Linear Time Succinct Indexable Dictionary Construction With Applications

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

[1] Department of Computer Science Bar-Ilan University, Ramat Gan 52900, Israel
[2] IBM Research - Haifa, Haifa University Campus, Haifa 3498825, Israel
[3] NorthBit, Herzliya, Israel
{feigeng, porately, shiftaa}@cs.biu.ac.il

Abstract. Indexable dictionaries, supporting rank and select queries, are used as building blocks for many algorithms. For a universe $U = \{0, ..., |U| - 1\}$ and an ordered set $S = \{s_0, ..., s_{n-1}\} \subseteq U$, an indexable dictionary supports rank and select queries in addition to membership queries; Select(j) query is used to get the j'th ranked element, and Rank(x) is used to retrieve the rank of x among all elements in S.
In this work, we give two time-linear, one-pass practical constructions of static succinct indexable dictionaries; both are deterministic in query time, but they differ by construction method and Rank(x) query time. The first supports Rank and Select queries in constant time and has expected linear construction time. The second supports Select queries in constant time, and Rank(x) queries in $O(\log\log\frac{|U|}{n})$ time, has worst-case linear construction time, and uses only $o(n)$ additional bits during construction. The latter one is fully indexable dictionary supporting Rank(x) queries on arbitrary x. These indexable dictionaries can be used where construction bounds matter, as in a dynamic algorithm that uses them as a building block; we exemplify this by showing how to utilize them to improve the query time of a dynamic dictionary matching algorithm.

1 Introduction

In the static dictionary problem, a set of n elements drawn from a universe U is maintained in a data structure that supports membership queries. In some cases, satellite data is associated with the elements in the dictionary and can be retrieved when querying an element. Formally, consider a universe $U = \{0, ..., |U| - 1\}$ and an ordered set $S = \{s_0, ..., s_{n-1}\} \subseteq U$ of size n. When $x \in U$ is queried, the dictionary returns whether $x \in S$ and possibly the data associated with it. An indexable dictionary (ID) supports two additional operations:

- Select(j) - for $0 \leq j \leq n-1$, return s_j, the j-th ranked element in S.
- Rank(x) - for $x \in U$, return $|\{s_i : s_i \leq x\}|$ if $x \in S$ and -1 otherwise.

A fully indexable dictionary (FID) supports rank operation on an element $x \notin S$ and returns $|\{s_i : s_i \leq x\}|$, which can be extended to a predecessor query by performing Select(Rank(x)). Traditional dictionary data structures require space proportional to the number of elements that are stored in them. Assuming there are n elements, space consumption becomes $O(n \log |U|)$ bits. Nevertheless, because there are $\binom{|U|}{n}$ possible dictionaries of n elements, the information-theoretic optimal number of bits needed is $B(|U|, n) = \lceil \log \binom{|U|}{n} \rceil = n \log \frac{|U|}{n} + O(n)$. Therefore, the traditional representation is inefficient, and alternative analysis models are considered. Suppose that Z is the information-theoretic optimal number of bits that are needed to maintain data D. 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

1068-0314/16 $31.00 © 2016 IEEE
DOI 10.1109/DCC.2016.70

and compressed if it requires space proportional to the space of a compressed representation of D, as measured by its zeroth-order entropy, $H_0(D)$, or its k-th-order entropy, $H_k(D)$. Applying this to the dictionary problem, a succinct representation can use up to $B(|U|, n) + o(B(|U|, n))$ bits.

1.1 Our Contribution

In this paper, we give two time-linear, one-pass constructions of static succinct indexable dictionaries. Both are deterministic in query time, but they differ in construction method and Rank(x) query time. The first construction described in Theorem 3.1 supports Rank(x) and Select(j) queries in constant time. The expected construction time is $O(n)$, and the working space during the construction is $O(n \log \frac{|U|}{n})$ bits. The second construction described in Theorem 4.1 supports Select(j) queries in constant time, and Rank(x) queries in $O(\log \log \frac{|U|}{n})$ time. The worst-case construction time is $O(n)$, and the extra working space needed during construction is $o(n)$ bits. In addition, this dictionary supports Rank queries on $x \notin S$, hence it is a fully indexable dictionary. In both cases, it is assumed that the elements are given in sorted order.

