Random Wheeler Automata

Ruben Becker
Ca' Foscari University of Venice, Italy

Davide Cenzato
Ca' Foscari University of Venice, Italy

Sung-Hwan Kim
Ca' Foscari University of Venice, Italy

Bojana Kodric
Ca' Foscari University of Venice, Italy

Riccardo Maso
Ca' Foscari University of Venice, Italy

Nicola Prezza
Ca' Foscari University of Venice, Italy

Abstract

Wheeler automata were introduced in 2017 as a tool to generalize existing indexing and compression techniques based on the Burrows-Wheeler transform. Intuitively, an automaton is said to be Wheeler if there exists a total order on its states reflecting the natural co-lexicographic order of the strings labeling the automaton's paths; this property makes it possible to represent the automaton's topology in a constant number of bits per transition, as well as efficiently solving pattern matching queries on its accepted regular language. After their introduction, Wheeler automata have been the subject of a prolific line of research, both from the algorithmic and language-theoretic points of view. A recurring issue faced in these studies is the lack of large datasets of Wheeler automata on which the developed algorithms and theories could be tested. One possible way to overcome this issue is to generate random Wheeler automata. Motivated by this observation of practical nature, in this paper we initiate the theoretical study of random Wheeler automata, focusing our attention on the deterministic case (Wheeler DFAs – WDFAs). We start by naturally extending the Erdős-Rényi random graph model to WDFAs, and proceed by providing an algorithm generating uniform WDFAs according to this model. Our algorithm generates a uniform WDFA with n states, m transitions, and alphabet's cardinality σ in $O(m)$ expected time ($O(m \log m)$ time w.h.p.) and constant working space for all alphabets of size $\sigma \leq m/\ln m$. The output WDFA is streamed directly to the output. As a by-product, we also give formulas for the number of distinct WDFAs and obtain that $n\sigma + (n-\sigma)\log\sigma$ bits are necessary and sufficient to encode a WDFA with n states and alphabet of size σ, up to an additive $\Theta(n)$ term. We present an implementation of our algorithm and show that it is extremely fast in practice, with a throughput of over 8 million transitions per second.

2012 ACM Subject Classification Theory of computation → Generating random combinatorial structures; Theory of computation → Sorting and searching; Theory of computation → Graph algorithms analysis

Keywords and phrases Wheeler automata, Burrows-Wheeler transform, random graphs

Digital Object Identifier 10.4230/LIPIcs.CPM.2024.5

Related Version *Full Version*: `https://arxiv.org/abs/2307.07267`

Supplementary Material
Software (Source Code): `https://github.com/regindex/Wheeler-DFA-generation`
archived at `swh:1:dir:8c1d3690d81c3e6600423dfcbe8ce8bffb769f64`

Funding *Ruben Becker, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Nicola Prezza*: Funded by the European Union (ERC, REGINDEX, 101039208). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Wheeler automata were introduced by Gagie et al. in [11] in an attempt to unify existing indexing and compression techniques based on the Burrows-Wheeler transform [5]. An automaton is said to be Wheeler if there exists a total order of its states such that (i) states reached by transitions bearing different labels are sorted according to the underlying total alphabet's order, and (ii) states reached by transitions bearing the same label are sorted according to their predecessors (i.e. the order propagates forward, following pairs of equally-labeled transitions). Equivalently, these axioms imply that states are sorted according to the co-lexicographic order of the strings labeling the automaton's paths. Since their introduction, Wheeler automata have been the subject of a prolific line of research, both from the algorithmic [7, 3, 10, 12, 9, 6, 13] and language-theoretic [2, 1, 8] points of view. The reason for the success of Wheeler automata lies in the fact that their total state order enables *simultaneously* to index the automaton for pattern matching queries and to represent the automaton's topology using just $O(1)$ bits per transition (as opposed to the general case, requiring a logarithmic number of bits per transition).

A recurring issue faced in research works on Wheeler automata is the lack of datasets of (large) Wheeler automata on which the developed algorithms and theories could be tested. As customary in these cases, a viable solution to this issue is to randomly generate the desired combinatorial structure, following a suitable distribution. The most natural distribution, the uniform one, represents a good choice in several contexts and can be used as a starting point to shed light on the combinatorial objects under consideration; the case of random graphs generated using the Erdős-Rényi random graph model [15] is an illuminating example. In the case of Wheeler automata, we are aware of only one work addressing their random generation: the WGT suite [6]. This random generator, however, does not guarantee a uniform distribution over the set of all Wheeler automata.

Our contributions

Motivated by the lack of formal results in this area, in this paper we initiate the theoretical study of random Wheeler automata, focusing our attention on the algorithmic generation of uniform deterministic Wheeler DFAs (WDFAs). We start by extending the Erdős-Rényi random graph model to WDFAs: our uniform distribution is defined over the set $\mathcal{D}_{n,m,\sigma}$ of all Wheeler DFAs over the *effective* alphabet (i.e. all labels appear on some edge) $[\sigma] = \{1, \dots, \sigma\}$, with n states $[n]$, m transitions, and Wheeler order $1 < 2 < \dots < n$. We observe that, since any WDFA can be encoded using $O(n\sigma)$ bits [11], the cardinality of $\mathcal{D}_{n,m,\sigma}$ is at most $2^{O(n\sigma)}$. On the other hand, the number of DFAs with n states over alphabet of size σ is $2^{\Theta(n\sigma \log n)}$ [15]. As a result, a simple rejection sampling strategy that uniformly generates DFAs until finding a WDFA (checking the Wheeler property takes linear time on DFAs [1]) would take expected exponential time to terminate. To improve over this naive solution, we start by defining a new combinatorial characterization of WDFAs: in Section 3, we establish a bijection that associates every element of $\mathcal{D}_{n,m,\sigma}$ to a pair formed by a binary matrix and a binary vector. This allows us to design an algorithm to uniformly sample WDFAs, based on the above-mentioned representation. Remarkably, our sampler uses *constant* working space and streams the sampled WDFA directly to output:

▶ **Theorem 1.** *There is an algorithm to generate a uniform WDFA from $\mathcal{D}_{n,m,\sigma}$ in $O(m)$ expected time ($O(m \log m)$ time with high probability) using $O(1)$ words of working space, for all alphabets of size $\sigma \le m/\ln m$. The output WDFA is directly streamed to the output as a set of labeled edges.*

As a by-product of our combinatorial characterization of WDFAs, in Theorem 19 we give an exact formula for the number $|\mathcal{D}_{n,m,\sigma}|$ of distinct WDFAs with n nodes and m edges labeled from alphabet $[\sigma]$ and in Theorem 20 we give a tight asymptotic formula for the number $|\mathcal{D}_{n,\sigma}|$ of distinct WDFAs with n nodes and any number of edges labeled from $[\sigma]$, obtaining that $n\sigma + (n-\sigma)\log\sigma$ bits are necessary and sufficient to encode WDFAs from such a family up to an additive $\Theta(n)$ term.

