MRCSI: Compressing and Searching String Collections with Multiple References

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

ABSTRACT

Efficiently storing and searching collections of similar strings, such as large populations of genomes or long change histories of documents from Wikis, is a timely and challenging problem. Several recent proposals could drastically reduce space requirements by exploiting the similarity between strings in so-called reference-based compression. However, these indexes are usually not searchable any more, i.e., in these methods search efficiency is sacrificed for storage efficiency. We propose Multi-Reference Compressed Search Indexes (MRCSI) as a framework for efficiently compressing dissimilar string collections. In contrast to previous works which can use only a single reference for compression, MRCSI (a) uses multiple references for achieving increased compression rates, where the reference set need not be specified by the user but is determined automatically, and (b) supports efficient approximate string searching with edit distance constraints. We prove that finding the smallest MRCSI is NP-hard. We then propose three heuristics for computing MRCSIs achieving increasing compression ratios. Compared to state-of-the-art competitors, our methods target an interesting and novel sweet-spot between high compression ratio versus search efficiency.

Keywords

Indexing, compression, dissimilar strings

1. INTRODUCTION

Information systems for strings are an old [19], yet still highly relevant research topic. The two most fundamental problems for such systems are storage efficiency and search efficiency: How can large string collections be stored in as little space as possible while still performing string searching as fast as possible. One particularly important type of search is k-approximate substring search: Given the collection of strings S and a query q, find all substrings of strings in S similar to q within a given error threshold k. Several applications exist for which scalability of such operations is tremendously important. We give two examples. In **Bioinformatics**, next generation sequencing produces billions of DNA sequences (called reads) in a single day, each between 50 and 400 characters long. Analyzing this data first requires to determine the position of each read within the sequenced genome [14]. This problem, called read mapping, amounts to perform a k-approximate search for each read in the genome. In **entity search**, one needs to find the named entities in text collections despite spelling variations [3]. For instance, the Technische Universität Berlin could be represented as 'Technische-Universität Berlin', 'Technische-Universitaet Berlin', or 'Technische- Universität in Berlin'. Such problems also frequently occur in series of Wikipedia versions [24, 26]. Given a set of versions, the task is to find all k-approximate occurrences of the entity in the collection.

Standard approximate string search methods create an index on either 1) each string in a collection separately or 2) a concatenation of all strings. Both approaches typically lead to an index that is larger than the sum of the length of all indexed strings. However, in applications like those just described, the indexed documents are highly similar to each other, which can be exploited to drastically reduce index sizes and indexing times. A number of methods have been proposed recently that first referentially compress each string in the collection against a pre-selected reference string [37, 38]. Then, searching is split up into two subtasks: 1) search the reference string and 2) search all deviations of strings from the reference as encoded by referential compression. However, all these methods are only applicable if all indexed strings are highly similar to each other. This is, for instance, a problem when indexing genomes, since compressing a set of human genomes actually means compressing 24 sets of highly similar chromosomes with almost no similarity between the sets. Another problematic scenario for these methods are histories of Wiki sites, as storing them requires methods that can deal with a changing set of pages, as pages may be removed or added. A second problem with being restricted to a single reference is that these methods cannot exploit similarities in-between the compressed strings – but only similarities between those strings and the reference. Consider consecutive versions of Wikipedia articles: Typically, each version is very similar to its version neighbours, but the first recorded version is usually quite different from the most recent one, and no single version (i.e. potential reference) is similar to all other versions.

In this paper, we propose the Multi-Reference Compressed Search Index (MRCSI) which is capable of working with multiple references to increase compression rates, chooses these references automatically, and also provides fast approximate search. The fundamental idea is simple (see Figure 1): MRCSI builds a compression hierarchy where strings are compressed w.r.t. various other strings. Implementing this idea yields a number of challenges, in particular to find appropriate algorithms and data structures (a) for allowing high compression speed, (b) for choosing the best compression hierarchy in terms of space, and (c) for achieving efficient k-approximate search in a given compression hierarchy. Our paper

Proceedings of the VLDB Endowment, Vol. 8, No. 5
Copyright 2015 VLDB Endowment 2150-8097/15/01.

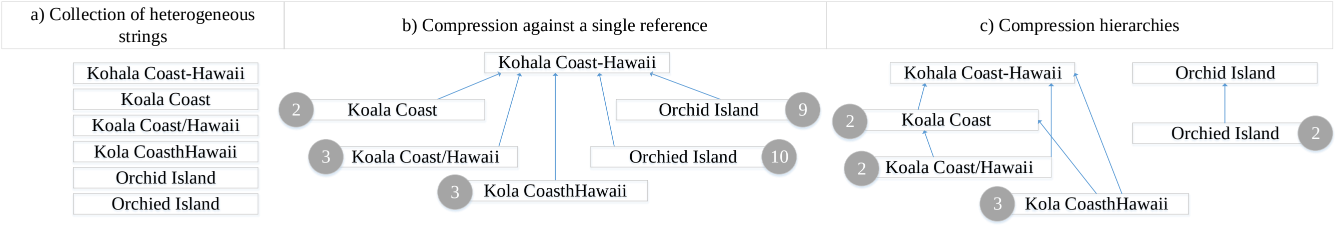


Figure 1: Basic idea of multi-reference referential compression. Boxes are to-be-compressed strings, edges represent referential compression; for instance, s='Koala Coast' could be represented with reference t='Kohala Coast-Hawaii' as $[(0,2)$, a, $(4,8)]$, indicating that $s = t(0,2)\circ$ a $\circ t(4,8)$. Numbers in circles show the number of entries in the compressed representation. Description: a) Collection of dissimilar strings. b) Prior work uses only one reference. Compression potential is lost. (c) Compression hierarchies exploit identical substrings.

provides solutions to all these challenges:

1. We develop a framework for multiple reference compressed search indexes (MRCSIs) for any kind of strings, e.g. text, biological sequences, and semi-structured documents.
2. The question arises which MRCSI is the best. We present a theoretical model for estimating the size of a given MRCSI.
3. We prove that finding the space-optimal MRCSI is NP-hard.
4. We propose three heuristics of increasing complexity for creating a MRCSI from a given string collection: Starting from flat compression forests over general compression forests to DAG-structured compression hierarchies. In all these settings, reference strings are picked automatically.
5. Experimental results on real-world datasets show that our heuristics build indexes up to 90% smaller than the state-of-the-art while providing comparable search times.

The rest of this paper is structured as follows: We review related work in Section 2. A formalization of string databases and referential compression is defined in Section 3. We present our framework for multi-reference compressed search indexing in Section 4 and also prove that finding the most compact MRCSI under a certain, intuitive cost model is NP-hard. We introduce three heuristics for building increasingly complex MRCSIs in Section 5. We report on experimental results and compare our algorithms to the most related approaches in Section 6. The paper concludes in Section 7.

2. RELATED WORK

We give an overview on prior work concerning string compression algorithms and report how compression techniques are used for solving indexing/approximate search problems over document collections in different areas.

String Compression: Compression has a long tradition in computer science and related areas. A relative novel development are referential compression algorithms [7, 17, 21, 30, 36] which encode (long) substrings of the to-be-compressed input with reference entries to another fixed string, called the reference. The compression ratio of these methods grows with increasing similarity between to-be-compressed strings and the reference and can skyrocket for very similar strings [8, 35]. For instance, standard compression schemes achieve compression rates of 4:1–8:1 for DNA sequences, while referential compression algorithms reach compressions rates of more than 1,000:1 [7, 36] when applied to a set of human genomes.

Compressing Semi-Structured Documents: Compressed inverted text indexes [4, 29] are used for indexing versioned documents. The general idea is to compress the list of occurrences for each word using a standard compression algorithm, for instance, PFOR-DELTA [39]. RLZ [16, 17] is a tool for referentially compressed storage of web-collections achieving high compression ratios, but does not directly support approximate search; in Section 6, we will compare against a modified version of this tool which does allow searching. A technique for increasing the compression ratio of RLZ-based compression was proposed recently [34]: By eliminating rarely used parts from the dictionary, significant improvements over the compression ratios are reported, but - as RLZ - without a search index.

Compressed Indexing in Specific Domains: Managing string collections is particularly important in Bioinformatics. Several algorithms recently emerged that compress a set of genomes against a reference. GenomeCompress [38] creates a tree representation of differences between a collection of genomes using an alignment technique. In our prior work, we presented RCSI [37] which uses an alignment-free referential compression algorithm [36] and additionally builds a compressed index for allowing approximate searches. We showed that RCSI outperforms GenomeCompress by at least an order of magnitude. Three very recent proposals are [6, 10, 33], which either 1) achieve impressive compression rates but exploit the existence of a multiple sequence alignment for all strings [6, 10] (very time consuming for long/many strings), or 2) do not find all matches for a given query [33], since they only construct a so-called pan-genome, which contains less information than the collection of sequences [6].

Theoretical Computer Science: Given upper bounds on pattern lengths and edit distances, [11] preprocesses the to-be-indexed text with LZ77 to obtain a filtered text, for which it stores a conventional index. But [11] has not been demonstrated to scale up to multi-gigabyte genomic data [6]. Grammar-based compressors, as XRAY [2], RE-PAIR [22], and the LZ77-based compressor LZ-End [20], have enormous construction requirements, limiting their application to small collections. Other work addresses related problems over highly-similar sequence collections, e.g., document listing [12] and top-k-retrieval [27].

3. PRELIMINARIES

A *string* s is a finite sequence of characters from an alphabet Σ. The concatenation of two strings s and t is denoted with $s \circ t$. A string s is a *substring* of string t, if there exist two strings u and v (possibly of length 0), such that $t = u \circ s \circ v$. The length of a string s is denoted with $|s|$ and the substring 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 string are zero-based, i.e., the first character is accessed by $s(0)$. Given strings 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 (replacing one symbol in s, deleting one symbol from s, adding one symbol to s). Given a string s and a string q, the set of *all k-approximate matches in s with respect to q*, denoted *search*$(s)^k_q$,

Algorithm 1 Compression against multiple references

```
Input: to-be-compressed sequence $s$ and collection of reference sequences $REF = \{ref_1, ..., ref_n\}$
Output: referential compression $rcs$ of $s$ with respect to $REF$
Let $rcs$ be an empty list
while $|s| \neq 0$ do
    Let $pre$ be the longest prefix of $s$, such that $(pos, pre) \in search(ref_i)^0_{pre}$, for a number $pos$, and there exists no $1 \leq j \leq n$, with $j \neq i$ and $ref_j$ contains a longer prefix of $s$ than $ref_i$
    if $s \neq pre$ then
        Add $(ref_i, pos, |pre|, s(|pre|))$ to the end of $rcs$
        Remove the first $|pre|+1$ symbols from $s$
    else
        Add $(ref_i, pos, |pre|-1, s(|pre|-1))$ to the end of $rcs$
        Remove the prefix $pre$ from $s$
    end if
end while
```

is defined as the set $search(s)^k_q = \{(i, s(i,j)) \mid s(i,j) \sim_k q\}$. This definition is naturally extended to searching a database (or collection) of strings: A *string database* S is a collection of strings $\{s_1, ..., s_n\}$. Given a query q and a parameter k, we define the *set of all k-approximate matches for q in S* as $DBsearch(S)^k_q = \{(l, search(s_l)^k_q) \mid s_l \in S\}$. For example *Orchid* $\sim_1$ *Orchied*, because the symbol e can be removed from *Orchied* with one delete operation to obtain *Orchid*. If $S = \{s_1, s_2\}$, with $s_1 = Orchid$ and $s_2 = Orchied$, $DBsearch(S)^1_{hid} = \{(1, \{(3, hi), (3, hid), (4, id)\}), (2, \{(3, hi), (3, hie), (3, hied)\})\}$.

