Proc. of 16th Japan-Korea Joint Workshop on Algorithms and Computation (WAAC 2012), Juy 2013 (talk only)

Applications of Succinct Dynamic Compact Tries to Some String Problems

Takuya Takagi[1], Takashi Uemura[2], Shunsuke Inenaga[3], Kunihiko Sadakane[4], and Hiroki Arimura[1]

[1] IST & School of Engineering, Hokkaido University, Japan
tkg@ist.hokudai.ac.jp, arim@ist.hokudai.ac.jp
[2] Chowa Giken Corporation, Japan
[3] Department of Informatics, Kyushu University, Japan
inenaga@inf.kyushu-u.ac.jp
[4] National Institute of Informatics, Japan
sada@nii.ac.jp

Abstract. The dynamic compact trie is a fundamental data structure for a wide range of string processing problems. In this paper, we report our recent work on succinct dynamic compact tries that stores a set of strings of total length n in $O(n \log \sigma)$ space supporting pattern matching and insert/delete operations in $O((|P|/\alpha)f(n))$ time, where P is a pattern string, $\alpha = \Theta(\log_\sigma n)$, and $f(n) = O((\log\log n)^2/\log\log\log n)$, and its applications to the following string processing problems: (i) online r-suffix tree construction with $O(\alpha/f(n))$ speed up, (ii) succinct substring index with almost same query time and $O(\alpha/f(n))$ speed up on preprocessing time, and (iii) dynamic dictionary matching with $O(\alpha/f(n))$ speed up.

1 Introduction

Backgrounds: The *dynamic compact trie* [6, 14] for a set of k strings over alphabet Σ is a fundamental data structure in string processing, where its binary version is called a *Patricia trie* [12]. The classical dynamic compacted trie data structure stores a set of k strings of total length n compactly supporting pattern matching in $O(|P| \log \sigma)$ time and insert/delete operations in $O(\log \sigma)$ time. One of its most important properties seems that if all label strings are take from a single reference string of length n, then the compact trie occupies at most $O(n \log \sigma + k \log n)$ bits of space even if their total length is quadratic [6]. For this virtue, it plays an essential role in a number of string problems such as dynamic dictionary matching [7], suffix tree [14], sparse suffix tree [10], succinct index [11], and external string index [5].

Jansson, Sadakane, and Sung [9] presented the compressed dynamic uncompacted trie data structure for a set of strings of total length n in $O(n \log \sigma)$ space supporting pattern matching in $O((|P|/\alpha)f(n))$ time and insert/delete operations in $O(f(n))$ time, where $\alpha = \log_\sigma n$ and $f(n) = ((\log\log n)^2/\log\log\log n)$ is the present best time bound for dynamic predecessor dictionary. Besides its advantage, it is not applicable to online linear time suffix tree construction because it has quadratic space complexity since it is uncompacted.

2 Main results

Our goal is to devise a dynamic *compacted* trie data structure that stores a set of strings of total length n in optimal $O(n \log \sigma)$ bits of space supporting pattern matching and other operations in sublinear time. By extending the work by Jansson *et al.*, we present the dynamic *compact* trie data structure that can store a set of K strings of a single reference string of length n in $O(n \log \sigma + K \log n)$ bits of space even if the total length of edge labels is quadratic in n, still supporting pattern matching and insert/delete operations in the same time complexity as Jansson *et al.*'s compressed dynamic uncompacted trie.

For proving the above result, 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. In this technique, we augment the nodes of a compact trie with any dynamic predecessor dictionary $\mathcal{D}$ on $O(\log n)$-bit integers to speed up branching. For handling a long non-branching path, we use the *packed string matching* approach [3], in which we read and process consecutive $\alpha = \Theta(\log_\sigma n)$ letters in one time step by using bit-wise Boolean and arithmetic operations on Word RAM expecting α times speed up.