We conclude by presenting an implementation of our algorithm, publicly available at `https://github.com/regindex/Wheeler-DFA-generation`, and showing that it is very fast in practice while using a negligible (constant) amount of working space.

2 Preliminaries and Problem Statement

With $\ln x$ and $\log x$, we indicate the natural logarithm and the logarithm in base 2 of x, respectively. For an integer $k \in \mathbb{N}^+$, we let $[k]$ denote the set of all integers from 1 to k. For a bit-vector $v \in \{0,1\}^k$, we denote with $\|v\| = \sum_{i\in[k]} v_i$ the L_1-norm of v, i.e., the number of set bits in v. For an integer $\ell \le k$, we denote with $v[1:\ell]$ the bit-vector $(v_1,\dots,v_\ell)$ consisting only of the first ℓ bits of v. For a bit-matrix $A \in \{0,1\}^{\ell\times k}$ and a column index $j \in [k]$, we denote the j'th column of A by A_j and the element at row i and column j as $A_{i,j}$. We let $\|A\| = \sum_{i\in[k],j\in[\ell]} A_{i,j}$ be the $L_{1,1}$-norm of A, which again counts the number set bits in A. For a bit-vector $v \in \{0,1\}^k$, we use the notation $\mathrm{rank}(v,i)$ to denote the number of occurrences of 1 in $v[1:i]$. For completeness, we let $\mathrm{rank}(v,0)=0$. We generalize this function also to matrices as follows. For a bit-matrix $A \in \{0,1\}^{\ell\times k}$, we let $\mathrm{rank}(A,(i,j)) = \sum_{r\in[j-1]} \mathrm{rank}(A_r,\ell) + \mathrm{rank}(A_j,i)$. We sometimes write bit-vectors from $\{0,1\}^k$ in string form, i.e., as a sequence of k bits.

In this paper we are concerned with deterministic finite automata.

▶ **Definition 2** (Determinisitic Finite Automaton (DFA))**.** *A Determinisitic Finite (Semi-) Automaton (DFA) D is a triple (Q,Σ,δ) where $Q=[n]$ is a finite set of n states with $1\in Q$ being the source state, $\Sigma=[\sigma]$ is the finite alphabet of size σ, and $\delta : Q\times\Sigma \to Q$ is a transition function containing m transitions.*

We omit to specify the final states of DFAs, since they do not play a role in the context of our problem. We use the shorthand $\delta_j(v)$ for $\delta(v,j)$. Furthermore, we write $\delta^{out}(v) := \{\delta_j(v) : j \in \Sigma\}$ for the set of all out-neighbors of a state $v \in Q$ and $\delta^{in}(v) := \{u \in Q : \exists j \in \Sigma \text{ with } v \in \delta_j(u)\}$ for the set of all in-neighbors of v. We assume DFAs to have non-zero in-degree for exactly the non-source states, i.e., $\delta^{in}(v) \neq \emptyset$ if and only if $v > 1$; This choice simplifies our exposition and it is not restrictive from the point of view of the languages accepted by such DFAs. We do not require the transition function δ to be complete; This choice is motivated by the fact that requiring completeness restricts the class of Wheeler DFAs [2]. Furthermore, we do not require DFAs to be connected; Also this choice is customary as it allows, for instance, to use our WDFA sampler to empirically study properties such as connectivity phase transition thresholds.

We say that the alphabet Σ is *effective* if and only if $(\forall j\in\Sigma)(\exists u,v\in Q)(\delta_j(u)=v)$, i.e. if every character of Σ labels at least one transition. We assume that the alphabet $\Sigma=[\sigma]$ is totally ordered according to the standard order among integers. Wheeler DFAs constitute a special class of DFAs that can be stored compactly and indexed efficiently due to an underlying order on the states: the *Wheeler order* (see Definition 3). As said in Definition 2, in this paper the states Q of an automaton D are represented by the integer set $[n]$ for some positive integer n; note that in the following definition we use the order on integers $<$ to denote the Wheeler order on the states.

▶ **Definition 3** (Wheeler DFA [11]). *A* Wheeler DFA *(WDFA) is a DFA D such that $<$ is a* Wheeler order*, i.e. for $a, a' \in \Sigma$, $u, v, u', v' \in Q$:*
(i) *If $u' = \delta_a(u)$, $v' = \delta_{a'}(v)$, and $a \prec a'$, then $u' < v'$.*
(ii) *If $u' = \delta_a(u) \neq \delta_a(v) = v'$ and $u < v$, then $u' < v'$.*

We note that the source axiom present in [11], which requires that the source state is first in the order, vanishes in our case as the ordering $<$ on the integers directly implies that the source state is ordered first. Notice that property (i) in Definition 3 implies that a WDFA is *input-consistent*, i.e., all in-going transitions to a given state have the same label.

▶ **Definition 4.** *With $\mathcal{D}_{n,m,\sigma}$ we denote the set of all Wheeler DFAs with effective alphabet $\Sigma = [\sigma]$, n states $Q = [n]$, m transitions, and Wheeler order $1 < 2 < \ldots < n$.*

Clearly, $\mathcal{D}_{n,m,\sigma}$ is a subset of the set $\mathcal{A}_{n,m,\sigma}$ of all finite (possibly non-deterministic) automata over the ordered alphabet $[\sigma]$ with n states $[n]$ and m transitions.

In this paper we investigate the following algorithmic problem:

▶ **Problem 5.** *For given n, m, and σ, generate an element from $\mathcal{D}_{n,m,\sigma}$ uniformly at random.*

Note that, since in Definition 4 we require $1 < 2 < \ldots < n$ to be the Wheeler order, Problem 5 is equivalent to that of uniformly generating pairs formed by a Wheeler DFA D and a valid Wheeler order for the states $Q = [n]$ of D, not necessarily equal to the integer order $1 < 2 < \cdots < n$. Throughout the whole paper, we assume that $n - 1 \le m \le n\sigma$ and $\sigma \le n - 1$ (due to input consistency), as otherwise $\mathcal{D}_{n,m,\sigma} = \emptyset$ and the problem is trivial.

3 An Algorithm for Uniformly Generating WDFAs

Our strategy towards solving Problem 5 efficiently is to associate every element D from $\mathcal{D}_{n,m,\sigma}$ to exactly one pair (O, I) of elements from $\mathcal{O}_{n,\sigma,m} \times \mathcal{I}_{m,n}$ (see Definition 6 below) via a function $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{O}_{n,\sigma,m} \times \mathcal{I}_{m,n}$ ("r" stands for *representation*). Formally, the two sets appearing in the co-domain of r are given in the following definition.

