QGramProjector: Q-Gram Projection for Indexing Highly-Similar Strings

Sebastian Wandelt and Ulf Leser

Knowledge Management in Bioinformatics,
Institute for Computer Science,
Humboldt-Universität zu Berlin, Germany
{wandelt,leser}@informatik.hu-berlin.de

Abstract. Q-gram (or n-gram, k-mer) models are used in many research areas, e.g. in computational linguistics for statistical natural language processing, in computer science for approximate string searching, and in computational biology for sequence analysis and data compression. For a collection of N strings, one usually creates a separate positional q-gram index structure for each string, or at least an index structure which needs roughly N times of storage compared to a single string index structure. For highly-similar strings, redundancies can be identified, which do not need to be stored repeatedly; for instance two human genomes have more than 99 percent similarity.

In this work, we propose QGramProjector, a new way of indexing many highly-similar strings. In order to remove the redundancies caused by similarities, our proposal is to 1) create all q-grams for a fixed reference, 2) referentially compress all strings in the collection with respect to the reference, and then 3) project all q-grams from the reference to the compressed strings.

Experiments show that a complete index can be relatively small compared to the collection of highly-similar strings. For a collection of 1092 human genomes (raw data size is 3 TB), a 16-gram index structure, which can be used for instance as a basis for multi-genome read alignment, only needs 100.5 GB (compression ratio of 31:1). We think that our work is an important step towards analysis of large sets of highly-similar genomes on commodity hardware.

Keywords: positional q-grams, k-mer, large sequences, similarity, referential compression.

1 Introduction

Indexing and searching large collections of *highly-similar* strings became a hot and challenging topic during the last years [14], for instance in order to perform population-scale genome analysis[1]. If one can identify the similarities between strings in the collection, then the amount of storage for saving/indexing the strings can be greatly reduced. In general, indexing and searching strings has a long history in computer science research [15]. The literature on string search in

B. Catania, G. Guerrini, and J. Pokorný (Eds.): ADBIS 2013, LNCS 8133, pp. 260–273, 2013.

general is vast and cannot be summarized here; we refer the reader to several excellent surveys [7,13].

Q-grams can be used for indexing large strings following the the seed-and-extend approach [3,19], e.g. to create a disk-based search index [6]. One well-known example from bioinformatics for the use of q-grams (there often called k-mers or seeds) is BLAST[2], which uses q-grams as anchors for finding longer approximate string matches.

In this work, we introduce QGramProjector, where we focus on finding all positions of a given (single) q-gram in a collection of highly-similar strings, for instance, genomes. Two randomly selected human genomes are app. 99% identical [17]. Instead of indexing all q-grams in all genomes, we propose to only explicitly store all q-grams of one chosen reference genome. Furthermore, we propose to compress all other strings in the collection referentially with respect to the reference genome. An index over the referential compressions is used to project all q-grams in the reference string into the other strings of the collection. Since this projection is not complete, because some q-grams do not occur in the reference, we have an additional (smaller) index keeping track of these derivations. In addition, we show a way to further reduce the necessary storage for our q-gram index: We rewrite the reference genome such that longer matches can be encoded.

The most similar work to ours is an index structure for computation of q-gram occurrence frequencies over straight line programs [9][1]. The index's purpose is to retrieve the total occurrence frequencies of all q-grams. For DNA data the index uses half as much size as the original data, achieving a compression ratio of 2:1. However, our projection-based index structure allows for compression (on human genomes) up to 31:1 for $q = 16$. Another related work, string dictionary lookup in a compressed string dictionary with edit distance one is discussed in [4]. Although the authors report nearly optimal complexity results for their matching algorithm (together with a way to trade-off query answering time for index space), there is no practical evaluation of the algorithm. Another approach concerned with searching larger compressed collection of strings is presented in [10]. However, this work has a focus on theoretical results, with a preliminary evaluation on a very small dataset; the program is not publicly available for evaluation on our big datasets.

The remainder of this paper is structured as follows. We introduce the problem of indexing q-grams and the foundations of referential compression in Section 2. Section 3 describes the algorithm used in QGramProjector in detail. Our algorithms are evaluated in Section 4, and Section 5 concludes the paper.

2 Referential Compression and Q-Grams

A *string* s is a finite sequence over an alphabet Σ. The length of a string s is denoted with $|s|$ and the substring starting at position i with length n is denoted

[1] Straight line programs can be seen as a generalization of many dictionary/grammar-based compression formats[18].

$s(i, n)$. $s(i)$ is an abbreviation for $s(i, 1)$. All positions in a string are zero-based, i.e. the first character is accessed by $s(0)$. The concatenation of two strings s and t is denoted with $s \circ t$. A string t is a *prefix of* a string s, if $s = t \circ u$, for some string u. A *q-gram* of string s is an string of length q over Σ.

