Succinct Representations for (Non)Deterministic Finite Automata *

Sankardeep Chakraborty[1][0000−0002−2395−4160], Roberto Grossi[2][0000−0002−7985−4222], Kunihiko Sadakane[3][0000−0002−8212−3682], and Srinivasa Rao Satti[4][0000−0003−0636−9880]

[1] National Institute of Informatics, Japan. sankar@nii.ac.jp
[2] Università di Pisa, Italy. grossi@di.unipi.it
[3] The University of Tokyo, Japan. sada@mist.i.u-tokyo.ac.jp
[4] Seoul National University, South Korea. ssrao@cse.snu.ac.kr

Abstract. Deterministic finite automata are one of the simplest and most practical models of computation studied in automata theory. Their extension is the non-deterministic finite automata which also have plenty of applications. In this article, we study these models through the lens of succinct data structures where our ultimate goal is to encode these mathematical objects using information theoretically optimal number of bits along with supporting queries on them efficiently. Towards this goal, we first design a succinct data structure for representing any deterministic finite automaton $\mathcal{D}$ having n states over a σ-letter alphabet Σ using $(\sigma-1)n\log n(1+o(1))$ bits, which can determine, given an input string x over Σ, whether $\mathcal{D}$ accepts x in optimal $O(|x|)$ time. We also consider the case when there are $N < \sigma n$ non-failure transitions, and obtain various time-space trade-offs in both the cases. When the input deterministic finite automaton $\mathcal{A}$ is acyclic, not only we can improve the above space bound significantly to $(\sigma-1)(n-1)\log n + O(n+\log^2\sigma)$ bits, we can also check if an input string x over Σ can be accepted by $\mathcal{A}$ optimally in $O(|x|)$ time. We also exhibit a succinct data structure for representing a non-deterministic finite automaton $\mathcal{N}$ having n states over a σ-letter alphabet Σ using $\sigma n^2 + n$ bits of space, such that given an input string x, we can decide whether $\mathcal{N}$ accepts x efficiently in $O(n^2|x|)$ time. Finally, we also provide time and space efficient algorithms for performing several standard operations such as union, intersection and complement on the languages accepted by deterministic finite automata.

Keywords: data and image compression · computational complexity.

1 Introduction

Automata theory is a branch of theoretical computer science that deals exclusively with the definitions, properties and applications of different mathematical

* The full version of this paper appears as [4]. The work of the first author is supported by JSPS KAKENHI Grants Number 18H05291

models of computation. These models play a major role in multiple applied areas of computer science. One of the most basic and fundamental models that is studied in automata theory since a long time ago is called the *finite automata*. They primarily come in two different types, *deterministic finite automata* (henceforth DFA) and *non-deterministic finite automata* (henceforth NFA) among others. There exists more complex and sophisticated models as well, for example, *Context-free grammars, Turing machines* etc. In what follows, let us formally define DFA and NFA in a nutshell as these are our primary subjects of study in this article. A DFA $\mathcal{D}$ is a quintuple $\mathcal{D} = (\Sigma, Q, q_0, \delta, F)$ where: Σ is an *alphabet* i.e., a finite set of letters, Q is the finite set of *states*, $q_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function* and finally, $F \subseteq Q$ is the *set of final states*. We often extend the transition function to $\delta : Q \times \Sigma^* \rightarrow Q$ which is defined recursively as follows: $\delta(q, \epsilon) = q$ for all $q \in Q$, where ϵ is the empty string; and $\delta(q, aw) = \delta(\delta(q, a), w)$ for all $q \in Q$, $a \in \Sigma$, and $w \in \Sigma^*$. Given the above definition, we say that the DFA accepts a string x over the alphabet Σ if and only if $\delta(q, x) \in F$. The *language* $\mathcal{L}$ accepted by a DFA $\mathcal{D}$ is defined as the set of all strings accepted by the DFA $\mathcal{D}$, and is denoted by $\mathcal{L}(\mathcal{D})$. In the rest of this paper, we assume that the alphabet Σ is $\{1, 2, \ldots, \sigma\}$, and the state set Q is $\{q_0, q_1, \ldots, q_{n-1}\}$. A deterministic automaton $\mathcal{A}$ is called *acyclic* [18] if it has a unique recurrent state where a state q is defined as *recurrent* if there exists a non-empty string x over Σ such that $\delta(q, x) = q$. Non-recurrent states are typically called *transient*, and the unique recurrent state (denoted by $q'' \in Q$) is classically called the *dead state* as $\delta(q'', \sigma) = q''$ for all $\sigma \in \Sigma$. An NFA is a conceptual extension of DFAs where the definition of the transition function is mainly extended. More specifically, for DFA, the transition function is defined as δ: $Q \times \Sigma \rightarrow Q$ whereas for NFA, the same is defined as δ: $Q \times \Sigma \rightarrow \mathcal{P}(Q)$ where $\mathcal{P}(Q)$ denotes the power set of Q. Another extension, which is sometimes used in the literature, is to simply allow more than one initial state in an NFA, and in this case, the third item in the tuple becomes I denoting the set of initial states, instead of singleton $\{q_0\}$. The rest of above quintuple definition remains as it is for NFA. Thus, in the case of NFA $\mathcal{N}$, the language $\mathcal{L}(\mathcal{N})$ is defined as $\{x \mid \exists_{q \in I} \exists_{q' \in F} [q' \in \delta(q, x)]\}$. We refer the readers to the classic texts of [16, 22] for a thorough discussions on these mathematical models and automata theory in general.