▶ **Definition 6.** *Let*

$$\mathcal{O}_{n,\sigma,m} := \left\{O \in \{0,1\}^{n\times\sigma} : \|O\| = m \text{ and } \|O_j\| \ge 1 \text{ for all } j \in [\sigma]\right\} \quad \text{and}$$
$$\mathcal{I}_{m,n} := \left\{I \in \{0,1\}^{m} : \|I\| = n - 1\right\}.$$

The intuition behind the two sets $\mathcal{O}_{n,\sigma,m}$ and $\mathcal{I}_{m,n}$ is straightforward: their elements encode the outgoing labels and the in-degrees of the automaton's states, respectively. In order to describe more precisely this intuition, let us fix an automaton $D = (Q, \delta, \Sigma) \in \mathcal{D}_{n,m,\sigma}$ and consider its image $r(D) = (O, I) \in \mathcal{O}_{n,\sigma,m} \times \mathcal{I}_{m,n}$ (see Figures 1 and 2 for an illustration):

- The matrix O is an encoding of the labels of the out-transitions of D. A 1-bit in position $O_{u,j}$ means that there is an out-going transition from state u labeled j. Formally,

$$O_{u,j} := \begin{cases} 1 & \text{if } \exists v : v = \delta_j(u) \\ 0 & \text{otherwise.} \end{cases} \qquad (1)$$

- The vector I is a concise encoding of the in-degrees of all states. It is defined as

$$I := (\underbrace{1, 0, \ldots, 0}_{|\delta^{in}(2)|}, \underbrace{1, 0, \ldots, 0}_{|\delta^{in}(3)|}, \ldots, \underbrace{1, 0, \ldots, 0}_{|\delta^{in}(n)|}), \tag{2}$$

  i.e, for all states i other than the source (that has no in-transitions), the vector contains exactly one 1-bit followed by $|\delta^{in}(i)| - 1$ 0-bits.

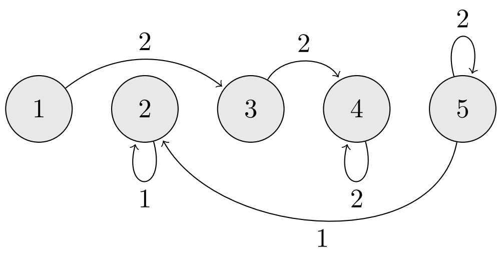

Figure 1 Running example: a WDFA D with $n = 5$ states, $m = 6$ edges, alphabet cardinality $\sigma = 2$, and Wheeler order $1 < 2 < 3 < 4 < 5$. Note that the WDFA has two connected components.

	1	2
1	0	1
2	1	0
3	0	1
4	0	1
5	1	1

2		3	4		5
1	0	**1**	1	0	1

Figure 2 Matrix O (left) and bit-vector I (right) forming the encoding $r(D) = (O, I)$ of the WDFA D of Figure 1. In matrix O, column names are characters from $\Sigma = [\sigma]$ and row names are states from $Q = [n]$. In bit-vector I, each state (except state 1) is associated with a bit set, in Wheeler order. Cells containing a set bit are named with the name of the corresponding state. Bits in bold highlight the states on which the character that labels the state's incoming transitions changes (i.e. state 2 is the first whose incoming transitions are labeled 1, and state 3 is the first whose incoming transitions are labeled 2).

Let us proceed with two remarks.

▶ Remark 7. As $\|O\| = m$ there are m transitions in total. As $\|O_j\| \geq 1$ for all $j \in [\sigma]$, the alphabet is effective, i.e., every character labels at least one transition.

▶ Remark 8. The vector I does not encode the letter on which a transition is in-going to a given state. Notice however that as D is a WDFA all these transitions have to be labeled with the same letter and we can reconstruct this letter for a given I once we know the total number of transitions labeled with each letter. This is because property (i) of Definition 3 guarantees that the node order is such that the source state (that has no in-going transitions) is ordered first followed by nodes whose in-transitions are labeled with character 1, followed by nodes with in-transitions labeled with character 2, etc. The information on how many transitions are labeled with each character is carried by the matrix O for which $r(D) = (O, I)$.

Let (O, I) be a pair from the image of r, i.e, $r(D) = (O, I)$ for some D. Then it will always be the case that I is contained in a subset $\mathcal{I}_O$ of $\mathcal{I}_{m,n}$ that can be defined as follows.

▶ **Definition 9.** *For a matrix $O \in \mathcal{O}_{n,\sigma,m}$, let*

$$\mathcal{I}_O := \left\{ I \in \mathcal{I}_{m,n} : I_{1+\sum_{k=1}^{j-1} \|O_k\|} = 1 \text{ for all } j \in [\sigma] \right\}.$$

Using our running example of Figure 2, the bits $I_{1+\sum_{k=1}^{j-1}\|O_k\|}$ that we force to be equal to 1 are those highlighted in bold, i.e. I_1 and I_3: noting that bits in I correspond to edges, those bits correspond to the leftmost edge labeled with a given character j (for any $j \in \Sigma$).

This leads us to define the following subset of $\mathcal{O}_{n,\sigma,m} \times \mathcal{I}_{m,n}$:

▶ **Definition 10.** $\mathcal{R}_{n,m,\sigma} := \{(O, I) : O \in \mathcal{O}_{n,\sigma,m}$ *and* $I \in \mathcal{I}_O\}$.

Based on the above definition, we can prove:

▶ **Lemma 11.** *For any* $D \in \mathcal{D}_{n,m,\sigma}$, $r(D) \in \mathcal{R}_{n,m,\sigma}$.

Proof. Note that the integers $\sum_{k=1}^{j-1}\|O_k\|$ for $j \in [\sigma]$ correspond to the number of edges labeled with letters $1, \dots, j-1$, hence the positions $1 + \sum_{k=1}^{j-1}\|O_k\|$ correspond to a change of letter in the sorted (by destination node) list of edges. Recalling that WDFAs are input-consistent (i.e., all in-transitions of a given node carry the same label) and that nodes are ordered by their in-transition letters, positions $1+\sum_{k=1}^{j-1}\|O_k\|$ for $j \in [\sigma]$ in I must necessarily correspond to the first edge of a node, hence they must contain a set bit. ◀

The co-domain of the function r can thus be restricted, and the function's signature can be redefined, as follows: $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{R}_{n,m,\sigma}$.

After describing this association of a WDFA $D \in \mathcal{D}_{n,m,\sigma}$ to a (unique) pair $r(D) = (O, I) \in \mathcal{R}_{n,m,\sigma}$, we will argue that function r is indeed a bijection from $\mathcal{D}_{n,m,\sigma}$ to $\mathcal{R}_{n,m,\sigma}$. It will follow that one way of generating elements from $\mathcal{D}_{n,m,\sigma}$ is to generate elements from $\mathcal{R}_{n,m,\sigma}$: this will lead us to an efficient algorithm to uniformly sample WDFAs from $\mathcal{D}_{n,m,\sigma}$, as well to a formula for the cardinality of $\mathcal{D}_{n,m,\sigma}$.

