BIOINFORMATICS ORIGINAL PAPER

Vol. 30 no. 24 2014, pages 3499–3505
doi:10.1093/bioinformatics/btu438

Sequence analysis Advance Access publication July 15, 2014

Journaled string tree—a scalable data structure for analyzing thousands of similar genomes on your laptop

René Rahn*, David Weese and Knut Reinert

Department of Mathematics and Computer Science, Freie Universität Berlin, Takustr. 9, 14195 Berlin, Germany

Associate Editor: Michael Brudno

ABSTRACT

Motivation: Next-generation sequencing (NGS) has revolutionized biomedical research in the past decade and led to a continuous stream of developments in bioinformatics, addressing the need for fast and space-efficient solutions for analyzing NGS data. Often researchers need to analyze a set of genomic sequences that stem from closely related species or are indeed individuals of the same species. Hence, the analyzed sequences are similar. For analyses where local changes in the examined sequence induce only local changes in the results, it is obviously desirable to examine identical or similar regions not repeatedly.

Results: In this work, we provide a datatype that exploits *data parallelism* inherent in a set of similar sequences by analyzing shared regions only once. In real-world experiments, we show that algorithms that otherwise would scan each reference sequentially can be speeded up by a factor of 115.

Availability: The data structure and associated tools are publicly available at http://www.seqan.de/projects/jst and are part of SeqAn, the C++ template library for sequence analysis.

Contact: rene.rahn@fu-berlin.de

Received on April 11, 2014; revised on June 30, 2014; accepted on July 2, 2014

1 INTRODUCTION

Next-generation sequencing (NGS) has revolutionized biomedical research in the past decade and led to a continuous stream of developments in bioinformatics, addressing the need for fast and space-efficient solutions for analyzing NGS data. Especially since the sequencing efficiency of modern NGS technologies outpaced the improvement of storage capacities, which directly leads to a growing economical issue, as storing and sharing the generated information is now bounded by the available storage and network resources (Kahn, 2011). The same technology led to the generation of comprehensive catalogs for genetic variations of the human (Durbin *et al.*, 2010; Frazer *et al.*, 2007) and other organisms as well (e.g. Auton *et al.*, 2012; Keane *et al.*, 2011). Moreover, the rapid advances in NGS made ambitious sequencing endeavors like the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2012), systematic studies of >25 000 cancerous genomes (The International Cancer Genome Consortium, 2010) or most eagerly the announced goal of the Personal Genome Project to sequence 100 000 human genomes (Ball *et al.*, 2012) possible. Those resources then provide detailed information about the genetic diversity of entire populations, which will be important to the societal health sector to acquire better understandings of the correlation between clinical conditions and phenotypes and their corresponding genotypes.

As a result, two major challenges need to be tackled. The first challenge clearly is to compress the available sequence data to relieve disk and network resources. Owing to the high redundancy of sequences originating from the same or related organism, *referential sequence compression* has been proven to be especially efficient for these kinds of data (e.g. Deorowicz and Grabowski, 2011; Kuruppu *et al.*, 2011; Pinho *et al.*, 2012; Wandelt and Leser, 2012). More recently Deorowicz *et al.* (2013) exploited cross-sequence correlations to achieve profitable compression ratios for the data of the 1000 Genomes Project.

The second challenge, however, is to devise algorithms and data structures that can handle the massive amounts of available data to incorporate those information in existing analyzing pipelines. Clearly, one solution would be to apply the algorithms sequentially to each sequence contained in the database, but the runtimes scale linearly to the number of sequences included. Thus, it is desirable to analyze the data in succinct form to archive runtimes proportional to the compressed size. This paradigm is also known as *compressive genomics* (Loh *et al.*, 2012). The FM-index (Ferragina and Manzini, 2000) and the compressed suffix array (Lippert, 2005), for example, are succinct representations of indices that can be searched efficiently. Yet, indexing thousands of genomes with these data structures would still exceed currently available memory capacities by far. In the past 5 years, this problem was subject in several publications (Huang *et al.*, 2013; Lam *et al.*, 2010; Loh *et al.*, 2012; Mäkinen *et al.*, 2009; Schneeberger *et al.*, 2009; Sirén *et al.*, 2011). Here, the focus was moved from considering each sequence individually to exploiting similarities among the sequences to reduce the overall memory footprint, and to gain substantial speedups opposed to the sequential case. Loh *et al.* (2012) presented compression-accelerated BLAST and BLAT, both tools to search patterns in a non-redundant sequence library approximatively. These methods, however, require the compressed sequence library to be generated in a computationally intensive preprocessing phase. Huang *et al.* (2013), Schneeberger *et al.* (2009) and Sirén *et al.* (2011) used as input a more general format consisting of a reference sequence and a set of variants for a collection of sequences, which is a common representation of the data produced by large sequencing endeavors such as the 1000 Genomes Project. Subsequently they built an index over the reference set exploiting the high

*To whom correspondence should be addressed.

similarities. Yet, the read-mapper BWBBLE (Huang *et al.*, 2013) was the first practical tool capable of mapping reads against thousand genomes simultaneously. In their approach, they indexed a multi-genome reference sequence with the FM-index and adapted the Burrows-Wheeler Aligner (BWA) (Li and Durbin, 2009) to work on the indexed multi-genome. To construct the multi-genome, they needed a context size to determine the sequence context left and right of genomic regions harboring insertions or deletions; thus, the entire multi-genome and the accompanied index must be reconstructed on changes of the context size.

