Theoretical Computer Science 475 (2013) 113–119

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Faster compressed dictionary matching☆

Wing-Kai Hon [a], Tsung-Han Ku [a,*], Rahul Shah [b], Sharma V. Thankachan [b], Jeffrey Scott Vitter [c]

[a] Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
[b] Department of Computer Science, Louisiana State University, Baton Rouge, LA, USA
[c] Department of Computer Science, The University of Kansas, Lawrence, KS, USA

ARTICLE INFO

Article history:
Received 31 March 2012
Received in revised form 1 August 2012
Accepted 11 October 2012
Communicated by G. Italiano

ABSTRACT

Given a set $\mathcal{D}$ of d patterns, the dictionary matching problem is to index $\mathcal{D}$ such that for any query text T, we can locate the occurrences of any pattern within T efficiently. When $\mathcal{D}$ contains a total of n characters drawn from an alphabet of size σ, Hon et al. (2008) [12] gave an $nH_k(\mathcal{D})+o(n\log\sigma)$-bit index which supports a query in $O(|T|(\log^{\epsilon} n+\log d)+occ)$ time, where $\epsilon > 0$ and $H_k(\mathcal{D})$ denotes the kth-order entropy of $\mathcal{D}$. Very recently, Belazzougui (2010) [3] has proposed an elegant scheme, which takes $n\log\sigma + O(n)$ bits of index space and supports a query in optimal $O(|T| + occ)$ time. In this paper, we provide connections between Belazzougui's index and the XBW compression of Ferragina and Manzini (2005) [8], and show that Belazzougui's index can be slightly modified to be stored in $nH_k(\mathcal{D}) + O(n)$ bits, while query time remains optimal; this improves the compressed index by Hon et al. (2008) [12] in both space and time.

1. Introduction

Given a set $\mathcal{D}$ of d patterns, the dictionary matching problem is to index $\mathcal{D}$ such that for any query text T, we can locate the occurrences of any pattern within T efficiently. Such a query is called the *dictionary matching query*. This problem is well-studied [1–3,5,12,16], and finds applications in computer virus detection and bio-informatics.

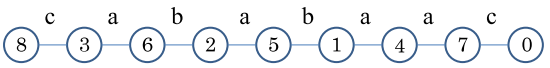
When $\mathcal{D}$ contains a total of n characters drawn from an alphabet of size σ, Aho and Corasick [1] proposed a data structure, now popularly known as the *Aho–Corasick automaton*, which supports the dictionary matching query in $O(|T| + occ)$ time. The automaton consists of a trie structure with $t \leq n + 1$ nodes, and can be stored in $O(t \log t)$ bits of space. Alternatively, we may also store the generalized suffix tree [14,17] for the patterns in $\mathcal{D}$, so that the query time remains $O(|T|+occ)$, while the space becomes $O(n \log n)$ bits. Recent research focussed on reducing the index space for this problem. Hon et al. [12] gave an $nH_k(\mathcal{D}) + o(n \log \sigma)$-bit index which supports a query in $O(|T|(\log^{\epsilon} n + \log d) + occ)$ time, where $\epsilon > 0$ and $H_k(\mathcal{D})$ denotes the kth-order entropy of the text formed by concatenating all the patterns in $\mathcal{D}$. Very recently, Belazzougui [3] has proposed an elegant scheme, which takes $tH_0(\mathcal{D}) + O(t)$ bits of index space and supports a query in $O(|T| + occ)$ time without slowdown. Note that $H_k(\mathcal{D}) \leq H_0(\mathcal{D}) \leq \log \sigma$.

In this paper, we provide connections between Belazzougui's index and the XBW compression of Ferragina et al. [7], and show that Belazzougui's index can be slightly modified to be stored in $tH_k(\mathcal{D}) + O(t)$ bits, while query time remains $O(|T| + occ)$; this improves the compressed index by Hon et al. [12] in both space and time. Note that the achieved space

☆ This work is supported in part by Taiwan NSC Grant 99-2221-E-007-123 (W. Hon) and US NSF Grants CCF-1017623 (R. Shah and J. Vitter) and CCF-1218904 (R. Shah).

