reference-free data structure for pan-genome storage. *Algorithms Mol. Biol.*, 11:3, 2016.
15. Morteza Hosseini, Diogo Pratas, and Armando J Pinho. A Survey on Data Compression Methods for Biological Sequences. *Information*, 7(4):56, 2016.
16. David A Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the IRE*, 40(9):1098–1101, 1952.
17. Daniel C Jones, Walter L Ruzzo, Xinxia Peng, and Michael G Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Res.*, 40(22):e171–e171, 2012.

18. Carl Kingsford and Rob Patro. Reference-based compression of short-read sequences using path encoding. *Bioinformatics*, 31(12):1920–1928, 2015.
19. Miriam Land, Loren Hauser, Se-Ran Jun, Intawat Nookaew, Michael R Leuze, Tae-Hyuk Ahn, Tatiana Karpinets, Ole Lund, Guruprased Kora, Trudy Wassenaar, et al. Insights from 20 years of bacterial genome sequencing. *Funct. Integr. Genomics*, 15(2):141–161, 2015.
20. Po-Ru Loh, Michael Baym, and Bonnie Berger. Compressive genomics. *Nat. Biotechnol.*, 30:627–630, 2012.
21. Ibrahim Numanagić, James K Bonfield, Faraz Hach, Jan Voges, Jörn Ostermann, Claudio Alberti, Marco Mattavelli, and S Cenk Sahinalp. Comparison of high-throughput sequencing data compression tools. *Nat. Methods*, 13(12):1005–1008, 2016.
22. Rob Patro and Carl Kingsford. Data-dependent bucketing improves reference-free compression of sequencing reads. *Bioinformatics*, 31(17):2770–2777, 2015.
23. Igor Pavlov. LZMA. `http://www.7-zip.org`. [Online: 20-Dec-2016].
24. Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.
25. Lukasz Roguski and Sebastian Deorowicz. DSRC 2–Industry-oriented compression of FASTQ files. *Bioinformatics*, 30(15):2213–2215, 2014.
26. Roye Rozov, Ron Shamir, and Eran Halperin. Fast lossless compression via cascading Bloom filters. *BMC Bioinformatics*, 15(9):S7, 2014.
27. Simanto Saha and Sanguthevar Rajasekaran. Efficient algorithms for the compression of FASTQ files. In *Proc. of the International Conference on Bioinformatics and Biomedicine (BIBM'14)*, pages 82–85, 2014.
28. S Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. *FOCS*, pages 320–328, 1996.
29. Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading Bloom filters to improve the memory usage for de Brujin graphs. *Algorithm. Mol. Biol.*, 9(1):2, 2014.
30. Genome Biology Editorial Team. Closure of the NCBI SRA and implications for the long-term future of genomics data storage. *Genome Biol.*, 12(3):402, 2011.
31. Hervé Tettelin, Vega Masignani, Michael J Cieslewicz, Claudio Donati, Duccio Medini, Naomi L Ward, Samuel V Angiuoli, Jonathan Crabtree, Amanda L Jones, A Scott Durkin, et al. Genome analysis of multiple pathogenic isolates of *Streptococcus agalactiae*: implications for the microbial pan-genome. *Proc. Natl. Acad. Sci. USA*, 102(39):13950–13955, 2005.
32. Hugh E Williams and Justin Zobel. Compressing integers for fast file access. *Comput. J.*, 42(3):193–201, 1999.
33. Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Commun. ACM*, 30(6):520–540, 1987.
34. Y William Yu, Noah M Daniels, David Christian Danko, and Bonnie Berger. Entropy-scaling search of massive biological data. *Cell Syst.*, 1(2):130–140, 2015.
35. Aleksey V Zimin, Guillaume Marçais, Daniela Puiu, Michael Roberts, Steven L Salzberg, and James A Yorke. The MaSuRCA genome assembler. *Bioinformatics*, 29(21):2669–2677, 2013.
36. Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23(3):337–343, 1977.