Improved Practical Compact Dynamic Tries

Andreas Poyias(✉) and Rajeev Raman

University of Leicester, Leicester, UK
{ap480,r.raman}@le.ac.uk

Abstract. We consider the problem of implementing a *dynamic trie* with an emphasis on good practical performance. For a trie with n nodes with an alphabet of size σ, the information-theoretic lower bound is $n \log \sigma + O(n)$ bits. The Bonsai data structure [1] supports trie operations in $O(1)$ expected time (based on assumptions about the behaviour of hash functions). While its practical speed performance is excellent, its space usage of $(1+\epsilon)n(\log \sigma + O(\log \log n))$ bits, where ϵ is any constant > 0, is not asymptotically optimal. We propose an alternative, *m-Bonsai*, that uses $(1+\epsilon)n(\log \sigma + O(1))$ bits in expectation, and supports operations in $O(1)$ expected time (again based on assumptions about the behaviour of hash functions). We give a heuristic implementation of m-Bonsai which uses considerably less memory and is slightly faster than the original Bonsai.

1 Introduction

In this paper, we consider *practical* approaches to the problem of implementing a *dynamic trie* in a highly space-efficient manner. A dynamic trie (also known as a dynamic *cardinal* tree [2]) is a rooted tree, where each child of a node is labelled with a distinct symbol from an alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$. We consider dynamic tries that support the following operations:

create(): Create a new empty tree.
getRoot(): return the root of the current tree.
getChild(v, i): return child node of node v with symbol i, if any (and return -1 if no such child exists).
addChild(v, i): add new child with symbol i and return the newly created node.
getParent(v): return the parent of node v.

We do not discuss deletions explicitly, but do indicate what is possible with regards to deletions. A trie is a classic data structure (the name dates back to 1959) and has numerous applications in string processing. A naive implementation of tries uses pointers. Using this approach, each node in an n-node binary trie uses 3 pointers for the navigational operations. A popular alternative for larger alphabets is the *ternary search tree (TST)* [3], which uses 4 pointers (3 plus a parent pointer), in addition to the space for a symbol. Other approaches include the *double-array trie (DAT)*, which uses a minimum of two integers per node, each of magnitude $O(n)$. Since a pointer must asymptotically use $\Omega(\log n)$ bits

C. Iliopoulos et al. (Eds.): SPIRE 2015, LNCS 9309, pp. 324–336, 2015.
DOI: 10.1007/978-3-319-23826-5_31

of memory, the asymptotic space bound of TST (or DAT) is $O(n(\log n + \log \sigma))$ bits. However, the information-theoretic space lower bound of $n \log \sigma + O(n)$ bits (see e.g. [2]) corresponds to one symbol and $O(1)$ *bits* per node. Clearly, if σ is small, both TST and DAT are asymptotically non-optimal. In practice, $\log \sigma$ is a few bits, or one or two bytes at most. An overhead of 4 pointers, or $32n$ bytes on today's machines, makes it impossible to hold tries with even moderately many nodes in main memory. Although tries can be *path-compressed* by deleting nodes with just one child and storing paths explicitly, this approach (or even more elaborate ones like [4]) cannot guarantee a small space bound.

Motivated by this, a number of space-efficient solutions were proposed [2,5–8], which represent *static* tries in information-theoretically optimal space, and support a wide range of operations. A number of asymptotic worst-case results were given in [9–12]. As our focus is on practical performance, we do not discuss all previous results in detail and refer the reader to e.g. [11] for a comparison. For completeness, we give a summary of some the results of [11,12]. The first uses almost optimal $2n + n \log \sigma + o(n \log \sigma)$ bits, and supports trie operations in $O(1)$ time if $\sigma = \text{polylog}(n)$ and in $O(\log \sigma / \log \log \sigma)$ time otherwise. The second [12, Theorem2] uses $O(n \log \sigma)$ bits and supports individual dynamic trie operations in $O(\log \log n)$ amortized expected time, although finding the longest prefix of a string in the trie can be done in $O(\log k / \log_\sigma n + \log \log n)$ expected time. Neither of these has been fully implemented, although a preliminary attempt (without memory usage measurements) was presented in [13]. Finally, we mention the *wavelet trie* [14] which is a data structure for a sequence of strings, and in principle can replace tries in many applications. Although in theory it is dynamic, we are not aware of any implementation of a dynamic wavelet trie.

