LZ-Compressed String Dictionaries

Julian Arz and Johannes Fischer

KIT, Karlsruhe, Germany.
`julian.arz@student.kit.edu,johannes.fischer@kit.edu`

Abstract. We show how to compress string dictionaries using the Lempel-Ziv (LZ78) data compression algorithm. Our approach is validated experimentally on dictionaries of up to 1.5 GB of uncompressed text. We achieve compression ratios often outperforming the existing alternatives, especially on dictionaries containing many repeated substrings. Our query times remain competitive.

1 Introduction

A *string dictionary* is a data structure that stores a set of words and identifies each word with a unique identifier. It has to support two straightforward operations: return a word, given its ID (*access*), and return the ID of a given word (*lookup*). Other operations, such as searching for all words with a certain suffix or prefix or containing some substring, are optional. String dictionaries are a basic tool for the processing and indexing of strings, whenever a mapping from a set of words to a unique ID is needed.

Though being a ubiquitous problem, to the best of our knowledge not much emphasis has been put on the *size* of string dictionaries. A likely explanation for this is that in the past the set of strings to be stored in a dictionary were not big enough to justify complex data structures (the baseline algorithms were sufficient). For example, a typical application for string dictionaries are geographical information systems, where all geographic names of a region need to be stored in a dictionary. But even on a continental-size map the size of such a dictionary is manageable with traditional techniques. However, this situation has changed in recent years, as string dictionaries are becoming larger and larger, and also because on mobile devices (e.g. GPS's), available memory is still a critical factor.

A rather classical application arises in information retrieval. Huge amounts of data have to be organized in a way that facilitates the extraction of small fragments. In document retrieval systems for example, collections of text documents are indexed so that a user can *query* them, say find all documents where a certain word occurs. Here, string dictionaries are used in an extended form: *inverted indexes* assign to every word present in the collection a list of documents they appear in, and a first step for locating a given term is to search it in the dictionary. These dictionaries can become quite big. While the total number of

words in the English language is estimated to about $1\,000\,000$ [8], recent crawls of web pages written in 10 different languages resulted in a data set of 200 million different words. This can be due to typing errors, but also stems from the fact that in some languages, for example in German, new words can be built by the concatenation of two existing words.

In databases of web platforms (e.g. social networks), string attributes are commonly used to store data such as user information, private messages or guest-book entries. Tables can consist of several string attributes and a unique primary key (ID), a number. This ID is usually stored as a foreign key in another table. A user is then able to either search for a string to use its ID in another context, or obtain the string to a given ID, which makes databases an ideal application for string dictionaries. Often, database columns are indexed to hasten the search, at the expense of additional space overhead. *Column based internal memory data bases* are another natural example where string dictionaries arise and are a hot research topic.

All of the given examples could profit from a reduction of the space overhead of the used string dictionary. Thus, the question arises how string dictionaries can be compressed. Two trivial solutions come to mind: First, we could regard a set of separate words concatenated to one string, and compress this string using any state of the art lossless data compression technique, e.g. *deflate* or other algorithms. This results in great compression ratios, but poor efficiency for the required operations, as in the worst case the whole text has to be decompressed to access one word. Second, we could compress each word separately and use typical string dictionary methods like hashing to provide the functionality. In that case, the two operations were almost as fast as in traditional string dictionaries, but the compression is far from optimal.

1.1 Related Work

The immediate solution to the string dictionary problem is to store the strings in a *trie*, where the leaves are annotated with the identifiers. A related but more practical idea is to sort the strings lexicographically and encode a string as a pair (ℓ, α), where ℓ is the length of the longest common prefix with its lexicographic predecessor and α is the remaining suffix. This idea is known as *front coding*. To support fast access- and lookup-operations, every k'th string is stored *verbatim*, for some suitably chosen value of k. However, none of these simple methods provides general-purpose compression.

