Practical String Dictionary Compression Using String Dictionary Encoding

Shunsuke Kanda*, Kazuhiro Morita, and Masao Fuketa
Graduate School of Advanced Technology and Science,
Tokushima University, Tokushima, Japan
**Research Fellow of Japan Society for the Promotion of Science, Japan*
Email: shnsk.knd@gmail.com, {kam,fuketa}@is.tokushima-u.ac.jp

***Abstract*—A string dictionary is a data structure for storing a set of strings that maps them to unique IDs. It can manage string data in compact space by encoding them into integers. However, instances have recently emerged in practice where the size of string dictionaries has become a critical problem for very large datasets in many applications. A number of compressed string dictionaries have been proposed as a solution. In particular, the application of Re-Pair, a powerful text compression technique, to tries and front coding can help to obtain compact string dictionaries that support fast dictionary operations. However, the cost of constructing such dictionaries using Re-Pair is impractical for large datasets. In this paper, we propose an alternative compression strategy using string dictionary encoding and develop several dictionary structures for it. We show that our string dictionaries can be constructed up to 422.5x faster than the Re-Pair versions with competitive space and operation speed, through experiments on real-world datasets.**

1. Introduction

A *string dictionary* is a data structure for storing a set of strings that maps them to unique IDs. In other words, it supports two primitive operations: LOOKUP returns the ID corresponding to a given string, and ACCESS returns the string corresponding to a given ID. As the mapping is very useful for string processing and indexing, string dictionaries are a basic tool in many kinds of applications for Natural Language Processing, Information Retrieval, Semantic Web, Bioinformatics, Geographic Information Systems, and so on [1]. Recently, Martínez-Prieto et al. [1] reported a number of real examples where the *size* of string dictionaries emerges as a critical problem for very large datasets. Therefore, a number of compressed string dictionaries, focusing on static applications, have been proposed as a solution [1]–[4].

To implement high-performance string dictionaries, two choices concerning implementation technique and compression strategy are very important. With respect to the former, string dictionaries based on *tries* [5], [6] and *front coding* [7] have yielded the best performance. With respect to the latter, *Re-Pair* [8] is a powerful text compression technique that can implement very small string dictionaries supporting fast LOOKUP and ACCESS operations. For example, Martínez-Prieto et al. [1] proposed compressed front coding dictionaries using Re-Pair. Grossi and Ottaviano [2] proposed cache-friendly compressed trie dictionaries using Re-Pair. However, Re-Pair compression incurs large construction costs, although it is theoretically executed in linear time and space over the length of a given text.

In this paper, we propose a compression strategy that uses string dictionaries for dictionary compression rather than Re-Pair. We encode strings appearing in dictionaries into integers using another string dictionary. This strategy is inspired by studies on compressing trie structures using the same structures [9], [10]. In Section 3, we show how to apply the strategy to string dictionaries developed in [1], [2]. In Section 4, we propose several novel dictionary structures for our strategy. In Section 5, we evaluate the developed dictionaries through experiments on real-world datasets.

2. Preliminaries

This section defines basic notations and introduces the basic tools for compact data structures.

2.1. Basic Notations

Strings are drawn from a finite alphabet Σ of size σ. An array that consists of n elements, $A[1]A[2]\dots A[n]$, is denoted by $A[1,n]$. Functions $\lfloor a \rfloor$ and $\lceil a \rceil$ denote the largest integer not greater than a and the smallest integer not less than a, respectively. For example, $\lfloor 2.4 \rfloor = 2$ and $\lceil 2.4 \rceil = 3$. The base of the logarithm used is 2 in this paper.

2.2. Rank Operation

Given a bit array $B[1,n]$, we define the basic operation $\text{RANK}_b(B,i)$ that returns the number of occurrences of bit $b \in \{0,1\}$ in $B[1,i]$. For example, $\text{RANK}_1(B,6) = 2$ and $\text{RANK}_0(B,4) = 3$ for $B[1,8] = [00100110]$. This operation can be performed in constant time with $o(n)$ additional bits [11], [12].

2.3. DFUDS Representation

The *DFUDS (depth-first unary degree sequence) representation* [13] is a succinct tree representation [14] that represents an ordered tree using parentheses (and). DFUDS

encodes a node with d children into d (s and one). For example, a node with three children is encoded into ((()). An ordered tree is represented by concatenating the sequence of parentheses in depth-first order while prepending an initial (. DFUDS supports basic navigation operations on a tree with n nodes in constant time with $2n + o(n)$ bits [11], [15].

2.4. Elias-Fano Representation

The *Elias-Fano representation of monotone sequences* [16], [17] is an encoding scheme to represent a non-decreasing sequence. When the sequence consists of m integers in $[0, n)$, the representation uses $2m+m\lceil \log \frac{n}{m} \rceil + o(m)$ bits while supporting direct constant-time access.

