Succinct representation for (non)deterministic finite automata

Sankardeep Chakraborty, Roberto Grossi, Kunihiko Sadakane, Srinivasa Rao Satti

Journal of Computer and System Sciences 131 (2023) 1–12

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Succinct representation for (non)deterministic finite automata ☆

Sankardeep Chakraborty [a], Roberto Grossi [b], Kunihiko Sadakane [a], Srinivasa Rao Satti [c,*]

[a] The University of Tokyo, Japan
[b] University of Pisa, Italy
[c] Norwegian University of Science and Technology, Norway

ARTICLE INFO

Article history:
Received 24 February 2022
Received in revised form 14 July 2022
Accepted 18 July 2022
Available online 9 August 2022

Keywords:
Succinct data structures
Deterministic finite automata
Encoding
Time-space tradeoff

ABSTRACT

(Non)-Deterministic finite automata are one of the simplest models of computation studied in automata theory. Here we study them through the lens of succinct data structures. Towards this goal, we design a data structure for any deterministic automaton $\mathcal{D}$ having n states over a σ-letter alphabet Σ using $(\sigma-1)n\log n(1+o(1))$ bits, that determines, given a string x, 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. Here some of our results are better than the recent work of Cotumaccio and Prezza (SODA 2021). We also exhibit a data structure for non-deterministic automaton $\mathcal{N}$ using $\sigma n^2 + n$ bits that takes $O(n^2|x|)$ time for string membership checking. Finally, we also provide time and space efficient algorithms for performing several standard operations on the languages accepted by finite automata.

1. Introduction

Automata theory is a branch of theoretical computer science that deals exclusively with the definitions, properties and applications of different mathematical 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 long time back is called the *finite automata*. It comes in two different types, *deterministic finite automata* (henceforth DFA) and *non-deterministic finite automata* (henceforth NFA). There exists more complex and sophisticated models as well, for example, *Context-free grammar, 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*; a finite set of letters,
- Q is the finite set of *states*,
- $q_0 \in Q$ is the *initial state*,
- $\delta : Q \times \Sigma \to Q$ is the *transition function* and

☆ A preliminary version of this work appeared in 14th-15th International Conference on Language and Automata Theory and Applications (LATA 2021).

* Corresponding author.
E-mail address: srinivasa.r.satti@ntnu.no (S.R. Satti).

https://doi.org/10.1016/j.jcss.2022.07.002

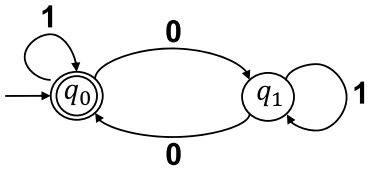

Fig. 1. The *state transition diagram* for a DFA $\mathcal{D}$ where $\mathcal{D} = (\Sigma, Q, q_0, \delta, F)$ such that (i) $\Sigma = \{0, 1\}$, (ii) $Q = \{q_0, q_1\}$, (iii) $q_0 = q_0$ (marked with an incoming arrow coming from nowhere), (iv) $F = \{q_0\}$, and (v) the transition function is defined as the following set, $\{\delta(q_0, 1) = q_0, \delta(q_0, 0) = q_1, \delta(q_1, 1) = q_1, \delta(q_1, 0) = q_0\}$. Precisely the DFA $\mathcal{D}$ accepts all the strings containing an even number of zeros over the binary alphabet.

- $F \subseteq Q$ is the *set of final states*.

We often extend the transition function to $\delta : Q \times \Sigma^* \to Q$ 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 $\delta(q_0, x) \in F$. 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 $\mathcal{L}(\mathcal{D})$. See Fig. 1 for a simple example. In the rest of this paper, we assume that the alphabet Σ is $\{0, 1, 2, \ldots, \sigma - 1\}$, and the state set Q is $\{q_0, q_1, \ldots, q_{n-1}\}$.

A deterministic automaton $\mathcal{A}$ is called *acyclic* [1] 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: $Q \times \Sigma \to Q$ whereas for NFA, the same is defined as δ: $Q \times \Sigma \to \mathcal{P}(Q)$ 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 [2,3] for a thorough discussion on these mathematical models.

Even if a DFA is defined as an abstract mathematical concept, still it has got myriad of practical applications. More specifically, it is used in text processing, compilers, and hardware design [3]. 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 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 not aware of, to the best of our knowledge, any study of DFAs and NFAs from the point of view of *succinct data structures* 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 still 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.

1.1. Related work

The field of *succinct data structures* originally started with the work of Jacobson [4], 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. A partial list of such combinatorial objects would be trees [5,6], various special graph classes like planar graphs [7], chordal graphs [8], partial k-trees [9], circle graphs [10], series-parallel graphs [11], bounded clique-width graphs [12], bounded treewidth graphs [9], interval graphs [13] along with arbitrary general graphs [14], permutations [15], functions [15], bitvectors [16], posets [17] among many others. Regarding encoding DFA/NFA, the work on Wheeler NFA [18] can be considered as an attempt to encode particular classes of NFA succinctly, and this work has recently been generalized to arbitrary NFA in [19]. In the last part of our paper, we will compare these results with ours. We refer the reader to the recent book by Navarro [20] for a comprehensive treatment of this field. The study of succinct data structures is motivated by both theoretical curiosity and also by the practical needs as these combinatorial structures do arise quite often in various applications.

For DFA and 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 and NFAs. Since the fifties there are plenty of attempts in exactly counting the number of DFAs and NFAs with n states over the alphabet Σ, and the state-of-the-art result is due to [21] for DFAs and [22] for NFAs respectively. We refer the readers to the survey (and the references therein) of Domaratzki [23] 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 data structures whose size

matches the lower bounds, hence consuming optimal space, along with capable of executing algorithms efficiently using this succinct representation, and this is the main contribution of this paper.