Research on compressed string dictionaries is more recent, and we are aware of only a few works tackling this problem. The first is the *compressed permuterm index* [5] that builds on the Burrows-Wheeler transformation. It supports a rich set of operations for IR tasks, but if restricted to our simple access/lookup functionality its space is not competitive. Brisaboa et al. [2] evaluate the practical performance of techniques like Huffman coding, hashing, front coding, grammar-based compression, and full text indexing. In brief, they find that (a) front

coding with Hu-Tucker character compression and (b) Re-Pair-based indices provide the best time/space trade-offs. The most recent work is due to Grossi and Ottaviano [7] and builds on previous ideas of the first author [3]. It augments the basic trie idea with *path decomposition*, and is shown to often perform better than [2]. All approaches employ engineered implementations of *succinct data structures* to achieve good practical performance.

1.2 Our Contribution

We improve the empirical performance of dictionary-based compressed string dictionaries, namely Lempel-Ziv (LZ) compressed string dictionaries. The only existing dictionary-based approach is Re-Pair [2]. Compared to the latter, we achieve very competitive compression rates, often better. Accesses are 2–4 times slower, but lookups are up to twice as fast. Construction of our data structure is one order of magnitude *faster* than Re-Pair. Compared to the path-decomposed tries [7], we achieve about the same compression ratios, but often slower operations. A notable exception is a data set containing many often repeated substrings, where we can compress to about a third of the original file size, whereas the path-decomposed trie achieves only 1/2. In total, our approach proves to be very robust due to the direct use of a general-purpose compressor from the LZ-family.

2 Preliminaries

In this section we introduce known concepts and tools on which our data structure is based. We start by formally defining our problem as follows. Let $\mathcal{S} = \{s_0, \ldots, s_{m-1}\} \subset \Sigma^\star$ be a set of m strings, and let $n = \sum_{i<m} |s_i|$ denote their combined length. A *string dictionary* over $\mathcal{S}$ is a data structure that supports the following operations:

- Lookup(s): return -1 if $s \notin \mathcal{S}$ or a unique identifier in $[0, m)$ otherwise.
- Access(i): get the string with identifier $i \in [0, m)$, Lookup(Access(i)) = i.

2.1 LZ78 Data Compression

Our new algorithm is based on the LZ78 compression algorithm [11] for a string $S[0, n)$, which we are going to describe next. The algorithm proceeds by parsing S from left to right and dividing it into blocks (called *phrases*) that are one-letter extensions of previously seen phrases. The set of current phrases is called the *phrase dictionary*. At the beginning of the algorithm the phrase dictionary contains only the empty string. Now assume that we have already parsed a prefix $S[0, i)$ of S, and that the phrase dictionary is $\mathcal{D}$. Then the next phrase is chosen to be $S[i, j]$ such that $S[i, j]$ is the longest string already in $\mathcal{D}$. Further, the new phrase $S[i, j]$ is added to $\mathcal{D}$. Due to the construction, the dictionary $\mathcal{D}$ is prefix-closed and is naturally represented with a trie.

2.2 Succinct Data Structures

Consider a *bit-string* $S[0, n)$ of length n. We define the fundamental *rank*- and *select*-operations on S as follows: $rank_1(S, i)$ gives the number of 1's in the prefix $S[0, i]$, and $select_1(S, i)$ gives the position of the i'th 1 in S, reading S from left to right ($0 \leq i < n$). Operations $rank_0(S, i)$ and $select_0(S, i)$ are defined similarly for 0-bits. The following lemma summarizes a by-now classic result; well-performing practical implementations of this lemma exist, we used the SDS-Library [6].

Lemma 1 (see, e.g., [9]). *A bit-string of length n can be represented in $n+o(n)$ bits such that rank- and select-operations are supported in $O(1)$ time.*

3 New Data Structure

We now present the theory of our new algorithm. We proceed by first showing a data structure that supports the access-operation (Sect. 3.1), and then modifying this data structure to also support the lookup-operation efficiently (Sect. 3.2).

3.1 Basic Idea: Supporting Access

We explain a first idea how to adapt the LZ78-parsing from Sect. 2.1 to the string dictionary problem. For ease of explanation, we assume that each string s_i is terminated by a unique letter $\#_i \notin \Sigma$. Then we can *concatenate* all strings into a single large string $S = s_0 s_1 \ldots s_{m-1}$ of length n, without losing information about the word boundaries. The basic approach is to compress S with the LZ78-parsing algorithm from Sect. 2.1 and store the resulting LZ-trie. Note that due to the unique separators $\#_i$, there is a phrase ending at the end of every string, and hence every string also starts with a new phrase.

