RCSI: Scalable similarity search in thousand(s) of genomes

Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser
Humboldt-Universität zu Berlin, Wissensmanagement in der Bioinformatik,
Rudower Chaussee 25, 12489 Berlin, Germany
{wandelt, starling, bux, leser}@informatik.hu-berlin.de

ABSTRACT

Until recently, genomics has concentrated on comparing sequences between species. However, due to the sharply falling cost of sequencing technology, studies of populations of individuals of the same species are now feasible and promise advances in areas such as personalized medicine and treatment of genetic diseases. A core operation in such studies is read mapping, i.e., finding all parts of a set of genomes which are within edit distance k to a given query sequence (k-approximate search). To achieve sufficient speed, current algorithms solve this problem only for one to-be-searched genome and compute only approximate solutions, i.e., they miss some k-approximate occurrences.

We present RCSI, Referentially Compressed Search Index, which scales to a thousand genomes and computes the exact answer. It exploits the fact that genomes of different individuals of the same species are highly similar by first compressing the to-be-searched genomes with respect to a reference genome. Given a query, RCSI then searches the reference and all genome-specific individual differences. We propose efficient data structures for representing compressed genomes and present algorithms for scalable compression and similarity search. We evaluate our algorithms on a set of 1092 human genomes, which amount to approx. 3 TB of raw data. RCSI compresses this set by a ratio of 450:1 (26:1 including the search index) and answers similarity queries on a mid-class server in 15 ms on average even for comparably large error thresholds, thereby significantly outperforming other methods. Furthermore, we present a fast and adaptive heuristic for choosing the best reference sequence for referential compression, a problem that was never studied before at this scale.

1. INTRODUCTION

Since the release of the first human genome [8], the cost for sequencing has rapidly decreased. As of now, the price is at approx. 2,000 USD per genome and is expected to fall further once third generation sequencing techniques become available [40]. In contrast to previous years, where typically only one individual of a species was sequenced (like humans, mice, E.coli etc.), the decrease in cost makes it possible to sequence large samples of a given population. Such studies, especially on humans, are interesting from many perspectives, such as correlation of specific mutations to the risk of developing a disease, to fine-tuned dosages of therapies, or simply to better understand the relationship between genotype and phenotype. For instance, the 1000 Genomes Project sequenced 1,092 human genomes to better understand population dynamics [2]; the International Cancer Sequencing Consortium is currently sequencing 50.000 human genomes to study the genetic basis of 25 types of cancer [9]; and the UK-10K project is sequencing 10.000 British individuals to better understand the impact of rare genetic mutations[1].

Studies at this scale use next generation sequencing (NGS) [40]. A property of NGS is that the sequences (reads) that are directly measured by the device are shorter (length in total of 30-200 base pairs) than with traditional Sanger sequencing, yet there are many more (hundreds of millions). Due to the highly repetitive nature of the human genomes, such short reads cannot be assembled to individual genomes; instead, the position of each read within a genome is typically determined by mapping the read against a single reference genome [20]. This problem is called read mapping or k-approximate search: Given a genome and a (short) read, find all substrings in the genome with an edit distance below k to the read, where typical values for k lie in the order of 1-2% of the read length, i.e., in the range of 1-3. k-approximate search is different from the classical bioinformatics problems of global alignment (measuring the difference between two entire sequences) or local alignment (finding maximally similar subsequences in the query and the to-be-searched sequence) [19], but of utmost importance when dealing with modern sequencing techniques [27]. A large number of algorithms for solving this problem appeared in recent years [26–28]. To scale to hundreds of millions of reads, all these algorithms compute an index (typically q-grams or variations of suffix trees) of the reference genome and perform several heuristic pruning tricks during the search, thus trading accuracy for time [5]. An inherent problem of this approach is the dominance of the reference sequence: Variations of individuals are defined by comparison to an arbitrarily chosen other individual. This is a severe constraint without any biological justification [41]; it merely exists for pure technical reasons, as no algorithm yet exists that can efficiently map reads against large sets of genomes.

In this paper, we present the Referentially Compressed Search Index (RCSI), which follows a radically different approach to the k-approximate search problem and scales to a thousand genomes. Given a set of to-be-searched genomes G, RCSI first selects a reference $r \in G$. Next, it uses a referential compression algorithm [45] to compress all genomes in G with respect to r. The intuition of a referential compression is to encode substrings of a to-be-compressed

[1] See `http://www.uk10k.org/`

Proceedings of the VLDB Endowment, Vol. 6, No. 13

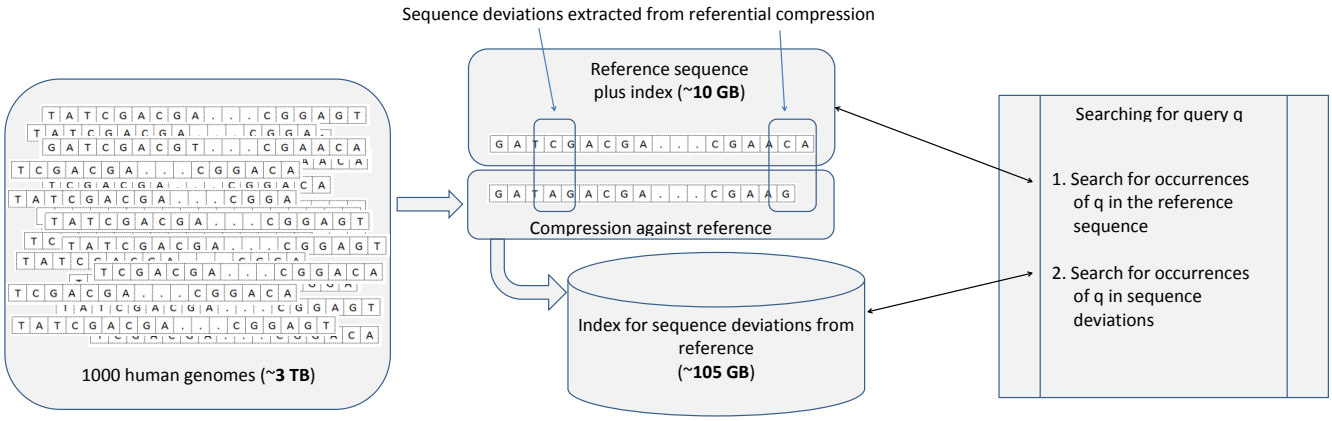


Figure 1: Overview of our Referentially Compressed Search Index.

string as positional references into the reference. This compression is lossless, can be computed quickly, and yields very high compression rates if applied to genomes of the same species; for instance, two randomly selected human genomes are approx. 99% identical [38]. The resulting data structure is a space-efficient representation of all common subsequences in G and of all differences. RCSI uses this data structure to solve the k-approximate search problem for all genomes in G with a single search. Conceptually, RCSI has to search in two data structures per indexed genome: those parts that are identical to r and those parts that are different from r; both parts may vary considerably between different genomes. We show how each of these types of information can be represented in a singleton data structure across all genomes by using a compressed suffix tree for the commonalities and another compressed suffix tree for the differences. Both data structures together allow solving the read mapping problem against multiple genomes efficiently and exactly. Our approach has the additional advantage that the index grows only very slowly with more and more genomes, i.e., RCSI scales very well with increasing data sets. The general idea of RCSI is depicted in Figure 1.

We evaluate RCSI on a set of 1,092 genomes recently released from the 1000 Genomes Project, a set that is more than 100 times larger than the data used for evaluation of similar methods within the last 13 months [30, 49]. RCSI compresses these 3 TB of raw sequence data down to a 115.7 GB (factor of 26:1) compressed search index. Using this index, RCSI answers k-approximate queries over 1,092 chromosomes in 1-20 ms on a laptop, and over 1,092 complete human genomes in 0.02-15.29 ms on a mid-sized server. Already for much smaller datasets, RCSI outperforms its competitors by a factor of at least 7.

RCSI always computes the exact answer to a given k-approximate search, regardless of which genome was chosen as reference for the compression. However, the achieved compression rates and the performance of searching do vary with different references. Therefore, we also study the problem of choosing the best reference for compression, i.e., the genome for which the compressed index for all genomes in G is the smallest. We present an efficient heuristic for this problem using partial compressions. This heuristic shows considerable improvement compared to a random selection strategy and is almost as good as the perfect reference selection (computed exhaustively on a sample), while being orders of magnitude faster.

Altogether, our paper makes the following contributions:

- We present a novel compression algorithm and highly tuned data structure to efficiently create compact, yet lossless representations of genomes with respect to a given sequence.
- We describe efficient algorithms operating on the compressed search index for answering k-similarity queries.
- We, for the first time, study the problem of selecting the best reference for referential compression and similarity search among a set of genomes and present an efficient and effective heuristic for solving this problem.
- We evaluate all algorithms on a set of 1092 genomes, a data set that is highly realistic already today, yet far greater than data sets used in previous studies.