1.2. DFA and NFA enumeration

After a number of efforts by several authors, finally Bassino and Nicaud [21] 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 with n (including a fixed initial and one or possibly more final) states over an alphabet Σ (where $|\Sigma| = \sigma$) is $\Theta(n2^{2n}S_2(\sigma n, n))$ where $S_2(n, m)$ denotes the Stirling numbers of the second kind.[1] Using the approximation of the Stirling numbers of the second kind [24], 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. [22] 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.

1.3. 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 $n\sigma \log n$ ($O(n^2\sigma \log n)$ 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, we start by listing all the preliminary data structures and graph theoretic terminologies that will be required in our paper in Section 2. Then, in Section 3.1 we first discuss the relevant prior work from [21], 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 3.3, we overcome this problem by designing a succinct data structure for DFA, which can also check the string acceptance optimally. We summarize our main result in the following theorem.

Theorem 1.1. *Given an initially-connected 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:*

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

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

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

The upper bounds in Theorem 1.1 save roughly $n \log n$ bits with respect to the straightforward representation of the DFA. The former upper bounds (Items *1* and *2*) are optimal as they match the information-theoretical lower bound in Section 1.2, up to lower order terms. As for the latter upper bounds (Items *3* and *4*), we do not know their 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.[2] 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 (since we are only considering initially-connected DFAs, and also since every node in the DFA should be reachable from the initial state; these conditions need not hold for edge-labeled-deterministic graphs).

Recently, Cotumaccio and Prezza [19] gave an encoding for DFAs using $\log p + \log \sigma + 2$ bits per transition (where $1 \leq p \leq n$ is a compressibility parameter); this also does not support fast queries on the DFA. For obtaining this efficient

[1] It is defined recursively as $S_2(0, 0) = 1$, $S_2(n, 0) = 0$ for all $n \geq 1$ and for all $n, m \geq 1$, $S_2(n, m) = mS_2(n-1, m) + S_2(n-1, m-1)$.

[2] 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 [25] 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 is 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 $d_u \leq \sigma$.

encoding, they make use of the following crucial assumption that the input DFAs are input-consistent, that is, all edges reaching the same state have the same label. Note that, as mentioned also in Cotumaccio and Prezza [19], one can enforce the input-consistency condition on the input DFA by simply replacing every state with σ copies of itself without changing the language. Thus, for arbitrary DFAs, their space usage is in fact, $\sigma(\log p + \log\sigma + 2)$ bits per transition (since making the original DFA to be input-consistent can blow-up its size, in particular the number of transitions, by a factor of σ in the worst-case). Note that the space usage of our structure (from the last item in Theorem 1.1) can be stated as $\log n + \log\sigma + 1.45$ bits per transition. Thus, their result does not directly compare with ours. In particular, our structure supports fast queries; and uses much less space than [19] for wide range of values of n, σ and p.

We can improve the above space bound if the given DFA is acyclic. More specifically, in Section 3.4, we obtain the following result in this case.

Theorem 1.2. *Given an initially-connected acyclic deterministic finite automaton $\mathcal{A}$ 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 in Section 3.5 where we prove the following result.

Theorem 1.3. *Given an initially-connected non-deterministic finite automaton $\mathcal{N}$ having n states and working over an alphabet Σ of size σ, there exists a succinct encoding for $\mathcal{N}$ taking $\sigma n^2 + n$ 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 in Section 4 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 [2,3], and here we provide a time and space efficient algorithm for performing this construction. More specifically, we show the following theorem.

Theorem 1.4. *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.*

Finally, we conclude in Section 5 with some final remarks.

2. Preliminaries

In this section we collect all the previous theorems and definitions that will be used throughout this paper.

2.1. Graph terminology and graph algorithms

We will assume the knowledge of basic graph theoretic terminology, like trees, paths etc., as given in [26] and basic graph algorithms mostly the depth first search (henceforth DFS), traversal of a graph and its related concepts as given in [27]. Perhaps at this point it may seem slightly unusual that we are talking about graphs here when the focus of this paper is DFA/NFA and their succinct representations. Essentially in this paper we view DFA/NFA, more specifically their graphical representation i.e., *state transition diagram*, as a special case of an edge labeled directed graph G having n nodes corresponding to the $n = |Q|$ states of DFA/NFA, $m = \sigma n$ edges where $|\Sigma| = \sigma$ as each node has exactly σ outgoing edges, and each edge is labeled with some elements from Σ. It is with this point of view, we will design our succinct data structures for DFA/NFA in this paper.

2.2. Succinct data structures

Rank-Select. For a bit vector B and any $a \in \{0,1\}$, the rank and select operations are defined as follows:

- $rank_a(B,i)$ = the number of occurrences of a in $B[1,i]$, for $1 \leq i \leq n$;
- $partial\_rank_1(B,i)$ = $rank_1(B,i)$ if $B[i]=1$, and -1 otherwise; and
- $select_a(B,i)$ = the position in B of the i-th occurrence of a, for $1 \leq i \leq n$.

We make use of the following theorems:

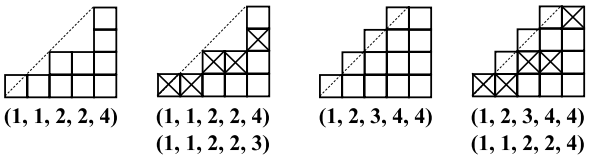

Fig. 2. A diagram of width $m = 5$ and height $n = 4$, a boxed diagram, a k-Dyck diagram and a k-Dyck boxed diagram with $k = 2$.