Definition 1. *A* positional q-gram *of a string s is a tuple $\langle p, g \rangle$, such that $g = s(p, q)$. The* set of all positional q-grams of s *is denoted with $PQG_q(s)$. The* set of positions for a q-gram g *in a string s is defined as $s^g = \{p \mid \langle p, g \rangle \in PQG_q(s)\}$. Given a set of strings $C = \{s_1, ..., s_n\}$, we define $C^g = \{< i, p > \mid s_i \in C \wedge p \in {s_i}^g\}$.*

Given a string s, there exist several approaches to compute s^g. One approach is to explicitly store a map for all q-grams to positions in s. The size of this map data structure (in byte) is usually larger than the size of the input. For instance storing the positions as 4-byte integer values for a string of length n , we have to encode $n-q+1$ positions, which needs $4*(n-q+1)$ bytes. Having m highly-similar strings, the data structure grows roughly by a factor of m, since we have to encode $4*(n_1-q+1)+...+4*(n_m-q+1)$ bytes for the positions, and in addition some kind of ID for each string.

Example 1. Given the two strings $s_1 = ACGACT$ and $s_2 = ACGAAT$, we have

$$PQG_2(s_1) = \{\langle 0, AC\rangle, \langle 1, CG\rangle, \langle 2, GA\rangle, \langle 3, AC\rangle, \langle 4, CT\rangle\}$$
$$PQG_2(s_2) = \{\langle 0, AC\rangle, \langle 1, CG\rangle, \langle 2, GA\rangle, \langle 3, AA\rangle, \langle 4, AT\rangle\}$$

Storing the q-grams for both strings creates redundancies, since the q-grams for positions $0-2$ are the same for s_1 and s_2. For string $s_3 = GACGACT$, we obtain $PQG_2(s_3) = \{\langle 0, GA\rangle, \langle 1, AC\rangle, \langle 2, CG\rangle, \langle 3, GA\rangle, \langle 4, AC\rangle, \langle 5, CT\rangle\}$. Although strings s_1 and s_3 are highly similar as well, there is no obvious redundancy in their positional q-grams. The reason is that all the positional q-grams are shifted one position right for s_3, compared to s_1.

The above examples show that it could be beneficial to identify similarities between strings to reduce the amount of storage for a positional q-gram index of a collection of strings. In the following, we use referential compression to identify and encode these similarities. Based on referentially compressed strings, we devise an index structure to retrieve positional q-grams from a collection of highly-similar strings. We will show that in some cases our q-gram index structure is orders of magnitude smaller than a conventional index. First, we introduce referential compression as it is used in bioinformatics recently [5,11,20]. Referentially compressing a string means to encode the string as a concatenation of substrings from a given reference string. Since there exists no standard format to represent referentially compressed strings, we define a very general notion for encoding referential matches first. The following is taken from [20].

Definition 2. *A* referential match entry *is a triple $\langle start, length, mismatch\rangle$, 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 rme, denoted $|rme|$, is $length + 1$.*

Given a reference ref and a to-be-compressed string s, the idea of referential compression is to find a small set of rme's of s with respect to ref which is a) sufficient to reconstruct s and b) as small as possible.

Definition 3. *Given strings s and ref, a* referential compression *of s with respect to ref, is a list of referential match entries,*

$$\downarrow^{ref}(s) = [\langle start_1, length_1, mismatch_1\rangle, ..., \langle start_n, length_n, mismatch_n\rangle],$$

such that

$$(ref(start_1, length_1) \circ mismatch_1) \circ (ref(start_2, length_2) \circ mismatch_2) \circ ... \circ (ref(start_n, length_n) \circ mismatch_n) = s.$$

Sometimes we use cs instead of $\downarrow^{ref}(s)$, if s and ref are known from the context. The offset *of a referential match entry rme_i in a referential compression $\downarrow^{ref}(s) = [rme_1, ..., rme_n]$, denoted $offset(\downarrow^{ref}(s), rme_i)$, is defined as $\sum_{j<i} |rme_j|$. Given a referential match entry $\langle start, length, mismatch\rangle$, we write $(start, length, mismatch) \in \downarrow^{ref}(s)$, if and only if $\langle start, length, mismatch\rangle$ is an element in the referential compression $\downarrow^{ref}(s)$.*

The offset of a referential match entry in a referential compression corresponds to the position of the entry in the uncompressed string. 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 string. We assume that the reference sequence contains each symbol from Σ, if it does not, then we just add all symbold of Σ to the end of the reference sequence.

Example 2. An example compression for $CGGACAAACTGACGTTCGACG$ with respect to the preselected reference $GACGATCGACGACGGACAAACA$ is as folows. The input is compressed into three referential match entries. The first referential match entry is $\langle 12, 9, T\rangle$, 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 $\langle 5, 5, G\rangle$ is $|\langle 12, 9, T\rangle| + |\langle 10, 4, T\rangle| = 15$.

