Interpreting the Subset Construction Using Finite Sublanguages

Mwawi Msiska* and Lynette van Zijl

Dept of Mathematical Sciences, Computer Science Division,
Stellenbosch University, Private Bag X1, 7602 Matieland, South Africa
mfmsiska@gmail.com
lvzijl@cs.sun.ac.za
http://www.cs.sun.ac.za

Abstract. We present a language-based approach to the well-known problem of the conversion between finite automaton (FA) types. We base our approach on the existence, for any FA, of a finite subset of the language of the FA, which we call the finite exhaustive language (FEL). An FA uses all its *reachable* transitions after computing all strings in its FEL. We convert the FA by *summarizing* its computations on strings from the FEL of its equivalent FA. We illustrate our approach using the well known nondeterministic finite automaton (NFA) to deterministic finite automaton (DFA) conversion. We describe a method to calculate the FEL of the DFA through graph traversals of the NFA, without first converting the NFA into a DFA. Using the FEL, we construct a DFA that has neither dead nor unreachable states. For an n-state NFA, we show that $O(e^{\sqrt{n \log n}})$ is an upper bound on the length of strings in the FEL of its equivalent DFA.

1 Introduction

For most finite automata, there are algorithms to simulate one type of automaton by another. For example, a deterministic finite automaton (DFA) can simulate a nondeterministic finite automaton (NFA) using the well-known subset construction [8]. In general, consider a simulation of finite automaton (FA) class A by FA class B. Most simulation algorithms are based on relating each state of an equivalent class B FA to some collection of states of the given class A FA. Transitions of the class B FA are constructed by relating properties of the state collections to the transition function of the given class A FA. Typically, arguments on the structure of these state collections in the simulations are fairly easy; however, deriving simulated transition functions describing the associations amongst these state collections can be quite complex. For example, see the derivation of transitions among *crossing sequences* in [4].

In contrast, we demonstrate an algorithm that is language-based, rather than state-based, for such simulations. Consider a simulation of a class A FA, M, by a class B FA, M', accepting a language $\mathcal{L}(M)$. Our approach constructs a finite sublanguage $\mathcal{L}'(M)$ of $\mathcal{L}(M)$ (called the finite exhaustive language, or FEL), by conducting restricted walks through the digraph (that is, the state diagram) of M. Our claim is that the equivalent class B FA, M', will then exhaust (visit) all its states and transitions to recognize $\mathcal{L}'(M)$. In our approach, we only need to know the structure of the collection of states of M relative to a single state of M'. The transitions are

* The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Mwawi Msiska, Lynette van Zijl: Interpreting the Subset Construction Using Finite Sublanguages, pp. 48–62.
Proceedings of PSC 2016, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05996-8

implied from actual runs of M on members of $\mathcal{L}'(M)$. We illustrate the approach for the simplest case, namely, the simulation of NFAs by DFAs.

In the literature, the construction of finite automata based on a finite language has previously been considered [2,3,9]. However, the context there is typically applications such as dictionary building, and not simulations between automata types per se. The interested reader may also note the similarities between the construction of the FEL based on walks through the digraph, and other rule-based descriptions of infinite languages such as regular expressions [4] and language equations [1]. The FEL corresponds to all strings that can be obtained by such formalisms, but with a bound on the length of the strings. The contribution of this paper is therefore the proof of the upper bound for the length of the strings in the FEL, and the algorithms to simulate one automaton class by another based on the FEL.

Section 2 recalls the classical one-way FA (NFAs and DFAs), and sets out the notation that we use throughout the paper. We introduce the notion of finite exhaustive languages for FAs in Sect. 3, where we also present a generalized simulation algorithm. In Sect. 4, we illustrate the generalized algorithm in the context of the well known NFA to DFA conversion. Specifically, we describe a method for calculating the FEL for the equivalent DFA by using specialized walks through the NFA digraph (state diagram), which we call simple computations.

2 Preliminaries

Let Σ be a nonempty finite set of symbols. We denote the free monoid over Σ by Σ^*. If $w \in \Sigma^*$, then we call w a string or a word. The length of w is denoted by $|w|$, and w_i denotes the i-th symbol of w, where $1 \leq i \leq |w|$. If $\mathcal{A}$ is a set, then $|\mathcal{A}|$ denotes the number of elements in $\mathcal{A}$, and $2^{\mathcal{A}}$ denotes the powerset of $\mathcal{A}$. We denote the set $\{0, 1, \ldots\}$ by $\mathbb{N}$.

We define a generic FA as a 5-tuple $(Q, \Sigma, \delta, S, F)$, where Q is a finite non-empty set of states and δ is a state transition function. Here, $S \subseteq Q$ and $F \subseteq Q$ denote the set of start states and the set of accept states, respectively, where $S \neq \emptyset$. Specific classes of FA are defined in the literature by placing restrictions on the last four members of the 5-tuple. For example, a DFA has a single start state, therefore we can replace S by a single state $q_0 \in Q$.

A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where $q_0 \in Q$ is called the start state, and $\delta : Q \times \Sigma \to Q$ is the transition function, which may be a partial function. The transition function δ can be extended to strings, so that $\widehat{\delta} : Q \times \Sigma^* \to Q$, such that $\widehat{\delta}(q_i, w) = q_{i+|w|}$ if there exists a sequence $q_i, q_{i+1}, \ldots, q_{i+|w|}$, where $q_{t+1} = \delta(q_t, w_{(1+t-i)}),\ i \leq t < (i + |w|)$. The DFA accepts a string w iff $\widehat{\delta}(q_0, w) = q_f$, where $q_f \in F$. Similarly, an NFA is a 5-tuple $(Q, \Sigma, \delta, S, F)$, where $\delta : Q \times \Sigma \to 2^Q$ is the transition function. Again, δ extends to $\widehat{\delta} : 2^Q \times \Sigma^* \to 2^Q$ where $\widehat{\delta}(\mathcal{A}, a) = \bigcup_{q \in \mathcal{A}} \delta(q, a)$ and $\widehat{\delta}(\mathcal{A}, aw) = \widehat{\delta}(\widehat{\delta}(\mathcal{A}, a), w)$ for some $\mathcal{A} \subseteq Q,\ a \in \Sigma$ and $w \in \Sigma^*$. An NFA accepts a string w iff $\widehat{\delta}(S, w) \cap F \neq \emptyset$. If M is an FA, then $\mathcal{L}(M)$ denotes *the language of* M, that is, the set of all the strings accepted by M.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an FA. By *transition*, we refer to the triple (p, a, q), such that $q \in \delta(p, a)$, or $q = \delta(p, a)$ in the case of a DFA, for some $p, q \in Q$ and $a \in \Sigma$. A *computation* of M on a string $w = w_1 w_2 \ldots w_k$ in Σ^*, starting from state q_0, is a sequence $q_0, q_1, \ldots, q_k$ of states in Q such that (q_i, w_{i+1}, q_{i+1}) is a transition for all $0 \leq i < k$. We denote this computation by $\vec{\delta}_{w,q_0}$, and if $q_0 \in S$, we simply

write $\vec{\delta}_w$. The computation is accepting if $q_0 \in S$ and $q_k \in F$. A state $q \in Q$ is *reachable* from state $p \in Q$ if there exists a computation from p to q, otherwise the state q is *unreachable* from p. A state $q \in Q \setminus F$ is *dead* if none of the states in F is reachable from q. The FA M is *trim* if no state in Q is dead or unreachable. The function $\text{TRIM}(M)$ removes, from M, all unreachable and dead states. Note that $\mathcal{L}(M) = \mathcal{L}(\text{TRIM}(M))$.

