

Incremental Construction of Minimal Deterministic Finite Cover Automata

Cezar CÂMPEANU^a Andrei PĂUN^{b,*} Jason R. SMITH^b

^a*Department of Computer Science and Information Technology,
University of Prince Edward Island, Charlottetown, P.E.I., Canada C1A 4P3;
email: ccampeanu@upei.ca*

^b*Department of Computer Science/ Institute for Micromanufacturing,
College of Engineering and Science, Louisiana Tech University,
Ruston, P.O. Box 10348, Louisiana, LA-71272 USA;
emails: {apaun, jrs026}@latech.edu*

Abstract

We present a fast incremental algorithm for constructing minimal DFCA for a given language. Since it was shown that the minimal DFCA for a language L has less states than the minimal DFA for the same language L , this technique seems to be the best choice for incrementally building the automaton for a large language, especially when the number of states in the DFCA is significantly less than the number of states in the corresponding minimal DFA. We have implemented the proposed algorithm and have tested it against the best known DFCA minimization technique.

Key words: Deterministic Finite Cover Automata (DFCA), DFA, DFCA minimization, Incremental Construction

1 Introduction

We have witnessed in recent years a growing interest in the design of incremental algorithms for finite automata [1,7,8,11,15–20]. The reason behind the (renewed) interest for such incremental algorithms used for building a minimal *Deterministic Finite Automata* (DFA) for a given dictionary (finite language) comes from the observation that such an incremental algorithm could have

* To whom correspondence should be addressed. E-mail: apaun@latech.edu; telephone: +1-318-257-5135; fax: +1-318-257-5104

much smaller memory requirements than a “global” minimization algorithm with little or no increase in the time complexity of the overall minimization process.

The small memory requirements for incremental algorithms (as opposed to “classical” minimization techniques) come from the fact that the DFA for a finite language is built word-by-word and minimized as words are inserted into the DFA. In this way the state complexity (and thus the memory requirements) of the incrementally built DFA could remain small as compared to the state complexity of a *trie* built for the whole dictionary as a first step and then minimizing the *trie* into a small DFA using a fast algorithm such as the one described by Hopcroft which requires $O(n \log(n))$ time and $O(n)$ space.

In the current paper we continue the research work in the incremental algorithms area, with the observation that for finite languages there is the recently defined concept of *Deterministic Finite Cover Automata* (DFCA) [6]. To conserve even more memory during the intermediate steps of the construction of the automaton we will devise an incremental algorithm for DFCA. We have proved in [4] that when transforming an NFA into a DFA and also into a DFCA the DFA can have exponentially more states than the DFCA; thus, there is a large class of languages for which the DFCA is the desirable representation as opposed to the DFA. For the recent results and properties of DFCA we refer the reader to [2–4,6,10,12,13]. We note that a Hopcroft-type algorithm (with $O(n \log(n))$ time and $O(n)$ space complexities) for the minimization of DFCA was described in [10], but no incremental algorithms for DFCA are known. Another advantage of incremental solutions, beside efficiency, is the maintenance of the automaton, since the technique for increasing the number of words in the dictionary is already built-in. We fill this gap by describing an incremental algorithm for DFCA in the current paper. We have implemented in the *Grail+* package [22] both the algorithm from [10] and the incremental algorithm proposed in the current paper; the preliminary tests suggest that the incremental algorithm is far superior with respect to the memory requirements as opposed to the Hopcroft-like algorithm, while no noticeable slow-down was observed for the languages tested.

The presented algorithm has complexity $O(kn)$ in time and $O(n^2)$ in space for adding a word of size k into a DFCA with n states. The time complexity is considered linear in literature in such a case (see [16]) due to the fact that the size of a word from the language is usually much smaller than the size of the automaton accepting the language. Thus, we provide a fast incremental algorithm (having the same time complexity as the known incremental algorithms for DFA described in [20]), but with a small increase in the memory requirements. This increase is small since, in practice, the complexity n is usually of logarithmical order of the complexity (trie size) of non-incremental algorithms. We will give an example of a language containing 2^n words for

which the Hopcroft-like algorithm for DFCA requires $O(2^n)$ space, whereas our algorithm requires $O(n^2)$ space.

For more information on incremental algorithms for DFA we refer the reader to [1,7,8,11,16–20]. It is worth noting that the paper [7] could be very interesting as it provides a comparison between the major algorithms (incremental/non-incremental) for the DFA, the comparison being performed using various dictionaries.