* Corresponding author. Tel.: +886 932321602.
E-mail addresses: wkhon@cs.nthu.edu.tw (W.-K. Hon), thku@cs.nthu.edu.tw (T.-H. Ku), rahul@csc.lsu.edu (R. Shah), thanks@csc.lsu.edu (S.V. Thankachan), jsv@ku.edu (J.S. Vitter).

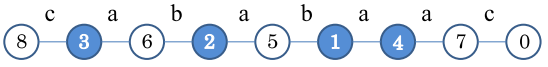
0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.10.050



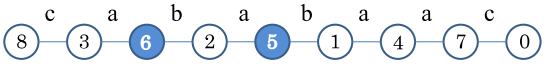
BWT = c b b c a a a a

Fig. 1. An example of the Burrows–Wheeler transform.

Locations that match with "a"



Locations that match with "ba"



Locations that match with "aba"

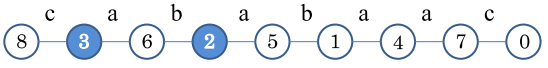


Fig. 2. An example of searching with the BWT.

bound is the same as that required by performing front coding [18] of the patterns with a subsequent kth-order entropy compression, which is likely to be smaller than the usual $nH_k(\mathcal{D}) + O(n)$ bits obtained by compressing each pattern independently.[1]

The organization of the paper is as follows. In Section 2, we give a review of the Burrows–Wheeler transformation (BWT) [4] and the XBW compression of Ferragina et al. [7]. Then in Section 3, we solve a related problem called *prefix matching* based on the XBW compression. Section 4 describes how to directly apply the result in Section 3 to obtain our index for the dictionary matching problem. We conclude the paper in Section 5.

2. Preliminaries

2.1. A review of the BWT

Let $T[1..n]$ be a text. The *Burrows–Wheeler transform* (BWT) of T is a permutation of the characters in T, such that the location of $T[i]$ is determined by the rank of the suffix $T[i+1..n]$ among all suffixes of T. (Here, we assume the rank of the empty suffix, $T[n+1..n]$, is 0.) For example, consider Fig. 1 which shows a text $T = \texttt{cababaac}$. The rank of each suffix is marked by the node to the left of the suffix. By storing the character preceding the ith smallest suffix in increasing order of i, we obtain the BWT of T.

Ferragina and Manzini [8] showed that we can store the BWT of T with some auxiliary data structures in $nH_k(T) + O(n)$ bits of space, such that we can locate the occurrences of any pattern $P[1..p]$ within T efficiently. Basically, the searching algorithm first identifies all locations in T that match with $P[p..p]$, and then iteratively identifies the locations matching $P[i..p]$ based on the locations that match with $P[i+1..p]$. For example, consider Fig. 2 which shows how to search $P = \texttt{aba}$ in the text T of Fig. 1. The searching algorithm starts by finding the locations that match with a, then the locations that match with ba, and finally the locations that match with aba. Note that in each step, the ranks of the matching locations (highlighted nodes) form a contiguous range. The searching algorithm makes use of this fact to *implicitly* represent the matches, and then to compute the desired range of the matches in the next step.

Ferragina et al. [9] showed that if the alphabet size σ is polylog(n), each step can be done in $O(1)$ time. In general, they showed that each step can be performed in $O(\log\sigma / \log\log n)$ time.

2.2. A review of the XBW

Ferragina et al. [7] proposed a generalization of the BWT that is capable for encoding rooted tries. Given a trie $\mathcal{T}$, we encode the tree structure and the edge labels separately, where the latter are stored as a text string *XBW* analogous to the

[1] Consider an extreme example, where we have a complete set of n binary strings, each with length $\log n$ bits. Each string shares a common prefix of $\log n - 1$ bits with its predecessor or successor (in the lexicographical order), which intuitively favors the front encoding. Indeed, these n strings can be encoded in $O(n)$ bits with front encoding [6], whereas the traditional nH_k-type encoding needs at least 1 bit per character, taking $\Omega(n \log n)$ bits.