Intuitively, the problem we study in this paper is the following: How can we encode a string database S, such that (a) the encoding requires as little space as possible and (b) k-approximate searches can be performed efficiently. The basic technique we use for storing strings in a compact manner is to compress them by only storing differences to other strings, called references [7, 21, 36].

DEFINITION 1 (REFERENTIAL COMPRESSION). *Let REF be a set of reference strings and s be a to-be-compressed string. A tuple* $rme = (refid, start, length, mismatch)$ *is a* referential match entry (RME) *if refid* $\in$ *REF is a (identifier for a) reference string, start is a number indicating the start of a match within refid, length denotes the match length, and mismatch denotes a symbol. A* referential compression *of s w.r.t. REF is a list of RMEs*
$rcs = [(refid_1, start_1, length_1, mismatch_1), ...,$
$(refid_n, start_n, length_n, mismatch_n)]$ *such that*
$(refid_1(start_1, length_1) \circ mismatch_1) \circ ... \circ$
$(refid_n(start_n, length_n) \circ mismatch_n) = s.$

The *size* of a RME $(refid, start, length, mismatch)$ is defined as $length + 1$. The *offset* of a RME rme_i in a referential compression $rcs = [rme_1, ..., rme_n]$, denoted $offset(rcs, rme_i)$, is defined as $\sum_{j<i} |rme_j|$.

Algorithm 1, shown here only for illustration, is a simple method for computing a referential compression against multiple references. It compresses the input string s from left to the right, replacing the longest prefix of s which can be found in any of the references in $REF = \{ref_1, ..., ref_n\}$ with a RME. This algorithm is space-optimal, if the storage necessary for RMEs is uniform [5]. Decompressing a referentially compressed string rcs is equally simple: We traverse rcs from left to right and replace each RME with its decompressed string, where the decompression of a single RME $rme = (refid, start, length, mismatch)$ is $refid(start, length)$ with the mismatch character *mismatch* concatenated to the end.

EXAMPLE 1. *We have a set of strings* $S = \{s_1, ..., s_6\}$ *with*

$s_1 = Kohala\ Coast\text{-}Hawaii, s_2 = Koala\ Coast,$

$s_3 = Koala\ Coast/Hawaii, s_4 = Kola\ CoasthHawaii,$

$s_5 = Orchid\ Island, s_6 = Orchied\ Island.$

One example for a referential compression of s_2 *against* s_1 *is* $rcs_2 = [(1,0,2,a)(1,4,7,t)]$, *since we have* $s_2 = s_1(0,2) \circ a \circ s_1(4,7) \circ t$. *Similarly, we can referentially compress* s_5 *against* s_1*:*

$$rcs_5 = [(1,0,0,O), (1,0,0,r), (1,0,0,c), (1,2,1,i),$$
$$(1,0,0,d), (1,6,1,I)(1,10,1,l)(1,3,1,n)(1,0,0,d)].$$

However, this referential compression is obviously not a good compression, since the number of RMEs (9) is very close to the number of symbols (13) in the original string s_5. *Intuitively, we would like to compress strings against similar references only, in order to exploit similarities for compression. Furthermore, exploitation of similarities against multiple references often decreases the number of RMEs further. Below is an example, where* s_1 *and* s_5 *are (uncompressed) references,* s_2 *is referentially compressed against* $\{s_1\}$, s_3 *is compressed against* $\{s_1, s_2\}$, s_4 *is compressed against* $\{s_1, s_2\}$, *and* s_6 *is compressed against* $\{s_5\}$*:*

$$comp(s_1, \{s_1\}) = [(1, 0, 18, i)]$$
$$comp(s_2, \{s_1\}) = [(1,0,2,a)(1,4,7,t)]$$
$$comp(s_3, \{s_1, s_2\}) = [(2, 0, 11, /), (1, 13, 5, i)]$$
$$comp(s_4, \{s_1, s_2\}) = [(2,0,2,l), (2,4,7,h)(1,13,5,i)]$$
$$comp(s_5, \{s_5\}) = [(5, 0, 12, d)]$$
$$comp(s_6, \{s_5\}) = [6 : (5,0,5,e), (5,5,7,d)].$$

For convenience of notation, a primary reference is denoted as compressed to itself; an example is $comp(s_1, \{s_1\})$ above. This allows us to simply represent every string as a compressed string.

Since we consider REF to be a set, as opposed to related work, this opens the novel and challenging question of which strings should be considered as references; furthermore, it opens the door to mix references and none-references, i.e., to also use compressed strings as references for other compressed strings. We formalize and study these problems in the following sections.

4. COMPRESSION AND INDEXING

We present a compressed search index using multiple references for indexing dissimilar string collections, called MRCSI. We present the general framework in Section 4.1. In Section 4.2, we show how this framework can be extended to allow efficient approximate search. A cost-model for MRCSI is described in Section 4.3, where we also show that the problem of finding the smallest MRCSI for a given string database under this cost model is NP-hard.

4.1 Compression of Dissimilar Strings

In general, a *multi-reference compressed string database* encodes only few strings from a string database in an uncompressed representation. These uncompressed strings, called primary references, are stored literally. They serve as anchor blueprints for the compressed strings; these are stored as referential compressions using all uncompressed primary references and all other compressed strings as possible references (we have to take care that the resulting structure is cycle-free).

DEFINITION 2 (MRCSD). *For string database* $S = \{s_1, ..., s_n\}$, *a multi-reference compressed string database (MRCSD) is a pair (PREF,COMP), where PREF* $\subseteq$ *S is the set of* primary references *and COMP is a set of referential compressions such that* $\forall rcs_i \in$ *COMP:* $decomp(rcs_i) = s_i$ *and the chain of references encoded in the referential compressions is cycle-free (with the exception of the compressed notation of primary references).*

There are several things to note in this definition. First, the set of primary references *PREF* may, in general, contain more than one element, i.e., multiple strings may serve as anchors, which allows higher compression rates by having more choices to find long matches stored as a single RME. Second, a MRCSD only stores strings in *PREF* in uncompressed form and all other strings from S as referential compressions; as *PREF* usually is small, most strings are stored in compressed format only. Third, strings represented in *COMP* are compressed not only using *PREF* as potential references, but also other strings encoded in *COMP* may serve as references. This helps to further reduce space, as it avoids enlarging *PREF* (which is not compressed) yet still allows to exploit commonalities between essentially all strings in the collection.

To avoid cyclic references, our definition requires that the structure of compressed-against relations of a MRCSD is cycle-free. This cycle-freeness is ensured by building a *reference dependency graph* (which we shall also use to visualize MRCSD). The reference dependency graph of a given MRCSD is a graph with compressed strings and its primary references as nodes, and there exists a path from node n_1 to node n_2, if n_2 occurs in at least one RME of n_1.[1]

DEFINITION 3 (REFERENCE DEPENDENCY GRAPH). *The* reference dependency graph $rdg = (N,E)$ *of* $(PREF, COMP)$ *is a digraph with* $N = PREF \cup COMP$ *and there is a path from x to y, if and only if* $x \in COMP \wedge \exists s,l,m : (y,s,l,m) \in x$.

Three possible reference dependency graphs for the strings in Example 1 are shown in Figure 2.

Clearly, the space required by a MRCSD for a given string database S critically depends on the structure of its reference dependency graph, as this graph determines how and where common substrings can be represented as RMEs (see Figure 1). Unfortunately, the space of possible MRCSDs is exponential in $n = |S|$, as the number of different cycle-free reference dependency graphs, i.e., $\mathcal{O}(2^{(n^2-n)})$, is a lower bound for the number of valid compressed string databases. We get back to the problem of finding an optimal compressed string database in Section 4.3.

4.2 MRCSI: Approximate Search in MRCSDs

So far, we only considered possible ways for compressing a string database using multiple references, but we disregarded our second goal, i.e., to allow efficient k-approximate search. Algorithms using referential compression with a single reference typically perform k-approximate search in two stages [37, 32]: In **Stage 1**, the query string q is searched only in the (single) reference, which, to support this stage efficiently, is stored in indexed form, for instance using suffix trees. Given all matches for q in the reference, matches are propagated to all RMEs subsuming these matches. Of course, any implementation must take care to find all k-approximate matches. In **Stage 2**, matches stretching over more than one RME must be detected. To this end, systems like RCSI build and index a set of additional strings each of which spans over (at least) two adjacent RMEs, thus covering those parts of the compressed strings not present as a consecutive substring in the reference. Combining the results from both stages yields all k-approximate matches of q in the compressed strings.

We apply the same idea to MRCSDs with their more complicated reference structures. First, we find all matches inside RMEs by searching an index over strings in *PREF*. For Stage 2, we extract all substrings from each string from *COMP* not represented within a single RME in a preprocessing step. Additionally, we have to propagate all found matches through the reference dependency graph using breadth-first traversal starting from all primary references (recall that the graph is cycle-free).

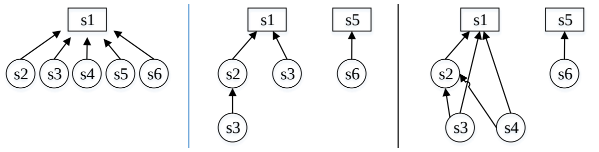


Figure 2: Possible reference dependency graphs for strings in Example 1. In the left graph, $PREF = \{s_1\}$**, in the middle and right graph,** $PREF = \{s_1, s_5\}$**.**

4.2.1 Stage 1: Searching the references

In our implementation, we store all primary references as compressed suffix trees (CSTs). We compute all matches for query q following the seed-and-extend paradigm [1], exploiting the fact that an alignment which allows at most k mismatches must contain at least one exact match (usually called *"seed"*) of a substring of length $\left\lfloor \frac{|q|}{k+1} \right\rfloor$. Second, we propagate matches from references to the compressed strings, by keeping track of all compressed references using a RME propagation map.

DEFINITION 4 (RME PROPAGATION MAP). *Given a referential compression* $rcs = [rme_1, ..., rme_n]$ *against references* $ref = \{s_1, ..., s_m\}$, *a* RME propagation map pm *maps string ids* $\{s_1, ..., s_m\}$ *to interval trees [25], such that interval* $(start, length)$ *is contained in interval tree* $pm(s_i)$*, if and only if there exists a RME* $rme_j = (s_i, start, length, mismatch)$ *in* rcs*. In addition to intervals, we add information of the RME offset to each interval tree entry.*

The definition of RME propagation maps are naturally extended to a set of referential compressions (*COMP* in MRCSDs), by keeping track of string ids in addition to RME offsets. We exploit RME propagation maps to propagate matches from reference strings to compressed strings. Assume that a k-approximate occurrence of q is found in reference s_1, starting at position $match_s$ with length $match_l$. The interval tree of $s1$ allows us to efficiently find all RMEs of compressed sequences which are containing the interval $(match_s, match_l)$. All compressed strings with these RMEs necessarily have a k-approximate occurrence of q. The position of an occurrence depends on 1) the relative start of the match to the start of the interval and 2) on the offset of the RME in the compressed sequence. Both are computed and retrieved with the interval tree, since we explicitly added RME offsets. We show an example for RME propagation maps below, after discussing Stage 2 of our search algorithm.

