Scalable and Versatile k-mer Indexing for High-Throughput Sequencing Data

Niko Välimäki, Eric Rivals

Scalable and Versatile k-mer Indexing for High-Throughput Sequencing Data

Niko Välimäki*
Genome-Scale Biology Research Program,
and Department of Medical Genetics, Faculty of Medicine,
University of Helsinki, Finland
niko.valimaki@helsinki.fi

Eric Rivals†
LIRMM - UMR 5506 CNRS & Université Montpellier 2,
and Institut de Biologie Computationelle,
France
rivals@lirmm.fr

March 29, 2013

Abstract

Philippe et al. (2011) proposed a data structure called *Gk arrays* for indexing and querying large collections of high-throughput sequencing data in main-memory. The data structure supports versatile queries for counting, locating, and analysing the coverage profile of k-mers in short-read data. The main drawback of the *Gk* arrays is its space-consumption, which can easily reach tens of gigabytes of main-memory even for moderate size inputs. We propose a compressed variant of *Gk* arrays that supports the same set of queries, but in both near-optimal time and space. In practice, the compressed *Gk* arrays scale up to much larger inputs with highly competitive query

*Funded by Academy of Finland CoE in Cancer Genetics Research (No 250345).
†Funded by the ANR Colib'read, MASTODONS, PICS.

times compared to its non-compressed predecessor. The main applications include variant calling, error correction, coverage profiling, and sequence assembly.

1 Introduction

High Throughput Sequencing (HTS) gives access to the whole complement of DNA or RNA sequences present in a biological sample. A single machine yields hundreds of million of short sequencing reads in a short time for a price that is steadily decreasing. Large sequencing centers produce daily tens of terabytes of data, and for instance the Beijing Genome Institute has launched in 2012 a project for sequencing 3 millions of genomes. Applications of HTS go far beyond genome sequencing, and are now used in the medical context for diagnostic and disease follow-up, or in ecology for monitoring biodiversity. In the later context, HTS sequence all DNA/RNA coming from all species present in an environmental sample (*i.e.*, a soil, a sea, or a gut sample). In such *meta-genomics* or *-transcriptomics* experiments, one aims at identifying the species or the genes they expressed in this environment, which is achieved by clustering and mining the reads based on sequence similarity.

HTS pushes life sciences in a Big Data era and fosters the development of efficient and scalable algorithms for analyzing huge read sets. A variety of computational questions need to be solved from genome assembly, to read clustering by similarity, going through read mapping (*i.e.* alignment) on a reference genome. Many tasks require indexing data structures that allow querying the reads for an exact or approximate sequence pattern. Most efficient programs for mapping reads onto a reference genome resort to a *FM-index* of the genome [7], which is small enough to fit in memory (*e.g.* [16, 20]). However, in many applications, a reference genome is missing and the read set must be mined on its own (de novo genomics, transcriptomics, or in meta-genomics), but the volume of reads is much larger than a reference genome. Let us review shortly existing work on read indexing data structures.

Error correction and k-mer counting. Sequencing errors cause important difficulties for read analysis or genome assembly. The correction or elimination of erroneous reads is made possible by the redundancy due to high sequencing coverage. The solution is to monitor the occurrence number of all k-mers within the reads to see if they conform to the expected cover-

age. For the task of k-mer counting, efficient hashing techniques have been developped using parallel algorithms or Bloom filters [17, 18]. However, their lack of scalability hinders indexing all 27-mers of typical Human sequencing data set [22]. A recent paper achieves scalability by partitioning the index between memory and disk [22]. Some assemblers use a parallel k-mer counting index to discard erroneous reads during the deBruijn graph construction [4]. Various error correction methods implement the same strategy using a hash table. For example, Coral [23] identifies sequencing errors by indexing k-mers into a hash table and then computing multiple alignments over reads that share a common k-mer. The hash table requires $\Theta(n \log n)$ bits [23], which can make the approach infeasible for HTS data.

Read indexing in mapping. Following the idea of error correction, it has been proposed to compute the *local coverage* of any k-mer in a read, that is the number of reads in which in appears. Inspecting the local coverage profile of k-mers along the read enables the tool CRAC to distinguish erroneous positions from point mutations directly during the mapping [20] (`http://crac.gforge.inria.fr`). For this, CRAC resorts to a data structure called *Gk* arrays which indexes all k-mers occurring within each read of the *collection*[1] using a modified suffix array and complementary tables [21]. It takes advantage from the fact that reads are often compared against themselves and that queried k-mers are taken from a read and can be given by a starting position rather than in extenso. *Gk* arrays offers seven types of locate and counting queries: either for getting the read identifiers in which a k-mer occurs (Q1/Q2), occurs at most once (Q5/Q6), and the occurrence positions with (Q7) or without this restriction (Q3/Q4). Table 1 gives an overview of the queries and theoretical properties of the data structure. *Gk* arrays uses a space proportional to the length of the read collection. Hence, indexing for instance a metagenomics dataset will exhaust the main memory of most computers. *Gk* arrays are also limited to queries on a single k value.

For similarity searching for assembly and clustering. When many transcribed RNAs are sequenced in proportion of their abudance, it is useful to reduce the data by clustering reads or ESTs that originate from the same molecule. EST clustering was already critical before the advent of HTS [3].