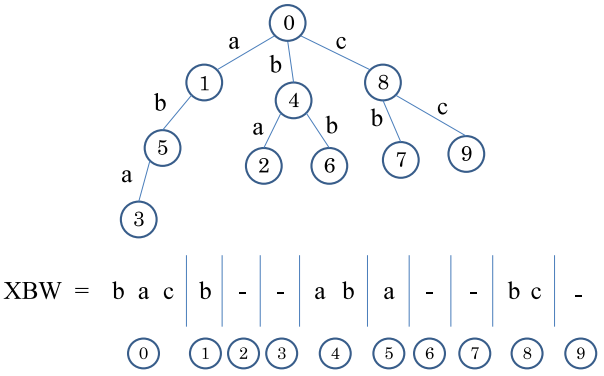


Fig. 3. An example of the XBW transform.

BWT. Precisely, for each node u in the trie, we define a string r_u, called the *reverse prefix string* of u, formed by concatenating the characters along the path from u to the root. Then each node u is associated with the rank of r_u among all the reverse prefix strings in the trie. Finally, the label on the edge (u, v) is stored in the *XBW* text in (monotonic) increasing order of the rank of r_u. For example, consider Fig. 3 which shows a trie. The rank of the reverse prefix of each node is marked inside the node. By storing the character(s) succeeding the node with the ith smallest reverse prefix in increasing order of i, we obtain the XBW of the trie. Note that when the trie contains exactly one path, the XBW is equivalent to the BWT. Also, with the XBW encoding, the ordering of concatenating the labels following a node does not matter.

Ferragina et al. [7] also extended the notion of entropy compression on a text T to a trie $\mathcal{T}$. For each string ρ, we define cover[ρ] to be the string formed by concatenating the labels on the edges (u, v) such that r_u is prefixed by ρ. For instance, in the example of Fig. 3, cover[b] $=$ aba $--$ and cover[ba] $=$ a (Note that the order of characters among the cover is irrelevant). The kth-order entropy of the trie $\mathcal{T}$, denoted by $H_k(\mathcal{T})$, is defined as:

$$H_k(\mathcal{T}) \equiv \frac{1}{t} \sum_{\text{length-}k\text{ string }\rho} |\text{cover}[\rho]| H_0(\text{cover}[\rho]),$$

where t is the number of nodes in $\mathcal{T}$, and $H_0(s)$ for a string s is the standard zeroth-order entropy of s.[2]

Then, Ferragina et al. showed that we can store the XBW of $\mathcal{T}$ with some auxiliary data structures in $tH_k(\mathcal{T}) + O(t)$ bits of space, such that for any pattern $P[1..p]$, we can locate all the subpaths (ancestor–descendent paths) within $\mathcal{T}$ efficiently. The searching algorithm is analogous to that of searching the BWT, where it first identifies all locations of the subpaths that match with $P[1]$, and then iteratively identifies the locations of the subpaths matching $P[1..i+1]$ based on the locations corresponding to $P[1..i]$.[3]

In each step, the ranks of the matching locations form a contiguous range. Again, we can make use of this fact to *implicitly* represent the matches, and then to compute the desired range of the matches in the next step. It is shown that if the alphabet size σ is polylog(n), each step can be done in $O(1)$ time. For larger alphabet size, each step can be performed in $o((\log\log\sigma)^{1+\epsilon})$ time, for any fixed $\epsilon > 0$, and with an extra term of $o(t\log\sigma)$ bits in the index space [7].

3. Compressed prefix matching with the XBW

Suppose that we want to index a trie $\mathcal{T}$, but instead of supporting subpath query, we want to check if a pattern $P[1..p]$ can be searched starting from the root. We call this a *prefix matching* query. Obviously, we can create a dummy root node z and connect z to the original root with a special character λ, and then reduce the prefix matching query to finding the subpath λP in the modified trie. In this way, the prefix matching query can be performed in $o(|P|(\log\log\sigma)^{1+\epsilon})$ time, while the index takes $nH_k(\mathcal{T}) + O(t)$ bits of space.

A natural question is whether we can reduce the query time to $O(|P|)$, for any alphabet size σ. Very recently, Belazzougui [3] has (implicitly) showed that it is possible. The idea is to find an encoding such that given a node u with rank x and a character c, we can determine if we can extend the node u by the character c in $O(1)$ time, and if so, the rank of the resulting node. His scheme is as follows:

1. For each character c in the alphabet, compute a list with the ranks of all reverse prefixes whose corresponding nodes are succeeded by c.[4] For instance, in the example of Fig. 3, we compute

[2] $H_0(s) \equiv (1/|s|)\sum_{\text{character } c} |n_c| \log(|s|/|n_c|)$, where n_c is the number of character c appearing in s.