3 Finite Exhaustive Languages and Generalized Simulations

Given any FA $M = (Q, \Sigma, \delta, S, F)$, it is possible to find a finite sublanguage $\mathcal{L}'(M) \subseteq \mathcal{L}(M)$ such that $\text{TRIM}(M)$ exhausts all its transitions (and states) after computing all strings in $\mathcal{L}'(M)$, since $|Q|$ is finite. We call $\mathcal{L}'(M)$ the finite exhaustive language (FEL) of M, and formally define it as:

Definition 1 (Finite exhaustive language). *Let $M = (Q, \Sigma, \delta, S, F)$ be an FA. Let $\mathcal{T}$ be the set of all transitions in $\text{TRIM}(M)$. Let $f(\vec{\delta}_w) = \{(p_i, a, p_{i+1}) \mid (p_i, a, p_{i+1})$ is a transition in $\vec{\delta}_w = p_0, \ldots, p_i, \ldots, p_{|w|}\}$. Then a finite sublanguage $\mathcal{L}'(M) \subseteq \mathcal{L}(M)$ is exhaustive if $\left(\bigcup_{w \in \mathcal{L}'(M)} f\left(\vec{\delta}_w\right)\right) = \mathcal{T}$.*

Note that the FEL for a given FA is not unique, unless the FA recognizes a finite language. Generally, we seek an FEL with the shortest strings, which ensures the coverage of all accept states.

When simulating a finite automaton M_A by a finite automaton M_B, we proceed as follows. First build the FEL $\mathcal{L}'(M_B)$ for M_B, using a finite set of finite walks on the digraph of M_A. Then run M_A on all strings in $\mathcal{L}'(M_B)$. Assuming M_B processes a string $w \in \mathcal{L}'(M_\text{B})$ in the left-to-right order, we can write down the computation of M_A on w as a sequence $C_\text{qB} = q_{B_0}, w_1, q_{B_1}, w_2, \ldots, w_{|w|}, q_{B_{|w|}}$, such that a triple $\rho_i = (q_{B_i}, w_{i+1}, q_{B_{i+1}})$ appears as subsequence whenever q_{B_i} is a collection of states of M_A that M_A is in whenever w_{i+1} is the next symbol, when the input head is moving right; $q_{B_{i+1}}$ is a collection of M_A states that M_A enters whenever the input head moves right, past w_{i+1}, to the symbol w_{i+2}. The collections q_{B_i} in C_qB are states of M_B and the triples ρ_i are transitions of M_B. The state $q_{B_{|w|}}$ is an accept state in M_B.

Example 2. Figure 1(a) shows an NFA, N_1, whose simulating DFA, M_1, is shown in Fig. 1(b). A candidate FEL for M_1 is {aa, ab, aaa, aab, baa, bab, abaa, abbaa}. Figure 1(c) shows a partially constructed M_1 (bottom) from a run (top) of N_1 on the string aa. We determine the accept state in the partial M_1 as the collection of N_1 states in which the run of N_1 on aa halts, since aa $\in \mathcal{L}(M_1)$.

The next section describes how the general concept of an FEL can be used in the specific case of a DFA that simulates an NFA.

4 NFA to DFA Conversion

We introduce the concept of *simple computations* in Sect. 4.1. We use simple computations to calculate the FEL of a DFA from an equivalent NFA, for NFAs with restrictions on their cycle structure. In Sect. 4.2 we show that simple computations do not always yield the FEL of the equivalent [1] DFA, for some subclass of NFAs. We then extend the simple computations to cover the general class of NFAs.

[1] Henceforth, an *equivalent* DFA refers to the subset-construction equivalent DFA.

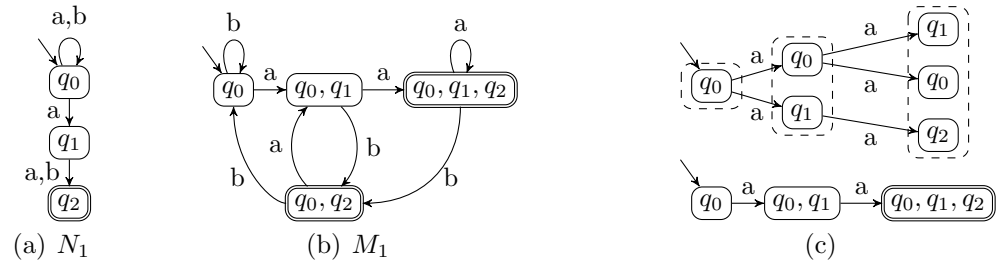


Figure 1. (a) An NFA N_1. (b) A DFA simulating N_1. (c) A run of N_1 on the string aa (top) giving part of the simulating DFA (bottom).

4.1 Simple Computations

Definition 3 (Computation path). *Let $N = (Q, \Sigma, \delta, S, F)$ be an NFA. Then the sequence $C_p = p_0, p_1, \ldots, p_k$ of states in Q is a* computation path *of N if there exists $a_{i+1} \in \Sigma$ such that $p_{i+1} \in \delta(p_i, a_{i+1})$ for all $0 \leq i < k$.*

The reader should note that we use the terms *computation* and *computation path* interchangeably. When the word w which plays a role in the sequence of states is to be emphasized, we use the former, and when we are more interested in the states that can occur in the state sequence, we use the latter.

A computation path $C_\mathrm{p} = p_0, p_1, \ldots, p_k$ is cyclic if there exist $i, j \in \{0, 1, \ldots, k\}$ such that $i < j$ and $p_i = p_j$. The subsequence $p_i, \ldots, p_j$ is a cycle. The cycle is simple if $p_u \neq p_v$ for all $u < v$ in $\{i, \ldots, (j-1)\}$. If a cycle is not simple, then Algorithm 1 can be used to recursively remove all the nested cycles, and return a simple cycle.