Clearly, we require the less rme's, the longer the matches, i.e., the shared substrings, are. Therein it does not matter, at which position of the reference these matches lie; in particular, matches need not be in any particular order. To create a referential compression of input string s with respect to ref, our algorithm (Algorithm 1) matches prefixes of s with substrings of ref using a compressed suffix tree on ref. The longest such prefix is removed from s, encoded as a rme and

Algorithm 1. Referential Compression Algorithm

Input: to-be-compressed string s and reference string ref
Output: referential compression $\downarrow^{ref}(s)$ of s with respect to ref

```
1: Let ↓^ref (s) be an empty list
2: while |s| ≠ 0 do
3:     Let pre be the longest prefix of s occurring in ref, and let i be a position
       of an occurrence of pre in ref
4:     if s ≠ pre then
5:         Add ⟨i, |pre|, s(|pre|)⟩ to the end of ↓^ref (s)
6:         Remove the first |pre| + 1 symbols from s
7:     else
8:         Add ⟨i, |pre| − 1, s(|pre| − 1)⟩ to the end of ↓^ref (s)
9:         Remove the prefix pre from s
10:    end if
11: end while
```

added to $\downarrow^{ref}(s)$. The algorithm terminates once s contains no more symbols. Note that a referential compression of a string 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 $\downarrow^{ref}(AA) = [\langle 0, 1, A\rangle]$ and $\downarrow^{ref}(AA) = [\langle 2, 1, A\rangle]$.

Note that a naive storage of rme's does not yield high compression ratios. We store the parsing in a form of delta-encoding, where the position of a rme is stored as a difference to the most recent rme's plus the length of the most recent rme. In addition, we serialize the length value with huffman encoding (the code tree is obtained by precomputation over sequences from different species).

3 Retrieval of Positional Q-Grams

In the following, we first describe a naive way to retrieve all positions of a q-gram in a compressed string $\downarrow^{ref}(s)$. Second, we propose an index structure over a collection of compressed strings. The index structure allows more efficient retrieval of q-gram positions than the naive application (see Section 4). In either case, we do not discuss how to compute ref^g, we assume that can be precomputed in an arbitrary way.

3.1 Retrieving Q-Grams in One String

In Algorithm 2, we describe a simple algorithm to compute the set of positions of a given q-gram in a string s, which is referentially compressed with respect to ref. We call this approach *projection*. Initially, the set $result$ is empty (Line 1). The compressed representation of string s, $\downarrow^{ref}(s)$, is traversed from left to right (Line 2-3). For each position of g in the reference string it is checked, whether the match is subsumed by the current referential match entry; if yes, a relative match is added to the result (Line 4-7). For each q-gram which overlaps the mismatch character, the algorithm checks whether the q-gram is equal to g (Line 8-9). If it is, then a relative match with respect to the referential match entry is added to

Algorithm 2. Projecting q-grams in compressed strings

Input: q-gram g, string s, reference string ref, ref^g
Output: s^g stored in result

1: Let $result = \emptyset$
2: Let $curpos = 0$
3: **for all** $\langle start, length, mismatch\rangle \in \downarrow^{ref}(s)$ **do**
4: Let $refmatches = \{p \mid p \in ref^g \wedge (p \geq start) \wedge (p + |g| \leq start + length)\}$
5: **for all** $p \in refmatches$ **do**
6: $result = result \cup \{curpos + (p - start)\}$
7: **end for**
8: Let $t = s[curpos + length - (|g| - 1), 2 * |g| - 1]$
9: **for all** $p \in t^g$ **do**
10: $result = result \cup \{curpos + length - (|g| - 1) + p)\}$
11: **end for**
12: $curpos = curpos + |\langle start, length, mismatch\rangle|$
13: **end for**
14: **return** result

the result (Line 10). At the end of the loop, the current position is set to the the beginning of the next referential match entry (Line 12). For Algorithm 2, ref^g can be precomputed (and stored), but t^g is computed at runtime.

Example 3. Let the reference be $ref = ACGACTAT$, $s_1 = GACGACTAC$. We obtain $\downarrow^{ref}(s_1) = \{\langle 2, 3, G\rangle, \langle 2, 4, C\rangle\}$. Given $g = AC$, we have $ref^g = \{0, 3\}$. The for loop (Line 3-13) of Algorithm 2 iterates two times. In the first iteration for referential match entry $\langle 2, 3, G\rangle$, we have $refmatches = \{3\}$ and add $\{0 + (3-2)\}$ to $result$. We have $t = CGA$ and $t^g = \emptyset$, and thus no additional matches are being added in the first iteration. In the second iteration for referential match entry $\langle 2, 4, C\rangle$, we have $refmatches = \{3\}$ again and add $\{4+(3-2)\}$ to $result$. We have $t = AC$ and $t^g = \{0\}$. Thus, $\{4 + 4 - (2 - 1) + 0)\}$ is added to result in Line 10. After the execution of Algorithm 2 we obtain $result = \{1, 5, 7\}$.

