Indexing Graphs for Path Queries with Applications in Genome Research

Jouni Sirén, Niko Välimäki, Veli Mäkinen

Abstract—We propose a generic approach to replace the canonical sequence representation of genomes with graph representations, and study several applications of such extensions. We extend the Burrows-Wheeler transform (BWT) of strings to acyclic directed labeled graphs, to support path queries as an extension to substring searching. We develop, apply, and tailor this technique to a) read alignment on an extended BWT index of a graph representing *pan-genome*, i.e. reference genome and known variants of it; and b) split-read alignment on an extended BWT index of a splicing graph. Other possible applications include probe/primer design, alignments to assembly graphs, and alignments to phylogenetic tree of partial-order graphs. We report several experiments on the feasibility and applicability of the approach. Especially on highly-polymorphic genome regions our pan-genome index is making a significant improvement in alignment accuracy.

Index Terms—Keywords: pan-genome indexing, graph indexing, read alignment, variation calling, extended Burrows-Wheeler transform

1 Introduction

Due to the advances in DNA sequencing [2], it is now possible to have complete genomes of individuals sequenced and assembled. It is almost a routine task to resequence individuals by aligning the high-throughput short DNA reads to the reference [3]. Combined with fragment assembly of different population groups, it is feasible to talk about human *pan-genome* [4], i.e. reference genome and its common variations appearing among the whole population.

We propose a novel index structure, the *generalized compressed suffix array (GCSA)*, to represent pan-genomic information. The index structure is built on a given multiple alignment of individual genomes, or alternatively for a single reference sequence and set of variations of interest. GCSA is capable of aligning a given pattern to any path taken along the multiple alignment, as illustrated in Fig. 1, or equivalently to any recombination of the variations among the population.

To build the index, we first create a finite automaton recognizing all paths through the multiple alignment, and then generalize Burrows-Wheeler transform (BWT) [5] -based self-index structures [6] 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 linear.

Applications for our index include the following:

- *Read alignment.* We can take the known variations 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 variations are no longer counted as errors, and the matches can represent novel recombinants not yet represented in the database.
- *Split-read alignment using splicing graphs.* In RNA-sequencing, reads should be mapped to the genome allowing an intron to splice it. By mapping all reads that align as a whole, one can predict possible splice-sites, and create a *splicing graph* [7] representing predicted exons as nodes and possible splice-junctions (e.g. all combinations of nearby exons) as edges. We can build our index on a splicing graph (turning nodes into non-branching paths of sequences they represent) and align the rest of the reads to the paths.
- *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.
- *Alignment to a phylogenetic tree of partial-order graphs.* Löytynoja, Vilella, and Goldman [8] propose to replace the profiles in progressive multiple alignment with partial-order sequence graphs, that encode the insertions and deletions as possibilities instead of fixing them like in the typical "once a gap, always a gap" approach. We can build our index on this tree of graphs, and align the reads efficiently.
- *Alignment to assembly graphs.* We can index *overlap graphs* and *de Bruijn graphs* to support *de novo* variant calling.

This is an extended version of our article in WABI 2011 [1].
J. Sirén is with University of Chile
N. Välimäki and V. Mäkinen are with University of Helsinki

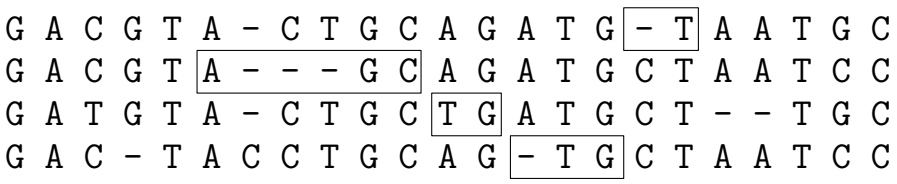

Fig. 1. Pattern AGCTGTGT matching the multiple alignment when allowing it to change row when necessary.

In this article, we focus on the read alignment and split-read alignment applications. Our experiments show that is is possible (yet challenging) to index human pan-genome with our technique and support single-end or paired-end read alignment with higher accuracy than competing tools, that either use more advanced search techniques or exploit pan-genomic information in a different manner. Especially on highly-polymorphic regions, that show strong enrichment of known, disease-causing mutations [9], our index is making a significant improvement in alignment accuracy. For split-read alignment, we show that realistic size splicing graphs can be indexed and moderate alignment accuracy can be obtained.

1.1 Related work

Our work builds on the self-indexing scenario [6], and more specifically is an extension of the *XBW transform* [10] that is an index structure for labeled trees. 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 [11]. 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.

A k-mer hashing based approach to pan-genome indexing [12] was proposed simultaneous to the first publication of our work. Similar idea was also adopted in [13] for larger indel calling, but replacing the k-mer indexing of [12] with BWT-index built for the extracted context around putative indels and their nearby combinations. Recently, this context extraction idea was also proposed for pan-genome indexing [14]. Our approach is more generic than the k-mer indexing and context extracting approaches above in that the read length is not fixed.

While we only consider indexing acyclic graphs in this work, our techniques can be adapted to index several classes of cyclic graphs. In a recent work [15], techniques similar to ours were used to build BWT-based self-indexes for de Bruijn graphs.

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 discusses the computational complexity aspects related to indexing finite languages, Sect. 6 shows how to construct the new index given a multiple alignment, Sect. 7 reports experiments on construction and query performance, on read alignment, and on split-read alignment, and Sect. 8 concludes the work by discussing future directions on using the techniques in other applications.

2 Definitions

Let $\Sigma = \{1, \ldots, \sigma\}$ be an *alphabet* of size σ. A *string* $S = S[1, n]$ over alphabet Σ is a *sequence* of *characters* $S[1] \cdots S[n]$, where $S[i] \in \Sigma$ for all i. For any string $S[1, n]$, we write $|S| = n$ to denote its length. A *substring* of S is written as $S[i, j] = S[i] \cdots S[j]$. Important types of substrings include *prefixes* $S[1, j]$ and *suffixes* $S[i, n]$. The concatenation of two strings $S[1, n]$ and $S'[1, n']$ is written as $SS' = S[1] \cdots S[n]S'[1] \cdots S'[n']$.

We define the *lexicographic order* $<$ among strings in the usual way. For any two strings $S[1, n]$ and $S'[1, n']$, we have $S < S'$, if either $S[1] < S'[1]$ or $S[1] = S'[1]$ and $S[2, n] < S'[2, n']$. The *empty string* λ of length 0 is a special case, with $\lambda < S$ for any non-empty string S. To avoid this special case, we often consider *text* strings $T = T[1, n]$ that are terminated by an end marker $T[n] = \$ \notin \Sigma$ with lexicographic value 0.

The (Levenshtein) *edit distance* between two strings S and S' is the minimum number of edit operations required to transform string S into string S'. Allowed edit operations include the *substitution* of one character with another, the *insertion* of one character into any position, and the *deletion* of one character. Any set of edit operations transforming string S into string S' can be represented as an *alignment* of the strings. This can be generalized for a set of strings, producing a *multiple alignment* of the strings (see Fig. 1).

A *graph* $G = (V, E)$ consists of a set $V = \{v_1, \ldots, v_{|V|}\}$ of *nodes* and a set $E \subseteq V^2$ of *edges*. 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). For every node $v \in V$, we define the *indegree* of the node $in(v)$ to be the number of incoming edges (u, v), and the *outdegree* $out(v)$ to be the number of outgoing edges (v, w).

In a *labeled* graph, we attach a *label* $\ell(v)$ to each node $v \in V$. A *path* $\mathsf{P} = u_1 \cdots u_{|\mathsf{P}|}$ is a sequence of nodes such that $(u_i, u_{i+1}) \in E$ for all $i < |\mathsf{P}|$. The label of path P is the string $\ell(\mathsf{P}) = \ell(u_1) \cdots \ell(u_{|\mathsf{P}|})$. A *cycle* is a path from a node to itself passing through 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 $\ell(v_1) = \#$

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

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. We assume that every node $v \in V$ is on some path from v_1 to $v_{|V|}$.

The *language* $L(A)$ *recognized* by automaton A is the set of labels of all paths from v_1 to $v_{|V|}$. We say that automaton A recognizes any string $S \in L(A)$, and that a suffix S' can be recognized from node v, if there is a path from v to $v_{|V|}$ with label S'. Note that all strings in the language are of form $\#x\$$, where x is a string over alphabet Σ. If the language contains a finite number of strings, it is called *finite*. A language is finite if and only if the automaton recognizing it 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)* [16] of text $T[1,n]$ is an array of pointers $\mathsf{SA}[1,n]$ to the suffixes of T in lexicographic order. It requires $n \log n$ bits of space in addition to the text, and can be constructed in $O(n)$ time with $2n$ bits of working space in addition to the text and the final index [17]. Given pattern P, we can find the range $\mathsf{SA}[sp, ep]$ containing the suffixes that have the pattern as their prefix in $O(|P| \log n)$ time by using binary search.