However, there exists a plethora of algorithms for sequence analysis that are sequential in nature. That means they scan the sequences to be analyzed from left to right and perform some computation, e.g. compute an approximate matching or alignment of a query sequence to a reference sequence, or scan sequences in the context of a hidden Markov model (HMM) (e.g. De Bona *et al.*, 2008). There are many more applications such as filtering and verification algorithms in read mappers (Weese *et al.*, 2012) or searching with position-specific scoring matrices (Henikoff *et al.*, 2000; Scordis *et al.*, 1999). After reading a character of the string, the algorithm will change its internal state and possibly report some results.

Barton *et al.* (2013) gave an algorithm to solve the pattern matching problem with Hamming distance for a set of extremely similar sequences. In their model, they assumed that each sequence differs in around 10 positions to every other sequence within the set. To find all occurrences of a pattern within all sequences they searched first the reference sequence and used then some auxiliary tables to check if at the reported positions the pattern can be found for the other sequences too. Besides the impractical error model for real biological data, they did not provide any experiments to evaluate their method.

Thus, to the best of our knowledge, we provide for the first time a general solution to generically speed up a large class of algorithms when working on sets of similar strings. We will mostly address the reduction of execution time and space of such algorithms by providing a data type, called *journaled string tree (JST)* that can be traversed similarly to a simple for-loop over all sequences while exploiting the high similarity. The algorithm must simply be able to store its state when asked, continue its computation from a stored state when presented with a new character and *work locally*, i.e. when scanning a sequence its current state solely depends on a fixed-size window of last seen characters, also called context. Our approach is based on a reference-based compression of shared sequence parts with some additional bookkeeping. For example, if lets say 1 Mb of a set of 100 genomes is shared and we want to compute a semi-global alignment on all sequences, we will execute the alignment only once on this stretch of 1 Mb. For regions that exhibit differences, a corresponding sequence context is constructed on-the-fly and then examined. We can show that our approach exhibits speedups of >100 times when analyzing a set of 2185 sequences of chromosome 1 [two haplotypes of 1092 sequences from the 1000 genomes project (Altshuler *et al.*, 2010) and a reference sequence] compared with running the algorithms sequentially while using only ~3.8 GB of space. This speedup includes all overheads. The speedup in searching alone is up to a factor of 570.

2 METHODS

We will first give an informal description of the JST data structure and our traversal method using the example in Figure 3 depicting (a) three similar sequences and (b) the corresponding JST. In the example, we search for a string of length 4 using an exact string matching algorithm.

In general, instead of iterating over all strings in the set sequentially, our method *simultaneously* traverses a set of strings from left to right. With this strategy, algorithms based on sequentially scanning a sequence can be easily extended to a large set of similar sequences, while only modest memory requirements are needed. The only additional and algorithm dependent information required is the length of the so-called *sequence context*, i.e. the window of last-seen characters that solely determine the internal state of the algorithm. For example, an online algorithm that searches a query of length n with up to k errors depends on a sequence context of length $n + k$, whereas in Figure 3 the context length for exact pattern matching is the length of the query, which is 4 in this example.

In our approach, we use the reference sequence r as the anchor coordinate system and store positions, so called *branch-nodes*, at which at least one other sequence has a Δ-event, i.e. a *deletion* of a substring of the reference sequence, an *insertion* of a string relative to the reference sequence or a *replacement* of a substring of the same length between a sequence in the set and the reference sequence. For example, in Figure 3, the gray points labeled L, M, N, O are branch-nodes. The respective positions in the other sequences will be determined on-demand while scanning from left to right using a reference-based compressed representation of the sequences called *journaled strings*. We use a bitvector to denote which sequences have a Δ-event for a given reference position.

During the traversal the method constructs the required sequence contexts on-the-fly using the bitvectors and respective journaled strings and presents them to the algorithm. Whenever a branch-node is encountered we store the state of the algorithm to continue computations later from that position. Once the deviating sequence part has been processed, the algorithm will be asked to restore its last state and continue with a new character. The algorithm can in addition signal after processing a character, whether it needs the information for which sequences the presented sequence context is valid, e.g. when a string matching algorithm found a match. Figure 1 depicts the general communication processes between the Journaled String Tree traversal and a sequential algorithm.

The article is organized as follows. In Section 2.2, we describe our simple, yet fast, reference-based compression scheme; in Section 2.3, we describe how to traverse the set of strings simultaneously and apply any algorithm that sequentially streams over sequences with a limited sequence context.

2.1 Definitions

A string $s = s_0 \dots s_{n-1}$ is sequence of characters of an alphabet Σ. The length of a string is denoted as $|s| = n$. A substring of s is denoted as $s[i:j] = s_i \dots s_{j-1}$, with $0 \leq i < j \leq n$. The special case $s[i:i+1]$ will be shortened to $s[i]$.

A Δ-event is a tuple (s, i, r, j, x, Δ) with $\Delta \in \{R, I, D\}$ describing the event that occurred in s at position i relative to position j in sequence r. x denotes the string associated with this event, i.e. if $\Delta = R$, then $x = s[i : i + |x|]$ is the string replacing $r[j : j + |x|]$ and if $\Delta = I$, then $x = s[i : i + |x|]$ is the inserted string in s, otherwise if $\Delta = D$, then $x = r[j : j + |x|]$ represents the deleted string in r.