[3] Unlike searching with the BWT, the pattern P is processed in the *forward* direction during the trie search. It is because we are essentially searching for those nodes whose reverse prefix strings begin with $P[p]P[p-1]\ldots P[1]$, so that we start by matching $P[1]$ first, and then $P[2]P[1]$, and so on.

[4] When the trie contains only one path, these lists are exactly the Ψ function of the compressed suffix arrays [10].

- the list for a: 0, 4, 5;
- the list for b: 0, 1, 4, 8;
- the list for c: 0, 8;

2. Construct an *indexable dictionary* on each of the list, so that for each list L, we can support the following operation in $O(1)$ time:
   - rank(x, L): If x is in L, report its rank among the other elements of L. Otherwise, report -1 indicating "x is not found in L".
3. Construct an auxiliary data structure such that for each character c, we can report in $O(1)$ time the value C$[c]$, which is the number of reverse prefixes lexicographically smaller than c.

Based on the above data structures, we can perform our desired query as follows. First, to check whether we can extend a node with rank x by the character c, we simply call rank(x, L_c) to check if the list L_c for c contains x, which requires $O(1)$ time. Next, suppose we can perform the extension. Then the rank of the desired node must be equal to $C[c] + rank(x, L_c)$, which also requires $O(1)$ time. Consequently, prefix matching can be performed in $O(|P|)$ time.

For the indexable dictionary, we use the result of [15], which takes $\log \binom{t}{n_c} + o(n_c) = n_c \log(t/n_c) + O(n_c)$ bits for the list for c, where n_c denotes the number of entries in the list. Note that n_c is exactly equal to the number of c in the XBW string X of $\mathcal{T}$. In total, the indexable dictionaries for all lists are stored in

$$\sum_{\text{character } c} n_c \log(t/n_c) + O(n_c) = tH_0(X) + O(t) \text{ bits.}$$

For the auxiliary data structure that supports computation of $C[c]$, it can be stored in $O(t + \sigma) = O(t)$ bits using Jacobson's constant-time rank and select index [13]. Thus, the overall space is bounded by $tH_0(X) + O(t)$ bits.

3.1. Compressing Belazzougui's scheme

We now show how to modify the above scheme so that the space becomes $tH_k(\mathcal{T}) + O(t)$, for any $k = o(\log_{\sigma+1} t)$.[5] Note that in any case, $H_k(\mathcal{T}) \leq H_0(X) \leq \log \sigma$. The key idea for the compression is to further divide the list into sublists, where each sublist contains the ranks of the reverse prefixes that begin with the same length-k string ρ; then, each sublist is encoded with a separate indexable dictionary. To see why compression can be achieved, we observe that the ranks in the sublist corresponding to ρ must be at least the rank of ρ (say x') among all the reverse prefixes, and can be at most $x' + |\text{cover}[\rho]| - 1$. Thus, we can store the rank x' of ρ, and replace each rank in the sublist with the difference with x'. As the number becomes smaller (between 0 and $|\text{cover}[\rho]| - 1$), we achieve compression. For instance, consider the example of Fig. 3 and the case where $k = 1$. Then the ranks 4, 5, 6, and 7 will be referred to as 0, 1, 2, 3, respectively, when we encode the sublist for $\rho =$ b.

More precisely, suppose that the sublist in the list c that corresponds to ρ contains $n_{c,\rho}$ ranks. Then the corresponding indexable dictionary of [15], which supports $O(1)$ time query, can be stored in at most

$$n_{c,\rho} \log(|\text{cover}[\rho]|/n_{c,\rho}) + O(n_{c,\rho}) \text{ bits,}$$

so that the sublist in all lists that correspond to ρ can be stored in at most

$$\sum_{\text{character } c} n_{c,\rho} \log(|\text{cover}[\rho]|/n_{c,\rho}) + O(n_{c,\rho}) = |\text{cover}[\rho]|H_0(\text{cover}[\rho]) + O(|\text{cover}[\rho]|) \text{ bits.}$$

Consequently, the total space of all sublists for all ρ's can be stored in at most

$$\sum_{\text{length-}k\ \rho} |\text{cover}[\rho]|H_0(\text{cover}[\rho]) + O(|\text{cover}[\rho]|) = tH_k(\mathcal{T}) + O(t) \text{ bits.}$$