In the algorithm, we assume that C_cycle has multiple nested cycles. The algorithm explores all possible orders of removing the nested cycles, removing one cycle at a time, until a simple cycle remains. Different orders of removing nested cycles may result in different simple cycles. We therefore collect the resulting simple cycles into a set R. For example, $\textsc{simplify}([q_0, q_1, q_2, q_3, q_1, q_3, q_0]) = \{[q_0, q_1, q_2, q_3, q_0], [q_0, q_1, q_3, q_0]\}$. We obtain the first simple cycle by deleting the nested cycle $[q_3, q_1, q_3]$; and the second by deleting the nested cycle $[q_1, q_2, q_3, q_1]$.

We now define a *simple computation*, which is a computation path that does not traverse any cycle more than once.

Definition 4 (Simple computation). *Let $C_p = p_0, p_1, \ldots, p_k$ be a computation path of an NFA $N = (Q, \Sigma, \delta, S, F)$. C_p is a simple computation if either*

1. *C_p has at most one cycle; or*
2. *if C_p has multiple cycles, then any two cycles C_{y_1} and C_{y_2} in C_p are such that $\textsc{simplify}(C_{y_1}) \cap \textsc{simplify}(C_{y_2}) = \emptyset$.*

A simple computation is *non-cyclic* if it does not include any cycle; otherwise, it is *cyclic*. If $p_0 \in S$ and $p_k \in F$, then the simple computation is *accepting*. We say that a simple computation $C_\mathrm{p} = p_0, p_1, \ldots, p_k$ *yields* a string $w = w_1 w_2 \ldots w_k$ if $p_{i+1} \in \delta(p_i, w_{i+1})$ for all $0 \leq i < k$. Note that a simple computation may actually yield a non-empty finite set of strings (see Example 5). Also note that if C_p is accepting, then its yield is in $\mathcal{L}(N)$. Henceforth, we denote by W_δ, the union of yields of all the accepting simple computations of an NFA $N = (Q, \Sigma, \delta, S, F)$.

Algorithm 1. Simplify

Require: $C_{\text{cycle}} = p_0, p_1, \ldots, p_m$ where $p_0 = p_m$
  function SIMPLIFY(C_{cycle})
    $R \leftarrow \emptyset$
    SIMPLIFY2(C_{cycle}, R)
    return R
  end function
  procedure SIMPLIFY2($C_{\text{cycle}}, \&R$) ▷ R is a reference parameter.
    $m \leftarrow |C_{\text{p}}| - 1$
    $\eta \leftarrow 0$
    for all $(i, j) \mid (i < j) \ \wedge \ (p_i = p_j) \ \wedge \ \neg(i = 0 \ \wedge \ j = m)$ **do**
      $\eta \leftarrow \eta + 1$
      $C_{\text{temp}} \leftarrow p_0, \ldots, p_{i-1}, p_j, \ldots, p_m$ ▷ Delete the cycle $p_i, \ldots, p_j$.
      SIMPLIFY(C_{temp}, R)
    end for
    if $\eta = 0$ **then** ▷ C_{cycle} is simple.
      $R \leftarrow R \cup \{C_{\text{cycle}}\}$
    end if
  end procedure

Example 5. Table 1 lists some of the simple computations and their corresponding yields for the NFA N_3, in Fig. 2. To illustrate Definition 4, note that q_0, q_1, q_2, q_2 is a simple computation, since it has only one cycle. Also, q_0, q_1, q_2, q_2, q_1 is a simple computation – although it has two cycles, it is easy to see that SIMPLIFY($[q_1, q_2, q_2, q_1]$) $\cap$ SIMPLIFY($[q_2, q_2]$) $= \{[q_1, q_2, q_1]\} \cap \{[q_2, q_2]\} = \emptyset$. On the other hand, the computation path $D_{\text{p}} = q_0, q_1, q_2, q_1, q_2, q_2, q_1, q_3$ is not a simple computation, since SIMPLIFY($[q_1, q_2, q_1]$) $\cap$ SIMPLIFY($[q_1, q_2, q_2, q_1]$) $= \{[q_1, q_2, q_1]\} \neq \emptyset$.

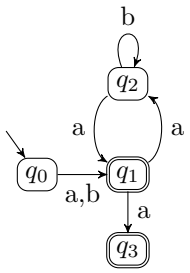


Figure 2. N_3.

Table 1. Some simple computations of N_3, and their yields.

Simple computation	Type	Yield
q_0, q_1, q_2	non-cyclic	{aa, ba}
q_0, q_1, q_2, q_2	cyclic	{aab, bab}
q_0, q_1	accepting non-cyclic	{a, b}
q_0, q_1, q_3	accepting non-cyclic	{aa, ba}
q_0, q_1, q_2, q_1	accepting cyclic	{aaa, baa}
q_0, q_1, q_2, q_2, q_1	accepting cyclic	{aaba, baba}
$q_0, q_1, q_2, q_2, q_1, q_3$	accepting cyclic	{aabaa, babaa}

It is known that there are n-state NFAs for which the smallest equivalent minimal DFA has $O(2^n)$ states. One therefore has to consider whether all simple computations of length up to 2^n must be considered when the FEL is constructed. To that end, we first show an upper bound on the union of the yields of all simple computations in an NFA.

Lemma 6. *If $N = (Q, \Sigma, \delta, S, F)$ is an NFA with $|Q| = n$ states and w is the longest string in W_δ, then $|w|$ is $O(n^3)$.*

Proof. Let $N = (Q, \Sigma, \delta, S, F)$ be an NFA. Let $G_{\text{N}} = (Q, E)$ be a digraph representing the transitions of N, where $E = \{(p, q) \mid q \in \delta(p, a)$ for some $a \in \Sigma\}$. Note that the edges of G_{N} are not labelled with transition symbols. Let C_{p} be an accepting

simple computation of the NFA N. If $|Q| = n$, then the maximum number of edges in G is $\left(2 \cdot \binom{n}{2} + n\right) = n^2$. If CY is a simple cycle in C_p, then at least one edge of CY is unique in C_p, otherwise the cycle would have been traversed more than once, thus contradicting Definition 4. Therefore the maximum number of simple cycles in a simple computation is n^2. Note that if C_p has n^2 simple cycles, then none of the edges has been repeated, hence the length of C_p is $(n^2 + 1)$, and the length of each string in the yield of C_p is n^2.

