Indexing Finite Language Representation of Population Genotypes

Jouni Sirén*, Niko Välimäki **, and Veli Mäkinen

Helsinki Institute for Information Technology (HIIT) &
Department of Computer Science, University of Helsinki, Finland
{jltsiren,nvalimak,vmakinen}@cs.helsinki.fi

Abstract. We propose a way to index population genotype information together with the complete genome sequence, so that one can use the index to efficiently align a given sequence to the genome with all plausible genotype recombinations taken into account. This is achieved through converting a multiple alignment of individual genomes into a finite automaton recognizing all strings that can be read from the alignment by switching the sequence at any time. The finite automaton is indexed with an extension of Burrows-Wheeler transform to allow pattern search inside the plausible recombinant sequences. The size of the index stays limited, because of the high similarity of individual genomes. The index finds applications in variation calling and in primer design.

1 Introduction

Due to the advances in DNA sequencing [17], it is now possible to have complete genomes of individuals sequenced and assembled. Already several human genomes have been sequenced [22, 9, 11, 23, 14] and it is almost a routine task to resequence individuals by aligning the high-throughput short DNA reads to the reference [7]. This rich and focused genotype information, together with the more global genotype information (common single nucleotide polymorphisms (SNPs) and other variations) created using earlier techniques, can be combined to do different population-wide studies, now first time directly on whole genome level.

We propose a novel index structure that simultaneously represents and extrapolates the genotype information present in the population samples. The index structure is built on a given multiple alignment of individual genomes, or alternatively for a single reference sequence and set of SNPs of interest. The index structure is capable of aligning a given pattern to any path taken along the multiple alignment, as illustrated in Figure 1.

To build the index, we first create a finite automaton recognizing all paths through the multiple alignment, and then generalize Burrows-Wheeler transform

* Funded by the Finnish Doctoral Programme in Computational Sciences, Academy of Finland project 1140727 and the Nokia Foundation.

** Funded by Helsinki Graduate School in Computer Science and Engineering and Finnish Centre of Excellence for Algorithmic Data Analysis Research

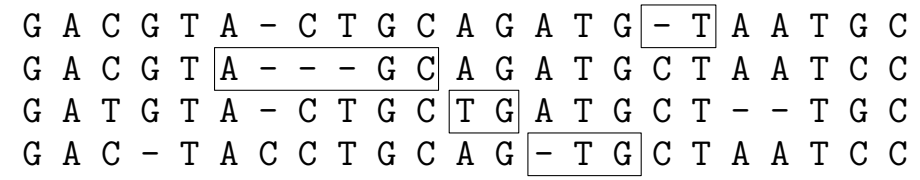

Fig. 1. Pattern `AGCTGTGT` matching the multiple alignment when allowing it to change row when necessary.

(BWT) [2] -based self-index structures [19] to index paths in labeled graphs. The backward search routine of BWT-based indexes generalizes to support exact pattern search over the labeled graph in $O(m)$ time, for pattern of length m. On general labeled graphs, such index can take exponential space, but on graphs resulting from finite automaton representation of multiple alignment of individual genomes, the space is expected to stay limited.

Applications for our index include the following:

- *SNP calling.* We can take the known SNPs into account already in the short read alignment phase, instead of the common pipeline of alignment, variation calling, and filtering of known SNPs. This allows more accurate alignment, as the known SNPs are no longer counted as errors, and the matches can represent novel recombinants not yet represented in the database.
- *Probe/primer design.* When designing probes for microarrays or primers for PCR, it is important that the designed sequence does not occur even approximately elsewhere than in the target. Our index can provide approximate search not only against all substrings, but also against plausible recombinants, and hence the design can be made more selective.
- *Large indel calling.* After short read alignment, the common approach for detecting larger indels is to study the uncovered regions in the reference genome. We can model deletions with our index by adding an edge to the automaton skipping each plausible deleted area. For insertions, we can apply *de novo* sequence assembly with the unmapped reads to generate plausible insertions, and add these as paths to the automaton.
- *Reassembly of donor genome.* Continuing from variation calling, one can use the realignment of the reads to the refined automaton to give a probability for each edge. It is then easy to extract, for example, the most probable path through the automaton as a consensus for the donor.

