Algorithmica (2017) 77:134–150
DOI 10.1007/s00453-015-0056-0

A Grouping Approach for Succinct Dynamic Dictionary Matching

Guy Feigenblat[1,2] · **Ely Porat**[1] · **Ariel Shiftan**[1,3]

Received: 16 June 2014 / Accepted: 17 August 2015 / Published online: 8 September 2015
© Springer Science+Business Media New York 2015

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.

Keywords Dictionary matching · Patterns matching · Online algorithms · Succinct · Algorithms · Compression · Aho Corasick

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

✉ Guy Feigenblat
feigeng@cs.biu.ac.il

Ely Porat
porately@cs.biu.ac.il

Ariel Shiftan
shiftaa@cs.biu.ac.il

[1] Department of Computer Science, Bar-Ilan University, 52900 Ramat Gan, Israel

[2] IBM Haifa Research Lab, Haifa University Campus, 31905 Haifa, Israel

[3] NorthBit, 5 Sapir St. Ampa Building, Entrance 5, Floor 1, 4685209 Herzliya, Israel

1 Introduction

Traditional pattern matching and dictionary matching are fundamental research domains in computer science. In the pattern matching scenario, a text $T = t_1, \ldots, t_m$ and a pattern $P = p_1, \ldots, p_t$ 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 $P_1, P_2, \ldots, P_d$ 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 become widely explored since their discoveries.

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 kth-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.

1.1 Preliminaries

Let P be a set of d patterns $P = \{P_1, \ldots, P_d\}$ 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

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

	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]	$O(\lvert T\rvert \log n + occ)$	$O(\lvert P\rvert \log \sigma + \log^2 n)$	$O(n \log \sigma)$
[7]	$O(\lvert T\rvert \log n + occ)$	$O(\lvert P\rvert \log \sigma + \log^2 n)$	$(1 + o(1))n \log \sigma + O(d \log n)$
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 $\epsilon = \frac{1}{\log\log n}$	$O(\lvert T\rvert(\log\log n)^2 + occ)$	$O(\lvert P\rvert \log n \log\log n)$	$(1 + o(1))n \log \sigma + O(d \log n)$

Space is given in bits

$O(\log \sigma)$ 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. [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 unmark 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]. The query takes $O(|T|(\frac{\log n}{\log\log n} + \log \sigma) + occ \cdot \frac{\log n}{\log\log n}$, which was improved to $O(|T|(\frac{\log n}{\log\log n} + \log \sigma) + occ)$ using an enhanced marked ancestor data structure [11].

Sahinalp and Vishkin [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 $O(|P|)$.

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

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 Ferragina and Manzini [6] and the compressed suffix arrays of Grossi and Vitter [5].

Hon et al. [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 [7,12].

Belazzougui [13] 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. [14] 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.

1.3 Outline

The rest of the paper is organized as follows. In Sect. 2 we present our algorithm for succinct dynamic dictionary matching. The algorithm utilizes the Aho-Corasick automaton construction that is sketched in Sect. 3, and detailed in Sect. 4.

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 Sects. 3 and 4. The main contribution of this section is the reduction in query time compared to the best previous result of Hon et al. [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 Rytter [15] 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, $\frac{n}{2\log\log n} \le |t_1| < \frac{n}{\log\log n}$; for the second level $\frac{n}{4\log\log n} \le |t_2| < \frac{n}{2\log\log n}$, and in general, $\frac{n}{2^{i+1}\log\log n} \le |t_i| < \frac{n}{2^i \log\log n}$. A pattern in the ith 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 4.2. 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 4.2, 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 4.2. Thus, the amortized insert cost is $O(|P|\log n)$ time. Assuming the total length of the merged patterns is n', according to Theorem 4.2, 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 4.2. 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 4.2). 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 4.2 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 Sahinalp and Vishkin [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] Any other non succinct algorithm that requires up to $O(\log \log n)$ query and update time can be used.

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 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 $\epsilon = \frac{1}{\log \log n}$. □

2.7 Generalizing Groups Medium and Small to Utilize Any Non-Succinct Algorithm

The strength of the approach described in this section is that it can be used to transform any non-succinct dynamic dictionary algorithm to a succinct one. Specifically, any dynamic dictionary algorithm can be used as a black box, for the Medium and Small groups, instead of the one by Sahinalp and Vishkin [2]. Assuming the memory footprint of the black box algorithm is $O(n \log^c n)$, for any constant c, the overall memory usage would still be succinct, by choosing the number of levels in the Large group to be $c \log \log n$. As to the query time, if the black box algorithm takes $f(n, |T|)$, the overall query time of the algorithm described in Theorem 2.1 becomes $f(n, |T|) + |T|(\log \log n) \log \sigma + occ$. Similarly, if the update time of the black box algorithm is $g(n, |P|)$, the overall update time becomes $O(\frac{1}{\epsilon}(g(n, |P|) + |P| \log n))$.

