ГОДИШНИК НА СОФИЙСКИЯ УНИВЕРСИТЕТ „СВ. КЛИМЕНТ ОХРИДСКИ“

ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА
Книга 2 — Приложна математика и информатика
Том 92, 1998

ANNUAIRE DE L'UNIVERSITE DE SOFIA „ST. KLIMENT OHRIDSKI“

FACULTE DE MATHEMATIQUES ET INFORMATIQUE
Livre 2 — Mathématiques Appliquée et Informatique
Tome 92, 1998

DIRECT CONSTRUCTION OF MINIMAL ACYCLIC FINITE STATES AUTOMATA

STOYAN MIHOV

This paper presents automaton construction algorithms based on the method for direct building of minimal acyclic finite states automaton for a given list [2]. A detailed presentation of the base algorithm with correctness and complexity proofs is given. The memory complexity of the base algorithm is $O(m)$ and the worst-case time complexity is $O(n\log(m))$, where n is the total number of letters in the input list, m is the size of the resulting minimal automaton. Further we present algorithms for direct construction of minimal automaton presenting the union, intersection and difference of acyclic automata. In the cases of intersection and difference only the first input automaton has to be acyclic. The memory complexity of those construction algorithms is $O(m)$, and the time complexity is $O(n\log(m))$ for union and $O(n_1 + n\log(m))$ for intersection and difference, where n_1 is the total number of letters in the first automaton language, n is the number of all letters in the resulting automaton language and m is the number of states of the resulting minimal automaton. For construction of minimal automata for large scale languages, in the practice our algorithms deliver significantly better efficiency than the standard algorithms.

Keywords: minimal acyclic finite states automaton, construction of minimal automaton

1991/95 Math. Subject Classification: 68Q68, 68Q45

1. INTRODUCTION

The standard methods for constructing a minimal finite states (FS) automaton proceed in two stages. On the first stage a deterministic FS automaton is built and on the second stage this automaton is minimized. For an overview on the modern automaton construction and minimization methods see [6, 7]. Those methods have the serious backdraw that the intermediate automaton is huge in respect to the corresponding minimal one.

For many practical applications the construction of large scale acyclic automata is an important task. Methods for construction and minimization of acyclic automata can be found in [3, 4]. Nevertheless, the Revuz' algorithm (which delivers the best efficiency) is also a two stage method and has the above mentioned backdraw. Therefore the use of Revuz' method for construction of very large automata is difficult.

We shall present bellow methods for direct construction of minimal automata where the whole construction is performed *in one stage* and *no intermediate automata are built*. Our methods are based on the method for direct construction of minimal automaton for a given lexicographically sorted list [2]. First we shall introduce the mathematical framework which is presented in more details in [2]. After that we present in details the corresponding algorithm and give correctness and complexity proofs. We proceed with a detailed presentation of the algorithms for direct construction of minimal automaton presenting the union, intersection and difference of acyclic automata. Those algorithms are direct descents of the base algorithm. At the end, we give some experimental comparisons of our algorithm with the corresponding Revuz' algorithm.

2. MATHEMATICAL CONCEPTS AND RESULTS

Definition 1. A deterministic FS automaton is a tuple $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$, where:

- Σ is a finite alphabet;
- S is a finite set of states;
- $s \in S$ is the starting state;
- $F \subseteq S$ is the set of final states;
- $\mu : S \times \Sigma \to S$ is a partial function called the transition function.

The function μ is extended naturally over $S \times \Sigma^*$ by induction:

$$\begin{cases} \mu^*(r, \varepsilon) = r, \\ \mu^*(r, \sigma a) = \begin{cases} \mu(\mu^*(r,\sigma), a), & \text{in case } \mu^*(r,\sigma) \text{ and } \mu(\mu^*(r,\sigma), a) \text{ are defined,} \\ \text{not defined} & \text{otherwise,} \end{cases} \end{cases}$$

where $r \in S$, $\sigma \in \Sigma^*$, $a \in \Sigma$.

We will work with a definition of FS automata with a partial transition function. The only difference from the definition with a total transition function is the absence of the necessity to introduce a dead state (a non-finite state r, for which $\forall a \in \Sigma\ (\mu(r,a) = r)$). Later, we will use $!\mu(r,\sigma)$ to denote that $\mu(r,\sigma)$ is defined, and when writing $\mu^*(r,\sigma) \cong x$, we will mean $!\mu(r,\sigma)\ \&\ \mu(r,\sigma) = x$.

Definition 2. Let $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$ be a deterministic FS automaton. Then the set $L(\mathcal{A}) \subseteq \Sigma^*$, defined as

$$L(\mathcal{A}) = \{\sigma \in \Sigma^* \mid !\mu^*(s,\sigma) \;\&\; \mu^*(s,\sigma) \in F\},$$

is called the language of the automaton $\mathcal{A}$ or the language recognized by $\mathcal{A}$.

