Practical Implementation of Space-Efficient Dynamic Keyword Dictionaries

Shunsuke Kanda[1,2](0000-0002-5462-122X), Kazuhiro Morita[1], and Masao Fuketa[1]

[1] Graduate School of Advanced Technology and Science, Tokushima University, Minamijosanjima 2-1, Tokushima 770-8506, Japan

[2] Research Fellow of Japan Society for the Promotion of Science, Japan

shnsk.knd@gmail.com, {kam,fuketa}@is.tokushima-u.ac.jp

Abstract. A keyword dictionary is an associative array with string keys. Although it is a classical data structure, recent applications require the management of massive string data using the keyword dictionary in main memory. Therefore, its space-efficient implementation is very important. If limited to static applications, there are a number of very compact dictionary implementations; however, existing dynamic implementations consume much larger space than static ones. In this paper, we propose a new practical implementation of space-efficient dynamic keyword dictionaries. Our implementation uses path decomposition, which is proposed for constructing cache-friendly trie structures, for dynamic construction in compact space with a different approach. Using experiments on real-world datasets, we show that our implementation can construct keyword dictionaries in spaces up to 2.8x smaller than the most compact existing dynamic implementation.

Keywords: Keyword dictionaries · Compact data structures · Tries · Path decomposition

1 Introduction

In modern computer science, managing massive string data in main memory is a fundamental problem. Many researchers have investigated space-efficient data structures for string processing. In this paper, we focus on the practical implementation of *keyword dictionaries* that are an associative array with string keys. Although the keyword dictionary is a classical data structure used in natural language processing and information retrieval, many recent applications require space-efficient implementations to store large string datasets, as reported in [16]. For example, Mavlyutov et al. [17] considered URIs of 14 GB for RDF data management systems.

As for static keyword dictionaries, very compact implementations have been proposed recently. For example, Martínez-Prieto et al. [16] proposed and practically evaluated compact implementations using some techniques. Grossi and Ottaviano [8] proposed a cache-friendly compact implementation using an ordered labeled tree structure known as a *trie* [14]. For the implementations, Kanda et al. [13] empirically evaluated some compression strategies. Also, Kanda et al. [12] proposed a fast and compact implementation using an improved double-array trie. While those implementations can store large datasets in compact space, their applications are limited because key insertion and deletion are not supported.

As for dynamic keyword dictionaries, there are some space-efficient implementations such as the HAT-trie [1], adaptive radix tree (ART) [15], Judy [3], and Cedar [22]. While those implementations attempt to improve the space efficiency by reducing pointer overheads, they still consume much larger space than the static implementations. For example, to store a geographic name dataset, the HAT-trie uses space 7.2x larger than the static implementation by Grossi and Ottaviano [8], from the experimental results in this paper and [12]. On the other hand, a number of practical compact dynamic trie representations have been presented. Darragh et al. [5] proposed the *Bonsai* tree, which is a compact hash-based trie representation. Recently, Poyias and Raman [19] improved the Bonsai tree, namely, *m-Bonsai*. The m-Bonsai tree can represent a trie in asymptotically information-theoretically optimal space while supporting basic tree operations in constant expected time. Takagi et al. [20] also proposed an efficient data structure for online string processing. However, there has been no discussion or evaluation about keyword dictionary implementation. Therefore, we must address the engineering of more space-efficient implementations.

In this paper, we propose a new implementation of space-efficient dynamic keyword dictionaries. Our implementation is based on a trie formed by *path decomposition* [6], which is a trie transformation technique. The path decomposition was proposed for constructing cache-friendly trie structures and was utilized in static applications [8, 11]; however, we use it for dynamic dictionary construction with a different approach. We implement space-efficient dictionaries by applying the m-Bonsai representation to this approach. From experiments using read-world datasets considering various applications, we show that our implementation is much more compact than existing dynamic implementations.

2 Preliminaries

2.1 Basic Notations and Definitions

A sequence A with n entries, $A[0]A[1]\dots A[n-1]$, is denoted by $A[0,n)$. For a sequence $A[0,n)$, $|A|$ denotes the length n. A *keyword* is a byte character string that always has a special terminator drawn by $\$$, that is, $\$ \notin w[0,n-1)$ and $w[n-1] = \$$ for a keyword $w[0,n)$. The base of the logarithm is 2 throughout this paper.