Theorem 2.1. *[28] We can store a bitstring B of length n with additional $o(n)$ bits such that rank and select operations can be supported in $O(1)$ time. Such a structure can also be constructed from the given bitstring in $O(n)$ time and space.*

Theorem 2.2. *[16] We can store a bitstring B of length n with m ones using $\log\binom{n}{m} + o(m) + O(\log\log n)$ bits such that $partial\_rank_1$ operations can be supported in $O(1)$ time. Such a structure can also be constructed from the given bitstring in $O(n)$ time and space.*

Theorem 2.3. *[29] We can store a bitstring B of length n from universe U (where $|U| = u$) using $\log\binom{u}{n} + O(n)$ bits such that $select_1$ can be supported in $O(1)$ time and rank operations in $O(\log(u/n))$ time.*

Succinct tree representation. We use following result from [5].

Theorem 2.4. *[5] Given a rooted ordered tree τ on n nodes, it can be succinctly represented as a sequence of balanced parenthesis of length $2n$ bits, such that given a node v, we can support subtree size and various navigational queries (such as parent and i-th child) on v in $O(1)$ time using an additional $o(n)$ bits. Such a structure can also be constructed in $O(n)$ time and space.*

Representation of a vector. We also make use of the following theorem from [30].

Theorem 2.5. *[30] There exists a data structure that can represent a vector $A[1..n]$ of elements from a finite alphabet Σ using $n\log|\Sigma| + O(\log^2 n)$ bits, such that any element of the vector can be read or written in constant time.*

3. 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.

3.1. Succinct encoding of DFA

Bassino and Nicaud [21] 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 [21] for complete details regarding the bijection, counting and many other details that we choose to not repeat here. However, we still need to provide some details/definitions (which basically follow their exposition) that are relevant to our own work and will also help to understand the results from their paper smoothly. Later, Almeida et al. [31] also analyzed the construction of Bassino and Nicaud [21] 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 [21] which is then used in Section 3.3 for efficient membership query support as well.

Following [21], 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 $x_m = n$, represented as a diagram of boxes. See Fig. 2 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 $1 \le i \le m$), the y_i-th box of the column i of the diagram is marked. Note that $1 \le y_i \le x_i$. Thus, a diagram can lead to $\prod_{i=1}^{m} x_i$ boxed diagrams. A *k-Dyck diagram* of height n is defined as a diagram of width $m := (k-1)n + 1$ and height n such that $x_i \ge \lceil i/(k-1) \rceil$ for all $i \le m - 1$. Finally, a *k-Dyck boxed diagram* of height n is boxed diagram where the first coordinate $(x_1, \ldots, x_{(k-1)n+1})$ is a k-Dyck diagram of height n. Given these definitions, Bassino and Nicaud [21] proved the following theorem.

Theorem 3.1. *[21] The set $\mathcal{D}_n$ containing 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 height n. Moreover, the construction involving going from transition diagram of the DFA to k-Dyck boxed diagram and vice versa runs in linear time and space.*

$\Sigma = \{a, b, c\}$
$Q = \{1,2,3,4,5,6,7\}$
$q_0 = \{1\}$
$F = \{7\}$

δ	a	b	c
1	2	7	6
2	3	5	4
3	1	4	4
4	2	3	1
5	3	6	1
6	4	2	3
7	4	5	3

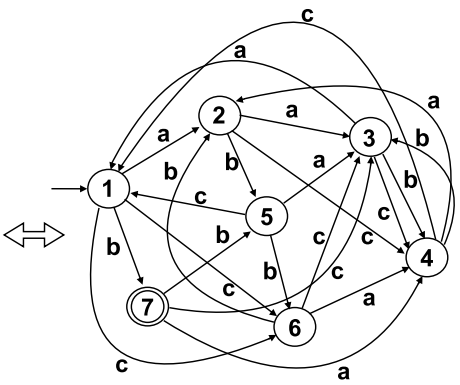

Fig. 3. Two ways to define the same DFA. This DFA will serve as the working example for our discussion. By using the techniques of [21], this DFA can be entirely represented by the $Max[1..15] = \{3, 4, 4, 4, 4, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7\}$ and $Boxed[1..15] = \{1, 2, 3, 1, 4, 3, 4, 2, 3, 1, 4, 4, 5, 3, 6\}$ arrays of length $(\sigma - 1)n + 1 = 15$ each.

Thus, by applying the above theorem, from any given DFA with n states and σ-letter alphabet, one can construct a σ-Dyck boxed diagram of height n using a depth-first search of the DFA [21]. As mentioned earlier, this σ-Dyck boxed diagram is a pair of sequences $((x_1, \ldots, x_m), (y_1, \ldots, y_m))$, where $m := (\sigma - 1)n + 1$. We store these two sequences in two integer arrays $Max[1..m]$ and $Boxed[1..m]$, respectively, of length m each. Furthermore, from these two arrays, it is possible to entirely reconstruct the DFA using the algorithm of Theorem 3.1. Thus, it is sufficient to store just these two arrays in order to encode any given DFA. For more details, readers are referred to [21]. For an example, see Fig. 3 which will also serve as the working example for this part of our paper. In particular, the DFA of Fig. 3 can be entirely encoded 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 $(\sigma - 1)n + 1 = 15$, and these can be computed using the algorithms of [21].

First, we observe that, by construction, the arrays satisfy $1 \leq Max[1] \leq Max[2] \leq \cdots \leq Max[m] \leq n$ 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 is 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 Theorem 2.2, 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 Theorem 2.5 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.

3.2. Reducing the space further

Now consider the case when there is a failure 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 Theorem 2.1, 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 Theorem 2.2, the bitvector Z can be represented in $\log \binom{\sigma n}{N} + o(N) + O(\log\log(\sigma n)) = N \log \frac{\sigma ne}{N} + o(N) \leq N \log \sigma + N \log e + o(N)$ 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 3.1 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 [21], and Almeida et al. [31]) 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.