Definition 1. A data structure provides suffix array-like functionality, if it supports the following queries efficiently: (a) *find* the suffix array range $\mathsf{SA}[sp, ep]$ containing the suffixes prefixed by pattern P; (b) given i, *locate* suffix $\mathsf{SA}[i]$ in the text; and (c) given i and j, *extract* substring $T[i,j]$.

Burrows-Wheeler transform (BWT) [5] is a permutation of the text closely related to the suffix array. The BWT of text $T[1,n]$ 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. The transform can be reversed by a permutation called *LF-mapping* [5], [18]. Let $C[0, \sigma+1]$ be an array such that $C[c]$ is the number of characters in $\{\$, 1, 2, \ldots, c-1\}$ occurring in the BWT, with $C[0] = C[\$] = 0$ and $C[\sigma+1] = n$. We define LF-mapping as $LF(i) = C[\mathsf{BWT}[i]] + \mathrm{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}, i)$, where $\mathrm{rank}_c(\mathsf{BWT}, i)$ is the number of occurrences of character c in prefix $\mathsf{BWT}[1,i]$.

The $\mathrm{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}, i)$ in the definition can be interpreted as the lexicographic rank of suffix $T[\mathsf{SA}[i], n]$ among the suffixes preceded by character $\mathsf{BWT}[i]$. Hence $LF(i)$ is the lexicographic rank of suffix $T[\mathsf{SA}[i]-1, n]$ (or $T[n]$, if $\mathsf{SA}[i] = 1$) among all suffixes of the text. This allows us to move from the suffix array position corresponding to suffix $T[\mathsf{SA}[i], n]$ to that of suffix $T[\mathsf{SA}[i]-1, n]$ without using the text or its suffix array.

By using LF-mapping, we can support *find* with just arrays C and BWT through *backward searching* [18] (see Fig. 2). When searching for pattern P, the algorithm maintains an invariant that $\mathsf{SA}[sp_i, ep_i]$ is the range of suffixes prefixed by $P[i, |P|]$. If $\mathsf{BWT}[j]$ and $\mathsf{BWT}[j']$ are the first and the last occurrences of character $P[i-1]$ in range $\mathsf{BWT}[sp_i, ep_i]$, then $\mathsf{SA}[LF(j), LF(j')]$ is the range of suffixes prefixed by $P[i-1, |P|]$.

```
function find(P)
    [sp, ep] ← [C[P[|P|] + 1, C[P[|P|] + 1]]
    for i ← |P| − 1 to 1 do
        sp ← C[P[i]] + rank_{P[i]}(BWT, sp − 1) + 1
        ep ← C[P[i]] + rank_{P[i]}(BWT, ep)
        if [sp, ep] = ∅ then
            return ∅
    return [sp, ep]
```

Fig. 2. Backward searching on Burrows-Wheeler transform [18]. The body of the loop constitutes one step of backward searching.

The inverse function of LF-mapping is Ψ. We compute $\Psi(i) = \mathrm{select}_c(\mathsf{BWT}, i - C[c])$, where c is the highest value with $C[c] < i$, and $\mathrm{select}_c(\mathsf{BWT}, j)$ is the position of the jth occurrence of character c in BWT [19]. We often write $\mathrm{char}(i)$ to denote such character c. This function allows us to move from the suffix array position of suffix $T[\mathsf{SA}[i], n]$ to that of suffix $T[\mathsf{SA}[i]+1, n]$. Function Ψ is strictly increasing in the range $C_c = [C[c]+1, C[c+1]]$ corresponding to suffixes starting with character $c \in \Sigma$.

Compressed suffix arrays (CSA) [19], [18] are compressed data structures that provide suffix array-like functionality (see Definition 1). They combine a compressed representation of the Burrows-Wheeler transform with some extra information that allows computing *rank* and *select* on it efficiently. Standard techniques [6] to support SA functionality include backward searching for *find*, and sampling some suffix array values for *locate* and *extract*.

Assume that we want to retrieve $\mathsf{SA}[i]$. If suffix array position i is sampled, we can just use the sampled value. Otherwise we compute $LF(i)$ and continue from that position. Eventually, after k steps, we find a sample $(LF^k(i), \mathsf{SA}[LF^k(i)])$. As $\mathsf{SA}[LF(i)] = \mathsf{SA}[i] - 1$ (unless $\mathsf{SA}[i] = 1$), we now know that $\mathsf{SA}[i] = \mathsf{SA}[LF^k(i)] + k$. The special case can be avoided by always sampling $(\mathsf{SA}^{-1}[1], 1)$. In a similar way, we can also use Ψ to find $\mathsf{SA}[i]$. As $\mathsf{SA}[\Psi(i)] = \mathsf{SA}[i] + 1$ (unless $\mathsf{SA}[i] = n$), we get $\mathsf{SA}[i] = \mathsf{SA}[\Psi^k(i)] - k$, where $(\Psi^k(i), \mathsf{SA}[\Psi^k(i)])$ is a sampled position.

To extract substring $T[i,j]$, we find the smallest $k \geq j$, for which $(SA^{-1}[k], k)$ has been sampled, and use LF to proceed backwards. After $k-j$ steps, we have reached $(LF^{k-j}(\mathsf{SA}^{-1}[k]), j)$, where we determine $T[j] = \mathrm{char}(LF^{k-j}(\mathsf{SA}^{-1}[k]))$. After that, we proceed with LF until we reach $T[i]$, and determine each character in the same way. Instead of using LF, we can also use Ψ by starting at the largest sampled value $k \leq i$ and moving forward until we reach $T[j]$.

The *XBW transform* [10] is a generalization of the

Burrows-Wheeler transform for labeled trees, where leaf nodes and internal nodes are labeled with different alphabets. Each internal node of the tree is represented as a concatenation of the labels of its children. These representations, sorted in lexicographic order according to the path labels from the node to the root, form sequence BWT. The starting position of each internal node in BWT is marked by an 1-bit in bit vector F, so that the node with lexicographic rank i can be found as $\mathsf{BWT}[\text{select}_1(F, i), \text{select}_1(F, i+1) - 1]$.

XBW supports tree navigation with generalizations of functions LF and Ψ. In downward functions such as LF, the lexicographic ranks returned by the regular versions of the functions are converted into BWT ranges by using *select* on bit vector F, as above. Upward functions such as Ψ work in the opposite way, converting BWT ranges into lexicographic ranks by using *rank* on bit vector F, before calling the regular version of the function.

function LF($[sp, ep], c$)
  $sp \leftarrow \text{select}_1(F, sp)$
  $ep \leftarrow \text{select}_1(F, ep + 1) - 1$
  $sp \leftarrow C[c] + \text{rank}_c(\mathsf{BWT}, sp - 1) + 1$
  $ep \leftarrow C[c] + \text{rank}_c(\mathsf{BWT}, ep)$
  $sp \leftarrow \text{rank}_1(M, sp)$
  $ep \leftarrow \text{rank}_1(M, ep)$
  return $[sp, ep]$

function $\Psi(i, j)$
  $c \leftarrow \text{char}(i)$
  $i \leftarrow \text{select}_1(M, i) + j - 1$
  $i \leftarrow \text{select}_c(\mathsf{BWT}, i - C[c])$
  $i \leftarrow \text{rank}_1(F, i)$
  return i

Fig. 3. Pseudocode for the basic navigation functions LF and Ψ.

4 Burrows-Wheeler transform for finite languages

Backward searching using BWT is based on the following property: text positions containing character c are sorted in the same order as text positions preceded by character c. If we consider the sequence a finite automaton, we could say that nodes labeled with character c are sorted in the same order as nodes with a predecessor labeled with character c. To use this idea to index finite automata, we need to solve two problems: handling nodes with multiple predecessors or successors (this section), and constructing an automaton that can be sorted in the desired way (Section 6).

4.1 Prefix-range-sorted automata

As mentioned above, backward searching using BWT relies on the property that the nodes of the automaton can be sorted in a certain way. We formalize this property as prefix-range-sortedness

Definition 2. Let $A = (V, E)$ be a finite automaton, and let $v \in V$ be a node. Let $rng(v)$ be the smallest (open, semiopen, or closed) lexicographic range containing all suffixes that can be recognized from node v. Node v is *prefix-range-sorted*, if no string $S \in rng(v)$ is recognized from any other node $v' \neq v$. Automaton A is prefix-range-sorted, if all nodes are prefix-range-sorted.

In the following, we use a stronger definition to simplify the discussion. The results for prefix-sorted automata generalize for prefix-range-sorted automata.