Predating most of this work, Darragh et al. [1] proposed the *Bonsai* data structure, which uses a different approach to support the above dynamic trie operations in $O(1)$ expected time (based on assumptions about the behaviour of hash functions). While its practical speed performance is excellent, we note here that the asymptotic space usage of the Bonsai data structure is $(1 + \epsilon)n(\log \sigma + O(\log \log n))$ bits, where ϵ is any constant > 0, which is not asymptotically optimal due to the addition of $O(\log \log n)$ term. The additive $O(n \log \log n)$ bits term can be significant in many practical applications where the alphabet size is relatively small, including one involving mining frequent patterns that we are considering. The Bonsai data structure also has a certain chance of failure: if it fails then the data structure may need to be rebuilt, and its not clear how to do this without affecting the space and time complexities.

In this paper, we introduce m-Bonsai[1], a variant of Bonsai. Again, based upon the same assumptions about the behaviour of [1], our variant uses $(1+\epsilon)n(\log \sigma + O(1))$ bits of memory in expectation, where ϵ is any constant > 0, which is asymptotically optimal, and operations take $O(1)$ expected time. We give two practical variants of m-Bonsai: m-Bonsai (γ) and m-Bonsai (recursive). Our implementations and experimental evaluations show that m-Bonsai (recursive) is consistently a bit faster than the original Bonsai and significantly more

[1] This could be read as mame-bonsai, a kind of small bonsai plant, or mini-bonsai.

space-efficient than the original, while m-Bonsai (γ) is even more space efficient but rather slower. Of course, all Bonsai variants use at least 20 times less space than TSTs for small alphabets and compare well in terms of speed with TSTs. We also note that our experiments show that the hash functions used in Bonsai appear to behave in line with the assumptions about their behaviour. Finally, for both Bonsai and m-Bonsai, we believe it is relatively easy to remove the $(1+\epsilon)$ multiplicative factor from the $n \log \sigma$ term, but since this is not our primary interest is robust practical performance, we have not pursued this avenue.

The rest of this paper is organized as follows. In Section 2, we talk about the asymptotics of Bonsai [1] and give a practical analysis. Section 3 summarizes m-Bonsai approach which is followed by Section 4 the experimental evaluation.

2 Preliminaries

Bit-vectors. Given a bit string $x_1, \ldots, x_n$, we define the following operations:

select$_1(x, i)$**:** Given an index i, return the location of i_{th} `1` in x.
rank$_1(x, i)$**:** Return the number of `1`s upto and including location i in x.

Lemma 1 (Pătraşcu [15]). *A bit string can be represented in* $n+O(n/(\log n)^2)$ *bits such that* $select_1$ *and* $rank_1$ *can be supported in* $O(1)$ *time.*

Asymptotics of Bonsai. We now sketch the Bonsai data structure, focussing on asymptotics. It uses an array Q of size M to store a tree with $n = \lfloor \alpha M \rfloor$ nodes for some $0 < \alpha < 1$ (we assume that n and M are known at the start of the algorithm). We refer to α as the *load factor*. The Bonsai data structure refers to nodes via a unique *node ID*, which is a pair $\langle i, j \rangle$ where $0 \leq i < M$ and $0 \leq j < \lambda$, where λ is an integer parameter that we discuss in greater detail below. If we wish to add a child w with symbol $c \in \Sigma$ to a node v with node ID $\langle i, j \rangle$, then w's node ID is obtained as follows: We create the *key* of w using the node ID of v, which is a triple $\langle i, j, c \rangle$. We evaluate a hash function $h : \{0, \ldots, M \cdot \lambda \cdot \sigma - 1\} \mapsto \{0, \ldots, M-1\}$ on the key of w. If $i' = h(\langle i, j, c \rangle)$, the node ID of w is $\langle i', j' \rangle$ where $j' \geq 0$ is the lowest integer such that there is no existing node with a node ID $\langle i', j' \rangle$; i' is called the *initial address* of w.

In order to check if a node has a child with symbol c, keys are stored in Q using open addressing and linear probing[2]. The space usage of Q is kept low by the use of *quotienting* [16]. The hash function has the form $h(x) = (ax \bmod p) \bmod M$ for some prime $p > M \cdot \lambda \cdot \sigma$ and multiplier a, $1 \leq a \leq p-1$. Q only contains the *quotient* value $q(x) = \lfloor (ax \bmod p)/M \rfloor$ corresponding to x. Given $h(x)$ and $q(x)$, it is possible to reconstruct x to check for membership. While checking for membership for x, one needs to know $h(y)$ for all keys y encountered during the search, which is not obvious since keys may not be stored at their initial address due to collisions. The Bonsai approach is to keep all keys with the same initial address in consecutive locations in Q (this means that keys may be moved

[2] A variant, *bidirectional* probing, is used in [1], but we simplify this to linear probing

after they have been inserted) and to use two bit-vectors of size M bits to effect the mapping from a node's initial address to the position in Q containing its quotient, for details see [1]. Clearly, being able to search for, and insert keys allows us to support *getChild* and *addChild*; for *getParent*(v) note that the key of v encodes the node ID of its parent.