[1] In a collection, each read sequence can occur many times, but differ by their identifier, sequence quality, or mate partner. It is a multi-set rather than a set.

Table 1: Theoretical time and space complexities. Here n is the input size, f is the query k-mer, σ is the alphabet size, and $H_h \leq H_0 \leq \log \sigma$ denotes the h-th order entropy. The output size of each query is denoted by $|\mathsf{Q7}| = |\mathsf{Q5}| \leq |\mathsf{Q1}| \leq |\mathsf{Q3}|$, where $|\mathsf{Q3}|$ is the total number of occurrences and, thus, can be significantly larger than the others. Philippe et al. [21] reported a linear time construction, but omitted their worst-case time of radix-sorting over $\lceil \log n \rceil$-bit integers.

Data structure		Compressed Gk arrays	Gk arrays
Construction	time	$O(n \log n)$	$O(n \log n)$
	space (bits)	$O(n(H_0 + 1))$	$\Theta(n \log n)$
Final index size (bits)		$nH_h \log \log_\sigma n + O(n)$	$\Theta(n \log n)$
Query time for	a k-mer	$O(k \log \sigma + \text{polylog}(n))$	$O(k \log n)$
	a position	$O(\log \log n)$	$O(1)$
Additional query time to answer:			
Q1 In which reads does f occur?		$O(\|\mathsf{Q1}\| \log \log n)$	$O(\|\mathsf{Q3}\|)$
Q2 In how many reads does f occur?		$O(1)$	$O(\|\mathsf{Q3}\|)$
Q3 What are the occurrence positions of f?		$O(\|\mathsf{Q3}\| \log \log n)$	$O(\|\mathsf{Q3}\|)$
Q4 What is the number of occurrences of f?		$O(1)$	$O(1)$
Q5 In which reads does f occur only once?		$O(\|\mathsf{Q5}\| \log \log n)$	$O(\|\mathsf{Q3}\|)$
Q6 In how many reads does f occur only once?		$O(1)$	$O(\|\mathsf{Q3}\|)$
Q7 What are the occurrence positions of f in the reads where it occurs only once?		$O(\|\mathsf{Q7}\| \log \log n)$	$O(\|\mathsf{Q3}\|)$

The effiency and scalabilty of clustering algorithms rely on their indexing strategy. Kaboom implements a modified suffix array for this task [10], but cannot scale up to nowadays huge read sets (as shown in [21]). The detection of similarity between reads is also used to discover overlaps and then build the overlap graph for genome assembly. The sparse representation of the relations between substrings and reads is major issue for scalable assembly programs, as exemplified by [6].

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14
T	B	A	N	A	N	A	$\$_1$	A	N	A	N	A	S	$\$_2$
SA	7	14	6	4	2	8	10	12	1	5	3	9	11	13
SA^{-1}	8	5	11	4	10	3	1	6	12	7	13	8	14	2
LCP	0	0	0	1	3	5	3	1	0	0	2	4	2	0
B_{lcp}	1	1	1	1	0	0	0	1	1	1	1	0	1	1
B_{last}	0	0	0	1	0	0	1	0	1	0	1	1	1	0
B_{once}	0	0	1	1	1	1								
$T[\mathsf{SA}[i]\,..\,n]$	$\$_1$	$\$_2$	A	A	A	A	A	A	B	N	N	N	N	S
			$\$_1$	N	N	N	N	S	A	A	A	A	A	$\$_2$
				A	A	A	A	$\$_2$	N	$\$_1$	N	N	S	
				$\$_1$	N	N	S		A		A	A	$\$_2$	
					A	A	$\$_2$		N		$\$_1$	S		
					$\$_1$	S			A			$\$_2$		
						$\$_2$			$\$_1$					

Figure 1: An example of the (inverse) suffix array and LCP array for the input string $T = \texttt{BANANA}\$_1\texttt{ANANAS}\$_2$ and resulting bit-vectors B_{lcp}, B_{last} and B_{once} for $k = 3$. Notice that the size of B_{once} is equal to $\mathsf{rank}_1(B_{\mathsf{last}}, n)$ and it is queried via the 1-bits in B_{last}.

2 Compressed *Gk* arrays

We aim at supporting the same set of queries as the original *Gk* arrays [21]. See Table 1 for the definition of each query Q1–Q7. Notice that Q3 and Q4 are the typical queries found in *full-text indexes* such as suffix trees and suffix arrays, and in their compressed variants (see [19] for a survey). Q1 and Q2 are also known as the *document listing problem* in the field of information retrieval [13].