Two automata $\mathcal{A}$ and $\mathcal{A}'$ are called equivalent when $L(\mathcal{A}) = L(\mathcal{A}')$. An automaton is called acyclic when $\forall r \in S \; \forall \sigma \in \Sigma^+ \; (\mu^*(r,\sigma) \not\cong r)$. The language of an acyclic FS automaton is finite.

Definition 3. Let $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$ be a deterministic FS automaton.

1. The state $r \in S$ is called reachable from $t \in S$ when $\exists \sigma \in \Sigma^* \; (\mu^*(t,\sigma) \cong r)$.
2. We define the subautomaton starting in $s' \in S$ as:

$$\mathcal{A}|_{s'} = \langle \Sigma, S', s', F \cap S', \mu|_{S' \times \Sigma} \rangle,$$

where $S' = \{r \in S \mid r \text{ is reachable from } s'\}$.

3. Two states $s_1, s_2 \in S$ are called equivalent when $L(\mathcal{A}|_{s_1}) = L(\mathcal{A}|_{s_2})$.

Definition 4. The deterministic FS automaton $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$ with language $L(\mathcal{A})$ is called minimal (with language $L(\mathcal{A})$) when for every other deterministic FS automaton $\mathcal{A}' = \langle \Sigma, S', s', F', \mu' \rangle$ with language $L(\mathcal{A}') = L(\mathcal{A})$ it holds $|S| \leq |S'|$.

From the classical FS theory the following theorem is well-known:

Theorem 5. *A deterministic FS automaton with non-empty language is minimal if and only if every state is reachable from the starting state, from every state a final state is reachable and there are no different equivalent states. There exists an unique (up to isomorphism) minimal automaton for a given language of FS automaton.*

MINIMAL EXCEPT FOR A WORD AUTOMATA

Bellow we will assume that a finite alphabet Σ is given and there is a linear order in Σ. Later, writing lexicographical order of words in Σ^*, we will understand the lexicographical order induced by the linear order of Σ.

Definition 6. Let $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$ be an acyclic deterministic FS automaton with language $L(\mathcal{A})$. Then the automaton $\mathcal{A}$ is called *minimal except for the word* $\omega \in \Sigma^*$ when the following conditions hold:

1. Every state is reachable from the starting state and from every state a final state is reachable.

2. ω is a prefix of the last word in the lexicographical order of $L(\mathcal{A})$.

In that case we can introduce the following notations:

$$\omega = w_1^{\mathcal{A}} w_2^{\mathcal{A}} \dots w_k^{\mathcal{A}}, \text{ where } w_i^{\mathcal{A}} \in \Sigma \text{ for } i = 1, 2, \dots, k, \quad (1)$$

$$t_0^{\mathcal{A}} = s; \quad t_1^{\mathcal{A}} = \mu(t_0^{\mathcal{A}}, w_1^{\mathcal{A}}); \quad t_2^{\mathcal{A}} = \mu(t_1^{\mathcal{A}}, w_2^{\mathcal{A}}); \quad \dots; \quad t_k^{\mathcal{A}} = \mu(t_{k-1}^{\mathcal{A}}, w_k^{\mathcal{A}}), \quad (2)$$

$$T = \{t_0^{\mathcal{A}}, t_1^{\mathcal{A}}, \dots, t_k^{\mathcal{A}}\}. \quad (3)$$

3. In the set $S \setminus T$ there are no different equivalent states.
4. $\forall r \in S \forall i \in \{0, 1, \dots, k\} \forall a \in \Sigma (\mu(r, a) \cong t_i \leftrightarrow (i > 0 \,\&\, r = t_{i-1} \,\&\, a = w_i^{\mathcal{A}}))$.

Bellow, when working with minimal except for a given word automaton, we will use the notations (1)–(3) introduced in the former definition. In case the notation is not ambiguous, we will write t_i, w_i instead of $t_i^{\mathcal{A}}$, $w_i^{\mathcal{A}}$. Clearly, if an automaton is minimal except for two different words, one is a prefix of the other.

Proposition 7. *Let the automaton $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$ be minimal except for ω. Then:*

1. $\forall r \in S \setminus T \; \forall a \in \Sigma \; (!\mu(r, a) \rightarrow \mu(r, a) \in S \setminus T)$.
2. $\mu^*(s, \sigma) \cong t_i \leftrightarrow \sigma = w_1 w_2 \dots w_i$.

Proposition 8. *An automaton which is minimal except for the empty word ε is minimal.*

Lemma 9. *Let the automaton $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$ be minimal except for $\omega = w_1 w_2 \dots w_k$, $\omega \neq \varepsilon$. Let there be no state equivalent to t_k in the set $S \setminus T$. Then $\mathcal{A}$ is also minimal except for the word $w_1 w_2 \dots w_{k-1}$.*

Lemma 10. *Let the automaton $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$ be minimal except for $\omega = w_1 w_2 \dots w_k$, $\omega \neq \varepsilon$. Let the state $p \in S \setminus T$ be equivalent to the state t_k. Then the automaton $\mathcal{A}' = \langle \Sigma, S', s, F', \mu' \rangle$ defined as follows:*

$$S' = S \setminus \{t_k\},$$

$$F' = F \setminus \{t_k\},$$

$$\mu'(r, a) = \begin{cases} \mu(r, a), & \textit{in case } r \neq t_{k-1} \vee a \neq w_k \textit{ and } \mu(r, a) \textit{ is defined,} \\ p, & \textit{in case } r = t_{k-1},\ a = w_k, \\ \textit{not defined} & \textit{otherwise,} \end{cases}$$

is equivalent to the automaton $\mathcal{A}$ and is minimal except for the word $w_1 w_2 \dots w_{k-1}$.