2.2 Journaled strings–compressed searchable strings

A *journaled string* λ is a referentially compressed version of a string s, which stores a pointer to a reference sequence r, an additional string *ib*, called *insertion buffer*, and a binary search tree $J(\lambda)$, called *journal tree*, over segments representing substrings of r or *ib*. We refer to such segments as *journal entries*. A *journal entry* is a tuple $e = (vp, pp, l, \sigma) \in \mathbb{N}^3 \times \{0, 1\}$,

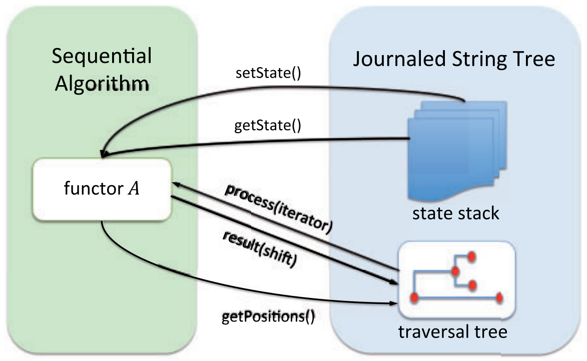

Fig. 1. Interaction between the JST and a sequential algorithm

where σ indicates the source of the corresponding segment ($0 \rightarrow r$; $1 \rightarrow ib$). We call an entry with $\sigma = 0$ an *original entry* and an entry with $\sigma = 1$ an *insertion entry*. *vp* denotes the *virtual position*, i.e. the begin position of this segment within the target sequence *s*. Accordingly, *pp* denotes the *physical position* referring to the begin position within *r*, if $\sigma = 0$, and *ib* otherwise. The length of *e* is denoted by the parameter *l*, and for any two journal entries *e* and *e′* we define $e < e' \Leftrightarrow vp < vp'$.

Generating journaled strings. Given a reference sequence *r*, a sequence *s* and *n* corresponding Δ-events sorted in ascending order, we can construct a journaled string in $\mathcal{O}(n)$ time as follows.

We create an original entry for each substring of *r* bounded by the end and begin coordinate of two adjacent Δ-events and cover each insertion by an insertion entry, while the inserted string is appended to the insertion buffer. Deletion events do not trigger the creation of a new entry but are covered by gaps between the physical end and begin position of two neighboring original entries, which can also be interleaved with insertion entries. Replacements are simply handled as an insertion followed by a deletion of the same size.

Accessing journaled strings. To randomly access the journaled string λ at a position *i*, we first search the journal tree $J(\lambda)$ for an entry (vp, pp, l, σ) with $vp \le i < vp + l$. The actual character can then be accessed at position $pp + (i - vp)$ in the corresponding substring of this entry. Hence, the random access time is $\mathcal{O}(\log |J(\lambda)|)$.

Scanning the whole journaled string λ from left to right can be realized via an in-order traversal of the journal tree (Fig. 2) in $\mathcal{O}(|\lambda| + |J(\lambda)|) = \mathcal{O}(|\lambda|)$ time, i.e. a single sequential access requires amortized $\mathcal{O}(1)$ time.

2.3 Traversing thousands of genomes

In the following section, we will give a formal description of the JST and describe the algorithm to traverse this data structure with any context-based algorithm. Given a set of strings $s^1, \ldots, s^t$ and their sets of Δ-events $D^1, \ldots, D^t$ according to a common reference sequence *r*. To reduce the space required to store the Δ-events, we collapse all events shared by a set of strings into a single structure, called the *branch-node*.

Formally, we say $u = (j, x, C, \Delta)$ is a branch-node of type Δ that occurs at branch-position *j* in the reference sequence *r*, with *x* being the corresponding string associated with the respective Δ-event and $C \subseteq \{1, \ldots, t\}$ being the set of the strings harboring this event. In the following, we will refer to *C* as the *coverage*. It holds that $k \in C \Leftrightarrow \exists_{i \in \mathbb{N}}(s^k, i, r, j, x, \Delta) \in D^k$. W.l.o.g. we assume that none of the sequences has two different Δ-events at the same position *j*, and thus, the coverages of all branch-nodes at the same branch-position *j* are disjoint. Let *label*(*u*), *pos*(*u*) and *cov*(*u*) be the values *x*, *j* and *C*, respectively, for any branch-node *u*.

A JST *T* is a data structure consisting of an array of branch-nodes $u \in V(T)$ sorted in ascending order according to their branch-position *pos*(*u*), a pointer to the common reference sequence *r* and a set of journal strings $\lambda^1, \ldots, \lambda^t$. The journaled strings are generated from the given set

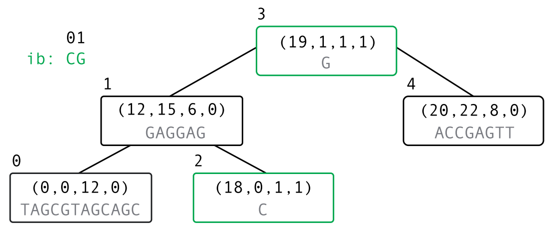

Fig. 2. Journaled string of s^1 referentially compressed against *r* from Figure 3. Original entries are drawn with black lines and insertion entries with green lines (node 2 and 3). The insertion buffer *ib* contains CG. The gray strings in each entry represent the corresponding substrings, which are not stored but depicted for clarification

of all Δ-events, and the corresponding sequences in *S* as outlined in the Section 2.2.