At the core of our data structure is a *compressed suffix array* (CSA) [9] built on top of all the input reads. More precisely, the collection of reads is given as a (multi-)set[3] of strings $\mathcal{R} = \{r_1, r_2, \ldots, r_d\}$. We assume that the strings are from ordered alphabet Σ of size $|\Sigma| = \sigma = O(\text{polylog}(n))$. We represent $\mathcal{R}$ as one long concatenated string, say $T = r_1\$_1 r_2\$_2 \cdots k_d\$_d$, where each $\$_i$ denotes a special separator-symbol having the lexicographical order $\$_{i-1} < \$_i < c \in \Sigma$ for all $i \in [1, d]$. Let n denote the total length of T. Substrings of T are denoted by $T[i\,..\,j] = T[i]T[i+1]\cdots T[j]$ for any $1 \leq i \leq j \leq n$. The *suffix array* $\mathsf{SA}[1, n]$ of string T stores all suffixes

of T in lexicographical order. The lexicographically i-th suffix is given by $T[\mathsf{SA}[i]\,..\,n]$. The *inverse suffix array* is $\mathsf{SA}^{-1}[j] = i$ iff $\mathsf{SA}[i] = j$. The *Longest Common Prefix (lcp)* table, denoted $\mathsf{LCP}[1, n]$, stores in $\mathsf{LCP}[i]$ the length of the lcp of suffixes $T[\mathsf{SA}[i-1]\,..\,n]$ and $T[\mathsf{SA}[i]\,..\,n]$ for any $1 < i \leq n$, and $\mathsf{LCP}[1] = 0$. Fig. 1 gives an example of the SA, SA^{-1}, and LCP values. The compressed suffix array [9] requires $nH_h \log\log_\sigma n + O(n)$ bits[2] and allows $t_{\mathsf{SA}} = O(\log\log_\sigma n + \log\sigma) = O(\log\log n)$ time access to SA and SA^{-1}.

We aim to support two query-types, that is, queries for both a *given k-mer* and for *a given position p* in the set of indexed reads. Let f denote the given k-mer or the k-mer at the given position p, say $f = T[p\,..\,p+k-1]$ for any $p \in [1, n-k+1]$. Our first problem is to identify the suffix array range $[s, e]$, which covers all the suffixes of T that have f as a prefix. If the query-type is a k-mer, we can simply utilize the search functionality built into the CSA to identify the range $[s, e]$ in $O(k \log\sigma + \text{polylog}(n))$ time [9]. In order to support queries for given read positions, we propose the following data structure:

Lemma 2.1 *Given the CSA of the string $T[1\,..\,n]$, a fixed constant k and a query position $p \in [1, n-k+1]$, we can identify the suffix array range $[s, e]$, which covers all the suffixes of T that have $f = T[p\,..\,p+k-1]$ as a prefix, using $n + o(n)$ additional bits and $O(\log\log n)$ time.*

Proof. We introduce a bit-vector $B_{\mathsf{lcp}}[1, n]$, which is set to $B_{\mathsf{lcp}}[i] = 1$ if and only if $\mathsf{LCP}[i] < k$. Fig. 1 gives an example of the arrays SA, SA^{-1} and LCP and the resulting B_{lcp}. See the following subsection on details about constructing B_{lcp}. Recall that the CSA can simulate the inverse suffix array in $O(\log\log n)$ time. We compute $j = \mathsf{SA}^{-1}[p]$, which gives us a position j in the suffix array. It follows that $s \leq j$ and $e \geq j$, because f is a prefix of suffix $T[\mathsf{SA}[j]\,..\,n] = T[\mathsf{SA}[\mathsf{SA}^{-1}[p]]\,..\,n] = T[p\,..\,n]$. Now we need to identify the suffixes surrounding j that also have f as a prefix. If such suffixes exists, they are identified by taking the smallest $s \in [1, j]$ and largest $e \in [j, n]$ such that $\mathsf{LCP}[i] \geq k$ holds for all $i \in [s+1, e]$. Notice that the bit-vector B_{lcp} encodes this information, and it is accessible in constant time using rank and select queries: The $\mathsf{rank}_b(B, i)$ query over a bit-vector $B[1\,..\,n]$ returns, for any $i \in [1, n]$, the number of times the bit $b \in \{0, 1\}$ occurs in

[2] We denote the *empirical entropy* of a string T with $H_0(T)$ (or simply H_0 if T is clear from the context). The *h-th order entropy* is denoted by $H_h(T)$ (or simply H_h). Notice that $0 \leq H_{h+1}(T) \leq H_h(T) \leq \log\sigma$ for all $h \geq 0$.

$B[1..i]$. The inverse query, $\mathsf{select}_b(B, j)$, returns the position of the j-th bit b in B (moreover, if $j > \mathsf{rank}_b(B, n)$, we agree that $\mathsf{select}_b(B, j)$ returns $n+1$). We first compute $r = \mathsf{rank}_1(B_{\mathsf{lcp}}, j)$, which leads us to the final answer $[s, e] = [\mathsf{select}_1(B_{\mathsf{lcp}}, r), \mathsf{select}_1(B_{\mathsf{lcp}}, r + 1) - 1]$. The rank and select queries over B_{lcp} can be computed in constant time using $n + o(n)$ bits [14]. **Q.E.D.**

Notice that the range $[s, e]$ immediately reveals the total number of occurrences f has in the reads, which is $e - s + 1$. The occurrence positions can be enumerated by outputting $\mathsf{SA}[j]$ for each $j \in [s, e]$. That said, the above lemma allows us to reveal the correct range $[s, e]$ and answer queries Q3 and Q4 in additional $O(|\mathsf{Q3}| \log \log n)$ and $O(1)$ time, respectively. We introduce another data structure to answer Q1 and Q2:

Lemma 2.2 *Given the CSA of the string $T[1..n]$, a fixed constant k and a suffix array range $[s, e]$ covering all the suffixes that have f as a prefix, we can answer the query Q1 (resp. Q2) using $n + o(n)$ additional bits and $O(|\mathsf{Q1}| \log \log n)$ time (resp. $O(1)$ time), where $|\mathsf{Q1}|$ is the number of reads having an occurrence of f.*

Proof. Let $B_{\mathsf{last}}[1, n]$ denote a bit-vector, which is initialized as follows: we set $B_{\mathsf{last}}[j] = 1$ if and only if the k-mer $f = T[\mathsf{SA}[j]..\mathsf{SA}[j] + k - 1]$ starting from text position $p = \mathsf{SA}[j]$ is the last occurrence of f within the corresponding read. That is, we mark the "unique" k-mers for each read and, as an important detail, this marking is stored in the suffix array order. See the following subsection on details about constructing B_{last}. Furthermore, k-mers that span over a separator-symbol are never marked. We can use B_{last} to directly count and enumerate the reads that contain at least one occurrence of f. Recall that $[s, e]$ covers all the suffixes that have f as a prefix. Now Q2 can be answered in constant time simply by computing $\mathsf{rank}_1(B_{\mathsf{last}}, e) - \mathsf{rank}_1(B_{\mathsf{last}}, s - 1)$. For Q1, we first compute $r = \mathsf{rank}_1(B_{\mathsf{last}}, s - 1) + 1$, and then output the values $\mathsf{SA}[i]$ for all $i = \mathsf{select}_1(B_{\mathsf{last}}, r')$ such that $r' \geq r$ and $i \leq e$. This requires in total $O(|\mathsf{Q1}| \log \log n)$ time, where $|\mathsf{Q1}|$ is the number of reads having one or more occurrences of f. Finally, $n + o(n)$ bits are required to compute rank and select over B_{last} in constant time [14]. **Q.E.D.**

To answer the queries Q5–Q7, we employ yet another data structure:

Lemma 2.3 *Given the CSA of the string $T[1..n]$, a fixed constant k and a suffix array range $[s, e]$ covering all the suffixes that have f as a prefix, we can answer the queries Q5 and Q7 (resp. Q6) using $n + o(n)$ additional*

bits and $O(|\mathsf{Q5}| \log\log n)$ *time (resp.* $O(1)$ *time), where* $|\mathsf{Q5}| = |\mathsf{Q7}|$ *is the number of reads having exactly one occurrence of* f*.*

Proof. Let B_{once} denote a bit-vector of length $\mathsf{rank}_1(B_{\mathsf{last}}, n)$. We set $B_{\mathsf{once}}[i] = 1$ if and only if the k-mer starting from text position $p = \mathsf{SA}[\mathsf{select}_1(B_{\mathsf{last}}, i)]$ occurs only once within the corresponding read. The following subsection describes how to construct the bit-vector B_{once}. Now, similar to previous lemma, the query Q6 can be answered in constant time by first computing $s' = \mathsf{rank}_1(B_{\mathsf{last}}, s-1) + 1$ and $e' = \mathsf{rank}_1(B_{\mathsf{last}}, e)$. Then the result for Q6 is given by $\mathsf{rank}_1(B_{\mathsf{once}}, e') - \mathsf{rank}_1(B_{\mathsf{once}}, s'-1)$. For Q5 and Q7, we output the values $\mathsf{SA}[\mathsf{select}_1(B_{\mathsf{last}}, i')]$ for all $i' \in [s', e']$ such that $B_{\mathsf{once}}[i'] = 1$. Such positions can be found using one select_1 operation (over B_{once}) per outputted element. Thus, the query is solved in $O(|\mathsf{Q5}| \log\log n)$ time using $\mathsf{rank}_1(B_{\mathsf{last}}, n)(1 + o(1)) \leq n + o(n)$ bits. **Q.E.D.**

Theorem 2.4 *Given a set of reads* $\mathcal{R} = \{r_1, r_2, \ldots, r_d\}$ *of total length* n*, and a fixed constant* k*, there exists a data structure that requires* $nH_h \log\log_\sigma n + O(n)$ *bits of space and supports the queries Q1–Q7 with the time complexities given in Table 1. If the query is a* k*-mer (resp. a position in* $\mathcal{R}$*), the queries require additional* $O(k \log\sigma + \mathrm{polylog}(n))$ *time (resp.* $O(\log\log n)$ *time).*

Proof. See the above lemmas about supporting each query Q1–Q7. The combined space complexity of the required bit-vectors and their rank and select data structures is $3n + o(n)$ bits. The space complexity is dominated by the compressed suffix array, which requires $nH_h \log\log_\sigma n + O(n)$ bits of space [9]. **Q.E.D.**

2.1 Construction

We propose a construction algorithm that can build the above data structures in $O(n \log n)$ time and $O(n(H_0 + 1))$ bits of space, assuming that the largest read-length in the collection is limited, that is, $\ell = \max\{|r_i| : r_i \in \mathcal{R}\} = O(n/\log n)$. The theoretical complexities can be achieved by (1) building the CSA, (2) building the LCP array, (3) scanning through the LCP array once to construct B_{lcp}, and finally (4) scanning through the (implicit) suffix array once to construct B_{last} and B_{once}. In practice, B_{lcp} is constructed directly (see Sect. 3).