Algorithm 2 has three starting points for optimization, if a collection of compressed strings is to be searched. First, the whole referentially compressed string needs to be traversed in order to find all q-grams. Second, we need to keep a copy of the uncompressed string (Line 8), or at least decompress parts of the string during the search for each q-gram. Third, we have to run Algorithm 2 on each compressed sequence separately. Partial decompression and repeated interval containment checks make this algorithm not scalable for a large number of strings in the to-be-searched collection. Both issues can be addressed by using appropriate index structures on referential match entries. We introduce these index structures in the following subsection.

3.2 Index Structure for Q-Gram Projection

We use two index structures for improving scalability of q-gram projection in collections of highly-similar referentially compressed strings. First, we devise an

index structure for managing all the referential match entries. Second, we develop an index structure for the q-grams overlapping mismatch characters, to avoid partial decompression during search.

Projection of Reference Q-Grams. Given a q-gram g, a reference string ref, all positions of g in ref, ref^g, and a set of referentially compressed strings $C = \{cs_1, ..., cs_n\}$, we want to find each $\langle start, length, mismatch\rangle$ in each compressed string $cs_i \in C$, such that there exists a $p \in ref^g$ with $(p \geq start)$ and $(p + |g| \leq start + length)$ (Line 4-7 of Algorithm 2). Since all referential match entries can be understood as intervals over the whole reference string, we use interval trees [12] for efficiently querying referential match entries. An interval tree for n intervals uses $O(n)$ storage and can be used to answer containment queries, i.e. find all intervals containing a given query point/interval, in time $O(k + \log n)$, where k is the number of matching intervals.

Definition 4. *A* reference interval for ref *is a tuple* $\langle start, length\rangle$*, such that* $0 \leq start$*,* $0 \leq length$*, and* $start + length < |ref|$*. A* occurrence annotation *is a tuple of the form* $\langle id, pos\rangle$*, where id and pos are natural numbers. We use id to store the identifier of a sequence inside the collection. Given a set of referentially compressed strings* $C = \{cs_1, ..., cs_n\}$ *with respect to* ref*, the* annotated reference interval set *for a reference* ref *and* C*, denoted* $ARIS_C^{ref}$*, is a tuple* $\langle intervals, \mathcal{T}, annotations\rangle$*, such that intervals is a set of reference intervals for* ref*,* $\mathcal{T}$ *is a interval tree over these intervals, and annotations is a function from intervals to a set of occurrence annotations, such that*

$$< id, pos > \in annotations(\langle start, length\rangle) \iff$$
$$cs_{id} \in C \wedge \exists m.\langle start, length, m\rangle \in cs_{id} \wedge offset(cs_{id}, \langle start, length, m\rangle) = pos).$$

Given $ARIS_C^{ref}$, we can project all q-grams from ref into referential match entries occurring in referentially compressed strings in C.

Searching Q-Grams Overlapping Mismatch Characters. In order to find all q-grams in compressed strings we need to take into account q-grams overlapping mismatch characters as well. We define a map structure which keeps track of all positions of q-grams overlapping at least one mismatch character in a compressed string in the collection.

Definition 5. *Given a set of referentially compressed strings* $C = \{cs_1, ..., cs_n\}$*, the* annotated overlap map *for the reference* ref *and* C*, denoted* AOM_C^{ref}*, is* $\langle grams, annotations\rangle$*, such that grams is a subset of* Σ^q *and annotations is a function from grams to a set of occurrence annotations, such that*

$$< id, pos > \in annotations(w) \iff (cs_{id} \in C \wedge \exists rme, d.rme \in cs_{id} \wedge$$
$$offset(cs_{id}, rme) + |rme| - d = pos) \wedge 1 \leq d \leq q \wedge \uparrow^{ref} (cs_{id})(pos, q) = w).$$

Algorithm 3. Searching q-grams in compressed strings

Input: ref, referential q-gram index $\langle PQG_q(ref), \langle i, \mathcal{T}, annotation_1\rangle, \langle grams, annotation_2\rangle\rangle$, q-gram g
Output: set of retrieved positions in $result$

```
1: Let result = ∅
2: for all p ∈ ref^g do
3:     Use T to identify all intervals in i which contain the interval ⟨p, q⟩, add all these intervals to refm
4:     for all ⟨start, length⟩ ∈ refm do
5:         for all ⟨id, pos⟩ ∈ annotation_1(⟨start, length⟩) do
6:             result = result ∪ {⟨id, pos + (p − start)⟩}
7:         end for
8:     end for
9: end for
10: result = result ∪ annotation_2(g)
11: return result
```

Referential Q-Gram Index. We combine the positional q-grams of a reference with an annotated reference interval set and an annotated overlap map to obtain a combined index structure for referentially compressed strings.