2.2 Path Decomposition

A trie [14] is an ordered labeled tree to store a set of strings, and is constructed by merging common prefixes. Path decomposition [6] is a technique that transforms the trie as follows. It first chooses a root-to-leaf path in the original trie, and then associates the path with a root of the new trie. Children of the new root are recursively defined as the roots of new subtries corresponding to original subtries hanging off the path.

The existing purpose of path decomposition is to reduce the number of node-to-node traversals by lowering the height of the resulting tree. Although the height depends on a strategy of choosing a path, any strategy can guarantee that the height is not greater than that of the *Patricia tree* [18]; therefore, some improvement in cache efficiency can be expected for all strategies. The following fact is important for our data structure:

Fact 1. Each node of a path-decomposed trie corresponds to some node-to-leaf path of the original trie; therefore, the number of path-decomposed trie nodes is the same as that of registered keywords because the original trie has the same number of leaf nodes owing to the special keyword terminator.

2.3 m-Bonsai

The m-Bonsai tree [19] is a compact dynamic trie representation that defines nodes on hash table Q with m slots using open addressing. Let n denote the number of trie nodes. We refer to $\alpha = n/m$ $(0 \leq \alpha \leq 1)$ as a *load factor*. We assume that n and m are pre-given. As each node is located at some slot, we denote node IDs using slot addresses. That is, a node with ID v (or node v) is located on $Q[v]$. Defining a new child from node v with symbol c is implemented in three steps, assuming that the alphabet size of symbols is σ. The first step creates a hash key $\langle v, c\rangle$ of the child. The second step obtains an *initial address* of the child, $u = h(\langle v, c\rangle)$, where h is a hush function such that $h : \{0, \ldots, m \cdot \sigma - 1\} \rightarrow \{0, \ldots, m - 1\}$. The third step locates the child on $Q[u']$, where u' is the first empty slot address from the initial address u using linear probing. In other words, the new child ID is defined as u'.

If we simply store the hash key $\langle v, c\rangle$ in $Q[u']$ to check for membership, each slot uses large space of $\lceil \log(m \cdot \sigma)\rceil$ bits. To reduce this space to $\lceil \log \sigma\rceil$ bits, m-Bonsai uses the *quotienting* technique [14, Exercise 6.13], while additionally introducing *displacement array* D such that $D[u']$ stores the distance of u' and u, that is, the number of collisions. The displacement array D can be represented in compact space since the average value is small from [19, Proposition 1].

Practical Implementation. Although Poyias and Raman [19] proposed two types of displacement array representation, we adopt a simply modified version of the practical one. In our representation, we first try to store values of D in an array D_0 with each entry using Δ_0 bits. If $D[i] < 2^{\Delta_0} - 1$, we set $D_0[i] = D[i]$. Otherwise, we set $D_0[i] = 2^{\Delta_0} - 1$ and store $D[i]$ to an auxiliary associative array implemented using a standard data structure. The original representation uses a small hash table based on the original Bonsai method as an additional second data structure; however, our version omits the hash table because it is difficult to estimate the predefined length of the table when adopted for our implementation. From preliminary experiments, we obtained the best parameter $\varDelta_0 = 6$ for $\alpha = 0.8$. We provide the details and source code at `https://github.com/kamp78/bonsais`.

3 New Implementation

This section presents a new dynamic keyword dictionary implementation through path decomposition, namely, the *dynamic path-decomposed trie (DynPDT).*

3.1 Basic Idea: Incremental Path Decomposition

We present a basic idea called *incremental path decomposition.* This idea constructs a path-decomposed trie by incrementally defining nodes corresponding to each keyword in insertion order. We show the data structure of the path-decomposed trie while describing the insertion procedure for a keyword w as follows:

- If the dictionary is empty, a root labeled with w is defined. In this paper, we denote such a label on node v by L_v.
- If the dictionary is not empty, a keyword search is started from the root with two steps by setting v to the root ID. The first step compares w with L_v. If $w = L_v$, the procedure terminates because the keyword is already registered; otherwise, the second step finds a child with symbol $\langle i, w[i] \rangle$ such that $w[i] \neq L_v[i]$ and $w[0, i) = L_v[0, i)$. If not found, the keyword is inserted by adding a new edge labeled with the symbol $\langle i, w[i] \rangle$ and a new child labeled with the remaining suffix $w[i+1, |w|)$. If found, the procedure returns to the first step after updating v to the child ID and w to the remaining suffix.