3.1 The Basic WDFA Sampler

Our overall approach is to (1) uniformly sample a matrix O from $\mathcal{O}_{n,\sigma,m}$ using Algorithm 2, then (2) uniformly sample a vector I from $\mathcal{I}_O$ using Algorithm 3 with input O, and finally (3) build a WDFA D using O and I as input via Algorithm 4. We summarize this procedure in Algorithm 1. A crucial point in our correctness analysis (Section 4) will be to show that uniformly sampling from $\mathcal{O}_{n,\sigma,m}$ and $\mathcal{I}_O$ does indeed lead to a uniform WDFA from $\mathcal{D}_{n,m,\sigma}$ (besides the bijectivity of r, intuitively, this is because $|\mathcal{I}_O| = |\mathcal{I}_{O'}|$ for any $O, O' \in \mathcal{O}_{n,\sigma,m}$).

As source of randomness, our algorithm uses a black-box *shuffler* algorithm: given a bit-vector $B \in \{0,1\}^*$, function `shuffle`(B) returns a random permutation of B. To improve readability, in this subsection we start by describing a preliminary simplified version of our algorithm which does not assume any particular representation for the matrix-bit-vector pair $(O, I) \in \mathcal{R}_{n,m,\sigma}$, nor a particular shuffling algorithm (for now, we only require the shuffling algorithm to permute uniformly its input). By employing a particular *sequential shuffler*, in Subsection 3.2 we then show that we can generate a sparse representation of O and I on-the-fly, thereby achieving *constant* working space and linear expected running time.

Algorithm 1 `sample_D`(n, m, σ).

```
1 $O :=$ sample_O$(n, m, \sigma)$
2 $I :=$ sample_I$(O)$
3 $D :=$ build_D$(O, I)$
4 **return** $D$
```

Out-transition Matrix. In order to sample the matrix O from $\mathcal{O}_{n,\sigma,m}$, in addition to function `shuffle` we assume a function $\texttt{reshape}_{k,\ell}$ that takes a vector x of dimension $k \cdot \ell$ and outputs a matrix A of dimension $k \times \ell$ with the j'th column A_j being the portion $x_{(j-1)\cdot k+1}, \ldots, x_{j \cdot k}$ of x. The algorithm to uniformly generate O from $\mathcal{O}_{n,\sigma,m}$ then simply samples a bit vector of length $n\sigma$ with exactly m 1-bits, shuffles it uniformly, reshapes it to be a matrix of dimension $n \times \sigma$ and repeats these steps until a matrix is found with at least one 1-bit in each column (rejection sampling).

Algorithm 2 `sample_O`(n, m, σ).

```
1 repeat
2 |   $O := \texttt{reshape}_{n,\sigma}(\texttt{shuffle}(1^m 0^{n\sigma-m}))$
3 until $\|O_j\| \geq 1$ for all $j \in [\sigma]$
4 return $O$
```

Looking at the running example of Figures 1 and 2, the shuffler is called as $\texttt{shuffle}(1^6 0^4)$. In this particular example, this bit-sequence is permuted as 0100110111 by function `shuffle`. Function $\texttt{reshape}_{n,\sigma}$ converts this bit-sequence into the matrix O depicted in Figure 2, left.

In-transition Vector. In order to generate the vector I from $\mathcal{I}_{m,n}$, we proceed as follows. The algorithm takes O as input and generates a uniform random element from the set $\mathcal{I}_O$ by first creating a "mask" that is a vector of the correct length m and contains σ 1-bits at the points $1 + \sum_{k=1}^{j-1} \|O_k\|$ for $j \in [\sigma]$. These are the points in I where the character of the corresponding transition changes and hence, by the input-consistency condition, also the state has to change. The remaining $m - \sigma$ positions in the mask are filled with the wildcard character $\#$. We then give this mask vector as the first argument to a function `fill` that replaces the $m - \sigma$ positions that contain the wildcard character $\#$ with the characters in the second argument (in order). Formally, the function `fill` takes two vectors as arguments a and b with the condition that a contains $|b|$ times the $\#$ character and $|a| - |b|$ times a 1-bit. The function then returns a vector c that satisfies $c_i = 1$ whenever $a_i = 1$ and $c_i = b_{i-\mathrm{rank}(a,i)}$ otherwise, i.e., when $a_i = \#$.

Algorithm 3 `sample_I`(O).

```
1 extract $n, m, \sigma$ from $O$
2 mask := $1\#^{\|O_1\|-1} 1\#^{\|O_2\|-1} \ldots 1\#^{\|O_\sigma\|-1}$
3 $I := \texttt{fill}(\text{mask}, \texttt{shuffle}(1^{n-\sigma-1} 0^{m-n+1}))$
4 return $I$
```

Going back to our running example of Figures 1 and 2, we have mask = $\mathbf{1}\#\mathbf{1}\#\#\#$ (that is, all bits but the bold ones in the right part of Figure 2 are masked with a wildcard). The shuffler is called as $\texttt{shuffle}(1100)$ and, in this particular example, returns the shuffled bit-vector 0101. Finally, function `fill` is called as $\texttt{fill}(\mathbf{1}\#\mathbf{1}\#\#\#, 0101)$ and returns the bit-vector I =101101 depicted in the right part of Figure 2.

Building the WDFA. After sampling O and I, the remaining step is to build the output DFA D. This is formalized in Algorithm 4. By iterating over all non-zero elements in O, we construct the transition function δ: the i'th non-zero entry in O corresponds to an in-transition at state rank$(I, i) + 1$ (we keep a counter named v corresponding to this rank). The origin state of the transition is the row in which we find the i'th 1 in O when reading O column-wise. The column itself corresponds to the label of this transition.

Algorithm 4 `build_D`(O, I).

```
1 extract $n, m, \sigma$ from $O$, $Q := [n]$, $\Sigma := [\sigma]$
2 $\delta := \emptyset$, $i := 1$, $v := 1$
3 for $j = 1, 2, \ldots, \sigma$ do
4     for $u = 1, 2, \ldots, n$ do
5         if $O_{u,j} = 1$ then
6             if $I_i = 1$ then $v := v + 1$
7             $\delta := \delta \cup \{((u, j), v))\}$, $i := i + 1$
8 return $D = (Q, \Sigma, \delta)$
```

3.2 Constant-Space WDFA Sampler

Notice that our Algorithm 4 accesses the matrix O and the bit-vector I in a sequential fashion: O is accessed column-wise and I from its first to last position. Based on this observation, we now show how our WDFA sampler can be modified to use *constant* working space. The high-level idea is to generate on-the-fly the positions of non-zero entries of O and I in increasing order.