The indexable dictionaries for the sublists are concatenated according to where it appears in the list. To facilitate the location of a particular dictionary, we use a conceptual bit-vector to mark the boundaries of the dictionaries. Such a bit-vector consists of $(\sigma + 1)^{k+1}$ 1's and $tH_k(\mathcal{T}) + O(t)$ 0's, which is stored with the *fully indexable dictionary* of [15] in $o(t)$ bits, and supports finding the desired dictionary for any sublist in $O(1)$ time. We can similarly store the rank of each ρ among the reverse prefixes by using a conceptual bit-vector with $(\sigma + 1)^k$ 1's and t 0's. The idea is to treat ρ as a $k \log \sigma$-bit integer, such that the number of 0's preceding the ρth 1 is exactly equal to the desired rank of ρ. The required space is $o(t)$ bits, and the desired rank can be reported in $O(1)$ time. Also, when given a rank x, we can compute in $O(1)$ time the number of 1's preceding the xth 0's; this corresponds to the rank of the string ρ whose sublist may contain x. Finally, we construct a table such that for each character c and each length-k string ρ, we can report in $O(1)$ time the value C$[c, \rho]$, which is the number of reverse prefixes lexicographically smaller than $c\rho$. The table contains $(\sigma + 1)^{k+1}$ entries, each taking $O(\log t)$ bits, so that the total space is bounded by $o(t)$ bits.

[5] With some minor adaptation, we can extend the range of k slightly to become $o(\log_\sigma n)$ instead.

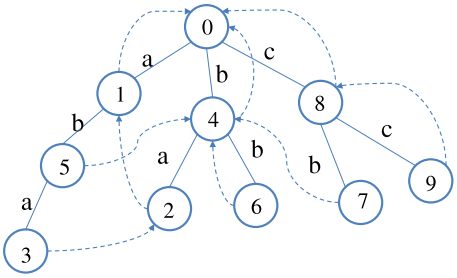


Fig. 4. An example of the failure function.

Given the above data structures, let us now examine how to perform the extension operation. Suppose we are given a node with rank x and a character c. First, we check which sublist of ρ that x may appear. Next, we check whether list c has a sublist corresponding to ρ. If not, we can conclude such an extension fails. Else, we retrieve the rank x' of ρ among the reverse prefixes, retrieve the indexable dictionary for the sublist $L_{c,\rho}$ of ρ within the list c, and check if $x - x'$ is stored in the sublist. If not, we can again conclude that the extension has failed. Else, we compute the desired rank r of the extended node as:

$$r = \text{rank}(x - x', L_{c,\rho}) + C[c, \rho].$$

So, we conclude in the following theorem.

Theorem 1. *We can construct an index to solve the prefix matching problem by using $tH_k(\mathcal{T}) + O(t)$ bits, and it supports prefix matching of a pattern P in $O(|P|)$ time.*

Proof. Our index contains the indexable dictionaries for the whole sublists, a bit-vector marking the boundaries of the dictionaries, a bit-vector storing the rank of each ρ among all the reverse prefixes, and a table for each character c and each length-k string ρ. So the total space of our index is $tH_k(\mathcal{T}) + O(t) + 3 \times o(t) = tH_k(\mathcal{T}) + O(t)$ bits.

For each step of extension operation, we first check which sublist of ρ that x may appear. We compute $rank_1(select_0(x))$ of the bit-vector which stores the rank of each ρ among all the reverse prefixes to get the rank of ρ. Then, we find the sublist by computing the bit-vector which marks the boundaries of the dictionaries. Furthermore, we compute the dictionary of the sublist. If there exists an extension for c, we compute $r = \text{rank}(x - x', L_{c,\rho}) + C[c, \rho]$ which is the next destination. Otherwise, there is no extension. Each step described on the above costs constant time, and the number of steps is also constant times. So each extension operation costs constant time to achieve. Then, we can conclude that our index supports prefix matching of a pattern P in the trie $\mathcal{T}$ in $O(|P|)$ time. □