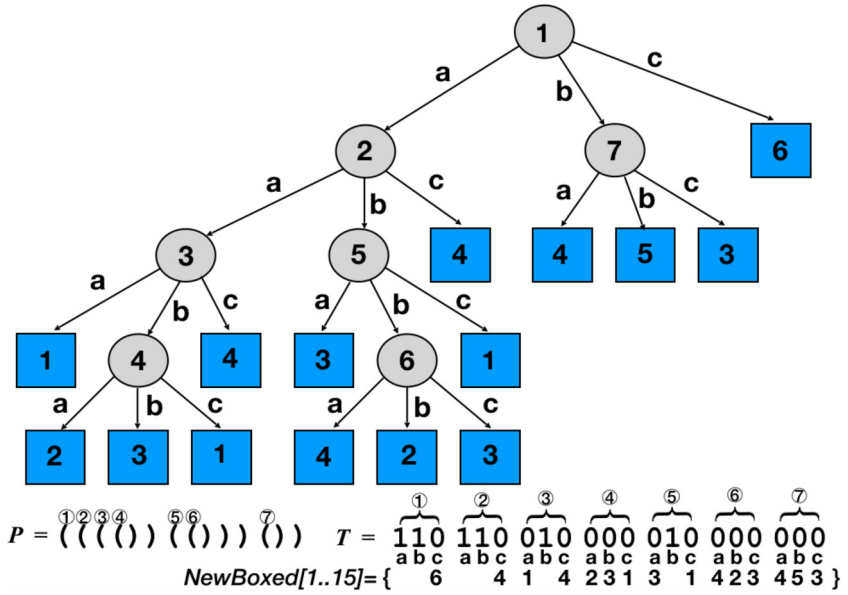

Fig. 4. The extended lex-DFS tree S of the automaton of Fig. 3 along with the corresponding bitvectors P, T, and the *NewBoxed*[1..15] array (the elements of this array are drawn exactly below the corresponding 0s with which they share one to one correspondence with). Note that, for the same automaton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

3.3. Succinct data structure for DFA

Data structure: 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 Fig. 3, 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 Theorem 2.4, 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.

Before explaining the other bitvector, T, required for our succinct encoding, we want to explain the contents of Fig. 4. 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* [27] (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 Fig. 4), 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 represent 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 Fig. 4 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 ($1 \le i \le n$), 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 ($1 \le c \le \sigma$). 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 two different ways to represent the T array. (1) In the first method, we simply encode T using the structure of Theorem 2.1 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 encode T using the data structure of Theorem 2.3 which takes $\log\binom{n\sigma}{n-1} + O(n) = n\log\sigma + O(n)$ bits while supporting the *rank* query in $O(\log\sigma)$ 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 nodes 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 Fig. 4 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 Theorem 2.4, hence it occupies $2n + o(n)$ bits. Depending on the two choices for storing T, the space requirement is (1) $\sigma n + o(\sigma n)$ or (2) $n\log\sigma + O(n)$ bits. Thus, the overall space usage is as stated in the Theorem 1.1.

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 3.2) 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 two choices as above for representing T, using (1) $\sigma n + o(\sigma n)$ or (2) $N\log\sigma + O(N)$ bits. Thus the overall space usage is (1) $(N - n)\log n + O(n\sigma)$, (2) $(N - n)\log n + (N + n)\log\sigma + O(N)$ 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 $1 \le q, q' \le n$ 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 Theorem 2.4). 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 two choices for storing the T array and supporting *rank*/*select* queries in it, we obtain two different query time bounds for accepting x. In option (1), the input string x can be accepted optimally in $O(|x|)$ time. In option (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 $O(|x|\log\sigma)$. 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 1.1.

3.4. Succinct data structures for acyclic DFA

As mentioned previously, an acyclic DFA $\mathcal{A}$ with total n states always has a unique dead state and $n - 1$ transient (i.e., non dead) states. Another way to visualize $\mathcal{A}$ is to see that the state transition diagram of $\mathcal{A}$ does not have any cycles except at the unique dead state. Given such a setting, one can always use the succinct encoding (of the previous section) of an arbitrary DFA to represent them. In that case, we end up using $(\sigma - 1)n\log n + O(n\log\sigma)$ bits of space. In what follows, we show that by exploiting the acyclic property, one can obtain improved space bound for representing $\mathcal{A}$.

We basically view the state transition diagram of $\mathcal{A}$ as a directed acyclic graph with a single source (i.e., the initial state), and a single sink i.e., the dead state (call it d). Given this, we first construct a spanning tree $W = (V, E)$ of $\mathcal{A}$ where $V = Q$ (i.e., the set of states of $\mathcal{A}$) and $E = \{(q_u, q_v) \mid \delta(q_v, c) = q_u \text{ where } q_v \neq d \text{ and for some } c \in \Sigma\}$ by making the dead state d as the root of this tree. One can construct such a spanning tree by performing a depth first search staring from d in time linear in the size of $\mathcal{A}$.

By applying Theorem 2.4, we encode the structure of W using $2n + o(n)$ bits to support the navigational queries on W (in particular, the parent query) in $O(1)$ time. As done previously in Section 3.3 while constructing the succinct data structures for DFA, here also we relabel all the states of $\mathcal{A}$ such that the i-th vertex (state) in W in preorder has label i, and modify the transition function accordingly. Note that the dead state d is labeled with label 0 in this ordering, and we do not need to store the transition function for the dead state. We also mark in a bitvector of size n all the final states of $\mathcal{A}$, and we store the label of the start state. We then store a two dimensional array $L[1..n-1][1..\sigma-1]$ such that $L[q][i] = \delta(q, i)$ using data structure of Theorem 2.5. Thus, the overall space usage is $(\sigma - 1)(n - 1)\log n + 3n + O(\log^2\sigma) + o(n)$ bits.