2 Preliminaries

We assume the reader is familiar with the basic notations of formal languages and finite automata, cf. e.g. [9,14,21]. The cardinality of a finite set A is denoted with $\#A$, the set of words over a finite alphabet Σ is denoted Σ^* , and the empty word is λ . The length of a word $w \in \Sigma^*$ is denoted $|w|$. The set of words over Σ of length at most (respectively, at least) n is denoted $\Sigma^{\leq n}$ (respectively $\Sigma^{\geq n}$).

For a DFA $D = (\Sigma, Q, q_0, \delta, F)$, we can always assume, without loss of generality, that $Q = \{0, 1, \dots, n-1\}$ and $q_0 = 0$; we will use this every time is convenient for simplifying our notations. If L is a finite language, we denote by l the maximum among the length of words in L .

Definition 1 *A language L' over Σ is called a cover language for the finite language L if $L' \cap \Sigma^{\leq l} = L$. A deterministic finite cover automaton (DFCA) for L is a deterministic finite automaton (DFA) A , such that the language accepted by A is a cover language of L .*

Definition 2 *Let $x, y \in \Sigma^*$. We define the following similarity relation by: $x \sim_L y$ if for all $z \in \Sigma^*$ such that $xz, yz \in \Sigma^{\leq l}$, $xz \in L$ iff $yz \in L$, and we write $x \not\sim_L y$ if $x \sim_L y$ does not hold.*

Definition 3 *Let $A = (Q, \Sigma, \delta, 0, F)$ be a DFA (or a DFCA). We define, for each state $q \in Q$, $level(q) = \min\{|w| \mid \delta(0, w) = q\}$.*

Definition 4 *Let $A = (Q, \Sigma, \delta, 0, F)$ be a DFCA for L . We consider two states $p, q \in Q$ and $m = \max\{level(p), level(q)\}$. We say that p is similar with q in A , denoted by $p \sim_A q$, if for every $w \in \Sigma^{\leq l-m}$, $\delta(p, w) \in F$ iff $\delta(q, w) \in F$. We say that two states are dissimilar if they are not similar.*

The following theorem gives the procedure to “merge” two similar states.

Theorem 5 *Let $A = (Q, \Sigma, \delta, s, F)$ be a DFCA of L . Suppose that for $p, q \in Q$, $p \sim_A q$, $p \neq q$ and $level(p) \leq level(q)$. Then we can construct a DFCA,*

$A' = (Q', \Sigma, \delta', s, F')$, for L such that $Q' = Q - \{q\}$, $F' = F - \{q\}$, and for each $t \in Q'$ and $a \in \Sigma$ we have $\delta'(t, a) = \begin{cases} \delta(t, a) & \text{if } \delta(t, a) \neq q, \\ p & \text{if } \delta(t, a) = q \end{cases}$.

We say that q is merged into p if we can apply the above theorem for p and q .

Definition 6 A DFCA A for a finite language is a minimal DFCA if and only if any two different states of A are dissimilar.

Theorem 7 Any minimal DFCA of L has the same number of states.

We refer the reader to [6] for the proofs of the above results.

Definition 8 In a DFA $A = (\Sigma, Q, 0, \delta, F)$, for a finite language L , we define the gap between two states p and q as the length of the shortest word $z \in \Sigma^*$ that distinguishes p from q : $gap_A(p, q) = \min(\{|z| \mid \delta(p, z) \in F \text{ and } \delta(q, z) \notin F\} \cup \{|z| \mid \delta(p, z) \notin F \text{ and } \delta(q, z) \in F\})$.

The gap between each pair of states will be a matrix called the ‘‘gap’’ table [12].

For better readability, we will set the notations for the subsequent results: $L \subset \Sigma^*$ is a finite language over an alphabet Σ and l is the length of the longest word(s) in L . We also consider $C = (\Sigma, Q_C, \delta_C, 0, F_C)$ a minimal DFCA for L ($L = L(C) \cap \Sigma^{\leq l}$), where $Q_C = \{0, 1, \dots, n - 1\}$. Let $w \in \Sigma^*$, $w = w_1 \dots w_k$, $w_i \in \Sigma$, $1 \leq i \leq k$ be the new word to be added to or deleted from the language L .

We denote the minimal DFA accepting $\{w\}$ by $W = (\Sigma, Q_W, 0, \delta_w, F_W)$, where $Q_W = \{0, 1, \dots, k + 1\}$, $\delta_w(i, w_{i+1}) = i + 1$, for all $0 \leq i < k$, $\delta_w(i, a) = k + 1$, for all $0 \leq i \leq k$ and $a \neq w_i$, or $i = k + 1$, $F_W = \{k\}$.