Definition 3. 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.

We can use the prefixes $p(v)$ to sort the nodes of a prefix-sorted automaton in lexicographic order. Consider now the list of outgoing edges (u, v), sorted by pairs $(p(u), p(v))$. The edges in this order are also sorted by sequences $\ell(u)p(v)$, which is the key for backward searching to work properly. For any given character c, all outgoing edges from nodes with label c are lexicographically adjacent, and they are sorted by the prefix $p(v)$ of the destination node. Similarly, all occurrences of character c in BWT encode an incoming edge from a node with label c, and these edges are also sorted by prefix $p(v)$ of the destination node. Hence the incoming edge labeled by the jth occurrence of character c is the same edge as the outgoing edge of rank $C[c] + j$. Note that $C[c]$ stores the number of occurrences of characters smaller than c in BWT, not the number of nodes with label smaller than c.

4.2 Generalizing the XBW transform

Bit vector F, mapping lexicographic ranks into BWT ranges in the XBW transform, allows a single node to have multiple predecessors. We can use a similar idea to allow multiple successors, extending XBW from trees to finite automata. We encode the number of outgoing edges in another bit vector M. For each node v in lexicographic order, we append an 1-bit and $out(v) - 1$ 0-bits to M. This allows us to compute the outdegree of the node with lexicographic rank i as $\text{select}_1(M, i + 1) - \text{select}_1(M, i)$. For convenience, we assume that the final node $V_{|V|}$ has a single outgoing edge to the initial node V_1.

Backward navigation (LF) first uses bit vector F to convert lexicographic ranks into a BWT range, then calls the regular version of the function, and finally uses bit vector M to convert the edge range into lexicographic ranks. Forward navigation (Ψ) uses bit vectors M and F in the opposite way. See Fig. 3 for pseudocode for basic navigation functions, and below for the definitions of the functions.

- $LF([sp, ep], c)$ is the lexicographic range of nodes with

label c that have a successor in the lexicographic range $[sp, ep]$. This is essentially a step of backward searching with character c.
- $\Psi(i, j)$ is the lexicographic rank of the jth successor of the node with lexicographic rank i.
- $\text{char}(i)$ is the label of the node with lexicographic rank i.

4.3 Searching

The basic navigation function can be used to support the following generalization of suffix array functionality (see Definition 1).

- *find*(P) returns the lexicographic range $[sp, ep]$ of nodes recognizing any suffix that has pattern P as its prefix.
- *locate*(i) returns a numerical value stored in the node with lexicographic rank i.
- *extract*(i, P) returns the label of path P starting from the node with lexicographic rank i.

We can support *find* by replacing the first two lines of the loop body in Fig. 2 with function LF from Fig. 3.

For *locate*, we assume that there is a (not necessarily unique) numerical value $id(v)$ stored in each node $v \in V$. Examples of these values include node ids (so that $id(v_i) = i$) and positions in a reference sequence or a 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 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_s.

As we have sampled all nodes with multiple successors, we can use the *locate* algorithm of a CSA with our new function Ψ. To retrieve $id(u)$ for node u of lexicographic rank i, we first check if $B_s[i] = 1$, and return sample $rank_1(B_s, i)$, 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.

In *extract*, we assume that the description of path P allows us to determine in constant time, which outgoing edge we should take. With such description, we can use function Ψ to move forward on the path, and function $\text{char}(\cdot)$ to read the next character of the path label. The algorithm is similar to the *extract* algorithm of a CSA, with the exception that we already know the lexicographic rank of the initial node. This is because we might be using a node value scheme that does not allow mapping node values to lexicographic ranks.

4.4 Analysis

Similar size bounds as for different variants of the compressed suffix array can be determined for compressed representations of BWT, if we first define a generalization of empirical entropy. Bit vectors F and M have $|V|$ 1-bits out of $|\mathsf{BWT}|$ and $|E|$, respectively. The number and the size of the samples depend greatly on the node value scheme.

In the following, we assume that BWT has been encoded with indicator bit vectors. For each character $c \in \Sigma$, we have bit vector B_c such that $B_c[i] = 1$ whenever $\mathsf{BWT}[i] = c$. With this encoding, we can compute $\text{rank}_c(\mathsf{BWT}, i) = \text{rank}_1(B_c, i)$ and $\text{select}_c(\mathsf{BWT}, i) = \text{select}_1(B_c, i)$.

Theorem 1. *Assume that rank and select on bit vectors require $O(t_B)$ time. GCSA with sample rate d supports* find*(P) in $O(|P| \cdot t_B)$ time,* locate*(i) in $O(d \cdot t_B)$ time, and* extract*(i, P) in $O(|\mathsf{P}| \cdot t_B)$ time.*

Proof: Basic operations LF, Ψ, and char take $O(t_B)$ time, as they require a constant number of bit vector operations. As *find* does one LF per character of the 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 $O(d \cdot t_B)$.

For each character of path P, operation *extract* determines in constant time, which forward edge to follow, and advances to the next character using Ψ in $O(t_B)$ time. □

An additional benefit of this encoding is that it makes bit vector F redundant. As a prefix-range-sorted automaton is reverse deterministic, each node can have at most one predecessor with a given label. Hence the section of BWT corresponding to a node can have at most one occurrence of each character, meaning that we can put all these predecessor labels into the same position in bit vectors B_c. Bit $B_c[i]$ now determines, whether the node with lexicographic rank i has a predecessor with label c. This is a major speedup in practice, as we get rid of one third of bit vector operations.

5 Complexity Aspects

In addition to all finite languages, we can index some infinite languages as well.

Theorem 2. *The class of languages recognized by prefix-range-sorted automata is strictly between finite languages and regular languages.*

Proof: In Section 6, we will show that every automaton recognizing a finite language can be transformed into an equivalent prefix-range-sorted automaton. Some infinite languages can also be recognized by such automata. Consider the regular language $\{\#x\$ \mid x \in \{a, b\}^*\}$. The minimal automaton recognizing this language is prefix-range-sorted, as each node has a distinct label.

Consider now the language

$$L = \{\#x\$ \mid x \in \{a, b\}^* \cup \{a, c\}^*\}.$$

Assume that there is a prefix-range-sorted automaton that recognizes the language. Suffixes $B_n = a^n b\$$ and

$C_n = a^n c\$$ must be recognized from different nodes, as bB_n is a suffix of language L, while bC_n is not. Because $B_{n+1} < C_{n+1} < B_n$, suffixes B_n and B_{n+1} must also be recognized from different nodes. As the automaton must have an infinite number of nodes, it cannot be a finite automaton. □

It turns out that prefix-range-sorted automata are a generalization of de Bruijn graphs. In the terminology of this paper, a k-dimensional de Bruijn graph is a prefix-sorted automaton, where all nodes are prefix-sorted by prefixes of length k.[2] Bowe et al. [15] discovered independently that a structure similar to GCSA can be used to index de Bruijn graphs.

In general, the prefix-range-sorted automaton can be exponentially larger than the finite automaton from which it is built. This is evident from the connection to determinizing finite automata as discussed in Sect. 6. For a concrete example, consider a finite automaton for language $(a\{b,c\})^n$. When the paths from a node of this automaton are sorted in lexicographic order, they all share long prefixes with the paths starting from other nodes. Hence, exponentially many nodes are created for the prefix-range-sorted automaton.

The worst case behaviour of prefix-range-sorted automaton raises the question whether there would be some other approach to index a finite language for path queries, that would have practical index size also in the worst case. There is an easy reduction from set intersection indexing problem showing that a less than quadratic size index supporting near-linear time path searching is unlikely to exist for all instances [20].

The above considerations suggest that studying indexes for graphs with good expected case space requirement is the only practical direction to consider given the huge inputs in genome research applications.

6 Index construction

A straightforward way of constructing a prefix-sorted automaton from a finite automaton recognizing a finite language is similar to the *prefix-doubling* approach to suffix array construction [21]. The algorithm consists mostly of sorting, scanning, and relational joins. Hence it can be efficiently implemented in parallel, distributed, and external memory settings.

Theorem 3. *Assume we have a length n multiple alignment of r sequences over alphabet of size σ. We can build a prefix-range-sorted automaton recognizing all paths through the alignment in $O(nr + |V'| \log n + |E'|)$ time and $O(nr \log \sigma + |V'| \log |V'| + |E'| \log |E'|)$ bits of space, where V' and E' are the largest intermediate sets of nodes and edges, respectively.*

Proof: From Lemmas 1, 2, and 3 below. □

2. Strictly speaking, a de Bruijn graph has to be complete in the sense that it contains all edges such prefix-sorted automaton can have.