In what follows, we explain how to check if $\mathcal{A}$ accepts any given string x over Σ. At any generic step, to compute $\delta(q, i)$, we simply output $L[q][i]$ if $i \in \{1, 2, \ldots, \sigma - 1\}$; otherwise (i.e., if $i = \sigma$) the value of $\delta(q, \sigma)$ is given by the parent of q in W i.e., $\delta(q, i) = parent(q)$. Thus $\delta(q, i)$ can be computed in constant time, and hence we can optimally decide if $\mathcal{A}$ accepts x in time proportional to the length of x. This completes the proof of Theorem 1.2.

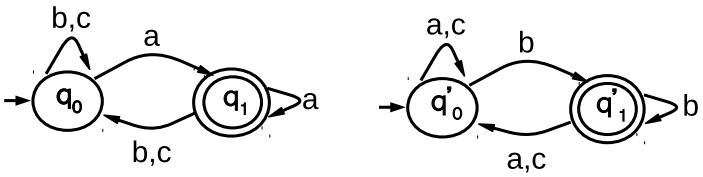

Fig. 5. The state transition diagram for a DFA $\mathcal{D}_1$ (on the left) where $\mathcal{D}_1 = (\Sigma, Q, q_0, \delta, F)$ such that (i) $\Sigma = \{a, b, c\}$, (ii) $Q = \{q_0, q_1\}$, (iii) $q_0 = q_0$ (marked with an incoming arrow coming from nowhere), (iv) $F = \{q_1\}$, and (v) the transition function is defined as the following set, $\{\delta(q_0, a) = q_1, \delta(q_0, b) = q_0, \delta(q_0, c) = q_0, \delta(q_1, a) = q_1, \delta(q_1, b) = q_0, \delta(q_1, c) = q_0\}$. Precisely the DFA $\mathcal{D}_1$ accepts all the strings that end with an a over Σ. Similarly the state transition diagram for a DFA $\mathcal{D}_2$ (on the right) where $\mathcal{D}_2 = (\Sigma, Q', q'_0, \delta', F')$ such that (i) $\Sigma = \{a, b, c\}$, (ii) $Q = \{q'_0, q'_1\}$, (iii) $q'_0 = q'_0$ (marked with an incoming arrow coming from nowhere), (iv) $F = \{q'_1\}$, and (v) the transition function is defined as the following set, $\{\delta(q'_0, a) = q'_0, \delta(q'_0, b) = q'_1, \delta(q'_0, c) = q'_0, \delta(q'_1, a) = q'_0, \delta(q'_1, b) = q'_1, \delta(q'_1, c) = q'_0\}$. Precisely the DFA $\mathcal{D}_2$ accepts all the strings that end with a b over Σ. These two DFAs will serve as the working example for our discussion of the product automaton construction.

3.5. Succinct encoding for NFA

As mentioned previously in Section 1.2, to encode an initially connected NFA on n states over a σ-letter alphabet Σ with a fixed initial state and one or more final states, we need at least $\sigma n^2 + n$ bits. In what follows, we show a very simple scheme achieving this bound.

We store a table H having n rows (corresponding to the n states of the input NFA) and σ columns (corresponding to each letter of the alphabet Σ). The entry $H[i][j]$ (where $0 \leq i \leq n-1$ and $1 \leq j \leq \sigma$) basically stores the corresponding transition function of the NFA i.e., $H[i][j] = \delta(q_i, j)$ where $q_i \in Q$ and $j \in \Sigma$. Now for an NFA, $\delta(i, j)$ is a subset of Q. If we store this subset explicitly, it might take $O(n \log n)$ bits in the worst case per transition of the NFA, leading to overall $\sigma n^2 \log n$ bits which is $O(\log n)$ multiplicative factor off from the optimal space requirement. Instead we simply store the charecteristic vector L of the subset (of length n, marking the corresponding states from the subset as 1, and rest of the bits in L are 0) where the state labeled i of the NFA moves to after reading the letter $j \in \Sigma$. Thus, the overall size of H is exactly σn^2 bits. Finally, we also mark in a separate bitvector (of length n) all the final states of the input NFA. Thus, in total the size of our encoding is given by $\sigma n^2 + n$ bits, which matches the lower bound. Hence, our encoding is succinct and optimal.

Now using our encoding, we can simply implement the classical algorithm (given in the texts of [2,3]) for checking if the NFA accepts a given input string or not, and this runs in $O(n^2|x|)$ time where x is the input string and $|x|$ denotes its length. Note that we also need two bitvectors of length n each (hence overall $2n$ bits) as working space to mark two sets of intermediate states between successive transitions while executing the string acceptance checking algorithm. Hence, we obtain the result mentioned in Theorem 1.3.

4. Operations on DFAs

In what follows we show how to support some standard operations namely union, intersection and complement on DFAs space efficiently. We start with the classical example of *product automaton* construction. More specifically, given the succinct representation of two DFAs, we want to construct a succinct representation of the product automaton accepting the language which is the union/intersection of the two input DFA's language.

