An Improved Query Time for Succinct Dynamic Dictionary Matching

Guy Feigenblat[1,2], Ely Porat[1], and Ariel Shiftan[1,⋆]

[1] Department of Computer Science, Bar-Ilan University, Ramat Gan 52900, Israel
[2] IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel
{feigeng,porately,shiftaa}@cs.biu.ac.il

Abstract. In this work, we focus on building an efficient succinct dynamic dictionary that significantly improves the query time of the current best known results. The algorithm that we propose suffers from only a $O((\log\log n)^2)$ multiplicative slowdown in its query time and a $O(\frac{1}{\epsilon}\log n)$ slowdown for insertion and deletion operations, where n is the sum of all of the patterns' lengths, the size of the alphabet is $polylog(n)$ and $\epsilon \in (0,1)$. For general alphabet the query time is $O((\log\log n)\log\sigma)$, where σ is the size of the alphabet.

A byproduct of this paper is an Aho-Corasick automaton that can be constructed with only a compact working space, which is the first of its type to the best of our knowledge.

1 Introduction

Traditional pattern matching and dictionary matching are fundamental research domains in computer science. In the pattern matching scenario, a text $T = t_1, ..., t_m$ and a pattern $P = p_1, ..., p_t$ are given, and the text is being searched for all of the occurrences of P in T. In the dictionary matching scenario, alternatively, a set of patterns $P_1, P_2, ..., P_d$ are indexed and given a text T, it is searched for all of the occurrences of the patterns in it. In the static case, the set of patterns is known in advance and cannot be changed, as opposed to the dynamic case, in which patterns can be added, deleted or updated.

Traditional dictionary matching algorithms, both static and dynamic [1–4], requires space, in words, that is proportional to the total length of the patterns stored in the structure. Assuming that the total length of the patterns is n, the space used by these algorithms is $O(n \log n)$; depending on the alphabet, this amount could be up to $O(\log n)$ times more than for the optimal algorithm, if we count the space used in bits rather than words. As the technology evolves and the amount of data that must be indexed becomes larger, researchers become interested in data structures that occupy space that is close to the information-theoretic lower bound without sacrificing the running time drastically. Grossi and Vitter [5] and Ferragina and Manzini [6] were pioneers in this field, and the field has became widely explored since their discoveries.

⋆ This research was supported by the Kabarnit Cyber consortium funded by the Chief Scientist in the Israeli Ministry of Economy under the Magnet Program.

A.S. Kulikov, S.O. Kuznetsov, and P. Pevzner (Eds.): CPM 2014, LNCS 8486, pp. 120–129, 2014.

Suppose that Z is the information-theoretic optimal number of bits that are needed to maintain some data D, and let σ be the size of the alphabet. An index for D is said to be succinct if it requires $Z+o(z)$ bits. It is said to be compact if it requires $O(Z)$ bits and compressed if it requires space proportional to the space of a compressed representation of D, as measured by its zeroth-order entropy, $H_0(T)$, or its k-th-order entropy, $H_k(T)$.

In this work, we focus on building an efficient succinct dynamic dictionary matching algorithm. Our succinct structure suffers from only a $O((\log\log n)^2)$ multiplicative slowdown in the query time which is much better than previous best known results of [7], where n is the sum of the lengths of all of the patterns and the alphabet size is poly-logarithmic in n. For insertion and deletion operations, which are less frequent, the bound becomes a bit worse, specifically $O(\frac{1}{\epsilon}\log n)$ multiplicative slowdown in time, for $\epsilon \in (0,1)$. As a byproduct, we show how an Aho-Corasick automaton can be constructed with only a compact working space, $O(n\log\sigma + d\log n)$ bits, which is the first of its type to the best of our knowledge. The construction is later being used as one of the building blocks of the dictionary matching algorithm.

For a comparison to previous results, refer to table 1.