To the best of our knowledge, this is the first work that gives a construction of a static succinct indexable dictionary with bounded, linear construction time in compact space, and still provides constant time queries. Hence, it can be utilized where construction bounds matter, as in a dynamic algorithm that uses an indexable dictionary as a building block.

We exemplify the importance of these results by showing how to improve a solution to a fundamental problem in the pattern-matching domain: the succinct dynamic dictionary matching problem. In the pattern-matching scenario, a text T and a pattern P, both over an alphabet of size σ, are given, and the text is searched for all occurrences of P. In the dictionary matching scenario, alternatively, a set of patterns $P_1, P_2, ..., P_d$ of total size m are indexed, and all occurrences of the patterns are searched for in a given text T. In the static case, the set of patterns is known in advance and cannot be changed, whereas in the dynamic case, patterns can be added, deleted, or updated. Utilizing the described indexable dictionaries, query time is reduced from $O(|T| \log \log m \cdot \log \sigma + occ)$ to $O(|T| \log \log m + occ)$, which slows the succinct solution by a factor of $\log \log m$ compared to the optimal non-succinct solution. The results are summarized in table 5.1. A byproduct from the usage of the indexable dictionary in the dynamic dictionary matching algorithm, is an Aho-Corasick automaton that can be constructed with only a compact working space and supports querying a text T in optimal time: $O(|T| + occ)$.

1.2 Related Work

Indexable dictionaries go back to Elias [1] and Fano [2] who proposed an encoding scheme that could be extended to an indexable dictionary. Raman and Rao [3] proposed a static dictionary supporting membership and rank queries in constant time using $n \log |U| + O(\log \log |U|)$ bits. Later, in a seminal paper, Raman et al. [4] described a succinct indexable dictionary structure for the RAM model that supported all queries in constant time. The space consumption was shown to be $B(|U|, n) + o(n) + O(\log \log |U|)$ bits. In addition, a fully indexable dictionary was shown that consumed $B(|U|, n) + O(\frac{|U| \log \log |U|}{\log |U|})$ bits. In the conclusion section the authors mentioned it may be possible to construct the dictionary in linear time on a sorted input, however they didn't provide a respective algorithm. In addition, the working space, which is important for dynamization, wasn't considered. In this paper, we achieve similar bounds for an indexable dictionary implementation, taking into account the construction time and space.

Gupta et al. [5] presented a compressed representation for both an indexable dictionary and a fully indexable dictionary with space bounds that depended on the gap measure, typically much less

than $\log\binom{|U|}{n}$ bits. The fully indexable dictionary structure consumed $gap + O(n\log\frac{|U|}{n}/\log n) + O(n\log\log\frac{|U|}{n})$ bits while the select query took $O(\log\log n)$ time; the rank query was a bit slower. The indexable dictionary structure consumed the same amount of space but supported both operations in $O(\log\log n)$ time. When considering worst-case analysis, their data-structure was not succinct and consumed $O(n\log\frac{|U|}{n})$ bits. Pătraşcu [6] showed that it is possible to represent a fully indexable dictionary using $\log\binom{|U|}{n} + \frac{|U|}{(\log|U|/t)^t} + \tilde{O}(|U|^{3/4})$ bits of memory, for a predefined t, supporting Rank and Select queries in $O(t)$ time.

Belazzougui et al. [7] described a monotone minimal perfect hash scheme that mapped a set of n keys into the set $\{0, ..., n-1\}$ bijectively. They showed that $O(n\log\log\log|U|)$ bits were sufficient to maintain such hash function with an evaluation time of $O(\log\log|U|)$. Alternatively, it was possible to use $O(n\log\log|U|)$ bits to maintain such a function with constant evaluation time. Notice that when utilizing such a function and maintaining the original keys in a table, both rank and select operations can be answered in constant time using $n\log|U| + O(n\log\log|U|)$ bits of space. The term $n\log|U|$ can be further reduced by using Elias' encoding [1] in a manner similar to that described here in Lemma 2.1. However, this does not yield a succinct indexable dictionary, because in many cases $O(n\log\log|U|)$ is not small enough. Specifically, the term is not succinct for large values of n, when $n \geq O(\frac{|U|}{\log|U|})$.

1.3 Definitions and Notations

The dictionary maintains n elements, drawn from a universe of integers $U = \{0, \ldots, |U|-1\}$ where $n << |U|$, each of size $w = \log|U|$. A unit-cost RAM model with word size $O(w)$ bits, in which standard arithmetic operations on word-sized operands can be performed in constant time, is assumed. Unless specified otherwise, logarithmic operations are always base 2. The symbol | indicates the binary concatenation operand (e.g. $001|110 = 001110$). In addition, e represents the natural number ($e = 2.718\ldots$).