By the pigeon hole principle, any subsequence of C_p having $(n+1)$ states contains a simple cycle. If $|C_\mathrm{p}| = n(n+1) = n^2 + n$, then C_p has at least n cycles. Since the number of simple cycles may be less than n^2, it may be possible to add more states to C_p while keeping it a simple computation. However if $|C_\mathrm{p}| = n^2(n+1) = n^3+n^2$, then C_p has at least n^2 simple cycles. Since $|C_\mathrm{p}| > (n^2+1)$, it follows that some edge in C_p is not unique, therefore C_p is not a simple computation. Note that when $|C_\mathrm{p}| = n^3$, we cannot be certain that C_p has at least n^2 cycles. Therefore, the maximum length of a simple computation, and thus the longest string in W_δ, is $O(n^3)$. □

Proposition 7. *Let $N = (Q, \Sigma, \delta, S, F)$ be an NFA with no accepting cyclic simple computations. Let M be the subset-construction equivalent DFA. Then* TRIM(M) *uses all its transitions after computing all the strings in W_δ.*

Proof. Since N has no accepting cyclic simple computations, it follows that all cycles of N (if any) are outside TRIM(M). Thus TRIM(M) is a directed acyclic graph. Therefore, $\mathcal{L}(N)$ is finite and $\mathcal{L}(N) = W_\delta$. □

Proposition 7 informally states that an accepting non-cyclic computation in an NFA has an isomorphic accepting non-cyclic computation in an equivalent DFA derived by the subset construction. The next section compares cycles in an NFA and its subset-construction equivalent DFA.

4.2 Extending Simple Computations

If an NFA contains cycles in its trim, then there is no guarantee that W_δ uses all transitions in any equivalent DFA, as accepting cyclic computations of an NFA are generally not isomorphic to accepting cyclic computations of its equivalent DFA. For example, W_δ for the NFA N_4 in Fig. 3 does not use some transitions of M_4, the minimal equivalent DFA.

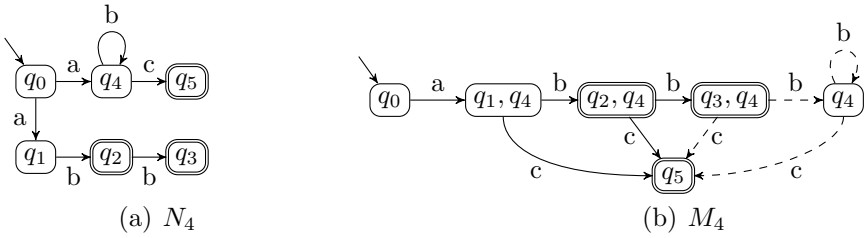


Figure 3. (a) An NFA N_4, with $W_\delta = \{\mathrm{ab}, \mathrm{abb}, \mathrm{abc}, \mathrm{ac}\}$. (b) The trim of the subset-construction equivalent DFA, M_4. No computation of M_4 on any string in W_δ uses any of the dashed transitions. Notice that the cyclic computation q_0, q_4, q_4, q_5 in N_4 is converted to the non-cyclic computation $\{q_0\}, \{q_1, q_4\}, \{q_2, q_4\}, \{q_5\}$ in M_4. Also notice that M_4 is minimal.

The transitions of an NFA's equivalent DFA missed by W_δ can be accounted for by extending computation paths that yield W_δ, by allowing them to traverse cycles

more than once. An upper bound on the number of times a cycle must be traversed to cover the missed transitions must exist, since the equivalent DFA has a finite number of transitions.

The lack of isomorphism between a cyclic computation of an NFA and a cyclic computation of its equivalent DFA is not surprising. A DFA's computation is generally a parallel composition of several computations of its equivalent NFA. Thus to replicate the cyclic behaviour of an NFA's cycle in its equivalent DFA, we consider the interaction between a cyclic simple computation and other computation paths of the NFA. We present the interactions, in order of increasing complexity, in Lemmas 9, 12 and 13. We first set out some new notation.

Let $C_{\mathrm{p}} = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ be an accepting cyclic simple computation of an NFA N, which includes a cycle $CY = p_i, \ldots, p_j$. Let x be a string in the yield of C_{p}. Let W'_{CY} be the union of the yields of all accepting computation paths of N that do not include the cycle CY. The largest integer, r, such that $w_1 w_2 \ldots w_u = x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^r$, for all $w \in W'_{CY}$, is the *wrap value* of the cycle CY. If $r = \infty$, then the cycle CY *overlaps* some other cycle in N. We formally define exactly when two cycles overlap in Definition 11.

Definition 8 (Free cycle). *Let $C_p = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ be an accepting cyclic simple computation of an NFA N, which includes a cycle $CY = p_i, \ldots, p_j$. Let $C_{p_i} = p_0, p_1, \ldots, p_i$ be a prefix of C_p such that $|C_{p_i}| = i + 1$. Let $C_{t_i} = t_0, t_1, \ldots, t_i$ be any computation path, where $t_0 \in S$ and $C_{p_i} \neq C_{t_i}$. Let W_{p_i} and W_{t_i} be the yields of C_{p_i} and C_{t_i}, respectively. Then the cycle CY is* free *if*

1. *The wrap value of CY is 0 and*
2. *$W_{p_i} \cap W_{t_i} = \emptyset$.*

Lemma 9. *Let $C_p = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ be an accepting cyclic simple computation of an NFA N, which includes a* free *cycle $CY = p_i, \ldots, p_j$. Let M be a DFA equivalent to N. Then M has a computation path isomorphic to C_p.*

Proof. Let $N = (Q, \Sigma, \delta, S, F)$. Let x be a string in the yield of C_{p}. Let $C_{\mathrm{t}} = t_0, t_1, \ldots, t_l$ be an accepting computation path yielding a string y. For brevity, we ignore all other computation paths besides C_{p} and C_{t}. Consider the prefixes $C_{p_i} = p_0, p_1, \ldots p_i$ and $C_{t_i} = t_0, t_1, \ldots t_i$ of C_{p} and C_{t}, respectively, where $|C_{p_i}| = |C_{t_i}| = i + 1$. If $C_{p_i} = C_{t_i}$, then the lemma holds true trivially, since Definition 8 is violated. Assume $C_{p_i} \neq C_{t_i}$. Since CY is a free cycle in N, then $y_\nu \neq x_\nu$, for some $1 \leq \nu \leq i$. In the worst case, when $y_1 \ldots y_{i-1} = x_1 \ldots x_{i-1}$, one of the branches of the parallel composition of C_{p} and C_{t} results in the simple computation path $C_{\mathrm{pt}} = \{p_0, t_0\}, \{p_1, t_1\}, \ldots, \{p_{i-1}, t_{i-1}\}, \{p_i\}, \ldots, \{p_j\}, \ldots, \{p_k\}$ in M. It is immediate that C_{p} is isomorphic to C_{pt}. □

Corollary 10. *If all cycles of an NFA $N = (Q, \Sigma, \delta, S, F)$ are free, then W_δ is the FEL of a DFA equivalent to N.*

Proof. Let M be the DFA equivalent to N. By Lemma 9, M has an accepting cyclic simple computation isomorphic to each accepting cyclic simple computation of N. Thus each accepting cyclic computation of N is isomorphic to some accepting cyclic computation of M. By Proposition 7, M has an accepting non-cyclic computation isomorphic to each accepting non-cyclic computation of N. □

Definition 11 (Cycle overlap). *Let* $C_p = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ *and* $C_t = t_0, t_1, \ldots, t_u, \ldots, t_v, \ldots, t_l$ *be accepting cyclic simple computations of an NFA* N*, which include cycles* $CX = p_i, \ldots, p_j$*, and* $CY = t_u, \ldots, t_v$*. The cycles* CX *and* CY overlap *if there exists a string* w *such that both* $p_0, p_1, \ldots, p_i, (p_{i+1} \ldots, p_j)^\gamma$ *and* $t_0, t_1, \ldots, t_u$, $(t_{u+1} \ldots, t_v)^\chi$ *contain* w *in their yields, for some* $\chi, \gamma \in \{1, 2, \ldots\}$.

