Compression of minimal acyclic deterministic FSAs preserving the linear accepting complexity

Kalin Georgiev
Faculty of Mathematics and Informatics, Sofia University
5 Bourchier str.
Sofia, 1164
kalin.georgiev@fmi.uni-sofia.bg

Abstract

In this paper we investigate the possibility to take advantage of the isomorphic sub graphs that can be found in the structure of minimal FSAs in order to achieve smaller volumes of their memory representations. We introduce a new abstract machine – *Recursive Automaton* – similar to the Recursive Transition Networks, which we utilize to accept the language of the initial FSA while some of the largest isomorphic sub graphs of that FSA are *merged* in a single sub graph. We also investigate the properties of the new machine, the languages it accepts and the formal grammar equivalent to that machine. We've shown that the class of languages accepted by recursive automatons is strictly between LL(1) and LR(1).

Keywords

Finite State Automatons, Formal Grammars, Compression

1. Introduction

If we consider the minimal acyclic deterministic FSAs as directed graphs, in the majority of cases we can observe a significant number of isomorphic sub graphs. In this work we present a new formal machine named *Recursive Automaton*, similar to the FSAs. The additional properties of RAs allow a class of isomorphic sub graphs to be "merged" in a single representative of the class. The new property of recursive automatons are the *recursive transitions* which are used to make a "call" to a sub automaton.

The properties of the recursive automatons are investigated. A class of formal grammars equivalent to the recursive automatons is characterized. The class of languages which are accepted by recursive automatons is shown to be strictly larger than LL(1) and strictly smaller than LR(1).

The presented compression algorithm has two major aspects. First, we identify isomorphic sub graphs of the initial FSA. We define a relation *mergeability* which sets the conditions under which two isomorphic sub graphs can be merged. We apply a greedy algorithm to construct a set of classes of mergeable sub graphs. Second, we apply a merging algorithm, which transforms the initial FSA into a recursive automaton, accepting the same language, but with a smaller number of states and transitions. The time complexity of accepting a word by a recursive automaton is linear amortized.

The experimental results show up to 8% reduction of the number of states and transitions for FSAs, built to accept natural language dictionaries, and up to 70% reduction for FSAa accepting bibliographic dictionaries.

This paper is a summary of the author's master's thesis. Most of the major results are cited here but proofs are not provided.

2. Recursive Automatons

2.1 Definition

In this section we define the *Recursive Automaton* and its execution over an input word.

Definition. By *Recursive Automaton* we denote $A = \langle \Sigma, Q, q_0, \delta, r, F \rangle$, where Σ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ and $r : Q \to Q$ are partial functions, and $F \subseteq Q$. □

The new property $r : Q \to Q$ defines the *recursive transitions* in A. Recursive transitions are applied when a *call* needs to be done to a *sub automaton* of A. As in a computer program, after a call is *completed* then the execution of A returns to the *caller* state. To define strictly the operation of this machine we'll introduce

Definition. Let A be a recursive automaton and $F = \{ f \mid f : Q^+ \to Q^+ \}$ be the set of partial functions in Q^+. By Γ_A we denote the operator $\Gamma_A : F \to F$ defined by:

$$\Gamma_A(f)(q_k \ldots q_0) = \begin{cases} q_k \ldots q_0 & \text{if } \neg\exists! r(q_k) \\ f(r(q_k) q_k \ldots q_0) & \text{otherwise} \end{cases}.$$ □

Lemma. Γ_A is continuous.

This is easily proven by showing that Γ_A is monotonous and finite.□

A word from Q^+ can be considered as the *current stack* of the recursive automaton. The smallest fixed point C_A of Γ_A is the "stack winder" of the recursive automaton. It traverses the longest path of recursive transitions starting from the top of the stack and pushes all the states it passes. Let

$$G_r(A) = \langle Q, \{(q,q') \mid !r(q) \wedge r(q) = q'\} \rangle$$

be the graph of recursive transitions in A. We can characterize the domain of C_A.