Asymptotic space usage. In addition to the two bit-vectors of M bits each, the main space usage of the Bonsai structure is Q. Since a prime p can be found that is $< 2 \cdot M \cdot \lambda \cdot \sigma$, it follows that the values in Q are at most $\lceil \log_2(2\sigma\lambda + 1) \rceil$ bits. The space usage of Bonsai is therefore $M(\log \sigma + \log \lambda + O(1))$ bits.

Since the choice of the prime p depends on λ, λ must be fixed in advance. However, if more than λ keys are hashed to any value in $\{0, \ldots, M-1\}$, the algorithm is unable to continue[3]. Thus, λ should be chosen large enough to reduce the probability of more than λ keys hashing to the same initial address to acceptable levels. In [1] the authors, assuming the hash function has full independence and is uniformly random, argue that choosing $\lambda = O(\log M / \log \log M)$ reduces the probability of error to at most M^{-c} for any constant c (choosing asymptotically smaller λ causes the algorithm almost certainly to fail). As the optimal space usage for an n-node trie on an alphabet of size σ is $O(n \log \sigma)$ bits, the additive term of $O(M \log \lambda) = O(n \log \log n)$ makes the space usage of Bonsai non-optimal for small alphabets.

However, even this choice of λ is not well-justified from a formal perspective, since the hash function used is quite weak—it is only 2-universal [17]. For 2-universal hash functions, the maximum number of collisions can only be bounded to $O(\sqrt{n})$ [18] (note that it is not obvious how to use more robust hash functions, since quotienting may not be possible). Choosing λ to be this large would make the space usage of the Bonsai structure asymptotically uninteresting.

Practical Analysis. In practice, we note that choosing $\lambda = 32$, and assuming complete independence in the hash function, the error probability for M up to 2^{64} is about 10^{-19} for $\alpha = 0.8$, using the formula in [1]. Choosing $\lambda = 16$ as suggested in [1] suggests a high failure probability for $M = 2^{56}$ and $\alpha = 0.8$. Also, in practice, the prime p is not significantly larger than $M\lambda\sigma$ [19, Lemma5.1]. The space usage of the Bonsai structure therefore is taken to be $(\lceil \log \sigma \rceil + 7)M$ bits for the tree sizes under consideration in this paper.

3 m-Bonsai

3.1 Overview

In our approach, each node again has an associated key that needs to be searched for in a hash table, again implemented using open addressing with linear probing and quotienting. However, the ID of a node x in our case is a

[3] Particularly for non-constant alphabets, it is not clear how to rebuild the data structure without an asymptotic penalty.

number from $\{0, \ldots, M-1\}$ that refers to the index in Q that contains the quotient corresponding to x. If a node with ID i has a child with symbol $c \in \Sigma$, the child's key, which is $\langle i, c \rangle$, is hashed using a multiplicative hash function $h : \{0, \ldots, M \cdot \sigma - 1\} \mapsto \{0, \ldots, M-1\}$, and an initial address i' is computed. If i'' is the smallest index $\geq i'$ such that $Q[i'']$ is vacant, then we store $q(x)$ in $Q[i'']$. Observe that $q(x) \leq \lceil 2\sigma \rceil$, so Q takes $M \log \sigma + O(M)$ bits. In addition, we have a *displacement* array D, and set $D[i''] = i'' - i'$. From the pair $Q[l]$ and $D[l]$, we can obtain both the initial hash address of the key stored there and its quotient, and thus reconstruct the key. The key idea is that in expectation, the average value in D is small:

Proposition 1. *Assuming h is fully independent and uniformly random, the expected value of $\sum_{i=0}^{M-1} D[i]$ after all $n = \alpha M$ nodes have been inserted is $\approx M \cdot \frac{\alpha^2}{2(1-\alpha)}$.*

Proof. The average number of probes, over all keys in the table, made in a successful search is $\approx \frac{1}{2}(1 + \frac{1}{1-\alpha})$ [16]. Multiplying this by $n = \alpha M$ gives the total average number of probes. However, the number of probes for a key is one more than its displacement value. Subtracting αM from the above and simplifying gives the result.

