Appeared in: Third Workshop on Algorithms for Large-Scale Information Processing
(ALSIP 2012), Technical Note, JSAI, Miyazaki, Japan, Dec. 2012.

Speeding Up Compact Trie Structures on Word RAM and Its Applications

Takuya Takagi[1], Takashi Uemura[2], and Hiroki Arimura[1]

[1] IST & School of Engineering, Hokkaido University, N14 W9, Sapporo, 060-0814 Japan
tkg@ec.hokudai.ac.jp, arim@ist.hokudai.ac.jp
[2] Chowa Giken Corporation, 1-12, 305, N18 W3, Sapporo, 001-0018 Japan

Abstract. In this paper, we show how to speed up operations of a compact trie on Word RAM with w-bits registers using bit-parallelism and any linear-space dictionary for integers with fast predecessor and insert operations, while keeping the linear space. As a main result, we present a faster compact trie for storing a set of K strings with total length N letters in $O(N \log \sigma + K \log N)$ bits that supports prefix search (from an arbitrary node) and insert operations in $O(\lceil P/\alpha \rceil \sqrt{w})$ worst-case time or in $O(\lceil P/\alpha \rceil \log w)$ average time, where P is the length of pattern and $\alpha = \lceil w/\log \sigma \rceil$. As an application, for any finite prefix code, online sparse suffix tree construction for encoded text of K codewords in N letters can be accelerated from $O(N \log \sigma)$ time to $O(\lceil \frac{N}{\alpha} + K \rceil \sqrt{w})$ time using the same amount of space.

1 Introduction

A *compact trie* [4, 13], also called a *Patricia trie* [10], for a set of K strings over alphabet Σ is a fundamental data structure in string processing. It plays essential role in many applications, e.g., dynamic dictionary matching [5], suffix tree [13], sparse suffix tree [8], succinct index [9], and external string index [3]. The compact trie $\mathcal{T}_S$ for a set of K strings $S = \{S_1, \ldots, S_K\}$ of total size N letters on alphabet Σ of size σ is obtained from an ordinary trie for S by compressing non-branching paths so that every internal node has at least two children and each edge has a substring of S as its label, reducing the tree structure in at most $2K - 1$ nodes. Further saving the space by representing the edge labels with pairs (i, j) of pointers to a single string $T = S_1 \ldots S_K$, a compact trie can store all strings of S in $O(N \log \sigma + K \log N)$ bits of space, supporting *general prefix search* (the prefix search from an arbitrary node v), insert, and delete operations with pattern P in $O(|P| \log \sigma)$ time independent of K and N. It also supports ordinary traversal operations for tries, such as $\mathsf{child}(v, a)$ for $a \in \Sigma$ and $\mathsf{parent}(v)$, in $O(\log \sigma)$ time as well.

2 Main results

In this paper, we show how to speed up operations of a compact trie on Word RAM using bit-parallelism and predecessor dictionaries without increasing the space.

Problems that we encounter: A basic idea is to use the *packed string matching* approach [2]. Since each letter is encoded with $\log \sigma$ bits and we can read w bits in constant time on Word RAM, we expect to process $\alpha = \lceil w/\log \sigma \rceil$ consecutive letters

in one time step. However, we encounter a difficulty caused by branching during traversal of a compact trie. Actually, in the worst case, prefix search with a pattern of length P requires $O(\lceil P/\alpha \rceil + d_P \log \sigma)$ time, where $d_p = O(K)$ is the number of branching nodes to visit. Using this modification, for example, we only obtain a construction algorithm for sparse suffix trees [8, 6, 12] of a text of K index points and N letters that runs in quadratic time in $K = O(N)$.