The sizes of the largest intermediate sets of nodes and edges are analyzed in a restricted model in Section 6.4. An example of construction can be seen in Figures 4 and 5 and Table 1.

6.1 Building a reverse deterministic automaton

With the following algorithm, we can build a reverse deterministic automaton that recognizes all paths through a multiple alignment of sequences. The same approach, when used with a reference sequence and a set of edit operations, is essentially a variant of the textbook algorithm for determinizing finite automata.

In the following, we assume that the alignment consists of sequences $S_1, \ldots, S_r$ of length n, possibly containing gap characters $-$. Sequences S_i and $S_{i'}$ are considered to be equivalent at position j, if $S_i[j] = S_{i'}[j] \neq -$. We can allow edit operations longer than one character by using a context to determine the equivalence of two positions. With context length $k \geq 0$, sequences S_i and $S_{i'}$ are equivalent at position j, if $S_i[j] = S_{i'}[j] \neq -$ and the next k non-gap characters in the sequences are also equal.

The algorithm works in one pass from right to left. Assume that we have already processed positions $j+1$ to n and created the corresponding part of the automaton. For each sequence S_i with a non-gap character in column j, we first create a temporary node $v_{i,j}$ and an edge from $v_{i,j}$ to the node corresponding to the next non-gap character in sequence S_i. Next, we merge the temporary nodes for those sequences that are equivalent at position j.

Finally, we find the preceding non-gap characters for all sequences with a non-gap character at position j. Assume that two or more sequences that are equivalent at position j have c as the preceding non-gap character. If these characters c occur at different positions, we move them all to the rightmost of these positions. This way, the node $v_{i,j}$ corresponding to the equivalent sequences will only have one predecessor with label c.

Lemma 1. *Let n be the length of the multiple alignment, r the number of sequences, and σ the size of the alphabet. Building a reverse deterministic automaton takes $O(nr)$ time and requires $O(nr \log \sigma + |E| \log |E|)$ bits of space, where E is the set of edges of the automaton.*

Note that each position can be processed in $O(r)$ amortized time, regardless of context length, by keeping the suffixes $S_i[j]$ in sorted order and maintaining the lengths of the longest common prefixes of lexicographically adjacent suffixes.

6.2 Creating a prefix-sorted automaton

Definition 4. Let A be a finite automaton recognizing a finite language, and let $k > 0$ be an integer. Automaton A is *k-sorted* if, for each node v, the labels of all paths from v to $v_{|V|}$ have a common prefix $p(v,k)$ of length k, or node v is prefix-sorted by prefix $p(v,k)$ of length at most k.

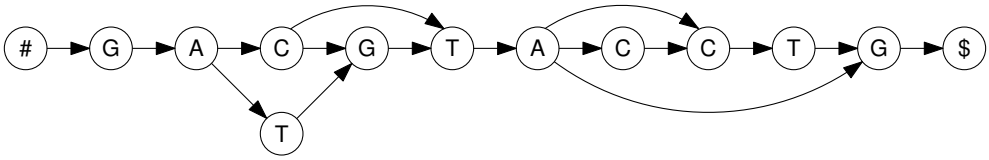

Fig. 4. A reverse deterministic automaton corresponding to the first 10 positions of the multiple alignment in Fig. 1.

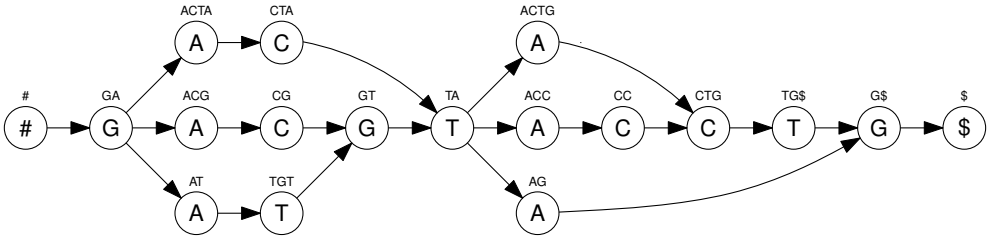

Fig. 5. A prefix-sorted automaton built for the automaton in Fig. 4. The strings above nodes are prefixes $p(v)$.

TABLE 1
GCSA for the automaton in Fig. 5. 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	$
M	1	1	1	1	1	1	1	1	1	1	1	1	100	1	100	1	1	1

Every automaton is 1-sorted. Automaton A is prefix-sorted if and only if it is n-sorted, where n is the length of the longest string in $L(A)$.

Starting from a reverse deterministic automaton $A = A_0$, we create the nodes 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. For every node $v \in V_i$, let $P(v)$ be the path of A corresponding to prefix $p(v, 2^i)$. Let $\mathit{from}(v)$ and $\mathit{to}(v)$ be the first and the last nodes of path $P(v)$, and let $\mathit{rank}(v)$ be the lexicographic rank of prefix $p(v, 2^i)$ among all distinct prefixes $p(u, 2^i)$ of nodes $u \in V_i$. We store node $v \in V_i$ as triples $(\mathit{from}(v), w, \mathit{rank}(v))$, one per successor w of node $\mathit{to}(v)$ in automaton A, or as a triple $(\mathit{from}(v), 0, \mathit{rank}(v))$ if $\mathit{to}(v)$ has no successors. If node v has unique $\mathit{rank}(v)$ value, then it is prefix-sorted.

The basic step of the algorithm is the *doubling* step from A_i to A_{i+1}. If node $u \in V_i$ is prefix-sorted, we *duplicate* it as $w \in V_{i+1}$, and set $\mathit{rank}(w) = (\mathit{rank}(u), 0)$. Otherwise we create a *joined* node $uv \in V_{i+1}$ for every node $v \in V_i$ such that $P(uv) = P(u)P(v)$ is a path in A, and set $\mathit{rank}(uv) = (\mathit{rank}(u), \mathit{rank}(v))$. As path $P(uv)$ exists if and only if there is a triple $(\mathit{from}(u), \mathit{from}(v), \mathit{rank}(u))$, this requires one relational join. When the nodes of A_{i+1} have been created, we sort them by their ranks, and replace the pairs of integers with integer ranks.

The doubling step is followed by the *pruning* step, where we merge equivalent nodes. The nodes in V_{i+1} are sorted by their $\mathit{rank}(\cdot)$ values. If all nodes sharing a certain $\mathit{rank}(\cdot)$ value also share their $\mathit{from}(\cdot)$ node, these nodes are equivalent, and can be merged. Merging makes the resulting node prefix-sorted.

Lemma 2. *Prefix-doubling algorithm creates the nodes of a prefix-sorted automaton equivalent to A in $O(|V'| \log n)$ time and $O(|V'| \log |V'|)$ bits of space in addition to automaton A, where V' is the largest set of nodes during construction, and n is the length of the longest string in $L(A)$.*

The lemma assumes using a linear-time integer sorting algorithm.

6.3 Creating the edges

Let $A = (V, E)$ be a reverse deterministic automaton recognizing a finite language, and let W be the set of nodes of an equivalent prefix-sorted automaton. To create the edges, we first merge nodes with adjacent $\mathit{rank}(\cdot)$ values, if they share their $\mathit{from}(\cdot)$ node. The resulting set V' is the set of nodes of a prefix-range-sorted automaton $A' = (V', E')$ equivalent to automaton A. The set of edges E' can be constructed efficiently from automaton A and the set of nodes V'.

The key to edge construction is that for each node $v \in V'$, the set of $\mathit{from}(u)$ nodes for the predecessors u of node v is the same as the set of predecessors of node $\mathit{from}(v)$. With automaton A and the set of nodes V', we can output the edges $(u, v) \in E'$ initially as pairs $(\mathit{from}(u), v)$, sorted by $(\ell(\mathit{from}(u)), \mathit{rank}(v))$. Note that by doing this, we have also sorted the edges by $\mathit{rank}(u)$.

We can map nodes $\mathit{from}(u)$ to nodes u by scanning the sorted lists of nodes and edges. As every node has at

least one outgoing edge, and no adjacent nodes share their $from(\cdot)$ value, all adjacent edges with the same $from(\cdot)$ values start from the current node. When the $from(\cdot)$ value changes in the list of edges, we advance to the next node.

Lemma 3. *Creating the edges of prefix-range-sorted automaton A' takes $O(|W| + |E'|)$ time and requires $O(|W|\log|W| + |E'|\log|E'|)$ bits of space, where W is the set of nodes of an equivalent prefix-sorted automaton.*

6.4 Expected case analysis

In the following, all random choices are independent and identically distributed.

We analyze the size of the automata created by the doubling algorithm in the following model. Let $S[1,n]$ be a reference sequence, and let p be the mutation rate. For each position $i = 1,\ldots,n$, the initial automaton A has a node u_i with label $\ell(u_i) = S[i]$, randomly chosen from alphabet Σ. With probability p, there is also another node w_i with a random label $\ell(w_i) \in \Sigma \setminus \{S[i]\}$. The automaton has edges from all nodes at position i to all nodes at position $i+1$.