The Sahinalp and Vishkin's algorithm [2] heavily relies on a complicated renaming technique, and in some cases it requires hashing for fast trie traversal, hence an alternative algorithm may be preferred. The Amir et al. algorithm [4] with Alstrup et al.

improvement [11] is currently the next best solution in terms of query time, and can be used as a black box alternative here. Applying it, the query time is dominated by $O(|T|(\frac{\log n}{\log\log n}+\log\sigma)+occ)$, and the update time remains $O(\frac{1}{\epsilon}|P|\log n)$. The query time is still better than Hon et al. [7]. Furthermore, if dynamic hashing is permitted during updates the $\log\sigma$ term can be discarded. Potentially, any future non-succinct improved dynamic dictionary solution can be transformed into a succinct solution in a similar manner.

3 Compact Construction of Succinct Aho-Corasick

As was described in Sect. 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 4.2:

Theorem 4.2: *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 [13], 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 sketch our construction. The detailed construction is given later in Sect. 4.

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 ith 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 Sadakane and Navarro [16] 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.

3.3 Static Indexable Dictionary

Here we describe an indexable dictionary construction based on Elias-Fano encoding scheme [17–19]. We use it as a building block for the construction of the Aho-Corasick automaton described in Sect. 4.

An indexable dictionary is a data-structure that stores n elements, drawn from the universe U, and supports two operations: *Select(i)*—return the ith ranked element, and *Rank(k)*—return the rank of the element k. Based on the Elias-Fano encoding scheme

[17–19], we use an indexable dictionary, which can be constructed in one pass over the elements.

The data structure is composed of two parts. The first part stores the higher $\log n$ bits of the elements, while the second part stores the remaining $\log \frac{|U|}{n}$ bits. For the first part, we use the unary encoding (0 for the next number and 1 for an element) and store it in a rank/select bit vector, using $2n + o(n)$ bits. The lower bits are stored in the second part implicitly in a normal array in $n \log \frac{|U|}{n}$ bits. The select(i) operation can be answered by selecting the ith 1 bit and ranking the number of 0 bits up to that location. The result is simply the concatenation of those bits with the remaining bits of the ith element in the array. The total time is $O(1)$. The rank(k) query is answered by first finding the index of the first and last elements in the same block of k, where the blocks are defined by the $\log n$ higher bits of the elements. Specifically, we first select the ith and $(i + 1)$th 0 bits and rank the number of 1 bits up to these locations, which will give us the indexes of the first and last elements in the same block. Then, we use a binary search on the range to find the lowest bits of the element. The total time is, therefore, $O(\log \frac{|U|}{n})$. The above result is summarized in the following lemma:

Lemma 3 *For n elements drawn from a universe U, there exists an indexable dictionary that can be built in one pass over the elements, and that uses* $2n + o(n) + n \log \frac{|U|}{n}$ *bits of space during the construction phase and afterward. The construction time is linear in n, and it supports Select queries in* $O(1)$ *time and Rank queries in* $O(\log \frac{|U|}{n})$ *time.*

4 Aho-Corasick Construction in Details

We now show how to construct a static, succinct Aho-Corasick compatible data-structure. We maintain three data-structures, namely, an indexable dictionary for the next transitions and two trees for the failure and report links.

4.1 Next Transition Construction

1. *Sort the patterns lexicographically*. This task is accomplished by concatenating the patterns to generate $S_1 = P_1\#P_2\#\ldots\#P_d$, where the character $\# \notin \Sigma$ is smaller lexicographically than all of the other characters in Σ. Next, build from S_1 a compressed suffix array (CSA) as in [20], which takes $O(n \log\log\sigma)$ time and $O(n \log\sigma)$ bits of working space. Additionally, store a bit vector $BEGIN_{S_1}$ of size $(n + d)$ to mark the indexes where the patterns start in S_1, i.e., the indexes in S_1 where the first character of P_i appears, for $1 \leq i \leq d$, .
2. *Build a trie*. We want to insert the patterns into a trie in lexicographic order and to mark which of the patterns' characters was the cause of adding a new node to the trie. We call these characters "real", and the others "virtual". To represent the trie, use a list of parentheses that maintain the relationships between the nodes, and an array to map between the pre-order id of a node and the character on the incoming edge.