Lemma. $!C_A(\omega)$ for every $\omega = \omega_0 ... \omega_k \in Q^+$ such that ω_0 can not be found in any cycle in $G_r(A)$. On the other hand, we have $\neg!C_A(\omega)$ for every ω_0 which is a part of a cycle. □

Finally, we re ready to define the operation of A.

Definition. Let $\Delta : Q^+ \times \Sigma \to Q^+$ be defined by:

$$\Delta(q_k ... q_0, c) = \begin{cases} R(C(q' q_{k-1} ... q_0)) & \text{if } \delta(q_k, c) \simeq q' \\ \neg! & \text{otherwise} \end{cases}$$

where

$$R(q_k ... q_0) = \begin{cases} q_k ... q_0 & \text{if } \exists c(!\delta(q_k, c)) \vee k = 0 \\ R(q_{k-1} .. q_0) & \text{otherwise} \end{cases}$$

and C is the smallest fixed point of Γ_A.□

The function *R* is the antipode of C_A. It "unwinds" the stack until a state is reached that actually accepts symbols from the input alphabet.

Definition. Let $\Delta^* : Q^+ \times \Sigma^+ \to Q^+$ be defined by:

$$\Delta^*(s, a_0 ... a_n) = \begin{cases} \Delta(s, a_0) & n = 0 \\ \Delta^*(\Delta(s, a_0), a_1 .. a_n) & ow \end{cases}$$

Then we say that A accepts $\omega \in \Sigma^+$ iff $!\Delta^*(q_0, \omega)$ and $\Delta^*(q_0, \omega) = q, q \in F$.□

An execution of A over a word $\omega = a_1 .. a_k \in \Sigma^+$ is the sequence $s_0, .., s_k$ of words from Q^+, where $s_0 = q_0$ and

$$\forall 0 \le i < k \left(!\Delta(s_i, a_{i+1}) \wedge s_{i+1} = \Delta(s_i, a_{i+1})\right).$$

A successful execution is every execution for which s_k is a single character from F. An alternative and equivalent definition of accepting would imply that there exists a successful execution over ω. The language of A is $L(A) = \{\omega \in \Sigma^+ \mid \Delta^*(q_0, \omega) = q, q \in F\}$.

To demonstrate the operation of a recursive automaton, let us consider Figure 1.

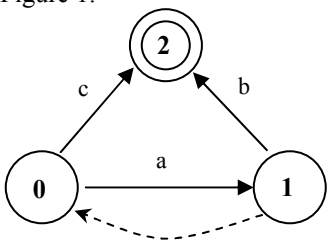

Figure 1.

Figure 1 shows a recursive automaton A. Its initial state is 0, 2 is its single final state, the solid lines indicate "ordinary" transitions and the dashed line indicates the single recursive transition in A. The execution of A over "*aaacbbb*" is the sequence: "*0*", "*10*", "*110*", "*1110*", "*111*","*11*", "*1*", "*2*". Before "*c*" comes at the input we have the configuration "*1110*". The next configuration is $\Delta(1110, c) = R(C(1112))$. But $C(1112) = 1112$ since there are no recursive transitions going out of *2*. The state *2* has no ordinary transitions either, so $R(1112) = 111$. We can see that the accepting complexity is amortized linear, where the amortization comes from the additional steps that sometimes *C* and *R* have to perform.

The language of A on Figure 1 is $\{a^i c b^i \mid i \in N^+\}$.

2.2 Normal recursive automatons

In this section we will introduce some restrictions to the recursive automatons which will not affect the class of accepted languages but will make further investigations easier.

There are three kinds of states which we will forbid. The first kind are the states q such that $!r(q) \wedge \forall c(\neg!\delta(q, c))$. There are recursive transitions going out of these states but no ordinary transitions. Such states will never be left on the top of the stack by *R* (unless they are the only state in the stack), so they can't affect the execution essentially. The second kind of states are the states q such that $\exists q'(r(q') = q) \wedge \forall c(\neg!\delta(q, c))$. These states are pointed by recursive transitions but again do not have ordinary transitions going out of them. They are also redundant. And finally, the states q such that they do not participate in any execution over any word, or

$\neg\left(\exists\omega\in\Sigma^*\right)\left(\Delta^*\left(q_0,\omega\right)=a_1..a_k q b_1..b_l, k,l\geq 0\right)$, will also be forbidden.