We made some feasibility experiments on SNP calling problem. We created our index on a multiple alignment of four instances of the 76 Mbp human chromosome 18. The total size of the index was about 67 MB. Aligning a set of 10 million Illumina Solexa reads of length 56 took 18 minutes, and about 1.1% of exact matches were novel recombinants not found when indexing each chromosome instance separately (see Sect. 5). Leaving these exact matches out from variation calling reduces the number of novel SNPs from 4203 to 1074.

Related work and extensions. Jumping alignments of protein sequences were studied in [21] as an alternative to profile Hidden Markov Models. They showed how to do local alignment across a multiple alignment of protein family so that jumping from one sequence to another is associated with a penalty. We study the same problem but from a different angle; we provide a compressed representation of the multiple alignment with an efficient way to support pattern matching.

Calling of large indels similar to our approach was studied in [1]. One difference is that they manage to associate probabilities to the putative genotypes, resulting into a more reliable calling of likely variants. However, their indexing part is tailored for this specific problem, whereas we develop a more systematic approach that can be generalized to many directions. For example, we can take the probabilities into account and index only paths with high enough probabilities, to closely simulate their approach (see Sect. 6).

Our work builds on the self-indexing scenario [19], and more specifically is an extension of the *XBW transform* [5] that is an index structure for labeled trees. Our extension to *labeled graphs* may be of independent interest, as it has potentially many more applications inside and outside computational biology.

The focus of this paper is the finite automaton representation of a multiple alignment. This setting is closely related to our previous work on indexing highly repetitive sequence collections [15]. In our previous work, we represented a collection of individual genomes of total length N, with reference sequence of length n, and a total of s mutations, in space $O(n \log \frac{N}{n} + s \log^2 N)$ bits in the average case (rough upper bounds here for simplicity). Exact pattern matching was supported in $O(m \log N)$ time. The index proposed in this paper achieves $O(n(1 + s/n)^{O(\log n)})$ bits in the expected case for constant-sized alphabets.

The paper is organized as follows. Section 2 introduces the notation, Sect. 3 reviews the necessary index structures we build on, Sect. 4 describes the new extension to finite languages, Sect. 5 gives some preliminary experiments on SNP calling problem, and Sect. 6 concludes the work by sketching the steps required for making the index into a fully applicable tool.

2 Definitions

A *string* $S = S[1, n]$ is a *sequence* of *characters* from *alphabet* $\Sigma = \{1, 2, \ldots, \sigma\}$. A *substring* of S is written as $S[i, j]$. A substring of type $S[1, j]$ is called a *prefix*, while a substring of type $S[i, n]$ is called a *suffix*. A *text* string $T = T[1, n]$ is a string terminated by $T[n] = \$ \notin \Sigma$ with lexicographic value 0. The *lexicographic order* "$<$" among strings is defined in the usual way.

A *graph* $G = (V, E)$ consists of a set $V = \{v_1, \ldots, v_{|V|}\}$ of *nodes* and a set $E \subset V^2$ of *edges* such that $(v, v) \notin E$ for all $v \in V$. We call $(u, v) \in E$ an edge from node u to node v. A graph is *directed*, if edge (u, v) is distinct from edge (v, u). In this paper, the graphs are always directed. For every node $v \in V$, the *indegree* $in(v)$ is the number of incoming edges (u, v), and the *outdegree* $out(v)$ the number of outgoing edges (v, w).

Graph G is said to be *labeled*, if we attach a *label* $\ell(v)$ to each node $v \in V$. A *path* $P = u_1 \cdots u_{|P|}$ is a sequence of nodes from u_1 to $u_{|P|}$ such that $(u_i, u_{i+1}) \in E$ for all $i < |P|$. The label of path P is the string $\ell(P) = \ell(u_1) \cdots \ell(u_{|P|})$. A *cycle* is a path from a node to itself containing, at least one other node. If a graph contains no cycles, it is called *acyclic*.

A *finite automaton* is a directed labeled graph $A = (V, E)$.[1] The *initial node* v_1 is labeled with $\ell(v_1) = \#$ with lexicographic value $\sigma + 1$, while the *final node* $v_{|V|}$ is labeled with $\ell(v_{|V|}) = \$$. The rest of the nodes are labeled with characters from alphabet Σ. Every node is assumed to be on some path from v_1 to $v_{|V|}$.