That is to say, the path-decomposed trie has node labels representing some suffixes of keywords. Additionally, it has edge labels composed of a node label position and a byte character. A keyword search is also performed by that procedure. The feature of the incremental path decomposition is to locate nodes corresponding to early inserted keywords near the root. In other words, the search cost for such keywords is low. However, Sect. 4 evaluates the performance of DynPDT for random-ordered keywords without considering the feature.

3.2 Implementation with m-Bonsai

To obtain high space efficiency, DynPDT represents the path-decomposed trie using m-Bonsai; however, this representation has the following problem. As the edge label is a pair $\langle i, c \rangle$ composed of node label position i and byte character c, the edge labels are drawn from an alphabet of size $\sigma = 256 \cdot \varLambda$, where $\varLambda$ denotes the maximum length of the node labels. The m-Bonsai representation requires the fixing of the σ parameter to predefine the allocation size of each Q slot and the hash function, but $\varLambda$, or σ, is an unfixed parameter in dynamic applications registering unknown keywords.

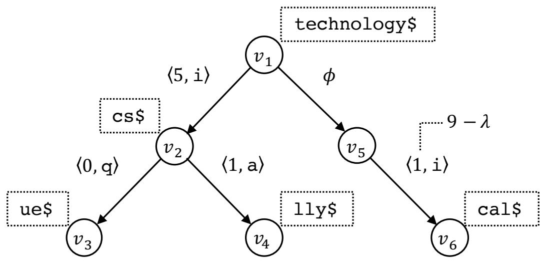


Fig. 1: Example of DynPDT when $\lambda = 8$.

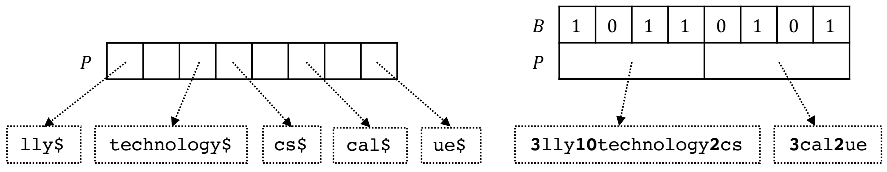
To solve this problem, we forcibly fix the alphabet size as $\sigma = 256 \cdot \lambda$ by introducing a new parameter λ. If position i on L_v is greater than or equal to λ, we create virtual nodes called *step nodes* with a special symbol ϕ by repeating to add child u from node v with symbol ϕ, to set v to u, and to decrement i by λ, until $i < \lambda$. This solution creates additional step nodes depending on λ. When λ is too small, many step nodes are created. When λ is too large, the space usage of Q becomes large because each slot uses $\lceil \log(256 \cdot \lambda) \rceil$ bits. Therefore, it is necessary to define a proper λ. Sect. 4.1 shows such parameters obtained from experiments using read-world datasets.

Examples. Fig. 1 shows an example of DynPDT constructed by inserting keywords `technology$`, `technics$`, `technique$`, `technically$`, and `technological$` in this order, setting $\lambda = 8$. The nodes are defined in order of $v_1, v_2, \ldots, v_6$. We show how to search `technically$` using the example. First, we set w to the query keyword and compare w with L_{v_1}. As $w[0,5) = L_{v_1}[0,5) =$ `techn`, we move to v_2 using symbol $\langle 5, w[5] \rangle = \langle 5, \mathtt{i} \rangle$ and update w to the remaining suffix `cally$`. Next, we compare w with L_{v_2}. As $w[0,1) = L_{v_2}[0,1) =$ `c`, we move to v_4 using symbol $\langle 1, w[1] \rangle = \langle 1, \mathtt{a} \rangle$ and update w to the remaining suffix `lly$`. Finally, we can see that the query keyword is registered from $w = L_{v_4}$.

We also show how to search `technological$`. In the same manner as above, we set w to the query keyword and compare w with L_{v_1}. The result is $w[0,9) = L_{v_1}[0,9) =$ `technolog`, but we cannot create symbol $\langle 9, \mathtt{i} \rangle$ because this symbol exceeds the alphabet size from $\lambda \leq 9$. Therefore, we move to step node v_5 using symbol ϕ. From $9 - \lambda < \lambda$, i.e., $1 < \lambda$, we can create symbol $\langle 1, \mathtt{i} \rangle$ and move to node v_6 using the symbol. Finally, we can see that the query keyword is registered because the remaining suffix `cal$` is the same as L_{v_6}.