Definition. By *normal* recursive automaton we will denote any recursive automaton which has not forbidden states.□

Theorem. For every recursive automaton *A* there exists a normal recursive automaton *A'* such that $L\left(A\right)=L\left(A'\right)$.□

Although straightforward, the proof of this theorem is rather long and tedious so we will spare it in this paper, as we will do with most of the proofs.

Further in all sections we will assume that all recursive automatons are normal.

3. RA Grammars

In this section we will introduce a class of formal grammars which we call RA grammars.

Definition. A formal grammar $\Gamma=\left\langle N,T,S,P\right\rangle$ is called an *RA Grammar* if: **(1)** Every rule $\gamma\in P$ has either the form $A\to a$ or $A\to a\omega$ where $a\in T$ and $\omega\in N^+$; **(2)** For every $A\in N$ and $a\in T$ there are no more than two rules of the form $A\to a\omega$ in P, where $\omega\in N^*$; **(3)** If there are two rules $A\to a\omega$ and $A\to a\omega'$ in P, then $\omega'=\omega B$, $B\in N$ and for every left sentential form $\beta_1...\beta_m$ of Γ we have $\forall j\left(1\leq j<m\right)\left(\beta_j\neq A\right)$, i.e. A is never a right-most non-terminal symbol in the left sentential forms of Γ. □

Definition. We say that a recursive automaton A and an RA grammar Γ are *equivalent*, denoted by $eq\left(A,\Gamma\right)$, if Γ can be presented in the form $\Gamma=\left\langle N\cup\overline{N},\Sigma,A_{q_0},P\cup\overline{P}\cup P_{rf}\right\rangle$, where

$$N=\left\{A_q\mid q\in Q\right\},$$

$$\overline{N}=\left\{\overline{A}_q\mid q\in Q\right\},$$

$$P=\left\{A_q\to a\overline{A}_{q_1}...\overline{A}_{q_{m-1}}A_{q_m}\mid\delta\left(q,a\right)=q'\wedge C\left(q'\right)=q_1..q_m\right\}$$

$$\cup\left\{\overline{A}_q\to a\mid\delta\left(q,a\right)=q'\wedge\forall c\neg!\delta\left(q',c\right)\wedge\neg!r\left(q'\right)\right\}$$

$$P_{rf}=\left\{A\to\alpha_1...\alpha_{k-1}\mid A\to\alpha_1..\alpha_{k-1}A_q\in P\wedge q\in F\wedge!r\left(q\right)\right\}$$

□

Lemma. For every A and Γ, $eq\left(A,\Gamma\right)$ leads to $L\left(A\right)=L\left(\Gamma\right)$.□

Lemma. For every recursive automaton A there exists an RA grammar Γ such that $eq\left(A,\Gamma\right)$.□

Lemma. For every RA grammar Γ there exists a recursive automaton A such that $eq\left(A,\Gamma'\right)$, where Γ' is equivalent to Γ.□

All these lead to

Theorem. The class of languages generated by RA grammars is the same as the class of languages accepted by recursive automatons. We call these languages RA languages.□

For example, consider the automaton on Figure 2.

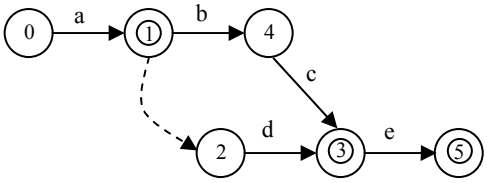

Figure 2.

The language accepted by this automaton is: $\left\{ade(1),adebc(3),adebce(5)\right\}$ (the number indicates the final state where the execution ends). The RA grammar which is equivalent to that automaton consists of the rules: $A_0\to a\overline{A}_2A_1$, $A_0\to a\overline{A}_2$, $\overline{A}_2\to d\overline{A}_3$, $\overline{A}_3\to e$, $A_1\to bA_4$, $A_4\to c$, $A_4\to cA_3$, $A_3\to e$ (some redundant rules are omitted).

For the automaton on Figure 1 we have the rules: $S\to aSA$, $S\to c$, $A\to b$.