Even if a DFA is defined as an abstract mathematical concept, still it has a myriad of practical applications. More specifically, it is used in text processing, compilers, and hardware design [22]. Quite often it is implemented in small hardware and software tools for solving various specific tasks. For example, a DFA can model a software that can figure out whether or not online user input such as email addresses are valid. DFAs/NFAs are also used for network packet filtering. In some of these applications, the alphabet is large and there is a failure/exit state so that only a subset of transitions go to non-failure states; so we call the latter ones *non-failure* transitions.

Despite having so many applications in practically motivated problems, we are aware of only a few studies of encoding arbitrary DFAs and NFAs from

the point of view of *succinct data structures* (refer to the related work section) where the goal is to store an arbitrary element from a set Z of objects using the information theoretic minimum $\log(|Z|) + o(\log(|Z|))$ bits of space while being able to support the relevant set of queries efficiently, which is what we focus on in this paper. We also assume the usual model of computation, namely a $\Theta(\log n)$-bit word RAM model where n is the size of the input.

Related Work: The field of *succinct data structures* originally started with the work of Jacobson [17], and by now it is a relatively mature field in terms of breadth of problems considered. To illustrate this further, there already exists a large body of work on representing various combinatorial objects succinctly, such as trees [19,20], interval graphs [1] etc. Regarding encoding DFA/NFA, the work on Wheeler NFA [15] can be considered as an attempt to encode particular classes of NFA succinctly, and this work has recently been generalized to arbitrary NFA in [7]. In the later part of our paper, we will compare these results with ours.

For DFA/NFA, other than the basic structure that is mentioned in the introduction, there exists many extensions/variations in the literature, for example, two-way finite automata, Büchi automata and many more. Researchers generally study the properties, limitations and applications of these mathematical structures. One such line of study that is particularly relevant to us for this paper is the research on counting DFAs/NFAs. Since the fifties there are plenty of attempts in exactly counting the number of DFAs/NFAs with n states over the alphabet Σ, and the state-of-the-art result is due to [3] for DFAs and [11] for NFAs. We refer the readers to the survey of Domaratzki [10] for more details. Basically, from these results, we can deduce the information theoretic lower bounds on the number of bits required to represent any DFA or NFA. Then we augment these lower bounds by designing succinct data structures capable of executing algorithms efficiently using this representation, and this is our main contribution.

DFA and NFA Enumeration After a number of efforts by several authors, finally Bassino and Nicaud [3] found a matching upper and lower bound on the number of non-isomorphic initially-connected (i.e., all the states are reachable from the initial state) DFA's. They showed that the number of DFAs with n states over an alphabet of size σ is $\Theta(n2^{2n}S_2(\sigma n, n))$ where $S_2(n, m)$ denotes the Stirling numbers of the second kind. Using the approximation of the Stirling numbers of the second kind [14], which states that $S_2(n, m) \approx \frac{m^n}{m!}$, we can obtain the information theoretic lower bound for representing any DFA having n states and σ-sized alphabet is given by $\lg(n2^{2n}S_2(\sigma n, n)) = (\sigma - 1)n \lg n + O(n)$ bits. On the other hand, Domaratzki et al. [11] showed that there are asymptotically $2^{\sigma n^2+n}$ initially connected NFAs on n states over a σ-letter alphabet with a fixed initial state and one or more final states. Thus, information theoretically, we need at least $\sigma n^2 + n$ bits to represent any NFA. In what follows later, we show that we can represent any given DFA/NFA using asymptotically optimal number of bits as mentioned here. Throughout this paper, we assume that the input DFAs/NFAs that we want to encode succinctly are initially connected.