In order to achieve this, we employ the *sequential shuffler* described by Shekelyan and Cormode [17]. Given two integers N and n, the function `init_sequential_shuffler`(N, n) returns an iterator S that can be used (with a stack-like interface) to extract n uniform integers without replacement from $[N]$, in *ascending order* and using a *constant* number of words of working space (that is, the random integers are generated on-the-fly upon request, from the smallest to the largest). More specifically, function S.`pop`() returns the next sampled integer, while S.`empty`() returns true if and only if all n integers have been extracted. The sequential shuffling algorithm is essentially a clever modification of Knuth's shuffle [14] (also referred to as Fisher-Yates shuffler). Knuth's shuffler, after going through the arbitrarily ordered set $[N]$, and in the i'th iteration (for i from 1 to n) swapping the i'th item with the item at a random position $[i, N]$, returns the first n items in the resulting permutation. Knuth's shuffler requires working space proportional to n as we need to remember which elements have been swapped from lower positions (i.e., index $\leq n$) into higher positions (i.e., index $> n$). The idea behind the sequential shuffler of Shekelyan and Cormode is to first sample just the cardinality H of the set of items in higher positions that Knuth's shuffler would swap into lower position. Then, in a second step the algorithm samples H actual items from higher positions with replacement, resulting in $h \leq H$ elements. Finally, in a third step, $n - h$ items are sampled from lower positions. We note that the distribution in the first step is chosen such that the sampling in the second step can be done with replacement – sampling duplicates simply increases the number of items sampled from lower positions. We refer the reader to the article by Shekelyan and Cormode [17] for further details.

Algorithm Description. We now describe Algorithm 5. We recall the mask employed in Algorithm 3: Algorithm 5 iterates, using variable i, over the ranks (i.e., i-th occurrences) of characters # (wildcards) in the mask. Variable i', on the other hand, stores the rank of the next wildcard # that is replaced with a set bit by the shuffler; the values of i' are extracted

Algorithm 5 `sample_D_constant_space`(n, m, σ).

```
1  $i := 1$                        /* Current position in the subsequence of #'s of the mask */
2  $v := 1$                        /* Destination state of current transition */
3  $S_O :=$ init_sequential_shuffler$(n\sigma, m)$
4  $S_I :=$ init_sequential_shuffler$(m - \sigma, n - \sigma - 1)$
5  $i' := S_I$.pop()               /* next nonzero position in sequence of #'s in the mask */
6  $j := 0$                        /* current column in $O$ */
7  $prev\_j := 0$                  /* previously-visited column in $O$ */
8  while not $S_O$.empty() do
9      $t := S_O$.pop()
10     $(u, j) := \Big(\big((t-1) \mod n\big) + 1, \big((t-1) \text{ div } n\big) + 1\Big)$   /* Nonzero coordinate in $O$ */
11     if $j > prev\_j + 1$ then
12         clear output stream and goto line 1   /* Rejection: $\|O_{prev\_j+1}\| = 0$ */
13     if $j = prev\_j + 1$              /* Column of $O$ changes */
14     then
15         $v := v + 1$
16         $prev\_j := j$
17     else
18         if $i = i'$ then
19             $v := v + 1$
20             $i' := S_I$.pop()        /* next nonzero position in sequence of #'s in the mask */
21         $i := i + 1$
22     output $((u, j), v)$             /* Stream transition to output */
23 if $j \neq \sigma$ then
24     clear output stream and goto line 1   /* Rejection: $\|O_\sigma\| = 0$ */
```

from the shuffler S_I. Now, whenever $i = i'$, we are looking at a bit set in bit-vector I (which here is not stored explicitly, unlike in Algorithm 4) and thus we have to move to the next destination state v. This procedure exactly simulates Lines 6 and 7 of Algorithm 4.

The iteration (column-wise) over all non-zero entries of matrix O is simulated by the extraction of values from the shuffler I_O (one value per iteration of the while loop at Line 8): each such value t extracted at Line 9 is converted to a pair (u, j) at Line 10. Variables j and $prev\_j$ store the columns of the current and previously-extracted non-zero entries of O, respectively. If $j > prev\_j + 1$, then column number $prev\_j + 1$ has been skipped by the shuffler, i.e., O_{prev_j+1} does not contain non-zero entries. In this case, we reject and start the sampler from scratch (Line 12; note that we need to clear the output stream before re-initializing the algorithm). If, on the other hand, $j = prev\_j+1$ (Line 13), then the current non-zero entry of O belongs to the next column with respect to the previously-extracted non-zero entry; this means that the character labeling incoming transitions changes and we therefore move to the next destination node by increasing $v := v + 1$ (Line 15). In this case we do not increment i, since the new destination node v is the first having incoming label j and thus it does not correspond to a character # in the mask. Variable i gets incremented only if $j = prev\_j$: this happens at Line 21. The other case in which we need to move to the next destination node ($v := v + 1$) is when $j = prev\_j$ and $i = i'$ (Line 19). In such a case, in addition to incrementing v we also need to extract from the shuffler S_I the rank i' of the next mask character # that is replaced with a set bit (Line 20). After all these operations, we write the current transition $((u, j), v)$ to the output stream (Line 22). The last two lines of Algorithm 5 check if the last visited column of matrix O is indeed O_σ. If not, $\|O_\sigma\| = 0$ and we need to reject and re-start the algorithm.

The remaining components of Algorithm 5 are devoted to simulate Algorithm 1, using as input the two sequences of random pairs/integers extracted from S_O and S_I, respectively. As a matter of fact, the two loops in Algorithm 4 correspond precisely to extracting the pairs (u, j) from S_O, and the check at Line 6 of Algorithm 4, together with the increment of i at Line 7, corresponds to extracting the integers i' from S_I. The rejection sampling mechanism (repeat-until loop in Algorithm 2) is simulated in Algorithm 5 by re-starting the algorithm whenever the column j of the current pair (u, j) is either larger by more than one unit than the column j_prev of the previously-extracted pair (i.e., $\|O_{j\_prev+1}\| = 0$, Line 12), or if the last pair extracted from S_O is such that j is not the σ-th column (i.e., $\|O_\sigma\| = 0$, Line 24).

Running Example. To understand how the sequential shuffler is used in Algorithm 5, refer again to the running example of Figures 1 and 2. In Algorithm 5 at Line 3, the sequential shuffler S_O is initialized as $S_O :=$ `init_sequential_shuffler`$(n\sigma = 10, m = 6)$, i.e. the iterator S_O returns 6 uniform integers without replacement from the set $\{1, 2, \dots, 10\}$. In this particular example, function `pop()` called on iterator S_O returns the following integers, in this order: 2,5,6,8,9,10. Using the formula at Line 10 of Algorithm 5, these integers are converted to the matrix coordinates $(2, 1), (5, 1), (1, 2), (3, 2), (4, 2), (5, 2)$, i.e., precisely the nonzero coordinates of matrix O in Figure 2, sorted first by column and then by row.