We can continue with a classification of the RA grammars.

Lemma. Every RA language is an LR(0) language. □

Lemma. Every LL(1) language is an RA language. □

4. Detaching a sub automaton

The recursive transitions enable states to indicate that a "call" must be made to another state, remembering the caller state. The process is analogous to subroutines of a computer program calling each other. The analogy to a computer program can be extended further, to reveal the idea of merging sub automatons.

Imagine a computer program being a single routine which consists of a sequence of operators. Let there be two non overlapping segments of the program which are syntactically and semantically equivalent. If we create a new subroutine with the same effect as these segments then we may replace both segments with a call to that subroutine. In most cases this will reduce the length of the program while adding insignificant time for its execution. We will apply exactly the same approach to the recursive automatons.

We consider every subset of the set of states Q of a given recursive automaton A as a "sub automaton" of A. Unlike a computer program where the sequence of operators being moved to a separate function could be simply deleted and replaced by a function call, "moving" a sub automaton P to a separate location needs additional work. These operations adjust the "boundary" states of P – the states that are outside P but link to P by recursive or by ordinary transitions. In a computer program, the boundary operator of a program segment is single – the operator just before the segment, while in an automaton there might be a large number of boundary states.

We can see from the definition of R that when a call is made in a recursive automaton, the execution "returns" to the caller when a state with no outgoing ordinary transitions is reached. We will call such states "returning states". With the intuition about what merging of sub automatons is and why detaching is necessary we carry on with the strict definitions of the process.

Definition. By *end* of the sub automaton P of the recursive automaton $A=\langle \Sigma, Q, q_0, \delta, r, F\rangle$ we will denote the set

$$End_P = \left\{ q \mid \exists c\left(\delta_p\left(q,c\right)=q^{'}\right), q^{'} \notin P \right\}.$$

and by *returning states* of P the set

$$R_P = \left\{ q \in P \mid \forall c \neg ! \delta_p\left(q,c\right) \right\}.$$

Here δ_p is the restriction of δ over $P \times \Sigma$. Finally $Exit_P = E_P \bigcup R_P$. □

Definition. A sub automaton P of a recursive automaton is called *detachable* if $Exit_P = \{q_e\}$ is a singleton, $\forall c\left(\delta\left(q_e,c\right) \notin P\right)$, $\forall q \in Q\left(r\left(q\right) \neq q_e\right)$, $\forall q \notin P \forall c\left(\delta\left(q,c\right) \neq q_e\right)$, $F \bigcap P \subseteq Exit_P$, $P - Exit_P \neq \varnothing$ and $q_0 \notin P$. □

Detaching a sub automaton P for which q_e is *not* returning is a technically more complicated procedure than the case when q_e is returning. The idea is demonstrated on Figure 3a and 3b.

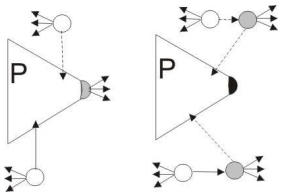

Figure 3a and 3b.

Figure 3a displays a sub automaton P with a non-returning exit state which is being pointed by two boundary states. The transformed P is shown on figure 3b, where a new state is added for each boundary state. The new states copy the transitions outgoing from the exit state, while these transitions are removed from the exit state. Finally, the new states point to P with a recursive transition.

The purpose of these operations is to separate P from the outer world by surrounding it only with recursive transitions, while the transitions going out of P are removed. This makes the *moving* of P possible. Formally

Definition. Let P be a detachable sub automaton of A such that $End_P = \{e_P\}$. By *termination of P* we will denote the operation

$$Term\left(A,P\right) = \left\langle \Sigma, Q, q_s, \delta', r, F \right\rangle,$$

where $\delta' = \delta \setminus \left\{ \left(q_e, c, q'\right) \in \delta \right\}$. □