A fundamental tool of Rank and Select queries will be used on bit vectors throughout this paper. For a bit array $V[0\ldots n-1]$ of length n, these operations are defined as follows: $rank_b(1, i, V)$ counts the number of 1 bits in bit vector V up to position i, and $select_b(1, j, V)$ returns the position of the j-th 1 bit in bit vector V. For example, consider the bit vector 0100011, $rank_b(1, 3, V) = 1$, $rank_b(0, 3, V) = 3$, and $select_b(1, 2, V) = 5$. By pre-processing the bit vectors in linear time and allocating an additional $o(n)$ bits [8, 9], these operations can be performed in constant time.

2 Preliminary Algorithm

In this section we describe a succinct indexable dictionary that will be used as a building block in sections 3 and 4. This dictionary will support the operations Rank(x) in $O(\log\frac{|U|}{n})$ time, and Select(j) in constant time. To do so, the binary representation of each element x is split into two parts, x_h and x_l, such that $x = x_h|x_l$. The first part, x_h, stores the higher $\log n$ bits of the elements while x_l stores the remaining $\log\frac{|U|}{n}$ bits. This is similar to the Elias-Fano encoding scheme [1, 2, 10].

Handling x_h: Storing the x_h of all elements is equivalent to storing n sorted integers, drawn from a universe $[0, \ldots, n-1]$, which can be maintained in a bit vector of size $2n$ using unary encoding. The bit vector is built by iterating over the integers in an increasing order and checking if the current integer (denoted by a) equals the previous one (denoted b). If it is equal, one 1-bit is appended to the vector, otherwise $a - b$ 0-bits are appended. The resulting bit vector is denoted by Ψ.

The above representation can be considered a logical division of the original universe U into n blocks. Specifically, the 0-bits represent block boundaries, and the 1-bits (between any two 0-bits) represent elements in the block. Because n elements have been logically divided to n blocks, some blocks may be empty and some may contain more than one element. Notice that the higher $\log n$ bits of an element, x_h, indicate the block's index and therefore the association to a block. Blocks are denoted by $B_0, \ldots, B_i, \ldots B_{n-1}$, and the block size, the number of elements associated with block B_i is denoted by $|B_i|$, such that $\sum_{i=0}^{n-1}(|B_i|) = n$.
Three operations can be performed on the bit vector Ψ:

1. ***num_elements***: The number of elements in block B_i such that $num\_elements(i, \Psi) = select_b(0, i+1, \Psi) - select_b(0, i, \Psi) - 1$.
2. ***first_index***: The index of the first element in the non-empty block B_i such that $first\_index(i, \Psi) = rank_b(1, select_b(0, i, \Psi) + 1, \Psi)$.
3. ***block_number***: The block number of the j-th element such that $block\_number(j, \Psi) = rank_b(0, select_b(1, j, \Psi), \Psi)$.

Handling x_l: The x_l parts of all elements are n integers, each of size $\log \frac{|U|}{n}$ bits. In a manner similar to that previously described, each block B_i for $i \in [0, n-1]$ is further divided into $|B_i|$ sub-blocks such that the higher $\log |B_i|$ bits of x_l, denoted by x_{l_h}, indicate the association to a sub-block. The rest of the bits of x_l are denoted by x_{l_l}. Notice that the binary division $x_l = x_{l_h}|x_{l_l}$ depends on the size of the block $|B_i|$ to which x belongs. As before, some blocks may be empty and some may contain more than one element.

The unary vector Π_i maintains all instances of x_{l_h} in block B_i. The space consumption of Π_i over all blocks is $\sum_{i=0}^{n-1}(2|B_i|) = 2n$ bits. The lower bits, x_{l_l}, of all elements are stored explicitly in each block using an array of size $|B_i| \log \frac{|U|}{n|B_i|}$ bits. Summarizing all blocks, the space required becomes $n \log \frac{|U|}{n} - \sum_{i=0}^{n-1}(|B_i| \log |B_i|)$.