The *language* $L(A)$ recognized by automaton A is the set of the labels of all paths from v_1 to $v_{|V|}$. If the language contains a finite number of strings, it is called *finite*. A language is finite if and only if the automaton is acyclic. Two automata are said to be *equivalent*, if they recognize the same language.

Automaton A is forward (reverse) *deterministic* if, for every node $v \in V$ and every character $c \in \Sigma \cup \{\#, \$\}$, there exists at most one node u such that $\ell(u) = c$ and $(v, u) \in E$ ($(u, v) \in E$). For any language recognized by some finite automaton, we can always construct an equivalent automaton that is forward (reverse) deterministic.

3 Compressed indexes

The *suffix array (SA)* of text $T[1, n]$ is an array of pointers $\mathsf{SA}[1, n]$ to the suffixes of T in lexicographic order. As an abstract data type, a suffix array is any data structure that supports the following operations efficiently: (a) *find* the SA range containing the suffixes prefixed by *pattern* P; (b) *locate* the suffix $\mathsf{SA}[i]$ in the text; and (c) *display* any substring of text T.

Compressed suffix arrays (CSA) [8, 6] support these operations. Their compression is based on the *Burrows-Wheeler transform (BWT)* [2], a permutation of the text related to the SA. The BWT of text T is a sequence $\mathsf{BWT}[1, n]$ such that $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$, if $\mathsf{SA}[i] > 1$, and $\mathsf{BWT}[i] = T[n] = \$$ otherwise.

BWT can be reversed by a permutation called *LF-mapping* [2, 6]. Let $C[1, \sigma]$ be an array such that $C[c]$ is the number of characters in $\{\$, 1, 2, \ldots, c-1\}$ occurring in the BWT. We also define $C[0] = C[\$] = 0$ and $C[\sigma+1] = n$. We define LF-mapping as $LF(i) = C[\mathsf{BWT}[i]] + rank_{\mathsf{BWT}[i]}(\mathsf{BWT}, i)$, where $rank_c(\mathsf{BWT}, i)$ is the number of occurrences of character c in prefix $\mathsf{BWT}[1, i]$.

The inverse of LF-mapping is $\Psi(i) = select_c(\mathsf{BWT}, i - C[c])$, where c is the highest value with $C[c] < i$, and $select_c(\mathsf{BWT}, j)$ is the position of the jth occurrence of character c in BWT [8]. By its definition, function Ψ is strictly increasing in the range $[C[c]+1, C[c+1]]$ for every $c \in \Sigma$. Note that $T[\mathsf{SA}[i]] = c$ and $\mathsf{BWT}[\Psi(i)] = c$ for $C[c] < i \leq C[c+1]$.

These functions form the backbone of CSAs. As $\mathsf{SA}[LF(i)] = \mathsf{SA}[i] - 1$ [6] and hence $\mathsf{SA}[\Psi(i)] = \mathsf{SA}[i] + 1$, we can use these functions to move the suffix array position backward and forward in the sequence. The functions can be efficiently

[1] Unlike the usual definition, we label nodes instead of edges.

implemented by adding some extra information to a compressed representation of the BWT. Standard techniques [19] for supporting SA functionality include using *backward searching* [6] for *find*, and sampling some suffix array values for *locate* and *display*.

XBW [5] is a generalization of the Burrows-Wheeler transform for labeled trees, where leaf nodes and internal nodes are labeled with different alphabets. Internal nodes of the tree are sorted lexicographically according to path labels from the node to the root. Sequence BWT is formed by concatenating the labels of the children of each internal node in lexicographic order according to the parent node. Every internal node v now corresponds to a substring $\mathsf{BWT}[sp_v, ep_v]$ containing the labels of its children. The first position sp_v of each such substring is marked with a 1-bit in bit vector F. Backward searching is used to support the analogue of *find*. Tree navigation is possible by using BWT and F.

4 Burrows-Wheeler transform for finite languages

In this section, we generalize the XBW approach to finite automata. We call it the *generalized compressed suffix array (GCSA)*. For the GCSA to function, we require that the automaton is prefix-sorted. Refer to Section 4.4 on how to transform an automaton into an equivalent prefix-sorted automaton.