Theorem 11. *Let the automaton $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$ be minimal except for $\omega' = w_1 w_2 \dots w_m$. Let $\psi \in L(\mathcal{A})$ be the last word in the lexicographical order of the language of the automaton. Let ω be a word which is greater in lexicographical order than ψ. Let ω' be the longest common prefix of ψ and ω. In that case we can denote $\omega = w_1 w_2 \dots w_m w_{m+1} \dots w_k; k > m$. Then the automaton $\mathcal{A}' = \langle \Sigma, S', s, F', \mu' \rangle$ defined as follows:*

$$t_{m+1}, t_{m+2}, \dots, t_k \textit{ are new states such that } S \cap \{t_{m+1}, t_{m+2}, \dots, t_k\} = \varnothing,$$

$$S' = S \cup \{t_{m+1}, t_{m+2}, \dots, t_k\},$$

$$F' = F \cup \{t_k\},$$

$$\mu'(r,a) = \begin{cases} t_{i+1}, & \textit{in case } r = t_i,\ m \le i \le k-1,\ a = w_{i+1}, \\ \mu(r,a), & \textit{in case } r \in S \textit{ and } \mu(r,a) \textit{ is defined and} \\ & r \ne t_m \vee a \ne w_{m+1}, \\ \textit{is not defined} & \textit{otherwise,} \end{cases}$$

is minimal except for ω *and recognizes the language* $L(\mathcal{A}) \cup \{\omega\}$.

Lemma 12. *Let the automaton* $\mathcal{A} = \langle \Sigma, S, s, F, \mu \rangle$ *be minimal except for* $\omega = w_1 w_2 \dots w_k$. *Then for* t_k *(refer to the notation introduced in Definition 6) the following statement holds:*

$$t_k \textit{ is equivalent to } r \in S \setminus T \ \leftrightarrow\ \Big((t_k \in F \leftrightarrow r \in F)$$

$$\& \ \forall a \in \Sigma\, ((\neg!\mu(t_k,a)\ \&\ \neg!\mu(r,a)) \vee (!\mu(t_k,a)\ \&\ !\mu(r,a)\ \&\ \mu(t_k,a) = \mu(r,a)))\Big).$$

The proofs of the above results are presented in [2].

3. ON-LINE ALGORITHM FOR BUILDING A MINIMAL FS AUTOMATON FOR A GIVEN LIST

First we describe the method informally and give an example. After that we give the pseudo-code in a Pascal-like language (like the language used in [1]) with correctness proof.

Let a non-empty finite list of words L in lexicographical order be given. Let $\omega^{(i)}$ denote the i-th word of the list. We start with the minimal automaton which recognizes only the first word of the list. This automaton can be built trivially and is also minimal except for $\omega^{(1)}$. Using it as basis, we carry out an induction on the words of the list. Let us assume that the automaton $\mathcal{A}^{(n)} = \langle \Sigma, S, s, F, \mu \rangle$ with language $L^{(n)} = \{\omega^{(i)} \mid i = 1, 2, \dots, n\}$ has been built and that $\mathcal{A}^{(n)}$ is minimal except for $\omega^{(n)}$. We have to build the automaton $\mathcal{A}^{(n+1)}$ with language $L^{(n+1)} = \{\omega^{(i)} \mid i = 1, 2, \dots, n+1\}$ which is minimal except for $\omega^{(n+1)}$.

Let ω' be the longest common prefix of the words $\omega^{(n)}$ and $\omega^{(n+1)}$. Using several times Lemma 9 and Lemma 10 (corresponding to the actual case), we build the automaton $\mathcal{A}' = \langle \Sigma, S', s, F', \mu' \rangle$ which is equivalent to $\mathcal{A}^{(n)}$ and is minimal except for ω'. Now we can use Theorem 11 and build the automaton $\mathcal{A}^{(n+1)}$ with language $L^{(n+1)} = L^{(n)} \cup \{\omega^{(n+1)}\} = \{\omega^{(i)} \mid i = 1, 2, \dots, n+1\}$ which is minimal except for $\omega^{(n+1)}$.

In this way by induction we build the minimal except for the last word of the list automaton with language the list L. At the end, using again Lemma 9 and Lemma 10, we build the automaton equivalent to the former one, which is minimal except for the empty word. From Proposition 8 we have that it is the minimal automaton for the list L. To distinguish efficiently between Lemma 9 and Lemma 10, we can use Lemma 12. □

Let us describe now the algorithm more formally. We will presume that there are given implementations for Abstract Data Types (ADT) representing the automaton state and the dictionary of automaton states. Later, we presume that NULL is the null constant for an arbitrary abstract data type.