Basic traversal. For a sequential algorithm $\mathcal{A}$ with a context length *w*, the traversal over a JST *T* simply shifts a window over the reference sequence *r*. After every shift the current context is evaluated by an external algorithm $\mathcal{A}$, which returns a shift length to move the window to the next required context. In addition, $\mathcal{A}$ can interact with the current state of the traversal, e.g. to request the positions of all strings that are valid for the current context.

Whenever the window reaches into a branch-node $u \in V(T)$, a new subtree, whose depth depends on the context length, is branched off. We iteratively traverse this subtree and use a stack $\mathcal{S}$, which stores the current state of the traversal and $\mathcal{A}$, for backtracking. In the following, we will describe how to generate all necessary subtree information on-the-fly. First, we explain the branching strategy when the current window intersects with a branch-node and subsequently discuss step by step how to refine the subtree traversal, while maintaining the invariant that for each context explored by the traversal always a valid coverage is sustained.

Branching. To proceed the traversal in the subtree starting at the branch-node *u*, it is obvious that the left and right sequence context flanking the respective Δ-event is needed to provide $\mathcal{A}$ with all necessary information. This can be achieved by transferring the current window to any journaled string harboring this event. A representative λ^k can be simply selected from the coverage $k \in cov(u)$.

To determine in $\mathcal{O}(\log |J(\lambda^k)|)$ time where the respective Δ-event occurs in λ^k, we augment each journal entry $e \in J(\lambda^r)$ by the rank of the associated branch-node in a preprocessing step. Once the entry has been found, the begin and end position of the window in λ^k can be computed from the respective begin position of the window within the reference and the branch-position *pos*(*u*). Moreover, the traversal over the current branch is stopped whenever the window exceeds the current Δ-event represented by *u*. Figure 3 depicts this pruning of the branches depending on the window length, which is 4 in the given example.

Context-based subtree construction. So far we have only considered the simple case where the current branch is solely dependent on the branch-node it originates from. However, depending on the context length and the distribution of the branch-nodes, it might happen that multiple branch-nodes affect the current branch, resulting in an expanding subtree. For example, the branch-node *O* in Figure 3 induces a split of the branch coming from *N*. We will refer to such branch-nodes inducing a split in the current branch as *split-nodes* to distinguish them from their original meaning. Accordingly, the node *O′* in Figure 3 represents the split-node induced by *O*.

Let *u* be the branch-node of the current branch and *v* its successor. Then *v* induces a split in the branch coming from *u* if the following conditions are satisfied:

(1) $cov(u) \cap cov(v) \neq \varnothing$.

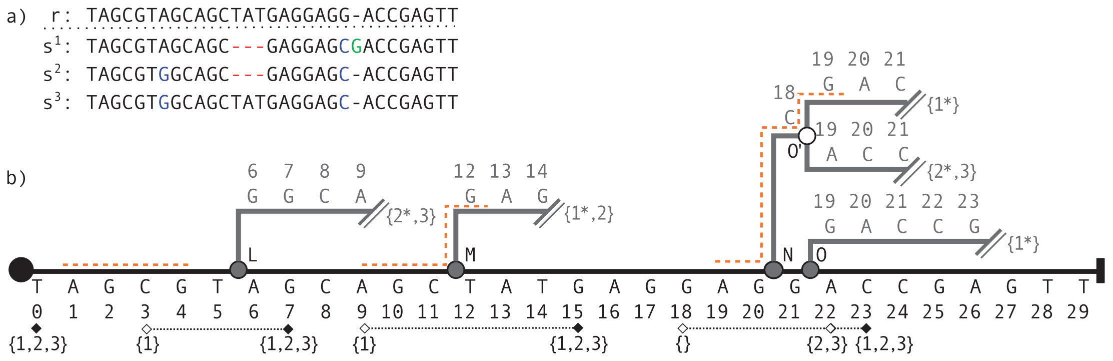
a)
r: TAGCGTAGCAGCTATGAGGAGG-ACCGAGTT
s1: TAGCGTAGCAGC---GAGGAGCGACCGAGTT
s2: TAGCGTGGCAGC---GAGGAGC-ACCGAGTT
s3: TAGCGTGGCAGCTATGAGGAGC-ACCGAGTT
b)
L
M
N
O
O'
{2*,3}
{1*,2}
{1*}
{2*,3}
{1*}
{1,2,3}
{1}
{1,2,3}
{1}
{1,2,3}
{}
{2,3}
{1,2,3}

Fig. 3. Online search of the pattern $p = AGCG$ over a JST depicted in (**b**) representing the four aligned sequences in (**a**), where r is chosen as the reference. The window length is 4. The bullets on the black line represent the branch-nodes: $L = (6, G, \{2, 3\}, R)$; $M = (12, TAT, \{1, 2\}, D)$; $N = (21, C, \{1, 2, 3\}, R)$; $O = (22, G, \{1\}, I)$. Below is plotted the reference with the reference coordinates. Branches are indicated by gray lines labeled with the respective sequence context depending on the window length and with the virtual positions of the chosen proxy (marked sequence number). The sets at the end denote the coverage. O' is a split-node induced by O in the branch of N. The dotted lines capped with diamonds below the reference positions represent the ranges in which invalid paths are suppressed. The orange (dashed) lines represent matches of the pattern p

(2) $cov(u) \cap cov(v) \subset cov(u)$.