For the recovery of the original strings in $\mathcal{S}$, we *link* the phrases from one string as follows. Suppose s_i is parsed as $s_i = p_0^i |p_1^i| \ldots |p_{k-1}^i|$, where the p_j^i are phrases. Note that each p_j^i corresponds to a unique node v_j^i in the LZ-trie. For $1 \leq j < k$, we make a link from v_j^i to v_{j-1}^i. We call those links the *predecessor links*. See Fig. 1 for an example. We can store an additional array $A[0, m)$ such that $A[i]$ points to the node corresponding to the last phrase p_{k-1}^i of s_i (the one ending with $\#_i$).

Now the access-operation can be easily supported. Suppose we want to answer Access(i), and that the parsing of s_i is $p_0^i |p_1^i| \ldots |p_{k-1}^i|$. Then we first go to node $v_{k-1}^i = A[i]$ and recover s_i's last phrase p_{k-1}^i by following the path from v_{k-1}^i towards the root. Then we follow the predecessor link of v_{k-1}^i to v_{k-2}^i and recover the penultimate phrase. This goes on until we have recovered the first phrase p_0^i, which happens iff the predecessor link is `nil`. As a result, we have recovered the i'th string from right to left.

Fig. 1: LZ-trie for the set of strings $\mathcal{S} = \{\texttt{a|b|a|, ab|aba|, abc|, abcb|, ba|, bac|bacb|, bacba|c|ba|, bc|a|}\}$, with the parsing indicated by "|". All non-solid lines are predecessor links enabling the efficient access to strings.

Unfortunately, this data structure does not readily support the lookup in optimal $O(|s|)$ time, because the parsing of phrases is not unique. For example, consider querying for the string $\texttt{aba}$ in the example of Fig. 1. Matching it greedily in the LZ-trie we arrive at the node spelling the string $\texttt{aba}$, from which we cannot derive a correct identifier. Indeed, what we should have done is matching $\texttt{aba}$ as it was parsed: first $\texttt{a}$, then $\texttt{b}$, and finally $\texttt{a}\#_1$, from which we could have derived that '1' is the true identifier of the string $\texttt{aba}$. However, the LZ-trie does not seem to contain sufficient information to decide that after matching the first $\texttt{a}$ we should have started a new phrase.

3.2   Modification for Supporting Lookup

The problem of the previous section is that the parsing of a string s_i depends on the past, i.e., on the parsing of all strings s_j for $j < i$. We resolve this problem by *reparsing* the strings to make their parsing unique and enable a greedy parsing of query strings. More precisely, we first construct the LZ-trie from Sect. 3.1 (excluding the predecessor links and end-of-string markers $\#_i$). Then, we run through the dictionary $\mathcal{S}$ *again* and reparse all strings by matching them *greedily* in the existing trie. This implies that phrases can now be used multiple times. The advantage is that the parsing is now *unique* in the following sense: say that s_i is parsed as $s_i = p_0^i|p_1^i|\ldots|p_{k-1}^i|$, and that a different string $s_j = p_0^j|p_1^j|\ldots|p_{\ell-1}^j|$ is prefixed by $p_0^i p_1^i \ldots p_{y-1}^i$ for some $y \leq k$. Then $p_x^i = p_x^j$ for all $x < y$, and further, if the longest common prefix between s_i and s_j extends r characters into p_y^i, then those r characters are also a prefix of p_y^j.