In the following results, $\mathcal{D}$ is any linear space dynamic predecessor dictionary [1, 2] that stores k $O(\log n)$-bit integers in $s(n) = O(k \log n)$ bits by supporting predecessor, successor, insert, and delete operations in $f(n)$ time. We also use the predecessor dictionary for implementing the node branching. Then, we have:

Theorem 1 (main result). *The proposed dynamic compact trie data structure stores a set S of k strings of total size n letters over Σ in $O(n \log \sigma + k \log n)$ bits, with supporting general prefix search in $O((|P|/\alpha) f(n))$ time, and insert in $O((|P|/\alpha) f(n))$ time, as well as traversal operations in $O(f(n))$ time all in the worst case, where P is a pattern string, and $\alpha = \Theta(\log_\sigma n)$.*

The above result accelerates prefix search without slowing down other trie operations by a factor of $O(\alpha / f(n))$. If we employ the dynamic data structure for $O(\log n)$-bit integers by Beame and Fich [1] as auxiliary structure, which supports $f(n) = O(((\log \log n)^2 / \log \log \log n))$ predecessor, successor, insertion, and deletion operations, then we have the following results.

Theorem 2. *The proposed dynamic compact trie data structure stores a set S of mutually distinct variable-length strings with total size n letters in $O(n \log \sigma)$ bits supporting pattern matching operation in $O((|P|/\alpha)((\log \log n)^2 / \log \log \log n))$ time and in insert/delete operation in $O((|P|/\alpha)((\log \log n)^2 / \log \log \log n))$ time both in the worst case, where $\alpha = \Theta(\log_\sigma n)$.*

3 Applications

As applications, we show that our data structure can be used to solve the following string processing problems.

Table 1. Summary of results, where n is the input length, $|P|$ is the length of a pattern string, and $log_\sigma n = \log n / \log \sigma$ is a speed-up factor. Moreover, we used $k = n/\log_\sigma n$ if necessary.

Result	Space (bits)	Query Time	Update Time
Compressed dynamic trie			
JSS'07 [9]	$O(n \log \sigma)$	$O((\frac{\|P\|}{\log_\sigma n})\frac{(\log\log n)^2}{\log\log\log n})$	$O((\frac{\|P\|}{\log_\sigma n})\frac{(\log\log n)^2}{\log\log\log n})$ *[5]
this	$O(n \log \sigma)$	$O((\frac{\|P\|}{\log_\sigma n})\frac{(\log\log n)^2}{\log\log\log n})$	$O((\frac{\|P\|}{\log_\sigma n})\frac{(\log\log n)^2}{\log\log\log n})$
Online sparse suffix tree construction			
KU'95 [10]	$O(n \log \sigma)$	$O(\|P\| \log \sigma)$	$O(n \log \sigma)$
this	$O(n \log \sigma)$	$O((\frac{\|P\| \log \sigma}{\log n})\frac{(\log\log n)^2}{\log\log\log n})$	$O((\frac{n \log \sigma}{\log n})\frac{(\log\log n)^2}{\log\log\log n})$
Succinct substring index			
KKS'11 [11]	$O(n \log \sigma)$	$O(\|P\| \log \sigma \frac{(\log\log n)^2}{\log\log\log n})$	$O(n \log \sigma)$
this	$O(n \log \sigma)$	$O(\|P\| \log \sigma \frac{(\log\log n)^2}{\log\log\log n})$	$O((\frac{n \log \sigma}{\log n})\frac{(\log\log n)^2}{\log\log\log n})$
Dynamic dictionary matching			
FAIPS'95	$O(n \log n)$	$O((n \log n + occ)(\frac{\log n}{\log\log n}))$	$O(\|P\|(\frac{\log n}{\log\log n}))$
HLSTV'09 [7]	$O(n \log \sigma)$	$O(n \log n + occ)$	$O(\|P\| \log \sigma + \log n)$
this	$O(n \log \sigma)$	$O(n(\log \sigma)\frac{(\log\log n)^2}{\log\log\log n})$	$O((\frac{\|P\| \log \sigma}{\log n})\frac{(\log\log n)^2}{\log\log\log n})$

Problem 1: The sparse suffix tree (SST) [10] 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 [10]. Inenaga and Takeda [8] showed that it is the case for the SSTs over word alphabets. Recently, Uemura and Arimura [13] extended their results for SSTs over any regular[6] prefix-code $\Delta \subseteq \Sigma^+$ of total size $\delta = ||\Delta||$ letters. However, it has been open if we could build the SST in sub-linear space. Applying our dynamic compact trie data structure, we have the following result.