The remainder of this paper is structured as follows: In the next section we present related work. We introduce the problems of similarity search and of referential compression in Section 3. Section 4 describes our novel algorithm and data structure, RCSI, in detail. Section 5 is devoted to the problem of finding the best reference. Our algorithms are evaluated in Section 6, and Section 7 concludes the paper.

2. RELATED WORK

Approximate search in strings (or sequences; we use both terms synonymously) has a long tradition in computer science [34]. If the to-be-searched string is large or many strings need to be searched, index-based methods provide the best performance. Many index structures have been proposed, such as suffix trees [15], suffix arrays [31], n-gram indexes [43], or prefix trees [39]. Besides the k-approximate search studied in this paper, other variations of the problem have been studied intensively, such as approximate dictionary matching [7], searching probabilistic strings [29], searching with generalized distance functions [50] or searching with regular expressions instead of edit-distance constraints [23]. Similarity search over strings is fundamentally important for bioinformatics due to the fact that DNA and proteins can, for many applications, be represented as long strings; in this area, local alignment search is particularly important [3, 21]. The literature on string similarity search in general is vast and cannot be summarized here; we refer the reader to several excellent surveys [14, 32].

The k-approximate search problem is important in various applications, such as entity extraction [48] or pattern matching in time series [17]. In recent years, its importance in bioinformatics has grown tremendously because novel sequencing machines produce

much shorter (yet many more) reads than the previous generation. Those reads cannot be assembled themselves into genomes. Instead, researchers map them to a given reference genome, which suffices to find mutations and variations and thus potential genetic predispositions of certain phenotypes or genetic diseases. This led to the development of a large number of algorithms for this so-called read mapping problem (e.g. [26–28]), which mostly build on n-gram hash indexes or on some variation of suffix trees or arrays. All these algorithms utilize thresholds for the number of allowed errors, and all perform some form of heuristic pruning to achieve high speed by sacrificing accuracy [5]. Furthermore, all of them map to a single reference, which is highly problematic from a biological point of view [41].

Our work uses a radically different approach. We first compress the set of genomes to be searched with respect to a reference genome. Next, we build an index over these compressed representations which is searched at query time. The idea of using compression as key idea to speed up string matching is not entirely new; recently, a number of works on this topic appeared almost concurrently. In [46], we describe an algorithm for exact search over a single compressed genome; here, we improve on this work by studying the much more important approximate case and by extending the search to multiple genomes. [30] sketches a prototype implementation for executing the popular search algorithm BLAST on compressed genomes; however, this method also gives-up on accuracy for being faster. Its scalability is unknown, as tests were only reported on small sets of small organisms (genomes less than 100 MB in size). The short-read aligner GenomeMapper [41] maps multiple reads simultaneously to multiple genomes using hash-based graph data structures for the reads and the genomes; thus, it actually studies a more general problem than we do (we always map only one read), but it cannot scale beyond a dozen genomes (Korbinian Schneeberger, personal communication). The most similar work to ours is GenomeCompress [49], though the proposed algorithm is considerably different. First, Yang et al. compress genomes using an alignment technique, while we greedily search for longest matches (see Section 3). Accordingly, the information encoded in the compressed genomes is quite different, leading to different search and indexing techniques. We will compare our method to this work in more detail in Section 6.5.

To search a large set of genomes as we do, one could also use a conventional method and build a separate index over each genome. Besides being highly space demanding (the size of a suffix tree for a sequence is 3-5 larger than the sequence itself [44], which implies that such an index structure for 1000 genomes would require 9-15 TB), searching these indexes in parallel for optimal performance would require significant investments in hardware. In contrast, our algorithm searches the same number of genomes in far less than a second on a modestly strong desktop computer.

There are also other interesting works not explicitly using genome compression. [15, 33] use specialized suffix trees for indexing collections of highly repetitive sequences, thus implicitly performing a sort-of compression by representing common parts only once. Tests were performed on a data set of 500 MB only, and it remains unclear if the space consumption of the suffix tree would be manageable for a data set like ours (6,000 times larger). [24] describes a self-index on LZ77 compressions of highly repetitive collections. The authors evaluate their approach on 37 DNA sequences of S. Cerevisiae, which sum up to an uncompressed size of merely 440 MB. In [42], multiple alignments of individual genomes are converted into a finite automaton and indexed with an extension of Burrows-Wheeler transform. The method is evaluated on four human chromosomes 18 (each one around 75 MB in size). Finally, [36] also uses reference sequences to speed up global or local alignment of a query, but does not work with compression.

There is also some more theoretical work on searching over compressed string representations. [16] studies the usage of grammars and LZ77 parsing for compression of similar sequence collections and improves complexity bounds with respect to space as well as time. Complexity bounds on searching LZ77 are also studied in [13]. Neither of these papers provide a practical system or experimental evaluation.

3. FOUNDATIONS

In this section, we formally introduce the k-approximate search problem over collections of large strings, and present the compression algorithm we use for RCSI.

3.1 k-approximate Search

In this work, a *sequence* s is a finite string[2] over an alphabet Σ. The length of a sequence s is denoted with $|s|$ and the subsequence starting at position i with length n is denoted with $s(i,n)$. $s(i)$ is an abbreviation for $s(i,1)$. All positions in a sequence are zero-based, i.e., the first character is accessed by $s(0)$. The concatenation of two sequences s and t is denoted with $s \circ t$. A sequence t is a *prefix of* a sequence s, if we have $s = t \circ u$, for a sequence u. A sequence s is a *subsequence* of sequence t, if there exist two sequences u and v (possibly of length 0), such that $t = u \circ s \circ v$.

DEFINITION 1. *Given two sequences s and t, s is* k-approximate similar to *t, denoted $s \sim_k t$, if s can be transformed into t by at most k edit operations. Edit operations are: replacing one symbol in s, deleting one symbol from s, and adding one symbol to s. Given a sequence s and a sequence q, the set of* all k-approximate matches in s with respect to *q, denoted $search(s)_q^k$, is defined as the set $search(s)_q^k = \{(i, s(i,j)) \mid s(i,j) \sim_k q\}$.*

DEFINITION 2. *A* sequence database *S is a set of sequences $\{s_1, ..., s_n\}$. Given a query q and a parameter k, we define the* set of all k-approximate matches for *q* in *S as*

$$DBsearch(S)_q^k = \{(l, search(s_l)_q^k) \mid s_l \in S\}.$$

EXAMPLE 1. *It holds that $ATCGG \sim_1 ATGG$, because the symbol C can be removed from $ATCGG$ with one delete operation to obtain $ATGG$. If $S = \{s_1, s_2, s_3\}$, with $s_1 = ACACTG$, $s_2 = ACTGA$, $s_3 = GGCTA$, we have*

$$DBsearch(S)_{CGA}^1 = \{(1, \{(1, CA)\}), (2, \{(1, CTGA), (2, TGA), (3, GA)\}), (3, \{(2, CTA)\})\}.$$

3.2 Indexing

Since we have several sequences in the sequence database $S = \{s_1, ..., s_n\}$, one can either build one index for each sequence or one combined index. A simple way to compute a combined index is to create an index on $s_1 \circ ... \circ s_n$. However, in many projects only genomes from one species are considered. These projects often deal with hundreds of highly similar sequences. We illustrate this scenario by an example.

EXAMPLE 2. *Consider $s_1 = AATGAGAGCGTAGTAGAA, s_2 = TATGAGAGCGTAGTAGAG$, and $S = \{s_1, s_2\}$. Assume that we search all 1-approximate matches for CGC, i.e., we want to compute $DBsearch(S)_{CGC}^1$. It is easy to see that $s_1 \sim_2 s_2$, since replacing the first symbol of s_1 with T and the last symbol with G does the job. Computing $DBsearch(S)_{CGC}^1$ sequence by sequence would imply that many substrings are checked twice.*

[2]We use the terms strings and sequences interchangeably during the rest of the paper.

Similarity between sequences can be exploited for sequence compression using so-called referential compression schemes [6], which encode the differences of an input sequence with respect to a preselected reference sequence. Using a space-efficient encoding of differences and efficient algorithms for finding long stretches of bases without differences, the best current referential compression algorithm we are aware of reports compression rates of up to 500:1 for human genomes [12], which is much higher than the compression rates of non-referential schemes. Due to the quickly increasing number of sequenced genomes, compression is considered a key technology for genomic labs [37]. In this work, we show that compression can also be used to speed up similarity search.

3.3 Referential Compression

We define a very general notion for encoding referential matches, similar to [45].

DEFINITION 3. *We define a* referential match entry *as a triple* $rme = (start, length, mismatch)$*, where start is a number indicating the start of a match within the reference, length denotes the match length, and mismatch denotes a symbol. The* length *of a referential match entry, denoted* $|rme|$*, is* $length + 1$.

Given a reference ref and a to-be-compressed sequence s, the idea of referential compression is to find a small set of rme's from s with respect to ref that is sufficient to reconstruct s.