Definition. Let P be a detachable sub automaton of A such that $End_P = \{e_P\}$, $\exists q, q_-, c \;\; q \in Q \setminus P$, $q_- \in P$, $\delta\left(q,c\right) = q_-$. If $q_1, ..., q_k$ are all states outside P such that $\exists c\left(\delta\left(q_i,c\right)=q_-\right)$, $i=1,...,k$, by $E-\delta$ *-division* of q_- we will denote the operation

$$s_e^{\delta}\left(A,P,q_-\right) = \left\langle \Sigma, Q \bigcup \{q_n\}, q_s, \delta', r', F \right\rangle,$$

where $q_n \notin Q$,

$$\delta' = \delta \setminus \left\{ \left(q_i, c, q_-\right) \in \delta \mid i = 1,...,k \right\} \bigcup$$
$$\left\{ \left(q_i, c, q_n\right) \mid \left(q_i, c, q_-\right) \in \delta, i = 1,...,k \right\}$$
$$\bigcup \left\{ \left(q_n, c, q\right) \mid \left(q_O, c, q\right) \in \delta \right\},$$
$$r' = r \bigcup \left\{ \left(q_n, q_-\right) \right\},$$
$$F = F \bigcup \begin{cases} \varnothing & q_e \notin F \\ q_n & q_e \in F \end{cases}. \square$$

Definition. Let P be a detachable sub automaton of A such that $End_P = \{e_P\}$ and $q_1, .., q_k$ are all different states such that $q_i \in P$, $\exists c, q' \notin P\left(\delta\left(q', c\right) = q_i\right)$ for every $i = 1, ..., k$. Let $A_0, A_1, ..., A_k$ be a sequence of recursive automatons such that $A = A_0$, and $\forall i = 1, ..., k$ $A_i = s_e^{\delta}\left(A_{i-1}, P, q_i\right)$. Then $S_e^{\delta}\left(P, A\right) = A_k$ is the δ-division of P. □

Definition. Let P be a detachable sub automaton of A such that $End_P = \{e_P\}$, $\exists q, q_-, c \; q_- \in P$, $r\left(q\right) = q_-$. If $q_1, ..., q_k$ are all different states such that $r\left(q_i\right) = q_-$ for $i = 1, ..., k$, then $E - r$-division of P is the operation $s_e^r\left(A, P, q_-\right) = \left\langle \Sigma, Q \bigcup \{q_n\}, q_0, \delta', r', F \right\rangle$, where

$q_n \notin Q$, $\delta' = \delta \cup \{(q_n, c, q) \mid (q_e, c, q) \in \delta\}$, $r' = r \setminus \{(q_i, q_-) \mid i = 1,...,k\} \cup \{(q_i, q_n) \mid i = 1,...,k\} \cup \{(q_n, q_-)\}$.□

Definition. Let *P* be a detachable sub automaton of *A* such that $End_P = \{e_P\}$ and $q_1,..,q_k$ are all different states such that $\exists q(r(q) = q_i)$ for every $i = 1,...,l$. Let $A_0, A_1,..., A_k$ be a sequence of recursive automatons such that $A = A_0$, and $\forall i = 1,...,k \quad A_i = s_e^r(A_{i-1}, P, q_i)$. Then $S_e^r(P, A) = Term(A_l)$.□

Finally $S_e(P, A) = Term(S_e^\delta(P, S_e^r(P, A)))$.□

Lemma. For any detachable *P*, such that $End_P = \{e_P\}$, $S_e(P, A)$ is a single normal recursive automaton.□

Lemma. For any detachable *P*, such that $End_P = \{e_P\}, L(A) = L(S_e(P, A))$.□

In the case where $Exit_P = R_P = \{e_P\}$, we may perform all the defined operations, but some of them will have no effect on the recursive automaton.

Lemma. For any detachable *P*, such that $R_P = \{e_P\}$, $L(A) = S_e^\delta(P, A)$.□

Finally, to wrap the two cases together

Definition. Let *P* be a detachable sub automaton of *A*. Then $S(P, A)$ is defined as

(1) $S_e^\delta(P, A)$ if $Exit_P \equiv R_P$

(2) $S_e(P, A)$ if $Exit_P \equiv O_P$.□

Theorem. For any detachable *P*

$$L(A) = L(S(P, A)).□$$

5. Merging sub automatons

Being able to isolate a sub automaton, we are ready to change its "location".