2.5. Re-Pair Compression

Re-Pair [8] is a practical technique of grammar-based compression [18]. It finds the most frequent pair xy in a text and replaces all its occurrences with a new symbol z. A new rule $z \rightarrow xy$ is added to dictionary R. This process iterates until all remaining pairs are unique in the text. As a result, the original text is encoded into a compressed sequence C with dictionary R. Each symbol of C is decoded by recursively expanding the rules in R. This time taken depends on its recursion depth.

3. String Dictionaries

A string dictionary is a data structure to store a set of strings $S \subset \Sigma^*$ and map each string to unique identifiers in $[1, |S|]$. The string dictionary supports two basic operations:

- LOOKUP(s) returns the ID if string $s \in S$.
- ACCESS(i) returns the string with ID $i \in [1, |S|]$.

Encoding strings $s_1, s_2, ..., s_n$ into integers $i_1, i_2, ..., i_n$ using the string dictionary is referred to as *dictionary encoding*. In most cases, the space required to store integers is less than that needed to store strings. In this paper, we attempt to improve existing string dictionaries by applying dictionary encoding to strings appearing in them. This section describes string dictionaries based on the *path-decomposed trie (PDT)* [19] and front coding. Moreover, we show how to apply dictionary encoding to these dictionaries.

In this section, we show examples for each string dictionary using a set of strings `ideal`, `ideas`, `ideology`, `tea`, `techie`, `technology`, `tie`, and `trie`.

3.1. Path-decomposed Trie (PDT)

PDT is a tree structure constructed by recursively decomposing a trie into node-to-leaf paths. Each node of a PDT corresponds to each path in the trie. We introduce a succinct PDT representation proposed by Grossi and Ottaviano [2].

The tree in Figure 1b is a PDT constructed by decomposing the trie in Figure 1a into node-to-leaf paths

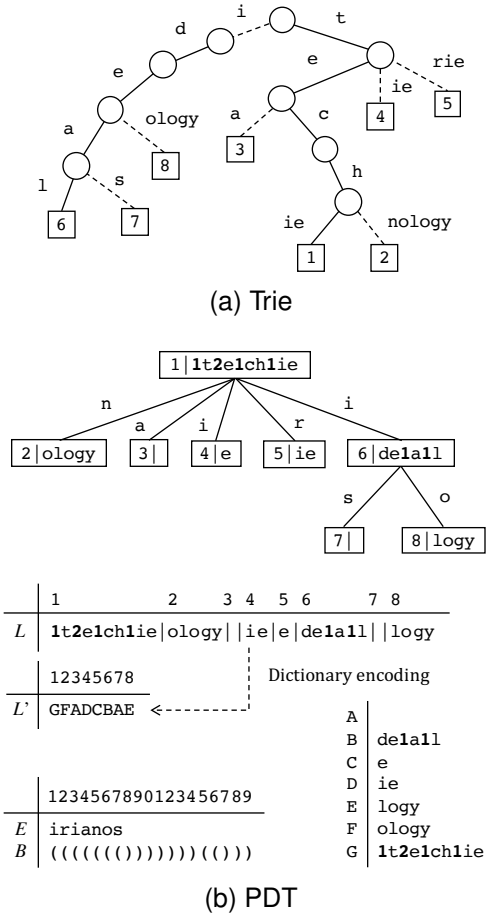

Figure 1. Examples of a trie and a PDT

connected by a solid line. The nodes and edges have string labels and branching characters, respectively. Such a PDT is constructed as follows. First, we choose a root-to-leaf path π in the trie. Second, we create a string by concatenating edge characters along path π, interleaved with special characters $\mathbf{1}, \mathbf{2}, \ldots$ that indicate how many subtries branch off that point along path π. Third, we associate the string with root node u_π of the PDT. The children of root node u_π are recursively defined as the root nodes of PDTs corresponding to each subtrie hanging off the path π.

Although a strategy of choosing a path is arbitrary, the example chooses the heavy path [19]. The heavy path always follows a heavy child, which is the one whose subtrie has the most leaves. This strategy is called *centroid path decomposition*. The height of the resulting tree is bounded by $O(\log |S|)$. Therefore, centroid path decomposition can reduce the number of node-to-node random accesses. In this paper, we use centroid path decomposition to implement PDT dictionaries.

For PDT representation, each node v is represented by three sequences: L_v stores the node label, E_v stores the branching characters from node v in reverse, and B_v is the DFUDS representation of node v. The node IDs are assigned in depth-first order. The PDT is represented by sequences L,

E, and B obtained by concatenating L_v, E_v, and B_v in order of node ID. To maintain the boundaries of the node labels, the Elias-Fano representation is used. We do not describe how to perform LOOKUP and ACCESS because they are complex. The interested reader can refer to the literature [2].

Compression Strategies. In [2], the sequence of node labels L was compressed using a variant of Re-Pair based on the approximate Re-Pair [20]. This Re-Pair can support scanning labels in constant time per character. In other words, the decoding and construction costs are reduced, but some space efficiency is sacrificed.