5

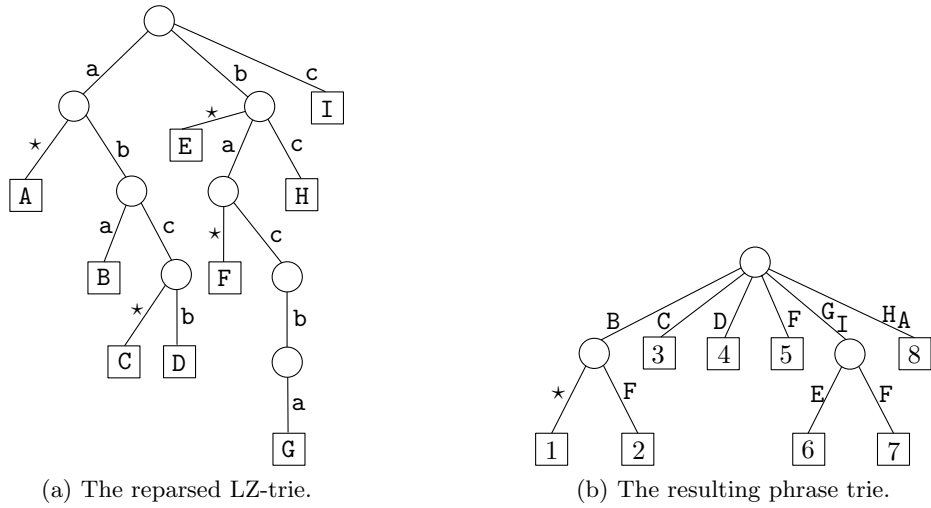(a) The reparsed LZ-trie.   (b) The resulting phrase trie.

Fig. 2: The final data structures. The set of strings from Fig. 1 is now parsed as $\mathcal{S} = \{$aba|, aba|ba|, abc|, abcb|, ba|, bacba|c|b|, bacba|c|ba|, bc|a|$\}$, which corresponds to the strings $\mathcal{S}' = \{$B, BF, C, D, F, GIE, GIF, HA$\}$ in the phrase alphabet.

For example, string bacbacb in Fig. 1 is now parsed bacba|c|b instead of bac|bacb. Although the new parsing is longer in this case, it is parsed similar to the next string bacba|c|ba. See Fig. 2a for the resulting LZ-trie, where phrases that are a prefix of a different phrase are terminated with a special character "⋆" in order to make all phrases end at a leaf of the LZ-trie. (This allows us to identify all phrases with leaf identifiers; in Fig. 2a denoted by upper case letters.)

Two issues arise that have to be dealt with now:

- Some nodes from the original LZ-trie could now be superfluous, since they are not reached anymore by any used phrase. Such nodes can simply be deleted from the resulting trie.
- More seriously, it could happen that the greedy parsing cannot continue, for example when a phrase is parsed longer than originally, but there is no outgoing edge from the root with the next character. This problem can be solved by initially inserting all single letters $a \in \Sigma$ into the trie.

Due to the multiple use of phrases we cannot work with plain predecessor pointers as before. To overcome this, we construct *another* trie consisting of the parsed phrases in the new *phrase alphabet*. We call this second trie the *phrase trie*. See Fig. 2b for an example. (This trie contains exactly the predecessor pointers, but arranged in a form more suitable for querying, as we shall see.)

Now Access(i) works slightly differently: jump to the i'th leaf of the phrase trie and find the phrases of s_i. Those phrases can now be easily recovered using the LZ-trie, since there they correspond to leaves. The time is the optimal $O(|s_i|)$.

The advantage of the new structure is that it also enables optimal Lookup(s): first, parse s greedily using the LZ-trie, e.g. $s = p_0|p_1|\ldots|p_{k-1}|$. Then we try matching the parsed phrases in the phrase trie; this takes $O(k)$ time (assuming perfect hashing). If matching is successful and ends in a leaf, we return the identifier stored there (the string ID). Otherwise s does not occur in $\mathcal{S}$. The total time for this process is optimal $O(|s|)$.

4 Implementation Details

In this section we describe our implementation that builds on the data structure from Sect. 3. As often in algorithmic engineering, we sometimes deviate from the theoretical proposal and sacrifice the optimal running times in exchange for a faster practical performance.

4.1 Representation of the LZ-Trie

The choice of the LZ-trie implementation offers a trade-off between time for each of the two operations and space overhead. Besides a trie, other data structures are possible as well: the required operations are access and `longest_prefix` (finding the longest element which is a prefix of a given string).