(3) The mapped virtual position of u falls into the current window.

The first and the second conditions ensure that there exists a subset of strings covering v that also covers u. Otherwise, the contexts for all strings covering u are either aware of the Δ-event represented by v or none of the strings harbor the respective Δ-event. The third condition checks that the current context is affected by the split-node.

The current branch is split if all conditions are fulfilled, and hence, the coverage of the active branch is partitioned into the sets $cov(u) \cap cov(v)$ and $cov(u) \backslash cov(v)$. Now the traversal is continued with the coverage set containing k, whereas the remaining set is stored on the stack $\mathcal{S}$ together with the current states of the traversal and $\mathcal{A}$.

Suppressing invalid sequences. In addition, we keep track of the coverages of all branch-nodes that currently fall into the window over the reference sequence, as well as the coverages of replacements and deletions that were encountered previously and still affect the current window. For example, the deletion represented by branch-node M in Figure 3 also affects all windows starting at position 13 and 14 in r (denoted by the dashed lines below the tree). Whenever $\mathcal{A}$ requests for it, we on-demand compute the sequences the context is valid for using the auxiliary information. We also use this information to skip windows that have been examined already. In Figure 3, for example, if the algorithm finished the branch at branch-node N and passes to the next branch-node O, it is clear that all windows beginning before the insertion in O have been searched already when processing N, as all sequences covering O also cover N. Hence, it is sufficient to move the context directly to the beginning of the insertion.

Processing blocks. The traversal over the JST can be conducted blockwise. To do so, we partition the set of branch-nodes into blocks of size $B > w$, according to their positions within the reference sequence. Whenever we enter a new block, we load it into memory and discard it after processing all contained branch-nodes. In general, no more than two blocks are kept in memory at the same time.

Dynamic updates. It is possible to add k sequences to the JST dynamically. To do so, the k sequences are represented as k single-sequence JSTs, i.e. each hosts one sequence only, referring to the common reference sequence. Then, the JSTs can be merged into the existing JST in $\mathcal{O}(nk\log k)$ time, where n is the size of the largest branch-node array using a $(k+1)$-way merge step. During the merge, we either add the index of the currently processed sequence to the coverage set if the branch-node already exists or insert a new branch-node at the corresponding position. In a new traversal, the journaled strings for the recently added sequences are then constructed on-demand as described in the previous sections.

3 RESULTS

3.1 Implementation

We implemented all data structures, programs and functions in SeqAn, the C++ software library for sequence analysis (Döring *et al.*, 2008).

To memory efficiently represent the coverage information, each branch-node stores a packed bitvector. The binary search tree of the journaled string is realized as a sorted array, as the variants are incorporated in left-to-right scan manner. We used binary search to look up a given entry within the array.

To make our JST data structure applicable to a wide range of algorithms, we implemented a templatized `Finder` object, which hides the traversal and the state management from the outer interfaces. The user can simply plug-in its own functor to this object and implement the functions `getState()` and `setState()` if necessary.

As an example, we provide functors for the *naive*, Horspool (1980), Shift-And and Shift-Or (Baeza-Yates and Gonnet, 1992) exact pattern search algorithms, as well as Myers' bitvector algorithm for approximate search (Myers, 1999). To evaluate running times and memory consumptions, we implemented a benchmark application, which is available on request to the authors. Furthermore, we have implemented a tool to transform a vcf file into our own genome delta format called *gdf*. This format simply stores the variants in a compact form and the respective bit vectors for each variant (Deorowicz *et al.*, 2013).

3.2 Experiments

We used data from the 1000 Genomes Project (The 1000 Genomes Project Consortium, 2012) to evaluate our traversal over the JST. We took the vcf file for chromosome 1 and transformed it into a gdf file for different sets of sequences. The gdf file storing 1092 genotypes without the reference sequence used only 446 MB instead of 87 GB needed for the vcf file. Note that the original size of chromosome 1 in fasta format is nearly 250 MB. In fact, we could compress the gdf file with gzip to 58 MB in only 14 s, suggesting that further studies of the compressibility of this format would yield good compression ratios.

Table 1. Timings and memory consumption for approximate string matching with Myers' bitvecor algorithm and a pattern of size 64

Number of sequences	Number of variants	Total time			Search time			Memory consumption	
		JST (s)	Naive (s)	Factor	JST (s)	Naive (s)	Factor	JST (GB)	Naive (GB)[a]
1	0	6.56	5.58	0.85	3.38	2.88	0.85	0.63	0.42
2	291 435	6.50	11.17	1.72	3.43	5.46	1.59	0.68	0.68
33	777 948	9.10	184.15	20.24	4.46	90.21	20.23	0.80	8.66
65	894 324	10.30	362.80	35.22	4.74	177.63	37.47	0.86	17.12
129	1 023 137	13.36	719.50	53.85	5.07	352.46	69.52	0.99	26.34
257	1 316 377	18.66	1434.70	76.89	5.83	702.12	120.43	1.15	26.34
513	2 106 043	33.04	2863.90	86.67	7.74	1401.44	181.06	1.54	26.34
1093	2 993 910	76.61	6101.94	79.65	10.58	2985.83	282.21	2.58	26.34
2185	2 993 758	106.46	12 198.38	114.59	10.34	5968.85	577.25	3.72	26.34

Note. Naive refers to the sequential processing of fasta sequences.
[a]At most 100 fasta sequences were loaded into memory at a time.