Lemma 12. *Let* $C_p = p_0, p_1, \ldots, p_i, \ldots, p_j, \ldots, p_k$ *be an accepting cyclic simple computation of an NFA* $N = (Q, \Sigma, \delta, S, F)$*, which includes a cycle* $CY = p_i, \ldots, p_j$*. Let* $M = (Q', \Sigma, \delta', q_0, F')$ *be the equivalent DFA. Let* r *be the wrap value of* CY*. Assume the cycle* CY *is not free. Let* W^λ *be the yield of a computation path of* N *that completes the cycle* CY λ *times. If* CY *does not overlap with any other cycle, then* W^{r+2} *exhaust all transitions of* M *that* M *would use to recognize* $W^{r+\phi}$*, for all* $\phi > 2$.

Proof. Let $x = x_1 x_2 \ldots x_i \ldots x_j \ldots x_k$ be a string in the yield of the accepting simple computation C_p. Let $x' = x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^{r+1}$. Let $C_\mathrm{t} = t_0, t_1, \ldots, t_l$ be an accepting computation path of N. Let y be a string in the yield of C_t such that $y_1 y_2 \ldots y_u = x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^r$. For brevity, we ignore all other computation paths besides C_p and C_t. Consider a computation path in M resulting from the parallel composition of C_p and C_t, $C_\mathrm{pt} = \{p_0, t_0\}, \{p_1, t_1\}, \ldots, \{p_i, t_i\}, \ldots, \{p_j, t_j\}$. Suppose $p_i = t_i$. Since $x' \neq y_1 y_2 \ldots y_{[i+(r+1)(j-i)]}$, it follows that if $\overrightarrow{\delta}\ (t_0, x') = t_0, t_1, \ldots, t_{[i+(r+1)(j-i)]}$ then

$$\overrightarrow{\delta}\ (t_0, x') = \emptyset. \tag{1}$$

But, $\delta'(\{p_i, t_i\}, x_{i+1}) = \delta'(\{p_i\}, x_{i+1}) = \{p_{i+1}, t_{i+1}\}$. Therefore

$$\delta'(\{p_{[i+(r+1)(j-i)]}, t_{[i+(r+1)(j-i)]}\}, x_{[i+(r+1)(j-i)+1]}) = \delta'(\{p_i, t_{[i+(r+1)(j-i)]}\}, x_{i+1}) = \{p_{i+1}, t_{i+1}\}\ .$$

Thus the computation of M on $x' x_{i+1}$ is cyclic. Since M is deterministic, if follows that its computation on $x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^{r+\eta}$, for all $\eta > 1$, repeats the cycle $\{p_{i+1}, t_{i+1}\}, \{p_{i+2}, t_{i+2}\} \ldots, \{p_{i+1}, t_{i+1}\}$, and does not use any new transitions.

Now suppose $p_i \neq t_i$. Consider the state

$$\{p_{[i+(r+1)(j-1)]}, t_{[i+(r+1)(j-1)]}\} = \{p_i, t_{[i+(r+1)(j-1)]}\} = \widehat{\delta'}(q_0, x')\ .$$

If $t_{[i+(r+1)(j-1)]} = t_i$, then we have a cycle, and we are done. If $t_{[i+(r+1)(j-1)]} \neq t_i$ then, because of (1), $t_{[i+(r+1)(j-1)]} \in \delta(p_{i-1}, x_i)$. Consequently $\{p_i, t_i\}$ includes the state $t_{[i+(r+1)(j-1)]}$, which contradicts $t_{[i+(r+1)(j-1)]} \neq t_i$. The only way to resolve this is by fixing $\widehat{\delta'}(q_0, x') = \{p_i\}$. Since the state $\{p_i\}$ has appeared in M, the subset construction dictates the existence of the cycle $CY' = \{p_i\}, \{p_{i+1}\}, \ldots, \{p_j\}$ in M. Extending x' to $x'' = x_i (x_{i+1} \ldots x_j)^{r+2}$ ensures that the computation of M on w'' completes this cycle. Again the computation of M on $x_1 x_2 \ldots x_i (x_{i+1} \ldots x_j)^{r+\phi}$, for all $\phi > 2$, repeats the cycle CY' and does not use any new transitions. □

Lemma 13. *Let* $C_p = p_0, \ldots, p_i, \ldots, p_j, \ldots, p_k$ *and* $C_t = t_0, \ldots, t_u, \ldots, t_v, \ldots, t_l$ *be accepting cyclic simple computations of an NFA* N*, which include cycles* $CX = p_i, \ldots, p_j$*, and* $CY = t_u, \ldots, t_v$*. Let* $l_{CX} = (j - i)$ *and* $l_{CY} = (v - u)$*, and* g *be the least common multiple (lcm) of* l_{CX} *and* l_{CY}*. Let* M *be a DFA equivalent to* N*. If* CX *and* CY *are the only overlapping cycles of* N*, then* M *has a cycle* C_{XY} *such that*

$|C_{XY}| = (g+1)$. *The shortest strings that can traverse the cycle* C_{XY} *is in the yield of the computation paths*

$$C_p^g = p_0, p_1, \ldots, p_i, (p_{i+1}, \ldots, p_j)^{g/l_{CX}}, p_{j+1}, \ldots, p_k \text{ or}$$
$$C_t^g = t_0, t_1, \ldots, t_u, (t_{u+1}, \ldots, t_v)^{g/l_{CY}}, t_{v+1}, \ldots, t_l \ .$$