1.1 Our Main Results and Paper Organization

The classical representation of DFAs/NFAs consists of explicitly writing the transition function δ in a two dimensional array $J[0..n-1][1..\sigma]$ having n rows corresponding to the n states of the DFA/NFA and σ (where $|\Sigma| = \sigma$) columns corresponding to the alphabet Σ such that $J[i][j] = \delta(q_i, j)$ where $q_i \in Q, j \in \Sigma$. For DFA, the entry in $J[i][j]$ is a singleton set whereas for NFA it could possibly contain a set having more than one state. Thus, the space requirement for representing any given DFA (NFA respectively) is given by $O(n\sigma \log n)$ ($O(n^2\sigma \log n)$ respectively) bits. These space bounds are clearly not optimal – for the DFAs, it is off by an additive $n \log n$ term from the information theoretic minimum, while for the NFAs, it is off by a multiplicative factor of $\log n$ from the optimal bound. We alleviate this discrepancy in the space bounds by designing optimal succinct data structures for these objects.

Towards this goal, in Section 2.1 we first discuss the relevant prior work from [3], and show that, by using suitable data structures, their work already gives a succinct encoding of DFA. But the major drawback of this encoding is that it is not capable of handling the problem of checking whether a string is accepted by the DFA extremely efficiently. In Section 2.3, we overcome this problem by designing a succinct data structure for DFA, which can also check the string acceptance almost optimally. We summarize our results below.

Theorem 1. *Given a DFA $\mathcal{D}$ having n states and working over an alphabet Σ of size σ, and a query string x for which we want to check the membership in $\mathcal{L}(\mathcal{D})$, there exists a succinct encoding for $\mathcal{D}$ taking:*

- $(\sigma - 1)n \log n + \sigma n + o(\sigma n)$ *bits, to support queries in $O(|x|)$ time;*
- $(\sigma - 1)n \log n + O(n \log \sigma)$ *bits, to support queries in $O(|x| \log \sigma)$ time; and*
- $(\sigma - 1)n \log n + n \log \sigma + O(n)$ *bits, to support queries in $O(|x| \log n)$ time.*

If $\mathcal{D}$ has $N < \sigma n$ non-failure transitions, then there exists an encoding taking:

- $(N - n) \log n + O(n\sigma)$ *bits, to support queries in $O(|x|)$ time;*
- $(N - n) \log n + O(N \log \sigma)$ *bits, to support queries in $O(|x| \log \sigma)$ time; and*
- $N(\log n + \log \sigma + 1.45)$ *bits, to support queries in $O(|x| \log n)$ time.*

The upper bounds in Theorem 1 save roughly $n \log n$ bits with respect to the immediate representation of the DFA. The former upper bound is optimal as it matches the information-theoretical lower bound in Section 1, up to lower order terms. As for the latter upper bound, we do not know its optimality but it is smaller than the information-theoretical lower bound of $\lceil \log \binom{n^2}{N} \rceil + \Theta(N \log \sigma)$ bits derived for edge-labeled deterministic directed graphs [13]. Indeed, DFAs can be seen as a special case of these graphs where n is the number of nodes, $N \geq n - 1$ is the number of arcs, and σ is the maximum node degree.[5]

[5] A directed graph with labels on its arcs is deterministic if no two out-neighbor arcs have the same label. Since there are $\lceil \log \binom{n^2}{N} \rceil$ directed graphs [13] with n nodes and N arcs, each deterministic graph $G = (V, E)$ can have $L = \prod_{u \in V} d_u!$ label assignments for its arcs, where d_u s the out-degree of node u and $N = \sum_{u \in V} d_u$. Note that $\log L = \Theta(N \log \sigma)$ when labels are from Σ and thus $d_u \leq \sigma$.

We can improve the above space bound significantly if the given DFA is acyclic along with obtaining optimal query time for string acceptance checking. More specifically, in full version of this paper, we obtain the following result in this case.

Theorem 2. *Given an acyclic DFA $\mathcal{A}$ having $n-1$ transient states, a unique dead state and working over an alphabet Σ of size σ, there exists a succinct encoding for $\mathcal{A}$ taking $(\sigma-1)(n-1)\log n+3n+O(\log^2\sigma)+o(n)$ bits of space, which can optimally determine, given an input string x over Σ, whether $\mathcal{A}$ accepts x in time proportional to the length of x, using constant words of working space.*

This is followed by the succinct data structure for NFA (see full version of the paper for the proof) where we prove the following result. Note that the running time for string acceptance checking in the following theorem is close to optimal, due to the lower bound of Equi et al. [12].

Theorem 3. *Given an NFA $\mathcal{N}$ having n states and working over an alphabet Σ of size σ, there exists a succinct encoding for $\mathcal{N}$ taking σn^2+n bits of space, which can determine, given an input string x over Σ, whether $\mathcal{N}$ accepts x in $O(n^2|x|)$ time, using $2n$ bits of working space.*

Next we move on to discuss how one can support several standard operations such as union and intersection of two languages accepted by the deterministic finite automata. Classically it is done via the product automaton construction [16, 22], and here we provide a time and space efficient algorithm for performing this construction. More specifically, we show the following theorem in the full version of our paper.