We present two representations. The first is a well-tuned existing trie implementation, the path-decomposed tries of Grossi and Ottaviano [7]. One thing to remark is that while the path decomposition has the advantage of fast operation times due to high cache locality, its drawback is that in its original form, only the leaves of the original trie can be accessed, whereas the inner nodes are "hidden" in the path decomposition. Instead of appending a unique character "$\star$" to the end of each phrase as in Sect. 3.2, we identify a phrase with a tuple consisting of the subpath it ends on in the path decomposition and the offset, counting from the beginning of this subpath. We map these tuples to ordinary numbers using two bit vectors enhanced with rank/select-support. We found this to be more space efficient than the explicit end-of-string character.

The second representation is based on a front coding dictionary, as the one described in Sect. 1.1. We support `longest_prefix` using a characteristic of our LZ-parsing: when a parsed string s and a phrase p have an lcp r, the longest prefix is at least as long as r. We search the given string in the dictionary, first with a binary search on the explicitly stored entries and then with a linear search in a bucket. If this search does not find prefix, we either search the lcp of that bucket's first entry and the string, or abort the search (and return -1) based on the already compared strings. Thus, at most two search operations are performed.
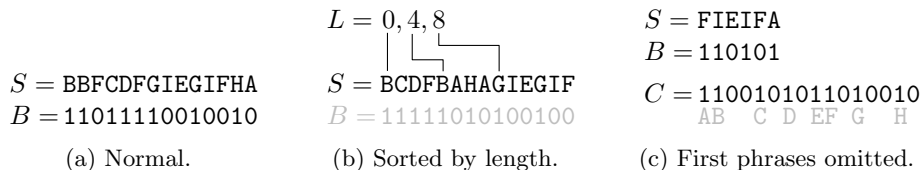
$$L = 0, 4, 8$$

$$S = \texttt{BBFCDFGIEGIFHA} \qquad S = \texttt{BCDFBAHAGIEGIF} \qquad S = \texttt{FIEIFA}$$

$$B = \texttt{11011110010010} \qquad B = \texttt{11111010100100} \qquad B = \texttt{110101}$$

$$C = \texttt{1100101011010010}$$
$$ \texttt{AB C D EF G   H}$$

(a) Normal. (b) Sorted by length. (c) First phrases omitted.

Fig. 3: The linearized phrase trie, where (b) and (c) show two optimization techniques. Information in gray need not be stored explicitly.

4.2 Representation of the Phrase Trie

Due to the potentially very large alphabet (p, the number of different phrases, can be very high), existing trie implementations cannot be used efficiently for storing the phrase trie. Instead, we chose a different approach, as explained next.

We linearize the phrase trie by juxtaposing all parsed phrases into a string S, see Fig. 3a for the running example. In order to know where a string s_i starts, we store an additional bit vector B of the same length as S, where a '1' indicates the beginning of a string. Then the parsing of the i'th string can be found by $select_1(B, i)$, and hence the access-operation can be easily supported. On the other hand, for Lookup(s) we now have to search the parsing of $s = p_0|p_1|\dots|p_{k-1}|$ in S; we do this by a *binary search* with the help of *select*-operations over B. As an example, consider searching $s = \texttt{bc|ba|a|}$, which corresponds to $s' = \texttt{HFA}$ in the phrase alphabet. Since there are $n = 8$ strings in $\mathcal{S}$, we first go to the 4th (middle) string by $select_1(B, 4)$, see that it is $\texttt{D}$, and since it is alphabetically smaller than the parsed query string $\texttt{HFA}$, we continue the search in the right half. To make the binary search work, the parsed strings are sorted before writing them to S.

Two optimizations can be applied to this base variant of the linearized phrase trie. The first is to rearrange the parsed strings in S such that they are first sorted by their length (number of phrases), and then lexicographically. The bit vector B can then be discarded, since the string beginnings can be calculated arithmetically within a range of strings of equal length (see Fig. 3b). We need an additional small array L to know where the phrases of a given length begin in S. The access-operation uses this array to find the range containing the requested index. For the lookup-operation, we can identify the correct range directly by counting the number of phrases the searched string was parsed into. Thus, during the binary search we omit the *select*-operations which, although constant-time, introduce a considerable time overhead due to several table lookups.