Lemma 2.1. *For n integer elements drawn from a universe U, a static succinct indexable dictionary that can be built in one pass over the elements can be stored in $n \log \frac{|U|}{n} + 4n + o(n) - \sum_{i=0}^{n-1}(|B_i| \log |B_i|)$ bits and can support Select(j) queries in $O(1)$ time and Rank(x) queries in $O(\min\{\log n, \log(\max_{i\in[0,n-1]} |B_i|),\ \log \frac{|U|}{n \cdot \max(1, \min_{i\in[0,n-1]} |B_i|)}\})$ time. Assuming the elements are given in sorted order, construction time is linear in n, and the additional working space during the construction is $O(n)$ bits.*

Proof. The Select(j) operation, which returns the j-th ranked element, is a bit concatenation $x_h|x_{l_h}|x_{l_l}$ and is fulfilled as follows:

1. x_h: The binary representation of $block\_number(j, \Psi)$, padded to $\log n$ bits.
2. x_{l_h}: After calculating the index of j in its block ($a = j - first\_index(x_h, \Psi)$) and the number of elements in the block ($b = num\_elements(x_h, \Psi)$), x_{l_h} is the binary representation of $block\_number(a, \Pi_{x_h})$ padded to $\log(b)$ bits.
3. x_{l_l}: The element in position a of the explicit array of block x_h.

The Rank(x) operation, which returns the rank of element x assuming that x belongs to the s-th sub-block of the k-th block, results in an integer summation of the following:

1. The index of the first element of the k-th block, $first\_index(k, \Psi)$, where the first $\log n$ bits of x (previously denoted by x_h) represent block k.

2. The index of the first element of the s-th sub-block among all elements of the k-th block, $first\_index(s, \Pi_k)$, where the next $\log(num\_elements(k, \Psi))$ bits of x (x_{l_h}) represent sub-block s.
3. The index of x among all elements of the s-th sub-block, found by performing a binary search (to find the index of x_{l_l} among all (x_{l_l})s of the sub-block), using the boundaries of sub-block s. The size of sub-block s is $num\_elements(s, \Pi_k)$.

The operations ***num_elements***, ***first_index*** and ***block_number*** are composed of Rank and Select operations on bit vectors and therefore can be performed in constant time using an additional $o(n)$ bits. Hence, the total time for querying Select(j), which is composed solely of these operations, is constant. The Rank(x) query involves a binary search. Assuming x resides in one of the sub-blocks of B_i, which contains up to $\frac{|U|}{n|B_i|}$ elements, the binary search time is bounded by $O(\log \frac{|U|}{n|B_i|})$. Therefore, in the worst case, Rank(x) can require up to $O(\min\{\log n, \log(\max_{i\in[0,n-1]} |B_i|), \log \frac{|U|}{n\cdot\max(1,\min_{i\in[0,n-1]} |B_i|)}\})$ time.

The space consumption is $4n + o(n)$ bits to maintain 2 levels of unary vectors. Specifically, Ψ is for the first level and $\Pi_0, \ldots, \Pi_{n-1}$ are for the sub-blocks. The remaining bits are stored explicitly using $n \log \frac{|U|}{n} - \sum_{i=0}^{n-1}(|B_i| \log |B_i|)$ bits. □

3 Improving Query Time Using Hashing

In this section we describe a static succinct indexable dictionary construction in which both Rank(x) and Select(j) are performed in constant time, can be constructed in $O(n)$ expected time, and consumes $O(n \log \frac{|U|}{n})$ bits during construction. To the best of our knowledge, this is the first implementation that gives a construction of a static indexable dictionary with bounded, linear construction time in compact space, and still provides constant time queries.

The idea presented here improves upon the approach described in Section 2. Comparing the space consumption presented in Lemma 2.1 to the maximal number of bits allowed by a succinct representation, there are an additional $|B_i| \log |B_i|$ bits available in each block. This extra space can be used to maintain a mapping between each element and its rank within the block and to replace the binary search with a constant time operation. When $|B_i|$ is large, this is achieved using a minimal perfect hash function, described by Hagerup and Tholey [11], and a table for mapping every x_l of every block to its relative rank inside the block. Elements in the remaining blocks are mapped by maintaining one static compressed function [12]. Because in the latter case one compressed function is used for multiple blocks, an x_l might be repeated, and therefore x would have to be mapped to its relative rank inside the block. An alternative approach for mapping both large and small blocks is to use either of the results [11–13], by carefully trading space and construction time. The result of [14] can be considered for a more practical hash function implementation.