Definition 1. *Let A be a finite automaton, and let $v \in V$ be a node. Node v is* prefix-sorted *by prefix $p(v)$, if the labels of all paths from v to $v_{|V|}$ share a common prefix $p(v)$, and no path from any other node $u \neq v$ to $v_{|V|}$ has $p(v)$ as a prefix of its label. Automaton A is prefix-sorted, if all nodes are prefix-sorted.*

Every node of a prefix-sorted automaton A corresponds to a lexicographic range of suffixes of language $L(A)$.

In XBW, bit vector F is used to mark both nodes and edges. If node v has lexicographic rank i, the labels of its predecessors are $\mathsf{BWT}[sp_v, ep_v] = \mathsf{BWT}[select_1(F, i), select_1(F, i+1)-1]$. On the other hand, if node u is a child of node v, and $\mathsf{BWT}[j]$ contains the label of node u, then $LF(j)$ is the lexicographic rank of the label of the path from node u through node v to the root. Hence $select_1(F, LF(j))$ gives us the position of edge (u, v).

While the latter functionality is trivial in trees, a node can have many outgoing edges in a finite automaton. Hence we will use another bit vector M to mark the outgoing edges.

Let $A = (V, E)$ be a prefix-sorted automaton. To build GCSA, we sort the nodes $v \in V$ according to prefixes $p(v)$. For every node $v \in V$, sequence BWT and bit vectors F and M contain range $[sp_v, ep_v]$ of length $n(v) = \max(in(v), out(v))$. See Figure 4 and Table 1 for an example.

- $\mathsf{BWT}[sp_v, ep_v]$ contains the labels $\ell(u)$ for all incoming edges $(u, v) \in E$, followed by $n(v) - in(v)$ empty characters.
- $F[sp_v] = 1$ and $F[i] = 0$ for $sp_v < i \leq ep_v$.
- $M[sp_v, ep_v]$ contains $out(v)$ 1-bits followed by $n(v) - out(v)$ 0-bits. For the final node $v_{|V|}$, the range contains one 1-bit followed by 0-bits.

```
function LF([sp_v, ep_v], c):
1   i ← C[c] + rank_c(BWT, ep_v)
2   if select_c(BWT, i − C[c]) < sp_v:
3     return ∅
4   i ← select_1(M, i)
5   sp_u ← select_1(F, rank_1(F, i))
6   ep_u ← select_1(F, rank_1(F, i) + 1) − 1
7   return [sp_u, ep_u]

function ℓ([sp_v, ep_v]):
8   return char(rank_1(M, sp_v))

function Ψ([sp_u, ep_u]):
9   c ← ℓ([sp_u, ep_u])
10  res ← ∅
11  low ← rank_1(M, sp_u)
12  high ← rank_1(M, ep_u)
13  for i ← low to high:
14    j ← select_c(BWT, i − C[c])
15    sp_v ← select_1(F, rank_1(F, j))
16    ep_v ← select_1(F, rank_1(F, j) + 1) − 1
17    res ← res ∪ {[sp_v, ep_v]}
18  return res
```

Fig. 2. Pseudocode for the basic navigation functions LF, Ψ, and ℓ.

Array C is used with some modifications. We define $C[\sigma + 1] = C[\#]$ in the same way as for regular characters, while $C[\sigma + 2]$ is set to be $|E|$. Assuming that each edge $(u, v) \in E$ has an implicit label $\ell(u)p(v)$, we can interpret $C[c]$ as the number of edges with labels smaller than c. We write $char(i)$ to denote character c such that $C[c] < i \leq C[c + 1]$.

4.1 Basic navigation

Let $[sp_v, ep_v]$ be the range of BWT corresponding to node $v \in V$. We define the following functions:

- $LF([sp_v, ep_v], c) = [sp_u, ep_u]$, where $\ell(u) = c$ and $(u, v) \in E$, or $\emptyset$ if no such u exists.
- $\Psi([sp_u, ep_u]) = \{[sp_v, ep_v] \mid (u, v) \in E\}$.
- $\ell([sp_v, ep_v]) = \ell(v)$.

These are generalizations of the respective functions on BWT. LF can be used to move backwards on edge (u, v) such that $\ell(u) = c$, while Ψ lists the endpoints of all outgoing edges from node u. These functions can be implemented by using BWT, F, M, and C, as seen in Figure 2.