To build the trie, we iterate over the suffixes of S_1 in lexicographic order by querying the elements of the suffix array (SA) iteratively using the CSA structure, check whether the index is a beginning of a pattern using $BEGIN_{S_1}$, and add it to a trie as follows.

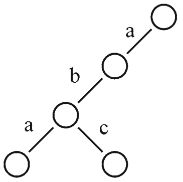
Start by inserting the first pattern in lexicographic order; for each character, we add an open parenthesis to the list, and the character itself to the array. Next, note that the LCP between any two consecutive patterns is the depth at which the path of these two patterns split in the trie. Hence, for the next patterns in lexicographic order, one of its prefixes might have already been added to the trie before, specifically the LCP between the pattern to the previous one. Therefore, to add the jth pattern ($1 < j \leq d$) in lexicographic order, i.e., pattern P_{t_j} where $P_{t_1} < P_{t_2} < \cdots < P_{t_d}$, we do the following:

(a) Find the LCP between $P_{t_{j-1}}$ and P_{t_j}.
(b) Add close parentheses according to the number of characters in $P_{t_{j-1}}$ that are not part of the LCP, to balance parentheses until the split node.
(c) Add open parentheses according to the number of characters in P_{t_j} that are not part of the LCP, for the new nodes.
(d) Add the characters from step c to the array ("real" characters).
(e) In case $j = d$ add close parentheses to balance the list.

We maintain another bit vector $REAL_{S_1}$ of size $n + d$, to indicate the indexes of the "real" characters.

The space used in this step is $2n$ bits for the parentheses and $n \log \sigma$ bits for the array of characters. The running time, dominated by the traversal over suffixes using the CSA; specifically, querying for the ith suffix costs $O(\log^{\epsilon} n)$ for any $\epsilon \in (0, 1)$, hence in total the time is $O(n \log^{\epsilon} n)$. Figure 1 shows an example of how the trie is constructed.

3. *Sort the prefixes in suffix lexicographic order*. Build $S_2 = P_{t_1}\#(P_{t_1})^r\ \#P_{t_2}\#(P_{t_2})^r\ \#\ldots\ P_{t_d}\#(P_{t_d})^r\#$, where the character $\# \notin \Sigma$ is smaller lexicographically than all of the other characters in Σ. Additionally, maintain a bit vector $REAL_{S_2}$ of size $2(n + d)$ to mark the "real" characters, where all of the characters in the reverse patterns and the # characters are treated as "virtual". Eventually, reverse S_2 into $(S_2)^r$, $REAL_{S_2}$ into $(REAL_{S_2})^r$, and build another CSA for $(S_2)^r$. The time and space of this step are dominated by the CSA construction.



	0	1	2	3	4	5	6
S1	a	b	c	#	a	b	a
BEGINs1	1	0	0	0	1	0	0
SA-1(S1)	3	5	6	0	2	4	1

Parenthesis list: ((( ) ( )))
↓ ↓
aba abc

Array:

0	1	2	3
a	b	a	c

Fig. 1 The trie for the set of patterns {"aba","abc"}, during the construction process, is composed of a parentheses list maintaining the trie structure, and an array mapping between the pre-order id of a node and the character on the incoming edge

4. *Name the automaton states according to the suffix lexicographic order*. Similar to [13], we name each state in the automaton according to its suffix lexicographic order.

   In the previous step we built CSA for $(S_2)^r$ to obtain the distinct prefixes of the original patterns $P_1 \ldots P_d$ in suffix lexicographic order. The reason that each pattern in S_2 is concatenated to its reversal is to ensure that the order of the prefixes is the same as the order used when building the trie in step 2; hence, we know that "real" characters of similar prefixes will appear before any "virtual" characters of the same prefix.