Using the same running example, the sequential shuffler S_I is initialized in Line 4 of Algorithm 5 as $S_I :=$ `init_sequential_shuffler`$(m - \sigma = 4, n - \sigma - 1 = 2)$, i.e. the iterator S_I returns two uniform integers without replacement from the set $\{1, 2, 3, 4\}$. In this particular example, function `pop()` called on iterator S_I returns the following integers, in this order: 2,4. Using the notation of the previous subsection, this sequence has the following interpretation: the 2-nd and 4-th occurrences of # of our mask 1#1### used in Algorithm 3 have to be replaced with a bit 1, while the others with a bit 0. After this replacement, the mask becomes 101101, i.e. precisely bit-vector I of Figure 2.

4 Analysis

4.1 Correctness, Completeness and Uniformity

Being Algorithm 5 functionally equivalent to Algorithm 1 (the only relevant difference between the two being the employed data structures to represent matrix O and bit-vector I), for ease of explanation in this section we focus on analyzing the correctness (the algorithm generates only elements from $\mathcal{D}_{n,m,\sigma}$), completeness (any element from $\mathcal{D}_{n,m,\sigma}$ can be generated by the algorithm) and uniformity (all $D \in \mathcal{D}_{n,m,\sigma}$ have the same probability to be generated by the algorithm) of Algorithm 1. These properties then automatically hold on Algorithm 5 as well.

We start with a simple lemma. The lemma says the following: Assume that $r(D) = (O, I)$ and O contains a 1 in position (u, j), meaning that there is a transition leaving state u, labeled with letter j. Then this out-transition is the $i = \text{rank}(O, (u, j))$ bit that is set to 1 in O and hence the entry in I corresponding to this transition can be found at $I[i]$. The state to which this transition is in-going is exactly the number of 1s in I up to this point, i.e., $\text{rank}(I, i)$, plus one (the offset is due to the source having no in-transitions).

▶ **Lemma 12.** *Let $D \in \mathcal{D}_{n,m,\sigma}$ and let $r(D) = (O, I)$. If $O_{u,j} = 1$ and* $\operatorname{rank}(O, (u, j)) = i$*, then* $\delta(u, j) = \operatorname{rank}(I, i) + 1$.

Proof. First, notice that since $O_{u,j} = 1$, it is clear that there is an outgoing transition from u labeled j. Furthermore, since $\operatorname{rank}(O, (u, j)) = i$, we know that this transition corresponds to the i-th entry in I. Now, by the definition of I, it follows that the destination state of the considered transition is $v = \operatorname{rank}(I, i) + 1$. ◀

Algorithm 4 is a deterministic algorithm and thus describes a function, say f, from the set of its possible inputs to the set of its possible outputs. The set of its possible inputs, i.e., the domain of f, is exactly $\mathcal{R}_{n,m,\sigma}$. The algorithm's output is certainly a finite automaton, i.e., the co-domain of f is $\mathcal{A}_{n,m,\sigma}$. We will in fact show that the range of f is exactly $\mathcal{D}_{n,m,\sigma}$. We will do so by showing that f is actually an inverse of r, more precisely we show that (1) r is surjective and (2) f is a left-inverse of r (and thus r is injective).

Surjectivity of r. We start with proving that r is surjective.

▶ **Lemma 13.** *It holds that $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{R}_{n,m,\sigma}$ is surjective.*

Proof. Fix an element $(O, I) \in \mathcal{R}_{n,m,\sigma}$, i.e., an $O \in \mathcal{O}_{n,\sigma,m}$ and $I \in \mathcal{I}_O$. We now construct an automaton $D = (Q, \Sigma, \delta)$ and then show that $r(D) = (O, I)$. We let $Q = [n]$, $\Sigma = [\sigma]$, and

$$\delta = \{((u, j), v) : O_{u,j} = 1 \text{ and } v = \operatorname{rank}(I, \operatorname{rank}(O, (u, j))) + 1\}.$$

Let $r(D) = (O', I')$ and let us proceed by showing that $O = O'$ and $I = I'$. Recall the definition of r, see Equations (1) and (2). It is immediate that $O' = O$ given the definition of O' and δ. In order to show that $I' = I$, first note that $I' = \prod_{i=2}^{n} 10^{|\delta^{in}(i)|-1}$. Then, consider the following relation between I and $\delta^{in}(i)$ for any state $i \in [n]$, which uses the definition of δ and Lemma 12:

$$\begin{aligned}
|\delta^{in}(i)| &= |\{(u, j) \in [n] \times [\sigma] : O_{u,j} = 1 \text{ and } \operatorname{rank}(I, \operatorname{rank}(O, (u, j))) + 1 = i\}| \\
&= |\{k \in [m] : k = \operatorname{rank}(O, (u, j)) \text{ for some } (u, j) \in [n] \times [\sigma] \text{ with } O_{u,j} = 1 \\
&\qquad\qquad \text{and } \operatorname{rank}(I, k) + 1 = i\}| \\
&= \max\{k \in [m] : \operatorname{rank}(I, k) = i - 1\} - \max\{k \in [m] : \operatorname{rank}(I, k) = i - 2\}.
\end{aligned}$$

Now, recall that $I \in \mathcal{I}_O$, hence the first bit in I is 1. Using the previous equality, it now follows that the second 1-bit in I is at position $|\delta^{in}(2)| + 1$. By using this equality another $n - 3$ times, we obtain that the first $\sum_{i=2}^{n-1} |\delta^{in}(i)| + 1$ positions of I are equal to $(10^{|\delta^{in}(2)|-1}10^{|\delta^{in}(3)|-1} \ldots 10^{|\delta^{in}(n-1)|-1})1$ and thus agree with I'. It remains to observe that this portion of I already contains $n - 1$ bits that are equal to 1 and thus the remaining bits have to be zero-bits as $I \in \mathcal{I}$. Hence, $I = I'$ and this completes the proof. ◀

Injectivity of r via Left-inverse f. In order to establish that f is the inverse of r, it remains to prove that f is a left-inverse of r (which implies that r is injective).

▶ **Lemma 14.** *The function f is a left-inverse of r, i.e., $f(r(D)) = D$ for any $D \in \mathcal{D}_{n,m,\sigma}$.*

Proof. Let $D = (Q, \Sigma, \delta) \in \mathcal{D}_{n,m,\sigma}$ and let $(O, I) = r(D)$. We have to show that $D' = f(O, I)$, i.e., the automaton $D' = (Q', \Sigma', \delta')$ output by Algorithm 4 on input (n, m, σ, O, I) is equal to D. Notice that clearly $Q = Q' = [n]$ and $\Sigma = [\sigma]$. It remains to show that $\delta = \delta'$. It is clear that Algorithm 4 adds m transitions to δ', one in each of the $m = \|O\|$ iterations. It thus remains to prove that each such transition $((u,j),v)$ added in some iteration i is contained in δ. Firstly, as $O_{u,j} = 1$ it is clear that D has an outgoing transition at state u with letter j, second it is clear that the algorithm maintains the property that $v = \text{rank}(I,i)+1$ and thus due to Lemma 12 it holds that $\delta(u,j) = v$ and thus this transition is also contained in δ. ◀

We can thus denote the function f with r^{-1}.