4.2.2 Stage 2: Searching overlaps

Matches that are not included inside a RME, must overlap at least two RMEs[2]. The position and length of these overlaps depends on the actual query length and error threshold k. Let $\delta = max_{ql} + max_{ed} - 1$, where max_{ql} is the maximum length of a query and max_{ed} is the maximum edit-distance threshold. The value δ restricts the area around mismatch characters (δ characters to the left/right) within which a match for a query can occur. We extract all these *overlap strings* and record their positions in an overlap map.

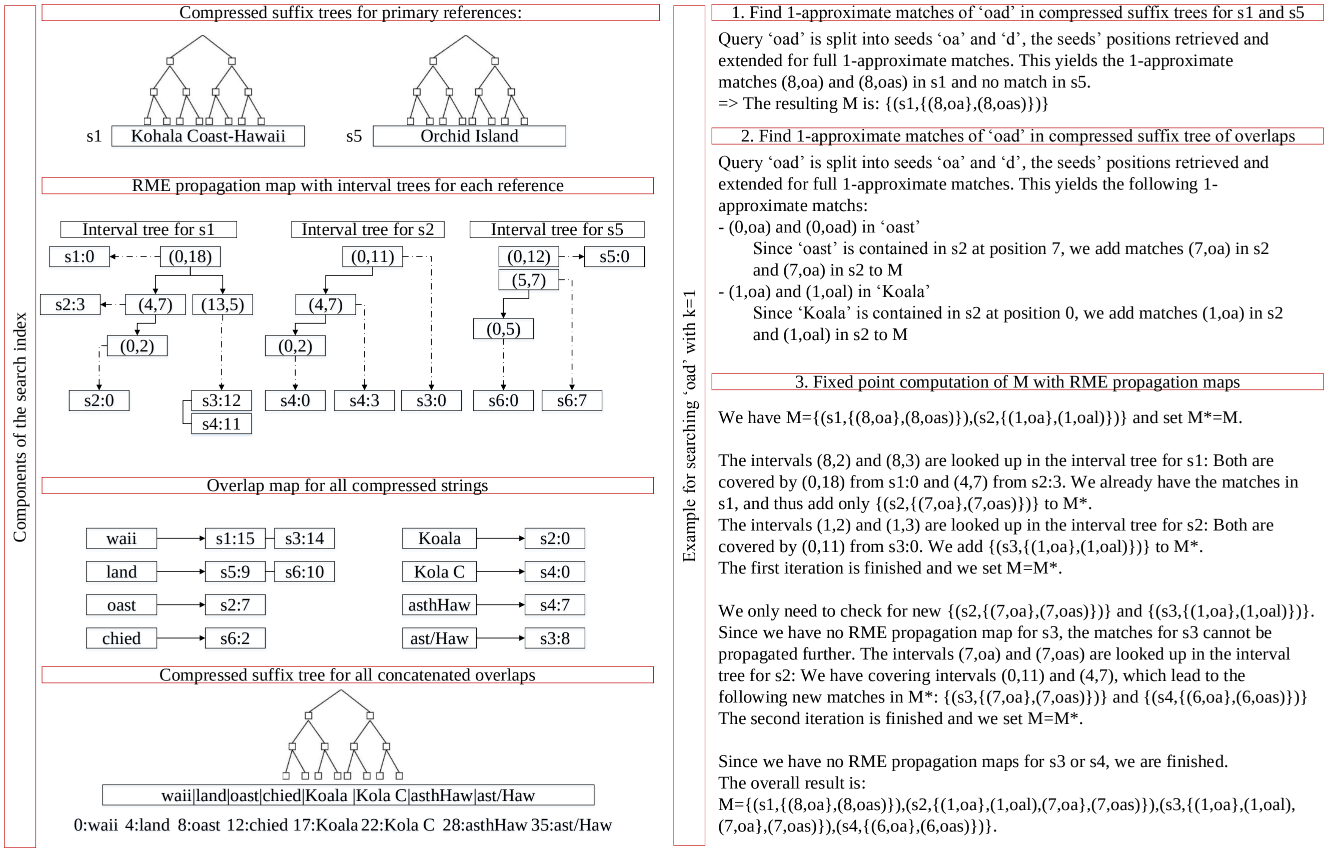
DEFINITION 5 (OVERLAP MAP). *The* overlap string *for a* $rme = (refid, start, length, mismatch)$ *in a referential compression* rcs *of* s *is defined as* $s(offset(rcs, rme) + length - \delta, \delta * 2 + 1)$. *Given a MRCSD* $(PREF, COMP)$*, its* overlap map *is a map from strings to a set of pairs* (rcs, pos)*, defined as* $ovl(o) = \{(rcs, pos) \mid rcs \in COMP \wedge o \text{ is an overlap string of a } rme \in rcs \text{ starting at } pos\}$[3].

With an overlap map we can recover the positions of overlap strings within the compressed strings. However, we still have to define

[1] We ignore the reflexive edges introduced by compressed representations of primary references, see Definition 2.

[2] More precisely, these matches must overlap at least the mismatch character of one RME.

[3] Note that we only store the ID of each rcs in each pair (rcs, pos).


Components of the search index
Compressed suffix trees for primary references:
s1
Kohala Coast-Hawaii
s5
Orchid Island
RME propagation map with interval trees for each reference
Interval tree for s1
s1:0
(0,18)
s2:3
(4,7)
(13,5)
(0,2)
s2:0
s3:12
s4:11
Interval tree for s2
(0,11)
(4,7)
(0,2)
s4:0
s4:3
s3:0
Interval tree for s5
(0,12)
s5:0
(5,7)
(0,5)
s6:0
s6:7
Overlap map for all compressed strings
waii
s1:15
s3:14
land
s5:9
s6:10
oast
s2:7
chied
s6:2
Koala
s2:0
Kola C
s4:0
asthHaw
s4:7
ast/Haw
s3:8
Compressed suffix tree for all concatenated overlaps
waii|land|oast|chied|Koala |Kola C|asthHaw|ast/Haw
0:waii 4:land 8:oast 12:chied 17:Koala 22:Kola C 28:asthHaw 35:ast/Haw
Example for searching 'oad' with k=1

1. Find 1-approximate matches of 'oad' in compressed suffix trees for s1 and s5

Query 'oad' is split into seeds 'oa' and 'd', the seeds' positions retrieved and extended for full 1-approximate matches. This yields the 1-approximate matches (8,oa) and (8,oas) in s1 and no match in s5.
=> The resulting M is: {(s1,{(8,oa},(8,oas)})}

2. Find 1-approximate matches of 'oad' in compressed suffix tree of overlaps

Query 'oad' is split into seeds 'oa' and 'd', the seeds' positions retrieved and extended for full 1-approximate matches. This yields the following 1-approximate matchs:

- (0,oa) and (0,oad) in 'oast'
  Since 'oast' is contained in s2 at position 7, we add matches (7,oa) in s2 and (7,oa) in s2 to M
- (1,oa) and (1,oal) in 'Koala'
  Since 'Koala' is contained in s2 at position 0, we add matches (1,oa) in s2 and (1,oal) in s2 to M

3. Fixed point computation of M with RME propagation maps

We have M={(s1,{(8,oa},(8,oas)}),(s2,{(1,oa},(1,oal)})} and set M*=M.

The intervals (8,2) and (8,3) are looked up in the interval tree for s1: Both are covered by (0,18) from s1:0 and (4,7) from s2:3. We already have the matches in s1, and thus add only {(s2,{(7,oa},(7,oas)})} to M*.
The intervals (1,2) and (1,3) are looked up in the interval tree for s2: Both are covered by (0,11) from s3:0. We add {(s3,{(1,oa},(1,oal)})} to M*.
The first iteration is finished and we set M=M*.

We only need to check for new {(s2,{(7,oa},(7,oas)})} and {(s3,{(1,oa},(1,oal)})}. Since we have no RME propagation map for s3, the matches for s3 cannot be propagated further. The intervals (7,oa) and (7,oas) are looked up in the interval tree for s2: We have covering intervals (0,11) and (4,7), which lead to the following new matches in M*: {(s3,{(7,oa},(7,oas)})} and {(s4,{(6,oa},(6,oas)})}
The second iteration is finished and we set M=M*.

Since we have no RME propagation maps for s3 or s4, we are finished.
The overall result is:
M={(s1,{(8,oa},(8,oas)}),(s2,{(1,oa},(1,oal),(7,oa},(7,oas)}),(s3,{(1,oa},(1,oal), (7,oa},(7,oas)}),(s4,{(6,oa},(6,oas)})}.

Figure 3: Index example for the compressed strings from Example 1. The left part of the figure visualizes the index components and the right parts sketches the 1-approximate search for query 'oad'. We have created the index with $max_{ql} = 3$ and $max_{ed} = 1$. In interval trees, dashed lines denote occurrence annotations.

an efficient technique to find matches in overlaps. Naive traversal over the overlap map at query time is slow, since each overlap string has to be accessed, while often only few overlap strings do contain a match for a seed. Therefore, we create an index for all overlap strings as follows: Given that the overlap map contains strings $o_1, ..., o_n$ as keys, we create a compressed suffix tree over the concatenation $o_{total} = o_1 \circ ... \circ o_n$. The overlap map together with the compressed suffix tree for o_{total} is called *indexed overlap map*. For very short RMEs and large δ, overlaps can overlap each other, since they are extracted from nearby locations in the strings. Indexing a concatenation of these overlaps possibly leads to a waste of space. It is interesting to address this problem in the future by using an index structure tailored towards storing short and possibly similar strings. However, naive approaches like Patricia tries will not work, since these only identical prefixes are exploited for space reduction. Alternatives are the use of directed acyclic word graphs [13] or partition-based approaches [18]. We leave this analysis for future work.

Clearly, using the compressed suffix tree over o_{total}, we can find all occurrences of seeds in each o_i, given a simple data structure which keeps track of the position of each o_i in o_{total}. At search time, we first look up query seeds in the CST of o_{total}, apply the extend algorithm to find all k-approximate matches (still working on o_{total} only), then project all matching positions from o_{total} to the o_i, and finally exploit the overlap map to project the matches from o_i back to the compressed strings, using location augmentations.

4.2.3 Finding all Matches

The above technique for finding k-approximate matches is still incomplete, since matches have to be propagated along paths in the reference dependency graph. The sound and complete algorithm, therefore, is described next: First, all matches contained in primary references are identified, with seed-and-extend and the compressed suffix trees for primary references. Second, all overlap matches are identified, with seed-and-extend and the compressed suffix tree for all concatenated overlaps. At this stage, we have a set of initial matches in *some* strings, which we denote with M. These initial matches in M are propagated from references to referential compressions, by exploiting the interval tree: for each match in M, we access the interval tree of the string containing the match and identify all intervals from compressed strings subsuming this match. From these intervals, we reproduce the original occurrence positions of matches and add them to M. After one iteration, we obtain a new M, which contains the initial matches together with the newly derived matches. The procedure is repeated for the new M, until a fixed point is reached, i.e. M does not give rise to new matches regarding the interval trees.