Definition 5. Let $k > 0$ be an integer. A *k-path* in an automaton is a path of length k, or a shorter path ending at the final node.

Let $k > 0$ be an integer. For any position i in the reference sequence, let $X_{i,k}$ be the number k-paths starting from a node at position i. If there are j mutated positions covered by these paths, then $X_{i,k} = 2^j$, and each of the paths has a different label. The number of mutations is binomially distributed, with the path length and the mutation probability as the parameters. From the moment-generating function for binomial distribution, we get

$$\mathrm{E}[X_{i,k}] = \sum_{j=0}^{k} \Pr(X_{i,k} = 2^j)2^j \leq (1+p)^k. \qquad (1)$$

For positions $i = 1,\ldots,n-k+1$, this is an equality.

Lemma 4. *Let A_h be a 2^h-sorted automaton equivalent to the original automaton A. Then $N(2^h) = n(1+p)^{2^h}+2$ is an upper bound for the expected number of nodes in A_h.*

Proof: For every 2^h-path starting from a position i in the reference sequence, there is at most one node in automaton A_h. On the other hand, every node in the automaton, except for the initial and the final nodes, corresponds to a path that can be extended to some such 2^h-path. Hence the total number of nodes is at most $\sum_{i=1}^{n} X_{i,k} + 2$. By Eq. 1, the expected number of nodes is at most $N(2^h)$. □

Lemma 5. *Let A_h be the 2^h-sorted automaton built from automaton A. Then $N(2^h)(1+p)$ is an upper bound for the expected number of edges in A_h.*

Proof: The indegree of the initial node of A_h is 0. For every other node v, let $pos(v)$ be the position of the reference sequence corresponding to $from(v)$. If $from(v)$ is the final node of A, then $pos(v) = n+1$. If there is no mutation at position $pos(v)-1$, then $in(v) = 1$. Otherwise $in(v) = 2$. Hence the expected number of edges is at most $(1+p)$ times the number of nodes. □

Consider the expectation $\mathrm{E}[X_{i,k}X_{i',k}]$ for a pair of text positions $i < i'$. If $i' \geq i+k$, then the random variables are independent, and the expectation becomes

$$\mathrm{E}[X_{i,k}X_{i',k}] = \mathrm{E}[X_{i,k}]\,\mathrm{E}[X_{i',k}] \leq (1+p)^{2k}. \qquad (2)$$

Otherwise assume that the paths starting from positions i and i' overlap in $k' < k$ positions. Then the expectation is a product of the expectations of three independent random variables $X_{i,k-k'}$, $X^2_{i',k'}$, and $X_{i'+k',k-k'}$. By using the moment-generating function, we get

$$\mathrm{E}[X_{i,k}X_{i',k}] \leq (1+p)^{2(k-k')}(1+3p)^{k'} \leq (1+p)^{3k}. \qquad (3)$$

Definition 6. A pair of nodes of automaton A_h *collides*, if the corresponding 2^h-paths have identical labels.

Lemma 6. *Let A_h be the 2^h-sorted automaton built from automaton A by using the doubling algorithm. The expected number of colliding pairs of nodes in automaton A_h is at most $C(2^h) = n^2(1+p)^{3\cdot 2^h}/\sigma^{2^h}$.*

Proof: If two paths start from the same position in the reference sequence, the corresponding nodes cannot collide. As the colliding paths must be of length 2^h (otherwise the nodes would be prefix-sorted), the probability of collision of any given pair is σ^{-2^h}. By Equations 2 and 3, the expected number of colliding pairs is at most

$$\sum_{i<i'} \mathrm{E}\left[X_{i,2^h}X_{i',2^h}/\sigma^{2^h}\right] \leq n^2(1+p)^{3\cdot 2^h}/\sigma^{2^h}.$$

The lemma follows. □

Lemma 7. *Let n be the length of the reference sequence, σ the size of the alphabet, and $p < \sigma^{1/3} - 1$ the mutation rate. For any $\varepsilon > 0$, the largest automaton created by the doubling algorithm has at most $n(1+p)^k + 2$ nodes with probability $1-\varepsilon$, where $k = 2\log_\sigma \frac{n^2}{\varepsilon}/(1-3\log_\sigma(1+p))$.*

Proof: We want to find $k = 2^h$, for an integer h, such that the expected number of colliding pairs in automaton A_h is at most ε. Then, by Markov's inequality, the probability of having a colliding pair is at most ε. If there are no colliding pairs, then the automaton is prefix-sorted. By Lemma 4, if this happens after h doubling and pruning phases, the expected number of nodes in the largest automaton created is at most $N(k) = n(1+p)^k+2$.

By using the bound for the expected number of colliding pairs from Lemma 6, we get

$$C(k) = \frac{n^2(1+p)^{3k}}{\sigma^k} \leq \varepsilon \iff \frac{\log_\sigma \frac{n^2}{\varepsilon}}{1-3\log_\sigma(1+p)} \leq k.$$

As k has to be a power of two, $2\log_\sigma \frac{n^2}{\varepsilon}/(1-3\log_\sigma(1+p))$ is an upper bound for the smallest suitable k. □

Lemma 8. *For a random reference sequence of length n and mutation rate $p \leq 0.1$, the expected number of edges in the largest automaton is at most $n(1+p)^{O(\log_\sigma n)} + O(1)$.*

Proof: For $p \leq 0.1$, the k in Lemma 7 is at most $4 \log_\sigma \frac{n^2}{\varepsilon}$. If we select $\varepsilon = \left(\frac{1}{n}\right)^i$, we get a node bound of $n(1+p)^{4 \cdot (2+i) \log_\sigma n} + 2$ with probability $1 - \varepsilon$. Hence the expected number of nodes is at most

$$n(1+p)^{12 \log_\sigma n} \sum_{i=0}^{\infty} \left(\frac{(1+p)^{4 \log_\sigma n}}{n} \right)^i + 2,$$

which is bounded from above by $n(1+p)^{O(\log_\sigma n)} + 2$. By Lemma 5, the expected number of edges is at most $(1+p)$ times that. □

Theorem 4. *Let n be the length of the reference sequence, σ the size of the alphabet, and p the mutation rate. If $p = O(1/\log_\sigma n)$, then the expected number of nodes and edges in the largest automaton created by the prefix-doubling algorithm is $O(n)$.*

Proof: From Lemma 8. □

7 Implementation and Experiments

We have implemented GCSA in C++, using the components from our implementation of RLCSA [11], [22].[3] For each character $c \in \Sigma \cup \{\#\}$, we use a gap encoded bit vector to mark the occurrences of c in BWT. Bit vector M is run-length encoded, as it usually consists of long runs of 1-bits. 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. Block size was set to 32 bytes in all bit vectors. The implementation supports multiple automata in a single index in a similar way as RLCSA supports multiple sequences.

In addition to the basic index, we have implemented a *backbone* structure to extend its functionality. A backbone is the path in the automaton corresponding to the reference sequence. The backbone structure consists of three components: a succinct bit vector marking the nodes belonging to the backbone, another succinct bit vector marking the nodes that belonged to the backbone of the original automaton (before prefix-sorting), and a run-length encoded bit vector marking the outgoing edges used to advance on the backbone. The structure can be used in e.g. read mapping to map paths in the automaton to the reference sequence.

The implementation was compiled on g++ version 4.6.4. Unless otherwise noted, we used a system with 32 gigabytes of memory and two quad-core 2.53 GHz Intel Xeon E5540 processors running Ubuntu 12.04 with Linux kernel 3.2.0 for our experiments. When multiple threads were used, parallelization was done using OpenMP and libstdc++ parallel mode.

7.1 Index construction

For the basic performance experiments, we chose the human reference genome[4] and the Finnish subset of the frequent mutations reported in dbSNP database[5] as our test data. Due to the memory requirements of GCSA construction, we did all index construction on a system with 1 terabyte of memory and four 8-core 2.00 GHz Intel Xeon X7550 processors running Red Hat Enterprise Linux 6 with Linux kernel 2.6.32 instead of our normal test environment. We used 24 CPU cores out of the available 32 physical and 64 logical cores. For these experiments, GCSA was compiled on g++ version 4.4.6. For the rest of the experiments, we used our normal test environment.

We built separate automata for chromosomes 1–22 and X, determinized them using the textbook algorithm, and built indexes for each of the chromosomes with sample rate 16. The results can be seen in Table 2. Determinization requires fairly constant resources, taking 2.8 to 3.0 microseconds and 225 to 239 bytes per node. For most chromosomes, index construction took 7.9 to 9.6 microseconds per node, while memory usage ranged from 73 to 121 bytes per node. The final sizes were 6.1 to 8.7 bits per node for the index and 2.4 to 3.1 bits per node for the backbone.