Before providing our construction, let us formally define the product automaton construction. Suppose, we are given two DFAs $\mathcal{D}_1 = (\Sigma, Q, q_0, \delta, F)$ and $\mathcal{D}_2 = (\Sigma, Q', q'_0, \delta', F')$ represented succinctly (as described in Section 3.3) and both working over the same alphabet Σ. Then a product automaton (denoted by $\mathcal{P}$) of $\mathcal{D}_1$ and $\mathcal{D}_2$ is defined as follows, $\mathcal{P} = (\Sigma, \mathcal{Q}, (q_0, q'_0), \delta_p, F_p)$ where $\mathcal{Q} = Q \times Q'$, and $\delta_p : \mathcal{Q} \times \Sigma \to \mathcal{Q}$. Moreover, for any $q \in Q, q' \in Q'$ and $c \in \Sigma$, $\delta_p((q, q'), c) := (\delta(q, c), \delta'(q', c))$. The start state of $\mathcal{P}$ is the pair (q_0, q'_0) whereas the final state can be defined in multiple ways. More specifically, if we set $F_p = F \times F'$, then $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{D}_1) \cap \mathcal{L}(\mathcal{D}_2)$. Similarly, if we set $F_p = (F \times Q') \cup (Q \times F')$, then $\mathcal{L}(\mathcal{P}) = \mathcal{L}(\mathcal{D}_1) \cup \mathcal{L}(\mathcal{D}_2)$. Now we show how one can directly construct a succinct representation of $\mathcal{P}$ given the succinct representations of $\mathcal{D}_1$ and $\mathcal{D}_2$ as input, and note that, to do so we just need to describe how one can create the three bitvectors F, P, T and the integer array $NewBoxed[1..m]$ corresponding to $\mathcal{P}$ from the succinct representations of $\mathcal{D}_1$ and $\mathcal{D}_2$ directly. See Fig. 5 and Fig. 6 for a visual description of our product automaton construction algorithm.

4.1. Product automation construction

For constructing the product automaton $\mathcal{P}$, our high level idea is to create the states and transitions of $\mathcal{P}$ by generating the states of $\mathcal{P}$ in the lexicographic DFS order using two passes. In the first pass, we generate the P and T arrays (both initialized with empty string), and this is followed by the construction of the $NewBoxed[1..m]$ array in the second pass. More specifically, we start by creating the initial state i.e., (q_0, q'_0) as the first circled node i.e., root in the extended lex-DFS tree corresponding to $\mathcal{P}$, store an entry corresponding to this node in the hash table along with storing its preorder number (which is 1 in the case of (q_0, q'_0)) as a satellite data in the hash table. Also we append σ zero bits to T corresponding to the root. In general, at any point of time during the execution of this algorithm, the hash table stores an entry corresponding to each of the circled nodes generated up to that point along with storing its preorder number and its parent node as satellite

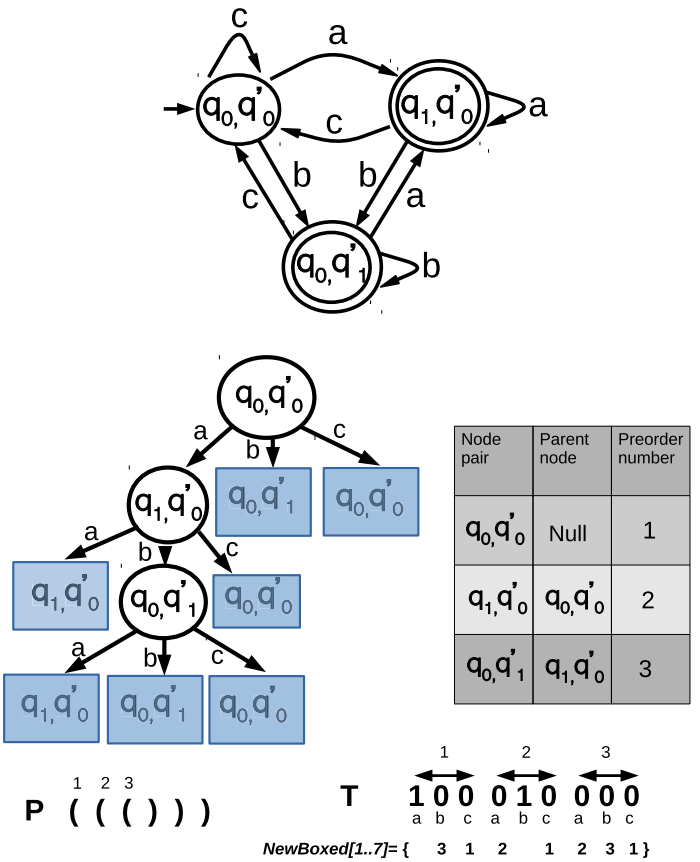

Fig. 6. The state transition diagram on the top depicts the product automaton $\mathcal{P}$ accepting the language $\mathcal{L}(\mathcal{D}_1) \cup \mathcal{L}(\mathcal{D}_2)$ (DFA $\mathcal{D}_1$ and $\mathcal{D}_2$ are defined in Fig. 5). The diagram on the bottom left depicts the extended lex-DFS tree of the product automaton $\mathcal{P}$ (defined above) whereas the rest of diagram contains the description of the other data structures i.e., the bitvectors P, T, the integer array *NewBoxed*[1..7], and finally the hash table. Note that the elements of the *NewBoxed*[1..7] array are drawn exactly below the corresponding 0s with which they share one to one correspondence.

data. Note that for the root, we don't need to store any parent information. Now to figure out the transitions out of any state, note that, if we use the method described in the query algorithm for DFA (as described in Section 3.3) we need to pay $O(\log \sigma)$ time per symbol of the alphabet Σ.

Instead, in what follows, we show how one can find each transition in $O(1)$ time per symbol out of any state using all the information that is already stored in the input i.e., succinct representations for $\mathcal{D}_1$ and $\mathcal{D}_2$. Assume for now that we can do so and also suppose that at some point of the algorithm, we created a new circled node (i, i'). Then we proceed as follows. First we append σ zero bits to the bit string T corresponding to the node (i, i'). This is followed by the *expansion* of the state (i, i') by generating the transitions $\delta_p((i, i'), c)$ in the lexicographic ordering of the alphabet characters $c \in \Sigma$, as follows.