Thus, encoding D using variable-length encoding could be very beneficial. For example, coding D in unary would take $M + \sum_{i=1}^{M} D[i]$ bits; by Proposition 1, and plugging in $\alpha = 0.8$, the expected space usage of D, encoded in unary, should be about $2.6M$ bits, which is smaller than the overhead of $7M$ bits of the original Bonsai. As shown in Table 1, predictions made using Proposition 1 are generally quite accurate. Table 1 also suggests that encoding each $D[i]$ using the γ-code, we would come down to about $2.1M$ bits for the D, for $\alpha = 0.8$.

Table 1. Average number of bits per entry needed to encode the displacement array using the unary, γ and Golomb encodings. For the unary encoding, Proposition 1 predicts $1.81\dot{6}$, 2.6 and 5.05 bits per value. For file details see Table 2.

	unary			γ			Golomb		
Load Factor	0.7	0.8	0.9	0.7	0.8	0.9	0.7	0.8	0.9
Pumsb	1.81	2.58	5.05	1.74	2.11	2.65	2.32	2.69	3.64
Accidents	1.81	2.58	5.06	1.74	2.11	2.69	2.33	2.69	3.91
Webdocs	1.82	2.61	5.05	1.75	2.11	2.70	2.33	2.70	3.92

3.2 Representing the Displacement Array

We now describe how to represent the displacement array. A *write-once dynamic array* is a data structure for a sequence of supports the following operations:

create(n)**:** Create an array A of size n with all entries initialized to zero.
set(A, i, v)**:** If $A[i] = 0$, set $A[i]$ to v (assume $0 < v \leq n$). If $A[i] \neq 0$ then $A[i]$ is unchanged.
get(A, i)**:** Return $A[i]$.

The following lemma shows how to implement such a data structure. Note that the apparently slow running time of *set* is enough to represent the displacement array without asymptotic slowdown: setting $D[i] = v$ means that $O(v)$ time has already been spent in the hash table finding an empty slot for the key.

Lemma 2. *A write-once dynamic array A of size n containing non-negative integers can be represented in space $\sum_{i=1}^{n} |\gamma(A[i]+1)| + o(n)$ bits, supporting get in $O(1)$ time and set(A, i, v) in $O(v)$ amortized time.*

Proof. We divide A into contiguous blocks of size $b = (\log n)^{3/2}$. The i-th block $B_i = A[bi..bi+b-1]$ will be stored in a contiguous sequence of memory locations. There will be a pointer pointing to the start of B_i. Let $G_i = \sum_{j=bi}^{bi+b-1} |\gamma(A[j]+1)|$.

We first give a naive representation of a block. All values in a block are encoded using γ-codes and concatenated into a single bit-string (at least in essence, see discussion of the *get* operation below). A *set* operation is performed by decoding all the γ-codes in the block, and re-encoding the new sequence of γ-codes. Since each γ-code is $O(\log n)$ bits, or $O(1)$ words, long, it can be decoded in $O(1)$ time. Decoding and re-encoding an entire block therefore takes $O(b)$ time, which is also the time for the *set* operation. A *get* operation can be realized in $O(1)$ time using the standard idea of to concatenating the unary and binary portions of the γ-codes separately into two bit-strings, and to use $select_1$ operations on the unary bit-string to obtain, in $O(1)$ time, the binary portion of the i-th γ-code. The space usage of the naive representation is $\sum_i G_i + O((\sum_i G_i)/(\log n)^2) + (n \log n)/b)$ bits: the second term comes from Lemma 1 and the third accounts for the pointers and any unused space in the "last" word of a block representation. This adds up to $\sum_i G_i + o(n)$ bits, as required.

Since at most b *set* operations can be performed on a block, if any value in a block is set to a value $\geq b^2$, we can use the $\Omega(b^2)$ time allowed for this operation to re-create the block in the naive representation, and also to amortize the costs of all subsequent *set* operations on this block. Thus, we assume wlog that all values in a block are $< b^2$, and hence, that γ-codes in a block are $O(\log b) = O(\log \log n)$ bits long. We now explain how to deal with this case. We divide each block into *segments* of $\ell = \lceil c \log n / \log \log n \rceil$ values for some sufficiently small constant $c > 0$, which are followed by an *overflow* zone of at most $o = \lceil \sqrt{\log n} \log \log n \rceil$ bits. Each segment is represented as a bit-string of concatenated γ-codes. All segments, and their overflow zones, are concatenated into a single bit-string. The bit-string of the i-th block, also denoted B_i, has length at most $G_i + (b/\ell) \cdot o = G_i + O(\log n (\log \log n)^2)$. As we can ensure that a segment is of size at most $(\log n)/2$ by choosing c small enough, we can decode an individual γ-code in any segment in $O(1)$ time using table lookup. We can also support a *set* operation on a segment in $O(1)$ time, by overwriting the sub-string of B_i that represents this segment, provided the overflow zone is large enough to accommodate the new segment.