Buiding the index for chromosome 17 required significantly more resources, while the system ran out of memory when indexing chromosomes 3, 6, 8, 11, 16, and 18. The precise reason for this behaviour is unknown, apart from the general fact that the construction may require exponential time and space in the worst case. With these chromosomes, the number of nodes suddenly increased from hundres of millions to up to hundreds of billions in doubling step 8, where the path length increased from 128 to 256. This suggests that the problems arise from variation in repetitive regions in the chromosomes.

Next we proceeded to build the index for the entire human genome. For most of the chromosomes, we used the same automata as in the previous experiment. For chromosomes 3, 6, 8, 11, 16, 17, and 18, where the index construction either failed or required unreasonable resources, we used a heuristic approach to limit the allowed combinations of nearby mutations. We built a multiple alignment of four sequences, where the first sequence was the reference sequence for that chromosome, and the rest of the sequences represented different subsets of the mutations. Note that this approach puts a limit on how many mutations can span over any single locus, that is, four sequences can represent up to three overlapping mutations. It turned out that more than 99.97% of the variation in the given Finnish subset can be represented with a multiple alignment of just four strains. We then used the algorithm from Section 6.1 with context length $k = 4$ to build a reverse deterministic automaton for the chromosome.

With automata for each of the chromosomes ready, we determinized them when necessary, and built GCSA with sample rate 16 for the 23 automata. To see the resource requirements in context, we also built RLCSA (sample rate 32) and BWA 0.7.4 [23] for the reference genome. The results can be seen in Table 3. As expected, building

3. http://www.cs.helsinki.fi/group/suds/gcsa/, May 2013 version was used in the experiments.

4. Genome Reference Consortium Human Build 37 patch release 5 (GRCh37.p5), ftp://ftp.ncbi.nih.gov/genomes/H_sapiens/Assembled_chromosomes/seq/hs_ref_GRCh37.p5_chr*.fa.gz

5. See http://www.ncbi.nlm.nih.gov/SNP/. VCF, automata and multiple alignments are available at http://www.cs.helsinki.fi/group/gsa/1000gen-FIN-WholeGenome/

TABLE 2
GCSA construction for individual chromosomes. Number of SNPs and nodes in the initial automaton, time and memory usage for determinization and index construction, and the final size of the index and the backbone.

			Determ.		Construction		Size	
Input	**SNPs**	**Nodes**	**Time**	**Space**	**Time**	**Space**	**Index**	**Backbone**
Chr 1	964K	250M	12 min	52 GB	39 min	17 GB	216 MB	76 MB
Chr 2	1,043K	244M	12 min	51 GB	37 min	17 GB	224 MB	76 MB
Chr 3	876K	199M	10 min	42 GB	–	–	–	–
Chr 4	904K	192M	9 min	40 GB	31 min	22 GB	183 MB	63 MB
Chr 5	798K	182M	9 min	38 GB	28 min	14 GB	172 MB	60 MB
Chr 6	818K	172M	9 min	36 GB	–	–	–	–
Chr 7	728K	160M	8 min	33 GB	24 min	12 GB	150 MB	52 MB
Chr 8	679K	147M	7 min	31 GB	–	–	–	–
Chr 9	535K	142M	7 min	30 GB	22 min	10 GB	117 MB	42 MB
Chr 10	625K	136M	7 min	30 GB	19 min	10 GB	123 MB	42 MB
Chr 11	616K	136M	7 min	30 GB	–	–	–	–
Chr 12	598K	134M	7 min	30 GB	21 min	11 GB	139 MB	49 MB
Chr 13	460K	116M	6 min	24 GB	17 min	8 GB	96 MB	35 MB
Chr 14	300K	108M	5 min	23 GB	15 min	7 GB	85 MB	31 MB
Chr 15	361K	103M	5 min	22 GB	15 min	7 GB	83 MB	31 MB
Chr 16	383K	91M	4 min	19 GB	–	–	–	–
Chr 17	339K	82M	4 min	17 GB	107 min	117 GB	615 MB	324 MB
Chr 18	357K	78M	4 min	16 GB	–	–	–	–
Chr 19	287K	59M	3 min	12 GB	8 min	5 GB	52 MB	19 MB
Chr 20	271K	63M	3 min	13 GB	9 min	5 GB	55 MB	19 MB
Chr 21	173K	48M	2 min	10 GB	6 min	3 GB	37 MB	14 MB
Chr 22	170K	51M	2 min	11 GB	7 min	4 GB	38 MB	15 MB
Chr X	413K	156M	8 min	33 GB	22 min	11 GB	133 MB	46 MB

GCSA requires significantly more resources than building a regular BWT-based index, while the final size of the index is similar.

7.2 Pattern matching

We compared the pattern matching performance of our implementation of GCSA to RLCSA, using the indexes built for the human genome in the previous section. RLCSA is an a compressed suffix array implementation for a collection of sequences, with similar design choices as in our implementation of GCSA. This way, any differences in performance should come from the fundamental differences between the indexes, and not from any implementation choices. The only major difference is in *locate*. While RLCSA reports each matching position only once, prefix-sorting may create multiple copies of a node in GCSA, making it necessary to filter out duplicates before reporting the results.

In addition to exact pattern matching, we also did approximate matching with edit distances 1, 2, and 3. We implemented a backtracking algorithm similar to the one used in BWA [23]. Unlike the algorithm in BWA, our algorithm is complete, reporting all matching positions with the minimum edit distance. As our indexes do not contain the reverse sequences, we had to match $O(|P|\log|P|)$ instead of $O(|P|)$ characters when building the lower bound array for pattern P, making approximate matching with small edit distances slower than in BWA.

The results of the pattern matching experiments can be seen in Table 4. With low edit distances, GCSA found significantly more matches and unique matches than RLCSA, while the differences became smaller with higher distances. This is probably due to the fact that the most of the mutations encoded in the automaton are SNPs or other simple mutations that can be handled by increasing the edit distance by 1 during pattern matching.

Theoretically, GCSA should be twice slower than RLCSA, as it requires twice as many bit vector operations to perform the same queries. With edit distance 0, the actual difference was 3.0 times, as GCSA had to locate duplicate positions and filter them out, while RLCSA located each matching position just once. With larger edit distances, the difference grew smaller, becoming 1.5 times with edit distance 3. This was due to the backtracking algorithm requiring larger part of the query time than plain *find*, and also due to the overhead in the algorithm.

We also compared GCSA to BWBBLE [14], a recent BWT-based read aligner for pan-genomes. Given an upper bound for read length, BWBBLE creates a new sequence for each known indel, with an amount of context before and after the indel depending on the upper bound. This way, reads mapping to a genome containing that indel can be mapped to the new sequence. SNPs are encoded

TABLE 3
Index construction for the human genome. Time and memory usage for index construction and the final size of the index and the backbone. Determinization includes merging the automata into a single file. Construction includes index and backbone construction from the merged automata. Overall includes determinization, construction, I/O, handling the files, and building the automata from the multiple alignments. RLCSA includes two construction options: build separate indexes for the chromosomes and merge them, or merge the chromosomes into a single file before building the index.

Index	Time	Space	Index	Backbone
GCSA (determinization)	1.9 h	98.9 GB		
GCSA (construction)	11.9 h	214.9 GB	2848 MB	992 MB
GCSA (overall)	14.1 h	214.9 GB		
RLCSA (merge indexes)	1.2 h	8.4 GB	2587 MB	–
RLCSA (single file)	0.2 h	47.0 GB		
BWA	1.5 h	4.2 GB	4343 MB	–
BWBBLE	1.4 h	59.5 GB	11.32 GB	–

TABLE 4
Pattern matching with GCSA, RLCSA (*find* and *locate*) and BWBBLE (*find*) with 10 million reads of length 56 on the human genome. Query times, the fraction of patterns matching to one or more positions, and the fraction of patterns matching to exactly one position. Only one CPU core was used. The matching patterns with edit distance k includes the matches with smaller edit distances. The backbone structures were not used in these experiments.

	GCSA			RLCSA			BWBBLE	
Errors	Time	Matches	Unique	Time	Matches	Unique	Time	Matches
0	84 min	86.47%	80.20%	28 min	82.70%	76.67%	416 min	86.54%
1	114 min	91.94%	84.21%	45 min	91.40%	83.67%	666 min	91.92%
2	271 min	94.04%	85.33%	157 min	93.87%	85.12%	2354 min	93.99%
3	2302 min	95.54%	86.02%	1540 min	95.44%	85.86%	3764 min	95.32%

directly in the sequences by using the power set of the alphabet, avoiding the need for creating new sequences for the SNPs as well. As each character of the original alphabet matches $2^{\sigma-1}$ different characters of the superset alphabet, the price of using the powerset alphabet is an increased amount of branching during *find*.