More precisely, the compressed suffix array can be constructed $O(n \log n)$ time using $O(n(H_0 + 1))$ bits [11]. The final index requires $nH_h \log\log_\sigma n +$

$O(n)$ bits and supports random access to the $\mathsf{SA}[i]$ and $\mathsf{SA}^{-1}[j]$ values in $t_{\mathsf{SA}} = O(\log \log n)$ time for polylog-sized alphabets [9]. The LCP array can then be constructed in $O(n \cdot t_{\mathsf{SA}})$ time and in $4n + o(n)$ bits of space on top of the CSA [12]. The resulting LCP array admits access to values $\mathsf{LCP}[i]$ in $O(t_{\mathsf{SA}})$ time, thus, we can also construct B_{lcp} in $O(n \cdot t_{\mathsf{SA}})$ time. The bit-vectors B_{last} and B_{once} can be attained as follows:

Lemma 2.5 *Given the CSA of the string $T[1..n]$, a fixed constant k and the bit-vector B_{lcp}, we can construct the bit-vectors B_{last} and B_{once} in $O(n \log \log n)$ time and $2n + o(n) + O(\ell \log n)$ additional bits. If the largest read-length is $\ell = O(n/\log n)$, the additional space is $O(\ell \log n) = O(n)$ bits.*

Proof. Let $B_{\mathsf{last}}[1, n]$ and $B'_{\mathsf{once}}[1, n]$ denote two bit-vectors. We initialize B_{last} to all zeros, and B'_{once} to all ones. We traverse over the suffixes of T in backwards order, say $T[n-1..n], T[n-2..n], \ldots, T[1..n]$. At each step i of the traversal, we first compute $j = \mathsf{SA}^{-1}[i]$ (in practice, we replace SA^{-1} with LF-mapping; see Sect. 3). Then we compute $r = \mathsf{rank}_1(B_{\mathsf{lcp}}, j)$ and check if the key r exists in a *y-fast trie* [25]. If it does not yet exists, we set $B_{\mathsf{last}}[j] = 1$, and insert the key-value pair $\langle r, j \rangle$ into the y-fast trie. Moreover, if the key r already exists in the y-fast trie with value j', we set $B'_{\mathsf{once}}[j'] = 0$. Finally, if $T[i]$ is a separator-symbol, we remove all elements in the current y-fast trie, thus, the maximum number of elements in the trie is bounded by $O(\ell)$. Since we traverse T in backwards order, we can easily keep track of the position of the nearest separator-symbol, and avoid marking B_{last} for k-mers that overlap a separator-symbol. All this requires $O(t_{\mathsf{SA}} + \log \log n)$ time per each step (with the exception of the removal of all trie elements, which can be amortized to $O(n \log \log n)$ time over all steps). The trie size is at most $O(\ell \log n)$ bits at any step of the construction. After traversing the whole text, we can construct the final bit-vector B_{once} of length $\mathsf{rank}_1(B_{\mathsf{last}}, n)$. We set $B_{\mathsf{once}}[\mathsf{rank}_1(B_{\mathsf{last}}, j)] = B'_{\mathsf{once}}[j]$ for each j such that $B_{\mathsf{last}}[j] = 1$. **Q.E.D.**

Corollary 2.6 *Given a set of reads $\mathcal{R} = \{r_1, r_2, \ldots, r_d\}$ of total length n, and a fixed constant k, the data structure in Theorem 2.4 can be constructed in $O(n \log n)$ time and $O(n(H_0 + 1))$ bits of space.*

2.2 Query extensions

Read coverage profile. The *coverage profile* of a read r gives, for each position $i \in [1, |r| - k + 1]$ in the read r, the number of reads that share

the k-mer $r[i \,..\, i+k-1]$. The coverage profile can be utilized, for example, to discriminate between sequencing errors and SNVs/SNPs [20]. The read coverage profile can be efficiently computed for any $r \in \mathcal{R}$ by resorting to $|r|-k+1$ calls to Q2, which requires in total $O(|r| \log\log n)$ time. (In practice, we use LF-mapping and backward search, and answer Q2 at each step via constant time query over B_{lcp} and B_{last}. The resulting time complexity is $O(|r| \cdot t_{\mathsf{LF}})$.)

Queries over multiple k. We can support queries over multiple $k_1, k_2, \ldots, k_z$ by building separate bit-vectors for each k_i. Now, the final index consists of one CSA built for the input reads, and z sets of bitvectors requiring in total $3nz + o(nz)$ bits of space. For any $z = O(\log \sigma)$, the total index size becomes $O(n \log \sigma)$ bits, which is still less than the original Gk arrays require for one fixed k. For large z, another time–space tradeoff is to replace all the LCP bit-vectors with the full LCP array, which requires just $4n + o(n)$ bits [12], and resort to *Previous Smaller Value* and *Next Smaller Value* queries over the LCP table similar to [8]. PSV and NSV can solved in sublogarithmic time with $o(n)$ extra bits of space.