On automaton state we shall need the following types and operations:

1. STATE is pointer to a structure representing an automaton state.

2. FIRST_CHAR, LAST_CHAR : char are the first and the last char in the automaton alphabet. We will assume that the chars are sequentially given in lexicographical order.

3. **function** NEW_STATE : STATE returns a new state.

4. **function** FINAL(STATE) : boolean returns true if the state is final and false otherwise.

5. **procedure** SET_FINAL(STATE,boolean) sets the finality of the state to the boolean parameter.

6. **function** TRANSITION(STATE,char): STATE returns the state to which the automaton transits from the parameter state with the parameter char.

7. **procedure** SET_TRANSITION(STATE,char,STATE) sets the transition from the first parameter state by the parameter char to the second parameter state.

8. **procedure** PRINT_AUTOMATON(file,STATE) prints the automaton starting from the parameter state to file.

Having defined the above operations, we make use of the following three functions and procedures:

```
function COPY_STATE (s : STATE) : STATE;
{ copies s to a new state}
var
    r : STATE;
    c : char;
begin
    r := NEW_STATE;
    SET_FINAL(r,FINAL(s));
    for c := FIRST_CHAR to LAST_CHAR do
        SET_TRANSITION(r,c,TRANSITION(s,c));
    return(r);
end; { COPY_STATE}

procedure CLEAR_STATE (s : STATE);
{ clears all transitions of s and sets it to non-final one }
var c : char;
begin
    SET_FINAL(s,false);
    for c := FIRST_CHAR to LAST_CHAR do
        SET_TRANSITION(s,c,NULL);
```

```
end; { CLEAR_STATE}

function COMPARE_STATES (s1,s2 : STATE) : integer;
{ compares two states}
var c : char;
begin
   if FINAL(s1) < FINAL(s2) then return(-1)
   else if FINAL(s1) > FINAL(s2) then return(1);
   for c := FIRST_CHAR to LAST_CHAR do
      if TRANSITION(s1,c) < TRANSITION(s2,c) then return(-1)
      else if TRANSITION(s1,c) > TRANSITION(s2,c) then return(1);
      { here we compare only the pointers }
   return(0);
end; { COMPARE_STATES}
```

The ADT on Dictionary of automaton states uses the COMPARE_STATES function above to compare states. For the dictionary we need the following operations:

1. **function** NEW_DICTIONARY : DICTIONARY returns a new empty dictionary;

2. **function** MEMBER(DICTIONARY,STATE) : STATE returns state in the dictionary equivalent to the parameter state or NULL if not present;

3. **procedure** INSERT(DICTIONARY,STATE) inserts state to dictionary.

Implementations of the above ADTs can be found in [1]. Later we assume that the time complexity of PRINT_AUTOMATON is proportional to the size of the automaton and all other operations on Automaton states including COPY_STATE, CLEAR_STATE and COMPARE_STATES are performed in constant time. Depending on the concrete implementation of the dictionary, we could have different bounds for the time complexity of the operations. Using a typical implementation by, e.g., AVL balanced trees, we will have a logarithmic time complexity for the MEMBER and INSERT operations and the size of the dictionary will be proportional to the number of its elements.

Now we are ready to present the pseudo-code of our algorithm.

Algorithm 1. For on-line construction of minimal automaton presenting the input list of words given in lexicographical order.

```
program Create_Minimal_FS_Automaton_for_Given_List (input, output);
var
   MinimalAutomatonStatesDictionary : DICTIONARY;
   TempStates : array [0..MAX_WORD_SIZE] of STATE;
   InitialState : STATE;
   PreviousWord, CurrentWord : string;
   i, PrefixLengthPlus1 : integer;
```

```
    function FindMinimized ( s : STATE) : STATE;
    { returns an equivalent state from the dictionary; if not present —
      inserts a copy of the parameter state to the dictionary and returns it}
     var r : STATE:
     begin
        r := MEMBER(MinimalAutomatonStatesDictionary,s);
        if r = NULL then begin
           r := COPY_STATE(s);
           INSERT(r);
        end;
        return(r);
     end; { FindMinimized}

begin
    MinimalAutomatonStatesDictionary := NEW_DICTIONARY;
    for i := 0 to MAX_WORD_SIZE do
        TempState[i] := NEW_STATE;
    PreviousWord := '';
    CLEAR_STATE(TempState[0]);

    while not eof(input) do begin
    { loop for the words in the input list}
        readln(input,CurrentWord);

        { the following loop calculates the length of the longest common}
          prefix of CurrentWord and PreviousWord}
        i := 1;
        while (i<length(CurrentWord)) and (i<length(PreviousWord)) and
              (PreviousWord[i] = CurrentWord[i]) do
            i := i+1;
        PrefixLengthPlus1 := i;

        { here we are minimizing the states of the last word }
        for i := length(PreviousWord) downto PrefixLengthPlus1 do
            SET_TRANSITION(TempStates[i-1],PreviousWord[i],
                                    FindMinimized(TempStates[i]));
        { this loop initializes the tail states for the current word}
        for i := PrefixLengthPlus1 to length(CurrentWord) do begin
            CLEAR_STATE(TempStates[i]);
            SET_TRANSITION(TempStates[i-1],CurrentWord[i],
                                    TempStates[i]);
        end;
        SET_FINAL(TempStates[length(CurrentWord)], true);

        PreviousWord := CurrentWord;
    end; { while}
```

```
44      { here we are minimizing the states of the last word}
45      for i := length(CurrentWord) downto 1 do
46          SET_TRANSITION(TempStates[i-1],PreviousWord[i],
                              FindMinimized(TempStates[i]));
47      InitialState := FindMinimized(TempStates[0]);
48      PRINT_AUTOMATON(output,InitialState);
49  end.
```