Our proposal: To solve this problem, we devise a novel speed up technique by using bit-parallelism and fast dictionary lookup, respectively, for processing long non-branching paths and dense branching subtrees. Based on this, we propose the *fast compact trie* on $\mathcal{D}$, that is a compact trie augmented with any dynamic predecessor dictionary $\mathcal{D}$ for w-bit integers. Suppose that $\mathcal{D}$ stores N w-bit integers in $s(w, N)$ bits by supporting predecessor and insert operations in $f(w, N)$ time, where s satisfies the inquality $s(w, N_1) + s(w, N_2) \leq s(w, N_1 + N_2)$ (*). Then, we have:

Theorem 1 (main result). *The fast compact trie stores a set S of K strings of total size N letters over Σ in $O(N \log \sigma + K \log N + s(w, K))$ bits, with supporting general prefix search in $O(\lceil P/\alpha \rceil f(w, N))$ time, and insert in $O(\lceil P/\alpha \rceil f(w, N) + \log \sigma)$ time, as well as ordinary compact trie operations in $O(\log \sigma)$ time, where $\alpha = w/\log \sigma$, $\sigma = |\Sigma|$, and P is the length of a query string.*

The above result accelerates prefix search without slowing down other trie operations. For the choice of predecessor dictionary, we use: The *q-fast trie* of Willard [14] uses $s(w, N) = O(N \log N)$ bits supporting $f(w, N) = O(\sqrt{w})$ predecessor and insert operations in the worst case. The *dynamic z-fast trie* [1] uses $s(w, N) = O(N \log N)$ bits supporting $f(w, N) = O(\log w)$ predecessor and insert operations on average. We note that the space $S(w, N) = O(N \log N)$ satisfies the inequality (*).

Corollary 1. *The fast compact trie can be implemented to store a set of K strings of total size N letters in $O(N \log \sigma + K \log N)$ bits with supporting general prefix search and insert operations in $O(\lceil P/\alpha \rceil \sqrt{w})$ time in the worst case, and in $O(\lceil P/\alpha \rceil \log w)$ time on average.*

Applications: As an application of our speed up technique for compact tries, we have the following results. A sparse suffix tree (SST) [8] is a compact trie for a subset of K suffixes of an input text of length N. It has been open that whether general SSTs can be constructed in online $O(N \log \sigma)$ time using $O(N \log \sigma + K \log N)$ bits of space [8]. Inenaga and Takeda [6] showed that it is the case for the SSTs over word alphabets. Recently, Uemura and Arimura [12] extended their results for SSTs over any regular[3] prefix-code $\Delta \subseteq \Sigma^+$ of total size $\delta = ||\Delta||$ letters.

Theorem 2 (faster sparse suffix tree construction). *For any finite prefix code Δ of total size δ, the SST for an encoded text of K codewords and N letters can be constructed online in $O((\lceil \frac{N}{\alpha} \rceil + K)\sqrt{w})$ time using $O(N \log \sigma + (K + \delta) \log N)$ bits.*

We note that the dynamic z-fast trie [1] can directly support prefix search (from the root), insert, and delete operations in $O(\lceil P/\alpha \rceil + \log L)$ time on average as shown in [1], where L is the maximum size of strings in S. However, we do not know how to support general prefix search with the above time, and also in the worst case. As future work, it will be interesting to apply our technique to the following: dynamic dictionary matching [5], succinct index [9], and external string index [3].

[3] A prefix code is *regular* if it is accepted by a finite automaton.

3 The details of the proposed data structure

3.1 Preliminaries

Basic definitions: We assume the *Word RAM model*, which executes the following operations on w-bit registers in constant time: memory operations, Boolean operations &, |, $\sim$, =, $\ll$, $\gg$, and arithmetic operations + and $*$, written in C-like notation. We assume that $w \geq \log N$, not that $\Theta(w) = \log N$. Since our algorithms is based on bit-parallelism, but not tabulation, they work for all $w \geq \log N$. Let Σ be an alphabet of size σ. A speed up factor is $\alpha = \lceil w/\log \sigma \rceil$. We assume that w is a multiple of $\log \sigma$. For strings X, Y, $|X|$ is the length of X, and $LCP(X, Y)$ is the length of the longest common prefix of X and Y, in letters. We define $[i, j] = \{i, i + 1, \ldots, j\}$ $(i \leq j)$.