Proof. Without loss of generality, assume the worst case, when $i = v + \lambda(v-u)$, for some integer $\lambda \geq 0$. Then the first state in the parallel composition of C_{p} and C_{t}, C_{pt}, in which states from both cycles appear is $\{p_i, t_{v+\lambda(v-u)}\}$. Let $\{p_i, t_{v+\lambda(v-u)}\}$ be the first state in the cycle C_{XY} of C_{pt}. To complete the cycle in C_{pt}, we must encounter the state $\{p_i, t_{v+\lambda(v-u)}\} = \{p_j, t_{v+\lambda(v-u)}\}$ again. Going through the two cycles simultaneously, starting from the first occurrence of $\{p_i, t_{v+\lambda(v-u)}\}$, the h^{th} occurrence of p_i is $p_{i+hl_{\mathrm{CX}}}$. The m^{th} occurrence of $t_{v+\lambda(v-u)}$ is $t_{v+\lambda(v-u)+ml_{\mathrm{CY}}}$. The state $\{p_i, t_{v+\lambda(v-u)}\}$ appears again when $i + hl_{\mathrm{CX}} = v + \lambda(v-u) + ml_{\mathrm{CY}}$. Since $i = v + \lambda(v-u)$, it follows that

$$hl_{\mathrm{CX}} = ml_{\mathrm{CY}} \ . \tag{2}$$

Since g is the lcm of l_{CX} and l_{CY}, the smallest values of h and m satisfying (2) are $h = g/l_{\mathrm{CX}}$ and $m = g/l_{\mathrm{CY}}$. □

Corollary 14. *If an NFA has* η *overlapping cycles with lengths*[2] $l_1, l_2, \ldots, l_\eta$, *and* g *is the lcm of the* l_i*'s, then the FEL for an equivalent DFA must include yield from accepting computation paths involving* $2, 3, \ldots, (g/l_i)$ *repetitions of the* i^{th} *overlapping cycle, for all* $1 \leq i \leq \eta$.

Remark 15. By Corollary 14, the longest cycle in a DFA simulating an n-state NFA occurs when all the overlapping cycles have relatively prime lengths and the states in any two cycles are disjoint. In this case $\sum_{i=1}^{\eta} l_i \leq n$. From the upper bound to Landau's function in [7,5], Corollary 16 follows.

Corollary 16. *The* g *in Corollary 14 is in* $O(e^{\sqrt{n \log n}})$.

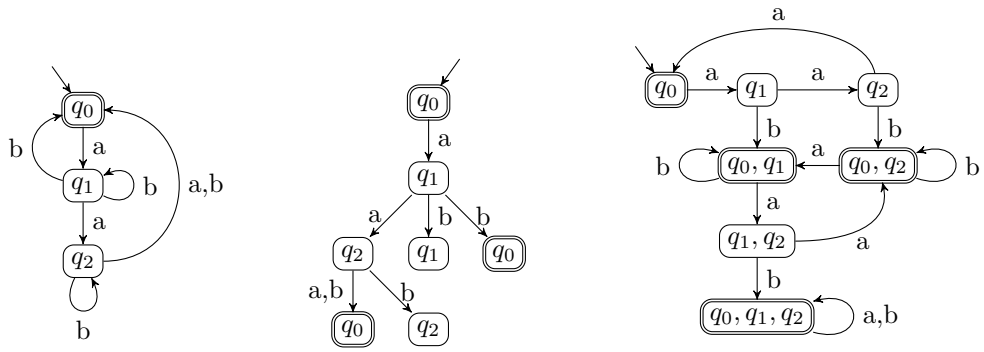
Note than although a DFA may require exactly 2^n states in order to simulate an n-state NFA, the longest cycle in the simulating DFA has only $O(e^{\sqrt{n \log n}})$ states. Furthermore, each simple cycle in an NFA can be traversed by an (n+1)-state computation path. Therefore the FEL of the equivalent DFA does not require strings longer than $O(e^{\sqrt{n \log n}})$.

We now consider the algorithm to construct an FEL for a DFA equivalent to a given NFA.

4.3 Construction Details

Representing NFAs using Cycle Trees From the discussion in Sect. 4.2, it is clear that we need to identify all cycles in a given NFA. We thus represent an NFA having a single start state by a tree, which we call a *cycle tree*. We create a cycle tree by performing depth first search from a start state, adding a tree node for each state encountered. We add an edge labelled a from a parent node q_p to a child node q_c if the NFA has a transition (q_p, a, q_c). A branch of the traversal stops when either it encounters a previously seen state, along the current path from the start state, or

[2] Here the length is the number of states in the cycle minus one.



(a) N_5. (b) N_5^T. (c) M_5.

Figure 4. (a) A 3-state NFA example from the set of NFAs in [6], for which an n-state NFA requires the simulating DFAs to have exactly $(2^n - 1)$ states. (b) A cycle tree representation of N_5. (c) The DFA simulating N_5.

when the current state does not have transitions. Algorithm 2 creates a cycle tree from a single start state of an NFA $N = (Q, \Sigma, \delta, S, F)$. [3] In the algorithm, $q_0 \in S$. The function CREATETREE(q_0) creates the cycle tree and returns a pointer to the root of the tree.

Algorithm 2. Creating a cycle tree

```
function CYCLETREE(q0)
    root ← q0
    BUILD(root, ∅)
    return root
end function
procedure BUILD(p, V_path)
    V_path ← {p} ∪ V_path
    for all a ∈ Σ do
        for all q ∈ δ(p, a) do
            EDGE(p, a, q)
            if q ∉ V_path then
                BUILD(q, V_path)
            end if
        end for
    end for
end procedure
```

In the procedure BUILD(p, V_{path}), p is a tree node corresponding to the current NFA state, and V_{path} is a set of tree node labels (these also correspond to NFA state labels) encountered along the current path from the root. The procedure EDGE(p, a, q) creates a directed edge from a tree node labelled p to a new tree node labelled q. The directed edge is labelled with the symbol a.

Example 17. Applying Algorithm 2 to the NFA N_5 in Fig. 4(a) results in the cycle tree N_5^T in Fig. 4(b).

Properties of NFAs correspond to properties of cycle trees as follows. If an NFA $N = (Q, \Sigma, \delta, S, F)$ has a cyclic computation path $C_{\text{p}} = p_0, p_1, \ldots, p_i, \ldots, p_j$, having

[3] If the NFA has multiple start states, we run the algorithm for each start state, resulting in a forest of cycle trees.

a single simple cycle $p_i, \ldots, p_j$ and $p_0 \in S$, then the path C_p also appears as a path in some cycle tree, where p_0 is the root and p_j is a leaf. If N has an accept state $q_f \in F$, then all occurrences of nodes labelled q_f in the cycle tree are marked as accepting. If the NFA is trim, then, in its cycle trees, every non-accepting leaf node represents an NFA state in a cycle. Note that an accepting leaf node may also indicate a cycle.

The advantage of representing an NFA by a cycle tree is that if the NFA is not trim, then the depth first search avoids all unreachable states. If the resulting cycle tree has a node representing a dead NFA state, then we can identify it as a non-cyclic non-accepting leaf node. Thus we can delete dead states in $O(h)$ time, where h is the depth of the cycle tree.