Table 2. Timings and memory consumption for exact string matching with the Horspool algorithm and a pattern of size 64

Number of sequences	Number of variants	Total time			Search time			Memory consumption	
		JST ()	Naive (s)	Factor	JST (s)	Naive (s)	Factor	JST (GB)	Naive (GB)[a]
1	0	3.74	3.48	0.93	0.51	0.45	0.88	0.63	0.42
2	291 435	4.28	6.68	1.56	0.72	0.90	1.25	0.68	0.68
33	777 948	6.38	111.64	17.50	1.24	16.52	13.32	0.80	8.66
65	894 324	7.04	219.95	31.24	1.41	32.17	22.82	0.86	17.12
129	1 023 137	9.91	436.42	44.04	1.61	63.46	39.42	0.99	26.34
257	1 316 377	15.44	869.29	56.30	2.07	126.03	60.88	1.15	26.34
513	2 106 043	27.95	1735.18	62.08	3.19	251.18	78.74	1.54	26.34
1093	2 993 910	63.30	3696.95	58.40	5.49	534.73	97.40	2.58	26.34
2185	2 993 758	93.55	7390.51	79.00	5.15	1068.78	207.49	3.72	26.34

Note. Naive refers to the sequential processing of fasta sequences.
[a]At most 100 fasta sequences were loaded into memory at a time.

Although we made all analyses based on the gdf format, we could also process vcf files directly at the expense of longer I/O times. Reading the complete gdf file of chromosome 1 containing 2 993 910 variants for 1092 individuals took merely 4.2 s. For the runtime and memory evaluation, we generated different datasets from the vcf file representing the genotypes of 1, 32, 64, 128, 256, 512 and 1092 sequences and one set with 2184 sequences generated from both haplotypes of the 1092 sequences and additionally included the reference sequence itself. We performed the traversal on each set with the exact online-search algorithm Horspool and with Myers' bitvector algorithm for approximate pattern search. The experiments were conducted on a Debian GNU/Linux machine with 72 GB of RAM and a 2.67 GHz Intel(R) Xeon(R) CPU X5650 processor. The presented results for the JST are obtained with a block size of 100 000.

Running time. The total running times are depicted in the columns under the respective heading in Table 1 for Myers' algorithm and in Table 2 for the Horspool algorithm. The total running time for the JST search includes loading the reference sequence and the gdf file as well as the generation of the journaled strings and the search itself. We compared the total running time with the sequential case, where we load at most 100 fasta sequences at a time into memory and then sequentially searched over the sequences. Except for the single reference search (the JST is empty) in row 1, the JST traversal is faster than the sequential processing for the same number of sequences for both algorithms. The *factor* column represents the speedup of the JST over the sequential search. The figures clearly reveal, that the more sequences are added the more the speedup grows, albeit there is a little decrease for 1093 sequences compared with 513 sequences. In addition, it can be seen that the running times for the Horspool algorithm are in general faster than for Myers' bitvector algorithm, while the speedup over the sequential case is smaller.

Additionally, we measured the search time without any generation or loading times and compared again the running times of the JST traversal with the sequential search. All figures for the search time are presented in Tables 1 and 2 too. For two sequences, our approach is already faster than the sequential case. Moreover, the speedup for Myers' bitvector algorithm increases almost continuously up to a factor of 577 for 2185 sequences. Again, the running times for the Horspool algorithm are less than for Myers' bitvector algorithm, but the speedup is higher for the latter one.

Memory consumption. The last two columns of Tables 1 and 2 present the memory usage of both strategies. The measured memory consumptions are identical for both algorithms. Again we can see the same behavior as for the running times. In the sequential case, the memory consumption for searching solely the reference is slightly less than for the JST. However, with two sequences both methods use roughly the same amount of memory. For more sequences, the JST memory consumption increases only sublinear in the size of the contained sequences. The analysis of 2185 sequences requires 3.72 GB of RAM.

Different block sizes. In addition to the block size of 100 k, we tested our approach with blocks of length 500 k and compared the performance with a version that loads all variants at once in memory. Clearly, the memory consumption increases with higher block sizes. One thousand ninety-three sequences require 4.73 GB of RAM with a block size of 500 k, and 17.06 GB if all variants are stored in memory. Thus, even for the setting with the largest memory consumptions, our approach stores 1093 sequences within 6% of the space the naive variant would require to store all sequences simultaneously. In addition, the measured running times for the 100 k blocks are slightly lower than using larger blocks.

Different pattern sizes. The JST traversal strongly depends on the number of variants contained in the set because the more branch-points are available the more often the algorithm needs to split branches and update coverages. We simulated this behavior by increasing the size of the pattern and searching with each pattern the set of 1093 sequences. We measured only the search times because the generation times are not affected by the pattern size. Figure 4 depicts the behavior of the search time for different pattern sizes. The blue line represents the search with the Horspool algorithm. The search time increases slowly to 13.78 s for a pattern of size 256. The red line represents Myers' bitvector search. Here we can see a major increase when increasing the pattern size from 64 to 128. For patterns larger than the word size, which is 64 bits in our case, the algorithm has a more complex procedure, which results in longer running times. Yet, the search time of 55.21 s for a pattern of size 256 is still 55 times faster than searching the sequences sequentially.