Compact tries: Let $S = \{S_1, \ldots, S_K\}$ be a set of K strings over Σ, whose total size is $N = \sum_i |S_i|$. Assume that no string of S is a prefix of other string in S. Let $\mathcal{T}_S$ be the *compact trie* (*trie*) for S. Then, $\mathcal{T}_S$ contains at most $K - 1$ internal nodes and K leaves in $O(K \log N)$ bits together with the associated text string $T = S_1 \cdots S_K$ over Σ of length N stored in $O(N \log \sigma)$ bits. Let v be any internal node. For any letter $a \in \Sigma$, we denote by $child(v, a)$ the a-child of v, that is, the node pointed by an edge outgoing from v whose label starts with a. If we use balanced binary tree for branching at v, $child(v, a)$ takes $O(\log \sigma)$ time. To each node v, we associate the path string $str(v)$ from the root to v. We store at v its letter-depth $d(v) = |str(v)|$.

We introduce the notion of real and virtual nodes [13] as follows. Suppose that we consider the uncompact version $\mathcal{T}'_S$ of the compact trie $\mathcal{T}_S$. Then, a node v in $\mathcal{T}'_S$ is called a *real node* if it is represented by a branching node in $\mathcal{T}_S$, and called a *virtual node* if it is not. Any virtual node in $\mathcal{T}_S$ is represented by a *reference* or a *pointer*, which is a triple $\tau = (p, k, j)$ that represents the virtual node reachable from the real node p by the substring $T[k, j]$. The *general prefix search* operation at a node v with a query string $P \in \Sigma^*$, denoted by $\mathsf{PREFIX\_SEARCH}(v, P)$, returns the reference to the node of $\mathcal{T}_S$ that is reachable from v by a prefix of P, which can be freely used as other nodes.

3.2 Speeding up prefix search: The small trie case

In this subsection, we present an efficient implementation of prefix search and insert operations with pattern of length w bits on a small trie with depth w.

Micro tries: A *micro trie* is a compacted trie S of letter-height α that stores L w-bits integers $l_1, ..., l_L$ as strings. We assume that leaves are ordered from left to right in the lexicographic order of their path strings. In preprocessing, for every $i \in [1, L]$, we associate to the i-th leaf ℓ_i the pointer $\mathsf{LCA}(\ell_i)$ to the lowest common ancestor (LCA) v of ℓ_i and ℓ_{i+1}. Since $\mathcal{S}$ is branching, this is done in $O(L)$ time. Suppose that $\mathcal{D}$ is a dynamic predecessor dictionary that stores L w-bit integers in $s(w, L)$ bits space supporting $f(w, L)$ time predecessor and insert operations.

We associate $\mathcal{D}$ to $\mathcal{S}$ as follows. Suppose that a micro trie $\mathcal{S}$ stores L w-bits binary strings $\{X_1, \ldots, X_L\}$ by its L leaves $\ell_1, \ldots, \ell_L$, and that there is an associated dictionary $\mathcal{D}$ containing key-value pairs (X_i, ℓ_i) for $i \in [1, L]$, where $\mathsf{Lab}(\ell_i) = X_i$. We assume that any internal node v has a pointer to $\mathcal{D}$. Recall that $\alpha = w/\log \sigma$.

The prefix search operation: Given a pattern X of α letters, packed in a w-bit word, this operation computes in $O(f(w, L))$ time the reference to the node ϕ in the micro trie $\mathcal{S} = \mathcal{S}_v$ that is reachable from the root v of $\mathcal{S}$ by X as follows:

1. First, we compute the letter-depth $c \in [0, \alpha]$ of ϕ as follows. We compute the leaves μ_L and μ_R, resp., as the predecessor and successor of X in $f(w, L)$ time. If $\mu_L = \mu_R$ then they coincide to ϕ, and thus return ϕ. Otherwise, μ_L and μ_R are consecutive, i.e., $R = L + 1$. Then, we compute c as the maximum length of $LCP(X, \mathsf{Lab}(\mu_L))$ and $LCP(X, \mathsf{Lab}(\mu_R))$, where LCP is computed by XOR and MSB operations in $O(\log w)$ time, where alignment of letters in $\log \sigma$ bits is done using carry propagation by integer addition ([7]).
2. Next, we will find the lowest real node p above ϕ as follows. We can see that ϕ is real if and only if the depth of the LCA of μ_L and μ_R coincide to c. (2.a) If ϕ is real, return ϕ as an answer. (2.b) Otherwise, we find the real node p as follows (See Fig. 1). Let $\beta = X[1, c]$ be the prefix of X with length c bits. We compute the successor ℓ_L of the left string $X_L = \beta\, 0^{(\alpha - c)\log \sigma}$ and the predecessor ℓ_R of the right string $X_R = \beta\, 1^{(\alpha - c)\log \sigma}$ in $\mathcal{D}$ in $f(w, L)$ time, where 0 and 1 denote single bits. Then, define the node p by the lower of $\mathsf{LCA}(\ell_{L-1})$ or $\mathsf{LCA}(\ell_R)$. Let $T[k, j]$ be the label of the edge out-going from p to the direction of ϕ.
3. Finally, we return the reference $\phi = (p, k, k + c - (d(p) - d(v)))$ as the answer.

Claim 1: The real node p coincides to the lower of $\mathsf{LCA}(\ell_{L-1})$ or $\mathsf{LCA}(\ell_R)$.

Proof for Claim 1: If we consider the subtree $\mathcal{S}_\phi$ under ϕ, there is at least one leaf ℓ inside the subtree such that $X_L \leq \mathsf{Lab}(\ell) \leq X_R$ since X_L and X_R are the leftmost and rightmost paths below ϕ. Taking the successor ℓ_L of X_L and the predecessor ℓ_R of X_R, both of them belong to $\mathcal{S}_\phi$. Since ℓ_{L-1} (resp. ℓ_{R+1}) locates outside of $\mathcal{S}_\phi$, the left-branching (resp. left-branching) real node immediate above ϕ coincides $\mathsf{LCA}(\ell_{L-1})$ (resp. $\mathsf{LCA}(\ell_R)$). This shows the claim. (*End of Proof for Claim 1*)

The above claim gives the correctness of Step 2 above. To deal with X with length less than α, we extend X by padding and explicitly store the length $|X|$ at leaf ℓ.

The insert operation: Initially, $\mathcal{S}$ contains one branch with the unique leaf ℓ, and no dictionary is associated. Suppose that the trie $\mathcal{S}$ contains L w-bits binary strings. Then, we insert a w-bit string X into $\mathcal{S}$ in $O(f(w, L) + \log \sigma)$ time as follows.

1. We first find the node ϕ by prefix search operation with pattern X. If $\phi = (p, k, j)$ is virtual, then we transform it to a real node. After that, we attach a new edge from v to a new leaf ℓ with label $X[c, \alpha]$, the rest of X, in $O(\log \sigma)$ time.
2. If $\mathcal{S}$ contains only one branch, then we associate an empty dictionary $\mathcal{D}$ with ϕ. Otherwise, we can retrieve $\mathcal{D}$ from ϕ in $O(\log \sigma)$ time since either p or its children q on which ϕ locates is real node. Then, we insert the key-value pair (X, ℓ) into $\mathcal{D}$ in $f(w, L)$ time, while we appropriately maintain LCA pointers to ϕ.

The delete operation can be implemented in the same time complexity only for the last inserted string, but we omit the details.

3.3 Speeding up prefix search: The large trie case

We present our fast compact trie by indexing based on the previous speed up technique for micro tries.

Micro trie decomposition: Suppose that we are given a compact trie $\mathcal{T}_S$ for a set S of K strings of total length N letters. Recall that consecutive α letters can be packed into a w-bit word. We assume that the height d of $\mathcal{T}_S$ is multiple of α. Then, we split the nodes of $\mathcal{T}_S$ into $\lceil d/\alpha \rceil$ levels as follows. A *boundary node* is a node v, either

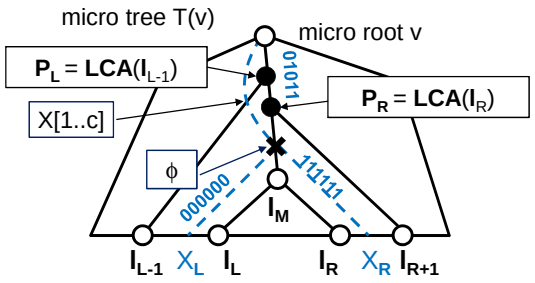

Fig. 1. Small trie case: the correctness of prefix search operation (Claim 1)