EXAMPLE 2 (MRCSI). *In Figure 3, we show an example for searching the strings from Example 1 for the query 'oad' with $k = 1$.* We would like to point out that multiple matches in the reference have to be evaluated and propagated separately, since a non-identical interval in the reference could have different subsuming intervals. Thus, even if the same match is found several times in the reference, they have to be propagated one-by-one.

4.2.4 *MRCSI*

The set of indexed primary references together with an indexed RME propagation map and indexed overlap map yields a multi-reference compressed search index (MRCSI).

DEFINITION 6 (MRCSI). *Given string database $S=\{s_1,...,s_n\}$ and a multi-reference compressed string database mrcsd for S, a* multi-reference compressed search index (MRCSI) *for S is a tuple $mrcsi=(IPREF, rmemap, ovlmap)$ where IPREF is a collection of compressed suffix trees for the primary references in mrcsd, rmemap is an indexed RME propagation map for mrcsd and ovlmap is an indexed overlap map for mrcsd.*

A MRCSI index structure contains all necessary for k-approximate search in string collections. We prove soundness and completeness of our search algorithm over MRCSIs.

PROPOSITION 1. *Given a string database $S=\{s_1,...,s_n\}$, every MRCSI of S finds all k-approximate matches (and only these).*

PROOF. k-approximate matches in a compressed string *rcs* have to either 1) be completely covered by a RME or 2) overlap at least two RMEs. For 1), if the match is contained inside a RME, then it must occur in one of the references of *rcs*. If the reference is primary, then them match is found via the compressed suffix trees in *IPREF* and forwarded to *rcs* with the RME propagation map. Otherwise, the reference is not primary (which is discussed below). For 2), the indexed overlap map contains all overlaps of length up to δ over two or more RMEs. Since a k-approximate match can never be longer than δ, the match is inside the overlap map and propagated to the position of the compressed sequence by using the offset information stored in the interval tree.

The remaining case, that the match is completely covered by a RME and reference is not primary, can be shown by induction, since all matches in the non-primary reference are eventually found (induction hypothesis) and the RME propagation map propagates all matches covered by RMEs in *rcs* to their positions (induction step). □

The space requirement of the basic MRCSI index structure is determined by the number N of strings, the maximum length L of strings, and their degree of similarity. In the worst-case, all strings are maximum dissimilar, e.g., their alphabets are disjoint, yet all strings are (inefficiently) compressed against the reference. The space complexity is estimated as follows: There exists exactly one reference string ref, with length at most L, and all other $N-1$ strings are compressed against ref. In the worst case, each character requires its own RME, yielding at most $(N-1)*L$ RMEs. The size of the compressed suffix tree for ref is in $\mathcal{O}(L)$ and the size of the RME propagation map is in $\mathcal{O}(L*N)$, since we have at most $L*(N-1)+1$ RMEs. The size of the overlap map is in $\mathcal{O}(L*N)$ as well, since we create one overlap for each RME, all of which are unique. We have $\mathcal{O}(L*N)$ augmentations of occurrences in the RME propagation map and in the overlap map. In total, we obtain a worst-case space complexity of $\mathcal{O}(N*L)$. The worst-case space complexity is the same as if we create a compressed suffix tree over the concatenation of all strings. We discuss the hardness of creating a space-optimal MRCSI next.

4.3 Optimal MRCSI's

We develop a cost-model for estimating the size (in bits) of a given $mrcsi=(IPREF, rmemap, ovlmap)$ for a string database S. We estimate the overall size by an estimation of the size of each of the three components. In the following we assume that L is the length of the longest string in S. Thus, position values and length values for strings can be stored with $B=\lceil log_2(L)\rceil$ bits. **Primary references:** For each (uncompressed) primary reference, we store a compressed suffix tree. Symbolic regression of index sizes for random strings showed that the size of a compressed suffix tree for a Σ-string s can be estimated with $8*|s|*(1.48+\sqrt{0.01*|\Sigma|})$ bits. Thus, the size of *IPREF* plus their suffix trees is estimated with $COST1=\sum_{s\in IPREF} 8*|s|*(1+1.48+\sqrt{0.01*|\Sigma|})$. Note that we store the original *PREF* in addition to the suffix trees, as extracting substrings from a compressed suffix tree is rather slow [36]. Thus, we perform the verification-phase of the seed-end-extend algorithm for approximate search on the original string and not on the compressed suffix tree. **Indexed RME propagation map:** The RME propagation map assigns interval trees with occurrence augmentations to reference sequences. A single interval entry consists of start and length of the interval, which yields $2*B$ bits. An occurrence augmentation consists of a string identifier and a position, which sums up to $\lceil log_2(|S|)\rceil+B$. The total amount of storage required for a single interval tree is estimated with $2*N*(2*B)+Y*\lceil log_2(|S|)\rceil+B)$, where N is the number of interval entries in the interval tree and Y is the number of occurrence records in the interval tree. Thus, in total we have $COST2=\sum_{z\in Z}(2*N_z*(2*B)+Y_z*\lceil log_2(|S|)\rceil+B))$ bits for storing the RME propagation map, where Z is the set of interval trees, i.e. references in the original MRCSD. **Indexed overlap map:** The overlap map asserts to each unique overlap string a list of occurrences in compressed strings. A single map entry needs $8*(\delta*2+1)$ bits for the overlap string and $N*(\lceil log_2(|S|)\rceil+B)$ bits for the occurrences. In total, the size of the overlap map is estimated with $COST3=X*(8*(\delta*2+1)+Y*(\lceil log_2(|S|)\rceil+B))+ovlcst$, where X is the number of unique overlap strings, Y is the number of occurrence records in the domain of the overlap map, i.e., number of RMEs in the original MRCSD, and *ovlcst* is the estimated length of the CST over all concatenated overlaps.

In total, we have an estimated cost (in bits) for a compressed search index of $COST((IPREF, rmemap, ovlmap))=$ $COST1+COST2+COST3$. Based on this model, we define the problem of finding the smallest MRCSI for a string database S.

DEFINITION 7 (MINIMAL MRCSI). *A multi-reference compressed search index mrcsi is* minimal *for a string database S, if there exists no other $mrcsi_2$ for S with $COST(mrcsi_2)<COST(mrcsi)$. The* MRCSI size optimization problem *is to find a minimal mrcsi for a string database S. The* MRCSI size decision problem *is to decide whether there exists a MRCSI with cost C for S.*

PROPOSITION 2. *MRCSI size decision problem is NP-hard.*

PROOF. First, we show that the problem is in NP. Given a multi-reference compressed search index *mrcsi*, we obviously can compute the cost of *mrcsi* and compare it to C. Now we prove that the problem is NP-complete, by reduction to the Subset-Sum decision problem, which is known to be NP-complete. Given a set of integers $I=\{i_1,...,i_n\}$ and value z, the question of Subset-Sum is whether there exists a subset of $I^*\subseteq I$, with $\sum_{i\in I^*} i=z$. It is easy to see that the decision problem for a simplified COST1-function alone is sufficient to model Subset-Sum. Let $COST2=COST3=0$ and $S=\{s_1,...,s_n\}$, such that each string s_j has length i_j. Furthermore, let $COST1=\sum_{s\in PREF}|s|$. The solution to the MRCSI size decision problem coincides with the Subset-Sum decision problem. Altogether, the MRCSI size optimization problem is NP-hard. □

5. HEURISTICS FOR COMPUTING MRCSI

Given Proposition 2, it is not possible to design a polynomial-time algorithm for solving the MRCSI size optimization problem, unless P=NP. In this section we develop three heuristics that each considers only a certain subset of all possible MRCSD structures, where the number of possible references is increased from heuristic to heuristic. This also leads to gradually increasing compression ratio in practice (see Section 6):

- **Partition (CPart)** in Section 5.1: We restrict MRCSI such that each string is compressed against a single primary reference, while there can be several primary references in total.
- **Compression Forests (CForest)** in Section 5.2: We extend CPart to exploit similarities between non-primary references, by introducing a new indexing technique, called compression forests.
- **Compression DAG (CDAG)** in Section 5.3: Compression forests are generalized to directed acyclic graphs.

Below, we only describe how to compute $PREF$ and $COMP$ for each heuristic. Computation of (indexed) RME propagation map and (indexed) overlap map is straight-forward and the same for all three heuristics.

5.1 Partition (CPart)

We introduce **CPart**, a compression strategy that splits a given string database $S = \{s_1, ..., s_n\}$ into a set of primary references and referentially compressed strings, such that each referentially compressed string is encoded w.r.t. exactly one primary reference. Note that finding a minimal such MRCSI is again NP-hard, since Subset-Sum can be reduced to this problem (proof omitted for brevity).

CPart indexes all strings in the collection S=$\{s_1, ...s_n\}$ iteratively. Starting with an empty MRCSD ($PREF = \emptyset$ and $COMP = \emptyset$), it begins with processing s_1. Since there is no primary reference yet, s_1 cannot be compressed against an existing primary reference. We add s_1 to $PREF$, and obtain $PREF = \{s_1\}$ and $COMP = \{s_1\}$[4]. Next, we process s_2. We have two options: 1) s_2 is added as a new primary reference or 2) s_2 is referentially compressed against s_1. Note that we can, in principle, compute the exact storage cost for both options (see Section 4.3) but this would be slow; how this can be approximated efficiently is described below. We choose the option which minimizes storage: If s_1 and s_2 are rather unrelated, we choose the first option because the referential compression (plus overlaps) would use more space than if we add s_2 as a new primary reference. In this case we get $PREF = \{s_1, s_2\}$ and $COMP = \{s_1, s_2\}$; if s_1 and s_2 are rather similar, we would get $PREF = \{s_1\}$ and $COMP = \{s_1, s_2\}$. When we next add s_3, we have three options: 1) s_3 is added as a new primary reference, or 2) s_3 is referentially compressed against s_1, or 3) s_3 is referentially compressed against s_2. CPart first chooses the primary reference which would allow for the best compression of s_3, i.e., it first chooses among options 2 and 3 and then compares the winner to option 1. The remaining strings are added in the same way. The complete CPart-algorithm is shown in Algorithm 2.

To decide which option is the least space-intensive, Algorithm 2 needs to compute the number of RME's necessary for compressing one string w.r.t. another. We call this property *compressibility*. This quantity can be computed quickly if the two strings are similar using a technique called local matching optimization [36]. In experiments, we achieved main-memory compression speed of more than 500 MB/s. However, if the to-be-compressed string and the reference have only few similarities, the compression speed degrades recognizably (down to 10-50 KB/s in tests). Algorithm 2 very often has to compute compressibility between probably dissimilar strings. To cope with this issue, CPart only estimates compressibility using N parts of the to-be-compressed string s: s is split into N blocks and the longest prefix of each block is looked up in the existing references. The lengths of the matches are averaged and exploited as an estimation of the average length of the RMEs for the whole strings. In our experiments, we fixed N to $log_2(|s|)$.