Table 1. Comparison of dynamic dictionary matching algorithms, where n is the total length of d patterns, σ is the alphabet size and $\epsilon \in (0,1)$. Space is given in bits.

	query time	update time	space
[8]	$O((\lvert T\rvert + occ)\log^2 n)$	$O(\lvert P\rvert \log^2 n)$	$O(n\sigma)$
[7]	$O(\lvert T\rvert \log n + occ)$	$O(\lvert P\rvert \log\sigma + \log n)$	$O(n\log\sigma)$
[7]	$O(\lvert T\rvert \log n + occ)$	$O(\lvert P\rvert \log\sigma + \log n)$	$(1+o(1))n\log\sigma + O(d\log n)$
This paper	$O(\lvert T\rvert(\log\log n)\log\sigma + occ)$	$O(\frac{1}{\epsilon}\lvert P\rvert \log n)$	$(1+\epsilon)(n\log\sigma + 6n)+$ $o(n) + O(d\log n)$
This paper, for $\sigma = polylog(n)$, using $\epsilon = \frac{1}{\log\log n}$	$O(\lvert T\rvert(\log\log n)^2 + occ)$	$O(\lvert P\rvert \log n\log\log n)$	$(1+o(1))n\log\sigma + O(d\log n)$

1.1 Preliminaries

Let P be a set of d patterns $P = \{P_1, ..., P_d\}$ of cumulative size $n = \sum_{i=1}^{d} |P_i|$ over an alphabet Σ of size σ, and let T be a text of m characters. Furthermore, denote by occ the number of occurrences of the patterns from the set P in a given text T. For any two patterns, $P_i < P_j$ denotes that P_i is lexicographically smaller than P_j. Additionally, $(P_i)^r$ is the reversal of P_i. We assume a unit-cost RAM with word size $O(log\sigma)$ bits, in which standard arithmetic operations on word-sized operands can be performed in constant time.

1.2 Related Work

The research of dictionary matching algorithms goes back to the known Aho-Corasick automaton from 1975 [3]. Their algorithm is a generalization of the KMP automaton [9], which enables us to solve the static dictionary matching problem optimally using an $O((|T|+n)\log\sigma+occ)$[1] time algorithm. Amir et al. in [1] proposed the first dynamic algorithm for that problem. Their algorithm is based on generalized suffix trees, and they utilized the McCreight [10] construction algorithm to insert and remove patterns. Shortly afterward Amir et al. [4] introduced an alternative faster algorithm. In the core of this algorithm, there is a reduction between the dynamic marked ancestors problem to the balanced parentheses problem, which provides an efficient way to mark and un-mark nodes in a suffix tree. Assuming that P is a pattern that is removed or added to the dictionary, a dictionary update costs $O(P\frac{\log n}{\log\log n}+\log\sigma)$ which is $O(\log\log n)$ faster than the previous solution [1].

Sahinalp and Vishkin in [2] closed the gap on that problem and proposed the first optimal dynamic algorithm. Their algorithm uses a deterministic labeling technique called the LCP (Local Consistent Parsing) procedure and they show how to assign labels for the patterns in a way that allows a fast comparison of sub-strings. As a result, querying for a pattern's existence becomes $O(|T|+occ)$ (assuming $|P|\leq|T|$), and the insertion or deletion of pattern P costs $O(|P|)$.

All of the above algorithms use space in words, which is linear in the total length of the patterns, namely $O(n\log n)$ bits. Chan et al. [8] were the first to propose a compact (for constant alphabet size), $O(n\sigma)$ bit, data structure for a dynamic dictionary that is based on suffix trees. Their algorithm builds a dynamic suffix tree that uses $O(n\sigma)$ bits of space and supports the updating of a pattern P in $O(|P|\log^2 n)$ time and a query in $O((|T|+occ)\log^2 n)$ time. Their structure utilizes the FM index of [6] and the compressed suffix arrays of [5].

Hon et al. in [7] proposed the first succinct dynamic index for dictionary matching. Their solution is much simpler than [8], and it is based on a orderly sampling technique. Their key observation was that it is possible to reduce the space by grouping characters of a pattern into a single meta character. The space used is $(1+o(1))n\log\sigma+O(d\log n)$ bits, the query time is $O(|T|\log n+occ)$ and the update time of a pattern P is $O(|P|\log\sigma+\log n)$. Their dynamic solution was preceded by a compressed static construction for dictionary matching based on a similar approach [11].