   Thus, we iterate lexicographically over the suffixes of $(S_2)^r$ using the CSA and reorder the bit vector $(REAL_{S_2})^r$ into a new bit vector G according to the lexicographic order, s.t. the ith bit specifies whether the ith lexicographic order index in $(S_2)^r$ is of a "real" character. We preprocess G to allow constant time rank queries as in [21]. As a result, one can obtain the state name of any index in S_2 by first using the CSA to obtain its lexicographic order; then, filter out "virtual" states with a rank operation on G and obtain the state name. The space used in this step is $O(n)$ bits, and the time is $O(n)$. Refer to Fig. 2 for an illustrative example.
5. *Build the next transition dictionary*. Recall that the next transitions are encoded using an indexable dictionary maintaining tuples of the form $(c_i, state(p_i))$, where $c_i \in \Sigma$ is the label on the incoming edge, and $state(p) \in [0, n-1]$ is the name of the previous state in the trie. To build the tuples, we now iterate over the CSA of $(S_2)^r$, for each suffix; utilizing $(REAL_{S_2})^r$, we first check if it is a "real" character. If so, it should be added to the dictionary, and it is left to find the name of the state preceding it. The previous state is represented by the previous character in S_2, which is the next character in $(S_2)^r$. We use the CSA to obtain the lexicographic order of the suffix starting at that character, and we obtain the real state name with a rank operation on G. Finally, during the iteration, we insert each tuple into a succinct static dictionary using Lemma 3, so that we will not maintain all of them explicitly during this phase. The space used in this step is dominated by the indexable dictionary, because we store n tuples from the domain $(\Sigma, [n])$, the space consumption is $2n + o(n) + n \log \sigma$. The overall time is dominated by the iterations of suffixes by using the CSA—$O(n \log^{\epsilon} n)$. Figure 2 shows an example of the various structures used during the build of the next transitions.

The following lemma summarizes the next transition construction:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
S2	a	b	c	#	c	b	a	#	a	b	c	a	#	a	c	b	a	#
REAL$_{S2}$	1	1	1	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
(S2)^r	#	a	b	c	a	#	a	c	b	a	#	a	b	c	#	c	b	a
SA^{-1}((S2)^r)	1	8	13	15	6	2	9	17	11	5	0	7	12	14	3	16	10	4
(REAL$_{S2}$)^r	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1	1	1
G	0	0	0	0	1	0	0	0	0	1	1	0	0	0	0	0	1	0

Fig. 2 The internal structures used during the construction of the next transition, for the set of patterns {"abc","abca"}. The computed dictionary tuples are $(a, 0)$, $(a, 4)$, $(b, 1)$, $(c, 3)$. For example, the second tuple, $(a, 4)$, is computed when iterating over the 9th suffix in $SA^{-1}((S2)^r)$. The first character of this suffix is 'a', and the next character in $S2^r$, which is the previous character in $S2$, is 'c'. 'c' is the first character of the 17th suffix and has rank 4 in G

Lemma 4 *The next transitions of n states, over an alphabet of size σ, can be constructed using $O(n \log \sigma)$ bits of working space and can be compressed to consume $2n + o(n) + n \log \sigma$ bits afterward. The total time for this construction is $O(\frac{n \log n}{\log \log n})$, and the data-structure supports forward navigation in $O(\log \sigma)$ and backward navigation in $O(1)$.*

Proof We use the above construction to build an indexable dictionary of Lemma 3 for the next transitions of n states. The total construction time is $O(n \log \log \sigma + n \log^{\epsilon} n)$, by choosing $\epsilon < 0.25$ it can be bound by $O(\frac{n \log n}{\log \log n})$. The dictionary contains tuples of the form $(c_i, state(p_i))$, where $c_i \in \Sigma$ is the label on the incoming edge, and $state(p) \in [0, n-1]$ is the name of the previous state in the trie. As the relative order of a tuple in the indexable dictionary is the state name, navigation forward is simply performed by Rank query, which takes $O(\log \sigma)$. Furthermore, navigation backward is performed in $O(1)$ time using Select query, by extracting the previous state name from the tuple. □

4.2 Failure Links

We build the failure links in BFS order fashion, using the above next transition dictionary. The next transition dictionary is essentially a trie of patterns. We build the failure tree iteratively over the trie, in level order, such that when handling a node v, its parent in the trie w must already be in the failure tree. Assuming the character on the edge between v and w is $c \in \Sigma$, we check whether there is an outgoing edge with character c from the failure of w. If so, this node is the failure; otherwise, continue recursively to the parent of the parent. We use the dynamic succinct tree of [16] to maintain the failure tree structure during the build. It remains to tackle two issues — mapping between state names during the build and finding the children of a node:

1. *Mapping between state names during the build*. Recall that the name of each state represents the index in the indexable dictionary and the pre-order enumeration id in the failure tree. As we build the failure tree, the states are partial; hence, their pre-order enumeration ids do not match the dictionary which holds the entire states. To overcome this problem we utilize a bit vector of size n initially set 0. When a state $i \in [0, n-1]$ is added to the failure tree, we flip the ith bit to 1. We maintain a dynamic rank/select bit vector (from [16]) to map evolving partial states to the domain of all states.
2. *Finding the children of a node*. Because we cannot use the indexable dictionary to locate the next child of a node in constant time, we utilize another auxiliary structure. During the creation of G as part of the next transition, we create an array of size $n \log \sigma$ that maintains the matching previous character in $(S_2)^r$ for each index. Using this structure along with G, we can iterate over the children in a constant amount of time per child.