Theorem 3.1. *For n integer elements drawn from a universe U, there exists a succinct static indexable dictionary that can be built in one pass over the elements, and that support Rank(x) and Select(j) queries in $O(1)$ time. The space consumption is $n \log \frac{|U|}{n} + 4n + n\cdot\max\{H_0', \log e\} + o(nH_0') + O(\log\log|U|) + O((\log\log \frac{|U|}{n})^2)$ bits. H_0' denotes the zero-th order entropy of up to n values taken from a universe of size $(\log\log \frac{|U|}{n})^2$, and therefore, in the worst case, it is bounded by $2\log\log\log \frac{|U|}{n}$. Assuming that the elements are given in sorted order, the expected construction time is $O(n)$, and the working space during construction is $O(n \log \frac{|U|}{n})$.*

Proof. Select(j) remains unchanged and is performed in the manner described in Lemma 2.1. Rank(x), assuming that x belongs to the k-th block, results in an integer summation of the following:

1. The index of the first element of the k-th block, $first\_index(k, \Psi)$, where the first $\log n$ bits of x (previously denoted by x_h) represent block k.
2. The relative index of x in the k-th block, maintained in a perfect hash table or a compressed static function, depending on the block size.

The existence of x is validated when Select(Rank(x)) equals x. The mapping between an element and its relative rank within the block is maintained:
When the block is large, when $|B_i| > (\log\log\frac{|U|}{n})^2$, the minimal perfect hash function of Hagerup and Tholey [11] is used, and a table of $|B_i|\log|B_i|$ bits for each block is maintained. The hash function maps x_l to an index in the table where the element's relative rank within the block resides. In each block, this maps $\log\frac{|U|}{n}$ bits to $\log|B_i|$ bits.
The remaining elements in all other blocks are mapped into their relative block's rank using one static compressed function [12]. In this case, because any given x_l can be repeated, the mapping is between all the bits of x to its relative rank within the block. Thus, the static function maps $\log|U|$ bits to $2\log\log\log\frac{|U|}{n}$ bits.

Time analysis: Both (1) and (2) are constant time operations, assuming use of [11] and [12]. Hence, Select(j) and Rank(x) are constant time operations.
Space analysis: When $|B_i| > (\log\log\frac{|U|}{n})^2$, the mapping in each block requires $|B_i|\log e + \log\log\frac{|U|}{n} + o(|B_i|) + o(\log\log\frac{|U|}{n})$ bits. An additional $|B_i|\log|B_i|$ bits are required for the table. For all other blocks, mapping requires $nH_0' + o(nH_0') + O((\log\log\frac{|U|}{n})^2) + O(\log\log|U|)$ bits. H_0' denotes the zero-th order entropy of the ranks of the elements within the small blocks; therefore, in the worst case, it is bounded by $2\log\log\log\frac{|U|}{n}$.

In total, there are no more than n elements in the compressed function and in the hash functions together. In addition, there are up to $\frac{n}{(\log\log\frac{|U|}{n})^2}$ hash functions. Taking into consideration the $4n + o(n) + n\log\frac{|U|}{n} - \sum_{i=0}^{n-1}(|B_i|\log|B_i|)$ bits described in Lemma 2.1, the total space is $n\log\frac{|U|}{n} + 4n + n\max\{H_0', \log e\} + o(nH_0') + O(\log\log|U|) + O((\log\log\frac{|U|}{n})^2)$.
Construction: Each hash function is constructed in $O(|B_i| + \log\log\frac{|U|}{n})$ expected time using $O(|B_i|\log\frac{|U|}{n})$ bits of working space. Because there can be up to $\frac{n}{(\log\log\frac{|U|}{n})^2}$ such hash functions, this becomes, in total, $O(n)$ expected time and $O(n\log\frac{|U|}{n})$ bits working space. The static compressed function can be constructed in $O(n)$ expected time. To achieve a compact space construction of $O(n\log\frac{|U|}{n})$ bits, $\log n$ compressed functions should be used, one function for each $\frac{n}{\log n}$ blocks, instead of one function for all small blocks. □