Definition 6. *Given a set of referentially compressed strings* $C = \{cs_1, ..., cs_n\}$ *with respect to a reference string* ref*, a* referential q-gram index *is defined as* $\langle PQG_q(ref), ARIS_C^{ref}, AOM_C^{ref}\rangle$.

We show our algorithm for retrieving all occurrences of a given q-gram in a referential q-gram index in Algorithm 3. The algorithm iterates over all matches for g in the reference string (Line 2-9). For each match in the reference, the reference intervals containing $\langle p, q\rangle$, i.e. the to-be-searched q-gram, are collected in $refm$ (Line 3). For all intervals in $refm$, the annotation $annotation_1$ is used to retrieve all projected positions in referentially compressed strings and add these positions to the result (Line 4-6). Finally, all occurrences of g overlapping one mismatch character are being added using the annotation map $annotations_2$ (Line 10).

Lemma 1. *For a given set of strings* $S = \{s_1, ..., s_n\}$*, a q-gram* g*, a reference string* ref*, and a set of referential compressions* $C = \{cs_1, ..., cs_n\}$ *for* S *with respect to* ref*, we have that after applying Algorithm 3 with referential q-gram index* $\langle PQG_q(ref), ARIS_C^{ref}, AOM_C^{ref}\rangle$*,* $result = S^q$.

3.3 Example

We conclude this section with a complete example on retrieving all positions of a given q-gram from a set of strings.

Example 4. Let S be a collection of strings $\{s_1, s_2, s_3, s_4\}$, $s_1 = ATCAGAATCT$, $s_2 = CATCGATCAGA$, $s_3 = ATCAGACATCGA$, and $s_4 = AGCCAAAATCT$.

Referentially compressing S with respect to $ref = ATCAGCATCG$ yields $C = \{cs_1, cs_2, cs_3, cs_4\}$ with

$$cs_1 = \{\langle 0,5,A\rangle, \langle 6,3,T\rangle\}, cs_2 = \{\langle 5,5,A\rangle, \langle 1,4,A\rangle\},$$
$$cs_3 = \{\langle 0,5,A\rangle, \langle 5,5,A\rangle\}, cs_4 = \{\langle 3,3,C\rangle, \langle 0,1,A\rangle, \langle 6,3,T\rangle\}$$

Let q=2, then we obtain the following index structures:

- $ARIS_C^{ref} = \langle i, \mathcal{T}, ann_1\rangle$, with $i = \{\langle 0,5\rangle, \langle 6,3\rangle, \langle 1,4\rangle, \langle 5,5\rangle, \langle 3,3\rangle, \langle 0,1\rangle\}$ and

$$ann_1 = \{\langle 0,5\rangle \rightarrow \{\langle 1,0\rangle, \langle 3,0\rangle\}, \langle 6,3\rangle \rightarrow \{\langle 1,6\rangle, \langle 4,6\rangle\}, \langle 1,4\rangle \rightarrow \{\langle 2,6\rangle\},$$
$$\langle 5,5\rangle \rightarrow \{\langle 2,0\rangle, \langle 3,6\rangle\}, \langle 3,3\rangle \rightarrow \{\langle 4,0\rangle\}, \langle 0,1\rangle \rightarrow \{\langle 4,4\rangle\}\}$$

- $AOM_C^{ref} = \langle grams, ann_2\rangle$, with

$$grams = \{GA, AA, CT, AT, AC, CC, CA\}$$
$$ann_2 = \{GA \rightarrow \{\langle 1,4\rangle, \langle 2,4\rangle, \langle 2,9\rangle, \langle 3,4\rangle, \langle 3,10\rangle\}$$
$$, AA \rightarrow \{\langle 1,5\rangle, \langle 4,5\rangle, \langle 4,6\rangle\},$$
$$CT \rightarrow \{\langle 1,8\rangle, \langle 4,9\rangle\}, AT \rightarrow \{\langle 2,5\rangle\},$$
$$AC \rightarrow \{\langle 3,5\rangle\}, CC \rightarrow \{\langle 4,2\rangle\}, CA \rightarrow \{\langle 4,4\rangle\}\}$$

Without referential compression we would need to store 40 two-grams for the four original strings in S. With referential compression only 14 explicit two-grams in the annotated overlap map plus 9 two-grams for the reference string (plus the overhead $ARIS_C^{ref}$) are necessary. Although, in this toy example, we do not save space[2], our evaluation in the next section shows that the approach does save much space in case of many highly-similar strings in practice.