We built BWBBLE for the reference sequence and the same set of frequent mutations as we used when building GCSA. As can be seen in Table 3, the construction of a BWBBLE index requires similar time as building traditional BWT-based indexes, such as RLCSA and BWA, but much more memory. The query times for BWBBLE are much slower than for GCSA, as seen in Table 4. There are three factors to consider:

1) BWBBLE uses a similar encoding for the BWT as BWA uses, leading to faster *rank*/*select* times than in GCSA.
2) When a read maps exactly to some path in the automaton, GCSA requires no branching, leading to a very fast *find*.
3) The approximate matching heuristic in BWBBLE is very similar to the one in BWA, leading to less branching with larger edit distances than in the one used by GCSA.

It should be noted that BWBBLE finds more exact matches than GCSA. The index includes all possible combinations of mutations, while GCSA has to use a heuristic that ignores some of them in the difficult chromosomes 3, 6, 8, 11, 16, 17, and 18. On the other hand, GCSA finds more inexact matches than BWBBLE, due to BWBBLE using a heuristic that places restrictions on the placement of gaps in the alignment.

7.3 Read mapping accuracy experiments

Typical variation calling pipelines rely heavily on the accuracy of the underlying read mapping software. In order to evaluate the read mapping accuracy of GCSA, we used BWA [23], which is one of the most widely used read mapping tools, BWBBLE [14] and data available from the Variathon challenge 2013 [24]. The Variathon challenge includes an artificial chromosome (dubbed chr 20) that aims to simulate frequent variations typically observed in human individuals. Since the read data was created artificially (see [24] for details), the read mapping accuracy of different tools can be easily validated: if a read is mapped into the correct position in the reference chromosome, it is called true positive (TP), and if a read is mapped into a wrong position in the reference chromosome, it is called false positive (FP). Similarly, if a decoy read (i.e. a read from a mouse genome used as contaminant in this case) is mapped in the reference chromosome, it is called false positive (FP), and if a decoy

read is not mapped, it is called true negative (TN). False negatives (FN) are the unmapped human reads.

Table 5 gives the read mapping accuracies for GCSA, BWBBLE and BWA. In the single-end case, each read was mapped independently of its mate. We ran BWA using default parameters and both *samse* and *sampe*, where the latter uses additional heuristics to determine the alignment of paired-end reads. BWBBLE was ran using default settings with `-n3` errors and did not support paired-end data. GCSA was ran using $k = 0$ to 4 errors. With $k = 4$, GCSA became more sensitive and finds more true positives than BWA (samse). Recall that BWA uses sophisticated seed-and-extend strategies to speed up its search, thus, the running times are not directly comparable to GCSA. We include a preliminary result for GCSA using seed-and-extend. To align paired-end data, we implemented a simple wrapper around the GCSA aligner that can rescue an unmapped mate by searching for a position that minimizes the edit distance for the unmapped mate within the range of suspected insert size. We observe that using the paired-end information greatly improves the alignment results for both BWA and GCSA. In fact, the Variathon challenge data appears to be too easy to show any significant improvements in the paired-end comparison.

7.4 Highly-polymorphic regions

Databases such as dbSNP already catalogue a clear majority of the observed variation in, for example, any European sample [25]. These type of databases have been successfully used to extract biologically motivated, highly-polymorphic regions, where variants are more likely to have strong phenotypic impact. Some examples include the MHC region in chr 6 [26] and highly-polymorphic regions that show strong enrichment of known, disease-causing mutations [9]. These regions cover only a small fraction of the entire genome (comparable to the exome) and constitute as an interesting target for future clinical studies.

In this experiment we studied the feasibility of indexing the highly-polymorphic regions of the human genome. As a proof of concept we collected the variation-rich regions from 93 genotyped Finnish individuals (1000 genomes project, phase 1 data). The 93 diploid genomes gave us a multiple alignment of 186 strains plus the GRCh37 consensus reference. We chose variation-rich regions that had 10 SNPs within 200 bases or less. The total length of these regions was 2.2 MB. To measure the read mapping performance, we generated 70bp single-end reads from each of the Finnish individuals at a time using wgsim and 2% error rate.

Figure 6 gives the read-alignment results for BWA (default settings), GCSA (k=3 errors) and the best possible performance one can expect (i.e. the donor genome is known and used as BWA's reference). For each case, we report the proportion of reads mapped with and without non-unique mappings. BWA was ran using the consensus sequence (GRCh37) and included only the variation-rich regions. The high variance in the BWA results is mostly due to the long indels present in a subset of the Finnish individuals. In this experiment, GCSA clearly outperforms the alignments against the GRCh37 consensus sequence (BWA) and is performing almost as well as aligning the reads against the individual's own genome.

7.5 Split-read alignment

We tested the feasibility of constructing *splicing graphs* [7] for split-read alignment as described in Sect. 1. This experiment was simulating the task of RNA-sequence alignment against the human chromosome 5. Our main focus here was to study the feasibility of constructing splicing graphs for high-numbers of predicted exons and splice-junctions.

We built the first splicing graph simply by taking the known genes[6] and all their transcripts in the human chromosome 5. The resulting graph contained 12,192 splice-junctions and was used as a gold standard for the rest of our splicing graph experiments. In order to measure the mapping accuracy, we generated a set of one million, 100 bp reads by sampling random positions of the exome. Majority (76%) of the reads mapped to unique positions on the gold standard splicing graph.

To test the scalability of the splicing-graph construction, we generated multiple sets of *predicted splice-junctions*. The idea was to scale-up the number of predicted exons and their transcripts until we hit a bottleneck in the automata construction. This was done by taking the list of known exon–intron boundaries and adding a splice-junction between all exons within a fixed-size window. The number of resulting splice-junctions grows, in the worst-case, in a quadratic-manner to the window-size. We constructed splicing graphs for increasing window-sizes until the graph construction became infeasible. Some examples of construction space and time are given in Table 6. The construction was found to be feasible at least up to 218,409 predicted splice-junctions (corresponding to window-size of 10^5). The resulting predicted splicing-graph allowed us to map up to 74% of the reads to their correct position.

There was a notable increase in the false-positive rate for the largest splicing graph, but this is more of a consequence of our extremely naive heuristics used for choosing the predicted splice-junctions — better prediction heuristics for splice-junctions are out of the scope of this study. These results should be weighted independed of the prediction algorithm used and, primarily, demonstrate the scalability of the graph construction in the context of spicing graphs.

8 Discussion

Our approach can be generalized to index labeled weighted graphs, where the weights correspond to probabilities for moving from one node to another. This does not increase space usage significantly, as the probabilities differ from 1.0 only in nodes with multiple outgoing edges. In the restricted model analyzed in the Section 6.4, the extra space

6. Exported on November 28th 2012 from the public MySQL server at ensembldb.ensembl.org.

TABLE 5
Read mapping accuracies on Variathon 2013 [24] data. GCSA indexing parameters were chosen to match the memory usage of BWA. Times are reported per one million single-end reads. GCSA seed+extend uses the first 32 bases as a seed.

			Single-end				Paired-end			
Method	**Errors**	**Time**	**TP**	**FP**	**TN**	**FN**	**TP**	**FP**	**TN**	**FN**
GCSA	0	1 min	2,402,679	21,641	9,998,964	7,576,716	4,122,080	11,402	10,000,000	5,866,518
GCSA	1	5 min	5,844,214	54,704	9,996,310	4,104,772	8,119,514	23,363	9,999,990	1,857,133
GCSA	2	12 min	8,271,468	81,110	9,992,215	1,655,207	9,594,486	29,005	9,999,963	376,546
GCSA	3	42 min	9,394,715	97,335	9,986,618	521,332	9,904,071	30,868	9,999,894	65,167
GCSA	4	210 min	9,779,248	108,091	9,979,526	133,135	9,956,085	31,573	9,999,776	12,566
GCSA seed+extend	2	27 min	9,577,410	163,478	9,967,103	292,009	9,916,508	53,819	9,998,309	31,364
BWA	default	2 min	9,522,906	101,828	9,984,898	390,368	9,951,808	41,000	9,984,877	22,315
BWBBLE	3	51 min	9,294,203	95,167	9,987,943	622,687	n/a	n/a	n/a	n/a

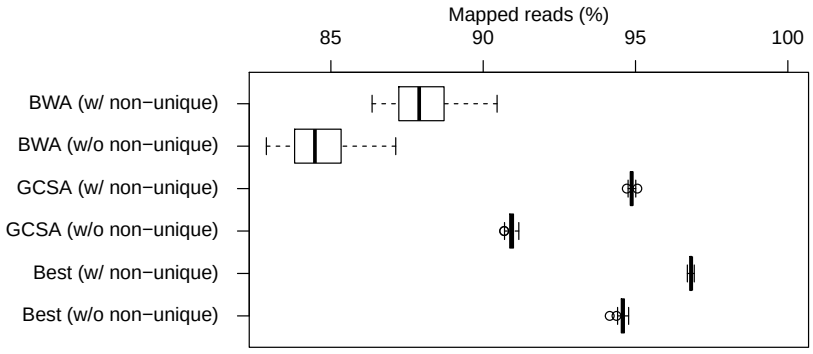

Fig. 6. Mapping reads simulated from highly-polymorphic regions in Finnish genotypes (1000 Genomes Project phase 1 data). BWA used the standard GRCh37 reference, GCSA used the known variation from the 1KGP data, and "Best" used the reference which the reads were simulated from (n=186 Finnish genotypes).