Belazzougui in [12] showed how the Aho-Corasick automaton can be converted to a static succinct structure that occupies $n(\log\sigma+3.443+o(1))+d\log(\frac{n}{d})$ bits. His data structure provides the optimal query time of $O(|T|+occ)$, can be built in linear time, but not with a compact working space (the working space required is $O(n\log n)$ bits). The index will be described in the following sections since here we use similar states' representation. Hon et al. [13] showed how Belazzougui's index can be slightly modified to be stored in $nH_k(D)+O(n)$ bits, while the query time remains optimal.

[1] $O(|T|+n+occ)$ when using hashing.

1.3 Outline

The rest of the paper is organized as follows. In section 2 we present our algorithm for succinct dynamic dictionary matching. The algorithm utilizes the Aho-Corasick automaton construction that is sketched in section 3. Due to lack of space, the construction's details are left to the full paper.

2 Succinct Dynamic Dictionary Matching Algorithm

In this section, we propose a new succinct data-structure for the dynamic dictionary matching problem. This data-structure utilizes the compact construction of the succinct Aho-Corasick automaton described in section 3. The main contribution of this section is the reduction in query time compared to the best previous result of [7]. This reduction in query time incurs a slight increase in update (insertion or removal of patterns) time, however in most practical usages query time is more prominent. Specifically, the query time of the proposed algorithm is $O(|T| \log\log n \log\sigma + occ)$, which is $O(|T|(\log\log n)^2 + occ)$ for $\sigma = polylog(n)$, while the insertion or removal of a pattern P costs $O(\frac{1}{\epsilon}|P| \log n)$, for $\epsilon \in (0, 1)$. For a detailed comparison to previous results, refer to table 1.

2.1 Overview of the Technique

Our technique achieves the reduction in query time by using a specific division of the patterns' domain into groups based on length, and handling each group differently with a designated data-strcture. For simplicity, assume that the total length n is known and thus the division is well defined, later on we will describe how to cope with the changing of n. The hierarchy we define is divided into four groups, which are defined as follows:

1. Group XL (Extra Large): for patterns that are longer than or equal to $\frac{n}{\log\log n}$
2. Group L (Large): for patterns that are longer than or equal to $\frac{n}{\log n \log\log n}$ and shorter than $\frac{n}{\log\log n}$. This group will be further divided into $O(\log\log n)$ levels, based on patterns' length, in the following section.
3. Group M (Medium): for patterns that are longer than $0.5 \log_\sigma n$ and shorter than $\frac{n}{\log n \log\log n}$. When this group becomes full, in case the total size of the patterns in this group exceeds $\frac{n}{\log n \log\log n}$, we move them all together into group L.
4. Group S (Small): for patterns that are shorter than or equal to $0.5 \log_\sigma n$.

Notice that as our goal is to build a succinct dynamic data-structure, we have to make certain that the total size of the whole data structure is succinct. However, due to our hierarchy we do have some flexibility to maintain certain groups unsuccinctly relatively to the total size of the patterns in this group. Specifically, the total length of the patterns in the groups "Small" and "Medium" is relatively small, hence we are not restricted to use only succinct data-structures for maintaining the patterns, as opposed to the upper levels.

We use the above hierarchy as follows. On query, upon given a text T, we query each of the designated data-structures of these groups separately. On insertion, upon given a pattern P, we first make certain that the pattern does not exist in the whole data-structure, by running a query procedure for $T = P$; then, if it does not exist, the pattern is inserted directly to the data-structure of the group that matches its length. Similarly, on deletion, we first query to find the pattern among the groups, and then if it exists delete it from there. The exception for the deletion case is the group "Large", in which we don't remove the pattern immediately, but only mark it as removed, as will be described later.