Let us denote by $s_i = \delta_C(0, w_1 \dots w_i)$, $1 \leq i \leq k$, $s_0 = 0$ and $S = \{s_i \in Q_C \mid 0 \leq i \leq k\}$. It is clear that $\#S \leq k + 1$.

We will consider two cases: adding the word w to the language L and deleting the word w from the language L . In the first case we also distinguish two other sub-cases: when $k \leq l$ and when $k > l$.

3 Adding a word to the language of a DFCA

In this section we will give the algorithm for adding a word to the language accepted by a minimal DFCA while keeping the new DFCA minimal. We want to construct the minimal DFCA A recognizing the language $L \cup \{w\}$.

We construct different algorithms for the two sub-cases: $k \leq l$ and $k > l$.

3.1 Adding a word shorter than the longest word in the DFCA ($k \leq l$)

We consider now the case where the newly added word w has length less than or equal to l . We will first modify the cover automaton C such that the new automaton A will accept a cover language for $L \cup \{w\}$. The construction is the standard Cartesian product between two automata [9]. We observe that the automaton W has a particular shape (a “line”), making many of the states in the Cartesian product unreachable.

Before giving the actual construction we note that the states of the form $(p, k+1)$ with $p \in Q_C$ from the Cartesian product will have the same transitions as in the automaton C . Moreover, such a state is final in A if and only if $p \in F_C$. Another crucial observation is that for each (p, i) of the new automaton where i is not the sink state of W (i.e. $i \neq k+1$), $p = s_i$. Due to the particular shape of the automaton W and the fact that C is deterministic, we have that the number of such states (p, i) is equal to the number of different prefixes for w . We now know that the number of states that can be reachable from the start state $(0, 0)$ can be at most $\#Q_C + k + 1$: $\#Q_C$ states of the form $(p, k+1)$ and at most $k + 1$ states of the form (p, i) with $i \neq k+1$. Thus, the original automaton C can be “embedded” in the new automaton with its states becoming the states $(p, k+1)$ in the Cartesian product. It should be clear that the following construction is equivalent to the standard Cartesian product between C and W .

We now construct the DFA $A = (\Sigma, Q_A, \delta_A, 0_A, F_A)$, with $Q_A = Q_C \cup Q_W$; each state $q \in Q_C$ is denoted in Q_A by $(q, k+1)$ and each state $i \in Q_W$ is denoted by (s_i, i) . The initial state is $(s_0, 0) = (0, 0)$, the set of final states is $F_A = \{(q, s) \mid q \in F_C \text{ or } s = k\}$, and the transition function δ_A is given by the the following formula:

$$\delta_A((p, i), a) = \begin{cases} (s_{i+1}, i+1) & \text{if } 0 \leq i \leq k, p = s_i, \text{ and } w_{i+1} = a \\ (\delta_C(p, a), k+1) & \text{otherwise.} \end{cases}$$

We can now use a standard breadth first search (BFS) algorithm to compute/update the levels of the states in Q_A , as well as detecting any unreachable

states of the form $(p, k + 1)$.

The next step is to minimize this DFCA; we are now interested in detecting all the similarities in the automaton A . To do this we will use the notions *level* and *gap* of the states as they were defined in Section 2.

Remark 9 *The level in A for the reachable states (p, i) is at least the level in C of the state p , for all values of i , $0 \leq i < k + 1$, since we do not introduce any “shortcuts”. For a state $(p, i) \in Q_A$, if $0 \leq i < k$, then $level_A((p, i)) = i$; also if $i = k + 1$, we have that $level_A((p, i)) \geq level_C(p)$.*

We shall call from now on the states $(p, k + 1)$, “original states” and the states (s_i, i) , “cloned” states (see also [8,20]). Therefore the set S is the set of states with clones in Q_C .

If $x \in \Sigma^*$ is not a prefix of w , we have $\delta_A((0, 0), x) = (\delta_C(0, x), k + 1)$ and for each $q \in Q_C - S$ if $\delta_A((0, 0), x) = (q, k + 1)$, then x is not a prefix of w , therefore $level_A(q, k + 1) = level_C(q)$. On the other hand, if x is a prefix of w , i.e., $x = w_1 \dots w_i$, we have $\delta_A((0, 0), x) = (s_i, i)$.

Lemma 10 *If $level_A(s_i, k + 1) = level_C(s_i)$, we have $level_A(s_{i+1}, k + 1) = level_C(s_{i+1})$ for $1 \leq i \leq k$.*

PROOF. Let us assume that $level_A((s_i, k + 1)) = level_C(s_i)$. We distinguish two cases: either $level_C(s_{i+1}) = level_C(s_i) + 1$ or $level_C(s_{i+1}) < level_C(s_i) + 1$.