Algorithm 2 Computation of MRCSI-CPart

```
Input: String database $S = \{s_1, ..., s_n\}$
Output: MRCSI $(IPREF, rmemap, ovlmap)$
Let $PREF = COMP = rmemap = ovlmap = \emptyset$
for $1 \le i \le n$ do
    Let $cost_1 = \infty$
    if $|PREF| > 0$ then
        Find $pref \in PREF$ such that $|comp(s_i)|$ is approximately minimal ($s_i$ is compressed with respect to pref)
        Let $rcs = comp(s_i)$ ($s_i$ compressed against $pref$)
        Let $cost_1$ be the cost if $rcs$ is added to MRCSI
    end if
    Let $cost_2$ be the cost if $s_i$ is added as a new primary reference
    if $cost_2 > cost_1$ then
        Let $COMP = COMP \cup \{rcs\}$
    else
        Let $PREF = PREF \cup \{s_i\}$
        Let $COMP = COMP \cup \{[(s_i, 0, |s_i| - 1, s_i[|s_i| - 1])]\}$
    end if
end for
Compute $IPREF$ from $PREF$
Compute $rmemap$ from (PREF,COMP)
Compute $ovlmap$ from (PREF,COMP)
Return $(IPREF, rmemap, ovlmap)$
```

This was sufficient to find any reference with sufficient degree of similarity.

Obviously, CPart essentially performs a compressibility-based clustering of the strings in the collection. Therefore, existing string clustering techniques might be applicable as well. However, the major challenge in clustering is to define a similarity criterion suitable for the particular problem at hand. A popular choice when clustering strings is the edit distance, but computing it is slow ($\mathcal{O}(|s_1| * |s_2|)$), and it is not always positively correlated to compressibility. For instance, the strings $s_1 = a^n b^n$ and $s_2 = b^n a^n$ have a large edit distance ($2 * n$), but still a high compressibility, i.e. s_2 can be encoded with two RMEs into s_1. Still, it remains an interesting topic for future work to experiment with other, more efficient similarity measures, such as the Jaccard coefficient over q-grams.

5.2 CForest

CPart overcomes the shortcoming of methods like RCSI to use only a single reference during compression. However, each string still is only compressed against a single primary reference. Similarities between compressed strings are not exploited, which neglects ample compression potential.

EXAMPLE 3. *We have a set of strings $S = \{s_1, ..., s3\}$ with*

s_1 =*Kohala Coast-Hawaii*, s_2 = *Kola CoasthHawaii*,

s_3 =*Kola CoasthHawii*.

We obtain the following referential compression:

$$comp(s_2, \{s_1\}) = [(1,0,2,l), (1,5,7,h), (1,13,5,i)]$$

$$comp(s_3, \{s_1\}) = [(1,0,2,l), (1,5,7,h), (1,13,3,i)(1,0,0,i)]$$

If s_2 is used as additional reference for s_3, then we obtain the shorter compression for s_3: $[(2,11,2,l)$, $(1,13,3,i)(1,0,0,i)]$, by replacing the identical RME sublist $[(1,0,2,l), (1,5,7,h)]$ with a reference into s_2.

We capture such ideas in so-called *second-order compression*. In contrast to CPart, second-order compression also considers references into referential compressions instead of only uncompressed strings. This is achieved by greedily rewriting an existing referential compression (against a primary reference) into one against other referential compressions, by replacing consecutive RMEs occurring in the to-be-rewritten compressed string and the compressed reference strings by one *second-order* RME into the compressed reference string.

Indexing RMEs: Given a collection of referential compressions $R = \{rcs_1, ..., rcs_n\}$, as obtained by CPart, we create a hashmap

[4] String s_1 is a primary reference, yet denoted with one RME as a compressed string.

Algorithm 3 Rewriting a referential compression against another referential compression

```
Input: To-be-rewritten referential compression $rcs_j$, reference referential compression $rcs_{ref}$
Output: Referential compression $result$
Let $result$ be an empty list of referential match entries
while $|s| \neq 0$ do
    Let $pre$ be the longest prefix of $rcs_j$, that is an infix of $rcs_{ref}$
    if $|pre| \leq 1$ then
        $result = result \circ rcs_j(0)$
        Remove $rcs_j(0)$ from $rcs_j$
    else
        Add $(i, offset(rcs_{ref}, pre(0)), (\sum_{rme \in pre} |rme|) - 1, c)$ to the end of $result$, where $c$ is the mismatch character of the last RME in $pre$
        Remove the prefix $pre$ from $rcs_j$
    end if
end while
Return $result$
```

Algorithm 4 Computation of MRCSI-CForest

```
Input: String database $S = \{s_1, ..., s_n\}$
Output: MRCSI $(IPREF, rmemap, ovlmap)$
 1: Let $IPREF$ be the set of primary references of CPart and $COMP$ the referential compression
 2: Let $NEWCOMP = \emptyset$
 3: Let $rdg$ be an empty reference dependency graph
 4: for $0 \leq i \leq |COMP - 1|$ do
 5:     Select (by sampling) the most similar referential compression to $COMP[i]$ from $COMP[0, i-1]$, let the result be $rcs$
 6:     Let $refcands$ be the set containing $rcs$ and all its direct/indirect parents in $rdg$
 7:     Rewrite $COMP[i]$ against $refcands$ and let the result be $ncomp$
 8:     Append $ncomp$ to $NEWCOMP$
 9:     Add $ncomp$ as a child of $rcs$ in $rdg$
10: end for
11: Compute $newrmemap$ from $(PREF, NEWCOMP)$
12: Compute $newovlmap$ from $(PREF, NEWCOMP)$
13: Return $(IPREF, newrmemap, newovlmap)$
```

from RMEs to offsets of RMEs in the referential compressions. For each RME in R we store the identifier of the referential compression and offsets where the RME occurs. Second-order compression is implemented, on top of this RME hashmap, with an algorithm very similar to the referential compression algorithm for computing RMEs: We simply replace the alphabet Σ with the set of all RMEs. This idea is implemented in Algorithm 3 for a single compressed reference string. The to-be-rewritten referential compression rcs_j is traversed from left to right. The longest prefix of rcs_j (i.e., a sequence of RMEs) that can be found in rcs_{ref} is replaced by one RME to rcs_{ref}. If no such prefix of length greater 1 is found, we simply copy the first RME from rcs_j to the result and continue searching with the next one. The compression algorithm terminates once all RMEs have been processed.

Given an algorithm for rewriting a referential compression against a set of referential compression (the extension is straight-forward), we develop a second heuristic for selecting rewrite candidates. In a first phase, the CPart algorithm is used to select primary references. In a second phase, we greedily try to rewrite each referentially compressed string against one or more of the other referentially compressed strings. If the new compression yields a smaller index structure, the rewriting is performed. To keep the effort of selecting rewritings low, the number of second-order references is restricted per case.

The reference dependency graph for CForest is a forest with the primary references as roots. Given the result of CPart (which is a special case of CForest with trees of height one), we process all compressed strings one-by-one. For each compressed string, CForest selects the most similar referential compression seen before based on sampling: it selects $\log_2(n)$ equally distributed RMEs from a referential compression with n RMEs. For each sample RME, the set of compressed strings containing the RME is computed. The compressed string rcs which contains the highest number of sampled RMEs is identified. The compressed string rcs plus all its parents in the reference dependency graph are set as second-order references for rewriting, following the procedure described above. After compression, the second-order compressed string is added as a child of rcs, thus, preserving a forest structure. The Algorithm is summarized in Algorithm 4. Note that we do not compress the sequences twice against primary references. Although CForest builds on the selection of primary references of CPart (based on compressibility estimation only), the actual compression takes place in Algorithm 4.

Overall, CForest tries to rewrite each compressed string only against compressed strings on the same path to the primary reference. Thus, CForest is a compromise: On the down side, it does not consider all possible compressions of sequences of RMEs in the entire MRCSD; but by doing so, it only has to consider few references for rewriting.

5.3 CDAG

We implemented a third heuristic, termed Compression DAG (or CDAG). It keeps the general approach of CForest, but now each referential compression is compressed against all previously compressed strings with the same primary reference (induced by CPart) instead of only the best one and its parents. We only perform a single pass over the strings in each partition, which keeps the dependency graph acyclic; the resulting structure is a DAG. The algorithm is very similar to Algorithm 4. The only differences are: $refcands = NEWCOMP$ replaces Lines 5–6 in Algorithm 4 and the reference dependency graph has to be updated accordingly (Line 9), by adding $COMP[i]$ as a child to all $refcands$. Note that CDAG still is heuristic, since is explores only a tiny fraction of the entire search space of all MRCSIs for a given string database; in particular, it is greedy in selecting primary references, and inherits the fairly simple method from CPart which determines the actual number of primary references.

6. EXPERIMENTAL EVALUATION

We experiment with different data sets measuring indexing time, space requirements, and query performance. All experiments were run a server with 1 TB RAM and 4 Intel Xeon E7-4870 (in total, hyperthreading enables the use of 80 virtual cores). However, all experiments were run in a single thread only. Code was implemented in C++, using the BOOST library, CST [28], and SeqAn [9]. Failed experiments are indicated with 'NA'. Our code for MRCSI can be downloaded[5] for free academic use.

6.1 Competitors and Datasets

As baseline comparison, we created a compressed suffix tree [28] (ConcCST) and an enhanced suffix array [9] (ConcESA) for the concatenation of all documents in an evaluation dataset. Approximate search on these structures was implemented using the same seed-and-extend algorithm as for MRCSI. We expected much worse compression rates for ConcCST/ConcESA (as similarity is not really exploited) and better query times (as matches are found directly without any need for propagation or decompression).

Second, we compare against an indexing technique [23] developed in the Bioinformatics community: Given a maximum query length

[5] http://www.informatik.hu-berlin.de/~wandelt/MRCSI

ID	Description	|Σ|	Count	Avg length	Total size (MB)
COMP	History of www.computer.org	97	510	109,750	56.0
GWB	Wikipedia page for George W. Bush	96	45,415	312,473	14,191.9
HG21	Human Chromosome 21	5	1,000	51,221,669	51,221.7
HEL	Wikipedia page for Helsinki	96	2,664	216,730	577.4
MOZ	History of www.mozilla.org	98	3,333	21,200	70.7

Table 1: Datasets for evaluation.