DEFINITION 4. *Given two sequences s and ref, a* referential compression *of s with respect to ref, denoted* $compress(s, ref)$*, is a list of referential match entries,*

$$compress(s, ref) = [(start_1, length_1, mismatch_1), ..., (start_n, length_n, mismatch_n)],$$

such that

$$(ref(start_1, length_1) \circ mismatch_1) \circ ... \circ (ref(start_n, length_n) \circ mismatch_n) = s.$$

Sometimes we also use rc instead of $compress(s, ref)$*, if s and ref are known from the context or not relevant. The* offset *of a referential match entry* rme_i *in a referential compression* $rc = [rme_1, ..., rme_n]$*, denoted* $offset(rc, rme_i)$*, is defined as* $\sum_{j<i} |rme_j|$*. The* inverse of a referential compression $compress(s, ref)$ *is denoted* $decompress(compress(s, ref), ref)$*. Given a referential match entry* $(start, length, mismatch)$*, we write* $(start, length, mismatch) \in compress(s, ref)$*, if and only if* $(start, length, mismatch)$ *is an element in the referential compression* $compress(s, ref)$.

It is easy to see that $decompress(compress(s, ref), ref) = s$. The offset of a referential match entry in a referential compression corresponds to the position of the entry in the uncompressed sequence. The inverse of a referential compression is the decompression of a referential compression with respect to the reference, such that we obtain the original input sequence.

EXAMPLE 3. *An example for the referential compression of the sequence CGGACAAACTGACGTTCGACG with respect to the reference sequence GACGATCGACGACGGACAAACA is shown in Figure 2. The input is compressed into three referential match entries.*

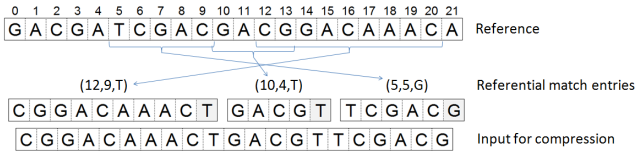


Figure 2: Example for one referential compression

The first referential match entry is $(12, 9, T)$*, which describes a match for the string CGGACAAACT at position* 12 *of the reference. The mismatch character is T (in the reference an A is found instead of a T). The second referential match entry compresses the string GACGT. A referential match entry for the string GACG in the reference at position* 10 *is introduced, together with a mismatch for symbol T. The last referential match entry compresses the string TCGACG. Although the string can be completely found in the reference, we only encode the first five symbols as a link to the reference and add G as a mismatch symbol. The offset of referential match entry* $(5, 5, G)$ *is* $|(12, 9, T)| + |(10, 4, T)| = 15$.

Clearly, we require the less rme's, the longer the matches, i.e., the shared subsequences, are. It does not matter at which position of the reference these matches lie; in particular, matches need not be in any particular order. We exploit this observation in Algorithm 1. To create a referential compression of input sequence s with respect to ref, the algorithm matches prefixes of s with substrings of ref using a compressed suffix tree of ref. The longest such prefix is removed from s, encoded as an rme and added to $compress(s, ref)$. The algorithm terminates once s contains no more symbols. Please note that a referential compression of a sequence with respect to a reference is not unique. A simple example for a non-unique referential compression with respect to the reference $ref = ATA$ is $compress(AA, ref) = [(0, 1, A)]$ and $compress(AA, ref) = [(2, 1, A)]$. The referential compression algorithm is greedy and optimal assuming that the storage necessary for referential match entries is uniform.

3.4 Referential Sequence Database Search

So far, we have considered only two sequences, an input and a reference. In the following, we study the problem of searching a database of sequences which are first referentially compressed with respect to a reference. We call this the referential sequence database search problem.

DEFINITION 5. *Let* $S = \{s_1, ..., s_n\}$ *be a sequence database and ref be a reference sequence. A* referential sequence database *RS for S and ref is a tuple* $\langle ref, rcs \rangle$*, such that* $rcs = [rc_1, ..., rc_n]$ *is a list of referential compressions with* $1 \leq i \leq n$*,* $rc_i = compress(s_i, ref)$*. Given a query q and a parameter k, we define* $RDBsearch(RS)_q^k$ *as the set of all k-approximate matches for q in the decompressed sequences of RS, i.e.,*

$$RDBsearch(RS)_q^k = \{(l, search(decompress(rc_l, ref))_q^k) \mid rc_l \in rcs\}.$$

By definition, we have $DBsearch(S)_q^k = RDBsearch(RS)_q^k$ for every sequence database S and each referential sequence database RS for S. Solving the referential sequence database search problem will immediately solve the corresponding sequence database search problem.

4. SEARCHING REFERENTIALLY COMPRESSED SEQUENCES

The main contribution of our work is the transformation of the problem of k-approximate searching in large, highly similar sequences into k-approximate searching in referentially compressed sequences. We emphasize that it is not necessary to decompress any compressed sequence during the online search phase.

EXAMPLE 4. *Suppose we want to search the referentially compressed sequence from previous Example 3 for occurrences of a string TTGA with* $k = 1$*. The situation is depicted in Figure 3. In this example, the query TTGA has four* 1*-approximate matches in CGGACAAACTGACGTTCGACG: substrings CTGA and TGA (both overlapping referential match entries 1 and 2) and substrings*

Algorithm 1 Referential Compression Algorithm

Input: to-be-compressed sequence s and reference sequence ref
Output: referential compression $compress(s, ref)$ of s with respect to ref

Let $compress(s, ref)$ be an empty list
while $|s| \neq 0$ **do**
  Let pre be the longest prefix of s, such that $(i, pre) \in search(ref)^0_{pre}$, for a number i
  if $s \neq pre$ **then**
    Add $(i, |pre|, s(|pre|))$ to the end of $compress(s, ref)$
    Remove the first $|pre|+1$ symbols from s
  else
    Add $(i, |pre|-1, s(|pre|-1))$ to the end of $compress(s, ref)$
    Remove the prefix pre from s
  end if
end while

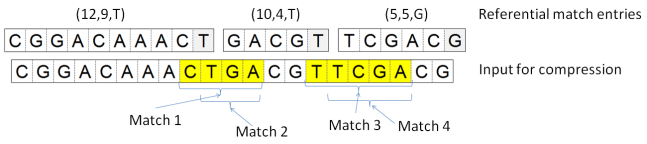


Figure 3: Searching for 1-approximate matches of the string $TTGA$ in sequence $CGGACAAACTGACGTTCGACG$.

TTCGA and TCGA (both overlapping referential match entries 2 and 3).

Each match in a referential compression must be either 1) a match inside the reference part of a referential match entry or 2) overlapping at least one mismatch character.

PROPOSITION 1. *Given a sequence s, a referential compression $compress(s, ref)$ for s with respect to a reference ref, a query q, and value k, then for each $(p, m) \in search(s)^k_q$, there exists at least one $rme = (start, length, mismatch) \in compress(s, ref)$ such that either*

1. *m is a subsequence of $ref(start, length)$ or*
2. *$p \leq offset(compress(s, ref), rme) + length$ and $p + |m| \geq offset(compress(s, ref), rme) + length$.*

This proposition gives rise to an algorithm for solving k-approximate string search problems over a single referentially compressed sequence (we extend this algorithm for multiple sequences below in this paper):

1. Find all k-approximate matches inside the reference sequence and map these matches to referential match entries in the compressed sequence and
2. find all matches in subsequences overlapping at least one mismatch character of any referential match entry in the referentially compressed sequence.

The first step can be performed by using an index structure on the reference sequence. In our case, we can reuse the index for the reference sequence, which was used to create the referentially compressed sequences. The second step, finding all matches in sequences overlapping mismatch characters, needs more thought. First of all, the number of these overlapping sequences is equal to the number of referential match entries, since we have to create one such sequence for each mismatch character. The maximum length of these overlap sequences depends on the actual query length and error threshold k.

4.1 Searching in Referential Match Entries

In order to find all matches inside referential match entries (to be more precise, inside the reference part of the referential match entries), the reference sequence is searched first and then all matches from the reference are post-filtered to identify all matches in referential match entries of the referentially compressed sequence.

The reference sequence is searched with the help of the index structure that was used to referentially compress the sequences. In our case, we have used a compressed suffix tree for the reference sequence. Exact matches can be found with a compressed suffix tree easily. For $k > 0$, we use the *"seed-and-extend"* paradigm, exploiting the fact that an alignment that allows at most k mismatches must contain at least one exact match (*"seed"*) of a substring of length $\left\lfloor \frac{|q|}{k+1} \right\rfloor$, where $|q|$ is the length of the query [4]. The query is broken up into $k+1$ parts and each part is searched exactly in the compressed suffix tree. All matches for one of the query parts are extended in order to identify full k-approximate matches. The result of the seed-and-extend search, $search(ref)^k_q$, is a set of n matches of the form (pos_i, m_i), such that each match is represented with a matching position pos_i in the reference and the matching sequence m_i. We define a projection operation, which transforms all matches in the reference into matches in referential match entries.