Let $j = \delta(i, c)$ and $j' = \delta'(i', c)$, then we check in the hash table if the state (j, j') has already been created before (by checking membership in the hash table). If yes, we create a squared node (j, j') as a child node of (i, i') (which is a circled node) and don't make any changes to the P array, mark the c-th bit corresponding to the node (i, i') in T as zero; and continue with the expansion of (i, i') with the next character in Σ. If not, we create a circled node (j, j') as a child of (i, i'), append an open parenthesis to the P array constructed so far, mark the c-th bit corresponding to the node (i, i') in T as one, and finally insert (j, j') into the hash table along with inserting (i, i') as its parent and its preorder number as its satellite data; and continue with the expansion of (j, j'). Finally, when we exhaust checking all the characters $c \in \Sigma$ out of (i, i'), we backtrack to the parent of (i, i') in the extended lex-DFS tree (using the parent information stored as a satellite data with the entry for the node (i, i')), and in this case, we simply append a close parenthesis to the P array constructed so far.

It is clear that using this procedure repeatedly we can successfully create P and T arrays corresponding to the product automaton $\mathcal{P}$. Finally, we create the all the auxiliary structures (mentioned in Section 2.2) on top of the arrays P and T (similar to the succinct data structure for DFA as described in Section 3.3) for supporting various navigational queries on the extended lex-DFS tree. Intuitively the P array stores the topology of the extended lex-DFS tree of the state transition diagram of the product automaton $\mathcal{P}$ and the T array stores the parent-child relationship between the nodes of the extended lex-DFS tree in a compact manner.

Now let's discuss how to find out the transitions efficiently. Note that it suffices to describe how one can find $j = \delta(i, c)$ in $\mathcal{D}_1$ ($j' = \delta'(i', c)$ in $\mathcal{D}_2$ can be found similarly). We consider the two cases: when the edge (i, j) is a (i) non-tree edge, or a (ii) tree edge. In case (i), $j = NewBoxed[rank_0(T, \sigma(i-1)+c)]$. In case (ii), $j = child(i, t)$ (can be supported using the Lemma 2.4 on the P array) where $t = rank_1(T, \sigma(i-1)+c) - rank_1(T, \sigma(i-1))$.

In what follows, we describe how to fill up the integer array $NewBoxed[1..m]$ with m (we discuss about fixing m later) entries which are initialized with all one. Note that, similar to the succinct DFA construction, this array should contain the preorder number of the node labels in the squared nodes of the extended lex-DFS tree in the order of their appearance in the T bitvector (from left to right). Moreover, these nodes are marked by 0s in T and they are in one-to-one correspondence with all the non-tree edges of the extended lex-DFS tree of the product automaton $\mathcal{P}$. To fill up $NewBoxed[1..m]$ array, we follow essentially the same lexicographic DFS traversal procedure as we did in the first pass except the following. More specifically, we start the second pass of the extended lex-DFS tree and whenever we encounter a non-tree edge, we retrieve the preorder number corresponding to the node label in the squared node (i.e., the other end point of that non-tree edge) from the hash table, and insert this number at the suitable position in the $NewBoxed$ array. In detail, suppose we are at a circled node (i, i') (with preorder number, say, k) and currently exploring the transition with the letter $c \in \Sigma$ out of (i, i'). Also assume that $\delta_p((i, i'), c) = (j, j')$ and (j, j') is a squared node (i.e., $((i, i'), (j, j'))$ is a non-tree edge) such that the preorder number associated with the node label (j, j') is d in the hash table. Then, we assign $NewBoxed[\ell] = d$ where $\ell = rank_0(T, \sigma(k-1)+c)$. Finally, depending on union or intersection operation, we also mark in another bitvector F (according to the definition given above) all the final states of the product automaton $\mathcal{P}$. Observe that once we have all the constituent data structures (including all the auxiliary data structures that we build on top of F, P, T arrays and the integer array $NewBoxed[1..m]$) for the succinct representation for $\mathcal{P}$ ready, we can essentially use the same query algorithm for string acceptance checking as we described for DFA in Section 3.3.

4.2. Time and space analysis

Let's analyze the resource requirements for our algorithm. Suppose $|Q| = n$ and $|Q'| = n'$, then the product automaton $\mathcal{P}$ can have nn' states at the worst case, but in general it could be much less as well. Let us suppose that $\mathcal{P}$ has n'' states, then $n'' \leq nn'$, and in what follows, we write our space requirement as a function of n''. If we implement the hash table using the data structure of [32], then it consumes $O(n'' \log n'')$ bits in total. Also note that this is the dominating term for the working space bound as other auxiliary data structures consume negligible space with respect to the space consumption for the hash table. Moreover, our algorithm runs in linear, i.e., $O(\sigma n'')$ expected time overall.

The randomized nature of our algorithm is due to the fact of using the hashing data structure of [32] whereas all the other parts of our algorithm are deterministic. As a result of our algorithm, we generate a representation for $\mathcal{P}$ and this is given by the following arrays. The bitvectors P and F consume $2n'' + o(n'')$, n'' bits respectively. For the T array, we compress it by using the data structure of Theorem 2.3 using $O(n'' \log \sigma)$ bits. Finally, the $NewBoxed$ array has m entries where $m = (\sigma - 1)n'' + 1$ and each entry could be up to n''. Thus, using the data structure of Lemma 2.5, $NewBoxed[1..m]$ can be encoded using $(\sigma - 1)n'' \log n'' + O(\log^2 m)$ bits. Thus, our algorithm produces a representation of the product automaton $\mathcal{P}$ using $(\sigma - 1)n'' \log n'' + O(n'' \log \sigma)$ bits overall, and this is succinct. This completes the description of the product automaton construction algorithm as stated in Theorem 1.4.