3 Experiments

We implement the compressed Gk arrays (CGkA) using the *FM-index* concept [7] and Huffman-shaped wavelet trees [19]. We use Heng Li's implementation of the BCR algorithm [15, 1] to construct the Burrows–Wheeler Transform (BWT) for the input reads. The resulting FM-index requires $nH_0(T) + o(n \log \sigma)$ bits of space and supports *LF-mapping* in average $t_{\mathsf{LF}} = O(H_0(T)) = O(\log \sigma)$ time. We build the B_{lcp} bit-vector directly from the wavelet tree in $O(n\sigma)$ time by adapting the algorithm of [2]. For Lemma 2.5, we use LF-mapping instead of explicitly computing $j = \mathsf{SA}^{-1}[i]$ for each step. It allows us to construct bit-vectors B_{last} and B_{once} simultaneously with the (inverse) suffix array samples, using one pass over the text. We store (inverse) suffix array samples for every s text positions, which allows an $t_{\mathsf{SA}} = O(s \cdot t_{\mathsf{LF}})$ time access to SA (SA^{-1}). We test the sampling rates $s \in \{2, 4, 8, 16, 32\}$.

We compare the compressed Gk arrays against a performant hash table Jellyfish 1.1.6 [17], a *Run-Length Compressed Suffix Array* (RLCSA[3] Jan.

[3] http://www.cs.helsinki.fi/group/suds/rlcsa/ The latest RLCSA package in-

2013 version) [24], and the original *Gk* arrays (GkA[4] version 1.0.1) [21]. GkA offer a native support for queries Q1–Q7, the RLCSA supports only queries Q3–Q4, and Jellyfish the counting query Q4. We run the RLCSA using sampling rates $s \in \{3, 4, 8, 16\}$ (the construction ran out of memory for $s = 2$) and *nibble encoded* bit-vectors, which are faster and, in our experiments, require only around 2% more space. We use block size 16 for the internal bit-vectors.

Remark 3.1 *Claude et al. [5] proposed a* compressed k-mer index *for indexing highly-repetitive biological sequences. However, their experimental results show that RLCSA is faster and uses less space for any $k \geq 6$ [5]. Also, the construction space of Claude et al. is about twice larger than RLCSA. We omit the compressed k-mer index of Claude et al. from our experiments for these reasons.*

The input reads are taken from a set of 151bp Illumina reads sequenced from an E. Coli strain MG1655[5]. We truncate the low-quality tails, using a Phred threshold of 10, and include only the full-length reads in the final set. This filtering leaves a total of 8.5 million reads. All experiments are ran using a single core of an Intel Xeon E5540 2.53GHz processor equipped with 32GB of main memory, Linux 3.2.0 (Ubuntu x86_64) and gcc 4.6.3. We report the final index size, average query times for Q1–Q4, and the construction time and space for each data structure.

The final index size of CGkA represents 10% to 60% of the size of the original *Gk* arrays depending on the sampling rate. CGkA require 5.6 GB for $s = 2$ and 1.3 GB for $s = 32$, while the non-compressed GkA require between 9.0–9.2 GB depending on k. Jellyfish and RLCSA have the smallest index sizes at the cost of supporting only Q4 and Q3–Q4, respectively.

Remark 3.2 *The RLCSA implementation could be extended to support Q1–Q7 by adding the bit-vectors B_{lcp}, B_{last} and B_{once} over the RLCSA. It would then give yet another time–space trade-off for the compressed Gk arrays: a smaller index size, but slightly slower query times as the results in Fig. 2 suggest.*

cludes an unpublished data structure to solve Q1, however, its construction time and space do not yet scale up gracefully (J. Sirén, Personal communication, 2013).

[4] http://crac.gforge.inria.fr/gkarrays/

[5] http://www.illumina.com/systems/miseq/scientific_data.ilmn

Figure 2: Average query times and the index size when the query is given as a k-mer. RLCSA supports queries Q3 and Q4, and Jellyfish only a counting query (Q4).