On the other hand, our strategy replaces node labels with integer IDs using dictionary encoding. As shown at the bottom of Figure 1b, the sequence of IDs L' is generated from L. Note that the node labels consist of characters drawn from $\Sigma' = \Sigma \cup \{\mathbf{1}, \mathbf{2}, \ldots, \sigma-\mathbf{1}\}$. In practice, $\Sigma' = [0, 511)$ because $\Sigma = [0, 256)$. We encode the characters into byte characters using VByte coding [21] to use dictionary encoding. To shorten the length of the byte sequence, we assign character code values from 0 in order of frequency of appearance. The Elias-Fano representation is not needed because L' is a fixed-length array.

3.2. Front Coding

Front coding [7] is a technique to compress lexicographically sorted strings. It encodes each string as a pair (ℓ, α), where ℓ is the length of the longest common prefix with its predecessor and α is the remaining suffix. This technique exploits the fact that real-world strings have similar prefixes, such as URLs and natural language words.

To allow for direct access, the strings are divided into buckets encoding b strings each. Each first string (referred to as the *header*) is explicitly stored. The remaining $b-1$ strings (referred to as *internal strings*) are differentially encoded, each with respect to its predecessor. The top of Figure 2 shows an example of front coding with $b = 4$. The headers are `ideal` and `techie`. The simplest implementation encodes the dictionary into a byte sequence and maintains the starting address of each header; this is called *plain front coding (PFC)* [1].

Dictionary Operations. LOOKUP(s) is performed in two steps. The first step consists of a binary search for string s in the set of headers to obtain the target bucket. The second step sequentially decodes the internal strings of the bucket while comparing each with s.

ACCESS(i) is performed in two steps as well. The first step determines the appropriate bucket ID with a simple division $\lceil i/b \rceil$. The second step sequentially decodes the internal strings of the bucket until it obtains the $((i-1) \bmod b)$-th internal string.

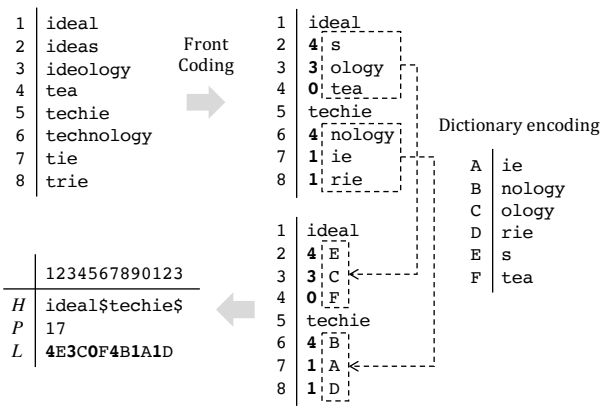

Figure 2. Examples of front coding with $b = 4$

Compression Strategies. In [1], the PFC was compressed by applying Re-Pair to the internal strings.[1] The authors used a public implementation of Re-Pair[2] based on the original [8]. Therefore, the compression rate was very high, but so was construction cost.

On the other hand, our strategy replaces internal strings with integer IDs using dictionary encoding. The bottom of Figure 2 shows an example. Our front coding dictionary consists of three arrays: H stores the header strings with a special terminator `$`, P stores the header initial addresses, and L interleaves the shared lengths and the IDs.

4. Auxiliary String Dictionaries

To avoid confusion, we refer to a string dictionary used for dictionary compression as an *auxiliary string dictionary*. As described in Section 3, it encodes strings appearing in the PDT and front coding dictionaries into integer IDs. Note that we do not restrict the integer range of the IDs. The auxiliary string dictionary supports the following operations:

- EXTRACT(i) returns the string with ID i.
- COMPARE(i, q) returns the result of comparison between strings EXTRACT(i) and q.

Although EXTRACT is the same as ACCESS, we redefine it to avoid notational confusion. COMPARE is called during LOOKUP. It is always supported when EXTRACT is supported; however, COMPARE can stop decoding when a mismatch occurs. The auxiliary string dictionary does not need string search operations such as LOOKUP because its role is to decode the stored strings.

In this section, we propose several auxiliary string dictionaries by considering the following:

- For each LOOKUP or ACCESS, EXTRACT and COMPARE are called multiple times; therefore, decoding speed is especially important.

1. Although [1] also proposed header compression with Hu-Tucker coding [6], we intend to compare Re-Pair compression with dictionary encoding; therefore, we do not evaluate header compression.