Definition. Let *P* be a detachable sub automaton of *A*. Let $\langle \Sigma, Q, q_0, \delta, r, F \rangle = \overline{A} = S(P, A)$ and $\gamma : P \to P'$ is a bijection, where P' is an arbitrary set. Then $A' = \langle \Sigma, Q' = (Q \setminus P) \cup P', q_0, \delta', r', F \setminus P \rangle$, where

$$\delta \mid \Sigma \times (Q \setminus P) \equiv \delta' \mid \Sigma \times (Q \setminus P),$$
$$\forall q \in P, c(\gamma(\delta(q, c)) \cong \delta'(\gamma(q), c))$$
$$\forall q \in Q \setminus P(r(q) \in Q \setminus P \leftrightarrow r'(q) = r(q)$$
$$\wedge r(q) \in P \leftrightarrow r'(q) = \gamma(r(q))),$$
$$\forall q \in P(r(q) \in Q \setminus P \leftrightarrow r'(\gamma(q)) = r(q),$$
$$\wedge r(q) \in P \leftrightarrow r'(\gamma(q)) = \gamma(r(q))),$$
$$\forall q \in P(r(q) \simeq r'(\gamma(q))).$$

A' is the *relocation of P in A by* γ.□

Lemma. Under the conditions of the definition above, P' is a detachable sub automaton of A'.□

Lemma. Let A' is the relocation of *P* in *A* by some γ. Then $L(A') = L(A)$.□

Now we are ready to define the actual "shrinking" of a recursive automaton.

Definition. For any two detachable sub automatons P_1 and P_2 of *A* we say that P_1 is *mergeable into* P_2, $P_1 \lhd P_2$, if $P_1 \bigcap P_2 = \varnothing$ and there exist is an injection $\gamma : P_1 \to P_2$ such that

$$\forall q_1, q_2 \in P_1, q_1 \neq q_2(\gamma(q_1) \neq \gamma(q_2)),$$
$$\forall q \in P_1, c(\gamma(\delta_{P_1}(q, c)) \cong \delta_{P_2}(\gamma(q), c)),$$
$$\forall q \in P_1(r_{P_1}(q) \simeq r_{P_2}(\gamma(q))),$$
$$Exit_{P_2} = \gamma[Exit_{P_1}].□$$

Definition. Let $P_1 \lhd P_2$ be any couple of detachable sub automatons of *A*. Then

$Merge(A, P_1, P_2) = \langle \Sigma, Q', q_0, \delta', r', F' \rangle$, where

$$A_s = S(P_1, S(P_2, A)) = \langle \Sigma, Q_s, q_0, \delta_s, r_s, F_s \rangle,$$
$$Q' = Q_s \setminus P_1, \ \delta' = \delta_s \mid \Sigma \times Q',$$
$$r' = r_s \setminus \{(q, q') \mid q' \in Q_{P_1}\}$$
$$\cup \{(q, \gamma(q')) \mid (q, q') \in r_s, q' \in Q_{P_1}\},$$
$$F' = F_s \bigcap Q'.□$$

Theorem. Let $P_1 \lhd P_2$ be any couple of detachable sub automatons of *A*. Then

$$L(A) = L\left(Merge\left(A, P_1, P_2\right)\right).\square$$

What we did was to detach P_1 and P_2, redirect all recursive transitions pointing to P_1 from the outside world to start pointing to P_2 and finally erase P_1.

Definition. A' is called *a reduction of A*, $A \to A'$, if there exist $P_1 \lhd P_2$ - detachable sub automatons of *A* and $A' = Merge\left(A, P_1, P_2\right)$. We also denote this by $A \overset{P_1 \lhd P_2}{\to} A'.\square$

Theorem. There ***are no*** infinite sequences of reductions $A_1 \to \ldots \to A_n \to \ldots.\square$

The last theorem tells us that we can not reduce a recursive automaton forever. Please note that reducing an automaton can actually increase the number of its states and transitions due to the splitting operation which adds a number of new places and transitions "surrounding" the sub automatons. This is why in the implementation which we use, the reductions are evaluated by the number of states and transitions that have been economized. Only reductions with a possible value are performed.