TABLE 6
Experiments for splicing-graph construction using either the known transcripts (here the gold standard) or the predicted transcripts over different window-sizes. We also report the number of uniquely mapped reads at their correct (true positives, TP) and incorrect positions (false positives, FP).

Splicing		**Construction**			**Alignment**			
graph	**Window**	**Junctions**	**Space**	**Time**	**Size**	**TP**	**FP**	**Time**
Known	–	12,192	8.0 GB	39 min	144 MB	755,661	0	3.8 min
Predicted	100	186	8.0 GB	38 min	144 MB	623,496	3,895	3.3 min
Predicted	1,000	3,939	8.0 GB	43 min	144 MB	656,957	3,549	3.6 min
Predicted	10,000	37,298	8.1 GB	45 min	145 MB	731,147	8,071	3.8 min
Predicted	100,000	218,409	33 GB	119 min	427 MB	740,845	17,326	4.3 min
Predicted	250,000	441,581	*failed*		–	–	–	–

requirement is $O(pn \log n)$ bits for a reference sequence of length n and mutation rate p. 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 [27].

The experiments conducted here aimed at demonstrating the feasibility and potential of the approach. As can be observed, our index can not be applied as black-box, but it gives powerful machinery to be tailored for each genome analysis application at hand. The current implementation has an interface compatible with variation calling workflows. The implementation also supports aligning reads to phylogenetic tree of partial-order graphs, and this is also part of ongoing research with our collaborators. We also plan to incorporate the split-read alignment to transcript expression prediction workflows and design a workflow for cancer genetics research with our collaborators. In fact, the experiment on highly-polymorphic regions strongly suggest that our index should be valuable in studying disease-causing mutations [9].

Acknowledgments

We wish to thank the organisers of the Variathon 2013 challenge (especially Krista Longi) for the data and for the evaluation methods. We also thank Juha Kärkkäinen for noting that the doubling step in the prefix-doubling algorithm can be done with just one join, and Eric Rivals for pointing out recombination hotspots. Publication of this article was supported by the Academy of Finland under grant 250345 (CoECGR) and the Jenny and Antti Wihuri Foundation.

References

[1] J. Sirén, N. Välimäki, and V. Mäkinen, "Indexing finite language representation of population genotypes," in *Proc. Algorithms in Bioinformatics (WABI 2011)*, ser. Lecture Notes in Computer Science, vol. 6833. Springer, 2011, pp. 270–281.

[2] M. L. Metzker, "Sequencing technologies – the next generation," *Nature Reviews Genetics*, vol. 11, pp. 31–46, 2010.

[3] P. Flicek and E. Birney, "Sense from sequence reads: methods for alignment and assembly," *Nature Methods*, vol. 6, pp. S6–S12, 2009.

[4] R. Li *et al.*, "Building the sequence map of the human pan-genome," *Nat Biotech*, vol. 28, no. 1, pp. 57–63, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1038/nbt.1596

[5] M. Burrows and D. J. Wheeler, "A block sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. 124, 1994. [Online]. Available: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html

[6] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys*, vol. 39, no. 1, p. 2, 2007.

[7] S. Heber, M. Alekseyev, S. Sze, H. Tang, and P. Pevzner, "Splicing graphs and est assembly problem," *Bioinformatics*, vol. 18, no. suppl 1, pp. S181–S188, 2002.

[8] A. Löytynoja, A. J. Vilella, and N. Goldman, "Accurate extension of multiple sequence alignments using a phylogeny-aware graph algorithm," *Bioinformatics*, vol. 28, no. 13, pp. 1684–1691, 2012.

[9] E. Khurana *et al.*, "Integrative annotation of variants from 1092 humans: Application to cancer genomics," *Science*, vol. 342, 2013.

[10] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Compressing and indexing labeled trees, with applications," *Journal of the ACM*, vol. 57, no. 1, p. article 4, 2009.

[11] V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki, "Storage and retrieval of highly repetitive sequence collections," *Journal of Computational Biology*, vol. 17, no. 3, pp. 281–308, 2010.

[12] K. Schneeberger, J. Hagmann, S. Ossowski, N. Warthmann, S. Gesing, O. Kohlbacher, and D. Weigel, "Simultaneous alignment of short reads against multiple genomes." *Genome biology*, vol. 10, no. 9, pp. R98+, Sep. 2009.

[13] C. Albers *et al.*, "Dindel: Accurate indel calls from short-read data," *Genome Research*, October 2010.

[14] L. Huang, V. Popic, and S. Batzoglou, "Short read alignment with populations of genomes," *Bioinformatics*, vol. 29, no. 13, pp. i361–i370, 2013.

[15] A. Bowe, T. Onodera, K. Sadakane, and T. Shibuya, "Succinct de Bruijn graphs," in *Proceedings of the 12th Workshop on Algorithms in Bioinformatics (WABI 2012)*, ser. LNCS, vol. 7534. Springer, 2012, pp. 225–235.

[16] U. Manber and G. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM Journal on Computing*, vol. 22, no. 5, pp. 935–948, 1993.

[17] G. Nong, S. Zhang, and W. H. Chan, "Linear time suffix array construction using D-critical substrings," in *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, ser. LNCS, vol. 5577. Springer, 2009, pp. 54–67.

[18] P. Ferragina and G. Manzini, "Indexing compressed text," *Journal of the ACM*, vol. 52, no. 4, pp. 552–581, 2005.

[19] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 378–407, 2005.

[20] P. Bille, June 2013, personal communication at Dagstuhl seminar on Indexes and Computation over Compressed Structured Data.

[21] S. J. Puglisi, W. F. Smyth, and A. H. Turpin, "A taxonomy of suffix array construction algorithms," *ACM Computing Surveys*, vol. 39, no. 2, p. 4, 2007.

[22] J. Sirén, "Compressed full-text indexes for highly repetitive collections," Ph.D. dissertation, University of Helsinki, 2012. [Online]. Available: http://urn.fi/URN:ISBN:978-952-10-8052-4

[23] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[24] "Variation Calling Challenge 2013 - Variathon v0.1," 2013. [Online]. Available: http://bioinf.dimi.uniud.it/variathon

[25] G. A. McVean *et al.*, "An integrated map of genetic variation from 1,092 human genomes," *Nature*, vol. 491, pp. 56–65, 2012.

[26] R. Horton *et al.*, "Variation analysis and gene annotation of eight MHC haplotypes: The MHC haplotype project," *Immunogenetics*, vol. 60, no. 1, 2007.

[27] S. Myers *et al.*, "A fine-scale map of recombination rates and hotspots across the human genome," *Science*, vol. 310, no. 5746, pp. 321–324, 2005.

Jouni Sirén received the MSc and PhD degrees in computer science from the University of Helsinki, Finland, in 2007 and 2012, respectively. He is currently working as a Postdoctoral Researcher at the University of Chile, Chile. His research interests include algorithm engineering, compressed data structures, and string algorithms.

Niko Välimäki obtained his PhD in computer science from the University of Helsinki, Finland, in 2012. He is currently a postdoctoral researcher in a research group working on tumor genomics. The group is part of the Finnish Center of Excellence in Cancer Genetics Research and the Department of Medical Genetics at the University of Helsinki.

Veli Mäkinen finished his PhD studies in computer science in 2003 at the University of Helsinki, Finland. He worked as a Postdoctoral Researcher (2004-2005) at Bielefeld University, Germany, and then back in Helsinki as Postdoctoral Research Fellow (2005-2007) and Academy Research Fellow (2007-2010). In 2010, he was appointed as a Professor in computer science at the University of Helsinki. Veli Mäkinen now heads the Genome-scale algorithmics research group that belongs to the Center of Excellence in Cancer Genetics Research.