4 DISCUSSION AND CONCLUSION

In this article, we presented a data structure called the JST, to represent a set of sequences as a set of variants based to a common coordinate system given by a common reference sequence and a set of accompanied bitvectors. We used searchable referentially compressed strings, called journaled strings, to generate a succinct representation for any sequence from this set.

Furthermore, we implemented a traversal over the JST, while searching regions shared by a subset of the sequences simultaneously. Moreover, we loaded parts of the journaled strings dynamically on-demand to decrease the memory consumptions tremendously, while the running time remained unchanged. During the traversal, we built only those parts of the JST that were necessary to evaluate the current context. These methods work dynamically for any window length and do not need any preprocessing phase.

The results show that any algorithm that processes an input sequence with a given window length can greatly benefit from our approach. We tested our approach on different-sized sets of chromosome 1 sequences generated from the vcf file of the 1000 Genomes Project. All our experiments showed a better performance and memory consumptions compared with the sequential processing of the same number of sequences in fasta format. The only case our approach was slower and used slightly more memory was when we analyzed the reference sequence only. For two sequences we already measured a gain over the sequential case. We analyzed the search and the total times for an exact pattern search with the Horspool algorithm and an approximate pattern search with Myers' bitvector algorithm.

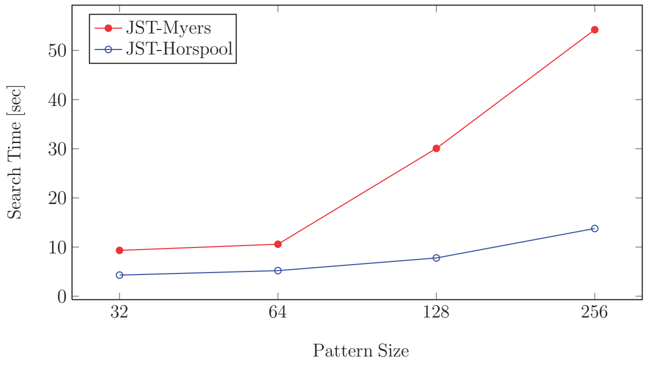

Fig. 4. JST search time depending on the pattern size over 1093 genotypes of chromosome 1

Comparing the total times for different number of genomes revealed a growing speedup factor opposed to the sequential running time, except when performing the search on 1093 sequences. Here the speedup factor dropped a little bit compared with its previous trend. The same drop cannot be seen when comparing the respective search times for both algorithms, and hence, it must be a result of longer generation times for the journaled strings. This seems to be plausible, as when increasing the number of the sequences the number of variants contained in the set also increased. The figures in the second column of Table 1 and 2 show the number of all variants contained in this set. For 1093 sequences, there were almost 1 million more variants contained in the set than for 513 sequences. This fact is supported by adding the running times for 2185 sequences to the comparison. We used both haplotypes of the 1092 sequences contained in the vcf file to generate the 2184 sequences plus the reference sequence. Hence, the number variants remains the same, while the number of sequences is again doubled. Here the speedup factor starts to grow again, with almost the same factor as it did before.

We observed that loading variations block-wise with a block size of 100 k not only lowers the expected memory consumption but also shows slightly better running time results. This behavior can be explained by considering the binary lookup necessary to find the journal entry representing a branch-node. If the set to be searched is smaller, then the lookup will be faster. Additionally, generating journaled strings for larger block sizes results in higher memory consumptions, and hence, caching effects can slow down the overall generation time.

We showed that our method scales in time as well as memory usage well with the number of sequences and the number of variants contained in the set. We gained speedups for 2184 sequences up to a factor of ∼577 for the search time and ∼115 for the total running time compared with the sequential case, while we used only 3.72 GB of RAM. Motivated by the low memory consumptions, we used an Apple MacBook Pro with 8 GB main memory to search the set of 1093 sequences with Myers' bitvector algorithm and a pattern of size 64. This took only 78.51 (9.4) s in total (search only) while using ∼2.2 GB of RAM.

Currently, we are extending the traversal to support multi-core parallelism. We will implement and test two different core-parallel approaches: parallelizing the traversal over the branch-nodes and splitting the search space over the reference into multiple chunks. In the first version, we implement a single-producer-multiple-consumer strategy, where one producer thread iterates over the branch-nodes and adds the states necessary to conduct a valid traversal over the branches to a thread-safe queue. A team of consumer threads can then dequeue the states independently from the queue and traverse the preceding reference segment starting from the predecessor node to the current branch-node and the branch itself.

In the second strategy, we divide the reference sequence into multiple chunks and let a team of threads process untouched chunks concurrently. To do so, we need a preprocessing step over the set of branch-nodes, which resolves possible conflicts between the coverage sets of two neighboring chunks. Imagine a deletion at the end of a chunk that also affects the first positions of the adjacent chunk right of it. If two threads process these two chunks in parallel, then the thread working on the affected chunk might report wrong results because its active coverage set does not reflect the deletion beginning in the previous chunk.

Furthermore, we will modify the journaled strings to represent single-nucleotide polymorphisms more efficiently, as they account for the largest number of variants available in public databases. This will further lower the memory consumption.