2. https://www.dcc.uchile.cl/~gnavarro/software/

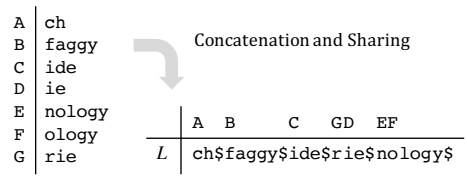

Figure 3. An example of TAIL

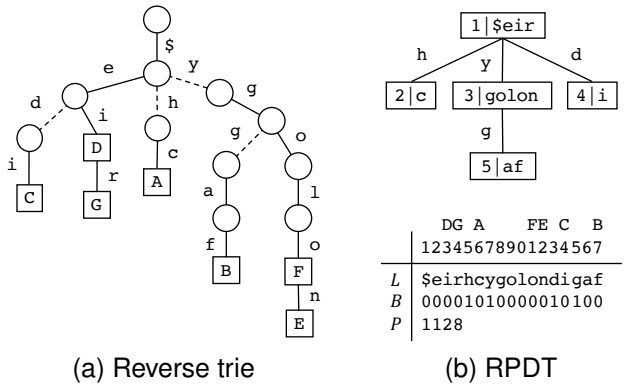

Figure 4. Examples of a reverse trie and RPDT

- As tries and front coding merge prefixes, the likelihood that the target strings for compression have many similar suffixes is high; therefore, we implement auxiliary string dictionaries by merging the suffixes.

We show examples for each auxiliary string dictionary using strings `ch`, `faggy`, `ide`, `ie`, `nology`, `ology`, and `rie` mapped to IDs `A`, `B`,..., and `G`, respectively.

4.1. Plain Concatenation and Sharing

The simplest data structure to implement an auxiliary string dictionary concatenates the strings with a terminator. Each starting address is obtained as an ID. When a string is included in the suffix of another, the suffix can be shared. Such an array is generally called *TAIL* [4], [22]. Figure 3 shows an example of TAIL. Strings `ie` and `ology` are shared by `rie` and `nology`, respectively. Compared to other auxiliary string dictionaries described below, its compression rate is low but its decoding speed is the fastest.

4.2. Reverse Trie

A *reverse trie* is constructed by merging suffixes rather than prefixes. The root corresponds to string terminations. The strings are decoded by traversing nodes to the root. That is to say, we can perform EXTRACT and COMPARE by maintaining the starting node IDs. Figure 4a shows an example of a reverse trie. For the purpose of illustration, the reverse trie has a super root with a special terminator `$`.

Several dictionary structures using reverse tries have been proposed [9], [10] and implemented as open-source software, such as the *ux-trie*[3] and *marisa-trie*[4] libraries. These structures implement reverse tries using the *double-array* [23] and *LOUDS (level-order unary degree sequence)* [11], [24] representations. The double array is a pointer-based data structure that can provide the fastest trie representation; however, its space efficiency is low. LOUDS is a succinct tree that can construct very small dictionaries; however, its node-to-node traversal is slow.

To solve the trade-off problem, the use of path decomposition is a workable alternative, but the existing representation [2] cannot immediately detect mismatches in COMPARE because the node label must be scanned from the head. Therefore, we propose a novel reverse trie representation with path decomposition, namely the *reverse path-decomposed trie (RPDT)*.

Reverse Path-decomposed Trie (RPDT). An implementation of RPDT is simpler than that in [2] because the reverse trie for auxiliary string dictionaries does not require finding children. Figure 4b shows an example of the RPDT constructed from the reverse trie of Figure 4a. The example RPDT is constructed by applying centroid path decomposition to the reverse trie and assigning node IDs in breadth-first order. The node labels do not contain the special characters $\mathbf{1}, \mathbf{2}, \ldots$ because it is not necessary to find children.

To represent the RPDT, we use three sequences: L stores strings obtained by concatenating pairs of branching characters and node labels in order of node ID, B is a bit sequence such that $B[i] = 1$ if $L[i]$ stores a branching character, and P stores addresses of L where each edge branches off in order of node ID. As P becomes a non-decreasing sequence, we can use the Elias-Fano representation. We perform EXTRACT on the sequences as in Algorithm 1.

Algorithm 1 EXTRACT(i) in RPDT

```
1: Initialize str to an empty string
2: while L[i] ≠ $ do
3:     Push back L[i] to str
4:     if B[i] = 1 then i ← P[RANK_1(B, i)]
5:     else i ← i − 1
6: end while
7: return str
```

Theoretical Analysis. We assume that the number of nodes in a reverse trie is n. L and B use $n\lceil\log\sigma\rceil$ bits and $n+o(n)$ bits, respectively. P uses $2m+m\lceil\log\frac{n}{m}\rceil+o(m)$ bits, where $|P| = m$. A succinct tree uses $2n + n\lceil\log\sigma\rceil + o(n)$ bits to represent the reverse trie. In roughly $2m + \lceil\log\frac{n}{m}\rceil < n$, the RPDT representation is smaller than the succinct tree representations. Note that m is the number of leaves in the reverse trie minus one. As shown in Figure 4, m becomes considerably smaller than n. Therefore, its space efficiency can outperform that of the succinct tree representation.