In order to cope with the variability of n, we count the total length of the inserted and removed patterns and rebuild the whole data-structure once this size exceeds $\epsilon \cdot n$, for $\epsilon \in (0, 1)$; this step is to ensure that the patterns remain at the appropriate group despite the change in n. Also, this ensures that the extra space in the "Large" group, caused by patterns that are pending for deletion, does not exceed ϵ factor of the size of the structure. Next, we explain these procedures in detail for each group.

2.2 Extra Large Group

In this group, by definition, there can be at most $\log \log n$ patterns, which we maintain explicitly. We utilize the constant space KMP of [14] for each pattern separately. On query, we iterate over all the patterns in this group, hence the query time is $O(|T| \log \log n + occ)$. Insertion or deletion of a pattern P are trivial and take $O(|P|)$ time.

2.3 Large Group

Recall that patterns in this group are longer than or equal to $\frac{n}{\log n \log \log n}$ and shorter than $\frac{n}{\log \log n}$. We further divide the patterns in this group into $\log \log n$ levels $t_1, t_2, \ldots t_{\log \log n}$, with exponentially decreasing size. Denote the total length of the patterns in level t_i by $|t_i|$. The patterns are divided into levels such that for level t_1, $\frac{n}{2 \log \log n} \leq |t_1| < \frac{n}{\log \log n}$; for the second level $\frac{n}{4 \log \log n} \leq |t_2| < \frac{n}{2 \log \log n}$, and in general, $\frac{n}{2^{i+1} \log \log n} \leq |t_i| < \frac{n}{2^i \log \log n}$. A pattern in the i-th level can either be: (a) A pattern that was originally inserted to this level because its length was between $\frac{n}{2^{i+1} \log \log n}$ and $\frac{n}{2^i \log \log n}$; (b) A pattern that was moved into this level from the $(i+1)$-th level, when it was full; (c) Applies only to level $t_{\log \log n}$, a pattern that was moved into this level from group "Medium" when it became full.

The patterns in each level are maintained in a static Aho-Corasick automatons of theorem 3.1. In each level except for t_1 there are at most two such structures, and t_1 contains at most $\log \log n$ such structures. We call a level from $t_2, \ldots t_{\log \log n}$ full when it contains two automatons. But, we merge and move them to an upper level only when attempting to insert a third one, as described below.

Insert. When a new pattern P is inserted, we construct an Aho-Corasick automaton for that pattern, using theorem 3.1, and we insert it into a level chosen according to its size, i.e., to the level that contains patterns of total length in the same order of its size. If now there are three structures in this level, then we merge two of them by building a new Aho-Corasick automaton that contains all of the two automatons' patterns and we put the resulting structure into an upper level. We keep merging and moving in this fashion until there are less than three structures in the same level or we get to level t_1. Notice that in level t_1 we allow up to $\log\log n$ structures because it may happen that all the patterns reach this level (by merges or direct insertions).

Lemma 1. *Insertion of a pattern P into the "Large" group is done in $O(|P|\log n)$ amortized time, with an extra $o(n\log\sigma) + O(d\log n)$ bits of working space needed just for constructing the automatons.*

Proof. The given pattern P can potentially climbs to at most $\log\log n$ levels and therefore can be part of at most $\log\log n$ constructions of the Aho-Corasick automaton of theorem 3.1. Thus, the amortized insert cost is $O(|P|\log n)$ time. Assuming the total length of the merged patterns is n', according to theorem 3.1, the construction of such structure uses $O(n'\log\sigma) + O(d\log n')$ extra working space. In our case, we can bound n' with $\frac{n}{\log\log n}$, which is the total length of patterns in each structure of level t_1, therefore the extra working space needed during each construction is $o(n\log\sigma) + O(d\log n)$ bits. □

Query. On query, we iterate over all of the levels and search for the pattern. The total time is $O(T\log\log n\log\sigma + occ)$, because we have $\log\log n$ levels, and querying the Aho-Corasick automatons takes $O(T\log\sigma + occ)$ time according to theorem 3.1. For cases where $\sigma = polylog(n)$ the total time is $O(|T|(\log\log n)^2 + occ)$.