Theorem 4. *Suppose we are given the succinct representations for two DFAs $\mathcal{D}_1$ (having n states) and $\mathcal{D}_2$ (having n' states) respectively such that both are working over the same alphabet Σ. Also suppose that the product automaton (denoted by $\mathcal{P}$) has n'' states where $n''\leq nn'$. Then, using $O(\sigma n'')$ expected time and $O(n''\log n'')$ bits of working space, we can directly construct a succinct representation for $\mathcal{P}$. Moreover, $\mathcal{P}$ can be represented optimally using $(\sigma-1)n''\log n''+O(n''\log\sigma)$ bits overall, and by suitably defining the final states of $\mathcal{P}$, we can make $\mathcal{P}$ accept either $\mathcal{L}(\mathcal{D}_1)\cup\mathcal{L}(\mathcal{D}_2)$ or $\mathcal{L}(\mathcal{D}_1)\cap\mathcal{L}(\mathcal{D}_2)$. Finally, given an input string x over Σ, we can decide whether $x\in\mathcal{L}(\mathcal{P})$ in $O(|x|\log\sigma)$ time using constant words of working space.*

We conclude in Section 3 with some remarks on future research avenues.

Preliminaries: We will assume the knowledge of basic graph theoretic terminology (like trees, paths etc) as given in [8] and basic graph algorithms (mostly the depth first search (henceforth DFS) traversal of a graph and its related concepts) as given in [6].

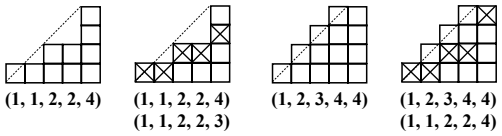

Fig. 1. A diagram of width $m = 5$ and height $n = 4$, a boxed diagram, a k-Dyck diagram and a k-Dyck boxed diagram with $k = 2$.

2 Succinct Representations for DFA and NFA

In this section, we provide all the upper bound results of our paper dealing with DFA/NFA. Throughout this section, whenever we mention DFA (NFA resp.), it should refer to an initially-connected deterministic (non-deterministic resp.) finite automata having n states and working over an alphabet Σ of size σ. With this notation in mind, we start with the succinct encoding of DFA first.

2.1 Succinct Encoding of DFA

Bassino and Nicaud [3] proved a beautiful bijection between the state transition diagram of any DFA and pairs of integer sequences which can be represented by boxed diagrams (will be defined shortly) along with providing an efficient algorithm to perform this construction. We will refer the readers to [3] for complete details regarding the bijection, counting and many other details that we choose to not repeat here. Later, Almeida et al. [2] also analysed the construction of Bassino and Nicaud [3] from which one can obtain a succinct encoding of DFA. These encodings do not support membership queries efficiently.

In what follows, we provide another succinct encoding based on the construction of Bassino and Nicaud [3] which is then used in Section 2.3 for efficient membership query support as well. Following [3], a *diagram* of width m and height n is defined as a sequence $(x_1, \ldots, x_m)$ of non-decreasing non-negative integers such that $x_m = n$. See Figure 1 for better visual description and understanding. A *boxed diagram* can be defined as a pair of sequences $((x_1, \ldots, x_m), (y_1, \ldots, y_m))$ where $(x_1, \ldots, x_m)$ is a diagram and for all i (such that $1 \leq i \leq m$), the y_i-th box of the column i of the diagram is marked. Note that $1 \leq y_i \leq x_i$. Thus, a diagram can lead to $\prod_{i=1}^{m} x_i$ boxed diagrams. A *k-Dyck diagram* of size n is defined as a diagram of width $m := (k-1)n + 1$ and height n such that $x_i \geq \lceil i/(k-1) \rceil$ for all $i \leq m-1$. Finally, a *k-Dyck boxed diagram* of size n is boxed diagram where the first coordinate $(x_1, \ldots, x_{(k-1)n+1})$ is a k-Dyck diagram of size n. Given these definitions, Bassino and Nicaud [3] proved the following theorem.

Theorem 5. *[3] The set $\mathcal{D}_n$ containing non-isomorphic initially-connected DFAs having n states and working over a σ-letter alphabet is in bijection with the set $\mathcal{B}_n$ of σ-Dyck boxed diagrams of size n. Moreover, the algorithms to construct the transition diagram of the DFA from k-Dyck boxed diagram, and vice versa run in linear time and space.*

Thus, by applying the above theorem, from any given DFA with n states and σ-letter alphabet, [3] produces a σ-Dyck boxed diagrams of size n, which can be in turn represented by two integer arrays $Max[1..m]$ and $Boxed[1..m]$ of length $m := (\sigma - 1)n + 1$ each. Furthermore, from these two arrays, it is possible to entirely reconstruct the DFA using the algorithm of Theorem 5. Thus, it is sufficient to store just these two arrays in order to encode any given DFA. For more details, readers are referred to [3]. For an example, see Figure 2 which will also serve as the working example for this part of our paper.