	HEL	Index size (MB)				Indexing time (s)			
	|Strings|	40	160	640	2560	40	160	640	2560
Compression only	RLZ.025	2.9	5.3	17.4	48.0	**4.8**	**8.5**	**35.4**	**120.4**
	RLZ.05	3.1	8.9	30.8	84.7	6.8	20.1	86.0	259.7
	RLZ.1	4.5	16.6	58.3	160.2	8.2	39.7	157.4	469.3
	Tong.025	7.3	**2.5**	**6.3**	**18.0**	15.8	14.5	53.6	172.2
	Tong.05	1.9	2.5	7.1	21.0	7.6	18.7	81.6	285.4
	Tong.1	**1.4**	2.9	9.4	26.1	9.4	39.7	175.6	537.1
Index-based	ConcCST	38.7	151.1	533.9	1,473.5	36.4	148.2	544.3	1,521.8
	ConcESA	443.2	1,722.6	6,077.8	16,642.7	49.3	217.5	733.7	1,990.6
	USConcCST	18.1	23.6	43.3	119.8	41.7	130.6	547.2	1,463.0
	USConcESA	169.1	221.5	406.4	1,121.5	43.2	122.7	460.9	1,711.9
	iRLZ.025	6.2	11.5	37.9	107.8	7.2	12.5	53.2	150.9
	iRLZ.05	6.3	18.6	63.9	180.3	6.8	18.4	88.4	269.6
	iRLZ.1	9.3	33.9	118.6	330.3	8.2	37.5	161.7	480.4
	iTong.025	14.6	6.4	16.2	51.4	21.3	17.0	62.2	198.8
	iTong.05	4.4	6.2	17.7	56.5	10.8	25.6	105.2	331.1
	iTong.1	3.4	6.8	21.8	65.5	10.3	41.9	183.8	582.5
	RCSI	2.7	5.3	21.7	115.8	**2.2**	5.1	27.0	165.0
	CPart	2.7	5.3	21.7	115.8	**2.2**	4.6	15.9	70.9
	CForest	2.7	4.4	11.3	44.1	2.3	4.8	17.9	121.9
	CDAG	**2.6**	**4.1**	**9.5**	**31.7**	**2.2**	**4.5**	**15.0**	**67.8**

Table 2: Index size and indexing time for HEL.

	GWB	Index size (MB)				Indexing time (s)			
	|Strings|	80	640	5120	40960	80	640	5120	40960
Compression only	RLZ.025	8.9	40.1	246.2	957.9	**19.7**	**127.0**	**1,038.1**	**5,354.9**
	RLZ.05	10.1	67.7	444.0	1,702.4	23.0	214.1	1,702.1	7,928.2
	RLZ.1	18.0	127.6	837.2	3,215.0	41.7	386.2	3,036.0	13,563.4
	Tong.025	7.2	**18.0**	**110.4**	**346.5**	34.7	194.3	1,550.5	7,415.3
	Tong.05	**4.7**	22.6	127.0	NA	32.2	289.1	2,419.6	NA
	Tong.1	5.2	29.9	152.4	491.2	54.7	488.0	4,108.1	19,323.5
Index-based	ConcCST	172.5	1,242.5	NA	NA	159.4	1,258.9	NA	NA
	ConcESA	1,921.1	13,891.8	NA	NA	200.6	1,564.5	NA	NA
	USConcCST	46.9	85.1	NA	NA	181.4	1,414.8	NA	NA
	USConcESA	436.4	796.5	NA	NA	203.3	1,437.3	NA	NA
	iRLZ.025	17.5	80.8	489.9	1,965.9	24.9	141.8	1,109.5	5,790.7
	iRLZ.05	21.6	137.7	877.5	3,460.2	24.9	227.2	1,741.7	8,423.0
	iRLZ.1	37.4	257.5	1,665.6	6,480.3	45.7	398.2	3,186.8	14,689.7
	iTong.025	14.2	39.2	208.5	768.7	38.8	195.9	1,733.1	8,168.2
	iTong.05	10.9	48.9	247.9	875.8	37.8	307.2	2,550.5	12,073.3
	iTong.1	12.2	60.1	302.3	1,034.0	57.4	512.6	4,315.6	20,626.1
	RCSI	10.5	46.1	530.0	4,421.5	9.7	71.5	1,031.3	12,049.2
	CPart	11.4	46.1	427.2	2,818.6	**9.3**	37.0	**437.8**	**4,132.1**
	CForest	10.3	22.5	122.7	778.8	9.6	39.1	887.6	15,778.0
	CDAG	**10.2**	**19.9**	**80.6**	**390.2**	9.5	**35.3**	441.7	4,307.1

Table 3: Index size and indexing time for GWB.

max_{ql} and an upper bound for the error rate max_{ed}, only unique substrings are recorded[6] and an index over the concatenation is created (USConcCST for compressed suffix tree as index, USConcESA for enhanced suffix array). This technique was proposed more as a proof of concept for human genomes, but has never be tested on datasets outside the Bioinformatics community.

Third, we compare against RLZ [17], which specifically addresses referential compression of dissimilar data sets. RLZ only focuses on space and does not per-se support searching the compressed data; for comparison, we implemented a search procedure on top of the RLZ archives which dynamically decompresses strings before searching them. RLZ has a parameter, called coverage value, which has to be set manually before compressing data. We found the impact of this parameter to be quite strong and therefore report results with different values (0.1, 0.05, and 0.025, as proposed in [17]). In addition we compare to a very recently proposed extension to RLZ: Tong [34]. By analyzing the dictionary and eliminating rarely used parts, the authors reported significant improvements of Tong over the results of RLZ. We expect RLZ and Tong to excel in compression ratio but to be (much) worse in terms of search speed. For both competitors, RLZ and Tong, the coverage value is appended to the name, e.g. RLZ.025 refers to RLZ with a coverage of 0.025 (dictionary sampling at a rate of 2.5%).

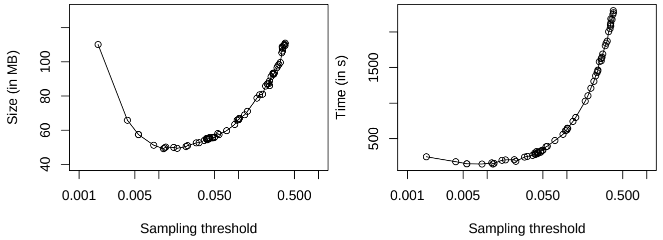


Figure 4: Impact of iTong's sampling threshold on index size (left) and indexing time (right) for HEL with 2,560 strings.

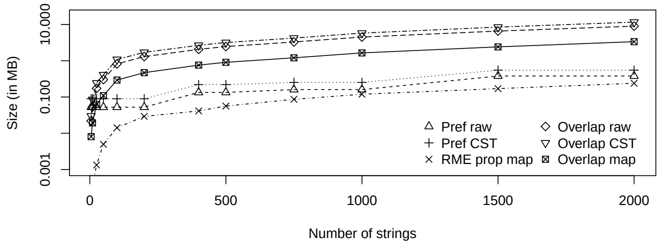


Figure 5: Size of index components for CDAG and HEL.

In addition, we extended the methodology behind RLZ/Tong and implemented a search *index* on top of their compression techniques, following the same idea as in MRCSI (as described in Section 4.2): the sampled dictionary is indexed as a reference and compressed strings are managed in an indexed RME propagation map and an indexed overlap map. These competitors are denoted with iRLZ/iTong and coverage value appended as above. We expect these competitors to be the most challenging. These competitors, however, have not been proposed in the literature and thus are considered as a virtual baseline only.

Finally, we compared against our own prior work RSCI [37], which uses the same search procedure as MRCSI but can only compress against a single reference. For homogeneous data sets, we expect similar or even better performance than MRCSI, but much worse results when collections are dissimilar. There are also some other somewhat related methods against which we do not compare to experimentally; these are described in Section 6.4. Overall, we have 20 competitors/setups, grouped as follows:

1. *Compression only*: Related-work techniques that only compress documents and have to use index-less search (RLZ.025, RLZ.05, RLZ.1, Tong.025, Tong.05, Tong.1).
2. *Index-based*: Related-work techniques with a search index (ConcCST, ConcESA, USConcCST, USConcESA, iRLZ.025, iRLZ.05, iRLZ.1, iTong.025, iTong.05, iTong.1, RCSI) and our multi-reference compressed search index (CPart, CForest, CDAG).

We evaluated a total of five datasets. These datasets are described in Table 1. To measure on semi-structured documents, we use the history of two web pages downloaded from Wayback Machine: COMP and MOZ. Second, we downloaded the complete history of versions for Wikipedia page Helsinki (HEL) and George W. Bush (GWB). All these datasets contain strings of varying similarity over a mid-sized alphabet. For instance, while the first ever recorded Wikipedia article in HEL is very different to today's article, consecutive versions of Wikipedia articles often only have minor modifications. Our biological dataset, HG21, consists of very long, highly-similar strings over a small alphabet.

For each dataset we generated a set of 5,000 queries, by taking random substrings of the input (for HG21 of length 80-100, for the other datasets of length 12-18). k-approximate searching was performed using values $k \in \{0,\ldots,5\}$.

[6] The original paper's supplementary file [23] also proposes to group similar non-unique strings, but does not state how to select/represent these similar strings.

	MOZ	Index size (MB)				Indexing time (s)			
	|Strings|	80	320	1280	3332	80	320	1280	3332
Compression only	RLZ.025	0.3	1.1	3.3	6.4	**0.5**	**1.9**	**6.1**	**12.4**
	RLZ.05	0.4	1.6	5.1	9.6	0.7	2.5	10.1	21.5
	RLZ.1	0.7	2.6	9.3	16.7	1.0	3.9	16.7	34.8
	Tong.025	**0.2**	0.9	1.9	**4.2**	0.8	2.9	8.7	19.7
	Tong.05	**0.2**	0.8	**1.8**	4.4	1.0	3.8	12.8	27.9
	Tong.1	**0.2**	**0.7**	2.4	4.9	1.5	5.6	21.3	44.6
Index-based	ConcCST	6.6	24.1	86.5	152.5	5.3	20.6	76.7	139.4
	ConcESA	73.5	267.1	954.9	1,683.8	3.1	15.9	83.8	170.0
	USConcCST	2.1	5.5	12.0	35.7	5.0	19.3	64.3	119.3
	USConcESA	19.6	51.7	112.4	331.9	4.0	13.8	54.4	112.9
	iRLZ.025	0.7	2.7	7.2	17.3	0.7	2.6	7.8	19.0
	iRLZ.05	0.9	3.4	10.9	23.2	0.8	2.8	10.1	23.7
	iRLZ.1	1.6	5.5	19.1	36.9	1.3	4.6	18.3	40.0
	iTong.025	0.6	2.3	4.9	14.2	1.1	4.1	11.3	27.4
	iTong.05	0.5	2.2	4.7	13.7	1.3	4.8	14.5	33.8
	iTong.1	0.5	1.7	5.5	14.3	1.8	6.1	23.9	51.9
	RCSI	0.5	4.8	38.8	79.0	0.6	4.3	40.2	79.6
	CPart	0.5	3.5	8.6	25.7	**0.5**	2.1	5.5	19.2
	CForest	**0.4**	1.5	3.6	12.9	**0.5**	4.4	13.2	32.9
	CDAG	**0.4**	**1.2**	**2.7**	**9.9**	**0.5**	**1.8**	**4.7**	**15.6**

Table 4: Index size and indexing time for MOZ.

6.2 Index generation

First, we analyze the results for Wikipedia datasets. Table 2 and Table 3 show the index size and indexing time for HEL and GWB, respectively. Tong always computes the smallest compression. The optimal coverage value depends on number of strings in the dataset: the more (similar) strings are to be compressed, the smaller coverage values become efficient. CDAG is always the smallest index-based competitor: up to 10 times smaller than RCSI and approx. half as small as best variant of iTong. We have further analyzed the impact of coverage value on index size (and indexing time) of iTong to see whether other coverage values lead to a smaller index. The results are shown in Figure 4. It can be seen that the coverage value 0.025 (index size: 51.4 MB) is relatively close to the optimal coverage value 0.0114 (index size: 50.3 MB).