The computation of a cycle tree is similar to that of an NFA, except when the computation reaches a *cyclic leaf node* v_cy having the same label as some internal node v_in, along the current path from the root. In this case, the computation jumps, without consuming any symbol, to the node v_in.

Algorithm 3. Simple computation

```
function allYield(N^T)
    V_an ← acceptNodes(N^T)
    W_δ ← ∅
    n_c ← cycles(N^T)
    for all 0 ≤ i ≤ n_c do
        for all v_an ∈ V_an do
            V_s ← ∅
            W ← ∅
            trace(W, v_an, V_s, i)
            W_δ ← W_δ ∪ W
        end for
    end for
    return reverse(W_δ)
end function
```

Simple Computations from a Cycle Tree We generate the yield of all accepting simple computations of a cycle tree using Algorithm 3. In the algorithm, the function allYield(N^T) returns the union of the yield of all simple computations of the cycle tree N^T; V_an is the set of accept nodes of N^T; W_δ is a set of strings, which becomes the union of yields of all accepting simple computations in the end; cycles(N^T) returns the number of simple cycles in N^T; and reverse(W_δ) reverses every string in W_δ. The procedure trace($W, v_\mathrm{an}, V_\mathrm{s}, i$) traces reversed simple computations, with at most i cycles, from the accept node v_an to the root of N^T, and builds the reverse of strings in the yield. A more detailed description of trace is given in Algorithm 4.

In Algorithm 4, trace($\&W_\delta, v_\mathrm{b}, V_\mathrm{s}, n_\mathrm{c}$) creates the reversed yields of all accepting simple computations that halt at the accept node, v_b. The parameter W_δ is a set of reversed strings; V_s is the set of cyclic leaf nodes visited so far; and n_c is the maximum number of cycles that the simple computation may go through. The procedure traces

Algorithm 4. Trace

```
procedure TRACE(&W_δ, v_b, V_s, n_c)                    ▷ R is a reference parameter.
    v_cd ← v_b, v_pt ← parent(v_cd)
    while v_cd ≠ root do
        V_lf ← JUMPTOSET(v_cd)
        if V_lf ≠ ∅ then
            W'_δ ← W_δ, W_δ ← ∅
            for all v_lf ∈ V_lf do
                if v_lf ∉ V_s, |V_s| > n_c then
                    W_nw ← W'_δ
                    V_sn ← V_s ∪ {v_lf}
                    TRACE(W_nw, v_lf, V_sn, n_c)
                    W_δ ← W_δ ∪ W_nw
                    V_sn ← V_s, W_nw ← W'_δ
                    EXTEND(W_nw, v_ch, v_pt)            ▷ W_nw is a reference parameter.
                    TRACE(W_nw, v_pt, V_sn, n_c)
                    W_δ ← W_δ ∪ W_nw
                else
                    EXTEND(W'_δ, v_ch, v_pt)            ▷ W'_δ is a reference parameter.
                    TRACE(W'_δ, v_pt, V_s, n_c)
                    W_δ ← W_δ ∪ W'_δ
                end if
            end for
            return
        else
            EXTEND(W_δ, v_cd, v_pt)                     ▷ W_δ is a reference parameter.
            v_pt ← parent(v_pt)
            v_cd ← parent(v_cd)
        end if
    end while
    V_lf ← JUMPTOSET(v_cd)
    if (V_lf ≠ ∅) ∧ (|V_s| > n_c) then
        for all v_lf ∈ V_lf do
            W_nw ← W'_δ
            if v_lf ∉ V_s then
                V_sn ← V_s ∪ {v_lf}
            else if |V_s| > n_c then
                V_sn ← {v_lf}
            end if
            TRACE(W_nw, v_lf, V_sn, n_c)
            W_δ ← W_δ ∪ W_nw ∪ W'_δ
        end for
    end if
end procedure
```

the reverse of all simple computations from an accept node v_b to the root node, while building the reverse of strings in the union of the yields of the reversed simple computations. When TRACE encounters an internal node v_in having the same label as a cyclic leaf node $v_\mathrm{lf} \notin V_\mathrm{s}$, such that a forward path exists from v_in to v_lf, the reversed simple computation is duplicated. One copy proceeds with the parent node of the node v_in. The other copy proceeds from the leaf node v_lf. The procedure EXTEND$(\&W_\delta, v_\mathrm{cd}, v_\mathrm{pt})$ collects all transition symbols on the edge $(v_\mathrm{pt}, v_\mathrm{cd})$ into a set A, and computes $W_\delta \leftarrow W_\delta \times A$. The function JUMPTOSET$(v_\mathrm{cd})$ returns a set of all

cyclic leaf nodes having the same label as an internal node labelled v_{cd}, if there is a forward path from v_{cd} to each of the cyclic leaf nodes.

Example 18. Algorithm 3 applied to N_5^{T}, in Fig. 4(b) yields the set {ϵ, aaa, aab, abaaa, abaab, abbaabb, abbaaba, aabbab, aaaabb, aabaab, aabb, aababb, aabbabb, abbaab, abaaba, abbaaa, abaabb, ababa, abb, ab, abababa, abababb, ababb, aaaab, aabab, abaa, abab, aaba, aabaabb, abaaab, ababaa, ababab, ababaab, ababbab}, which exhausts all transitions of M_5 in Fig. 4(c).

Extending simple computations The algorithm for generating the yield of extended computations is similar to Algorithm 3, except that a cycle CY is repeated $2, 3, \ldots, \lambda$, where $\lambda = \max([r+2], g/l_{\text{CY}})$, if r is the wrap value of the cycle, l_{CY} is the length of CY (measured in terms of the number of transitions) and g is the lcm of lengths of all cycles that overlap with CY, including CY. In place of the set V_s, we use a table that maps each cyclic leaf to the number of times it has been encountered.

We determine the wrap value of a cycle from the cycle tree using Algorithm 5. In the algorithm, V_{lf} is the set of all leaves and V_{cy} is the set of all cyclic leaves. The variable `map` is a table that associates each node in V_{cy} to an integer, representing the maximum number of times a cycle is used in a computation. The function pathYield(root, v_{lf}) returns the yield of the computation path from the root to the leaf v_{lf}. The procedure run(map, v_{lf}, w) considers all possible runs of the string w on the NFA represented by the forest of cycle trees. Each run avoids the state v_{lf} because the string w is already known to be in the yield of a computation path root, $\ldots, V_{\text{lf}}$. Each run also keeps track of the number of times each state in V_{cy} is encountered. At the end of each run, a table entry $(v_{\text{cy}}, \mu) \in$ `map` is updated to (v_{cy}, ν) if $\mu < \nu$ where ν is the number of times the node v_{cy} was encountered and $v_{\text{cy}} \in V_{\text{cy}}$. In the end, the table `map` associates each cyclic leaf node with its wrap value.