DEFINITION 6. *Given a referential compression rc with respect to sequence ref and given $search(ref)^k_q$, the* set of projected matches, *denoted $project(search(ref)^k_q, rc)$, is defined as*

$$
\begin{aligned}
&project(search(ref)^k_q, rc) = \{ \\
&\quad (p, m) \mid \exists i, pref.((start_i, length_i, mismatch_i) \in rc \wedge \\
&\quad (p_{ref}, m) \in search(ref)^k_q \wedge (p_{ref} \geq start_i) \wedge \\
&\quad (p_{ref} + |m| \leq start_i + length_i) \wedge \\
&\quad p = p_{ref} - start_i + offset(rc, (start_i, length_i, mismatch_i)))\}.
\end{aligned}
$$

Depending on the number of referential match entries in the referential compression and the number of results in $search(ref)^k_q$, different strategies for computing $project(search(ref)^k_q, rc)$ show different performance. We have used an index structure for the start positions of all referential match entries (using hash buckets), in order to speed up the lookup of subsuming referential match entries for a set of given matches in $search(ref)^k_q$. Especially in case of multiple queries, index structures over the referential match entries can improve the time needed for computation of projected matches.

EXAMPLE 5. *Let sequence $s = GACTATAACAGGATAC$ and $ref = AACAGGACTTTATAC$. One referential compression of s with respect to ref is $rc = [(5,4,A), (10,2,A), (2,5,T), (1,1,C)]$. Now assume a query AC and $k = 0$. It follows that $search(ref)^0_{AC} =$*

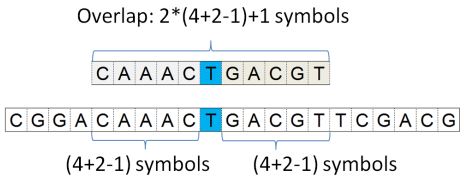


Figure 4: Extracting one overlap sequence for the referential match entries $(12,9,T)$ and $(10,4,T)$ of the referentially compressed sequence of $CGGACAAACTGACGTTCGACG$ with $QL_{max} = 4$ and $k_{max} = 2$.

$\{(1,AC),(6,AC),(13,AC)\}$. *As a result of the projection we obtain* $project(search(ref)^0_{AC}, rc) = \{(1,AC)\}$, *because* $(6,AC)$ *is contained in the rme* $(5,4,A)$. *For the other two matches in* ref *we cannot find an rme.*

4.2 Searching across Sequence Deviations

Finding all matches overlapping at least one mismatch character is described next. In the example from Figure 3, the 1-approximate match $CTGA$ overlaps the first and the second referential match entry. The length of these to-be-checked overlapping sequences depends on the actual query length and k. In the worst case, for exact string matching, the last (first) character of the query q matches the mismatch character. At most $|q|-1$ characters can be found to the left (right) of the mismatch character. In the case of approximate search, in the worst case, k symbols might be inserted in the sequence. Therefore, in addition k characters need to be extracted from the left and the right of the mismatch character. Assuming a maximum query length QL_{max} and a maximum edit distance k_{max}, the overlap sequence is built by extracting the first $QL_{max}+k_{max}-1$ characters from the left of the mismatch character, concatenating the mismatch character, and concatenating the next $QL_{max}+k_{max}-1$ characters to the right of the mismatch character.

DEFINITION 7. *Given a referential compression rc for the sequence s, and a referential match entry* $rme_i \in rc$, *with* $rme_i = (start, length, mismatch)$, *the* overlap sequence of rme_i with respect to rc, *denoted* $ovl^{ref}_{rc}(rme_i)$, *is defined as*

$$ovl^{ref}_{rc}(rme_i) = (decompress(rc, ref))(offset(rc, rme_i) + length - (QL_{max} + k_{max} - 1), 2*(QL_{max} + k_{max} - 1) + 1).$$

The set of overlap sequences for a referential compression rc *with respect to* ref, *denoted* $overlaps(ref, rc)$, *is defined as*

$$overlaps(ref, rc) = \{(offset(rc, rme) + length - (QL_{max} + k_{max} - 1), ovl^{ref}_{rc}(rme)) \mid rme \in rc\}.$$

In fact, the referentially compressed sequence does not have to be decompressed completely in order to compute $ovl_{rc}(rme_i)$. It is sufficient to partially decompress $QL_{max}+k_{max}-1$ symbols to the left and to the right of the mismatch character in rme_i.

EXAMPLE 6. *An example for the extraction of an overlap sequence is shown in Figure 4. Given* $QL_{max} = 4$ *and* $k_{max} = 2$, *the overlap sequence for the referential match entries* $(12,9,T)$ *and* $(10,4,T)$ *is extracted, yielding* $CAAACTGACGT$. *The length of the overlap sequence is* $(4+2-1)+1+(4+2-1) = 11$ *symbols. The referential compression for* $CGGACAAACTGACGTTCGACG$ *with respect to the reference* $GACGATCGACGACGGACAAACA$ *contains three referential match entries. Therefore, two more overlap sequences have to be extracted:* $TGACGTTCGAC$ *(for the overlap of entries number two and three) and* $TCGACG$ *(for the final referential match entry, which is only extended to the left).*

Algorithm 2 Referential Search Algorithm

Input: Referential sequence database $\langle ref, rcs\rangle$ with $rcs = [rc_1, ..., rc_n]$, query q, and k,
Output: Solution $\langle ref, rcs\rangle^k_q$

1: $\langle ref, rcs\rangle^k_q = \emptyset$
// First step – search reference
2: $refmatches = search(ref)^k_q$
3: **for** $1 \le i \le n$ **do**
4: Add $(i, project(refmatches, rc_i))$ to $\langle ref, rcs\rangle^k_q$
5: **end for**
// Second step – search overlap sequences
6: **for** $1 \le i \le n$ **do**
7: **for** $(pos, t) \in overlaps(ref, rc_i)$ **do**
8: **for** $(pos_2, u) \in search(t)^k_q$ **do**
9: Add entry $(i, (pos + pos_2, u))$ to $\langle ref, rcs\rangle^k_q$
10: **end for**
11: **end for**
12: **end for**

In order to completely search the referentially compressed sequence of $CGGACAAACTGACGTTCGACG$, the reference sequence needs to be searched (following the seed-and-extend approach), and, in addition, three shorter strings have to be searched, i.e., one for each referential match entry. For the sake of simplicity, the length of referential matches are chosen rather small in this example. In general, the number of overlap sequences is equal to the number of referential match entries in these sequences. However, in case of collections of highly-similar DNA sequences, many extracted overlap sequences turn out to be identical, as most differences between human genomes are rather short and there exist only three possible deviations. This effect is the stronger for shorter maximum query lengths, and it would be weaker if strings over a larger alphabet were searched. We evaluate the number of non-identical overlaps in Section 6.

4.3 Searching Referential Sequence Databases

The complete referential search algorithm is shown in Algorithm 2. The algorithm solves a referential sequence database search problem $\langle ref, rcs\rangle^k_q$. In Line 2, all k-approximate matches inside the reference sequence are computed. In our implementation we have used compressed suffix trees (with seed-and-extend for $k > 0$). The for-loop from Line 3 to Line 5 projects these matches from the reference sequence onto the referential match entries in the referential sequence database. The remaining part of the algorithm (Line 6-13) finds all matches in overlapping sequences. The first loop iterates over all the referential compressions in the referential sequence database. The second inner loop (starting Line 7) iterates over all overlapping sequences of the referential compression rc_i. The innermost loop (starting Line 8) iterates over all k-approximate matches inside the current chosen overlap sequence u and adds these matches to the solution set. Note that adding elements to $\langle ref, rcs\rangle^k_q$ might require care, in case a match for a sequence with the same identifier exists already.