The goal of this optimization is thus not so much a reduction in space (as B is small compared to S) but rather an acceleration of the lookup-operation. In our experiments we found that most of the strings are parsed into two or three phrases (mean ≈ 2.7), but a small percentage is parsed into more (up to the hundreds) of phrases. To account for this, we chose to sort by length only the

strings with a small number of phrases (at most 5), and handle all larger strings as before (*with* the bit vector B, which is now very short). The additional time overhead for the access-operation is then negligible.

The second optimization is to *omit* the first phrase from every string in a lexicographically sorted range of S. Then we need an additional bit vector C that encodes how many strings start with a given character in the phrase alphabet. This can be done, e.g., by writing a 1 for every phrase character, followed by k 0's if there are k strings starting with that phrase (see Fig. 3c). Preparing C for select queries (on both 0- and 1-bits) then allows to recover the original contents of S. For Access(i), we count the number of 1-bits up to the i'th 0-bit in C to retrieve the first phrase. For the remaining phrases, we proceed similarly for B. During the lookup-operation, we only need to search in the range of parsings starting with p_0. We find this range with a $select_1(C, p_0)$.

This optimization turned out to be very effective for large dictionaries, since with p phrases we save $n \lg p$ bits by dropping the first phrases, whereas array C occupies only $p + n$ bits, much less than $n \lg p$ for large n and typical values of $p = O(n/\lg n)$. The lookup-operation is faster as well, because less comparisons are performed during the binary search.

Both optimizations can be combined. Then for each range of strings with equal parsing length ℓ a bit vector C_ℓ is used. Again we only support the ranges up to a certain number of phrases. If there are n_ℓ strings in a range, the size of C_ℓ is $n_\ell + p$ bits, opposed to the savings of ℓn_ℓ bits for omitting B in that range. Thus this combination yields better compression rates only if the number of strings in one range is large enough to compensate for the p bits added for every supported range.

5   Experimental Results

We perform several experiments on real world data as well as a synthetic data set to evaluate our algorithm. We also compare our data structure to other relevant structures for compressed string dictionaries.

Setting. We use the following data sets, of which URLs were also used in two previous experiments [2,7], and Wiki was used only by Grossi and Ottaviano [7]:

Wiki consists of all page titles of the English Wikipedia of April 2011,
URLs are the URLs of a 2002 crawl by the UbiCrawler [1] on the `.uk` domain,
DNA is the DNA data set from Pizza&Chili Corpus [4], split into strings of 30 characters, and
synth-$\alpha\beta\alpha$ is a data set we constructed artificially. These are strings of the form $\alpha_1 \beta \alpha_2$, where the α_i's are randomly chosen strings which occur multiple times. Strings β are short strings to separate these blocks. See Appendix A for technical details on this data set. This data set was chosen to show that

Table 1: Comparison of the size of LZ-parse before and after the reparsing. All values are 10^3.

	synth-$\alpha\beta\alpha$		Wiki		URLs		DNA	
	before	after	before	after	before	after	before	after
#Nodes	24340	15722	13597	12536	46716	34032	24832	24054
#Phrases (p)	24340	6873	13597	8624	46716	17652	24832	18105
#Parsing	24340	21324	13597	19782	46716	55008	24832	37123

> path decomposition does not always result in better compression ratios than purely dictionary-based methods.

Our testing machine is an AMD Opteron 8350, equipped with 4 cores (of which we only use one) and 64 GiB of main memory. The CPU is clocked with 2.0 GHz and is supported by 2 MiB Cache. The machine is running Ubuntu 10.04.4 LTS (kernel 2.6.32). All algorithms were implemented in C++ and compiled using the GNU C++ compiler version 4.4.3 with optimization level -O3. We chose 1,000,000 random indices and strings from each data set for the timing of the operations. All measured times are averaged over 3 runs.

Reparsing. Table 1 gives numbers on the size of the LZ-trie before and after the reparsing. #Nodes is the number of nodes in the (uncompacted) trie. The smaller numbers after reparsing are a result of cutting off entire subtrees from the LZ-trie. The second row gives the number of nodes representing phrases (lower after reparsing since not every prefix of a phrase is necessarily also a phrase). The row "#Parsing" gives the size of the parsing. All these numbers are equal before the reparsing as every node in the trie corresponds to a phrase and every phrase is used exactly once in the parsing. We observe that the reparsing reduces the size of the LZ-trie by about 27% for URLs and 8% for Wiki. For our synthetic data set, the reduction is 35%. These results confirm the intuition that the LZ-based reparsing strategy excels for collections of strings where sufficiently long substrings occur multiple times, e.g. "`http://`", "`index`" or "`.html`" for URLs. The size of the parsing is increased by between 18% and 50%, but this change is not reflected in the size of the trie, and it is the role of our phrase trie implementation to cope with this increase.