Algorithm 5. Wrap value

```
map ← V_cy × {0}
for all v_lf ∈ V_lf do
    W ← pathYield(root, v_lf)
    for all w ∈ W do
        run(map, v_lf, w)
    end for
end for
return map
```

We determine overlapping cycles using Algorithm 6. The algorithm groups cyclic leaf nodes that represent overlapping cycles into a set. Note that the equality of cyclic leaf nodes is not based on labels. In the algorithm, the function getPath(root, v_{cy}) returns a computation path from the root to the node v_{cy}. Therefore C_{PATH} is the set of computation paths from the root to every cyclic leaf node. The function stretchYield(C_{p}) extends the computation path C_{p} by going through the cycle several times until the length of the extended computation path is at least $2|Q|+1$. The function then returns the yield of this extended computation path. The function pathRun(C_{q}, w) runs the string w on the sub-NFA defined by C_{p}. The function returns `true` if the computation does not hang, otherwise it returns `false`. The function lastState(C_{q}) return the last state in the computation path C_{q}.

Algorithm 6. Cycle overlap

$C_{\text{PATH}} \leftarrow \bigcup_{v_{\text{cy}} \in V_{\text{cy}}} \text{GETPATH}(\text{root}, v_{\text{cy}})$
$V_{\text{overlap}} \leftarrow \emptyset$
for all $C_{\text{p}} \in C_{\text{PATH}}$ **do**
  $W \leftarrow \text{STRETCHEDYIELD}(C_{\text{p}})$
  for all $w \in W$ **do**
    $R_{\text{overlap}} \leftarrow \emptyset$
    for all $C_{\text{q}} \in C_{\text{PATH}} \setminus \{C_{\text{p}}\}$ **do**
      if $\text{PATHRUN}(C_{\text{q}}, w)$ **then**
        $R_{\text{overlap}} \leftarrow R_{\text{overlap}} \cup \text{LASTSTATE}(C_{\text{q}})$
      end if
    end for
    $V_{\text{overlap}} \leftarrow V_{\text{overlap}} \cup R_{\text{overlap}}$
  end for
end for

Constructing the DFA using FEL Given an NFA and its cycle trees, Algorithm 7 summarizes the construction of its simulating DFA.

Algorithm 7. DFA construction

$W \leftarrow \text{YIELD}(N^{\text{FT}})$
$q_0 \leftarrow S$, $Q' \leftarrow \emptyset$
$Q' \leftarrow \{q_0\} \cup Q'$
for all $w \in W$ **do**
  $q \leftarrow q_0$
  for all $0 < i \leq |w|$ **do**
    $q' \leftarrow \delta'(q, w_i)$
    if $q' = \emptyset$ **then**
      $q' = \bigcup_{r \in q} \delta(r, w_i)$
      $Q' \leftarrow \{q'\} \cup Q'$
    end if
    $\text{TRANS}(q, q', w_i)$
    $q \leftarrow q'$
  end for
  $\text{ACCEPT}(q)$
end for

In the algorithm, N^{FT} is an NFA represented as a forest of cycle trees. The given NFA is $N = (Q, \Sigma, \delta, S, F)$, and the equivalent DFA to be constructed is $M = (Q', \Sigma, \delta', q_0', F')$. The function $\text{YIELD}(N^{\text{FT}})$ creates simple and extended computation yields from all cycle trees in N^{FT} and returns $\mathcal{L}'(M)$, the FEL of the DFA M. The function $\text{TRANS}(q, q', w_i)$ creates a transition on the symbol w_i from the state q to the state q'. The function $\text{ACCEPT}(q)$ marks the state q as an accept state. The algorithm incrementally creates the equivalent DFA. The partial DFA runs each string in W either to its end or until the computation dies in the middle of the string. If the partial DFA consumes the entire string, we mark the last state reached as an accept state. If the computation dies in state q, in the middle of a string, we determine the next DFA state, q', as the set of states that the NFA would be in after consuming the next symbol, w_i. If q' is not already a state in the partial DFA, we add it. Then we add transition (q, w_i, q'), and continue the computation from q'.

5 Conclusion

We argued that if a finite automaton M' in finite automaton class B simulates another finite automaton M in class A, then it is possible to generate the finite exhaustive language, $\mathcal{L}'(M)$, of M' from walks through the diagraph of M. Furthermore, if we know the structure of a collection of states of M relative to a single state of M', we can then construct M' by summarizing the behaviour of actual runs of M on all strings in $\mathcal{L}'(M)$. We presented algorithms to construct an FEL from an NFA, and to use the FEL to build a simulating DFA. We noted that the length of the strings in the FEL has an upper bound of $O(e^{\sqrt{n \log n}})$.

The algorithms to construct the FEL in the NFA to DFA conversion problem are somewhat intricate compared to the subset construction. Nevertheless, our finite exhaustive language approach is more general, and we claim that it applies to all simulations of one finite automata class by another. Moreover, it provides a string-based approach to the problem, as opposed to the traditional state-based approach.

For future work, we are in the process of illustrating the concept for two-way finite automata.

References

1. J. A. Brzozowski and E. Leiss: *On equations for regular languages, finite automata, and sequential networks.* Theoretical Computer Science, 10(1) 1980, pp. 19–35.
2. M. Crochemore, L. Giambruno, and A. Langiu: *On-line construction of a small automaton for a finite set of words.* International Journal of Foundations of Computer Science, 23(2) 2012, pp. 281–301.
3. J. Daciuk: *Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings*, in Lecture Notes in Computer Science, J.-M. Champarnaud and D. Maurel, eds., vol. 2608, 2003, pp. 255 – 261.
4. J. Hopcroft and J. Ullman: *Introduction to automata theory, languages and computation*, Addison-Wesley, 1979.
5. E. Landau: *Handbuch der Lehre von der Verteilung der Primzahlen*, vol. 1, Chelsea, 2 ed., 1953.
6. A. Meyer and M. Fischer: *Economy of description by automata, grammars, and formal systems*, in 12th Annual Symposium on Switching and Automata Theory, 1971, pp. 188–191.
7. J.-L. Nicolas: *On Landau's function g(n)*, in The Mathematics of Paul Erdös I, Springer, 1997, pp. 228–240.
8. M. O. Rabin and D. Scott: *Finite automata and their decision problems.* IBM Journal of Research and Development, 3(2) 1959, pp. 114–125.
9. B. Watson: *A new algorithm for the construction of minimal acyclic DFAs.* Science of Computer Programming, 48(2) 2003, pp. 81–97.