4 Improving Query Time Using Deterministic Signatures

A fully deterministic alternative to the algorithm presented in Section 3 is established by keeping the Select(j) operation constant, and allowing Rank(x) to consume $O(\log\log\frac{|U|}{n})$ time. By doing so we can support Rank queries on $x \notin S$, hence it is a fully indexable dictionary. For the purpose of this algorithm, Lemma 2.1 is simplified. Instead of dividing each of the blocks $B_0, \ldots, B_i, \ldots B_{n-1}$ into sub-blocks, the x_l parts are maintained explicitly (in an array), as in the Elias-Fano encoding scheme. This is summarized in the following lemma (given without proof):

Lemma 4.1. *For n integer elements drawn from a universe U, a static succinct indexable dictionary that can be built in one pass over the elements can be stored in $n \log \frac{|U|}{n} + 2n + o(n)$ bits, and can support Select(j) query in $O(1)$ time and Rank(x) query in $O(\log \frac{|U|}{n})$ time. Assuming that the elements are given in sorted order, construction time is linear in n, and the extra working space needed during construction is $o(n)$ bits.*

To reduce Rank(x) query time, a grouping approach is taken. Each block B_i that contains more than $(\log \frac{|U|}{n})^2$ elements is divided into groups of size $(\log \frac{|U|}{n})^2$ elements; any residue elements are added to the group before it (last group). From each group, the first element is selected as a representative and is inserted into a data structure that supports predecessor queries. Assuming x is given on query, the block in which it resides is determined, and if the block size is larger than $(\log \frac{|U|}{n})^2$, its predecessor is found among the representatives and a binary search is performed over the representative's group. Otherwise, a binary search is performed over all elements in the block.

Theorem 4.1. *For n integer elements drawn from a universe U, a succinct static fully indexable dictionary than can be built in one pass over the elements is stored in $n \log \frac{|U|}{n} + 2n + o(n)$ bits and supports Rank(x) query in $O(\log \log \frac{|U|}{n})$ time and the Select(j) query in $O(1)$ time. Assuming that the elements are given in sorted order, construction time is linear, and the extra working space needed during construction is $o(n)$ bits.*

Proof. The Select(j) operation, which returns the j-th ranked element, is a bit concatenation of $x_h|x_l$ and is fulfilled as follows:

- x_h: The binary representation of $block\_number(j, \Psi)$ padded to $\log n$ bits.
- x_l: The element in position j of the explicit array.

Recall that the Rank(x) operation returns the rank of the largest element that is less or equal x, and assume that this element belongs to the k-th block. For this operation, we utilize a deterministic static predecessor data structure, proposed by Ružić [15]. In the word RAM model, for n' elements (of size less than a machine word) from a universe of size $|U'|$ the deterministic structure supports predecessor queries in $O(\log \log |U'|)$ time. With a sorted input, the structure can be constructed in linear time, and the space required is $O(n' \log |U'|)$ bits. Here, we use it to index the groups' representatives from the explicit array, per block. Recall that each x_l is drawn from a universe of size $\frac{|U|}{n}$, and that in each block, the number of representatives is $\lfloor \frac{|B_i|}{(\log \frac{|U|}{n})^2} \rfloor$. As a result, a predecessor query to attain a group's representative in each block takes $O(\log \log \frac{|U|}{n})$ time. Overall space consumption for the predecessor structures is $O\left(\sum_{i=0}^{n-1} (\frac{|B_i|}{(\log \frac{|U|}{n})^2} \log \frac{|U|}{n})\right) = o(n)$ bits since $\sum_{i=0}^{n-1} (|B_i|) = n$.
The result of the Rank(x) operation is an integer summation of the following:

1. The index of the first element of the k-th block, $first\_index(k, \Psi)$, where the first $\log n$ bits of x (previously denoted by x_h) represent block k.
2. The index of x among all elements of the k-th block. The boundaries of the group that element x belongs to are found and then a binary search on x_l is performed. The number of elements in block k is $num\_elements(k, \Psi)$, and if it is less than $(\log \frac{|U|}{n})^2$, the group is the whole block. Otherwise, the predecessor query is performed on the block to find the representative of the group. The group's boundaries in this case are the $(\log \frac{|U|}{n})^2$ proceeding the representative unless this is the last group, when the upper boundary is extended to the end of the block. Hence, The number of elements in the last group is as many as $2(\log \frac{|U|}{n})^2 - 1$. A binary search is performed using these boundaries.

The operations ***num_elements***, ***first_index*** and ***block_number*** are composed of Rank and Select operations on bit vectors. The operations can be performed in constant time by allocating additional $o(n)$ bits. Hence, the total time for querying Select(j), which is solely composed of these operations, is constant. The Rank(x) query involves a predecessor query and a binary search, and therefore takes $O(\log\log\frac{|U|}{n})$ time.