Other Data Structures. We compare our data structure to previously known techniques. We did not include uncompressed dictionaries (like TX-trie [10]) in our evaluation, as they were already shown to use much more space in previous studies [2,7]. LZT is our approach of the reparsed LZ-trie, using the implementation described in Sect. 4. We examine the trade-offs for two different representations of the LZ-trie, one using path decomposition (LZT-pd) and the other using front coding with bucket size 16 (LZT-fc). Table 2 gives more information about the distribution of space consumption on our two components. In all cases,

Table 2: The compression ratio of our data structure broken down into its two components. All percentages are in relation to the original file size.

	(a) LZ-Trie represented by PDT (LZT-pd)				(b) by front coding (LZT-fc)			
	synth-$\alpha\beta\alpha$	Wiki	URLs	DNA	synth-$\alpha\beta\alpha$	Wiki	URLs	DNA
File Size [MB]	212	171	1 439	400	212	171	1 439	400
LZ-Trie [%]	11.1	11.6	3.3	8.6	15.8	20.5	7.7	19.1
Phrase Trie [%]	23.8	23.0	8.8	21.4	23.8	23.0	8.8	21.4
Combined [%]	34.9	34.6	12.2	30.0	39.6	43.4	16.4	40.5

the phrase trie representation accounts for about than two thirds of the overall space (slightly more than two thirds for LZT-pd, and slightly less for LZT-fc).

Tables 3a to 3d present the experimental comparison with other approaches. PDT is the centroid path-decomposed trie [7] in the compressed variant. The comparison between LZT and PDT is interesting, as we actually use the PDT in our implementation. There is a remarkable duality between the two: PDT uses a grammar-based compression on a (linearized) trie built on all strings, while LZT builds a trie on a grammar-based compression scheme. Re-Pair, front coding (FC) and Hu-Tucker front coding (HTFC) are examined in [2]. We experimentally deduced the best bucket size to be 8 for both FC and HTFC, slightly favoring the compression.

We observe that the compression ratio of LZT-pd is among the lowest for all data sets and is at most 12 % higher than the best for each set. Remarkably, it achieves the best compression ratio for the URLs data set. The nature of this set (long common prefixes) should intuitively favor the front coding–based data structures, but they do not detect the common substrings which are not at the beginning of the strings. For our synthetic data set the difference to PDT is even more pronounced, as our data structure compresses to 34.0%, whereas PDT achieves only 53.2% compression rate. Re-Pair is even slightly better in this case, but the construction is more than 15 times longer. As expected, the compression comes at the cost of higher times for the operations. This is because the strings are parsed into an average of 2.3–4 phrases (depending on the data set), and therefore this amount of elementary trie operations has to be performed, while PDT only requires one such operation. The variant LZT-fc alleviates this deficiency by sacrificing slightly more space in exchange for much faster access-times. Lookup-times are also sped up (though not as significantly, but still faster than Re-Pair).

11

Table 3: Comparison of our data structure with others.

(a) synth-$\alpha\beta\alpha$ (5.4×10^6 strings)

	constr [s]	cmpr [%]	access [µs/ID]	lookup [µs/str]
LZT-pd	177.8	34.9	15.3	16.7
LZT-fc	96.0	39.6	5.4	12.8
Re-Pair	2959.7	31.0	3.8	14.4
PDT	291.0	53.2	4.1	4.1
FC	0.55	68.0	0.59	2.0
HTFC	2.52	49.4	4.8	7.0

(b) Wiki (8.5×10^6 strings)

	constr [s]	cmpr [%]	access [µs/ID]	lookup [µs/str]
LZT-pd	99.0	34.6	9.9	10.7
LZT-fc	53.7	43.4	3.3	8.2
Re-Pair	1017.6	41.5	3.8	14.4
PDT	85.3	32.1	4.1	4.1
FC	0.79	60.2	0.62	2.1
HTFC	2.35	43.2	2.6	5.0