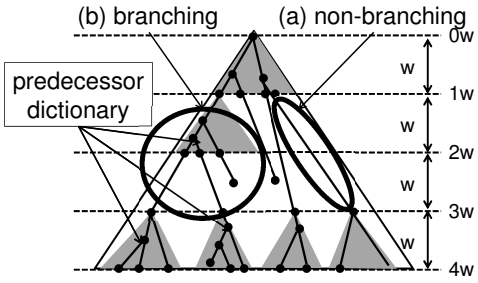

Fig. 2. Large trie case: augmentation by predecessor dictionary

real or virtual, whose letter-depths $d(v)$ is a multiple of α, i.e., $0\alpha, 1\alpha, 2\alpha, \ldots, \lceil d/\alpha \rceil$. For each $i \in [0, \lceil d/\alpha \rceil - 1]$, the i-th level consists of all nodes ϕ with letter-depth $d(\phi)$ in $[i\alpha, (i+1)\alpha - 1]$. For each boundary node v with level i, we augment the corresponding micro trie region $\mathcal{S}_v$ by a predecessor dictionary $\mathcal{D}_v$ if it contains at least one branching nodes except its leaves as described in Sec. 3.2.

More precisely, the *micro trie region* (or *micro trie*) $\mathcal{S}_v$ rooted at v of level i, where $d(v) = i\alpha$, is the connected subtree of $\mathcal{T}_S$ consisting of all descendants of v each of which is either an internal node of level i or a leaf in the $(i+1)$-th boundary such that $\mathcal{S}_v$ contains at least one branching node except its leaves. Each branching, thus real, node in $\mathcal{S}_v$ has a pointer to $\mathcal{D}_v$. Now, we define the *fast compact trie* for S, in the large case, as a pair $(\mathcal{T}_S, \mathcal{D}_S)$ of $\mathcal{T}_S$ and a set $\mathcal{D}_S$ of the associated dictionaries.

The prefix search operation: Given a pattern X of length $P = O(2^w)$, we implement general prefix search from any node v in $(\mathcal{T}_S, \mathcal{D}_S)$ to run in $O(\lceil P/\alpha \rceil f(w, N))$ time as follows. We assume w.o.l.g. that v is a boundary node in some level $i \geq 0$; Otherwise, we can start from the micro root associated to v.

1. First, we test if a micro trie region $\mathcal{S}_v$ is associated to v. If so, we make a prefix search with the α-prefix $X[1, \alpha]$ on the dictionary $\mathcal{D}_v$ as in Sec. 3.2. If the answer ϕ is an internal node, return ϕ. Otherwise, ϕ is a boundary node of level $i+1$. Then, we repeat the above process after setting $X = X[\alpha + 1, |X|]$ and $v = \phi$.
2. Otherwise, v is the upper end of a non-branching edge labeled with a path string Y. We match a prefix of X against Y by reading α consecutive letters in one time step by computing the LCP length c of $X[1, \alpha]$ and $Y[1, \alpha]$ in $O(\log w)$ time as before. If $c < w$, then we return the reference ϕ to the disagreement node. Otherwise, we continue the search for the rest of pattern $X = X[\alpha + 1, |X|]$.

The insert operation: Initially, the trie $\mathcal{T}_S$ consists of the root only. Then, we can insert the initial string to $\mathcal{T}_S$ as usual. Given a non-empty trie $\mathcal{T}_S$, we insert a new string X at specified node ψ in $O(\lceil P/\alpha \rceil f(w, N) + \log \sigma)$ time as follows.