6. Finding sub automatons to merge

The majority of works investigating the "isomorphic sub graphs of graph" problem, for example [3], consider non directed graphs $G = \langle V, E \rangle$ with unweighted edges. They seek a break down of the set of vertices V in the form $V = P_1 + P_2 + Q$, where

$$P_1 \bigcap P_2 = \varnothing,\ P_1, P_2 \bigcap Q = \varnothing,$$

and P_1 is isomorphic to P_2. Generally two types of algorithms are being applied – heuristic approaches which make a number of "attempts" each time starting from two arbitrary sub graphs, extending them with vertices greedily; complete solutions are also available – these methods include finding maximal cliques in the "second degree" of a graph.

Our major goal is to have a compressed version of an automaton which still accepts words for linear time, so we can afford a higher complexity of the merging since we'll be doing it only once. Unfortunately, the "second degree" approach can not be directly applied to our problem since we work with directed weighted graphs. We have developed a greedy approach for finding classes of mergeable sub automatons.

Definition. A *component over Q of order k, generated by the state* $q \in Q$ is the set

$Q_q^k = \{q' \mid \exists q = q_1, \ldots, q_m = q'$ - a path in A, where m $\le$ k+1$\}$

We will also say *k-generated component over Q* if we do not care of the generating state.$\square$

Definition. We say that $Q_{q_1}^k$ and $Q_{q_2}^k$ are *isomorphic*, $Q_{q_1}^k \sim Q_{q_2}^k$ if $Q_{q_1}^k \bigcap Q_{q_2}^k = \varnothing$ and there is a bijection $\gamma : Q_{q_1}^k \to Q_{q_2}^k$ such that

$$\left(\forall q_1, q_2 \in Q_{q_1}^k\right)\left(\forall c \in \Sigma\right)\left(\left(\delta\left(q_1, c\right) = q_2 \leftrightarrow \delta\left(\gamma\left(q_1\right), c\right) = \gamma\left(q_2\right)\right)\right.$$
$$\left.\wedge\left(r\left(q_1\right) \cong r\left(\gamma\left(q_1\right)\right)\right)\right)$$
$$\wedge\left(\forall q \in Q\right)\left(q \in F \leftrightarrow \gamma\left(q\right) \in F\right).\square$$

Two components are isomorphic if the graphs of their δ transitions are isomorphic and the respective recursive transitions point to the same places. Note that this relation is not transitive because of the non-intersection requirement.

Definition. The set $E^k \subseteq 2^Q$ is called an *isomorphic class over Q of order k* if every $e \in E$ is a k-generated component over *Q* and $\left(\forall e_1, e_2 \in E\right)\left(e_1 \sim e_2\right).\square$

Definition. An isomorphic class E^k is maximal if $\forall e\left(e \in 2^Q \setminus E^k\right)\ \exists e' \in E^k$ such that $\neg\left(e \sim e'\right).\square$

Since there is a $k_{\max}$ such that there are no components of order greater than $k_{\max}$, our problem can be states as: "find all maximal isomorphic classes of any order in *A*". If Q_k is the set of all components of order *k*, we can consider the graph

$$G = \left(Q^k, \left\{\left(Q_1, Q_2\right) \mid Q_1, Q_2 \in Q^k, Q_1 \sim Q_2\right\}\right)$$

Then our problem is to find all different maximal $\sim$ cliques in *G*. Unfortunately, this NP complete problem leads to unbearable amount of time for a practical application of the algorithm. This is why we take a greedy approach.

Definition. Let Q_q^k be a component of order *k*. Then $s\left(Q_q^k\right) = Q_q^{k+1}.\square$

Definition. $o : 2^Q \to 2^{2^Q}$ is a *generating operation* if it satisfies:

(1) $\forall e \in o\left(E^k\right)$ is an isomorphic class of order $k+1$

(2) $\left(\forall Q_q^k \in E^k\right)\left(s\left(Q_q^k\right) \in \bigcup o\left(E^k\right)\right)$