EXAMPLE 7. *We want to search for* 0*-approximate occurrences of* $q = AA$ *(with fixed* $QL_{max} = 3$ *and* $k_{max} = 0$*) in sequences:*

- $s_1 = CGGACAAACTGACGTTCGACG$
- $s_2 = CGGACAAACAGACGTTCGACC$
- $s_3 = CGGACAAACTGACGTTCGAA$

With $ref = GACGATCGACGACGGACAAACA$, we obtain the following three referential compressions:

- $rc_1 = [(12,9,T),(10,4,T),(5,5,G)]$
- $rc_2 = [(12,9,A),(10,4,T),(5,5,C)]$
- $rc_3 = [(12,9,T),(10,4,T),(5,4,A)]$

We want to compute $\langle ref,[rc_1,rc_2,rc_3]\rangle^0_{AA}$. During the first step of the algorithm, we obtain

$$refmatches = search(ref)^0_{AA} = \{(17,AA),(18,AA)\}.$$

Projecting these reference matches onto the referential compression rc_1, we add $(1,\{(5,AA)\})$ and $(1,(\{6,AA\}))$ to the solution $\langle ref,[rc_1,rc_2,rc_3]\rangle^0_{AA}$.
For rc_2 and rc_3 we add the results $\{(2,\{(5,AA)\}),(2,\{(6,AA)\})\}$ and $\{(3,\{(5,AA)\}),(3,\{(6,AA)\})\}$, respectively. In the second step, all matches in overlapping sequences are added. The overlap sequences are:

$$overlaps(ref,rc_1) = \{(7,ACTGA),(12,CGTTC),(18,ACG)\}$$
$$overlaps(ref,rc_2) = \{(7,ACAGA),(12,CGTTC),(18,ACC)\}$$
$$overlaps(ref,rc_3) = \{(7,ACTGA),(12,CGTTC),(17,GAA)\}$$

The only overlap sequences with 0-approximate match for AA is GAA for referential compression rc_3. Therefore, the algorithm adds the match $(3,\{(18,AA)\})$ to $\langle ref,[rc_1,rc_2,rc_3]\rangle^0_{AA}$. The overall result of Algorithm 2 for the example is:

$$\langle ref,[rc_1,rc_2,rc_3]\rangle^0_{AA} = \{(1,\{(5,AA),(6,AA)\}),$$
$$(2,\{(5,AA),(6,AA)\}),(3,\{(5,AA),(6,AA),(18,AA)\})\}$$

One interesting observation from the example is that the number of unique overlap sequences can be smaller than the total number of referential match entries. In total, we have to check nine overlap sequences: one for each referential match entry inside a referentially compressed sequence. However, extracting the actual overlaps yields that we only have six unique overlaps, since $CGTTC$ occurs three times and $ACTGA$ occurs twice. This observation is important when searching the overlaps for k-approximate matches: instead of naively searching each overlap sequence, we find (and remove) identical overlap sequences in a preprocessing step. The preprocessing step is implemented using hash tables and can greatly speed up the actual query answering time.
All unique overlap sequences are searched for matches using a very simple approach: all overlaps are concatenated to a large string (separated by k_{max} fresh symbols, not contained in the alphabet of sequences). For instance, the overlaps AGT and AC are stored as $AGT^{**}AC$, if $k_{max} = 2$.
We use a compressed suffix tree to find matches in the concatenated string in the same way as we search the reference sequence. Following the seed-and-extend approach, we can find all k-approximate matches in all overlaps efficiently. Matches in the large concatenated string are being projected back to the single overlaps. The idea is depicted in Figure 5. Although this approach is quite simple, it scales surprisingly well. Standard data structures for k-approximately searching collections of strings [47] should improve search times. Such an optimization is left for future work. Note that we do not uncompress any overlaps during the search phase, since we build the above compressed suffix tree over all overlaps.

5. BEST REFERENCE SELECTION

One open problem when searching compressed sequences is the selection of a best reference sequence with respect to our referential compression algorithm. With increasing similarity between reference and to-be-compressed sequence, longer referential match entries can be found and the compression ratio is increasing. Thus, choosing a proper reference will increase the compression ratio and also reduce search times.

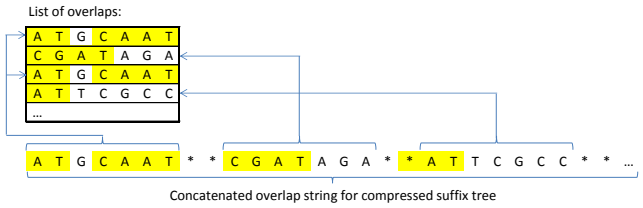


Figure 5: Approximately searching overlaps.

Algorithm 3 Reference Selection RSbitX

Input: set of to-be-compressed sequences $s_1,...,s_n$, set of candidate reference sequences $ref_1,...,ref_m$, a base reference sequence ref_{base}, and a speedup value X
Output: index b for best reference

1: **for** $1 \le j \le n$ **do**
2: Split s_j into 1000 blocks $b_1,...,b_{1000}$ of equal length
3: Let sx_j be the concatenation of each X-th block of $b_1,...,b_{1000}$
4: **end for**
5: **for** $1 \le i \le m$ **do**
6: Let $val_i = 0$
7: **for** $1 \le j \le n$ **do**
8: $val_i = val_i + |rsim(compress(sx_j, ref_{base}), compress(ref_i, ref_{base}))|$
9: **end for**
10: **end for**
11: Find the smallest val_{min} from $val_1,...,val_m$ and let $b = min$

DEFINITION 8. *Given a sequence s and a set of candidate references $\{ref_1,...,ref_m\}$, ref_i is called an* optimal reference *iff there does not exists a $j \neq i$ with $|compress(s,ref_j)| < |compress(s,ref_i)|$.*
A naive strategy to find an optimal reference sequence is to compress all the to-be-compressed sequences against all possible reference sequences and select the reference that yields the least number of referential match entries, named RSbest. If sequences are long, as in our case, this is a highly time consuming undertaking as we need to compute m^2 referential compressions, where m is the number of candidate reference sequences. If one wants to compress 1,000 sequences, choosing the best reference following this strategy does not scale.
In the following, we describe a heuristic which scales well in the number of candidates and, as shown in the evaluation section, in many cases identifies near-optimal references. However, the problem of efficiently finding an optimal reference remains unsolved and is an important topic for future work (see Section 7).
Instead of compressing a sequence against all candidate references, we compare the referential compression of the sequence and the referential compression of the reference candidates with respect to *one* chosen base reference. We then select that reference whose referential compression against this base references has the highest *referential similarity* to the referential compressions of all sequences to the base reference. The idea is that two referential compressions are more similar if they share more referential match entries.

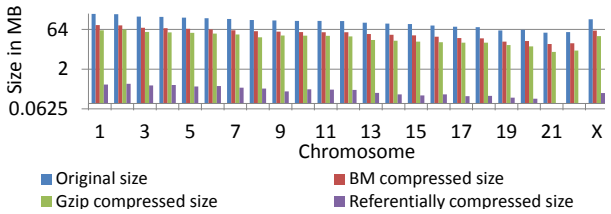


Figure 6: Compressing human chromosomes against HG19. Values are averaged over 1,092 genomes.

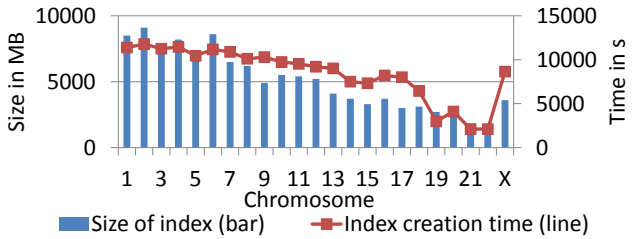


Figure 7: Size and creation time of RCSI per chromosome.

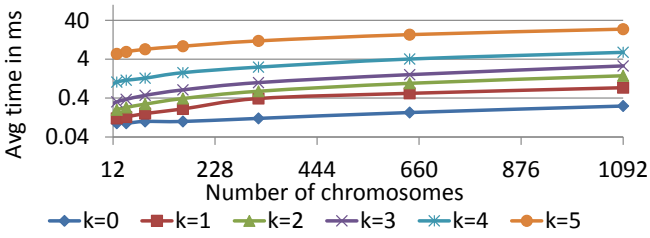


Figure 8: Average query run times on different number of chromosomes and using different error thresholds.

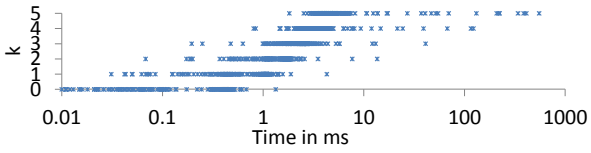


Figure 9: Query search time against k.

DEFINITION 9. *The* referential similarity *of two referential compressions rc_1 and rc_2, denoted $rsim(rc_1, rc_2)$, is defined as*

$$rsim(rc_1, rc_2) = |rc_1 \cup rc_2| - |rc_1 \cap rc_2|.$$

As a second optimization, we use only parts of the sequences to select the best reference, building on the fact that the degree of similarity between similar long sequences does not change significantly between local stretches. The full algorithm, called RSbitX, is shown in Algorithm 3. RSbit5 stands for compressing only $\frac{1}{5}$ of each sequence. Note that RSbitX computes $m+n$ referential compressions and n*m referential similarities.

6. EVALUATION

We evaluated our RCSI algorithm using different data sets, settings, and computers. Most experiments were run without any parallelization on an Acer Aspire 5950G with 16 GB RAM and an Intel Core i7-2670QM processor; the exception is Section 6.4, for which we used a server with 1 TB RAM and 4 Intel Xeon E7-4870 (in total, hyperthreading enables the use of 80 virtual cores). Code was implemented in C++, using the BOOST library, CST [35], and libz.