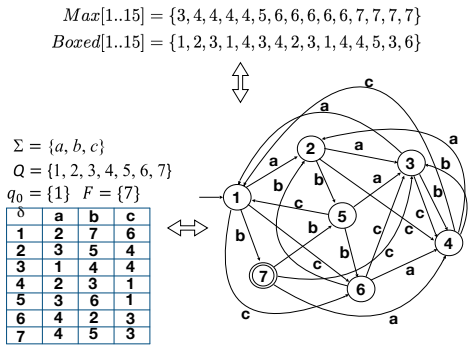

Fig. 2. Three ways to define the same DFA. This DFA will serve as the working example for our discussion. By using the techniques of [3], this DFA can be entirely represented by the $Max[1..15] = \{3, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7\}$ and $Boxed[1..15] = \{1, 2, 3, 1, 4, 3, 4, 2, 3, 1, 4, 4, 5, 3, 6\}$ arrays of length $(\sigma - 1)n + 1 = 15$ each.

First, we observe that, $1 \leq Max[1] \leq Max[2] \leq \cdots \leq Max[m] \leq n$ and $1 \leq Boxed[i] \leq Max[i]$ for each $i = 1, 2, \ldots, m$. This happens precisely because the translation is obtained by following a DFS on the DFA using the lexicographic order of words, and on each backtracking edge adding to the first vector the number of states scanned so far, and to the second vector the state reached. This also explains why each entry of these two arrays are upper bounded by n, the number of states of the given DFA. Now we consider the number of bits needed to encode the array $Max[1..m]$. It is easy to transform this array into an equivalent bit vector M of length $n + m$ with n ones in it (by storing the multiplicities of each of the n values from 1 to n in unary). Now, we encode the bit vector M using the structure of [21] using $\log \binom{(n+m)}{n} + o(n) + O(\log\log(m+n))$ bits. Since $m = (\sigma-1)n+1$, this space is $n \log \sigma + O(n)$ bits. Next we consider the number of bits required for array $Boxed[1..m]$. Because each entry of this array is an integer from 1 to n, we can use the result of [9] to represent the $Boxed[1..m]$ array using $(\sigma - 1)n \log n + O(\log^2 m)$ bits. Thus, in total, the size of the representation is $(\sigma-1)n \log n + n \log \sigma + O(n)$ bits. Because the information theoretic lower bound

is $(\sigma-1)n\log n + O(n)$ bits for the representation of DFA, this representation is succinct. In what follows, we show how to further reduce the encoding space.

2.2 Reducing the Space Further

Now consider the case when there is a failure/exit state labeled 0, and only N transitions among all the σn transitions go to non-failure states, for some $n \leq N \leq \sigma n$. Note that $Boxed$ has $N-n+1$ non-zero values. In this case we can reduce the space for $Boxed[1..m]$ by using a new bitvector $Z[1..m]$ which has $N-n+1$ ones. We use a new array $Boxed'[1..N-n+1]$ which stores non-zero values of $Boxed[1..m]$. Then $Boxed[i]$ is computed as follows. If $Z[i]=0$, $Boxed[i]=0$ (transition to the failure state). If $Z[i]=1$, $Boxed[i] = Boxed'[partial\_rank_1(Z,i)]$. If we use the data structure of [5], Z is represented in $\sigma n + o(\sigma n)$ bits, which is asymptotically smaller than the space lower bound of $(\sigma-1)n\log n + O(n)$. But, by using the data structure of [21], the bitvector Z can be represented in $\log\binom{\sigma n}{N} + o(N) + O(\log\log(\sigma n)) = N\log\frac{\sigma n e}{N} + o(N) \leq N\log\sigma + N\log e + o(N)$ bits. The space for $Boxed'$ is $(N-n+1)\log n$ bits. Therefore the total space for representing a DFA with N non-failure transitions is $(N-n)\log n + N\log\sigma + N\log e + o(N)$ bits, i.e., less than $\log n + \log\sigma + 1.45$ bits per transition.