If the overflow zone is exhausted, the time taken by the *set* operations that have taken place in this segment alone is $\Omega(\sqrt{\log n} \log \log n)$. Since the length of B_i is at most $O(\sqrt{\log n} \log \log n)$ words, when any segment overflows, we

can simply copy B_i to a new sequence of memory words, and while copying, use table lookup again to rewrite B_i, ensuring that each segment has an overflow zone of exactly o bits following it (note that as each segment is of length $\Omega(\log n/\log\log n)$ bits and the overflow zones are much smaller, rewriting a collection of segments that fit into $O(\log n)$ bits gives a new bit-string which is also $O(\log n)$ bits).

One final component is that for each block, we need to be able to find the start of individual segments. As the size of a segment and its overflow zone is an integer of at most $O(\log\log n)$ bits, and there are only $O(\sqrt{\log n}\log\log n)$ segments in a block, we can store the sizes of the segments in a block in a single word and perform the appropriate prefix sum operations in $O(1)$ time using table lookup, thereby also supporting get in $O(1)$ time. This proves Lemma 2.

Theorem 1. *For any given integers M and σ and constant $0 < \alpha < 1$, there is a data structure that represents a trie on an alphabet of size σ with n nodes, where $n \leq \alpha M$, using $M\log\sigma + O(M)$ bits of memory in expectation, and supporting create() in $O(M)$ time, getRoot and getParent in $O(1)$ time, and addChild and getChild in $O(1)$ expected time. The expected time bounds are based upon the assumption that the hash function has full randomness and independence.*

Proof. Follows directly from Proposition 1 and Lemma 2, and from the observation that $|\gamma(x+1)| \leq x+2$ for all $x \geq 0$.

Remark 1. In the Bonsai (and m-Bonsai) approaches, deletion of an internal node is in general not $O(1)$ time, since the node IDs of all descendants of a node are dependent on its own node ID. It is possible in m-Bonsai to delete a leaf, taking care (as in standard linear probing) to indicate that a deleted location in Q previously contained a value, and extending Lemma 2 to allow a *reset*(i) operation, which changes $A[i]$ from its previous value v to 0 in $O(v)$ time.

3.3 Alternate Representation of the Displacement Array

The data structure of Lemma 2 appears to be too complex for implementation, and a naive approach to representing the displacement array (as in Lemma 2) may be slow. We therefore propose a practical alternative, which avoids any explicit use of variable-length coding.

The displacement array is stored as an array D_0 of *fixed-length* entries, with each entry being Δ_0 bits, for some integer parameter $\Delta_0 \geq 1$. All displacement values $\leq 2^{\Delta_0} - 2$ are stored as is in D_0. If $D[i] > 2^{\Delta_0} - 2$, then we set $D_0[i] = 2^{\Delta_0} - 1$, and store the value $D'[i] = D[i] - 2^{\Delta_0} + 1$ as satellite data associated with the key i in a second hash table.

This second hash table is represented using the original Bonsai representation, using a value $M' \sim \alpha' n'$, where n' is the number of keys stored in the second hash table, and α' is the load factor of this secondary hash table. The satellite data for this second hash table are also stored in an array of size M' with fixed-length entries of size Δ_1, where Δ_1 is again an integer parameter.

If $D'[i] \leq 2^{\Delta_1} - 2$, it is stored explicitly in the second-level hash table. Yet larger values of D are stored in a standard hash table. The values of α', Δ_0 and Δ_1 are currently chosen experimentally, as described in the next section.

In what follows, we refer to m-Bonsai with the displacement array represented as γ-codes as m-Bonsai (γ) and the representation discussed here as m-Bonsai (recursive), respectively.

4 Experimental Evaluation

4.1 Implementation

We implemented m-Bonsai (recursive), m-Bonsai (γ) and Bonsai in C++, and compared these with Bentley's C++ TST implementation [3]. The DAT implementation of [20] was not tested since it apparently uses 32-bit integers, limiting the maximum trie size to 2^{32} nodes, which is not a limitation for the Bonsai or TST approaches. The tests of [20] suggest that even with this "shortcut", the space usage is only a factor of 3 smaller than TST (albeit it is ~ 2 times faster).

Both Bonsai implementations used the `sdsl-lite` library [21]. The original Bonsai data structure mainly comprises three `sdsl` containers: firstly, the `int_vector<>`, which uses a fixed number of bits for each entry, is used for the Q array (also in m-Bonsai). In addition, we use two `bit_vector`s that to distinguish nodes in collision groups as in [1]. In m-Bonsai (γ), D is split into consecutive blocks of 256 values (initially all zero) each, which are stored as a concatenation of their γ-codes. We used `sdsl`'s `encode` and `decode` functions to encode and decode each block for the *set* and *get* operations.