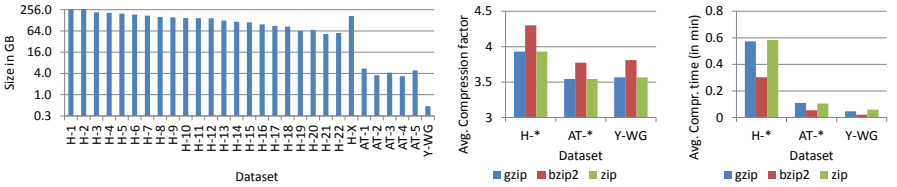
4 Evaluation

In the following section, we evaluate our proposed compression scheme. All experiments have been run on a Acer Aspire 5950G with 16 GB RAM and Intel Core i7-2670QM, on Fedora 16 (64-Bit, Linux kernel 3.1). All size measures are in byte, e.g. 1 MB means 1,000,000 bytes. Below, the term compression factor is used to denote the inverse compression ration, e.g. a compression factor of 100 means a compression ratio of 100:1.

We have evaluated our algorithms for referential compression and q-gram projection on three biological datasets: a collection of human genomes, a collection of genomes from Arabidopsis thaliana, and a collection of yeast genomes. The raw size of the datasets is shown in Figure 1(a).

Our first datasets of human genomes was created from 1092 genomes of the 1000 Genome project[1].We use H-# to represent the set of all 1092 sequences

[2] Note that in this example the strings in S are quite short, S is rather small, and the referential match entries are very short as well.



(a) Evaluation datasets. (b) Standard compression algorithms

Fig. 1. Size of evaluation datasets (left) and initial evaluation with standard compression algorithms (right)

for human Chromosome #, e.g. H-1 for human Chromosome 1. The union of all 23 human datasets is denoted with H-*. The largest human dataset is H-1 at 272.1 GB, the smallest dataset is H-22 at 55.9 GB, and the size of H-* is 3.3 TB. Our datasets for Arabidopsis thaliana are taken from the 1001 Genomes project[21] from release GMINordborg2010.[3]. We have extracted 180 genomes with each 5 chromosomes. The Arabidopsis thaliana datasets are prefixed with AT, e.g. AT-1 stands for 180 Chromosome 1 of Arabidopsis thaliana. The union of all 5 Arabidopsis thaliana datasets is denoted with AT-*. The largest Arabidopsis thaliana dataset is AT-1 at 5.4 GB, the smallest dataset is AT-4 at 3.3 GB, and the size of AT-* is 21.4 GB The last dataset, is a collection of yeast genomes[4]. In total, we have downloaded 38 yeast strains, each of them was provided in FASTA format. The yeast dataset is denoted with Y-WG. The size of Y-WG is 0.4 GB.

4.1 Existing Standard Compression Algorithms

We have used three standard compression programs to create initial statistics about self-referential compression: gzip, bzip2, and zip. For each species and each chromosome, we have randomly selected five sequences and applied each of the compression algorithms. The results are shown in Figure 1(b). bzip2 is the best compression program among the three tested programs. The best average compression ratio is obtained by bzip2 for all three species and bzip2 is the fastest compression program as well, outperforming the other two programs by a factor of two in average. Using bzip2, it should be possible to compress H-* down to 0.7 TB using bzip2, but the run time is expected to be around 126 hours. AT-* can be compressed down to 5.6 GB in 48 minutes. The compression factor is relatively stable within species for H-*(min: 3.91 for H-3, max: 5.82 for H-22) and AT-*(min: 3.74 for AT-2, max: 3.80 for AT-1).

[3] `http://1001genomes.org/data/GMI/GMINordborg2010/releases/current/`
[4] `http://www.yeastgenome.org/download-data/sequence`

						search time (s)		
	approach	data size (MB)	index size (MB)	total size (MB)	indexing time (h)	min	max	avg
H-22	grep	55,932		55,932		427.92357	23,821.69608	8,826.85440
	decomp/search	78		78		98.28773	251.16323	218.43293
	CST		139,776	139,776	5.16	0.01420	55.69200	31.13650
	FMIndex 2	-	-	-	-	-	-	-
	SQLite		*1,183,370*	1,183,370	*490.27*	*0.00963*	*0.99628*	*0.64103*
	QGramProjectionOnline	78		78		75.64537	82.30472	77.45636
	QGramProjector		1,800	1,800	*0.32*	0.00005	0.31250	0.03413

						search time (s)		
	approach	data size (MB)	index size (MB)	total size (MB)	indexing time (h)	min	max	avg
AT-1	grep	5,477		5,477		65.36063	1330.07628	756.92232
	decomp/search	48		48		20.73434	36.82837	32.41139
	CST		10,373	10,373	0.62	0.00324	4.86677	1.87505
	FMIndex 2		1,926	1,926	0.32	0.03300	0.72100	0.03800
	SQLite		*113,568*	*113,568*	*47.33*	*0.00148*	*0.01294*	*0.00530*
	QGramProjectionOnline	48		48		44.21570	60.90107	54.55710
	QGramProjector		1,200	1,200	*0.20*	0.00005	0.01360	0.00129

						search time (s)		
	approach	data size (MB)	index size (MB)	total size (MB)	indexing time (h)	min	max	avg
Y-WG	grep	473		473		3.99718	109.63334	34.42344
	decomp/search	6		6		3.04898	3.42124	3.15477
	CST		642	642	0.05	0.00046	0.20634	0.09337
	FMIndex 2		156	156	0.03	0.01900	0.21200	0.02500
	SQLite		*9,984*	*9,984*	*3.96*	*0.00014*	*0.00370*	*0.00089*
	QGramProjectionOnline	6		6		9.34890	9.45812	9.36278
	QGramProjector		263	263	*0.02*	0.00005	0.00231	0.00026