4. Our index for compressed dictionary matching

The Aho–Corasick automaton [1] is an index for a set $\mathcal{D}$ of patterns that support dictionary matching query in $O(|T| + occ)$ time, where occ denotes the total number of occurrences of the patterns in T. The automaton consists of a trie $\mathcal{T}$ storing all the patterns in $\mathcal{D}$, together with two functions called *failure* and *report* that respectively facilitate the matching algorithm and the occurrence reporting. In particular, the failure function maps a node u to a node v such that its reverse prefix r_v is a proper prefix of r_u, and among all nodes v has the longest r_v. See Fig. 4 for an example, where the mapping from each node to its desired node is shown by the dotted arrows. For instance, the failure function for Node 3, with label aba, will be the node with label ba (i.e., Node 2). Note that if the latter node does not exist in the automaton, the failure function will be the node with label a instead (i.e., Node 1).

For the report function, it maps a node u to a node v such that (i) its reverse prefix r_v is a proper prefix of r_u, (ii) v itself corresponds to a pattern in $\mathcal{D}$ (that is, the reverse of r_v is a pattern in $\mathcal{D}$), and (iii) among all nodes v has the longest r_v. Note that the report function of a node u may not exist due to condition (ii). The space of the index is $O(t \log t)$ bits, where t is the number of nodes in $\mathcal{T}$.

In [3], Belazzougui showed that if each node u is represented with the rank of the reverse prefix r_u, then both the failure function and the report function can be encoded in $O(t)$ bits, and be computed in the same complexity as those in the uncompressed Aho–Corasick automaton. Indeed, Belazzougui's idea of encoding the failure function is very elegant: consider storing all the *reverse prefixes* of $\mathcal{T}$ in a compact trie $\mathcal{C}$, and marking each node in $\mathcal{C}$ that corresponds to a reverse prefix. Let u_C denote the marked node in $\mathcal{C}$ that corresponds to r_u. It is easy to check that the rank of r_u is exactly the pre-order rank of u_C among all the marked nodes in $\mathcal{C}$. For instance, the node corresponding to the pattern ba has label 2 in the Aho–Corasick automaton; consequently, the node corresponding to the reverse pattern ab will have pre-order rank equal to 2 among all the marked nodes in C. See Fig. 5 for an example.

Now, to compute the node v mapped by the failure function of u in $\mathcal{T}$, we simply locate u_C, find its lowest marked ancestor v_C (which corresponds to the desired node v in $\mathcal{T}$), and return the pre-order rank of v_C among all the marked nodes. To support the above computation, we need only to store the tree structure of $\mathcal{C}$ (which contains $O(t)$ nodes), a lowest marked ancestor data structure, and a data structure for returning the pre-order rank of a marked node. All these

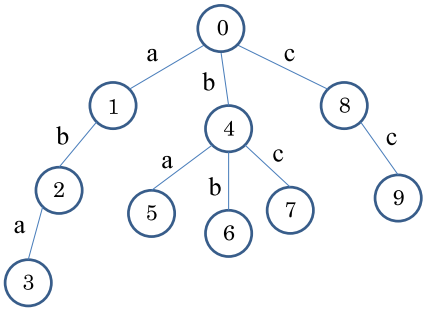


Fig. 5. An example of the compact trie of reverse prefixes. All nodes in this example are marked, and the label of a node shows its pre-order rank among all marked nodes. Note that in general some nodes in the compact trie may not be marked.

can be stored easily in $O(t)$ bits using the standard technique (see [3] for details) so that the above computation is done in $O(1)$ time.

As for the report function, Belazzougui showed that it can be represented by a tree with $d = |\mathcal{D}|$ internal nodes and $O(t)$ leaves so that it can be encoded in $O(d \log(t/d) + d) = O(t)$ bits, and the function can be computed in $O(1)$ time. Thus, by combining the above with the prefix matching index in Section 3, where each node can be extended by any character c in $O(1)$ time, dictionary matching can be performed as if we are using the uncompressed Aho–Corasick automaton. This gives the following theorem.