Eventually, we convert the tree into the static structure of Sadakane and Navarro [16], which supports parent lookup in constant time, and has a compact space usage during the build.

Lemma 5 *The failure links of d patterns and n states, over an alphabet of size σ, can be constructed using $n\log\sigma + O(n) + d\log n$ bits of space and can be compressed to consume $2n + o(n)$ bits afterward. The total time for the construction is $O(\frac{n\log n}{\log\log n})$, and the data-structure supports locating the failure of a state in $O(1)$ time.*

Proof We use the above construction and build a static succinct tree (from [16]) that represents the failure links. Creation of the auxiliary structure takes $n\log\sigma$ bits, the BFS traversal consumes $d\log n$ bits, and the conversion to the static structure requires $O(n)$ bits. The static structure of Sadakane and Navarro [16] consumes $2n + o(n)$. The time is dominated by the usage of the dynamic succinct tree—$O(\frac{n\log n}{\log\log n})$. □

4.3 Report Links

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. Furthermore, we make certain that the pre-order rank of each node is preserved. Note that each node in this tree is either a leaf or a marked node and that this step can be performed compactly in one pass over the failure tree while utilizing a dynamic tree of Sadakane and Navarro [16]. Theoretically, this arrangement could be sufficient; however, it would slow our query time by a factor of $O(\frac{\log n}{\log\log n})$. Therefore, we eventually convert the tree into a static one (in [16]) to allow a constant time for the parent lookup.

The overall space usage during this step is $O(n)$ bits, and the time is $O(\frac{n\log n}{\log\log n})$, which is dominated by the usage of the dynamic succinct tree.

Lemma 6 *The report links of d patterns and n states, over an alphabet of size σ, can be constructed using $O(n)$ bits of space and can be compressed to consume $2n + o(n)$ bits afterward. The total time for the construction is $O(\frac{n\log n}{\log\log n})$, and the data-structure supports iterating the report links in $O(1)$ time per reported state.*

Proof We use the above construction to build the report links. We traverse the nodes of the failure tree in a DFS fashion, and insert each node to the report tree as the direct right most (last) child of its nearest market ancestor in the failure tree. This make certain that the DFS order of the report tree is the same as of the failure tree. During the traversal we use the dynamic tree of Sadakane and Navarro [16], and upon completion we covert the dynamic tree to a static one of Sadakane and Navarro [16]. As a result, this activity requires $O(n)$ bits and $O(\frac{n\log n}{\log\log n})$ time, and iterating the report link can now be accomplished in $O(1)$ time per reported state by parent lookup in the static tree. □

We summarize the compact construction of the succinct Aho-Corasick automaton, which is represented by the three above structures with the following theorem:

Theorem 4.1 *The Aho-Corasick automaton of d patterns and n states, over an alphabet of size σ, can be constructed using $O(n\log\sigma) + d\log n$ bits of space and can be compressed to consume $n\log\sigma + 6n + o(n)$ bits for maintaining it. 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 $O(|T|\log\sigma + occ)$ time.*

Proof Applying Lemmas 4, 5 and 6 provides the desired result. □

4.4 Enhanced Report Links

The report links tree, as being part of the Aho-Corasick automaton, is static. We would like to partially dynamize it by supporting un-marking of marked nodes (terminal states), instead of completely removing them from the tree. This will allow us to query the Aho-Corasick automaton without traversing patterns that are no longer marked, which logically deletes the pattern without essentially removing it. Since we only need to support un-marking of a node, this reduces the problem to the decremental marked ancestor problem [11,22], which can be solved in $O(1)$ amortized time for un-marking and querying. However, maintaining all the nodes of the report tree in this structure consumes $O(n\log n)$ bits of space. Nevertheless, one can observe that we do not really need to maintain all the nodes, since that in the report tree there are only d marked nodes, and each internal node must be marked. Therefore, it suffices to maintain only the d marked nodes in a structure that supports un-marking of nodes. Then, one can use the report tree to find the original marked ancestor of a node, and locate the updated one by using the above structure. Overall we use $O(d\log n)$ bits to keep relationship of the d patterns (i.e. the d marked nodes). Although some leaves's marked ancestors are no longer marked ancestor, but that would just affect only one traversal time. Thus, the reporting time is still $O(occ)$ time.