▶ **Corollary 15.** *Function $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{R}_{n,m,\sigma}$ is bijective.*

The above lemma has several consequences. First, it shows that the output of Algorithm 4 is always a WDFA. Second, as the function r is bijective, this means that generating uniform pairs from the range of r results in a uniform distribution of WDFAs from $\mathcal{D}_{n,m,\sigma}$.

▶ **Lemma 16.** *Algorithm 1 on input n, m, σ generates uniformly distributed WDFAs from $\mathcal{D}_{n,m,\sigma}$.*

Proof. In the light of r being a bijection and Algorithm 4 implementing the function r^{-1}, it remains to argue that the statements $O :=$ `sample_O`(n,m,σ) and $I :=$ `sample_I`(O) from Algorithm 1 in fact generate uniformly distributed pairs from the domain of r^{-1}, i.e., from $\mathcal{R}_{n,m,\sigma}$. It is clear that `sample_O`(n,m,σ) results in a uniformly distributed element O from $\mathcal{O}_{n,\sigma,m}$ and that `sample_I`(O) results in a uniformly distributed element I from $\mathcal{I}_O$. It thus remains to observe that $|\mathcal{I}_O|$ is identical for all $O \in \mathcal{O}_{n,\sigma,m}$, namely $|\mathcal{I}_O| = \binom{m-\sigma}{n-\sigma-1}$ for all $O \in \mathcal{O}_{n,\sigma,m}$. This completes the proof. ◀

4.2 Run-time and Space

We now analyze the number of iterations of Algorithm 2, that is, the expected number of rejections before extracting a bit-matrix O with $\|O_j\| > 0$ for all $j \in [\sigma]$. Algorithm 5 is clearly equivalent to Algorithm 1 also under this aspect, since at Lines 12 and 24 we re-start the algorithm whenever we generate a column O_j without non-zero entries. We prove:

▶ **Lemma 17.** *Assume that $m \geq \sigma \ln(e \cdot \sigma)$. The expected number of iterations of Algorithm 2 (equivalently, rejections of Algorithm 5) is at most 1.6. Furthermore, the algorithm terminates after $O(\log m)$ iterations with probability at least $1 - m^{-c}$ for any constant $c > 0$.*

We refer the reader to the full version of this article [4] for the proof of the above lemma. Now assume that $\sigma \leq m/\ln m$. This implies that $e \cdot \sigma \leq m$ (for m larger than a constant), which together with the initial assumption implies that $\sigma \ln(e \cdot \sigma) \leq \sigma \ln m \leq m$. This is exactly the condition in Lemma 17. Hence, if $\sigma \leq m/\ln m$ then the expected number of rejections of Algorithm 5 is $O(1)$ (or $O(\log m)$ with high probability). Our main Theorem 1 follows from the fact that the sequential shuffler of [17] uses constant space, its functions `pop`() and `empty`() run in constant time, and the while loop at Line 8 of Algorithm 5 runs for at most m iterations (less only in case of rejection) every time Algorithm 5 is executed.

5 Counting Wheeler DFAs

In this section, we use the WDFA characterization of Section 3.1 to give an exact formula for the number $|\mathcal{D}_{n,m,\sigma}|$ of WDFAs with n nodes and m edges on effective alphabet $[\sigma]$ with Wheeler order $1 < 2 < \cdots < n$. All proofs of this section can be found in the full version of the article [4]. From our previous results, all we need to do is to compute the cardinalities of $\mathcal{O}_{n,m,\sigma}$ and $\mathcal{I}_O$.

▶ **Lemma 18.** $|\mathcal{O}_{n,m,\sigma}| = \sum_{j=0}^{\sigma}(-1)^j\binom{\sigma}{j}\binom{n(\sigma-j)}{m}$.

This lemma is obtained via an inclusion-exclusion argument. From Algorithm 3, it is immediate that $|\mathcal{I}_O| = \binom{m-\sigma}{n-\sigma-1}$ for all $O \in \mathcal{O}_{n,m,\sigma}$ (see also the proof of Lemma 16). Since $r : \mathcal{D}_{n,m,\sigma} \to \mathcal{R}_{n,m,\sigma}$ is bijective (Corollary 15), we obtain an exact formula for $|\mathcal{D}_{n,m,\sigma}|$:

▶ **Theorem 19.** *The number $|\mathcal{D}_{n,m,\sigma}|$ of WDFAs with set of nodes $[n]$ and m transitions labeled from the effective alphabet $[\sigma]$, for which $1 < 2 < \cdots < n$ is a Wheeler order is*

$$|\mathcal{D}_{n,m,\sigma}| = \binom{m-\sigma}{n-\sigma-1}\sum_{j=0}^{\sigma}(-1)^j\binom{\sigma}{j}\binom{n(\sigma-j)}{m}.$$

Using similar techniques, in the case where σ is not arbitrarily close to n, i.e., $\sigma \leq (1-\varepsilon)\cdot n$ for some constant ε, we moreover obtain a tight formula for the logarithm of the cardinality of $\mathcal{D}_{n,\sigma} = \bigcup_m \mathcal{D}_{n,m,\sigma}$, the set of all Wheeler DFAs with n states over effective alphabet $[\sigma]$ and Wheeler order $1 < 2 < \cdots < n$:

▶ **Theorem 20.** *The following bounds hold:*
1. $\log|\mathcal{D}_{n,\sigma}| \geq n\sigma + (n-\sigma)\log\sigma - (n + \log\sigma)$*, for any n and $\sigma \leq n-1$, and*
2. $\log|\mathcal{D}_{n,\sigma}| \leq n\sigma + (n-\sigma)\log\sigma + O(n)$*, for any $n \geq 2/\varepsilon$ and $\sigma \leq (1-\varepsilon)\cdot n$, where ε is any desired constant such that $\varepsilon \in (0, 1/2]$.*

Note that $\log|\mathcal{D}_{n,\sigma}|$ is the information-theoretic worst-case number of bits necessary (and sufficient) to encode a WDFA from $\mathcal{D}_{n,\sigma}$. Our Theorem 20 states that, up to an additive $\Theta(n)$ number of bits, this value is of $n\sigma + (n-\sigma)\log\sigma$ bits. As a matter of fact, our encoding $r(D) = (O, I)$ of Section 3, opportunely represented using succinct bitvectors [16], achieves this bound up to additive lower-order terms and supports efficient navigation of the transition relation.