(3) if $Q_{q_1}^k, Q_{q_2}^k \in E^k \wedge Q_{q_1}^k \sim Q_{q_2}^k \wedge s\left(Q_{q_1}^k\right) \sim s\left(Q_{q_2}^k\right)$

then $\left(\exists e \in o\left(E^k\right)\right)\left(|e|>1\right)$. □

Definition. $O: 2^{2^Q} \to 2^{2^Q}$ is

$$O(E) = \bigcup \{ o(e) \mid e \in E \},$$

where $o: 2^Q \to 2^{2^Q}$ is a generating operation. □

The idea behind our greedy algorithm is to perform a number a steps in the beginning of which we have a set of isomorphic classes of the same order and in the end of which we have another set of isomorphic classes of the next order. The generating operation is a set of weak rules describing the way we get the classes "extended".

In the implementation we used, we've taken the following approach: On the first step, generate all isomorphic classes of order 0 by finding the factor classes of the following $Q \times Q$ relation:

$$q_1 \sim_{out} q_2 \leftrightarrow (\forall c)\left(!\delta(q_1, c) \leftrightarrow !\delta(q_2, c)\right).$$

Then out of each factor class F_i we produce the isomorphic class $E_i^0 = \{\{q\} \mid q \in F_i\}$. Each isomorphic class or order zero is a set of singletons, containing states having the same symbols on their outgoing transitions.

Then on each step *k*, the generating operation first makes $\{s(e) \mid e \in E_i^k\}$ for every isomorphic class and then greedily constructs a number of cliques of the $\sim$ relation out of it. These cliques are the isomorphic classes of order k+1.

Finally, we have a set of isomorphic classes which cannot be further extended. It is clear that components in a class may intersect with components of other classes. If we merge the components in one of the intersecting classes we can no longer merge the corresponding components from the other class – the sub automatons are being changed. The rule we apply to select which class to merge first is simple – we order the classes by the number of states and transitions what would be economized if their components are merged.

We apply this algorithm on minimal acyclic determined finite state automatons build from arbitrary languages to produce a recursive automaton with a smaller number of states and transitions. Then we apply it again on the result and keep applying it until no profit is gained.

7. Experimental results

The analysis of our experimental results showed that most of the FSAs build on practically applied dictionaries have a significant number of isomorphic sub automatons. However, the majority of these sub automatons have three or two states and merging them is not perspective – the number of the added new states and transitions is actually bigger than the number of erased states and transitions. Table 1 summarizes experimental results of applying the described idea over a number of English and Bulgarian dictionaries and on a bibliographic list.

Table 1.

	before merging		after merging	
automaton	**states**	**transitions**	**states**	**transitions**
small-wordlist-1k	1 853	3 004	1 763 (95%)	2 982 (99%)
wordlist-bg-760k	28 381	74 365	25 742 (91%)	73 255 (99%)
wordlist-bg-70k	31 171	66 006	27 874 (89%)	64 634 (98%)
yawl-260k	77 207	181 685	68 794 (89%)	178 007 (98%)
titles-first-10k	191 432	201 366	69 180 (36%)	107 750 (54%)
titles-every12th-10k	239 252	249 774	86 489 (36%)	132 642 (53%)
titles-every6th-20k	437 081	458 087	150 373 (34%)	235 338 (51%)

We can see seven dictionaries, the initial number of their states and transitions and the number of states and transitions of the resulting recursive dictionary. The percentage in brackets shows the ratio of the resulting number and the initial number. The dictionaries applied are: small-wordlist-1k – a small dictionary of 1,000 English words; wordlist-bg-760k - a dictionary of 760,000 Bulgarian words; wordlist-bg-70k – 70,000 Bulgarian words; yawl-260k - 260,000 English words; titles-first-10k – the first 10,000 lines of a bibliographic list with 120,000 lines; titles-every12th-10k – every twelfth lines of the same list; titles-every6th-20k – every sixth line of the same list.

8. References

[1] Hopkroft, Ulman, Automata

[2] J. Autbert, J. Berstel, L. Boasson. Context-Free languages and Pushdown Automata.

[3] Sabine Bachl, Computing and drawing isomorphic subgraphs, Journal of graph algorithms and applications, http://jgaa.info, vol.8, no.2, pp. 215-238 (2004)