3. https://github.com/hillbig/ux-trie
4. https://github.com/s-yata/marisa-trie

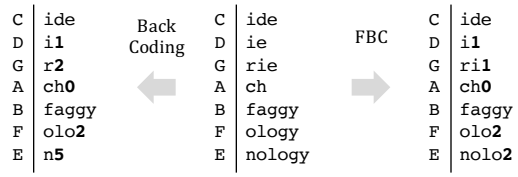

Figure 5. Examples of back coding with $b = 4$

Moreover, its cache efficiency is high because of centroid path decomposition.

4.3. Back Coding

We implement an auxiliary string dictionary by applying front coding to suffixes. In this paper, we refer to the technique as *back coding*. The left part of Figure 5 shows an example of back coding. In the same manner as the front coding dictionaries, we divide strings into buckets of size b and encode them from each header. Since auxiliary string dictionaries need fast operations, we implement the back coding dictionary using PFC.

Faster Decodable Implementation. To support faster decoding, we introduce an alternative implementation using the differences among headers instead of predecessors [25]. We refer to the technique as *fast back coding (FBC)*. The right part of Figure 5 shows an example of FBC. The technique does not need to decode internal strings other than the target string. In other words, the maximum number of memory copies for each EXTRACT or COMPARE is 2. However, some space efficiency is sacrificed because the number of shared characters decreases.

5. Experimental Evaluation

This section analyzes the practical performance of string dictionaries compressed by the proposed dictionary encoding on real-world datasets.

5.1. Settings

We carried out the experiments on Intel Xeon E5540 @2.53 GHz, with 32 GiB of RAM (L2 cache: 1 MiB; L3 cache: 8 MiB), running Ubuntu Server 16.04 LTS. The data structures were implemented in C++ and compiled using g++ (version 5.4.0) with optimization -O9. The runtimes were measured using *std::chrono::duration_cast*.

Data Structures. We applied the auxiliary string dictionaries described in Section 4 (i.e., TAIL, RPDT, back coding, and FBC) to the string dictionaries described in Section 3 (i.e., PDT and front coding). For back coding and FBC, we tested two bucket sizes of 4 and 8. The dictionaries are referred to as *BC4*, *BC8*, *FBC4*, and *FBC8*.

We also evaluated Re-Pair compression. Although some Re-Pair implementations are available, we used those used by each proponent described in Section 3. Note that we can choose other lightweight compression techniques, such as Huffman coding [26] and online grammar compression [27], [28]; however, Re-Pair is the most popular compression tool at present because its compression rate is very high and its decoding speed is fast.

To implement PDT and front coding dictionaries, we used the *path_decomposed_tries*[5] and *libCSD*[6], respectively. We set the bucket size of front coding to 8 based on past experiments [2]–[4]. To implement the basic tools described in Section 2, we used the *Succinct* library [29].

TABLE 1. INFORMATION CONCERNING DATASETS

	Size	Strings	Ave. length	Chars
GEONAMES	101.2	6,784,722	15.6	96
NWC	439.4	20,722,756	22.2	180
ENWIKI	227.2	11,519,354	20.7	199
INDOCHINA	612.9	7,414,866	86.7	98
UK	2,723.3	39,459,925	72.4	103
DBPEDIA	3,326.5	64,626,232	54.0	95

Datasets. We used the following real-world datasets:

- GEONAMES: Geographic names from the asciiname column of the GeoNames dump.[7]
- NWC: Japanese word $n$grams in the Nihongo Web Corpus 2010.[8]
- ENWIKI: All page titles from English Wikipedia in February 2015.[9]
- INDOCHINA: URLs of a 2004 crawl by the UbiCrawler [30] on the country domains of Indochina.[10]
- UK: URLs of a 2005 crawl by the UbiCrawler [30] on the .uk domain.[11]
- DBPEDIA: URIs extracted from the dataset generated by the DBpedia SPARQL Benchmark [31].[12]

Table 1 summarizes relevant statistics for each dataset, where *Size* is the total length of strings (i.e., the raw size) in MiB, *Strings* is the number of different strings, *Ave. length* is the average number of characters per string, and *Chars* is the number of different characters in the dataset.

Table 2 summarizes statistics of target strings of the auxiliary string dictionaries, where *Size* is the total length of strings in MiB, *Strings (before)* is the number of strings, *Strings (after)* is the number of strings obtained by removing duplication (i.e., the string set size), *Reduction* is its reduction rate in percentage, *Ave. length* is the average number of characters per string in the string set, and *Ave. calls* is the average number of COMPARE calls for each LOOKUP

5. https://github.com/ot/path_decomposed_tries
6. https://github.com/migumar2/libCSD
7. http://download.geonames.org/export/dump/allCountries.zip
8. http://dist.s-yata.jp/corpus/nwc2010/ngrams/word/over999/filelist
9. https://dumps.wikimedia.org/enwiki/
10. http://data.law.di.unimi.it/webdata/indochina-2004/indochina-2004.urls.gz
11. http://data.law.di.unimi.it/webdata/uk-2005/uk-2005.urls.gz
12. DS2 at https://exascale.info/projects/web-of-data-uri/