6.1 Data and Queries

We tested our method on a set of 1,092 human genomes from the 1000 Genomes Project [2]. The data is originally provided in the Variant Call Format (VCF) [10][3], which we converted into raw consensus sequences for each chromosome of each genome. The total dataset has 3.09 TB uncompressed, or 700 GB when compressed with GZip. Note that in the following experiments we sometimes search on sets of chromosomes and sometimes on the full set of genomes.

We measured search times on different sets of sequences and different edit distance thresholds. For those measurements, we created a set of 50,000 queries of length 120-170 (equally distributed) for each chromosome by (1) randomly extracting substrings and (2) adding modifications: Bases were randomly replaced with a probability of 0.05 and single bases were added/removed with a probability of 0.01 percent, respectively. For scalability studies, we used randomly chosen subsets of the input genomes of sizes 5, 10, 20, 40, 80, 160, 320, 640, and 1,092.

6.2 Compression

We first evaluate our compression scheme using as reference the human genome HG19 [22], which is commonly used as a human reference genome (note that the experiments described below used a different – and better from a compression point of view – reference). The size of the uncompressed chromosomes of HG19 varies between 50 MB (Chromosome 22) and 250 MB (Chromosome 1).

[3]ftp://ftp.1000genomes.ebi.ac.uk/vol1/ftp/release/20110521/

We created a compressed suffix tree for each chromosome for performing the compression of the 1,092 genomes. These suffix trees are on average 1.72 times larger than the input sequences. The time required for creating them grows linearly with the size of the input and took 17-118 s (Chromosome 22, Chromosome 2), which amounts to a throughput of approx. 2.4 MB/s. Compressing a complete genome took approx. 30 s on average; compressing all 1,092 genomes took roughly eight hours.

Figure 6 shows the average sizes of all 1,092*23 compressed chromosomes using different compression techniques. With bit manipulation techniques (BM), i.e., encoding three symbols within one byte, we obtain a ratio of 3:1. GZip achieves a ratio of 4:1 on average. In contrast, RCSI yields an average compression ratio of 436:1.

6.3 Searching 1092 Chromosomes

We compressed all 1092 genomes against the reference chosen by RSbit5 (see Section 6.6) and created RCSI for $QL_{max} = 200$ and $k_{max} = 5$. In total, this process took 54 hours (40 min for Chr22, 50 MB, and 3.2 hours for Chr2, 250 MB) on our test laptop. The size of the entire search index is 115 GB (1.7 GB for Chr22 and 9.1 GB for Chr2); details are shown in Figure 7.

We next ran 50,000 chromosome-specific queries on each of the chromosome-specific indexes (refer to Section 6.4 for searching all 1,092 genomes). Figure 8 studies the impact of the error threshold and the number of genomes on search performance. For exact string matching, average search time per query is between 0.06 ms (for five sequences) and 0.25 ms (for 1,092 sequences). Runtimes are fairly constant for the different chromosomes, i.e., the size of

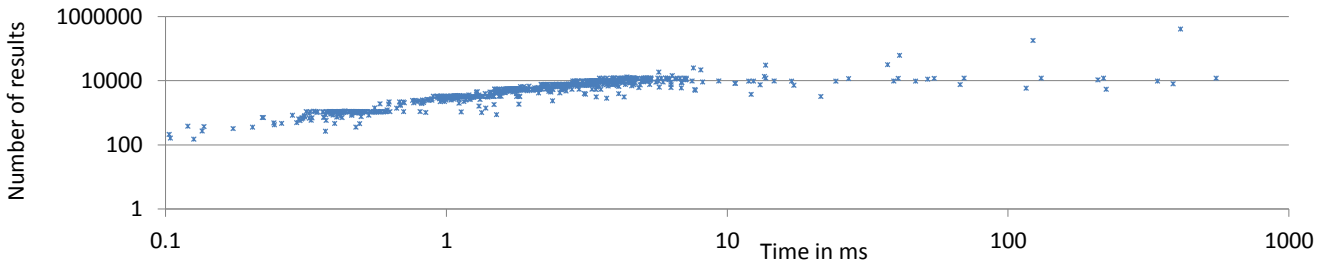


Figure 10: Query search time against number of results.

a chromosome only has a negligible effect on runtime (data not shown). For $k = 1$, the average search time per query is still below 1 ms even for 1,092 chromosomes. Starting from $k = 4$, average search runtimes increase recognizably. For $k = 4$ and 1,092 chromosomes, we need 6 ms to search one query. For $k = 5$, searching 1,092 sequences already takes 23.6 ms on average; however, the median is only 4.6 ms. In total, searching 1,092 sequences takes only 10 times more search time than searching five sequences, showing that the use of compression gives our algorithm very good scalability in the size of the data set.

We further analyze query runtimes in Figure 9. While the slowest exact query only takes around 1 ms, the slowest 5-approximate query needs almost 1 s. However, for values of $k < 5$, almost all queries (exactly 98.4%) can be answered in less than 10 ms; even for $k = 5$, 85.5% of queries take less than 10 ms. The reason for this deviation can be seen in Figure 10, where we break down search times by result sizes. Due to the high number of repeats in the human genome (up to 60% of the human genome consist of repetitive elements [11]), some queries fall into regions that appear very often throughout the genome (recall that queries were sampled at random). These queries with 10,000 and more results require excessive runtimes, which explains the large deviation between mean and median runtimes. Note however, that in biological applications such regions are typically masked before performing searches as results are uninformative for all but very few questions (see, for instance, RepeatMasker). One way to solve the problem with large result sets might be to introduce some kind of polynomial delay algorithm [18], returning the most similar matches first, or those with the fewest number of occurrences.

6.4 Searching 1,092 Genomes

So far, all our measurements were obtained by running chromosome-specific queries on a commodity laptop. Clearly, the obtained speed can be scaled up by using 23 cores (one for each chromosome) and sufficient memory to load the entire index into memory. To show the feasibility of this strategy, we performed an experiment on a mid-size server with 1 TB RAM and 80 virtual cores. We created a workload of 23,000 queries (from each chromosome, 1,000 queries were taken from the set described in the beginning of Section 6), and searched each query against all 1,092 genomes in parallel, where each core searched one chromosome-specific index. Average runtimes per query are shown in Figure 11. On average, exact matching takes 0.02 ms, 1-approximate matching 0.09 ms, and 5-approximate matching 15.29 ms. The difference between these average runtimes and those reported in previous chapters are due to the more powerful CPU of the server compared to our laptop.

6.5 Competitive Evaluation

We are aware of only one other tool that follows a similar approach to RSCI: GenomeCompress. Other algorithms either create indexes

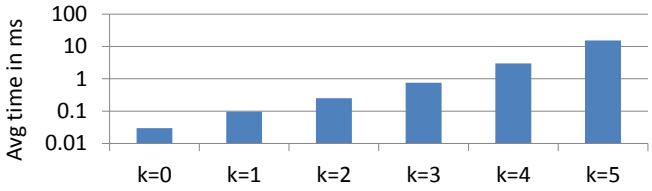


Figure 11: Average query runtime (in ms)

that are much larger than the to-be-searched sequences and are thus not applicable for the data sets we target [31, 35], or provide only incomplete solutions and often solve slightly different problems. Still, we find it instructive to compare against such tools as it shows that, for the special setting of searching similar sequences, RSCI scales as well or even better than these heuristics. Therefore, in the following we compare RSCI against GenomeCompress, BLAST, and Bowtie 2.

We compare our runtimes against a range of competitors, namely BLAST, the standard search tool for local alignments, Bowtie 2, a state-of-the-art read mapper, and GenomeCompress, an algorithm following a similar approach as RCSI.

First, we illustrate the speed-up of RCSI compared to the popular sequence search tool BLAST [3]. BLAST is capable of searching high data volumes; for instance, it is used at the GenBank servers where it runs on large clusters to serve thousands of queries per day on the archive currently containing approx. 145 GB (which is 20 times smaller than our data set). Note that RCSI is not directly comparable to BLAST, as RCSI exactly answers k-approximate searches, while BLAST is a heuristic to find local alignments; still, we believe that the differences in runtimes are interesting. We indexed only HG19 [22] with BLAST, leading to an index of 4.9 GB. BLAST queries (with default parameters) on this single genome took 12 s on average, with extreme cases taking several minutes. Experiments with one to eight Chromosome 1 showed that search times grow linear with the number of chromosomes (63 ms for one chromosome and 328 ms for eight chromosomes); the same holds for database size (62 MB for one chromosome and 498 MB for eight chromosomes). This is in stark contrast to the runtimes and scalability behaviour of RCSI.