Now we will prove the correctness and calculate the time and space complexity of the algorithm.

Theorem 13. *Given a lexicographically sorted list of words in the input file, Algorithm* 1 *builds the minimal FS automaton for the list and prints it out on the output file.*

Proof. To prove the theorem, we carry out an induction on the words of the input list.

In lines 20–24 the algorithm initializes the Dictionary of states of the minimal automaton to the empty dictionary and the temporary states. In line 24 *Tempstate*[0] is initialized to a non-final state with no transition. This corresponds to the automaton for the empty language. Line 27 reads the first word from the input. Because of the initialization in line 23, we have that at that moment *PreviousWord* is the empty string. Hence *PrefixLengthPlus1* will be set to 1 in lines 29–32. The loop in lines 33–35 will not be triggered and the loop 36–40 will construct a chain of states for recognizing the first word. In this way the algorithm constructs the minimal automaton for the language consisting of the first word in the input list. Clearly, this automaton is also minimal except for the first word $\omega^{(1)}$. In that moment the automaton is minimal except for $\omega^{(1)}$ and the states $t_0, t_1, \ldots, t_k$ are presented in *Tempstate*. In line 42 the first word is assigned to the string *PreviousWord*.

Now we will show that the loop in lines 25–43 adds the next word from the list to the automaton and produces a minimal except for this word automaton.

Let us assume that on stage j the algorithm has built the automaton recognizing $\{\omega^{(i)} \mid i = 1, 2, \ldots, j-1\}$, which is minimal except for $\omega^{(j-1)}$. The states $t_0, t_1, \ldots, t_k$ are presented in the array *TempStates*, all other states of the automaton are in *MinimalAutomatonStatesDictionary* and *PreviousWord* is $\omega^{(j-1)}$.

Being in line 27, the word $\omega^{(j)}$ is read from the input file into *CurrentWord*. The loop 28–32 calculates the longest common prefix of *PreviousWord* and *CurrentWord* with values at that moment $\omega^{(j-1)}$ and $\omega^{(j)}$. We will show that the loop 33–35 builds the equivalent automaton minimal except for the longest common prefix of $\omega^{(j-1)}$ and $\omega^{(j)}$. In downward order the transition to the state t_i is replaced by a transition to the state which returns the function *FindMinimized*, where i varies from the length of *PreviosWord* to *PrefixLengthPlus1* in reverse order. The function *FindMinimized* searches a state equivalent to the argument in *MinimalAutomatonStatesDictionary*. Here the conditions of Lemma 12 are fulfilled, therefore the use of COMPARE_STATE in the function MEMBER will identify the equivalent to t_i state. If such a state exists, it is returned as result. This corresponds to the condition of Lemma 10. In the other case, the function copies the state and inserts it into *MinimalAutomatonStatesDictionary.* The copy of the state is returned as result. This corresponds to Lemma 9. According to those lemmata, in both cases the new automaton will be equivalent to the former and minimal except for the shorter prefix. After finishing the loop, we have an automaton recognizing $\{\omega^{(i)} \mid i = 1, 2, \ldots, j-1\}$, which is minimal except for the longest common prefix of $\omega^{(j-1)}$ and $\omega^{(j)}$.

The loop 36–40 simply constructs a tail of states in the array *TempStates* in order to recognize *CurrentWord.* In line 41 the last state is marked as final. This corresponds exactly to the conditions of Theorem 11. Therefore we have built the automaton for the language $\{\omega^{(i)} \mid i = 1, 2, \ldots, j\}$ minimal except for the word $\omega^{(j)}$. After assigning $\omega^{(j)}$ to *PreviousWord* in line 42, we are closing the main loop.