TABLE 2. INFORMATION ABOUT STRINGS

	Size	Strings (before)	Strings (after)	Reduction	Ave. length	Ave. calls
GEONAMES						
PDT	36.4	6,784,722	1,670,795	24.6	11.5	6.16
Front coding	34.6	5,936,631	1,354,818	22.8	10.9	3.50
NWC						
PDT	77.8	20,722,756	1,873,988	9.0	18.5	6.59
Front coding	69.6	18,132,411	202,058	1.1	8.5	3.50
ENWIKI						
PDT	90.3	11,519,354	3,762,531	32.7	14.4	6.26
Front coding	83.0	10,079,434	2,921,168	29.0	13.5	3.50
INDOCHINA						
PDT	134.6	7,414,866	1,299,775	17.5	44.7	6.61
Front coding	121.6	6,488,007	1,204,600	18.6	42.5	3.50
UK						
PDT	655.4	39,459,925	11,957,102	30.3	35.2	6.96
Front coding	591.7	34,527,434	10,756,301	31.2	34.2	3.50
DBPEDIA						
PDT	1423.3	64,626,232	12,199,056	18.9	30.1	7.41
Front coding	1274.8	56,547,953	9,969,214	17.6	30.3	3.50

(or EXTRACT calls for each ACCESS). From the table, we can considerably reduce the number of strings by removing duplication. In front coding, the average number of calls is essentially 3.5 because the number of internal strings for each bucket is 7.

5.2. Results

Tables 3 and 4 show the experimental results for the construction time in seconds (*Constr*), percentage of compression ratio between the data structure and the raw data size (*Cmpr*), and average running times of LOOKUP and ACCESS in microseconds (*Lookup* and *Access*). The top two results are highlighted. To measure the running times of LOOKUP, we chose one million random strings from each dataset. The running times of ACCESS were measured for one million IDs corresponding to the random strings. Each test was averaged over 10 runs.

Results for PDT (Table 3). All auxiliary string dictionaries yield short construction times. Compared to Re-Pair, our strategy provides up to 8.2x faster construction. The compression rate of Re-Pair is the lowest, except for INDOCHINA and UK. In INDOCHINA and UK, BC8 and RPDT construct slightly smaller dictionaries. For the runtimes of LOOKUP and ACCESS, TAIL is the fastest in all cases. Compared to Re-Pair, TAIL respectively provides up to 1.7x and 1.5x speed up on LOOKUP and ACCESS; however, its compression rate is the worst.

Overall, RPDT and FBC appear to be good data structures taking into account all aspects. RPDT altogether outperforms Re-Pair on INDOCHINA. FBC altogether outperforms Re-Pair on INDOCHINA and UK. In the other datasets, RPDT and FBC provide much shorter construction times than Re-Pair with the competitive compression rates and operation times. Back coding is compact but requires long operation times. If fast LOOKUP and ACCESS operations are needed, TAIL becomes a viable alternative.

Results for Front Coding (Table 4). Since this Re-Pair implementation is based on the original, its compression rates are better than those of the PDT results, but its construction and decoding costs are higher. On UK and DBPEDIA, Re-Pair respectively takes approximately 3.5 and 3 hours because the size of the target strings is very large. These times are impractical. On the other hand, all auxiliary string dictionaries provide practically short construction times as well as satisfactory PDT results. When comparing TAIL and Re-Pair, the differences are from 30.8x to 422.5x. In terms of compression rate, Re-Pair is the smallest in all cases. Compared to the second smallest dictionaries, Re-Pair is up to 4% smaller. However, its LOOKUP and ACCESS times are slower.

Overall, our dictionaries are competitive with the Re-Pair dictionaries except in terms of construction time. Further, our construction times are very short and practical. Back coding is not very slow compared to the PDT results because the number of calls is small. TAIL can provide very fast LOOKUP and ACCESS operations as well as PDT results.

6. Conclusion

In this paper, we have proposed several dictionary structures to compress string dictionaries. Moreover, we have evaluated the proposed structures by experiments using real-world datasets. The results have shown that our strategy can construct high-performance string dictionaries in a short time. In particular, RPDT and FBC are comprehensively superior to Re-Pair.