Implementation Remarks. Arbitrary values associated with each keyword can be maintained using the space of each node label. Keyword deletion can be simply implemented by introducing flags for each node (i.e., for each keyword) in a manner similar to open address hashing.

To use the m-Bonsai representation, it is necessary to predefine the number of Q slots depending on the number of nodes and the load factor. In other words, it is necessary to estimate the number of nodes expected for a dataset. Fortunately,



(a) Plain management (b) Bitmap management in $\ell = 4$

Fig. 2: Examples of node label management for DynPDT in Fig. 1.

we can roughly estimate the number of nodes of DynPDT easier than a plain trie because this is the same as the number of keywords (from Fact 1) and some step nodes depending on a proper λ.

3.3 Node Label Management

The node labels are stored separately from the m-Bonsai structure (i.e., the hash table and the displacement array) because these labels are variable-length strings. The plainest implementation uses pointer array P of length m such that $P[i]$ stores a pointer to L_i. This implementation can perform to access and append a node label in constant time, but it uses large space with m pointers. We call this implementation *plain management*. Fig. 2a shows an example of plain management.

We present an alternative compact implementation that reduces the pointer overhead in a manner similar to *sparsetable* of Google Sparse Hash at `https://github.com/sparsehash/sparsehash`. This implementation divides node labels into *groups* of ℓ labels over the IDs. That is, the first group consists of $L_0 \ldots L_{\ell-1}$, the second group consists of $L_\ell \ldots L_{2\ell-1}$, and so on. Moreover, we introduce bitmap B such that $B[i] = 1$ if L_i exists. The implementation concatenates node labels L_i such that $B[i] = 1$ in each group, while keeping the ID order. The length of P becomes $\lceil m/\ell \rceil$ by maintaining pointers to the concatenated label strings for each group. We call this implementation *bitmap management*.

Using array P and bitmap B, accessing L_i is performed as follows. If $B[i] = 0$, L_i does not exist; otherwise, we obtain the target concatenated label string from $P[g]$, where $g = \lfloor i/\ell \rfloor$. We also obtain bit chunk $B_g = B[g \cdot \ell, (g+1) \cdot \ell)$ over the target group. Let j be the number of occurrences of 1s in $B_g[0, i \bmod \ell + 1)$. L_i is the j-th node label of the concatenated label string. As ℓ is constant, counting the bit occurrences in chunk B_g is supported in constant time using the *popcount* operation [7]. Therefore, the access time is the same as the time of scanning the concatenated label string until the j-th node label.

By simply concatenating node labels (e.g., the second group in Fig. 2a is `cal$ue$` in $\ell = 4$), the scan is performed by sequentially counting terminators in $O(\ell \cdot \Lambda)$ time, where Λ again denotes the maximum length of the node labels. We shorten the scan time using the *skipping* technique used in *array hashing* [2]. This technique puts its length in front of each node label using *VByte* encoding [21]. Note that we can omit the terminators of each node label. The skipping technique allows us to jump ahead to the start of the next node label; therefore, the scan is supported in $O(\ell)$ time. Fig. 2b shows an example of the bitmap management with the skipping technique.

Comparison of Space Usage. We compare plain and bitmap management in terms of space usage, assuming a 64-bit memory address architecture. In plain management, the pointer array P uses $64m$ bits. The space usage of storing node labels is $8N$ bits, where N denotes the total length, i.e., $N = \sum_{i<m} |L_i|$. In bitmap management, the pointer array P uses $64\lceil m/\ell \rceil$ bits, and the bitmap B uses m bits. The total length of the node labels using VByte encoding becomes equal to N if all node labels are shorter than 128 because such a code length is 1 byte. Fortunately, almost 100% of the node labels were shorter than 128 in all datasets in Sect. 4. Therefore, the VByte encoding does not become a significant overhead.

For simplicity, we assume that there are no overheads related to the VByte encoding and memory allocation. The overall space usage of plain management is $64m + 8N$ bits. The overall space usage of bitmap management is $64\lceil m/\ell \rceil + m + 8N$ bits. That is, in roughly $\ell > 1.02$, the space usage of bitmap management is smaller than that of plain management. Moreover, bitmap management works more efficiently when N is small because the pointer overhead of $64m$ bits becomes relatively large over $8N$ bits in plain management.

4 Experimental Evaluation

This section analyzes the practical performance of DynPDT. The source code of our implementation is available at `https://github.com/kamp78/dynpdt`.