Delete. For deletion, we first run query operation in order to locate the level that contains the pattern; then, we "remove" the pattern from the Aho-Corasick automaton by un-marking it in $O(1)$ time (according to theorem 3.1). We defer the actual deletion to a later stage, ignoring the un-marked patterns when we rebuild the whole data structure.

The above results are summarized in following lemma:

Lemma 2. *A pattern P can be inserted to the "Large" group in $O(\frac{1}{\epsilon}|P|\log n)$ amortized time and marked for deletion in $O(1)$ amortized time. Querying the "Large" group for a text T takes $O(T\log\log n\log\sigma + occ)$ time, which is $O(|T|(\log\log n)^2 + occ)$ for cases where $\sigma = polylog(n)$. The space needed to accommodate the patterns in this group is at most $(1+\epsilon)(n\log\sigma + 6n) + o(n) + O(d\log n)$, for $\epsilon \in (0,1)$.*

Proof. The running time for Insert, Delete and Query directly follows the details described above and in lemma 1. As we do not completely delete patterns from this group, only un-mark them, we need to make certain that the extra space does not grow too much. Hence, we count the total length of the inserted and removed

patterns and rebuild the group once it exceeds $\epsilon \cdot n$, for $\epsilon \in (0,1)$; this ensures that we do not have more than $1+\epsilon$ factor in space. We use the structures from theorem 3.1 in each level, and each consumes $n' \log \sigma + 6n' + o(n') + O(d \log n')$ bits of space, where n' is the total length of the patters in it. In the "worst case" all the dictionary's patterns are located it this group, and therefore, by linearity, the total space consumption is at most $(1+\epsilon)(n \log \sigma + 6n) + o(n) + O(d \log n)$. □

2.4 Medium Group

We maintain patterns in this group in the dynamic dictionary by Sahinalp and Vishkin of [2][2]. The query time for a pattern is $O(|T| + occ)$, while the time for insertion and deletion is $O(|P|)$. Following the groups definition above, when the total size of the patterns in this group exceeds $\frac{n}{\log n \log\log n}$, we move them all together into group "Large". Hence, the space consumption is $O(\frac{n}{\log n \log\log n} \log \frac{n}{\log n \log\log n}) = O(\frac{n}{\log\log n}) = o(n)$.

2.5 Small Group

Because the total length of the patterns in this group is very small, the patterns can be stored permanently in a dictionary structure of [2], without the need to move up the hierarchy. Similarly to the analysis in [7], the overall space requires to store all the patterns is $o(n)$ because there can be at most $\Theta(\sqrt{n})$ such patterns, with the total length of $\Theta(\sqrt{n} \log_\sigma n)$ and total size of $\Theta(\sqrt{n}(\log_\sigma n)^2) = o(n)$ bits. The query, insert and delete operations are optimal in the size of the query or the pattern, respectively.

2.6 Putting it all Together

The construction is summarized in the following theorem:

Theorem 2.1. *There is a data-structure for the dynamic dictionary matching problem of d patterns with total length n, over alphabet of size σ, for $\epsilon \in (0,1)$, that uses $(1+\epsilon)(n \log \sigma + 6n) + o(n) + O(d \log n)$ bits of space. It supports querying all occurrences of the d patterns in a text T in $O(|T|(\log\log n)\log\sigma + occ)$ time, and insertion or deletion of a pattern P in $O(\frac{1}{\epsilon}|P| \log n)$ amortized time.*

Proof. Query, insertion and deletion are done on each group separately as was described above and in lemma 2. It takes optimal time to execute these operations on the "Small" and "Medium" groups, while on the "Extra Large" group there is a $\log\log n$ slowdown factor. Hence, the overall time bounds are dominated by the operations on the "Large" group. As for the space consumption, it is also dominated by the "Large" group. In addition, as we do not completely delete patterns from this group, we need to make certain that the extra space does not grow too much. Hence, we count the total length of the inserted and

[2] Any other non succinct algorithm that requires up to $O(\log\log n)$ query and update time can be used.