Remark. In the light of the above discussion, consider the following. Suppose we are given as input a succinct representation for a DFA $\mathcal{D}$ whose language is $\mathcal{L}(\mathcal{D})$, and our goal is to construct the succinct representation for the DFA (say $\mathcal{D}'$) which accepts complement of $\mathcal{L}(\mathcal{D})$ i.e., $\mathcal{L}(\mathcal{D}') = \Sigma^* - \mathcal{L}(\mathcal{D})$. In order to construct the succinct representation for $\mathcal{D}'$, we start with the succinct representation for $\mathcal{D}$ (that is given in terms of three bit vectors F, P, T and the integer array $NewBoxed[1..m]$), and simply convert (in the F array) each final state in $\mathcal{D}$ into a non-final state in $\mathcal{D}'$ and convert each non-final state in $\mathcal{D}$ into a final state in $\mathcal{D}'$ without changing any other data structures. As a consequence, it is easy to see that, we will end up with what we desired.

5. 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. Another direction would be to prove lower bounds for the query time vs space trade-off.

CRediT authorship contribution statement

All the authors contributed roughly equally in obtaining the technical results, as well as the writing of the paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] V.A. Liskovets, Exact enumeration of acyclic deterministic automata, Discrete Appl. Math. 154 (2006) 537–551.
[2] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, international edition (2 ed.), Addison-Wesley, 2003.
[3] M. Sipser, Introduction to the Theory of Computation, PWS Publishing Company, 1997.
[4] G.J. Jacobson, Succinct static data structures, PhD thesis, Carnegie Mellon University, 1998.
[5] J.I. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM J. Comput. 31 (2001) 762–776.
[6] G. Navarro, K. Sadakane, Fully functional static and dynamic succinct trees, ACM Trans. Algorithms 10 (2014) 16.
[7] L.C. Aleardi, O. Devillers, G. Schaeffer, Succinct representations of planar maps, Theor. Comput. Sci. 408 (2008) 174–187.
[8] J.I. Munro, K. Wu, Succinct data structures for chordal graphs, in: ISAAC, 2018, pp. 67:1–67:12.
[9] A. Farzan, S. Kamali, Compact navigation and distance oracles for graphs with small treewidth, Algorithmica 69 (2014) 92–116.
[10] H. Acan, S. Chakraborty, S. Jo, K. Nakashima, K. Sadakane, S.R. Satti, Succinct representations of intersection graphs on a circle, in: DCC, IEEE, 2021, pp. 123–132.
[11] S. Chakraborty, S. Jo, K. Sadakane, S.R. Satti, Succinct data structures for small clique-width graphs, in: DCC, IEEE, 2021, pp. 133–142.
[12] S. Chakraborty, S. Jo, K. Sadakane, S.R. Satti, Succinct data structures for series-parallel, block-cactus and 3-leaf power graphs, in: COCOA, in: LNCS, vol. 13135, Springer, 2021, pp. 416–430.
[13] H. Acan, S. Chakraborty, S. Jo, S.R. Satti, Succinct data structures for families of interval graphs, in: WADS, 2019, pp. 1–13.
[14] A. Farzan, J.I. Munro, Succinct encoding of arbitrary graphs, Theor. Comput. Sci. 513 (2013) 38–52.
[15] J.I. Munro, R. Raman, V. Raman, S.S. Rao, Succinct representations of permutations and functions, Theor. Comput. Sci. 438 (2012) 74–88.
[16] R. Raman, V. Raman, S.R. Satti, Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets, ACM Trans. Algorithms 3 (2007) 43.
[17] T. Yanagita, S. Chakraborty, K. Sadakane, S.R. Satti, Space-efficient data structure for posets with applications, in: 18th SWAT, in: LIPIcs, vol. 227, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, pp. 33:1–33:16.
[18] T. Gagie, G. Manzini, J. Sirén, Wheeler graphs: a framework for bwt-based data structures, Theor. Comput. Sci. 698 (2017) 67–78.
[19] N. Cotumaccio, N. Prezza, On indexing and compressing finite automata, in: D. Marx (Ed.), SODA, SIAM, 2021, pp. 2585–2599.
[20] G. Navarro, Compact Data Structures - A Practical Approach, Cambridge University Press, 2016.
[21] F. Bassino, C. Nicaud, Enumeration and random generation of accessible automata, Theor. Comput. Sci. 381 (2007) 86–104.
[22] M. Domaratzki, D. Kisman, J. Shallit, On the number of distinct languages accepted by finite automata with n states, J. Autom. Lang. Comb. 7 (2002) 469–486.
[23] M. Domaratzki, Enumeration of formal languages, Bull. Eur. Assoc. Theor. Comput. Sci. 89 (2006) 117–133.
[24] P. Flajolet, R. Sedgewick, Analytic Combinatorics, Cambridge University Press, 2009.
[25] A. Farzan, J.I. Munro, Succinct encoding of arbitrary graphs, Theor. Comput. Sci. 513 (2013) 38–52.
[26] R. Diestel, Graph Theory, 4th edition, Graduate Texts in Mathematics, vol. 173, Springer, 2012.
[27] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3 ed., MIT Press, 2009.
[28] D.R. Clark, Compact Pat Trees, PhD thesis, University of Waterloo, Canada, 1996.
[29] A. Orlandi, Advanced rank/select data structures: succinctness, bounds, and applications, PhD thesis, 2021.
[30] Y. Dodis, M. Patrascu, M. Thorup, Changing base without losing space, in: STOC, 2010, pp. 593–602.
[31] M. Almeida, N. Moreira, R. Reis, Enumeration and generation with a string automata representation, Theor. Comput. Sci. 387 (2007) 93–102.
[32] M. Dietzfelbinger, A.R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, R.E. Tarjan, Dynamic perfect hashing: upper and lower bounds, SIAM J. Comput. 23 (1994) 738–761.