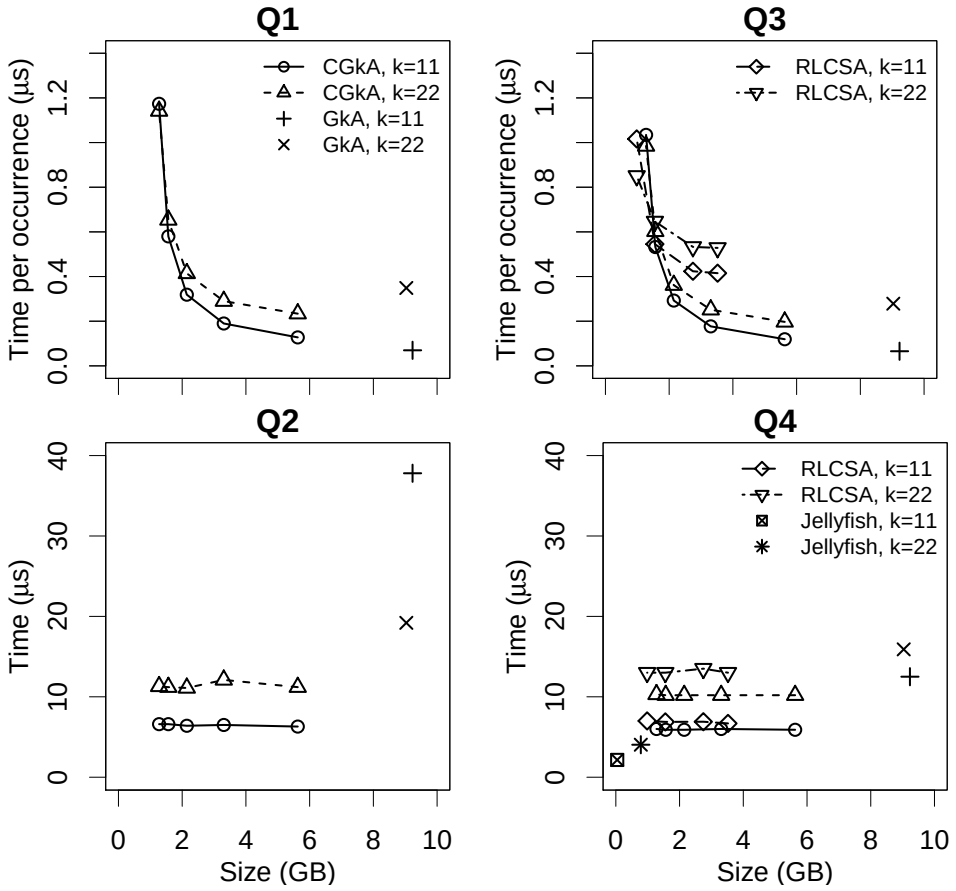

Figure 3: Average query times and the index size when the query is given as a read position. RLCSA and Jellyfish do not support these types of queries.

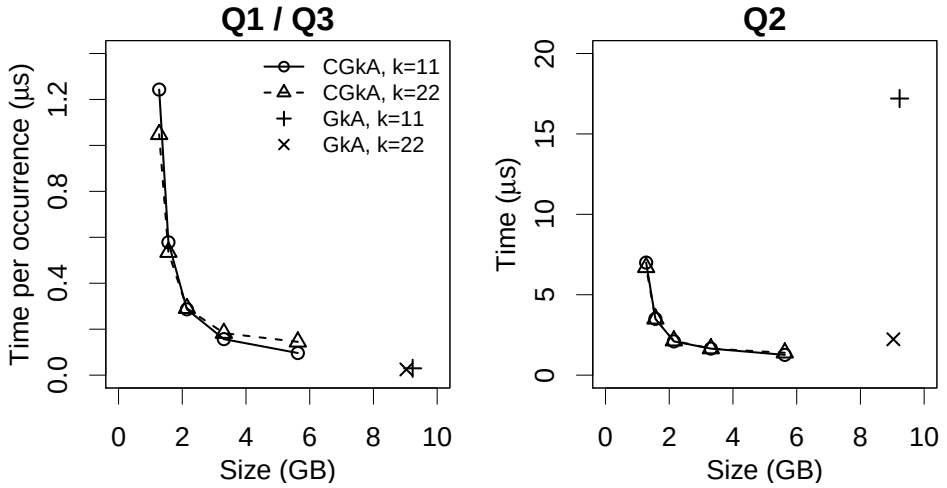

Fig. 2 gives an overview of the average query times for Q1–Q4, when querying a set of 1–100 million randomly chosen k-mers (depending on size of the k-mer). Jellyfish is the most space-efficient and also the fastest, since its hash table is tailored for simple counting queries (Q4). The compressed data structures are still competitive regarding both query time and space, while providing a more versatile set of queries. The differences for Q2 are more significant with $k = 11$ due to the $O(|\mathsf{Q3}|)$ worst-case time of GkA. Regarding the locate queries (Q1 and Q3), the sampling rates $s \leq 8$ are competitive against the non-compressed GkA for $k = 22$. This is mostly due to small numbers of occurrences (i.e. large k) and faster backward search. For smaller k, the numbers of occurrences are significantly higher, and the time to locate the suffix array interval has a smaller impact on the average query times. Fig. 3 gives the query times for Q1–Q3, when the query is given as a randomly chosen position from the indexed read set. The query times are averaged over 1–100 million randomly chosen positions (depending on query). The compressed representation is slower for all queries but Q2, mostly because the (inverse) suffix array values must be computed via the sampled array. CGkA have $O(t_{\mathsf{SA}}) = O(s \cdot t_{\mathsf{LF}})$ time access to inverse SA, which is notably slower than the constant time access within GkA. However, for Q2 the compressed representation can be faster due to the worst-case $O(|\mathsf{Q3}|)$ query time of GkA. RLCSA and Jellyfish do not support these types of queries.