Corollary 1. *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 $O(((n/\alpha) + k)((\log\log n)^2/\log\log\log n))$ time using $O(n \log \sigma + (k + \delta) \log n)$ bits, where $\alpha = \Theta(\log_\sigma n)$.*

If we consider the r-evenly spaced sparse suffix tree for $r = \log_\sigma n$ [10], then we have a sublinear time algorithm for online r-suffix tree construction in $p = O((n/\alpha)f(n))$ time.

[5] According to the original paper [9], $O(\frac{(\log\log n)^2}{\log\log\log n})$ update time is achieved if the location of the leaf node to be inserted/deleted is given. Here, for consistency to our results the time needed to find the location is taken account, and then the update time becomes $O((\frac{|P|}{\log_\sigma n})\frac{(\log\log n)^2}{\log\log\log n})$.

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

Problem 2: Kolpakov, Kucherov and Starikovskaya [11] proposed a dynamic succinct substring indexes using sparse suffix trees. We apply the dynamic data structure to this problem, preprocessing time is $O(\alpha/f(n))$ times speed up to the original algorithm for succinct index construction by Kolpakov *et al.*. The search time is not improved yet.

Corollary 2. *For a text of n letters, the succinct substring indexes using r-sparse suffix trees, where $r = \log_\sigma n$, can solve a pattern matching problem in $O(|P|f(n)\log\sigma + n\log\sigma)$ time using $O(n\log\sigma)$ bits of space and $O((n/\alpha)f(n))$ preprocessing, where P is a pattern string.*

Problem 3: Hon, Lam, Shah and Vitter [7] showed a succinct dynamic dictionary matching in $O(n\log n)$ time using $O(m\log\sigma)$ bits and $O(m\log\sigma)$ preprocessing time. We apply the dynamic data structure to this problem, we have the following result.

Corollary 3. *For a text of n letters, the succinct dynamic dictionary matching can be solved in $O(n(\log\sigma)f(n))$ time and $O(m\log\sigma)$ bits of space and $O((m/\alpha)f(n))$ preprocessing, where m is the total length of pattern strings in a set $S \subseteq \Sigma^*$.*

Our algorithm slightly improves on the $O(n\log n)$-time algorithm of Hon et al. [7] by factor of $O(\log n/(f(n)\log\sigma)) = O(\alpha/f(n))$ with the same $O(n\log\sigma)$ bits of space. In Table 1, we show the summary of results in Sec. 2 and Sec. 3.

4 Experimental Results

Experimental settings: As datasets, we used Japanese text files in UTF-8 encoding, whose statistics are shown in Table 2. *Japanese Wikititles* dataset consisted of titles of Japanese Wikipedia pages from the Wikipedia site[7], which we prepared the sorted and unsorted versions. *Japanese Newspapers* dataset consisted of Japanese newspaper articles from KyotoCorpus4.0 [8].

We implemented the accelerated version of the sparse suffix tree construction algorithm [13] using the proposed dynamic compact trie data structure in C/C++ as described in Sec. 2 and Sec. 3. Specifically, we implemented the following versions of sparse suffix tree (SST) construction algorithms:

- ST: Ukkonen's suffix tree (ST) construction algorithm [14].
- SST: The SST construction algorithm by Uemura *et al* [13].
- SST_L and SST_LH: Faster SST construction algorithms based on dynamic compacted trie in Sec. 4 that uses bit-parallel LCP computation (L) only, and both LCP and the hash table [9] (LH), where the table is attached to the root and stores labels of length three.

[7] `http://dumps.wikimedia.org/jawiki/`.