Line 1 of LF is similar to the regular LF, determining the rank of edge label $cp(v)$ among all edge labels. On lines 2 and 3, we determine if there is an occurrence of c in $\mathsf{BWT}[sp_v, ep_v]$. On line 4, we find the position of edge (u, v) in bit vector M, and on lines 5 and 6, we find the range $[sp_u, ep_u]$ containing this position.

4.2 Searching

As the generalized compressed suffix array is a CSA, most of the algorithms using a CSA can be modified to use GCSA instead. In this section, we describe how to support two of the basic SA operations (see Sect. 3):

– *find*(P) returns the range $[sp, ep]$ of BWT corresponding to those nodes v, where at least one path starting from v has pattern P as a prefix of its label.
– *locate*$([sp_v, ep_v])$ returns a numerical value corresponding to node v.

We use backward searching [6] to support *find*. The algorithm maintains an invariant that $[sp_i, ep_i]$ is the range returned by *find*$(P[i, m])$. In the initial step, we start with the edge range $[C[P[m]] + 1, C[P[m] + 1]]$, and convert it to range $[sp_m, ep_m]$ by using bit vectors M and F. The step from $[sp_{i+1}, ep_{i+1}]$ to $[sp_i, ep_i]$ is a generalization of function LF for a range of nodes. We find the first and last occurrences of character $P[i]$ in BWT$[sp_{i+1}, ep_{i+1}]$, map them to edge ranks by using C and BWT, and convert the ranks to sp_i and ep_i by using F and M.

For *locate*, we assume that there is a (not necessarily unique) numerical value $id(v)$ attached to each node $v \in V$. Examples of these values include node ids (so that $id(v_i) = i$) and positions in the multiple alignment. To avoid excessive sampling of node values, $id(v)$ should be $id(u) + 1$ whenever (u, v) is the only outgoing edge from u and the only incoming edge to v.

We sample $id(u)$, if there are multiple or no outgoing edges from node u, or if $id(v) \neq id(u) + 1$ for the only outgoing edge (u, v). We also sample one out of d node values, given sample rate $d > 0$, on paths of at least d nodes without any samples. The sampled values are stored in the same order as the nodes, and their positions are marked in bit vector B ($B[sp_u] = 1$, if node u is sampled).

Node values are retrieved in a similar way as in CSAs [19]. To retrieve $id(u)$, we first check if $B[sp_u] = 1$, and return sample $rank_1(B, sp_u)$, if this is the case. Otherwise we follow the only outgoing edge (u, v) by using function Ψ, and continue from node v. When we find a sampled node w, we return $id(w) - k$, where k is the number of steps taken by using Ψ.

4.3 Analysis

For each node $v \in V$, the length of range $[sp_v, ep_v]$ is the maximum of $in(v)$ and $out(v)$. As every node must have at least one incoming edge and one outgoing edge (except for the initial and the final nodes), the length of BWT is at most $2|E| - |V| + 2$.

Theorem 1. *Assume that rank and select on bit vectors require $O(t_B)$ time. GCSA with sample rate d supports* find*(P) in $O(|P| \cdot t_B)$ and* locate*$([sp_v, ep_v])$ in $O(d \cdot t_B)$ time.*

Proof. We use bit vectors Ψ_c that mark the occurrences of character $c \in \Sigma \cup \{\#\}$ to encode BWT. This reduces *rank* and *select* on BWT to the same operations on bit vectors. Basic operations ℓ and LF take $O(t_B)$ time, as they require a constant number of bit vector operations. Ψ also takes $O(t_B)$ time, if the current node has outdegree 1. As *find* does one generalized LF per character of pattern, it takes $O(|P| \cdot t_B)$ time.

Operation *locate* checks from bit vector B if the current position is sampled, and follows the unique outgoing edge using Ψ if not. This requires a constant number of bit vector operations per step. As a sample is found within $d-1$ steps, the time complexity is $O(d \cdot t_B)$. □

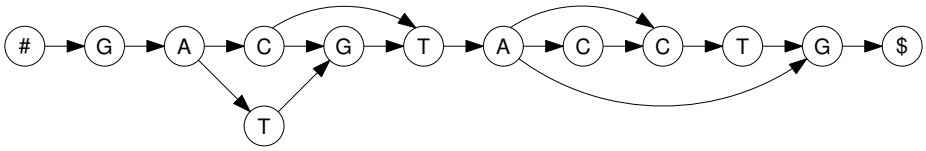

Fig. 3. A reverse deterministic automaton corresponding to the first 10 positions of the multiple alignment in Figure 1.