Table 2: Construction time and space for 8.5 million 151bp Illumina reads.

	k	Time (s)	Memory usage (MB)
Gk arrays [21]	11	611	9,452
Gk arrays	22	605	9,251
RLCSA [24] ($s = 16$)	n/a	1,095	16,446
Coral [23] (GNU C++ hash_map)	11–22	861	16,016
Jellyfish [17] (counting only, $M = 2^{24}$)	11	88	2,911
Jellyfish [17] (counting only, $M = 2^{24}$)	22	405	2,965
Compressed Gk arrays ($s = 16$)	11	957	2,881
Compressed Gk arrays ($s = 16$)	22	1,086	2,881
CGkA construction steps:			
ropebwt+BCR [15, 1]	n/a	288	506
Wavelet tree (Huffman)	n/a	44	1,471
B_{lcp}	11	10	1,471
B_{lcp}	22	139	1,471
B_{last}, B_{once}, sampling SA, SA^{-1}	n/a	615	2,881

Construction time and space. We also measure the construction time and space for all data structures. For RLCSA, we use the fastest construction method in the Jan. 2013 package [24]. As a second hash table approach, we include Coral version 1.4 [23], which use the GNU C++ hash_map for storing a list of occurrence positions for each k-mer. Table 2 reports the construction times and maximum memory usages for the 8.5 million 151bp Illumina reads, including the figures for each construction step of the compressed Gk arrays.

Here, CGkA take roughly the same construction time as RLCSA, but use less memory. CGkA require only twice the construction time of non-compressed data structures, and achieves the most space-efficient construction (Jellyfish could use less memory by balancing between the hash table size and merge cost). Hence, the CGkA achieve a much better scalability in term of memory requirements than its uncompressed version, while still offering competitive query times.

4 Discussion

We presented a space-efficient data structure for indexing all k-mers in HTS data. The data structure supports a comprehensive set of locate and count queries with competitive query times. It is also more scalable than its non-compressed predecessor, the *Gk* arrays [21], due to the time–space trade-off we achieve: for a fixed amount of main memory, the compressed representation can index up to seven times more data (regarding the final index size). Both the construction and query algorithms are completely different from those of *Gk* arrays. It also allows queries over multiple $k_1, k_2, \ldots, k_z$ with an overhead of 3.19 bits per input character per k_i. A parallized, secondary memory [1] construction, as well as enhancements to allow navigation in a de Bruijn graph belong to future work.

References

[1] M. J. Bauer, A. J. Cox, and G. Rosone. Lightweight BWT construction for very large string collections. In *Proc. 22nd Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 6661, pages 219–231. Springer, 2011. 10, 14, 15

[2] T. Beller, S. Gog, E. Ohlebusch, and T. Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. In *18th Intl. Symp. String Processing and Information Retrieval*, LNCS 7024, pages 197–208, 2011. 10

[3] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram Based Database Searching Using a Suffix Array (QUASAR). In *3rd Int. Conf. on Computational Molecular Biology*, pages 77–83. ACM Press, 1999. 3

[4] R. Chikhi and D. Lavenier. Localized genome assembly from reads to scaffolds: Practical traversal of the paired string graph. In *WABI*, pages 39–48, 2011. 3

[5] F. Claude, A. Fariña, M. A. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. 10th IEEE Intl. Conf. on Bioinformatics and Bioengineering*, pages 86–91, 2010. 11

[6] T.C. Conway and A.J. Bromage. Succinct Data Structures for Assembling Large Genomes. *Bioinformatics*, 27(4):479–486, 2011. 4

[7] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398. IEEE Computer Society, 2000. 2, 10

[8] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009. 10

[9] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *14th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 841–850, 2003. 5, 6, 8, 9

[10] S. Hazelhurst and Z. Lipták. Kaboom! a new suffix array based algorithm for clustering expression data. *Bioinformatics*, 27(24):3348–3355, 2011. 4

[11] W.-K. Hon, T.-W. Lam, K. Sadakane, W.-K. Sung, and S.-M. Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, June 2007. 8

[12] W.-K. Hon and K. Sadakane. Space-economical algorithms for finding maximal unique matches. In *CPM*, LNCS 2373, pages 144–152, 2002. 9, 10

[13] W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *FOCS*, pages 713–722. IEEE Computer Society, 2009. 5

[14] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie–Mellon, 1989. 7

[15] H. Li. Implementation of BCR. https://github.com/lh3/ropebwt. 10, 14

[16] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009. 2

[17] G. Marçais and C. Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of *k*-mers. *Bioinformatics*, 27(6):764–770, 2011. 3, 10, 14

[18] P. Melsted and J. Pritchard. Efficient counting of k-mers in dna sequences using a bloom filter. *BMC Bioinformatics*, 12(1):333, 2011. 3

[19] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007. 5, 10

[20] N. Philippe, M. Salson, T. Commes, and E. Rivals. CRAC: an integrated approach to read analysis. *Genome Biology*, 2013. In press. 2, 3, 10

[21] N. Philippe, M. Salson, T. Lecroq, M. Léonard, T. Commes, and E. Rivals. Querying large read collections in main memory: a versatile data structure. *BMC Bioinformatics*, 12:242+, 2011. 3, 4, 5, 11, 14, 15

[22] G. Rizk, D. Lavenier, and R. Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, page Advance access, January 2013. 3

[23] L. Salmela and J. Schröder. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461, 2011. 3, 14

[24] J. Sirén. *Compressed Full-Text Indexes for Highly Repetitive Collections*. PhD thesis, Dept. of Computer Science, Report A-2012-5, University of Helsinki, 2012. 11, 14

[25] D. E. Willard. Log-logarithmic worst-case range queries are possible in space Theta(N). *Inf. Process. Lett.*, 17(2):81–84, 1983. 9