Given a string x over Σ, it takes linear time (in the size of the DFA, i.e., $O(\sigma n)$ time) to decide whether the DFA accepts the string x, which is clearly not optimal as ideally it should be performed in time $O(|x|)$. This happens because the algorithm of Theorem 5 actually unravels the DFA from these two arrays $Max[1..m]$ and $Boxed[1..m]$, and then checks whether the input string can be accepted or not. Thus, from the point of view of string acceptance, these encodings of DFA (including the encodings of Bassino and Nicaud [3], and Almeida et al. [2]) are not optimal, whereas from the space requirement point of view, these are optimal. This motivates the need for a succinct encoding of a given DFA, where the problem of string acceptance can be performed in optimal time. In what follows, we provide such an encoding.

2.3 Succinct Data Structure for DFA

To design a succinct data structure for DFA, we need the following three bitvectors F, P and T in addition to an integer array $NewBoxed[1..m]$ (that can be obtained from the $Boxed[1..m]$ array of the previous section, as described later), which are defined as follows. P is a balanced parentheses sequence of length $2n$ obtained from the lexicographic depth-first search (DFS) tree of the given input automaton $\mathcal{D}$. More specifically, given any DFA $\mathcal{D}$, we first perform the lexicographic DFS on $\mathcal{D}$ to generate the lexicographic DFS tree R of $\mathcal{D}$, i.e., while looking for a new edge to traverse during DFS, the algorithm always searches in lexicographic order of edge labels. For example, in Figure 2, from any vertex, lexicographic DFS first tries to traverse the edge labeled a, followed by b and finally c. The tree R is represented as a balanced parenthesis sequence P together with auxiliary structures to support the navigational queries on R, as mentioned

in [19], using $2n + o(n)$ bits. The bitvector F is used to mark all the final states of the input DFA, hence it takes n bits.

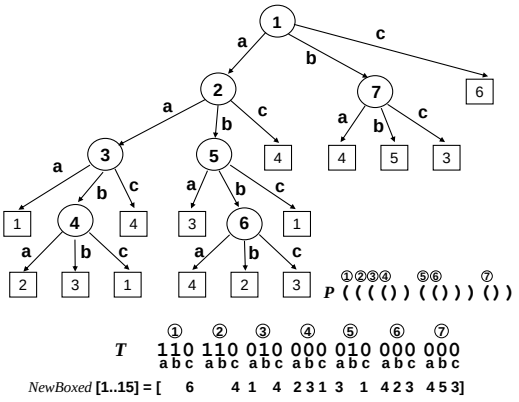

Fig. 3. The extended lex-DFS tree S of the automaton of Figure 2 along with the corresponding bitvectors P, T, and the *NewBoxed*[1..15] array (the elements of this array are drawn exactly below the corresponding 0s with which they share one to one correspondence with). Note that, for the same automaton *Boxed*[1..15] array is given as $Boxed[1..15] = \{1, 2, 3, 1, 4, 3, 4, 2, 3, 1, 4, 4, 5, 3, 6\}$.

Before explaining the other bitvector, T, required for our succinct encoding, we want to explain the contents of Figure 3. The tree depicted in the figure is what we call an *extended lexicographic DFS tree* or *extended lex-DFS tree* (denoted by S) in short. If we delete the squared nodes and their incident edges (originating from the circled nodes), we obtain the lexicographic DFS tree of the automaton $\mathcal{D}$. Actually these edges represent the *back edges/cross edges/forward edges* [6] (i.e., non-tree edges) in the DFS tree of the automaton $\mathcal{D}$. Traditionally the vertices in the square are not drawn (as in our case of Figure 3), rather the edges point to the nodes in the circle only (hence all the nodes appear only once). We have chosen to draw and define the extended lex-DFS tree this way as it helps us to design and explain our succinct data structure well. Also note that, edges originating from a circled node and going to another circled node represents tree edges whereas edges from circled to squared nodes represent non-tree edges.

Now given the extended lex-DFS tree S, we visit the nodes of S in DFS order and append a bit string of length σ for each vertex v of S marking which of its children are attached to v via tree edges (marked with 1) and which are attached to v via non-tree edges (marked with 0) in the lexicographic order of the edge labels. The string obtained this way is referred to as T. Thus, T is a bit-vector of length σn which captures the information about the tree and non-tree edges of S. More specifically, it has exactly $n - 1$ ones, which have one-to-one correspondence with the tree edges of the lexicographic DFS tree

of DFA $\mathcal{D}$, and has exactly $(\sigma-1)n+1$ zeros, which correspond to non-tree edges of the lexicographic DFS tree of DFA $\mathcal{D}$. See Figure 3 for an example. We relabel all the states of $\mathcal{D}$ such that the i-th vertex (state) in R in preorder has label i, and also modify the transition function accordingly. Now it is easy to see that, for the state with label i ($1 \leq i \leq n$), the corresponding node in the lexicographic DFS tree has exactly σ outgoing edges, and we encode the tree edges among them using the bits in the range $T[\sigma(i-1)+1..\sigma i]$. More specifically, $T[\sigma(i-1)+c]=1$ if and only if the outgoing edge labeled c is a tree edge ($1 \leq c \leq \sigma$). Similarly, we can also find the j-th outgoing tree edge from the state i by $select_1(T, j + rank_1(T, \sigma(i-1)))$.