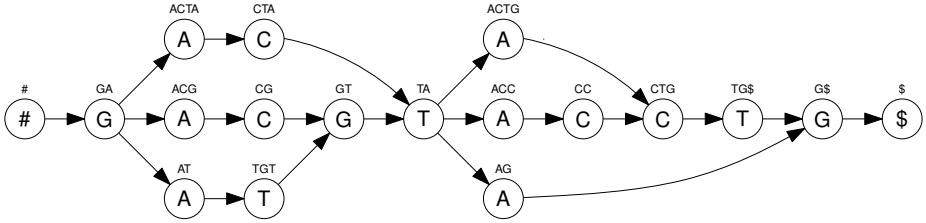

Fig. 4. A prefix-sorted automaton built for the automaton in Figure 3. The strings above nodes are prefixes $p(v)$.

Table 1. GCSA for the automaton in Figure 4. Nodes are identified by prefixes $p(v)$.

	$	ACC	ACG	ACTA	ACTG	AG	AT	CC	CG	CTA	CTG	G$	GA	GT	TA	TG$	TGT	#
BWT	G	T	G	G	T	T	G	A	A	A	AC	AT	#--	CT	CG-	C	A	$
F	1	1	1	1	1	1	1	1	1	1	10	10	100	10	100	1	1	1
M	1	1	1	1	1	1	1	1	1	1	10	10	111	10	111	1	1	1

4.4 Index construction

Our construction algorithm is related to the *prefix-doubling* approach to suffix array construction [20]. We start with a reverse deterministic automaton (see Figure 3), convert it to an equivalent prefix-sorted automaton (Figure 4), and build the GCSA (Table 1) for that automaton.

Definition 2. *Let A be a finite automaton recognizing a finite language, and let $k > 0$ be an integer. A is* k-sorted *if, for every node $v \in V$, the labels of all paths from v to $v_{|V|}$ share a common prefix $p(v, k)$ of length k, or if node v is prefix-sorted by prefix $p(v, k)$ of length at most k.*

Starting from a reverse deterministic automaton $A = A_0$, we build a series of automata $A_i = (V_i, E_i)$ for $i = 1, 2, \ldots$ that are 2^i-sorted, until we get an automaton that is prefix-sorted. The construction algorithm is described and analyzed in more detail in the full paper.[2]

[2] arXiv:1010.2656 [cs.DS]

Due to the lack of space, we just summarize the result of the analysis here. Assume that we have a random reference sequence of length n from an alphabet of size σ, with a random SNP at each position with probability $p < 0.25$. Then there are $f(n,p) = n(1+p)^{O(\log_\sigma n)} + O(1)$ nodes and edges in the expected case, and the algorithm uses $O(f(n,p) \log f(n,p) \log n)$ time and $O(f(n,p) \log f(n,p))$ bits of space. As we mostly use sorting, scanning and database joins, the algorithm can be implemented efficiently in parallel, distributed, and external memory settings.

5 Implementation and Experiments

We have implemented GCSA in C++, using the components from on our implementation of RLCSA [15].[3] For each character $c \in \Sigma \cup \{\#\}$, we use a gap encoded bit vector to mark the occurrences of c in BWT. Bit vectors F and M are run-length encoded. Bit vector B is gap encoded, while the samples are stored using $\lceil \log(id_{\max} + 1) \rceil$ bits each, where $id_{\max}$ is the largest sampled value.

The implementation was compiled on g++ version 4.3.3. Index construction was done on a system with 128 gigabytes of memory and four quad-core Intel Xeon X7350 processors running at 2.93 GHz, while the other experiments were done on another system with 32 gigabytes of memory and two quad-core Intel Xeon E5540 processors running at 2.53 GHz. We used only one core in all experiments. Both systems were running Ubuntu 10.04 with Linux kernel 2.6.32.