4.1 Settings

We carried out experiments on an Intel Xeon E5540 @2.53 GHz with 32 GB of RAM (L2 cache: 1 MB, L3 cache: 8 MB), running Ubuntu Server 16.04 LTS. The data structures were implemented in C++ and compiled using g++ (version 5.4.0) with optimization -O9. We used `/proc/<PID>/statm` to measure the resident set size. We used `std::chrono::duration_cast` to measure the runtimes of operations.

Datasets. We selected six real-world datasets:

Geonames is composed of geographic names in the *asciiname* column of the GeoNames dump, available at `http://download.geonames.org/export/dump/`.

Wiki is page titles of English Wikipedia in February 2015, available at `https://dumps.wikimedia.org/enwiki/`.

UK is URLs obtained from a 2005 crawl of the `.uk` domain performed by UbiCrawler [4], available at `http://law.di.unimi.it/webdata/uk-2005/`.

Table 1: Information concerning datasets.

Dataset	Size	Keywords	Nodes	NPK	BPK	BPNL
Geonames	101.2	6,784,722	48,240,884	7.1	15.6	10.1
Wiki	227.2	11,519,354	110,962,030	9.6	20.7	12.6
UK	2,723.3	39,459,925	748,571,709	19.0	72.4	22.0
WebBase	6,782.1	118,142,155	1,426,314,849	12.1	60.2	15.1
LUBM	3,194.1	52,616,588	247,740,552	4.7	63.7	7.7
DNA	189.3	15,265,943	36,223,473	2.4	13.0	5.4

WebBase is URLs of a 2001 crawl performed by the WebBase crawler [10], available at `http://law.di.unimi.it/webdata/webbase-2001/`.

LUBM is URIs extracted from the dataset generated by the Lehigh University Benchmark [9] for 1,600 universities, from DS5 available at `https://exascale.info/projects/web-of-data-uri/`.

DNA is substrings of 12 characters found in the DNA dataset from Pizza&Chili corpus, available at `http://pizzachili.dcc.uchile.cl/texts/dna/`.

Table 1 summarizes relevant statistics for each dataset, where *Size* is the total length of keywords in MiB, *Keywords* is the number of distinct keywords, *Nodes* is the number of nodes in a plain trie, *NPK* is the average number of plain trie nodes per keyword, *BPK* is the average number of bytes per keyword, and *BPNL* is the average number of bytes per node label in DynPDT.

Dictionary Data Structures. We compared the performance of DynPDT with that of m-Bonsai. For DynPDT, we tested plain and bitmap management denoted by *Plain* and *Bitmap-ℓ*, respectively. For Bitmap-ℓ, we considered that ℓ is 8, 16, 32, and 64. We set the `int` data type to associated values in DynPDT. We tested m-Bonsai based on a plain trie without maintaining associated values. For both DynPDT and m-Bonsai, we implemented the auxiliary associative array using `std::map`.

We also compared some existing dynamic dictionary implementations. We selected five space-efficient ones as follows: *Sparsehash* is Google Sparse Hash that is an associative array with keys and values of arbitrary data types, *Judy* is a trie implementation developed at Hewlett-Packard Research Labs [3], *HAT-trie* is a keyword dictionary implementation with the combination of a trie and a cache-conscious hash table [1], *ART* is a trie implementation designed for efficient main-memory database systems [15], and *Cedar* is a state-of-the-art double-array prefix trie implementation [22]. In common with DynPDT, we set the `int` data type to associated values. For HAT-trie and ART, we used the implementations available at `https://github.com/dcjones/hat-trie` and `https://github.com/armon/libart`, respectively. As Cedar uses 32-bit integers to represent trie nodes, we could not run the test on WebBase.

Parameters. Both DynPDT and m-Bonsai have two parameters α and Δ_0. We set $\alpha = 0.8$ in common with previous settings [5, 19]. We set the number of Q

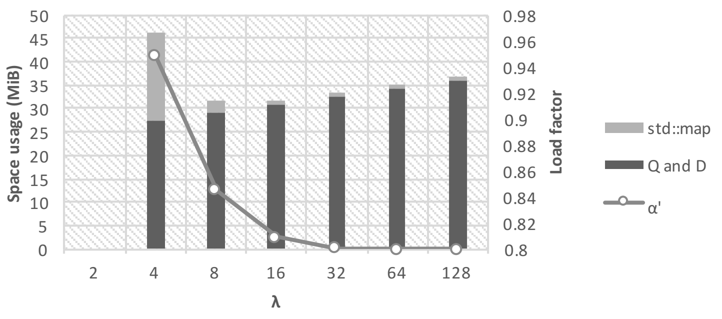


Fig. 3: Result of parameter test for λ on Wiki.