The results for GWB provide interesting insights in terms of heterogeneity: Over time, this Wikipedia page has undergone a huge number of (major) revisions, where a single revision often changes the page completely (especially in 2003, during the US's war against Iraq). It can be seen that additional references can have a huge positive impact on the index size. RCSI, which only uses one reference, needs 4,421.5 MB, while CPart can already reduce the search index down to 2,818.6 MB. Further exploitation of similarities can reduce the index down to 390.2 MB using CDAG.

We show the distribution of index components for CDAG and HEL in Figure 5. The results demonstrate how the number of primary references (size Pref raw and Pref CST) is steadily increased with a growing number of strings in the collection. Remarkably, the storage for overlaps (raw + CST) consumes the largest part of the index. Thus, further analysis regarding less redundant storage of overlaps could significantly reduce the size of the search index. We have further analyzed the number of RMEs for competitors for HEL with 2,560 strings: CDAG has the smallest number of RMEs (sum over all compressed strings) of all index-based competitors (298,176). The next best competitors have already almost one order of magnitude more RMEs: iRLZ.025 (1,294,190), CForest (1,337,072), and iTong.025 (1,767,274). RCSI has 10,590,470 RMEs. In general, fewer RMEs, yield a smaller index, but the size of the reference index has to be taken into account as well, e.g., iTong.025 produces more RMEs than iRLZ.025, but the final index is 50% smaller.

CDAG/CPart are often the fastest competitors, even outperforming compression-only competitors. The rationale is as follows: since RLZ/Tong create a dictionary from sampling, the maximum match length is restricted by the sample block size (1,000 in our experiments). During compression, following at most 1,000 matching symbols, it is always necessary to perform a (slow) lookup in the index of the primary references to find the next longest prefix. Our MRCSI techniques, in contrast, index whole documents, which allows for much longer matches and often much fewer lookups in the index of the primary references. The time saving is often so big, that the additional overhead for second-order indexing, e.g. in CDAG, is compensated. The fact that CDAG often (slightly) outperforms CPart is counter intuitive, since CDAG is based on the primary references of CPart and initially performs the same compression. Our analysis showed that much time in CPart is spent on creating the RME propagation map and overlap map. Because CDAG often has fewer RMEs than CPart, since similarities to other compressed strings are exploited, this phase is often much faster in CDAG, since less RMEs have to be managed.

	COMP	Index size (MB)				Indexing time (s)			
	|Strings|	10	40	160	509	10	40	160	509
Compression only	RLZ.025	2.1	1.7	4.0	7.2	2.3	2.7	**6.7**	**14.5**
	RLZ.05	**0.7**	1.1	3.7	8.9	**1.1**	**1.8**	**6.7**	18.9
	RLZ.1	**0.7**	1.6	6.0	13.9	**1.1**	2.2	10.7	28.6
	Tong.025	1.7	3.8	3.4	6.2	4.1	8.1	11.5	24.5
	Tong.05	1.1	0.8	1.9	5.7	2.6	2.9	9.8	28.5
	Tong.1	1.0	**0.6**	**1.7**	**5.2**	2.6	3.3	14.8	41.1
Index-based	ConcCST	3.5	13.6	51.4	120.1	2.9	12.2	49.6	118.6
	ConcESA	38.7	151.8	565.3	1,395.3	1.7	11.1	62.1	128.8
	USConcCST	8.5	10.0	28.3	60.4	8.2	16.4	58.0	144.2
	USConcESA	78.3	92.9	261.2	584.3	7.1	15.9	58.2	118.4
	iRLZ.025	6.6	3.7	10.8	20.3	5.7	4.0	12.8	23.4
	iRLZ.05	2.3	2.6	10.3	23.1	2.0	2.3	9.4	22.8
	iRLZ.1	2.0	3.4	14.3	32.3	1.7	2.7	13.5	36.0
	iTong.025	6.7	9.0	11.1	21.7	7.9	12.1	18.9	37.9
	iTong.05	4.4	2.0	8.2	18.5	4.9	3.6	14.5	39.0
	iTong.1	3.9	1.6	7.1	16.7	4.7	4.0	18.8	50.4
	RCSI	**0.9**	1.5	15.2	64.3	**0.7**	**1.2**	14.9	73.1
	CPart	**0.9**	1.5	7.6	23.3	**0.7**	**1.2**	6.0	20.0
	CForest	**0.9**	**1.4**	6.1	15.5	**0.7**	**1.2**	6.0	19.9
	CDAG	**0.9**	**1.4**	**6.0**	**14.4**	**0.7**	**1.2**	**5.9**	**18.0**

Table 5: Index size and indexing time for COMP.

	HG21	Index size (MB)				Indexing time (s)			
	|Strings|	10	40	160	640	10	40	160	640
Compression only	RLZ.025	175.6	380.9	561.1	**2,039.4**	**218.5**	**853.9**	**5,125.8**	**67,943.7**
	RLZ.05	**161.3**	332.2	956.9	NA	261.5	965.5	6,607.7	NA
	RLZ.1	178.9	460.8	1,799.8	NA	379.7	1,496.5	10,311.6	NA
	Tong.025	185.5	738.0	225.3	NA	469.1	1,857.9	6,808.7	NA
	Tong.05	183.5	204.4	**223.8**	NA	655.8	2,136.4	10,246.1	NA
	Tong.1	204.0	**131.7**	294.8	NA	1,307.5	4,336.4	22,574.2	NA
Index-based	ConcCST	1,139.6	NA	NA	NA	1,378.7	NA	NA	NA
	ConcESA	12,028.2	NA	NA	NA	1,126.7	NA	NA	NA
	USConcCST	11,729.0	NA	NA	NA	18,555.9	NA	NA	NA
	USConcESA	NA	NA	NA	NA	NA	NA	NA	NA
	iRLZ.025	1,616.4	1,287.9	1,154.5	4,101.7	1,828.9	1,901.6	5,678.3	68,463.0
	iRLZ.05	1,233.4	911.3	1,965.0	NA	1,391.0	1,332.3	6,869.9	NA
	iRLZ.1	958.7	1,008.9	3,691.0	NA	1,080.3	1,630.9	10,594.1	NA
	iTong.025	2,130.8	2,718.8	512.9	NA	2,544.7	4,511.2	7,309.1	NA
	iTong.05	2,038.7	698.8	516.0	NA	2,745.3	2,792.4	10,747.0	NA
	iTong.1	1,906.1	362.5	664.5	NA	3,218.1	4,526.1	23,058.4	NA
	RCSI	277.5	314.7	380.6	687.0	432.3	**499.4**	**693.6**	**1,562.4**
	CPart	277.5	314.7	380.6	687.0	**416.8**	507.8	806.1	1,671.8
	CForest	276.4	309.4	357.0	581.9	435.4	502.8	764.1	1,894.8
	CDAG	**275.9**	**305.6**	**341.5**	**512.9**	433.6	509.0	745.0	1,745.0

Table 6: Index size and indexing time for HG21.

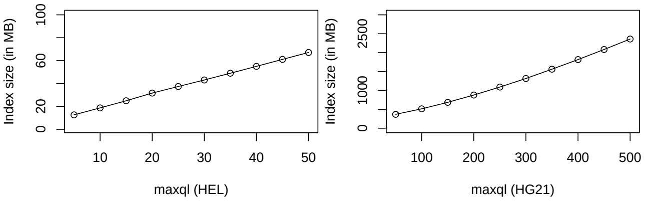


Figure 6: Index size against variable max_{ql} for HEL with 2,560 strings (left) and HG21 with 640 strings (right).

We analyze the results for our webpage datasets next. Table 4 and Table 5 show the index size and indexing time for MOZ and COMP, respectively. Overall, the results confirm our analyses of Wikipedia datasets before. CDAG creates the smallest index-based representation, while Tong creates the smallest compression-only representation, where the best coverage value depends on the number strings in the dataset. For COMP, however, the largest coverage value of 0.1 produces the smallest representation.

Table 6 shows index size (in MB) and indexing time (in s) for up to 640 human chromosome 21. Many competitors fail to create an index within a single day. For 640 strings of HG21, ConcESA and ConcCST would need to create a compressed suffix tree/enhanced suffix array over a string of approx. 35 GB length. Even the other referential competitors (RLZ/Tong) cannot compute an index for several reasons: if the coverage value is too big, then the initial in-

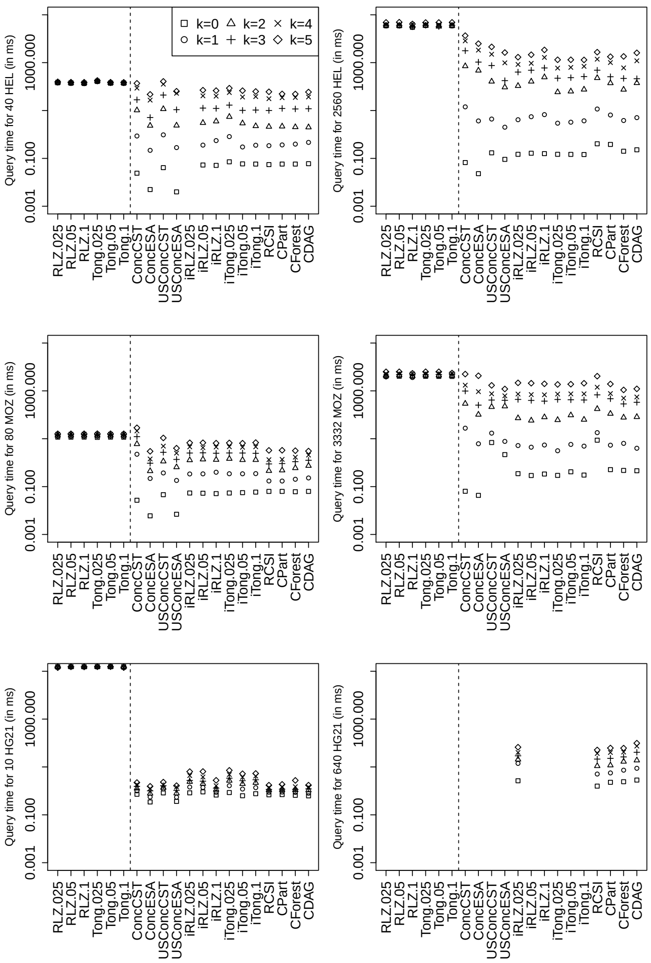


Figure 7: Median query answering times for HEL (top), MOZ (middle), and HG21 (bottom).

dex is large (and takes a lot of time to create) and if the coverage value is too small, then the compression takes a long time, since only short matches can be found in the dictionary. CDAG computes the smallest MRCSI index and also the smallest overall index (25% smaller than RCSI, eight times smaller than iRLZ.025). It is interesting to note, that the compressed search index of CDAG is even four times smaller than the compression-only representation of RLZ.025. This demonstrates that the selection of a dictionary sample is in fact a very difficult problem, and heuristics depend on the number of strings and their alphabets. The shortest indexing time is achieved by RCSI, which was developed exactly for compression of chromosomes from the same species.