We built a multiple alignment for four different assemblies of the human chromosome 18 (about 76 Mbp each). Three of the assemblies were from NCBI[4]: the assemblies by the Genome Reference Consortium (*GRCh37*), Celera Genomics, and J. Craig Venter Institute. The fourth sequence[5] was from Beijing Genomics Institute. The sequences were aligned by the Mauve Multiple Genome Alignment software [3]. We ran progressiveMauve 2.3.1 assuming collinear genomes, as we do not support rearrangements. The multiple alignment took a few hours to build: about 89.4% of nucleotides aligned perfectly, 0.19% with one or more mismatches, and 10.4% were inside of a gap. The number of gaps was high mainly because of the differences in the centromere region.

We constructed a GCSA (sample rate 16) for the alignment with various context lengths m. For each base $S_j[i]$ of sequence S_j, we used the next m bases as a context. We created a node for each base of each sequence, with an edge to the next base in the sequence. The nodes for aligned bases $S_j[i]$ and $S_{j'}[i']$ were then merged, if the bases and their contexts were identical. We also built RLCSA (sample rate 32) and BWA 0.5.8a [12] for the four sequences.

We searched for exact matches of 10 million Illumina/Solexa reads of length 56, sequenced from the whole genome, as both regular patterns and reverse complements. Table 2 lists results of these experiments. GCSA was 2.5–4 times slower than RLCSA and 3.5–5 times slower than BWA. About 1% of the reads matched

[3] `http://www.cs.helsinki.fi/group/suds/gcsa/`
[4] `ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes/`
[5] `ftp://public.genomics.org.cn/BGI/yanhuang/fa/`

Table 2. Index construction and exact matching with GCSA (sample rate 16), RLCSA (sample rate 32), and BWA on a multiple alignment of four sequences of human chromosome 18. Times for *locate* include the time used by *find*. GCSA-m denotes GCSA with context length m.

		Construction		Matching		
Index	**Size**	**Time**	**Space**	**Matches**	**Find**	**Locate**
GCSA-2	71.8 MB	342 min	77 GB	389,292	16 min	25 min
GCSA-4	66.9 MB	295 min	29 GB	388,521	16 min	18 min
GCSA-8	64.5 MB	286 min	22 GB	387,807	16 min	17 min
RLCSA	165.0 MB	11 min	2.3 GB	384,400	6 min	7 min
BWA	212.4 MB	4 min	1.4 GB	384,400	-	5 min

by GCSA were not matched by the other indexes. Construction requirements for GCSA were high (but see Sect. 6 for discussion).

The performance gap between GCSA and RLCSA reflects differences in fundamental techniques, as the implementations share most of their basic components and design choices. Theoretically GCSA should be about 3 times slower, as it requires six bit vector operations per base in *find*, while RLCSA uses just two. The differences between RLCSA and BWA come from implementation choices, as RLCSA is intended for highly repetitive sequences and BWA for fast pattern matching with DNA sequences.

To test GCSA in a more realistic alignment algorithm, we implemented BWA-like approximate searching [12] for both GCSA and RLCSA. There are some differences to BWA: i) we return all best matches; ii) we do not use a seed sequence; iii) we have no limits on gaps; and iv) we have to match $O(|P| \log |P|)$ instead of $O(|P|)$ characters to build the lower bound array $D(\cdot)$ for pattern P, as we have not indexed the reverse sequence. We used context length 4 for GCSA, as it had the best trade-off in exact matching.

The results can be seen in Table 3. GCSA was about 2.5 times slower than RLCSA, while finding from 1.1% (exact matching) to 2.5% (edit distance 3) more matches in addition to those found by RLCSA. BWA is significantly faster (e.g. finding 1,109,668 matches with $k = 3$ in 40 minutes), as it solves a slighly different problem, ignoring a large part of the search space with biologically implausible edit operations. A fair comparison with BWA is currently impossible without significant amount of reverse engineering. With the same algorithm in all three indexes, the performance differences should be similar as in exact matching.

Finally, we made a preliminary experiment on the SNP calling application mentioned in Sect. 1 using in-house software. We called for SNPs from chromosome 18 with minimum coverage 2, using all 10 million reads, as well as only those reads with *no* exact matches on GCSA-4. The number of called SNPs was 4203 with all reads and 1074 with non-matching ones. We did not yet compare how much of the reduction can be explained by exact matches on recombinants that would also be found using approximate search on one reference and how much by more accurate alignment due to richer reference set.