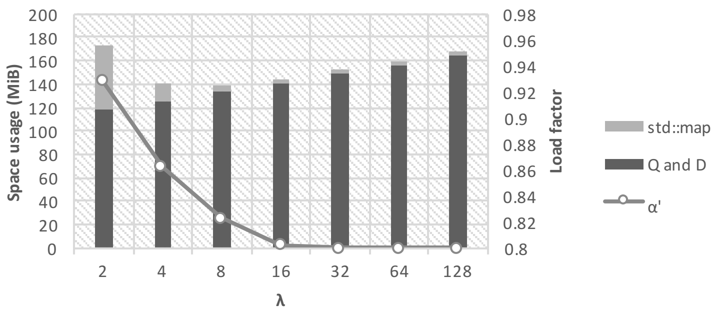


Fig. 4: Result of parameter test for λ on LUBM.

slots to that of keywords divided by 0.8 in DynPDT. Note that the resulting load factor α' in DynPDT is increased from α, depending on the number of step nodes. In m-Bonsai, we set the number of Q slots to that of plain trie nodes divided by 0.8. Note that the difference of the number of Q slots between DynPDT and m-Bonsai closes with decreasing NPK. We set $\varDelta_0 = 6$ from the preliminary experiments in Sect. 2.3.

DynPDT also has parameter λ, which involves α' and the space usage of hash table Q and the auxiliary associative array. When λ is large, the allocation size of Q becomes large. When λ is small, the number of step nodes, or α', is increased. The latter poses slow operations and a large auxiliary associative array because the average value of D is increased. To search a proper λ, we pretested $\lambda = 2^x$ in $2 \leq x \leq 7$ for each dataset. Figures 3 and 4 show the results on Wiki and LUBM, respectively. The figures show the sum space usage of Q, D, and the auxiliary `std::map`. The parameter α' is also shown. We could not construct the dictionary for $\lambda = 2$ on Wiki because α' became too large. From the results, α' closes 0.8, and the space usage is moderately increased from some λ. The

Table 2: Results of space usage in bytes per keyword.

Data Structure	Geonames	Wiki	UK	WebBase	LUBM	DNA
Plain	46.0	46.6	54.4	47.5	45.0	44.8
Bitmap-8	18.7	21.2	31.3	24.0	15.5	13.0
Bitmap-16	16.8	18.8	28.2	21.0	13.8	11.0
Bitmap-32	15.0	17.4	27.1	19.8	12.1	9.8
Bitmap-64	14.5	16.9	26.4	19.2	11.5	9.0
m-Bonsai	17.7	23.6	46.1	29.3	11.4	5.9
Sparsehash	62.3	71.1	131.0	119.0	122.0	43.4
Judy	47.6	50.5	60.3	53.5	33.9	24.3
HAT-trie	35.4	40.2	82.3	68.9	64.7	28.9
ART	87.1	93.1	140.9	126.9	118.9	71.1
Cedar	30.5	41.1	58.4	–	29.7	22.1

same tendency appeared for the other datasets.[1] In the experiments, we chose the smallest λ such that $\alpha' \leq 0.81$ for each dataset. We set λ to 16, 16, 64, 32, 16, and 4 on Geonames, Wiki, UK, WebBase, LUBM, and DNA, respectively.

4.2 Results

We constructed the dictionaries by inserting keywords in random order. We measured the resident set size required for the construction. We measured the insertion and search runtimes without I/O overheads. The insertion time was averaged on 3 runs. To measure the search time, we chose 1 million random keywords from each dataset. The search time was averaged on 10 runs.

Space Usage. Table 2 shows the results. It is obvious that bitmap management can reduce the pointer overhead of plain management. Bitmap-64 is up to 5x smaller than Plain on DNA. When BPNL is small, the compression rate is high based on the comparison analysis in Sect. 3.3. Compared with m-Bonsai, Bitmap-64 is 1.2–1.7x smaller except for LUBM and DNA. Bitmap-64 is 1.5x larger on DNA because the difference of the Q lengths is small from NPK. Note that m-Bonsai did not maintain associated values of the `int` type. If m-Bonsai maintained those values ideally without any overhead, 4 bytes (i.e., `sizeof(int)`) are added per keyword. That is, Bitmap-64 becomes smaller than m-Bonsai on all the datasets. In the existing dictionaries, Cedar is basically small although 32-bit integers are used to represent node pointers. In Wiki and WebBase, HAT-trie and Judy are the smallest. Compared with the smallest existing dictionaries, Bitmap-64 is 2.1–2.8x smaller. On UK and WebBase whose BPNL is large, Plain is also smaller than the existing dictionaries because the pointer overhead is relatively small over the overall space usage.