In Figure 6, we show the index size for a variable maximum query length for two selected datasets: HEL (up to $max_{ql} = 50$) and HG21 (up to $max_{ql} = 500$). The index size grows approx. linearly with the maximum query length, since the overlap map contains longer overlaps. The super-linear increase for HG21 is explained as follows: With an increasing max_{ql}, some overlaps which were previously identical, become distinct, since they start to overlap more RMEs than before. These now-distinct overlaps have to be indexed separately.

6.3 Query answering

We evaluated a set of 5,000 queries for datasets HEL, MOZ, and HG21. Median query answering times for $k \in \{0,...,5\}$ and a variable number of strings are shown in Figure 7. Compression-only competitors have a orders of magnitudes higher median query answering times compared to index-based competitors, since strings are decompressed and searched at query time. The fastest competitor for small k is usually ConcESA, which stores an enhanced suffix array over the concatenation of all strings. For all index-based competitors working with referential compression, there is no clear trend for a fastest competitor. There is a trade-off between number of RMEs and maximum path length in reference dependency graphs. On one hand, if a string is compressed with a lot of RMEs, many overlaps have to be generated and thus increase index size and overhead for managing overlaps. If fewer RMEs are needed, however, the number of overlaps is reduced at the cost of longer time spent on propagation of results through the reference dependency graph. In Figure 8, we analyze median query answering times for HG21 and k up to 10, which yields an error rate of approx. 10%. As expected, with increasing k, median query answering times significantly increase, since 1) more results are returned and 2) much more false positives have to be verified by the seed-and-extend algorithm. For other datasets, error rates of more than 25% have been already covered by the experiments in Figure 7, since queries are much shorter (only 12-18 characters).

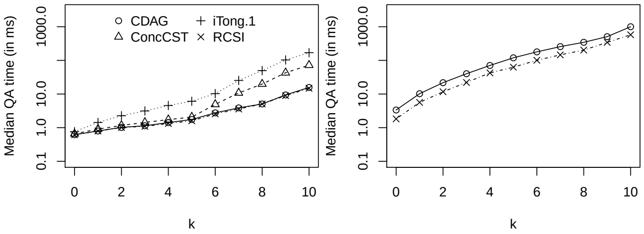


Figure 8: Median query answering times for large k and HG21: 10 strings (left) and 640 strings (right).

Competitor	RCSI	iRLZ*	iTong*	RLZ*	Tong*	CPart	CForest	CDAG
COMP	1:1	3:1	3:1	8:1	11:1	2:1	4:1	4:1
GWB	3:1	7:1	18:1	15:1	41:1	5:1	18:1	36:1
HG21	75:1	12:1	NA	25:1	NA	75:1	88:1	100:1
HEL	5:1	5:1	11:1	12:1	32:1	5:1	13:1	18:1
MOZ	1:1	4:1	5:1	11:1	17:1	3:1	5:1	7:1

Table 7: Summary table: Compression ratio for main competitors. (*) We show the best results of RLZ/Tong only.

6.4 Discussion

We show the compression ratios of the most challenging competitors in Table 7. CDAG generates often the most compact search index, sometimes as compact as compression-only competitors (RLZ/ Tong). At the same time CDAG is always among the fastest competitors for creating the index structure. The median query run times are quite similar for all competitors which create an index based on referential compression. The fast competitors create an index over the concatenated strings in the collection. The slower competitors are based on online search in compressed strings and need to decompress strings at query run time.

Besides our competitors RLZ/iRLZ [17], Tong/iTong [34], RCSI [37], and CaBLAST [23], there are also other methods which have partly a similar scope: GenomeCompress [38], MuGI [6], and AliBI [10] create an index structure based on multiple sequence alignments (MSAs). Creating an optimal MSA is computationally expensive. We computed multiple sequence alignments following a consistency-based progressive alignment strategy [31] implemented in SeqAn [9]. Even for a small fraction of our evaluation dataset (up to 20 strings), the alignment time alone is larger than indexing time of best methods for the complete dataset. This shows that MSA-based approaches cannot scale up well with the number of strings. For instance, the computation of a MSA for two strings from HEL already takes 10 minutes and for three strings 23 minutes (CDAG needs a little more than one minute to index the whole dataset HEL). Computing an optimal MSA of four human chromosomes 22 takes more than one day with highly-parallelized GPU-based implementations. Notably, all compressed indexing techniques in the Bioinformatics area work on MSAs, since data from sequencing individuals of the same species is usually published in a pre-aligned format. We have already shown in previous work [37] that RCSI outperform GenomeCompress in terms of index size. GCSA [33] cannot find all matches, since their index is build on a pan-genome [6].

Furthermore, for an index over dissimilar strings, it is desirable that the space consumption during construction is smaller than the actual text size (the sum of the length of all strings). For instance, a collection of 1000 human genomes requires more than three TB of storage. Computing intermediate data structures larger than the actual text, what we call *blow-up* effect, makes the index construction infeasible, similarly to computing an alignment. Other related techniques have a severe blow-up effect. Some indexes need a complete suffix array of the to-be-indexed text as input (see Section 6.2 for sizes and indexing times of suffix arrays over concatenation of strings). LZ-End creates an intermediate index roughly five times larger than the input. Switching to compact constructions [15] can reduce the space requirements, but poses the problem that the space is compressed in terms of, at best, the k-th order empirical entropy of the text, not in terms of the size of its LZ77 parse [20]. In contrast, all MRCSI variants and iRLZ/iTong ran on all our data sets with just a few gigabyte of memory; this also applies for the hard chromosomes (data not shown).

7. CONCLUSION

In this work, we presented a novel framework for referential compression supporting approximate search. The main advantage to prior works lies in the fact that our algorithms are capable of exploiting multiple references during compression, which makes them applicable also to dissimilar string collections, which usually are out-of-scope of referential compression methods.

An interesting direction for future work is the investigation of cost-models for query answering times, instead of index size only. Furthermore, it is interesting to analyze/restrict the main memory usage during index creation.

8. REFERENCES

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

[2] A. Cannane and H. E. Williams. A general-purpose compression scheme for large collections. *ACM Trans. Inf. Syst.*, 20(3):329–355, July 2002.

[3] T. Cheng, X. Yan, and K. C.-C. Chang. Entityrank: Searching entities directly and holistically. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 387–398. VLDB Endowment, 2007.

[4] F. Claude, A. Fariña, et al. Indexes for highly repetitive document collections. In *CIKM*, pages 463–468, New York, NY, USA, 2011. ACM.

[5] M. Cohn and R. Khazan. Parsing with prefix and suffix dictionaries. In *Data Compression Conf.*, pages 180–189, 1996.

[6] A. Danek, S. Deorowicz, and S. Grabowski. Indexing large genome collections on a PC. *CoRR*, abs/1403.7481, 2014.

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

[8] S. Deorowicz and S. Grabowski. Data compression for sequencing data. *Algorithms for Molecular Biology*, 8:25, 2013.

[9] A. Döring, D. Weese, et al. Seqan an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9, 2008.

[10] H. Ferrada, T. Gagie, et al. AliBI: An Alignment-Based Index for Genomic Datasets. *ArXiv e-prints*, July 2013.

[11] H. Ferrada, T. Gagie, et al. Hybrid indexes for repetitive datasets. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 372(2016), 2014.

[12] T. Gagie, K. Karhu, et al. Document listing on repetitive collections. In J. Fischer and P. Sanders, editors, *Combinatorial Pattern Matching*, volume 7922 of *Lecture Notes in Computer Science*, pages 107–119. Springer Berlin Heidelberg, 2013.

[13] S. Gerdjikov, S. Mihov, et al. Wallbreaker: Overcoming the wall effect in similarity search. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 366–369, New York, NY, USA, 2013. ACM.

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

[15] W. Hon, K. Sadakane, and W. Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM Journal on Computing*, 38(6):2162–2178, 2009.

[16] C. Hoobin, S. Puglisi, and J. Zobel. Sample selection for dictionary-based corpus compression. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '11, pages 1137–1138, New York, NY, USA, 2011. ACM.

[17] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *PVLDB*, 5(3):265–273, 2011.

[18] Y. Jiang, D. Deng, et al. Efficient parallel partition-based algorithms for similarity search and join with edit distance constraints. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, EDBT '13, pages 341–348, New York, NY, USA, 2013. ACM.

[19] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[20] S. Kreft and G. Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483(0):115 – 133, 2013.

[21] S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative lempel-ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of SPIRE 2010*, pages 201–206, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] N. Larsson and A. Moffat. Offline dictionary-based compression. In *Data Compression Conference, 1999. Proceedings. DCC '99*, pages 296–305, Mar 1999.

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

[24] A. Max and G. Wisniewski. Mining naturally-occurring corrections and paraphrases from wikipedias revision history. In N. Calzolari, K. Choukri, et al., editors, *LREC 2010*, Valletta, Malta, 2010. European Language Resources Association.

[25] E. McCreight. Efficient algorithms for enumerating intersection intervals and rectangles. Technical report, Xerox Paolo Alte Research Center, 1980.

[26] S. M. Mola-Velasco. Wikipedia vandalism detection. In S. Srinivasan, K. Ramamritham, et al., editors, *WWW (Companion Volume)*, pages 391–396. ACM, 2011.

[27] G. Navarro, S. Puglisi, and J. Siren. Document retrieval on repetitive collections. In A. Schulz and D. Wagner, editors, *Algorithms - ESA 2014*, volume 8737 of *Lecture Notes in Computer Science*, pages 725–736. Springer Berlin Heidelberg, 2014.

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

[29] M. Patil, S. V. Thankachan, et al. Inverted indexes for phrases and strings. In *SIGIR 2011*, pages 555–564, New York, NY, USA, 2011. ACM.

[30] A. J. Pinho, D. Pratas, and S. P. Garcia. Green: a tool for efficient compression of genome resequencing data. *Nucleic Acids Research*, Dec. 2011.

[31] T. Rausch, A.-K. Emde, et al. Segment-based multiple sequence alignment. *Bioinformatics*, 24(16):i187–i192, 2008.

[32] K. Schneeberger, J. Hagmann, et al. Simultaneous alignment of short reads against multiple genomes. *Genome biology*, 10(9):R98+, Sept. 2009.

[33] J. Siren, N. Välimäki, and V. Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, (accepted).

[34] J. Tong, A. Wirth, and J. Zobel. Principled dictionary pruning for low-memory corpus compression. In *Proceedings of the 37th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '14, pages 283–292, New York, NY, USA, 2014. ACM.

[35] S. Wandelt, M. Bux, and U. Leser. Trends in genome compression. *Current Bioinformatics*, 9(3):315–326, 2014.

[36] S. Wandelt and U. Leser. FRESCO: Referential compression of highly similar sequences. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 10(5):1275–1288, Sept. 2013.

[37] S. Wandelt, J. Starlinger, et al. RCSI: Scalable similarity search in thousand(s) of genomes. *PVLDB*, 6(13):1534–1545, 2013.

[38] X. Yang, B. Wang, et al. Efficient direct search on compressed genomic data. In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *ICDE*, pages 961–972. IEEE Computer Society, 2013.

[39] M. Zukowski, S. Heman, et al. Super-scalar RAM-CPU cache compression. In *ICDE*, pages 59–70, Washington, DC, USA, 2006. IEEE Computer Society.