The space consumption is $2n + o(n)$ bits to maintain the unary vector which maintains all x_h parts. The remaining bits are stored explicitly using $n\log\frac{|U|}{n}$ bits, and the predecessor structure consumes $o(n)$ bits. □

5 Applications for Dynamic Dictionary Matching

The aforementioned indexable dictionary can be utilized in many applications. As an example we show how to reduce the query time of a succinct dynamic dictionary matching algorithm. Specifically, Theorem 3.1 can be used to reduce the best known query time by a factor of $\log\sigma$ to $O(|T|\log\log m + occ)$ while the update time remains the same but becomes "expected". For cases where the worst-case update time is preferred, by utilizing Theorem 4.1, the query time is reduced by a factor of $\frac{\log\sigma}{\log\log\sigma}$ to $O(|T|\log\log m\cdot\log\log\sigma + occ)$. In both cases, the total length of patterns $P_1, P_2, ..., P_d$ is denoted by m, and the text size is denoted by $|T|$; the alphabet is of size σ. The main contribution presented here is a significant reduction in query time, making the succinct solution slower by a factor of $\log\log m$ when compared to the optimal non-succinct solution.

This result is based on a recent work [16] that classifies patterns into groups according to their lengths. In one of the groups, an indexable dictionary is used to maintain and query patterns. Replacing the indexable dictionary with the structure described in either Theorem 3.1 or Theorem 4.1 achieves reduction in query time. Results are summarized it table 5.1. Another possible application of these indexable dictionaries is in Aho-Corasick automaton construction [16]. This yields a compact working space construction that supports querying a text in optimal time: $O(|T| + occ)$.

Table 5.1. Comparison of dynamic dictionary matching algorithms where m is the total length of d patterns and σ is the alphabet size. Space consumption is $(1 + o(1))m\log\sigma + O(d\log m)$ bits for the three cases.

	Query time	Update time
[17]	$O(\|T\|\log m + occ)$	$O(\|P\|\log\sigma + \log^2 m)$
[16]	$O(\|T\|\log\log m\cdot\log\sigma + occ)$	$(\|P\|\log m\log\log m)$
Using Theorem 3.1	$O(\|T\|\log\log m + occ)$	$O(\|P\|\log m\log\log m)$ expected
Using Theorem 5.1	$O(\|T\|\log\log m\cdot\log\log\sigma + occ)$	$O(\|P\|\log m\log\log m)$

5.1 Sketch of Dictionary Matching Algorithm

We now briefly describe the dictionary matching algorithm and how the new indexable dictionaries blend. The technique [16] achieves reduction in query time using a specific division of the patterns' domain into groups based on length, and each group is handled differently with a designated data structure. The patterns' domain is divided into four groups:

1. Group XL (Extra Large): for patterns that are longer than or equal to $\frac{m}{\log\log m}$

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

In a query upon a given text T, each of the designated data structures of these groups is queried separately. On insertion, given a pattern P, if the pattern does not exist in the dictionary, it is inserted directly into the data structure of the group that matches its length. Similarly, upon deletion, the pattern is first queried among all of the groups, and deleted if it exists.

Each level of group Large maintains static Aho-Corasick automatons [18] that are used for queries on texts. The Aho-Corasick automaton is a generalized KMP [19] structure which is essentially a trie with three types of links between internal states (nodes): next transition, failure, and report links. The next transition 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. The algorithm used is based on observations of how to represent each of the Aho-Corasick components using succinct data structures [20].

Assuming that p_i is the i-th prefix in suffix lexicographic order, and c_i is the label on the incoming edge in the trie that represents state p_i, s.t. $p_i = p'_i c_i$; the next transition is encoded using an indexable dictionary that maintains m' ($m' \leq m$) tuples of the form $(c_i, state(p'_i))$ where the name of a state x, $state(x) \in [0, m'-1]$ is the rank of the state in the suffix lexicographic order, and m' is the number of states in the automaton. Eventually, the rank of $(c_i, state(p'_i))$ in the indexable dictionary is i.