Theorem 2. *The Aho–Corasick automaton with a trie $\mathcal{T}$ can be stored in $tH_k(\mathcal{T}) + O(t)$ bits, such that for any query text T, the dictionary matching query can be performed in $O(|T| + occ)$ time.*

5. Conclusion

We have slightly modified Belazzougui's scheme for encoding an Aho–Corasick automaton so that the index space becomes entropy compressed, while it supports dictionary matching query in $O(|T| + occ)$ time. Note that our scheme and Belazzougui's scheme both contain an $O(t)$-bit space term; however, the hidden constant in our scheme is slightly larger. More precisely, if we ignore the index space for the failure and report functions, which are common in both schemes, then the hidden constant is 3 in our scheme (which comes from the XBW encoding) while the hidden constant is $\log_2 e \approx 1.443$ in Belazzougui's scheme (which comes from the zeroth-order compression of its auxiliary bit-vectors).

Although Belazzougui's scheme can successfully (and elegantly) solved the dictionary matching problem, it seems that the scheme cannot readily support dynamic operations, where a pattern $P[1..p]$ may be inserted to or deleted from the set $\mathcal{D}$ from time to time. If randomized amortized update is allowed, then we remark that we can apply the technique of [11] so that each update requires $O(p + t^{\epsilon})$ randomized amortized time, while query time remains $O(|T| + occ)$; here, $\epsilon > 0$ and t denotes the number of states in the Aho–Corasick automaton. A challenging open question is to support worst-case update operation, while keeping the search time as close to $O(|T| + occ)$ as possible.

References

[1] A. Aho, M. Corasick, Efficient string matching: an aid to bibliographic search, Communications of the ACM 18 (6) (1975) 333–340.
[2] A. Amir, M. Farach, Y. Matias, Efficient randomized dictionary matching algorithms (extended abstract), in: Proceedings of Symposium on Combinatorial Pattern Matching, 1992, pp. 262–275.
[3] D. Belazzougui, Succinct dictionary matching with no slowdown, in: Proceedings of Symposium on Combinatorial Pattern Matching, 2010, pp. 88–100.
[4] M. Burrows, D.J. Wheeler, A block-sorting lossless data compression algorithm, Technical Report 124, Digital Equipment Corporation, Paolo Alto, CA, USA, 1994.
[5] H.L. Chan, W.K. Hon, T.W. Lam, K. Sadakane, Compressed indexes for dynamic text collections, ACM Transactions on Algorithms 3 (2) (2007). Article No. 21.
[6] P. Ferragina, R. Grossi, A. Gupta, R. Shah, J.S. Vitter, On searching compressed string collections cache-obliviously, in: Proceedings of Symposium on Principles of Database Systems, 2008, pp. 181–190.
[7] P. Ferragina, F. Luccio, G. Manzini, S. Muthukrishnan, Compressing and indexing labeled trees, with applications, Journal of the ACM 57 (1) (2009). Article No. 4.
[8] P. Ferragina, G. Manzini, Indexing compressed text, Journal of the ACM 52 (4) (2005) 552–581.
[9] P. Ferragina, G. Manzini, V. Mäkinen, G. Navarro, Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms 3 (2) (2007).
[10] R. Grossi, J.S. Vitter, Compressed suffix arrays and suffix trees with applications to text indexing and string matching, SIAM Journal on Computing 35 (2) (2005) 378–407.
[11] A. Gupta, W.K. Hon, R. Shah, J.S. Vitter, A framework for dynamizing succinct data structures, in: Proceedings of International Colloquium on Automata, Languages and Programming, 2007, pp. 521–532.
[12] W.K. Hon, T.W. Lam, R. Shah, S.L. Tam, J.S. Vitter, Compressed index for dictionary matching, in: Proceedings of Data Compression Conference, 2008, pp. 23–32.
[13] G. Jacobson, Space-efficient static trees and graphs, in: Proceedings of Symposium on Foundations of Computer Science, 1989, pp. 549–554.
[14] E.M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM 23 (2) (1976) 262–272.
[15] R. Raman, V. Raman, S.S. Rao, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Transactions on Algorithms 3 (4) (2007). Article No. 43.

[16] A. Tam, E. Wu, T.W. Lam, S.M. Yiu, Succinct text indexing with wildcards, in: Proceedings of International Symposium on String Processing and Information Retrieval, 2009, pp. 39–50.
[17] P. Weiner, Linear pattern matching algorithms, in: Proceedings of Symposium on Switching and Automata Theory, 1973, pp. 1–11.
[18] I. Witten, A. Moffat, T. Bell, Managing Gigabytes: Compressing and Indexing Documents and Images, Morgan Kaufmann Publishers, Los Altos, CA, USA, 1999.