The m-Bonsai (recursive) uses an alternative approach for the displacement array. D_0 has fixed length entries of Δ_0-bits, thus `int_vector<>` is the ideal container. If a displacement value is larger than Δ_0, we store it as a satellite data in a Bonsai data structure. The satellite data is stored again in an `int_vector<>` of Δ_1-bit entries. Finally, if the displacement value is even larger, then we use the standard C++ `std::map`. In Figure 1, we show how we chose the parameters for this approach. The three parameters α',Δ_0 and Δ_1 are selected given the trade-off of runtime speed and memory usage. For this example we have $\alpha' = 0.8$. Each line represents a different Δ_0 value in bits. The y-axis shows the total bits required per displacement value and the x-axis shows the choice of Δ_1 sizes in bits. As shown, there is a curve formed where its minimum point is when $\Delta_1 = 7$ for any Δ_0 values. $\Delta_0 = 3$ is the parameter with the lower memory usage. $\Delta_0 = 4$ uses relatively more memory and even though $\Delta_0 = 2$ is closer to $\Delta_0 = 3$ in terms of memory, it is slower in terms of runtime speed. This happens since less values are accessed directly from D_0 when $\Delta_0 = 2$, therefore we chose $\Delta_0 = 3$ and $\Delta_1 = 7$. Finally, we consider $\alpha' = 0.8$ as a good choice to have competitive runtime speed and at the same time good memory usage.

4.2 Experimental Analysis

The machine used for the experimental analysis is an Intel Pentium 64-bit machine with 8GB of main memory and a G6950 CPU clocked at 2.80GHz

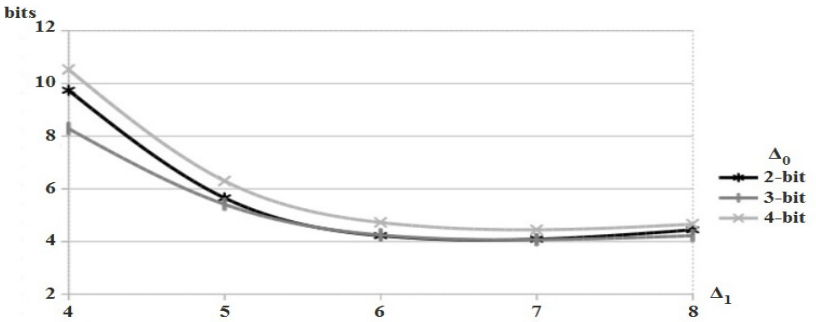

Fig. 1. This graph is an example based on Webdocs8 used in m-Bonsai (recursive) data structure with $\alpha = 0.8$. The y-axis shows the bits per M in the displacement array. The x-axis shows parameter Δ_1 and each line is based on parameter Δ_0.

Table 2. The average bits per node for datasets used for different purposes. In some cases the TST processes were unable to finish execution due to large memory required.

Datasets	Node Number	σ	m-Bonsai (r)	m-Bonsai(γ)	Bonsai	TST
Pumsb	1125375	7117	20.45	18.91	24	390.87
Accidents	4242318	442	15.65	14.12	19.2	388.26
Webdocs8	63985704	5577	20.45	18.91	24	386.79
Webdocs	231232676	5267657	27.04	30.91	36	385.1
splitPumsb	6702990	5	8.45	6.75	12	383.92
splitAccidents	17183926	5	8.45	6.75	12	387.07
splitWebdocs8	333444484	5	8.45	6.76	12	
splitWebdocs	1448707216	5	8.45	6.78	12	
SRR034939.fastq	3095560	5	8.45	6.73	12	385.88
SRR034944.fastq	21005059	5	8.45	6.76	12	385.76
SRR034940-1.fastq	1556235309	5	8.45	6.68	12	
SRR034945-1.fastq	1728553810	5	8.45	6.68	12	

with 3MB L2 cache, running Ubuntu 12.04.5 LTS Linux. All the code was compiled using g++ 4.7.3 with optimization level 6. To measure the resident memory (RES), `/proc/self/stat` was used. For the speed tests we measured wall clock time using `std::chrono::duration_cast`.