removed patterns and rebuild the whole data-structure once this size exceeds $\epsilon \cdot n$, for $\epsilon \in (0,1)$; this ensures that we do not have more than $1+\epsilon$ factor in space, and that the patterns remain at the appropriate group despite the change in n. Notice that we do not need to rebuild the "Large" group separately, since the whole data-structure rebuild described here is sufficient. □

Theorem 2.2. *There is a data-structure for the dynamic dictionary problem of d patterns with total length n, over alphabet of size $\sigma = polylog(n)$, that uses $(1+o(1))n\log\sigma + O(d\log n)$ bits of space. It supports querying all occurrences of the d patterns in a text T in $O(|T|(\log\log n)^2 + occ)$ time, and insertion or deletion of a pattern P in $O(|P|\log n\log\log n)$ amortized time.*

Proof. Directly Follows from theorem 2.1, and choosing $\epsilon = \frac{1}{\log\log n}$. □

3 Compact Construction of Succinct Aho-Corasick

As was described in section 2, the algorithm uses static Aho-Corasick automatons as building blocks, that have to be merged from time to time. The requirement for keeping the merge operation succinct is that the construction of the merged automaton is done by using only a compact working space, relative to the total size of the merged patterns. In this section we sketch such a construction and its main contribution is that it uses only a compact working space, which is the first of its type to the best of our knowledge. We further extend the automaton to support un-marking of patterns, i.e., terminal states can be removed from the report tree. The construction is summarized in theorem 3.1.

Theorem 3.1. *The Aho-Corasick automaton of d patterns, over an alphabet of size σ, that support un-marking of a pattern, can be constructed using $O(n\log\sigma)+O(d\log n)$ bits of space and can be compressed to consume $n\log\sigma + 6n + o(n) + O(d\log n)$ bits for maintaining it, where n is the number of states during the construction (before any deletion occurs). The total time for the construction is $O(\frac{n\log n}{\log\log n})$. Furthermore, this approach supports querying all of the occurrences of the d patterns in a text T in $O(|T|\log\sigma + occ)$ and un-marking a pattern in $O(1)$ amortized time.*

The data structure uses similar states' representation to as described by Belazzougui in [12], who showed how to construct static, succinct Aho-Corasick data-structure in linear time, that supports optimal query time. However, his construction uses more than compact working space, i.e. $O(n\log n)$ bits. As in Belazzougui's construction, we maintain three data-structures, namely, an indexable dictionary for the next transitions and two trees for the failure and report links. We modified the internal structures and the way they are built, and show how an Aho-Corasick automaton can be constructed using only a compact working space. In the rest of this section we first describe Belazzougui's idea, and then we sketch our construction. Due to lack of space the details are left to the full paper.

3.1 Belazzougui's Succinct Static Aho-Corasick Automaton

The Aho-Corasick automaton is a generalized KMP structure, which is essentially a trie with three types of links between internal states (nodes): next transitions, failure and report links. The next transitions links are the regular links of the trie. The failure link points to the longest suffix of the current pattern which is also a prefix of some other pattern in the trie. Similarly, the report link points to the longest suffix of the current pattern that is marked (the terminal state). These links allow a smooth transition between states without the need for backtracking.

Belazzougui showed how to represent each of the Aho-Corasick components using succinct data structures. His main observation was to name the states according to the suffix-lexicographic order of the concatenation of the labels on the edges of the trie. Specifically, the name of a state x, $state(x) \in [0, n-1]$, is the rank of the state in that order, where n is the number of states in the automaton. This arrangement has enabled him to maintain a failure tree that is ordered in states with increasing order. Assume that p_i is the i-th prefix in suffix lexicographic order, and let c_i be the label on the incoming edge in the trie that represents state p_i, s.t. $p_i = p'_i c_i$; then, the next transition is encoded using an indexable dictionary that maintains tuples of the form $(c_i, state(p'_i))$. In addition, the failure and report transitions are maintained in a succinct tree, such that a DFS traversal of the tree will enumerate the states in increasing order. The data-structure is encoded using $n(\log \sigma + 3.443 + o(1)) + d \log(\frac{n}{d})$ bits, where σ is the size of the alphabet and d is the number of patterns, and the query time is $O(|T| + occ)$. For cases where $\sigma < (n)^{\epsilon}$ for any $0 < \epsilon < 1$, the construction can be compressed to $n(H_0 + 3.443 + o(1)) + d(3 \log(\frac{n}{d}) + O(1))$ bits.