Theorem 4.2 *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.*

Proof Applying Theorem 4.1, and adding the data-structure from [11,22] to allow un-marking internal nodes from the original report links tree, provides the desired result. □

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, IEEE Computer Society, pp. 320–328 (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., Lapoutre, J.A., Schaffer, A.A.: Improved dynamic dictionary matching. Inf. Comput. **119**(2), 258–282 (1995)

5. Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM J. Comput. **35**(2), 378–407 (2005)
6. Ferragina, P., Manzini, G.: Indexing compressed text. J. ACM **52**(4), 552–581 (2005)
7. Hon, W.K., Ku, T.H., Lam, T.W., Shah, R., Tam, S.L., Thankachan, S.V., Vitter, J.S.: Compressing dictionary matching index via sparsification technique. Algorithmica **72**(2), 515–538 (2014)
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 '05, Philadelphia, PA, USA, pp. 13–22. Society for Industrial and Applied Mathematics (2005)
9. Knuth, D.E., Morris, J., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
10. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–272 (1976)
11. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proceedings of the 39th Annual Symposium on Foundations of Computer Science, FOCS '98, Washington, DC, USA, p. 534. IEEE Computer Society (1998)
12. 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 '08, Washington, DC, USA, pp. 23–32. IEEE Computer Society (2008)
13. Belazzougui, D.: Succinct dictionary matching with no slowdown. In: Proceedings of the 21st Annual Conference on Combinatorial Pattern Matching, CPM'10, pp. 88–100. Springer, Berlin (2010)
14. Hon, W.K., Ku, T.H., Shah, R., Thankachan, S.V., Vitter, J.S.: Faster compressed dictionary matching. In: Proceedings of the 17th International Conference on String Processing and Information Retrieval, SPIRE'10, pp. 191–200. Springer, Berlin (2010)
15. Rytter, W.: On maximal suffices and constant-space linear-time versions of kmp algorithm. In: Rajsbaum, S. (eds.) LATIN. Volume 2286 of Lecture Notes in Computer Science, pp. 196–208. Springer (2002)
16. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '10, Philadelphia, PA, USA, pp. 134–149. Society for Industrial and Applied Mathematics (2010)
17. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM **21**(2), 246–260 (1974)
18. Fano, R.: On the Number of Bits Required to Implement an Associative Memory. Computation Structures Group Memo. MIT Project MAC Computer Structures Group (1971)
19. Grossi, R., Orlandi, A., Raman, R., Rao, S.S.: More haste, less waste: lowering the redundancy in fully indexable dictionaries. CoRR arXiv:0902.2648 (2009)
20. kai Hon, W., Sadakane, K., kin Sung, W.: Breaking a time-and-space barrier in constructing full-text indices. In: Proceedings of 44th Annual Symposium on Foundations of Computer Science, pp. 251–260. IEEE (2003)
21. Munro, J.I.: Tables. In: FSTTCS. pp. 37–42 (1996)
22. Gabow, H.N., Tarjan, R.E.: A linear-time algorithm for a special case of disjoint set union. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, STOC '83, New York, NY, USA, pp. 246–251. ACM (1983)

Algorithmica (2017) 77:151
DOI 10.1007/s00453-016-0183-2

ERRATUM

Erratum to: A Grouping Approach for Succinct Dynamic Dictionary Matching

Guy Feigenblat[1,2] · Ely Porat[1] · Ariel Shiftan[1,3]

Published online: 12 July 2016
© Springer Science+Business Media New York 2016

Erratum to: Algorithmica DOI 10.1007/s00453-015-0056-0

This research was carried out as part of the PhD thesis of Guy Feigenblat under supervision of Professor Ely Porat from Bar-Ilan University.

The online version of the original article can be found under doi:10.1007/s00453-015-0056-0.

✉ Guy Feigenblat
feigeng@cs.biu.ac.il

Ely Porat
porately@cs.biu.ac.il

Ariel Shiftan
shiftaa@cs.biu.ac.il

[1] Department of Computer Science, Bar-Ilan University, 52900 Ramat Gan, Israel

[2] IBM Haifa Research Lab, Haifa University Campus, 31905 Haifa, Israel

[3] NorthBit, 5 Sapir St. Ampa Building, Entrance 5, Floor 1, 4685209 Herzliya, Israel