Theorem 5.1. *There is a succinct data-structure for the dynamic dictionary matching problem of d patterns with total length m, over an alphabet of size σ. It supports querying all occurrences of the d patterns in a text T in $O(|T|\log\log m + occ)$ time and insertion or deletion of a pattern P in $O(|P|\log m \log\log n)$ expected amortized time.*

Proof. In the result described in Theorems 2.1 and 2.2 of [16], the query time bottleneck is with the Large group. Queries on the Small, Medium and Extra Large groups are bounded by $O(|T|\log\log m + occ)$ time, while on the Large group it is bounded by $O(|T|\log\log m \cdot \log\sigma + occ)$. The $\log\log m$ factor is a result of the $O(\log\log m)$ internal structures, and the $\log\sigma$ factor is a result of querying the indexable dictionary of the Aho-Corasick automaton. Recall that the indexable dictionary maintains up to m tuples of the form ($c \in [0, \sigma-1]$, $i \in [0, m-1]$), yielding a total of m elements out of a universe of size $\sigma \cdot m$. Therefore, utilizing Theorem 3.1, a constant query time is achieved in an indexable dictionary that consumes $m\log\sigma + o(m\log\sigma)$ bits. □

6 Conclusion and Future Work

In this work we described two time and space bounded constructions of succinct indexable dictionaries. Both are simple and suitable for practical usages. We exemplified a possible application in the pattern matching domain, by improving the query time of a succinct dynamic dictionary matching algorithm. As a future work it is left to show whether it is possible to reduce the Rank query time of the deterministic algorithm. In addition, whether an indexable dictionary, supporting constant time queries, can be constructed using succinct working space.

References

1. Elias, P.: Efficient storage and retrieval by content and address of static files. J. ACM **21**(2) (1974) 246–260
2. Fano, R.: On the Number of Bits Required to Implement an Associative Memory. Computation Structures Group Memo. MIT Project MAC Computer Structures Group (1971)
3. Raman, V., Rao, S.S.: Static dictionaries supporting rank. In: In Proc. ISAAC 99, LNCS 1741, SpringerVerlag (1999) 18–26
4. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4) (2007)
5. Gupta, A., Hon, W.K., Shah, R., Vitter, J.S.: Compressed data structures: Dictionaries and data-aware measures. In: Data Compression Conference, 2006. DCC 2006. Proceedings, IEEE (2006) 213–222
6. Patrascu, M.: Succincter. In: Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science. FOCS '08, Washington, DC, USA, IEEE Computer Society (2008) 305–313
7. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: Searching a sorted table with o(1) accesses. In: Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '09 (2009) 785–794
8. Munro, J.I.: Tables. In: FSTTCS. (1996) 37–42
9. Clark, D.R.: Compact Pat Trees. PhD thesis, Waterloo, Ont., Canada, Canada (1998) UMI Order No. GAXNQ-21335.
10. Grossi, R., Orlandi, A., Raman, R., Rao, S.S.: More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In: 26th International Symposium on Theoretical Aspects of Computer Science. Volume 3 of Leibniz International Proceedings in Informatics (LIPIcs). (2009) 517–528
11. Hagerup, T., Tholey, T.: Efficient minimal perfect hashing in nearly minimal space. In: STACS 2001, Springer (2001) 317–326
12. Belazzougui, D., Venturini, R.: Compressed static functions with applications. In: Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM (2013) 229–240
13. Porat, E.: An optimal bloom filter replacement based on matrix solving. In: Computer Science-Theory and Applications. Springer (2009) 263–273
14. Botelho, F., Pagh, R., Ziviani, N.: Simple and space-efficient minimal perfect hash functions. In: Algorithms and Data Structures. Volume 4619 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 139–150
15. Ružić, M.: Making deterministic signatures quickly. ACM Trans. Algorithms **5**(3) (2009) 26:1–26:26
16. Feigenblat, G., Porat, E., Shiftan, A.: A grouping approach for succinct dynamic dictionary matching. Algorithmica (2015) 1–17
17. Hon, W., Ku, T., Lam, T.W., Shah, R., Tam, S., Thankachan, S.V., Vitter, J.S.: Compressing dictionary matching index via sparsification technique. Algorithmica **72**(2) (2015) 515–538
18. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6) (1975) 333–340
19. Knuth, D.E., Morris, J., Pratt, V.R.: Fast pattern matching in strings. SIAM Journal of Computing **6**(2) (1977) 323–350
20. Belazzougui, D.: Succinct dictionary matching with no slowdown. In: CPM. (2010) 88–100