Fig. 2. Comparison of approaches for finding 16-mers

4.2 Indexing Sequences

In order to compare our approach to related work, we have selected the following competitors:

grep : Unix grep on the uncompressed raw sequences on the hard disk. We have used grep with options '-o -b' to only report positions of matches.
decomp/search : Index-less search with decompression from the hard disk on the fly.
CST : One compressed suffix tree[16] per sequence.
FMindex 2 : a compressed substring index[8] over all sequences based on the Burrows-Wheeler transform.
SQLite : A SQLite database with all pre-computed k-mers for each sequence.
QGramProjectionOnline : an online version of our approach, where ARIS and AOM are created from the compressed files on-the-fly.
QGramProjector : an index-based version with precomputed ARIS and AOM for all sequences.

We have used 300 queries (16-mers), 100 each of the three species in the dataset. The results for datasets H-22, AT-1 and Y-WG are shown in Figure 2. For the other datasets we obtained similar results (within a species and with increasing sequence length, all tested approach show roughly linear index size and runtime behaviour). Note that we have used complete datasets now, e.g. 1092 sequences for the human genomes.

Using grep, the search speed is limited by the throughput of the hard disk. The hard-disk throughput is no longer a limit with naive search/decompress on compressed sequences. The average search time is reduced by a factor of 10-40.

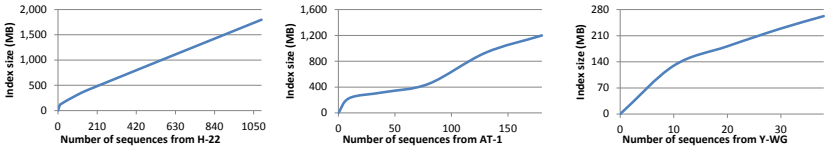


Fig. 3. Size of our index structure for different number of sequences

In addition, the amount of storage is clearly reduced, e.g. by a factor of 720:1 for for H-22. Using compressed suffix trees[16], the search times are three and more orders of magnitudes shorter than before. If there are many q-gram positions to be retrieved, then the compressed suffix tree solution slows down recognizably, e.g. down to 55.7 seconds for H-22. We were unable to run FMIndex 2 on the human genome datasets. For H-22 and even only one sequence, FMIndex 2 never returns a after sequences larger than 40 MB (we let it run on one sequence for several hours without any result). For the other datasets FMIndex 2 worked well. It is interesting that the index is actually 3-4 times smaller then the input dataset (as opposed to compressed suffix trees). In average, FMIndex 2 is faster than CST, which is mainly due to the fact that often occuring k-mers can be retrieved more efficiently. Our implementation based on SQLite shows very good average search times. The index structure becomes very large, as expected, since the position of each k-mer for each sequence has to be stored. For H-22 and AT-1 the indexing size and search times were extrapolated from a smaller set of 50 sample sequences.

QGramProjectionOnline (without precomputed index) has similar search times like decomp/search. It seems like the overhead of computing the index structures on the fly for each sequence takes too much time, in general. For highly-similar sequence, e.g. H-22, there can be small advantages, while for less similar sequences the projection algorithm doe snot pay off. Finally, QGramProjector has the best min, max, and avg search times in the test for all species. In average, even 3-4 times faster than using all precomputed k-mers in a SQLite database. The indexing overhead is in between FMIndex 2 and CST for not so similar sequences, but clearly outperforms both for highly-similar sequences.

We analyse the size of QGramProjector's index for a different number of sequences in Figure 3. The index grows linearly with the number of sequences for our human genome dataset H-22, as shown in Figure 3(left). This is because of the high similarity among the sequence. Each sequences adds roughly a similar amount of new ARIS and AOM entries. For AT-1 and Y-WG, Figure 3(middle and right), the picture is different: the size of the index grows not as regular as H-22. The reason is that the sequences are less similar (with respect to the reference) and each sequence adds a different amount of new annotations to ARIS and AOM.