We use benchmark datasets arising arising in frequent pattern mining [22], where each "string" is a subset of a large alphabet (up to tens of thousands). In some frequent pattern mining algorithms such as [23], these strings need to be traversed in sorted order, which takes a slow $O(n\sigma)$ time in all Bonsai variants because they do not support the next-sibling operation. To get around this, we divide each symbol into 3-bit values, which optimizes the trade-off in runtime

Table 3. The wall clock time in seconds for the construction of the Trie. Note that the TST was affected by thrashing in Webdocs and splitWebdocs8.

Datasets	m-Bonsai (r)	m-Bonsai(γ)	Bonsai	TST
Pumsb	0.55	5.97	0.86	0.64
Accidents	2.06	21.70	3.12	2.33
Webdocs8	27.03	255.25	35.13	19.38
Webdocs	110.35	886.17	125.06	608.91
splitPumsb	3.30	37.03	5.21	2.29
splitAccidents	7.72	82.95	10.92	5.69
splitWebdocs8	128.88	1287.49	173.25	1862.49
splitWebdocs	626.20	5439.71	832.8	
SRR034939.fastq	0.561	9.82	0.74	0.61
SRR034944.fastq	6.041	72.38	6.84	4.39
SRR034940-1.fastq	746.005	5801.6	936.88	
SRR034945-1.fastq	851.164	6456.18	1054.43	

speed and memory usage. Finally, we used sets of short read genome strings given in the standard FASTQ format.

Memory Usage: For the memory usage experiments we set $\alpha = 0.8$ for all Bonsai data structures. Then, we insert all the strings of each dataset in the trees and we measure resident memory. Table 2 is showing the average bits per n. It is obvious that the Bonsai data structure is quite badly affected on datasets with low σ. By converting the values of Table 2 in scale of bits per M (as explained in Section 3.1 $n = \alpha M$), we prove the practical analysis of Section 2, showing that Bonsai requires $10M$-bits for the FASTQ sequences out of which $7M$-bits are used only to map the nodes in Q array. The m-Bonsai (γ) performance is very good which needs more than 40% less memory than Bonsai on lower σ datasets. The m-Bonsai (recursive) is also performing better than Bonsai and it is obvious that as σ gets lower the recursive approach becomes more efficient by avoiding the relatively big overhead of Bonsai.

Tree construction (Runtime speed): In Table 3 we show the wall clock time in seconds for the construction of the Tree. The m-Bonsai (recursive) is proved to be competitively fast and even faster than TST for some cases like Pumsb and Accidents. This happens since m-Bonsai (recursive) is able to fit a big part of data structure in cache memory. However, when both data structures use more heavily the main memory (Webdocs8), the pointer-based TST is 1.4 times faster. The Bonsai implementation is consistently slower than TST and m-Bonsai (recursive). Since the m-Bonsai (recursive) has a write once linear probing approach, when inserting a node in empty location $Q[i]$, we know that $D[i]$ is free for insertions. Now, if $D[i]$ is supposed to be zero then we don't even need to access D as it is already initialised to zeros[4]. However, Bonsai always needs to

[4] Approximately 48% of the nodes have 0 displacement value at $\alpha = 0.8$.

Table 4. The wall clock time in nanoseconds per successful search operations.

Datasets	m-Bonsai (r)	m-Bonsai(γ)	Bonsai	TST
Pumsb.search	237	1345	358	105
Webdocs8.search	332	1672	608	117
splitWebdocs.search	416	2037	657	
SRR034940-1.search	403	1932	658	

access at least one more bit-vector to reassure and mark the empty location. Additionally, in case of collision Bonsai requires to swap elements in Q and one of the bit-vectors, to make space for the new node at a matching location. Also, if any satellite data(not included in this experiment) is required, it has to move to match location as well thus potentially impacting the runtime performance. Finally, the compact m-Bonsai (γ) is about ten times slower. This is due to the O(b) time required to access each value as explained in Section 3.1 $n = \alpha M$.

Successful search runtime speed: For this experiment we designed our own *.search* datasets, where we randomly picked 10% of the strings from each dataset. As shown in Table 4 we selected some datasets from our repository mainly due to space limit. After the tree construction, we measured the time needed in nanoseconds per successful search operation. It is obvious that TST is the fastest approach. However, m-Bonsai (recursive) remains competitive with TST and consistently faster than Bonsai by at least 1.5 times, whereas m-Bonsai (γ) in the slowest. Note that there is an increase in runtime speed per search operation for all Bonsai data structures as the datasets get bigger. However, we can't prove this for TST, since it is not able to process the larger datasets.

5 Conclusion

We have demonstrated a new variant of the Bonsai approach to store large tries in a very space-efficient manner. Not only have we (re)-confirmed that the original Bonsai approach is very fast and space-efficient on modern architectures, both m-Bonsai variants we propose are significantly smaller (both asymptotically and in practice) and and one of them is a bit faster than the original Bonsai. In the near future we intend to investigate other variants, to give a less ad-hoc approach to m-Bonsai (recursive), and to compare with other trie implementations.