Given the promising results of our approach, we will extend the set of functors to allow more complex algorithms, e.g. filter and verification algorithms, and integrate them into existing tools such as read mappers and variant caller to make them applicable to large collections of sequences. Moreover, our approach would allow to immediately incorporate new reference sequences into the existing set or to assemble domain-specific reference sets dynamically from an external pool, opening the door to tailored medical applications.

Funding: This work was supported by the Deutsche Forschungsgemeinschaft [1712/4-1, Algorithmic engineering for high throughput sequencing to R.R.] and by the Federal Ministry of Education and Research [16V0080 to D.W.].

Conflict of Interest: none declared.

REFERENCES

Altshuler,D. *et al.* (2010) A map of human genome variation from population-scale sequencing. *Nature*, **467**, 1061–1073.

Auton,A. *et al.* (2012) A fine-scale chimpanzee genetic map from population sequencing. *Science*, **336**, 193–198.

Baeza-Yates,R. and Gonnet,G.H. (1992) A new approach to text searching. *Commun. ACM*, **35**, 74–82.

Ball,M.P. *et al.* (2012) A public resource facilitating clinical use of genomes. *Proc. Natl Acad. Sci. USA*, **109**, 11920–11927.

Barton,C. *et al.* (2013) Querying highly similar sequences. *Int. J. Comput. Biol. Drug Des.*, **6**, 119–130.

De Bona,F. *et al.* (2008) Optimal spliced alignments of short sequence reads. *Bioinformatics*, **24**, i174–i180.

Deorowicz,S. *et al.* (2013) Genome compression: a novel approach for large collections. *Bioinformatics*, **29**, 2572–2578.

Deorowicz,S. and Grabowski,S. (2011) Robust relative compression of genomes with random access. *Bioinformatics*, **27**, 2979–2986.

Döring,A. *et al.* (2008) SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, **9**, 11.

Durbin,R.M. *et al.* (2010) A map of human genome variation from population-scale sequencing. *Nature*, **467**, 1061–1073.

Ferragina,P. and Manzini,G. (2000) Opportunistic data structures with applications. In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. FOCS'00, pp. 390–398.

Frazer,K. *et al.* (2007) A second generation human haplotype map of over 3.1 million SNPs. *Nature*, **449**, 851–861.

Henikoff,J.G. *et al.* (2000) Increased coverage of protein families with the blocks database servers. *Nucleic Acids Res.*, **28**, 228–230.

Horspool,R.N. (1980) Practical fast searching in strings. *Softw. Pract. Exper.*, **10**, 501–506.

Huang,L. *et al.* (2013) Short read alignment with populations of genomes. *Bioinformatics*, **29**, i361–i370.

Kahn,S.D. (2011) On the future of genomic data. *Science*, **331**, 728–729.

Keane,T.M. *et al.* (2011) Mouse genomic variation and its effect on phenotypes and gene regulation. *Nature*, **477**, 289–294.

Kuruppu,S. *et al.* (2011) Optimized relative lempel-ziv compression of genomes. In: *Proceedings of the Thirty-Fourth Australasian Computer Science Conference*. Vol. 113, ACSC'11, pp. 91–98.

Lam,T. *et al.* (2010) Indexing similar dna sequences. In: Chen,B. (ed.) *Algorithmic Aspects in Information and Management*. Vol. 6124 of *Lecture Notes in Computer Science*. Springer, Berlin Heidelberg, pp. 180–190.

Li,H. and Durbin,R. (2009) Fast and accurate short read alignment with burrows–wheeler transform. *Bioinformatics*, **25**, 1754–1760.

Lippert,R. (2005) Space-efficient whole genome comparisons with burrows-wheeler. *J. Comput. Biol.*, **12**, 407–415.

Loh,P.-R. *et al.* (2012) Compressive genomics. *Nat. Biotechnol.*, **30**, 627–630.

Mäkinen,V. *et al.* (2009) Storage and retrieval of individual genomes. In: Batzoglou,S. (ed.) *Research in Computational Molecular Biology*. Vol. 5541 *of LNCS*. Springer, Berlin Heidelberg, pp. 121–137.

Myers,E.W. (1999) A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, **46**, 395–415.

Pinho,A.J. *et al.* (2012) GReEn: a tool for efficient compression of genome resequencing data. *Nucleic Acids Res.*, **40**, e27.

Schneeberger,K. *et al.* (2009) Simultaneous alignment of short reads against multiple genomes. *Genome Biol.*, **10**, R98.

Scordis,P. *et al.* (1999) Fingerprintscan: intelligent searching of the prints motif database. *Bioinformatics*, **15**, 799–806.

Sirén,J. *et al.* (2011) Indexing finite language representation of population genotypes. *Algorithms Bioinformatics*, **6833**, 270–281.

The 1000 Genomes Project Consortium. (2012) An integrated map of genetic variation from 1,092 human genomes. *Nature*, **491**, 56–65.

The International Cancer Genome Consortium. (2010) International network of cancer genome projects. *Nature*, **464**, 993–998.

Wandelt,S. and Leser,U. (2012) Adaptive efficient compression of genomes. *Algorithms Mol. Biol.*, **7**, 30.

Weese,D. *et al.* (2012) Razers 3: faster, fully sensitive read mapping. *Bioinformatics*, **28**, 2592–2599.