Next, we compared RCSI against Bowtie 2 [25], a state-of-the-art tool for mapping long sequence reads (50-1000 bp) against a reference genome. In contrast to many other read mappers which only cope with base substitutions, Bowtie 2 also allows small gaps in matches and is thus in principle comparable to RCSI. However, there are also important differences: (1) Bowtie 2 only reports best matches, while RCSI computes all matches within the error threshold; (2) Bowtie uses several heuristics to filter repetitive or generally uninformative matches, while RCSI finds all matches;

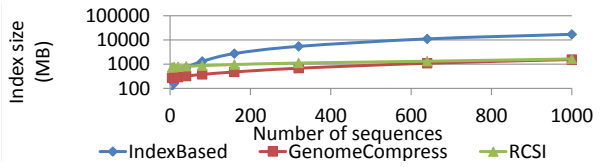


Figure 12: Index size (in MB).

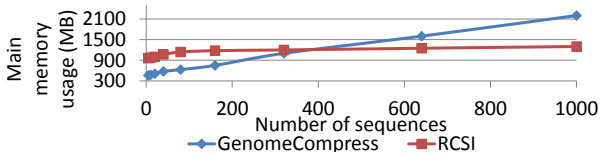


Figure 13: Main memory usage (in MB).

(3) Bowtie searches a single reference, while RCSI searches many references implicitly in parallel. Indexing HG00096 with Bowtie 2 took around three hours and the index size is 3.9 GB. Query times (with default parameters set) are very fast, with an average time of 0.11 ms, which is about the time RCSI needs to answer 1-approximate queries when working in parallel (see Section 6.4). We tried to generate a composite index for several human genomes (by simply concatenating them), but failed to do so as Bowtie 2 can handle only sequences up to around 3.6 GB. For larger sequences the developers of Bowtie 2 propose to split up the input sequence into smaller chunks and create single indexes for each chunk. For 1,092 human genomes, one would need to create more than 1,000 indexes, summing up to more than 3 TB of storage. Unless all these indexes can be queried in parallel (on more than 1,000 cores), the average query time will increase recognizably.

Finally, we compared RCSI to two other methods which solve exactly the same problem. As a baseline, we simply built one compressed suffix tree for each sequence and searched these one-after-the-other (we call this method IndexBased in the following). Further, we installed GenomeCompress [49], a very recent tool that also builds on genome compression (see Section 2). GenomeCompress takes so-called delta files as input, i.e., files that describe the difference to a reference. The tool transforms these files into a compressed index over multiple sequences. As delta files represent sets of edit operations, the algorithm to search the compressed index is considerably more complex than ours which essentially searches just strings - either from the reference or from an input sequence or from both. Unfortunately, the code provided with GenomeCompress does not contain an algorithm to generate those delta files (X. Yang, personal communication); however, since delta files are not unique and the concrete representation has an impact on the compressed index, we think that re-implementing this step could add bias to the comparison. Therefore, we could compare GenomeCompress to RCSI only on those sequences used in [49] (for those, delta files are provided). This dataset consists of up to 1,000 sequences taken from the first 10 MB of a Chromosome 1, giving a total size of only approx. 10 GB (uncompressed). Queries were randomly generated as before.

Figure 12 shows the size of indexes for a growing numbers of sequences. The size of IndexBased grows linearly since one compressed suffix tree is created for each sequence. For five sequences the index size of GenomeCompress is roughly three times smaller (275 MB) than for RCSI (736 MB), but with increasing number of sequences, index sizes of GenomeCompress and RCSI become very similar. For 1,000 sequences (of length 10 MBases), GenomeCompress requires 1,550 MB and RCSI needs 1,650 MB. Besides the footprint on disk, we also measured memory consumption of RCSI and GenomeCompress for different number of sequences; see Figure 13. Interestingly, the runtime memory footprint of RCSI is larger than that of GenomeCompress for sets of up to 400 sequences, but grows only very slowly with more sequences. The slope of GenomeCompress is much steeper. For 1,000 sequences, the main memory usage already almost doubled (2,200 MB for GenomeCompress, and 1,300 MB for RCSI). If the main memory usage keeps on growing in this way then GenomeCompress will not be able to manage an index for 1,000 complete genomes in 1 TB of main memory. The indexing times for GenomeCompress are a little bit higher than for RCSI: roughly a factor of five. Note RCSI works directly on raw sequences, while GenomeCompress is run on preprocessed input, produced by a process similar to global sequence alignment. The time for this preprocessing step of all input sequences is not included in the indexing times for GenomeCompress, but will increase indexing times considerably.

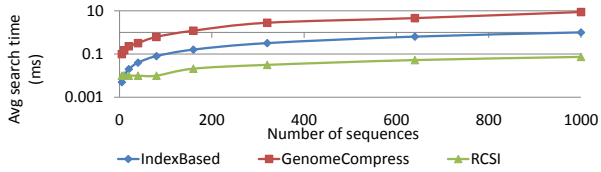
Average search time for different numbers of sequences are compared in Figure 14. Clearly, search times for IndexBased grow linearly with the number of sequences. Searching 1,000 sequences with IndexBased takes 1 ms (for exact search) and 230 ms (for 3-approximate search; recall that here we only search 10 MB of each Chromosome 1), respectively. GenomeCompress on average needs 8.8 ms for exact search in 1,000 sequences while RCSI needs only 0.07 ms. For 3-approximate search, RCSI is roughly 7 times faster than GenomeCompress (5.3 ms vs. 35 ms) on the full set, and the advantage seems to grow with more compressed and larger sequences. The main memory storage required per sequence is roughly constant for GenomeCompress (1.7 MB/sequence). Therefore, doubling the number of sequences will yield double main memory usage. For RCSI the necessary storage is decreasing with an increasing size of the database (1.2 MB/sequence at 160 sequences, 0.27 MB/sequence at 1,000 sequences). This shows improved scalability of RCSI over GenomCompress for the very small dataset already. We conjecture that 1,092 complete genomes cannot be kept within even 1 TB of main memory with the current implementation of GenomeCompress.

Please note that in our experiments GenomeCompress did not find all k-approximate matches because it cannot find matches shorter than a given threshold when compression is enabled (X. Yang, personal communication).

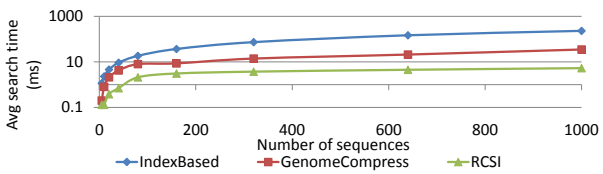
6.6 Best Reference Selection

We tested our methods for finding the best references using a set of nine randomly chosen candidates: HG00236, HG01048, HG01360, NA06994, NA18946, NA19028, NA19445, NA20508, and HG19. Testing against all genomes would require an estimated time of 7,500 hours; however, since the range of obtained compression rates remains fairly robust (exhaustive evaluation of all 1,092 reference candidates for Chromosome 22; data not shown), we believe that using a randomly selected subset is sufficient to show the achievable improvements. Compression sizes for the best, worst, and average reference are shown in Figure 15. In total, storing all genomes compressed with respect to the best reference requires 7.1 GB, while storing them with respect to the worst reference needs 8.8 GB (difference approx. 20%). Note that we always chose the optimal reference for each chromosome separately, i.e., the complete reference is composed of chromosomes from different individuals. Compressing all genomes with respect to all nine references took 67 hours.

We compared the space consumption and compression time for RSbit1 and RSbit5 (i.e. RSbitX with X=1 and X=5, see Section 5),



(a) Exact search.



(b) 3-approximate search.

Figure 14: Average search time (in ms) for up to 1,000 sequences of length 10 MB.

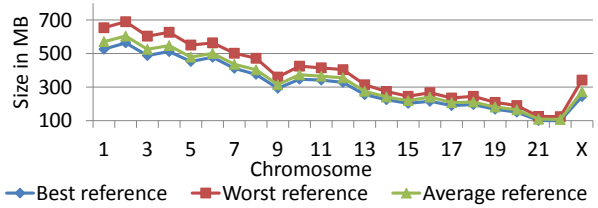


Figure 15: Difference between optimal, average and worst compression rate per chromosome, depending on reference.

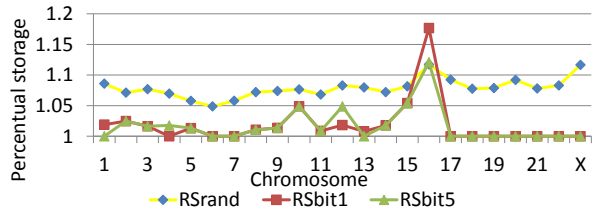


Figure 16: Space overhead by reference selection method compared to best reference. Except for Chr16, RSbit1 and RSbit5 select references clearly better than RSrand.

the exhaustive strategy RSbest, and a random selection strategy (RSrand) on our test set of nine genomes. Figure 16 shows the increase in storage depending on the reference selection method with baseline RSbest. RSrand leads to an average increase in storage of 7.6% compared to the optimal reference. RSbit1 performs significantly better and leads to only 1.8% increase. RSbit5 even slightly outperforms RSbit1 (1.7%). Only for Chromosome 16, RSbit1 and RSbit5 choose a reference worse than RSrand. Our experiments indicate that this is caused by the extremely high number of repeats in Chromosome 16 [1].