[1] All the results are provided at `https://github.com/kamp78/dynpdt/wiki`.

Table 3: Results of insertion time in microseconds per keyword.

Data Structure	Geonames	Wiki	UK	WebBase	LUBM	DNA
Plain	1.00	1.14	1.65	2.37	1.65	1.35
Bitmap-8	1.25	1.38	1.99	2.64	1.91	1.58
Bitmap-16	1.37	1.57	2.29	2.93	1.99	1.66
Bitmap-32	1.69	1.93	2.91	3.47	2.29	1.91
Bitmap-64	2.13	2.65	4.12	4.60	2.87	2.30
m-Bonsai	1.62	2.22	7.13	7.69	4.80	1.03
Sparsehash	4.31	5.15	9.13	11.32	8.72	1.99
Judy	0.93	1.06	2.15	2.94	1.53	0.90
HAT-trie	0.96	1.13	1.63	1.75	2.58	0.84
ART	1.07	1.19	2.20	2.98	1.44	0.87
Cedar	1.05	1.07	2.56	–	2.50	0.90

Table 4: Results of search time in microseconds per keyword.

Data Structure	Geonames	Wiki	UK	WebBase	LUBM	DNA
Plain	1.01	1.13	1.53	2.20	1.12	1.08
Bitmap-8	1.22	1.38	2.15	2.40	1.26	1.26
Bitmap-16	1.38	1.61	2.47	2.74	1.43	1.38
Bitmap-32	1.71	2.06	3.25	3.72	1.61	1.83
Bitmap-64	2.31	3.01	4.88	5.29	2.16	2.18
m-Bonsai	1.47	2.06	6.69	8.30	3.08	0.86
Sparsehash	0.34	0.44	0.67	0.80	0.67	0.29
Judy	0.70	0.88	2.02	2.42	0.79	0.44
HAT-trie	0.31	0.35	0.61	0.80	0.51	0.22
ART	0.81	1.03	1.84	2.68	0.67	0.63
Cedar	0.42	0.69	2.51	–	0.69	0.22

Insertion Time. Table 3 shows the results. In DynPDT, Plain is the fastest and Bitmap-64 is the slowest essentially, but Bitmap-8 is not much slower than Plain. We compare Bitmap-8 for the following. Compared with m-Bonsai, Bitmap-8 is faster except for DNA. In particular, the difference is very large on datasets whose BPK is large owing to the path decomposition. Bitmap-8 is 3.6x, 2.9x and 2.5x faster than m-Bonsai on UK, WebBase, and LUBM, respectively. Bitmap-8 is 1.5x slower than m-Bonsai on DNA. Compared with the existing dictionaries, Bitmap-8 is not the fastest but is very competitive on UK, WebBase, and LUBM; however, Bitmap-8 is the slowest on the other datasets except for Sparsehash.

Search Time. Table 4 shows the results. Like the insertion time results, Plain is the fastest in DynPDT, and Bitmap-8 is faster than m-Bonsai except for DNA. On the other hand, DynPDT is basically slower compared with the existing dictionaries. Bitmap-8 is up to 5.7x slower than the fastest HAT-trie. On UK and WebBase, Bitmap-8 is close to Judy, ART, and Cedar.

5 Conclusion and Future Work

In this paper, we presented DynPDT, which is a new practical implementation of space-efficient dynamic keyword dictionaries through incremental path decomposition. Our experimental results showed that DynPDT uses much smaller space than existing keyword dictionary implementations, but its time performance is not very high. The main cause is that the node-to-node traversal using m-Bonsai is slow compared with pointer-based representations. Another disadvantage is that the hash table cannot be easily resized owing to open addressing. Therefore, we will improve upon those disadvantages by engineering an alternative trie representation in the future. Moreover, we will address more compression because DynPDT still consumes a larger space compared with existing static compact implementations. For example, the static data structure by Grossi and Ottaviano [8] can implement a dictionary in space 2.9x smaller than DynPDT for the geographic name dataset.