In what follows, we show three different ways to represent the T array. (1) In the first method, we simply encode T using the structure of [5] as a bitvector of length σn having exactly $n-1$ ones while supporting constant time $rank/select$ queries in T. (2) In the second method, we compress T by observing that the positions of 1s in the T array form an increasing sequence, hence by using the data structure $D(n-1, \sigma n, \epsilon)$ of [23], T can be encoded in $O(n \log \sigma)$ bits (by setting $\epsilon = 1/\log(\sigma-1)$) while supporting $access$, $rank$ and $select$ operations in $O(\log \sigma)$ time. (3) Finally, we can encode T using the data structure of [21] in $n \log \sigma + O(n)$ bits (like we did for Max array in Section 2.1) while supporting the $rank$ query in $O(\log n)$ time and $select$ in $O(1)$ time.

Now let us define the new integer array $NewBoxed[1..m]$. First, observe that elements of the array $Boxed[1..m]$ are nothing but the leaves (i.e., node labels in the squared nodes) of the extended lex-DFS tree S in the left to right order. More specifically, they are the node labels of the destinations of the non-tree edges emanating from the nodes of the lexicographic DFS tree of the automaton $\mathcal{D}$ in their preorder. Instead of this specific ordering (followed in the $Boxed[1..m]$ array), $NewBoxed[1..m]$ lists the same node labels in the order of their appearance in the T bitvector (from left to right). Note that, as mentioned previously, these node are marked by 0s in T and they are in one-to-one correspondence with all the non-tree edges of the lexicographic DFS tree of the automaton $\mathcal{D}$. Thus, the $NewBoxed[1..m]$ array contains the same node labels as the $Boxed[1..m]$ array, but in a different order. See Figure 3 for an example. This completes the description of our succinct data structure for DFA. Note that Max is no longer used in our data structure.

Space complexity. We now analyze the space complexity of our data structure. The array $NewBoxed[1..m]$ takes $(\sigma-1)n \log n + O(\log^2 m)$ bits (by similar analysis as before for the $Boxed[1..m]$ array). The bitvector F consumes n bits. The bitvector P is stored using the structure of [19], hence it occupies $2n + o(n)$ bits. Depending on the three different choices for storing T, the space requirement is (1) $\sigma n + o(\sigma n)$, (2) $O(n \log \sigma)$ or (3) $n \log \sigma + O(n)$ bits. Thus, the overall space usage is as stated in the Theorem 1.

It is easy to further reduce the size if the DFA has only $N < \sigma n$ non-failure transitions. Using the bitvector $Z[1..m]$ (as defined in Section 2.2) for indicating non-failure transitions, the array $NewBoxed[1..m]$ is compressed to $N - n + 1$ non-zero values, which can be stored in $(N-n)\log n + O(\log^2 n)$ bits. Since Z

is a bit vector of length σn with N ones in it, we can use the three choices as above for representing T, using (1) $\sigma n + o(\sigma n)$, (2) $O(N \log \sigma)$ or (3) $N \log \sigma + 1.44N + o(N)$ bits. Thus the overall space usage is (1) $(N-n)\log n + O(n\sigma)$, (2) $(N-n)\log n + O(N \log \sigma)$, or (3) $(N-n)\log n + (N+n)\log \sigma + N \log e + o(N) \leq N(\log n + \log \sigma + 1.45)$ bits, depending on the representation used for T and Z.

Query algorithm. Suppose we are given an input string x of length y over Σ, and we need to decide if the DFA $\mathcal{D}$ accepts x or not. We start the following procedure from the initial state (stored explicitly using $O(\log n)$ bits) and repeat until the end of the input string x. At any generic step, to figure out the transition function $\delta(q, c) := q'$ where $1 \leq q, q' \leq n$ are the states, we first look at the bit $T[\sigma(q-1)+c]$. If it is 1, the outgoing edge labeled c from state q is a tree edge. Let $j := rank_1(T, \sigma(q-1)+c) - rank_1(T, \sigma(q-1))$. Then the outgoing edge is the j-th tree edge of node q in the lex DFS tree. Therefore $q' = child(q, j)$ (supported using the structure of [19]). If the bit is 0, the outgoing edge labeled c from state q is a non-tree edge. Let $j := rank_0(T, \sigma(q-1)+c)$. Then the edge is the j-th non-tree edge in the DFA, and q' is obtained by $q' := NewBoxed[j]$. Hence, when we reach the end of x, and if we are at an accepting/final states (can be figured out from the bitvector F), we say that the DFA $\mathcal{D}$ accepts x. Now, depending on the three choices for storing the T array and supporting *rank*/*select* queries in it, we obtain three different query time bounds for accepting x. In option (1), the input string x can be accepted optimally in $O(|x|)$ time. In (2), as the *rank* operations on T take $O(\log \sigma)$ time while all other operations, at each step, take $O(1)$ time, the overall run time for checking the membership of x is $O(|x| \log \sigma)$. Finally, in (3), because of the $O(\log n)$ time *rank* operation on T, the membership of x can be checked in $O(|x| \log n)$ time. The query times remain the same, even in the case where we have $N < \sigma n$ non-failure transitions (using the data structures mentioned in the previous paragraph). This completes the proof of Theorem 1.