7. CONCLUSIONS

We presented RCSI, a novel method for searching thousands of human genomes. RCSI first compresses all genomes with respect to a reference. The resulting data structure, when encoded properly, is much smaller than the raw data and can be searched efficiently, thereby implicitly searching all genomes in parallel. Experiments with 1,092 genomes show that runtimes on a commodity laptop are in the range of 1-20 ms when searching a specific chromosome, or roughly in the same range when searching entire genomes on a mid-class server. We showed that RCSI considerably outperforms close and less close competitors. We also studied the problem of reference selection and presented heuristics that result in an additional 15% space reduction compared to a random selection.
Though this is not a dramatic space reduction, we believe that the topic of reference selection deserves more research in the future. First, the question of efficiently finding an optimal reference remains open. Second, there is no need to choose the reference from the set of sequences to be compressed (as we did); instead, any other or also an artificial sequence could be used. Accordingly, another open problem is that of efficiently creating an optimal reference for a set of to-be-compressed genomes.

8. REFERENCES

[1] An integrated map of genetic variation from 1,092 human genomes. *Nature*, 491(7422):56–65, Oct. 2012.

[2] 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, Oct. 2010.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct. 1990.

[4] R. A. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. *Information Processing Letters*, 59(1):21–27, 1996.

[5] S. Bao, R. Jiang, W. Kwan, B. Wang, X. Ma, and Y. Song. Evaluation of next-generation sequencing software in mapping and assembly. *Journal of human genetics*, 56(6):406–414, 2011.

[6] S. Christley, Y. Lu, C. Li, and X. Xie. Human genomes as email attachments. *Bioinformatics (Oxford, England)*, 25(2):274–275, Jan. 2009.

[7] R. Cole, L.-A. Gottlieb, and M. Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of STOC*, pages 91–100, New York, NY, USA, 2004. ACM.

[8] I. H. G. S. Consortium. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, February 2001.

[9] Consortium ICG. International network of cancer genome projects. *Nature*, 464(7291):993–998, Apr. 2010.

[10] P. Danecek, A. Auton, G. Abecasis, and 1000 Genomes Project Analysis Group. The variant call format and VCFtools. *Bioinformatics (Oxford, England)*, 27(15):2156–2158, Aug. 2011.

[11] A. P. J. de Koning, W. Gu, T. A. Castoe, M. A. Batzer, and D. D. Pollock. Repetitive elements may comprise over two-thirds of the human genome. *PLoS Genetics*, 7(12):e1002384, 12 2011.

[12] S. Deorowicz and S. Grabowski. Robust Relative Compression of Genomes with Random Access. *Bioinformatics (Oxford, England)*, Sept. 2011.

[13] H. H. Do, J. Jansson, K. Sadakane, and W.-K. Sung. Fast relative lempel-ziv self-index for similar sequences. In *Proceedings of FAW-AAIM 2012, Beijing, China, 2012.*, volume 7285 of *LNCS*, pages 291–302. Springer, 2012.

[14] P. Ferragina. String algorithms and data structures. *CoRR*, abs/0801.2378, 2008.

[15] J. Fischer, V. Mäkinen, and G. Navarro. An(other) entropy-bounded compressed suffix tree. In *Proceedings of 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 5029, pages 152–165, 2008.

[16] T. Gagie, P. Gawrychowski, J. Kärkkäinen, and Y. Nekrich. A faster grammar-based self-index. In *LATA'12*, pages 240–251, Berlin, Heidelberg, 2012. Springer.

[17] X. Ge and P. Smyth. Deformable markov model templates for time-series pattern matching. In *Proceedings of SIGKDD*, pages 81–90, New York, NY, USA, 2000. ACM.

[18] L. A. Goldberg. *Efficient algorithms for listing combinatorial structures*, volume 5, page 7. Cambridge University Press, 2009.

[19] D. Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[20] O. Harismendy, P. Ng, et al. Evaluation of next generation sequencing platforms for population targeted sequencing studies. *Genome Biology*, 10(3):R32+, 2009.

[21] W. J. Kent. BLAT-The BLAST-Like Alignment Tool. *Genome Research*, 12(4):656–664, Apr. 2002.

[22] W. J. Kent, C. W. Sugnet, T. S. Furey, K. M. Roskin, T. H. Pringle, A. M. Zahler, and D. Haussler. The human genome browser at UCSC. *Genome Research*, 12(6):996–1006, 2002.

[23] Y. Kim, K.-G. Woo, H. Park, and K. Shim. Efficient processing of substring match queries with inverted q-gram indexes. In *Proceedings of ICDE 2010,Long Beach, California, USA*, pages 721–732.

[24] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[25] B. Langmead and S. L. Salzberg. Fast gapped-read alignment with bowtie 2. *Nat Meth*, 9(4):357–359, Apr. 2012.

[26] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25–10, Mar. 2009.

[27] H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics (Oxford, England)*, 25(14):1754–1760, 2009.

[28] Y. Li, A. Terrell, and J. M. Patel. Wham: a high-throughput sequence alignment method. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Athens, Greece, June 12-16*, pages 445–456. ACM, 2011.

[29] Z. Li and T. Ge. Online windowed subsequence matching over probabilistic sequences. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 277–288, New York, NY, USA, 2012. ACM.

[30] P.-R. Loh, M. Baym, and B. Berger. Compressive genomics. *Nature Biotechnology*, 30(7):627–630, July 2012.

[31] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[32] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[33] G. Navarro. Indexing highly repetitive collections. In *Proceedings 23rd International Workshop on Combinatorial Algorithms (IWOCA)*, LNCS 7643, pages 274–279, 2012.

[34] G. Navarro and M. Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge University Press, New York, NY, USA, 2002.

[35] E. Ohlebusch, J. Fischer, and S. Gog. Cst++. In *SPIRE'10*, pages 322–333, 2010.

[36] P. Papapetrou, V. Athitsos, G. Kollios, and D. Gunopulos. Reference-based alignment in large sequence databases. *Proceedings of the VLDB Endowment*, 2(1):205–216, Aug. 2009.

[37] E. Pennisi. Will Computers Crash Genomics? *Science*, 331(6018):666–668, Feb. 2011.

[38] D. E. Reich, S. F. Schaffner, M. J. Daly, G. McVean, J. C. Mullikin, J. M. Higgins, D. J. Richter, E. S. Lander, and D. Altshuler. Human genome sequence variation and the influence of gene history, mutation and recombination. *Nature Genetics*, 32(1):135–142, Aug. 2002.

[39] A. Rheinländer, M. Knobloch, N. Hochmuth, and U. Leser. Prefix tree indexing for similarity search and similarity joins on genomic data. In *Proceedings of the 22nd SSDBM*, pages 519–536, Berlin, Heidelberg, 2010. Springer.

[40] E. E. Schadt, S. Turner, and A. Kasarskis. A window into third-generation sequencing. *Human molecular genetics*, 19(R2):R227–R240, Oct. 2010.

[41] 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*, 10(9):R98+, Sept. 2009.

[42] J. Sirén, N. Välimäki, and V. Mäkinen. Indexing finite language representation of population genotypes. In *Proceedings of the 11th international conference on Algorithms in bioinformatics*, WABI'11, pages 270–281, Berlin, Heidelberg, 2011. Springer.

[43] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In *Proceedings of the Third Annual European Symposium on Algorithms*, ESA '95, pages 327–340, London, UK, 1995. Springer.

[44] N. Vaelimaeki, V. Maekinen, W. Gerlach, and K. Dixit. Engineering a compressed suffix tree implementation. *ACM Journal of Experimental Algorithmics*, 14, 2009.

[45] S. Wandelt and U. Leser. Adaptive efficient compression of genomes. *Algorithms for Molecular Biology*, 7:30, 2012.

[46] S. Wandelt and U. Leser. String searching in referentially compressed genomes. In *Proceedings of the 4th Int. Conf. on Knowledge Discovery and Information Retrieval, Barcelona, Spain*, 2012.

[47] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.

[48] W. Wang, C. Xiao, X. Lin, and C. Zhang. Efficient approximate entity extraction with edit distance constraints. In *Proceedings of the ACM SIGMOD International Conference on Management of data*, pages 759–770, New York, NY, USA, 2009. ACM.

[49] X. Yang, B. Wang, C. Li, J. Wang, and X. Xie. Efficient direct search on compressed genomic data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE), Australia (to appear; preprint at http://www.ics.uci.edu/ xhx/publications/genomecompress.pdf)*.

[50] H. Zhu, G. Kollios, and V. Athitsos. A generic framework for efficient and effective subsequence retrieval. *Proceedings VLDB Endow.*, 5(11):1579–1590, July 2012.