From the induction we have that after finishing the loop 25–43 the algorithm will build the automaton for the input list which is minimal except for the last word. The lines 44–47 in the same way as the loop 33–35 build the equivalent automaton which is minimal except for ε. From Proposition 8 we have that this is the minimal automaton for the list. Line 48 prints the automaton on the output file. □

Theorem 14. *Algorithm* 1 *builds the minimal automaton for a given alphabetically sorted list of words in $O(n \log(m))$ time, where n is the total number of letters in the input list and m is the size (number of states) of the resulting minimal automaton. The space complexity of Algorithm* 1 *is $O(m)$.*

Proof. For each letter from the input list the algorithm passes either through line 31 or through lines 38–39. Each of the statements of those lines are performed in constant time. In case we have passed through the lines 38–39, we later have to pass through line 35 or 46. The time complexity of the lines 35, 46 depends on the time complexity of *FindMinimized.* By using balanced tree implementation of the dictionary we have that the complexity of *FindMinimized* is logarithm of the size of the dictionary. The dictionary has at most m elements, where m is the number of the states of the minimal automaton for the list. Hence the time complexity of the whole algorithm is $O(n \log(m))$.

Clearly, the space needed by the algorithm is equal to the size of the dictionary of states of the minimal automaton plus the size of the *TempStates* array plus the constant size of the other fixed-size variables. *TempStates* is proportional to the size of the longest word in the list and in the case of using balanced tree implementation, the size of the dictionary of states of minimal automaton is proportional to the number of states of the minimal automaton. Clearly, the size of the longest word in the list is lower than the size of the minimal automaton representing this list. Therefore the space complexity is $O(m)$. □

The main advantage of our method is the excellent space to time proportion.

4. ALGORITHMS FOR DIRECT CONSTRUCTION OF MINIMAL AUTOMATON PRESENTING UNION, INTERSECTION AND DIFFERENCE OF ACYCLIC AUTOMATA

The standard methods for construction of automaton presenting union, intersection and difference are building first a temporary automaton which states are Cartesian product of states of the input automata. This temporary automaton in general is huge with respect to the resulting minimal automaton. Here we will present a new method for direct constructing the minimal automaton which drastically improves the efficiency.

By traversing an acyclic deterministic FS automaton in depth first by choosing the transitions in lexicographical order we can produce the automaton language in lexicographical order. Using this property, we can produce the lexicographical ordered list which is union, intersection or difference of the languages of the input automata. Using this list as input for Algorithm 1, we can construct directly the minimal automaton for the union, intersection and difference. Moreover, we do not have to build explicitely the whole lists in the memory. We can proceed word by word using only the top words of the lists. Bellow we give the formal description of our algorithm.

We will need the following declaration:

type States_Stack = **array** [1..MAX_WORD_SIZE+1] of STATE — type array of automaton states.

We will use array of states for representing automaton path. If we have a word w: string, we will have $S[i+1] = \mathrm{TRANSITION}(S[i], w[i])$, $i = 1, 2, \ldots, \mathrm{lenght}(w)$, where $S[0]$ is the initial automaton state.

For producing the language of an automaton word by word, we will use a function which for a given word and corresponding path returns the next word in lexicographical order in automaton language.

Algorithm 2. Given a word and a corresponding automaton, path returns the next word in lexicographical order in the automaton language.

We will assume that from any automaton state a final state is reachable.

```
function NEXT_AUTOM_WORD($S$ : States_Stack; var $w$ : string) : boolean;
var
  $c$ : char;
  $sp$ : integer;

  function FIND_FORWARD_WORD : boolean;
  begin
    $c$ := FIRST_CHAR;
    while ($c$<=LAST_CHAR) and (TRANSITION($S[sp], c$) = NULL)
        do $c$ := succ($c$);
    if c>LAST_CHAR then return(false);
    $S[sp+1]$ := TRANSITION($S[sp], c$);
```

```
        $sp := sp + 1$;
        $w :=$ concat($w, c$);
        **while not** FINAL($S[sp]$) **do begin**
            $c :=$ FIRST_CHAR;
            **while** TRANSITION($S[sp], c$) = NULL **do** $c :=$succ($c$);
            $S[sp + 1] :=$ TRANSITION($S[sp], c$);
            $sp := sp + 1$;
            $w :=$ concat($w, c$);
        **end**;
        **return**(true);
    **end**;

**begin**
    $sp :=$ length($w$);
    **if** FIND_FORWARD_WORD **then return**(true);
    **repeat**
        **if** $sp = 1$ **then return**(false);
        $sp := sp - 1$;
        $c := w$[length($w$)];
        delete($w$, length($w$), 1);
        **while** ($c <=$ LAST_CHAR) **and** (TRANSITION($S[sp], c$) = NULL)
            **do** $c :=$ succ($c$);
    **until** $c <=$ LAST_CHAR;
    $S[sp + 1] :=$ TRANSITION($S[sp], c$);
    $sp := sp + 1$;
    $w :=$ concat($w, c$);
    **if not** FINAL($S[sp]$)
        **then return**(FIND_FORWARD_WORD) **else return**(true);
**end**;
```

Theorem 15. *For a given word and a corresponding automaton, path Algorithm 2 finds the next word in lexicographical order in the automaton language and returns true or returns false in case there are no more words.*

We can proof the above theorem by induction on the words in the automaton language.