Note that Cotumaccio and Prezza [7] recently gave an encoding for DFAs using $\log p + \log \sigma + 2$ bits per transition (where $1 \leq p \leq n$ is compressibility parameter), while our data structure takes $\log n + \log \sigma + 1.45$ bits per transition with efficient support for membership checking.

3 Concluding Remarks

We considered the problem of succinctly encoding any given DFA $\mathcal{D}$, acyclic DFA $\mathcal{A}$ or NFA $\mathcal{N}$ so as to check efficiently if they accept a given input string. To this end, we successfully designed succinct data structures for them that also support the string acceptance query efficiently for DFAs, acyclic DFAs, and NFAs. We believe that our work will spur further interest in designing succinct data structures for other mathematical models from the world of automata theory in future. For example, it would be interesting to see if other variants of automata can also be succinctly encoded with efficient query support mechanism.

References

1. Acan, H., Chakraborty, S., Jo, S., Satti, S.R.: Succinct data structures for families of interval graphs. In: WADS (2019)
2. Almeida, M., Moreira, N., Reis, R.: Enumeration and generation with a string automata representation. Theor. Comput. Sci. **387**(2), 93–102 (2007)
3. Bassino, F., Nicaud, C.: Enumeration and random generation of accessible automata. Theor. Comput. Sci. **381**(1-3), 86–104 (2007)
4. Chakraborty, S., Grossi, R., Sadakane, K., Satti, S.R.: Succinct representation for (non)deterministic finite automata. CoRR **abs/1907.09271** (2019)
5. Clark, D.R.: Compact Pat Trees. PhD thesis, University of Waterloo, Canada (1996)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms (3. ed.). MIT Press (2009)
7. Cotumaccio, N., Prezza, N.: On indexing and compressing finite automata. CoRR **abs/2007.07718** (2020)
8. Diestel, R.: Graph Theory, 4th Edition, Graduate texts in mathematics, vol. 173. Springer (2012)
9. Dodis, Y., Patrascu, M., Thorup, M.: Changing base without losing space. In: STOC. pp. 593–602 (2010)
10. Domaratzki, M.: Enumeration of formal languages. Bulletin of the EATCS **89**, 117–133 (2006)
11. Domaratzki, M., Kisman, D., Shallit, J.: On the number of distinct languages accepted by finite automata with n states. Journal of Automata, Languages and Combinatorics **7**(4), 469–486 (2002)
12. Equi, M., Grossi, R., Mäkinen, V., Tomescu, A.I.: On the complexity of string matching for graphs. In: 46th ICALP. LIPIcs, vol. 132, pp. 55:1–55:15 (2019)
13. Farzan, A., Munro, J.I.: Succinct encoding of arbitrary graphs. Theor. Comput. Sci. **513**, 38–52 (2013)
14. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)
15. Gagie, T., Manzini, G., Sirén, J.: Wheeler graphs: A framework for bwt-based data structures. Theor. Comput. Sci. **698**, 67–78 (2017)
16. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation - international edition (2. ed). Addison-Wesley (2003)
17. Jacobson, G.J.: Succinct static data structures. PhD thesis, Carnegie Mellon University (1998)
18. Liskovets, V.A.: Exact enumeration of acyclic deterministic automata. Discrete Applied Mathematics **154**(3), 537–551 (2006)
19. Munro, J.I., Raman, V.: Succinct representation of balanced parentheses and static trees. SIAM J. Comput. **31**(3), 762–776 (2001)
20. Navarro, G., Sadakane, K.: Fully functional static and dynamic succinct trees. ACM Transactions on Algorithms **10**(3), 16 (2014)
21. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Algorithms **3**(4), 43 (2007)
22. Sipser, M.: Introduction to the theory of computation. PWS Publishing Company (1997)
23. Sumigawa, K., Sadakane, K.: An efficient representation of partitions of integers. In: IWOCA. pp. 361–373 (2018)