Neither of our approaches is very close to the information-theoretic lower bound of $(\sigma \log \sigma - (\sigma - 1) \log(\sigma - 1))n - O(\log(kn))$ bits [2]. For example, for $\sigma = 5$, the lower bound is $3.61n$ bits, while m-Bonsai (γ) takes $\sim 5.6M \sim 7n$ bits. Closing this gap would be an interesting future direction. Another interesting open question is to obtain a practical compact dynamic trie that has a wider range of operations, e.g. being able to navigate directly to the sibling of a node.

References

1. Darragh, J.J., Cleary, J.G., Witten, I.H.: Bonsai: a compact representation of trees. Softw., Pract. Exper. **23**(3), 277–291 (1993)
2. Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica **43**(4), 275–292 (2005)
3. Bentley, J., Sedgewick, B.: Ternary search trees (1998). http://www.drdobbs.com/database/ternary-search-trees/184410528
4. Nilsson, S., Tikkanen, M.: An experimental study of compression methods for dynamic tries. Algorithmica **33**(1), 19–33 (2002)
5. Jacobson, G.: Space-efficient static trees and graphs. In: Proc. 30th Annual Symposium on Foundations of Computer Science, pp. 549–554. IEEE Computer Society (1989)
6. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms **3**(4) (2007)
7. Farzan, A., Munro, J.I.: A uniform paradigm to succinctly encode various families of trees. Algorithmica **68**(1), 16–40 (2014)
8. Farzan, A., Raman, R., Rao, S.S.: Universal succinct representations of trees? In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part I. LNCS, vol. 5555, pp. 451–462. Springer, Heidelberg (2009)
9. Munro, J.I., Raman, V., Storm, A.J.: Representing dynamic binary trees succinctly. In: Kosaraju, S.R. (ed.) Proc. 12th Annual Symposium on Discrete Algorithms, pp. 529–536. ACM/SIAM (2001)
10. Raman, R., Rao, S.S.: Succinct dynamic dictionaries and trees. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 357–368. Springer, Heidelberg (2003)
11. Arroyuelo, D., Davoodi, P., Satti, S.: Succinct dynamic cardinal trees. Algorithmica, 1–36 (2015) (online first)
12. Jansson, J., Sadakane, K., Sung, W.: Linked dynamic tries with applications to lz-compression in sublinear time and space. Algorithmica **71**(4), 969–988 (2015)
13. Takagi, T., Uemura, T., Inenaga, S., Sadakane, K., Arimura, H.:Applications of succinct dynamic compact tries to some stringproblems (presented at WAAC 2013). http://www-ikn.ist.hokudai.ac.jp/~arim/papers/waac13takagi.pdf
14. Grossi, R., Ottaviano, G.: The wavelet trie: maintaining an indexed sequence of strings in compressed space. In: PODS, pp. 203–214 (2012)
15. Patrascu, M.: Succincter. In: 49th Annual IEEE Symp. Foundations of Computer Science, pp. 305–313. IEEE Computer Society (2008)
16. Knuth, D.E.: The Art of Computer Programming. Sorting and Searching, vol. 3, 2nd edn. Addison Wesley Longman (1998)
17. Carter, L., Wegman, M.N.: Universal classes of hash functions. J. Comput. Syst. Sci. **18**(2), 143–154 (1979)
18. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. J. ACM **31**(3), 538–544 (1984)
19. Pagh, R.: Low redundancy in static dictionaries with constant query time. SIAM J. Comput. **31**(2), 353–363 (2001)
20. Yoshinaga, N., Kitsuregawa, M.: A self-adaptive classifier for efficient text-stream processing. In: COLING 2014, 25th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, August 23–29, 2014, Dublin, Ireland, pp. 1091–1102 (2014)

21. Gog, S., Beller, T., Moffat, A., Petri, M.: From theory to practice: plug and play with succinct data structures. In: Gudmundsson, J., Katajainen, J. (eds.) SEA 2014. LNCS, vol. 8504, pp. 326–337. Springer, Heidelberg (2014)
22. Goethals, B.: Frequent itemset mining implementations repository. http://fimi.ua.ac.be/
23. Schlegel, B., Gemulla, R., Lehner, W.: Memory-efficient frequent-itemset mining. In: Proceedings of the 14th International Conference on Extending Database Technology, EDBT 2011, Uppsala, Sweden, March 21–24, 2011, pp. 461–472 (2011)