6 Implementation

We implemented our uniform WDFA sampler in `C++`.[1] We tested our implementation by generating WDFAs with a broad range of parameters: $n \in \{10^6 \cdot 2^i : i = 0, \ldots, 6\}$, $m \in \{n \cdot 2^i - 1 : i = 0, \ldots, 7\}$ and $\sigma = 128$. To analyze the impact of streaming to disk on the running time, we tested two versions of our code: (1) We stream the resulting WDFA to disk (SSD). (2) We stream the WDFA to a pre-allocated vector residing in internal memory. Note that constant working space is achieved only in case (1). Our experiments were run on a server with Intel(R) Xeon(R) W-2245 CPU @ 3.90GHz with 8 cores, 128 gigabytes of RAM, 512 gigabytes of SSD, running Ubuntu 18.04 LTS 64-bit. Working space was measured with `/usr/bin/time` (Resident set size).

Figure 3 shows the running time of both variants (left: (1) streaming to SSD; right: (2) streaming to RAM). Both versions exhibit a linear running time behavior, albeit with a different multiplicative constant. The algorithm storing the WDFA in internal memory is between 1.2 and 1.7 times faster than the version streaming the WDFA to the disk (the relatively small difference is due to the fact that we used an SSD). We measured a throughput of at least 5.466.897 and 7.525.794 edges per second for the two variants, respectively. In our experiments we never observed a rejection: this is due to the fact that $\sigma \ll m$, making it extremely likely to generate bit-matrices O containing at least one set bit in each column.

[1] Implementation available at `https://github.com/regindex/Wheeler-DFA-generation`.

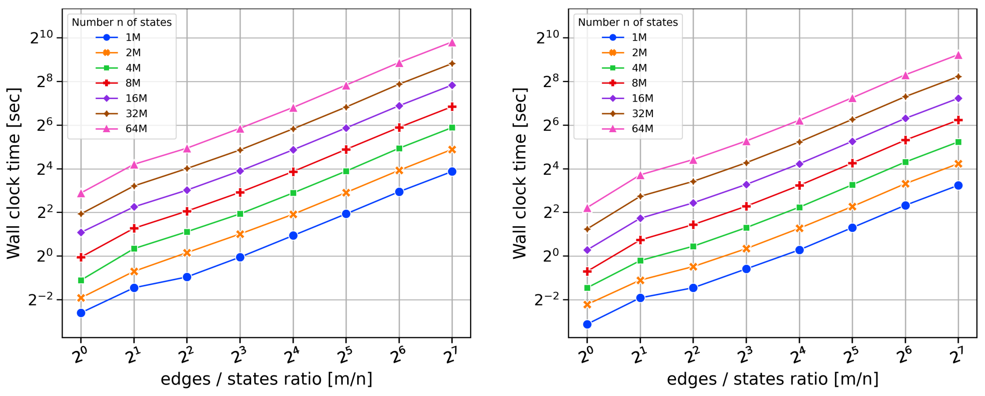

Figure 3 Wall clock time for generating random WDFAs using Algorithm 5. Left: running time for the algorithm in case (1), i.e., streaming the resulting WDFAs to disk. Right: running time in case (2), i.e., storing WDFAs in internal memory.

As far as space usage is concerned, version (1), i.e., streaming the WDFA to disk, always used about 4 MB of internal memory, independently from the input size (this memory is always required to load the `C++` libraries). This confirms the constant space usage of our algorithm, also experimentally. As expected, the space usage of version (2) is linear with the input's size. Nevertheless, both algorithms are extremely fast in practice: in these experiments, the largest automaton consisting of 64 million states and more than 8 billion edges was generated in about 15 and 10 minutes with the first and second variant, respectively.

References

1 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Regular Languages Meet Prefix Sorting. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '20, pages 911–930, USA, 2020. Society for Industrial and Applied Mathematics.

2 Jarno Alanko, Giovanna D'Agostino, Alberto Policriti, and Nicola Prezza. Wheeler languages. *Information and Computation*, 281:104820, 2021. URL: `https://www.sciencedirect.com/science/article/pii/S0890540121001504`.

3 Jarno Alanko, Travis Gagie, Gonzalo Navarro, and Louisa Seelbach Benkner. Tunneling on wheeler graphs. In *2019 Data Compression Conference (DCC)*, pages 122–131. IEEE, 2019.

4 Ruben Becker, Davide Cenzato, Sung-Hwan Kim, Bojana Kodric, Riccardo Maso, and Nicola Prezza. Random wheeler automata. *CoRR*, abs/2307.07267, 2023. `doi:10.48550/arXiv.2307.07267`.

5 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

6 Kuan-Hao Chao, Pei-Wei Chen, Sanjit A Seshia, and Ben Langmead. WGT: Tools and algorithms for recognizing, visualizing and generating Wheeler graphs. *bioRxiv*, pages 2022–10, 2022.

7 Alessio Conte, Nicola Cotumaccio, Travis Gagie, Giovanni Manzini, Nicola Prezza, and Marinella Sciortino. Computing matching statistics on wheeler dfas. In *2023 Data Compression Conference (DCC)*, pages 150–159, 2023. `doi:10.1109/DCC55655.2023.00023`.

8 Giovanna D'Agostino, Davide Martincigh, and Alberto Policriti. Ordering regular languages and automata: Complexity. *Theoretical Computer Science*, 949:113709, 2023. URL: `https://www.sciencedirect.com/science/article/pii/S0304397523000221`.

9 Lavinia Egidi, Felipe A Louza, and Giovanni Manzini. Space efficient merging of de Bruijn graphs and Wheeler graphs. *Algorithmica*, 84(3):639–669, 2022.

10 Travis Gagie. On Representing the Degree Sequences of Sublogarithmic-Degree Wheeler Graphs. In *String Processing and Information Retrieval: 29th International Symposium, SPIRE 2022, Concepción, Chile, November 8–10, 2022, Proceedings*, pages 250–256. Springer, 2022.

11 Travis Gagie, Giovanni Manzini, and Jouni Sirén. Wheeler graphs: A framework for BWT-based data structures. *Theoretical Computer Science*, 698:67–78, 2017. `doi:10.1016/j.tcs.2017.06.016`.

12 Daniel Gibney and Sharma V Thankachan. On the complexity of recognizing wheeler graphs. *Algorithmica*, 84(3):784–814, 2022.

13 Adrián Goga and Andrej Baláž. Prefix-Free Parsing for Building Large Tunnelled Wheeler Graphs. In *22nd International Workshop on Algorithms in Bioinformatics*, 2022.

14 Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998. URL: `https://www.worldcat.org/oclc/312898417`.

15 Cyril Nicaud. Random Deterministic Automata. In *Proceedings of the 39th International Symposium on Mathematical Foundation of Computer Science (MFCS)*, pages 5–23, 2014.

16 Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, pages 233–242, USA, 2002. Society for Industrial and Applied Mathematics.

17 Michael Shekelyan and Graham Cormode. Sequential random sampling revisited: Hidden shuffle method. In Arindam Banerjee and Kenji Fukumizu, editors, *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, volume 130 of *Proceedings of Machine Learning Research*, pages 3628–3636. PMLR, April 2021. URL: `https://proceedings.mlr.press/v130/shekelyan21a.html`.