We will use the above function for producing the lexicographically sorted list representing the union, intersection and difference of automaton languages.

Algorithm 3. For producing the next word in lexicographical order of the union of two acyclic automaton languages.

We shall need the following global variables:

```
var
    p1, p2, f1, f2 : boolean;
    s1, s2 : States_Stack;
    w1, w2 : string;
```

We shall assume that they are initialized by the following procedure:

```
procedure INIT_NEXT_WORD;
begin
   s1[1] := init_state1;
   s2[1] := init_state2;
   w1 := '';
   w2 := '';
   p1 := true;
   p2 := true;
   f1 := true;
   f2 := true;
end;
```

init_state1, *init_state2* are the initial states of the two automata. In that case the function NEXT_WORD produces in the variable *w* the next word in lexicographical order of the union list or returns false in case there are no more words.

```
function NEXT_WORD(var w : string) : boolean;
begin
   if not f1 and not f2 then return(false);
   if p1 then f1 := NEXT_AUTOM_WORD(s1,w1);
   if p2 then f2 := NEXT_AUTOM_WORD(s2,w2);
   if not f1 and not f2 then
      return(false)
   else if (f1 and f2) and (w1 = w2) then begin
      w := w1;
      p1 := true;
      p2 := true;
   end else if not f1 or ((f1 and f2) and (w1 > w2)) then begin
      w := w2;
      p1 := false;
      p2 := true;
   end else if not f2 or ((f1 and f2) and (w1 < w2)) then begin
      w := w1;
      p1 := true;
      p2 := false;
   end;
   return(true);
end;
```

The function NEXT_WORD proceeds as follows: in the variables *p1*, *p2* we mark the necessity for reading the next word from the corresponding automaton. In the variables *f1*, *f2* we mark the ending of the corresponding automaton. In case both automata are traversed, the function returns false. In the other case the two current words from the automata lists are compared. In case the words are equal, we return one of them and mark in *p1*, *p2* that on the next call of the function the words from both lists have to be read. In case one of the words preceeds the other, we return that word and mark the corresponding automaton in order to read the next word from it. In casè one of the automaton languages has finished, the other is listed until both are finished.

We have to note that the function NEXT_WORD returns word by word the list of the words in the union of the two input automata without using any extra memory for generating the lists. The time for listing the words in the union is proportional to the sum of the lengths (in letters) of the languages of the input automata. This follows from the next facts. To list the words, in the union the paths are traversing from the initial to the final states in the input automata. The sum of all those paths in an automaton is equal to the number of all letters in the automaton language. □

For producing the list of the intersection or difference of two automata, we proceed similar to the method above. But we shall present a more efficient method which is applicable also in case the second automaton is not acyclic.

First we present an additional function which returns true in case the word is recognized by the automaton and false otherwise.

```
function RECOGNIZE_WORD ($w$: string; $s$: STATE) : boolean;
var $i$ : integer;
begin
    $i := 1$;
    while $i <= \text{length}(w)$ do begin
        if TRANSITION($s, w[i]$) = NULL then return(false);
        $s :=$ TRANSITION($s, w[i]$);
        $i := i + 1$;
    end;
    return(FINAL($s$));
end;
```

The functionality of the above function is clear. We have only to note that the recognition time for a word is proportional to the length of the word.

Algorithm 4. For producing the next word in lexicographical order of the intersection of an acyclic deterministic FS automaton with a deterministic FS automaton language.

We shall need the following global variables:

```
var
    f1 : boolean;
```

```
        s1 : States_Stack;
        w1 : string;
```

We shall assume that they are initialized by the following procedure:

```
procedure INIT_NEXT_WORD;
begin
   s1[1] := init_state1;
   w1 := '';
   f1 := true;
end;
```

init_state1, *init_state2* are the initial states of the two automata. In that case the function NEXT_WORD produces in the variable w the next word in lexicographical order of the intersection list or returns false in case there are no more words.

```
function NEXT_WORD(var w: string) : boolean;
begin
   if not f1 then return(false);
   repeat
      f1 := NEXT_AUTOM_WORD(s1, w1);
   until not f1 or RECOGNIZE_WORD(w1, init_state2);
   if not f1 then return(false);
   w := w1;
   return(true);
end;
```

In that case the function NEXT_WORD proceeds as follows: word by word the first automaton language is listed in lexicographical order. In case the current word is recognized by the second automaton, this word is returned as the next word in the intersection.

Here we have to note that the function NEXT_WORD produces the intersection list word by word without using extra memory for generating the whole lists. The time for producing the intersection list is obviously proportional to the number of all letters in the first automaton. □

For deriving an algorithm producing the next word from the difference of an acyclic deterministic FS automaton with a deterministic FS automaton, we have to make in the above algorithm the following change:

```
        until not f1 or RECOGNIZE_WORD(w1, init_state2);
```

have to be exchanged with

```
        until not f1 or not RECOGNIZE_WORD(w1, init_state2);
```

There will be almost no changes in the functionality of the algorithm and the time complexity for producing the difference list will be again proportional to the number of all letters in the first automaton language.