1. We append X to the text T. We make prefix search with X from ψ and obtain a reference $\phi = (p, k, j)$. Let v be the boundary node above ϕ, and $\mathcal{S}_v$ be the associated micro trie region.
2. If no dictionary is associated to $\mathcal{S}_v$, we create a new empty dictionary $\mathcal{D}_v$ and associated it to $\mathcal{S}_v$. Materialize the root and the leaves of $\mathcal{S}_v$ as real, and associate $\mathcal{D}$ to these nodes. Otherwise, we obtain $\mathcal{D}_v$ from some real node, which is at least a parent or a child of ϕ. Then, insert the key-value pair $(X[1, \alpha], \ell)$ for a new real node ℓ, and attach to ℓ a new edge labeled by the rest of pattern $X[\alpha + 1, |X|]$.

Analysis: The time complexity is clear from the discussion in the previous subsections. On space complexity, if each micro trie $\mathcal{S}_v$ contains at most K_v nodes including its leaves, the dictionary $\mathcal{D}_v$ uses $O(s(w, K_i))$ bits. Moreover, every real, branching node belongs to at most two micro trie regions. If the space $s(w, N)$ satisfies the inequality (*) in Sec. 2, the total space is $S = \sum_i O(s(w, K_i)) = O(s(w, N))$, this proves Theorem 1. Since it is the case for the space $s(w, N) = O(N \log N)$ of the q-fast trie or the dynamic z-fast trie, Corollary 1 follows. We can prove Theorem 2 from Corollary 1 and the result of [12], whose proof will appear elsewhere ([11]).

4 Conclusion

We presented a faster compact trie with linear space that supports general prefix search and insert operations in sublinear time, $O(\lceil P/\alpha \rceil \sqrt{w})$ worst-case time and $O(\lceil P/\alpha \rceil \log w)$ average-case time, in pattern size P on the Word RAM with w-bit words, where σ is the alphabet size and $\alpha = \lceil w/\log \sigma \rceil$.

Acknowledgements. The authors would like to thank Koji Tsuda, Shin-ichi Minato, Shunsuke Inenaga, Masayuki Takeda, Ayumi Shinohara, Takuya Kida, Shuhei Denzumi for their discussions and valuable comments. This research was partly supported by MEXT Grant-in-Aid for Scientific Research (A), 24240021, FY2012–2015.

References

1. D. Belazzougui, P. Boldi, and S. Vigna. Dynamic z-fast tries. In *Proc. SPIRE 2010*, Vol. 6393, *LNCS*, 159–172, 2010.
2. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann. Optimal packed string matching. In *Proc. FSTTCS 2011*, Vol. 13, *LIPIcs*, 423–432, 2011.
3. P. Ferragina and R. Grossi. The string B-tree: a new data structure for string search in external memory and its applications. *J. ACM*, 46(2):236–280, 1999.
4. D. Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer science and computational biology.* Cambridge, 1997.
5. W.-K. Hon, T.-W. Lam, R. Shah, S.-L. Tam, and J. Vitter. Succinct index for dynamic dictionary matching. In *Proc. ISAAC'09, LNCS 5878*, 1034–1043, 2009.
6. S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *Proc. CPM'06*, Vol. 4009, *LNCS*, 60–71, 2006.
7. Y. Kaneta, H. Arimura, and R. Raman. Faster bit-parallel algorithms for unordered pseudo-tree matching and tree homeomorphism. *JDA*, 14:119–135, 2012.
8. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. COCOON'96*, Vol. 1090, *LNCS*, 219–230, 1996.
9. R. Kolpakov, G. Kucherov, and T. Starikovskaya. Pattern matching on sparse suffix trees. In *Proc. CCP'11*, 92–97, 2011.
10. D. R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
11. T. Takagi, T. Uemura, and H. Arimura. Faster sparse suffix tree construction on Word RAM. Technical report, SIG-AL-142-9, IPSJ, Nov. 2012. (in Japanese) `http://www-ikn.ist.hokudai.ac.jp/~t-takagi/paper/SIGAL142.pdf`.
12. T. Uemura and H. Arimura. Sparse and truncated suffix trees on variable-length codes. In *Proc. CPM'11*, Vol. 6661, *LNCS*, 246–260, 2011.
13. E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 13(3):249–260, 1995.
14. D. E. Willard. New trie data sturucture which support very fast search operations. *Journal of Computer and System Sciences*, 28:379–394, 1984.