Table 3. Approximate matching with GCSA and RLCSA. The reported matches for given edit distance k include those found with smaller edit distances.

	GCSA-4		RLCSA	
k	**Matches**	**Time**	**Matches**	**Time**
0	388,521	18 min	384,400	7 min
1	620,482	103 min	609,320	39 min
2	876,877	256 min	856,373	101 min
3	1,147,404	1,751 min	1,118,719	534 min

6 Discussion

Based on our experiments, GCSA is 2.5–4 times slower than a similar implementation of CSA used in the same algorithm. With typical mutation rates, the index is also not much larger than a CSA built just for the reference sequence. Hence GCSA does not require significantly more resources than a regular compressed suffix array, while providing biologically relevant extended functionality.

While our current construction algorithm uses much resources, recent developments have improved it significantly. The next implementation should be faster and use several times less memory than the current one. A parallel external memory implementation should allow us to build an index for the human genome and all known SNPs in a few days. Extrapolating from current results, the final index should be 2.5–3 gigabytes in size. We are also working on a different construction algorithm in the MapReduce framework [4].

To improve the running time of short read alignment and related tasks, most of the search space pruning mechanisms (in addition to the one mechanism we already used from [12]) to support approximate matching on top of BWT [10, 12, 13, 16] can be easily plugged in.

As mentioned in Section 1, an obvious generalization is to index labeled weighted graphs, where the weights correspond to probabilities for jumping from one sequence to another in the alignment. This does not increase space usage significantly, as the probabilities differ from 1.0 only in nodes with multiple outgoing edges. During the construction of the index, it is also easy to discard paths with small probabilities, given a threshold. This approach can be used e.g. to index recombinants only in the recombination hotspot areas [18].

The experiments conducted here aimed at demonstrating the feasibility and potential of the approach. Once we have the genome-scale implementation ready, we are able to test our claims on improving variant calling and primer design accuracy with the index. Both require wet-lab verification to see the true effect on reducing false positives.

Acknowledgements

We wish to thank Eric Rivals for pointing out recombination hotspots and Riku Katainen for running the variation calling experiment.

References

1. C.A. Albers et al. Dindel: Accurate indel calls from short-read data. *Genome Research*, October 2010.
2. M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.
3. A.E. Darling et al. progressiveMauve: Multiple Genome Alignment with Gene Gain, Loss and Rearrangement. *PLoS ONE*, 5(6):e11147, 06 2010.
4. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI 2004*, pages 137–150. USENIX Association, 2004.
5. P. Ferragina et al. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):article 4, 2009.
6. P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.
7. P. Flicek and E. Birney. Sense from sequence reads: methods for alignment and assembly. *Nature Methods*, 6:S6–S12, 2009.
8. R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.
9. E. S. Lander et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.
10. B. Langmead et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.
11. S. Levy et al. The diploid genome sequence of an individual human. *PLoS Biol.*, 5(10):e254, 2007.
12. H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 27(14):1754–60, 2009.
13. R. Li et al. SOAP2. *Bioinformatics*, 25(15):1966–1967, 2009.
14. R. Li et al. De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, 20(2):265–72, 2010.
15. V. Mäkinen et al. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.
16. V. Mäkinen et al. Unified view of backward backtracking in short read mapping. In *Algorithms and Applications*, volume 6060 of *LNCS*, pages 182–195. Springer, 2010.
17. M. L. Metzker. Sequencing technologies – the next generation. *Nature Reviews Genetics*, 11:31–46, 2010.
18. S. Myers et al. A fine-scale map of recombination rates and hotspots across the human genome. *Science*, 310(5746):321–324, 2005.
19. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.
20. S. J. Puglisi et al. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.
21. R. Spang et al. A novel approach to remote homology detection: Jumping alignments. *Journal of Computational Biology*, 9(5):747–760, 2002.
22. J. C. Venter et al. The sequence of the human genome. *Science*, 291(5507):1304–51, 2001.
23. D. A. Wheeler et al. The complete genome of an individual by massively parallel DNA sequencing. *Nature*, 452(7189):872–6, 2008.