In this paper, we have addressed string dictionary compression; however, the proposed auxiliary string dictionaries

TABLE 3. RESULTS OF PDT

	GEONAMES				NWC				ENWIKI			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	26.6	**31.5**	2.17	2.11	58.0	**16.9**	2.73	2.72	62.1	**31.6**	2.59	2.50
TAIL	**4.0**	44.4	**1.51**	**1.54**	**11.0**	26.7	**2.10**	**2.04**	**8.3**	41.7	**1.72**	**1.76**
RPDT	5.8	**34.9**	**1.92**	**1.90**	16.0	23.0	2.68	**2.53**	11.8	**32.4**	2.40	**2.26**
BC4	5.4	38.5	3.51	3.66	14.2	22.9	5.14	5.11	10.6	36.0	4.56	4.66
BC8	**5.2**	36.6	3.91	4.05	14.8	**22.2**	5.58	5.80	**10.5**	33.8	5.13	5.33
FBC4	5.3	39.1	1.93	2.12	14.2	23.2	**2.48**	2.62	10.6	37.0	**2.29**	2.38
FBC8	5.3	38.0	2.11	2.11	**14.1**	22.8	2.72	2.90	10.6	35.9	2.49	2.62

	INDOCHINA				UK				DBPEDIA			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	55.0	11.8	3.70	3.61	437.2	17.5	4.00	3.98	798.9	**14.7**	2.30	**2.32**
TAIL	**7.8**	13.5	**2.19**	**2.40**	55.7	20.7	**2.59**	**2.94**	**102.8**	18.5	**1.59**	**1.82**
RPDT	9.3	**10.7**	3.32	3.38	58.6	**16.4**	4.16	3.91	158.3	15.8	2.40	2.52
BC4	**8.6**	10.9	6.41	6.91	63.4	16.9	7.02	7.60	124.8	15.9	6.26	6.68
BC8	**8.6**	**10.3**	7.50	8.05	**53.3**	**15.9**	7.93	8.45	**122.1**	**15.2**	6.69	7.16
FBC4	**8.6**	11.3	**2.74**	**3.03**	**53.5**	17.3	**3.24**	**3.55**	133.2	16.2	**2.17**	2.41
FBC8	**8.6**	11.2	3.20	3.44	**53.5**	16.9	3.66	3.91	132.8	15.9	2.56	2.78

TABLE 4. RESULTS OF FRONT CODING

	GEONAMES				NWC				ENWIKI			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	83.6	**38.4**	2.09	1.25	289.2	**25.4**	2.16	1.09	667.2	**36.5**	2.63	1.59
TAIL	**2.7**	48.3	**1.24**	**0.53**	**7.7**	28.3	**1.47**	**0.50**	**6.1**	45.3	**1.51**	**0.66**
RPDT	4.7	**41.6**	1.65	0.85	14.2	27.7	**1.64**	**0.58**	9.6	**38.7**	2.23	1.21
BC4	**4.3**	44.5	1.80	0.98	13.8	27.4	1.84	0.87	**8.6**	41.9	2.16	1.19
BC8	4.4	43.0	2.07	1.15	13.8	**27.3**	1.95	0.90	10.0	40.2	2.29	1.33
FBC4	4.5	45.0	**1.57**	**0.82**	**13.7**	27.4	1.68	0.70	**8.6**	42.7	**2.00**	**1.09**
FBC8	**4.3**	44.2	1.68	0.93	**13.7**	27.4	1.82	0.85	8.8	42.0	2.04	1.13

	INDOCHINA				UK				DBPEDIA			
	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access	Constr	Cmpr	Lookup	Access
Re-Pair	2042.8	**18.2**	3.88	2.25	11835.2	**22.3**	4.28	2.41	10774.6	**22.6**	4.27	2.85
TAIL	**4.3**	25.0	**2.03**	**0.71**	**28.0**	31.4	**2.32**	**0.77**	**55.3**	28.6	**1.53**	**0.74**
RPDT	6.7	22.3	2.60	1.17	50.0	27.1	3.64	1.73	123.1	27.0	**2.02**	**1.28**
BC4	**5.9**	22.5	2.71	1.32	**42.6**	27.7	3.18	1.59	94.0	27.0	2.17	1.40
BC8	**5.9**	**22.0**	3.05	1.65	43.4	**26.8**	3.60	1.97	92.2	**26.4**	2.41	1.61
FBC4	**5.9**	22.9	**2.45**	**1.14**	43.3	28.1	**3.02**	**1.47**	**91.6**	27.3	2.05	1.34
FBC8	6.1	22.7	2.73	1.30	43.4	27.7	3.04	1.51	96.6	27.0	2.16	1.45

can be used to compress other data structures partly containing strings. In the future, we will investigate such data structures and conduct application experiments.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 17J07555. We would like to thank Editage (www.editage.jp) for English language editing.

References

[1] M. A. Martínez-Prieto, N. Brisaboa, R. Cánovas, F. Claude, and G. Navarro, "Practical compressed string dictionaries," *Information Systems*, vol. 56, pp. 73–108, 2016.

[2] R. Grossi and G. Ottaviano, "Fast compressed tries through path decompositions," *ACM Journal of Experimental Algorithmics*, vol. 19, no. 1, p. Article 1.8, 2014.