Let us present now the algorithm for direct construction of minimal automaton.

Algorithm 5. For direct construction of minimal automaton presenting the union, intersection or difference of acyclic automaton languages. (In case of intersection and difference, only the first automaton has to be acyclic.)

We shall use as base Algorithm 1. We assume that the global variables *init_state1*, *init_state2* are given, which represent the initial states of the first and second automaton. We shall assume also that the global variables of Algorithm 3 or Algorithm 4 and the corresponding functions NEXT_AUTOM_WORD, NEXT_WORD, INIT_NEXT_WORD, RECOGNIZE_WORD are defined. We need further the following changes of Algorithm 1:

1. Between lines 24 and 25 we have to call the initialization procedure

    INIT_NEXT_WORD;

2. Line 25 has to be changed to

    while NEXT_WORD(*CurrentWord*) **do begin**

3. Line 27 has to be deleted.

The only difference between the above algorithm and Algorithm 1 is the use of an input list which presents the union, intersection or difference of the input automata. We derive the following complexity results. For the union the time complexity of Algorithm 5 is $O((n_1+n_2)+n\log(m))$, where n_1, n_2 are the number of letters of the two input automata languages, n is the number of letters in the union language and m is the size (number of states) of the resulting minimal automaton. We obviously have that $n_1+n_2 < 2n$, hence the time complexity is $O(n\log(m))$. For the intersection and difference we have that the time complexity is $O(n_1+n\log(m))$. The memory complexity in all cases of Algorithm 5 is $O(m)$. □

5. IMPLEMENTATION RESULTS AND COMPARISONS

We have implemented various tools for constructing, updating and processing of lexicons presented as minimal automaton. They are programmed in GNU-C and JAVA. For a more efficient implementation we have used an open hash structure for the lexicon of automaton states presentation. This provides a nearly linear time complexity for practical applications.

We have experimented with grammatical lexicons for Bulgarian and Russian[1] languages. The middle-sized lexicon for Bulgarian common lexica has about 500000 wordforms and the Russian one — about 1500000 wordforms. They are encoded according to the DELAF format [5]. To provide additional grammatical information to the words, we have used a FS automaton with labels on the final states. A trivial change is needed to modify Algorithm 1 to build minimal automata with such labels. In the INTEX system [5] there is a similar tool for building the same kind of FS

[1] The DELAF format of the Russian lexicon is build in cooperation with the Computer Fond of the Russian Language in Moscow.

Table 1. Comparison of our and the INTEX tool for building minimal automata

		INTEX tool	our tool
Bulgarian lexicon	Size (Wordforms)	524473	
	Memory used	33450 KB	1660 KB
	Time needed	2:07 min	0:29 min
Russian lexicon	Size (Wordforms)	1486552	
	Memory used	129000 KB	4400 KB
	Time needed	17 min	2:04 min

automata. This tool is a highly efficient implementation of the Revuz' algorithm. Table 1 shows a comparison between our tool and the corresponding INTEX tool.

All time and memory parameters given in the paper are measured on a 32MB RAM Pentium 180 machine running under NEXTSTEP. The large time requirements of the INTEX tool for the Russian lexicon are explained with the heavy usage of virtual memory. On a small lexicon (when the trie structure for the INTEX tool fits into the operating memory) our tool is only slightly faster than the INTEX one.

6. CONCLUSION

The presented methods and algorithms are successfully used for construction and operations on large scale dictionaries. They are distinguished with significantly better memory efficiency than the others.

An open question is the existence of a method for direct construction of minimal automaton presenting the concatenation of acyclic automaton languages. There seems to be a problem producing the concatenation list lexicographically sorted.

Acknowledgements. The author would like to thank Professor Klaus Schulz, Professor Dimiter Skordev and Anton Zinoviev for the very valuable remarks on the presentation.

REFERENCES

1. Aho, A., J. Hopcroft, J. Ullman. Data Structures and Algorithms. Addison-Wesley, Reading, Massachutes, 1983.
2. Mihov, S. Direct Building of Minimal Automaton for Given List. *Ann. Sof. Univ.*, Fac. Math. et Inf., **91**, 1, 1999, 33–40.
3. Revuz, D. Dictionaires et lexiques, méthodes et algorithmes. Ph.D. dissertation, Université Paris 7, Paris, 1991.
4. Revuz, D. Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science*, **92**, 1, 1992.
5. Silberztein, M. Dictionnaires électroniques et analyse automatique de textes: le système INTEX. Masson, Paris, 1993.

6. Watson, B. W. A Taxonomy of Finite Automata Construction Algorithms. Computing Science Report 93/43, Fac. of Math. and Comp. Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1993.
7. Watson, B. W. A Taxonomy of Deterministic Finite Automata Minimization Algorithms. Computing Science Report 93/44, Fac. of Math. and Comp. Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1993.

Received November 13, 1998

Linguistic Modelling Laboratory
LPDP — Bulgarian Academy of Sciences
E-mail: **stoyan@lml.acad.bg**