(c) URLs (18.5×10^6 strings)

	constr [s]	cmpr [%]	access [µs/ID]	lookup [µs/str]
LZT-pd	311.9	12.2	15.8	16.7
LZT-fc	197.8	16.4	4.7	14.4
Re-Pair	12 069.7	12.4	4.2	31.0
PDT	244.2	13.6	6.3	6.2
FC	2.8	32.7	0.74	3.5
HTFC	9.9	24.4	5.7	9.9

(d) DNA (12.9×10^6 strings)

	constr [s]	cmpr [%]	access [µs/ID]	lookup [µs/str]
LZT-pd	217.8	30.0	16.0	16.0
LZT-fc	120.1	40.5	3.9	11.5
Re-Pair	15 537.4	37.2	2.3	12.7
PDT	188.1	26.7	5.9	5.9
FC	1.3	69.7	0.59	2.3
HTFC	5.3	30.9	2.5	5.1

6 Conclusions

We proposed a new LZ-like parsing for string dictionaries that allows greedy trie-based pattern matching. In our implementation we combined this parsing with space-efficient representation techniques for the resulting two tries. Our data structure competes with other data structures regarding query times, and it often achieves better compression ratios, particularly for string collections containing highly repetitive patterns. Our work was not focused on the trie implementation. Representations better suited for phrases of an LZ-parse can achieve even better results. We also aim to investigate other parsings and grammars which might have convenient characteristics.

Acknowledgments

References

1. P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.
2. N. R. Brisaboa, R. Cánovas, F. Claude, M. A. Martínez-Prieto, and G. Navarro. Compressed string dictionaries. In *Proc. SEA*, volume 6630 of *LNCS*, pages 136–147. Springer, 2011.
3. P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter. On searching compressed string collections cache-obliviously. In *Proc. PODS*, pages 181–190. ACM, 2008.
4. P. Ferragina and G. Navarro. The Pizza&Chili Corpus. `http://pizzachili.dcc.uchile.cl/texts/dna/`.
5. P. Ferragina and R. Venturini. The compressed permuterm index. *ACM Transactions on Algorithms*, 7(1):Article No. 10, 2010.
6. S. Gog. Succinct Data Structures Library. `http://simongog.github.io/sdsl/`.
7. R. Grossi and G. Ottaviano. Fast compressed tries through path decompositions. In *Proc. ALENEX*, pages 65–74. SIAM Press, 2012.
8. J.-B. Michel et al. Quantitative analysis of culture using millions of digitized books. *Science*, 331(6014):176–182, 2011.
9. J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comput.*, 31(3):762–776, 2001.
10. D. Okanohara. Tx: Succinct trie data structure. `http://code.google.com/p/tx-trie/`.
11. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inform. Theory*, 24(5):530–536, 1978.

A Synthetic Data Sets

synth-$\alpha\beta\alpha$. Let $\Sigma^\alpha = \{a, \ldots, z\}$ and $\Sigma^\beta = \{!, \ldots, @\}$. We have $|\Sigma^\alpha| = 26$, $|\Sigma^\beta| = 32$ and $\Sigma^\alpha \cap \Sigma^\beta = \emptyset$. We define two pools (a.k.a. multisets) of substrings. Every element of a pool is only used once in a constructed string. Let Γ be a pool of randomly generated strings out of $(\Sigma^\alpha)^{16}$. Each string occurs 32 times. Another pool Φ is built upon all lexicographically ordered strings of length 6 over the alphabet Σ^β. There are $\binom{32}{6}$ of these strings. Each of these strings occurs 6 times, so the size of Φ is $6\binom{32}{6} = 5\,437\,152$. Then, synth-$\alpha\beta\alpha$ consists of $|\Phi|$ strings of the form $\alpha_1\beta\alpha_2$, where the α_i are taken at random from Γ and the β from Φ. Thus, Γ has to be twice as large as Φ, which means it has to contain $|\Phi|/16 = 339\,822$ different strings.