[8] `http://nlp.ist.i.kyoto-u.ac.jp/nl-resource/corpus/`

[9] In the experiment, we actually used the string container `map<string>` in C++/STL library, which is implemented with the balanced binary tree [4].

Table 2. Description of the datasets

Data set	Code	Total size (Byte)	Number of strings	Ave. string length (Byte)	Ave. code length (Byte)
Japanese Wikititles	UTF-8	30,414,297	1,372,988	22.1	2.44
Japanese Newspaper	UTF-8	30,400,000	102,674	296.1	2.97

Table 3. The summary of experimental results

Data set	Tree size (nodes)		Construction time(Sec.)			
	ST	SST	ST	SST	SST_L	SST_LH
Japanese Wikititles (unsorted)	45,526,142	18,409,345	72.52	29.76	**21.91**	23.35
Japanese Wikititles (sorted)	46,663,980	18,809,217	72.62	24.93	**13.41**	14.79
Japanese Newspaper	48,484,390	16,273,137	55.83	20.17	**13.61**	15.22

In the above implementations, we use the speed up factor $\alpha = 4$ since our machine is 32-bit PC. We compiled the programs with g++ using -O2 option. ran the experiments on a 1.7 GHz Intel Core i5 processor with 4 GB of memory, running Mac OS X 10.7.5. We measured the total construction time of the suffix trees by each algorithm on a given input string, and also that of the generalized suffix trees.

Results: In Table. 3, we show our experimental results, which showed the total number of the nodes of trees for ST (suffix trees) and sparse suffix trees, and the construction time of the ordinary and sparse suffix trees by the above algorithms. From the result, we observed that the algorithm SST_L with LCP was 36% faster than SST, and moreover 231% faster than ST, while SST_LH augmented by both of LCP and Hash table was even 6% slower than SST_L. This result shows that the proposal of speeding-ups by LCP-computation is useful for long non-branching paths, while it remains still open to efficiently handle the complex branching computation of compacted tries preserving the dynamic nature of the compact trie.

5 Conclusion

We presented a faster dynamic compact trie with linear space that supports pattern matching, and insert/delete operations in $O(\lceil(|P|/\alpha)\rceil f(n))$ worst-case time in pattern size P on the Word RAM with $w = O(\log n)$-bit words, where σ is the alphabet size, $\alpha = \lceil w/\log\sigma\rceil$, and $f(n) = O((\log\log n)^2/\log\log\log n)$.

References

1. P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38 – 72, 2002.
2. D. Belazzougui, P. Boldi, and S. Vigna. Dynamic z-fast tries. In *Proc. SPIRE 2010*, Vol. 6393, *LNCS*, 159–172, 2010.
3. O. Ben-Kiki, P. Bille, D. Breslauer, L. Gasieniec, R. Grossi, and O. Weimann. Optimal packed string matching. In *Proc. FSTTCS 2011*, Vol. 13, 423–432, 2011.

4. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2 edition, 2001.
5. 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.
6. D. Gusfield. *Algorithms on Strings, Trees, and Sequences – Computer science and computational biology*. Cambridge, 1997.
7. 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.
8. S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *Proc. CPM'06*, Vol. 4009, *LNCS*, 60–71, 2006.
9. J. Jansson, K. Sadakane, and W.-K. Sung. Compressed dynamic tries with applications to lz-compression in sublinear time and space. In *Proc. FSTTCS 2007, LNCS 4855*, 424–435, 2007.
10. J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. COCOON'96*, Vol. 1090, *LNCS*, 219–230, 1996.
11. R. Kolpakov, G. Kucherov, and T. Starikovskaya. Pattern matching on sparse suffix trees. In *Proc. CCP'11*, 92–97, 2011.
12. D. R. Morrison. PATRICIA: Practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, 1968.
13. T. Uemura and H. Arimura. Sparse and truncated suffix trees on variable-length codes. In *Proc. CPM'11*, Vol. 6661, *LNCS*, 246–260, 2011.
14. E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 13(3):249–260, 1995.