In Figure 4, we show that retrieval time for 100.000 randomly created 16-grams over H-*, which corresponds to 3 TB of data. This test was performed on a server system with a TB of main memory. Each point corresponds to one

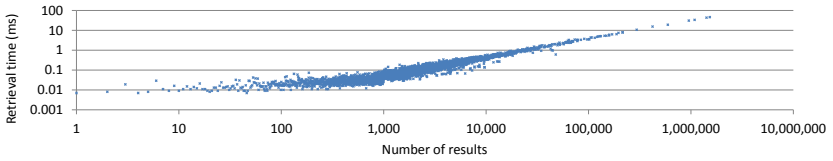


Fig. 4. Retrieval of 16-grams

q-gram (only q-grams with at least one occurrence in any of the 1000 genomes are shown). The 16-gram retrieval time is between 0.01 ms and 120 ms (average 0.64 ms; median 0.24 ms). Once a q-gram occurs in many sequences at different positions, the retrieval time increases linearly with the number of results, which is mainly caused by the time spent on decompressing the AOM/ARIS-annotations. We apply a special delta encoding on the sorted positions in occurrence annotations. Without such a compression of the annotations the index structure would increase by a factor of 2-4.

5 Conclusions

This work is another step towards analysis of a collection of highly-similar strings based on q-grams. We have shown that our approach allows to manage the q-grams of 1000 genomes using orders of magnitude less memory than the actual data size, and thus can be kept in main memory of a commodity hardware. Clearly, q-gram projection works particularly well for highly-similar sequences with a small alphabet size and large chunks of repeats across strings. Whether it can be directly applied to other areas is still to be evaluated. Another direction for future work is to implement a system for approximate string searching over large sets of genomes, using the q-gram index and following the seed-and-extend approach [3]. Finally, improving average retrieval times for q-grams is an important issue. The main challenge seems to retrieve q-grams with many occurrences. Alternatively, it would be interesting to define a notion of important matches. Since the reference sequence is always searched first, one could try to anticipate the overall number of matches, before retrieving all results. If the reference contains already more matches than a given threshold, one could limit the retrieval over the compressed sequences.

References

1. 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. Nature, 467(7319), 1061–1073 (October 2010)
2. Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of Molecular Biology 215(3), 403–410 (1990)
3. Baeza-Yates, R.A., Perleberg, C.H.: Fast and practical approximate string matching. In: Apostolico, A., Galil, Z., Manber, U., Crochemore, M. (eds.) CPM 1992. LNCS, vol. 644, pp. 185–192. Springer, Heidelberg (1992)

4. Belazzougui, D., Venturini, R.: Compressed string dictionary look-up with edit distance one. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 280–292. Springer, Heidelberg (2012)
5. Deorowicz, S., Grabowski, S.: Robust Relative Compression of Genomes with Random Access. Bioinformatics (September 2011)
6. du Mouza, C., Litwin, W., Rigaux, P., Schwarz, T.: As-index: a structure for string search using n-grams and algebraic signatures. In: Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, pp. 295–304. ACM, New York (2009)
7. Ferragina, P.: String algorithms and data structures. CoRR, abs/0801.2378 (2008)
8. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581 (2005)
9. Goto, K., Bannai, H., Inenaga, S., Takeda, M.: Speeding up q-gram mining on grammar-based compressed texts. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 220–231. Springer, Heidelberg (2012)
10. Kreft, S., Navarro, G.: On compressing and indexing repetitive sequences. Theor. Comput. Sci. 483, 115–133 (2013)
11. Kuruppu, S., Puglisi, S., Zobel, J.: Optimized relative lempel-ziv compression of genomes. In: Australasian Computer Science Conference (2011)
12. McCreight, E.: Efficient algorithms for enumerating intersection intervals and rectangles. Technical report, Xerox Paolo Alte Research Center (1980)
13. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. 33(1), 31–88 (2001)
14. Navarro, G.: Indexing highly repetitive collections. In: Arumugam, S., Smyth, B. (eds.) IWOCA 2012. LNCS, vol. 7643, pp. 274–279. Springer, Heidelberg (2012)
15. Navarro, G., Raffinot, M.: Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences. Cambridge University Press, New York (2002)
16. Ohlebusch, E., Fischer, J., Gog, S.: CST++. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 322–333. Springer, Heidelberg (2010)
17. Reich, D.E., Schaffner, S.F., Daly, M.J., McVean, G., Mullikin, J.C., Higgins, J.M., Richter, D.J., Lander, E.S., Altshuler, D.: Human genome sequence variation and the influence of gene history, mutation and recombination. Nature Genetics 32(1), 135–142 (2002)
18. Rytter, W.: Application of lempel–ziv factorization to the approximation of grammar-based compression. Theor. Comput. Sci. 302(1-3), 211–222 (2003)
19. Sutinen, E., Tarhio, J.: On using q-gram locations in approximate string matching. In: Spirakis, P.G. (ed.) ESA 1995. LNCS, vol. 979, pp. 327–340. Springer, Heidelberg (1995)
20. Wandelt, S., Leser, U.: Adaptive efficient compression of genomes. Algorithms for Molecular Biology 7, 30 (2012)
21. Weigel, D., Mott, R.: The 1001 Genomes Project for Arabidopsis thaliana. Genome Biology 10(5), 107+ (2009)