3.2 Sketch of the Construction

The main challenge was to build the tuples of the next transition dictionary in compact working space. Specifically, how to name each node in the trie of all patterns by its suffix lexicographic order, without maintaining a map of at least $n \log n$ bits. To overcome it, we use compressed suffix arrays that enable us to sort the suffixes efficiently utilizing a compact working space. To build the next transition dictionary, we start by sorting the patterns lexicographically, then we build a trie of the sorted patterns. Finally, we sort the prefixes in suffix lexicographic order by sorting the reversals of the patterns, and name each state in the trie according to this order. As for the failure links, we build them iteratively, in BFS order fashion, using the next transition dictionary. For each node, we use a standard technique to find its failure, as follows: visit the failure of its parent and check whether there is an outgoing edge with the same character. If so, this node is the failure; otherwise, continue recursively to the parent of the parent. We use the dynamic succinct tree of [15] to maintain the failure tree structure during the build. Eventually, we convert the failure links tree into a report tree, choosing the nearest marked ancestor (end of some pattern) as the direct parent of each node or the root in case there is no such node.

References

1. Amir, A., Farach, M., Galil, Z., Giancarlo, R., Park, K.: Dynamic dictionary matching. J. Comput. Syst. Sci. 49(2), 208–222 (1994)
2. Sahinalp, S.C., Vishkin, U.: Efficient approximate and dynamic matching of patterns using a labeling paradigm. In: FOCS, pp. 320–328. IEEE Computer Society (1996)
3. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM 18(6), 333–340 (1975)
4. Amir, A., Farach, M., Idury, R.M., La Poutré, J.A., Schäffer, A.A.: Improved dynamic dictionary matching. In: Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1993, pp. 392–401. Society for Industrial and Applied Mathematics, Philadelphia (1993)
5. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, STOC 2000, pp. 397–406. ACM, New York (2000)
6. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM 52(4), 552–581 (2005)
7. Hon, W.-K., Lam, T.-W., Shah, R., Tam, S.-L., Vitter, J.S.: Succinct index for dynamic dictionary matching. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1034–1043. Springer, Heidelberg (2009)
8. Chan, H.L., Hon, W.K., Lam, T.W., Sadakane, K.: Dynamic dictionary matching and compressed suffix trees. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, pp. 13–22. Society for Industrial and Applied Mathematics, Philadelphia (2005)
9. Karp, R.M., Miller, R.E., Rosenberg, A.L.: Rapid identification of repeated patterns in strings, trees and arrays. In: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, STOC 1972, pp. 125–136. ACM, New York (1972)
10. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM 23(2), 262–272 (1976)
11. Hon, W.K., Lam, T.W., Shah, R., Tam, S.L., Vitter, J.S.: Compressed index for dictionary matching. In: Proceedings of the Data Compression Conference, DCC 2008, pp. 23–32. IEEE Computer Society, Washington, DC (2008)
12. Belazzougui, D.: Succinct dictionary matching with no slowdown. In: Amir, A., Parida, L. (eds.) CPM 2010. LNCS, vol. 6129, pp. 88–100. Springer, Heidelberg (2010)
13. Hon, W.-K., Ku, T.-H., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster compressed dictionary matching. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 191–200. Springer, Heidelberg (2010)
14. Rytter, W.: On maximal suffices and constant-space linear-time versions of kmp algorithm. In: Rajsbaum, S. (ed.) LATIN 2002. LNCS, vol. 2286, pp. 196–208. Springer, Heidelberg (2002)
15. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010, pp. 134–149. Society for Industrial and Applied Mathematics, Philadelphia (2010)