[3] J. Arz and J. Fischer, "LZ-compressed string dictionaries," in *Proc. Data Compression Conference (DCC)*, 2014, pp. 322–331.

[4] S. Kanda, K. Morita, and M. Fuketa, "Compressed double-array tries for string dictionaries supporting fast lookup," *Knowledge and Information Systems*, vol. 51, no. 3, pp. 1023–1042, 2017.

[5] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[6] D. E. Knuth, *The art of computer programming, 3: sorting and searching*, 2nd ed. Redwood City, CA, USA: Addison Wesley, 1998.

[7] I. H. Witten, A. Moffat, and T. C. Bell, *Managing gigabytes: compressing and indexing documents and images*. San Francisco, CA, USA: Morgan Kaufmann, 1999.

[8] N. J. Larsson and A. Moffat, "Off-line dictionary-based compression," *Proc. the IEEE*, vol. 88, no. 11, pp. 1722–1732, 2000.

[9] J. Aoe, K. Morimoto, M. Shishibori, and K.-H. Park, "A trie compaction algorithm for a large set of keys," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 3, pp. 476–491, 1996.

[10] S. Yata, "Dictionary compression by nesting prefix/patricia tries (in Japanese)," in *Proc. 17th Annual Meeting of the Association for Natural Language*, 2011.

[11] G. Jacobson, "Space-efficient static trees and graphs," in *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1989, pp. 549–554.

[12] R. González, S. Grabowski, V. Mäkinen, and G. Navarro, "Practical implementation of rank and select queries," in *Poster Proc. 4th Workshop on Experimental and Efficient Algorithms (WEA)*, 2005, pp. 27–38.

[13] D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao, "Representing trees of higher degree," *Algorithmica*, vol. 43, no. 4, pp. 275–292, 2005.

[14] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane, "Succinct trees in practice," in *Proc. 11st Meeting on Algorithm Engineering and Experimentation (ALENEX)*, 2010, pp. 84–97.

[15] J. I. Munro and V. Raman, "Succinct representation of balanced parentheses and static trees," *SIAM Journal on Computing*, vol. 31, no. 3, pp. 762–776, 2001.

[16] P. Elias, "Efficient storage and retrieval by content and address of static files," *Journal of the ACM*, vol. 21, no. 2, pp. 246–260, 1974.

[17] R. M. Fano, *On the number of bits required to implement an associative memory*. Cambridge, MA: Memorandum 61, Computer Structures Group, MIT, 1971.

[18] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," *IEEE Transactions on Information Theory*, vol. 51, no. 7, pp. 2554–2576, 2005.

[19] P. Ferragina, R. Grossi, A. Gupta, R. Shah, and J. S. Vitter, "On searching compressed string collections cache-obliviously," in *Proc. 27th Symposium on Principles of Database Systems (PODS)*, 2008, pp. 181–190.

[20] F. Claude and G. Navarro, "Fast and compact web graph representations," *ACM Transactions on the Web*, vol. 4, no. 4, p. 16, 2010.

[21] H. E. Williams and J. Zobel, "Compressing integers for fast file access," *Computer Journal*, vol. 42, no. 3, pp. 193–201, 1999.

[22] S. Yata, M. Oono, K. Morita, T. Sumitomo, and J. Aoe, "Double-array compression by pruning twin leaves and unifying common suffixes," in *Proc. 1st International Conference on Computing & Informatics (ICOCI)*, 2006, pp. 1–4.

[23] J. Aoe, "An efficient digital search algorithm by using a double-array structure," *IEEE Transactions on Software Engineering*, vol. 15, no. 9, pp. 1066–1077, 1989.

[24] O. Delpratt, N. Rahman, and R. Raman, "Engineering the LOUDS succinct tree representation," in *Proc. 5th International Workshop on Experimental and Efficient Algorithms (WEA), LNCS 4007*, 2006, pp. 134–145.

[25] I. Müller, C. Ratsch, and F. Faerber, "Adaptive string dictionary compression in in-memory column-store database systems," in *Proc. 17th International Conference on Extending Database Technology (EDBT)*, 2014, pp. 283–294.

[26] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[27] S. Maruyama, H. Sakamoto, and M. Takeda, "An online algorithm for lightweight grammar-based compression," *Algorithms*, vol. 5, no. 2, pp. 214–235, 2012.

[28] S. Maruyama, Y. Tabei, H. Sakamoto, and K. Sadakane, "Fully-online grammar compression," in *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, 2013, pp. 218–229.

[29] R. Grossi and G. Ottaviano, "Design of practical succinct data structures for large data collections," in *International Symposium on Experimental Algorithms (SEA)*, 2013, pp. 5–17.

[30] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Software: Practice and Experience*, vol. 34, no. 8, pp. 711–726, 2004.

[31] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo, "DBpedia SPARQL benchmark–performance assessment with real queries on real data," in *Proc. 10th International Semantic Web Conference*, 2011, pp. 454–469.