We have discussed dictionary structures supporting only search operations for a given keyword as a basic associative array. On the other hand, dictionary structures supporting invertible mapping between strings and unique IDs, known as *string dictionaries*, are also important in many applications [16]. In principle, DynPDT can provide such invertible mapping because m-Bonsai supports leaf-to-root traversal operations. Therefore, we will propose and evaluate DynPDT structures adapting to string dictionaries.

Acknowledgments

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. Askitis, N., Sinha, R.: Engineering scalable, cache and space efficient tries for strings. The VLDB Journal 19(5), 633–660 (2010)
2. Askitis, N., Zobel, J.: Cache-conscious collision resolution in string hash tables. In: Proc. 12th International Symposium on String Processing and Information Retrieval (SPIRE). pp. 91–102 (2005)
3. Baskins, D.: Judy IV Shop Manual (2002)
4. Boldi, P., Codenotti, B., Santini, M., Vigna, S.: Ubicrawler: A scalable fully distributed web crawler. Software: Practice and Experience 34(8), 711–726 (2004)
5. Darragh, J.J., Cleary, J.G., Witten, I.H.: Bonsai: a compact representation of trees. Software: Practice and Experience 23(3), 277–291 (1993)
6. Ferragina, P., Grossi, R., Gupta, A., Shah, R., Vitter, J.S.: On searching compressed string collections cache-obliviously. In: Proc. 27th Symposium on Principles of Database Systems (PODS). pp. 181–190 (2008)
7. González, R., Grabowski, S., Mäkinen, V., Navarro, G.: Practical implementation of rank and select queries. In: Poster Proc. 4th Workshop on Experimental and Efficient Algorithms (WEA). pp. 27–38 (2005)

8. Grossi, R., Ottaviano, G.: Fast compressed tries through path decompositions. ACM Journal of Experimental Algorithmics 19(1), Article 1.8 (2014)
9. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web 3(2), 158–182 (2005)
10. Hirai, J., Raghavan, S., Garcia-Molina, H., Paepcke, A.: WebBase: A repository of web pages. Computer Networks 33(1), 277–293 (2000)
11. Hsu, B.J.P., Ottaviano, G.: Space-efficient data structures for top-k completion. In: Proc. 22nd International Conference on World Wide Web (WWW). pp. 583–594 (2013)
12. Kanda, S., Morita, K., Fuketa, M.: Compressed double-array tries for string dictionaries supporting fast lookup. Knowledge and Information Systems 51(3), 1023–1042 (2017)
13. Kanda, S., Morita, K., Fuketa, M.: Practical string dictionary compression using string dictionary encoding. In: Proc. 3rd International Conference on Big Data Innovations and Applications (Innovate-Data). pp. 1–8 (2017)
14. Knuth, D.E.: The art of computer programming, 3: sorting and searching. Addison Wesley, Redwood City, CA, USA, 2nd edn. (1998)
15. Leis, V., Kemper, A., Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: Proc. IEEE 29th International Conference on Data Engineering (ICDE). pp. 38–49 (2013)
16. Martínez-Prieto, M.A., Brisaboa, N., Cánovas, R., Claude, F., Navarro, G.: Practical compressed string dictionaries. Information Systems 56, 73–108 (2016)
17. Mavlyutov, R., Wylot, M., Cudre-Mauroux, P.: A comparison of data structures to manage URIs on the Web of data. In: Proc. 12th European Semantic Web Conference (ESWC). pp. 137–151 (2015)
18. Morrison, D.R.: PATRICIA: practical algorithm to retrieve information coded in alphanumeric. Journal of the ACM 15(4), 514–534 (1968)
19. Poyias, A., Raman, R.: Improved practical compact dynamic tries. In: Proc. 22nd International Symposium on String Processing and Information Retrieval (SPIRE). pp. 324–336 (2015)
20. Takagi, T., Inenaga, S., Sadakane, K., Arimura, H.: Packed compact tries: A fast and efficient data structure for online string processing. In: Proc. 27th International Workshop on Combinatorial Algorithms (IWOCA). pp. 213–225 (2016)
21. Williams, H.E., Zobel, J.: Compressing integers for fast file access. Computer Journal 42(3), 193–201 (1999)
22. Yoshinaga, N., Kitsuregawa, M.: A self-adaptive classifier for efficient text-stream processing. In: Proc. 24th International Conference on Computational Linguistics (COLING). pp. 1091–1102 (2014)