In the first case $level_C(s_{i+1}) \leq level_A(s_{i+1}, k + 1) \leq level_A(s_i, k + 1) + 1 = level_C(s_i) + 1 = level_C(s_{i+1})$, thus we have that $level_C(s_{i+1}) = level_A((s_{i+1}, k + 1))$.

The second case means that there is $u \in \Sigma^*$ such that $\delta_C(0, u) = s_{i+1}$, $|u| < level_C(s_i) + 1$, $u \neq w_1 \dots w_i w_{i+1}$, and without the loss of generality we can choose u the shortest with these properties. Therefore, $\delta_A((0, 0), u) = (s_{i+1}, k + 1)$ and we have $level_C(s_{i+1}) \leq level_A(s_{i+1}, k + 1) \leq |u| = level_C(s_{i+1})$. \square

One can note that we may have several equalities $q = s_i = s_j$ for some $i \neq j$ if the same state q is reached by two different prefixes (of lengths i and j) of the new word w . We call a state (s_i, i) to be the *first cloned state* of q if $q = s_i$ and index i is the lowest with this property, i.e., $i = \min(\{j \mid \delta_C(0, w_1 w_2 \dots w_j) = q\})$. The following lemma shows that for each cloned state the level could increase only for the first cloned state or only for the original state, but not for both.

Lemma 11 *Let i , $0 \leq i \leq k$, be such that (s_i, i) is the first cloned state of a*

state $q \in Q_C$. Then we have the following properties:
if $level_A(s_i, k+1) > level_C(s_i)$, then $level_A(s_i, i) = level_C(s_i)$;
if $level_A(s_i, i) > level_C(s_i)$, then $level_A(s_i, k+1) = level_C(s_i)$.

PROOF. For $i = 0$, the lemma is true, since $level(s_0, 0) = 0$. Let us now consider $i > 0$, then $level_A(s_i, i) \geq level_C(s_i)$ and $level_A(s_i, k+1) \geq level_C(s_i)$.

It is enough to prove that $level_A(s_i, i) > level_C(s_i)$ implies $level_A(s_i, k+1) = level_C(s_i)$. Assume $level_A(s_i, i) > level_C(s_i)$ then we have $\delta_C(0, w_1 \dots w_i) = s_i$ and, $\delta_C(0, u) = s_i$ for some $u \in \Sigma^{<i}$. We choose u such that $|u| = level_C(s_i)$. We can note that u cannot be a prefix of w (otherwise it will contradict the assumption that (s_i, i) is the first clone of the state s_i). We can now compute the level of $(s_i, k+1)$: $level_A(s_i, k+1) = \min\{|v| \mid \delta_A((0, 0), v) = (s_i, k+1)\}$ and since u is not a prefix of w , we have $u \in \{|v| \mid \delta_A((0, 0), v) = (s_i, k+1)\}$ and u has the smallest length of all those words (from the construction of the automaton A and the way u was chosen), thus $level_A(s_i, k+1) = |u| = level_C(s_i)$. \square

Corollary 12 *There are at most $k+1$ states (s_i, i) , $(s_i, k+1)$, $0 \leq i \leq k$ for which we have that $level_A((s_i, i)) > level_C(s_i)$ or $level_A((s_i, k+1)) > level_C(s_i)$.*

PROOF. The total number of distinct states we consider is $k+1 + \#S$. We observe that there is a one to one correspondence between the states s in S and their first clone (s_i, i) , where $s_i = s$. Using Lemma 11 for each $s \in S$ we count at most once for the two states $(s, k+1)$ and (s, i) (where, again, i gives the index of the first clone). Hence, have at most $k+1 + \#S - \#S = k+1$ states for which $level_A((s_i, i)) > level_C(s_i)$ or $level_A((s_i, k+1)) > level_C(s_i)$. \square

Remark 13 *The gap between two “original” states is the same in A as it is in C , i.e., $gap_A((p, k+1), (q, k+1)) = gap_C(p, q)$.*

Remark 14 *We have that two states (p, i) and (q, j) are similar (by definition) in A if $gap_A((p, i), (q, j)) + \max\{level_A((p, i)), level((q, j))\} > l$.*

The following result drastically reduces the number of possible similarities in the automaton A ; we will see that we need to check for similarities only between states considered in Lemma 11 and the other states in A .

Lemma 15 *The states $(p, k+1)$ and $(q, k+1)$ are dissimilar if $level_A(p, k+1) = level_C(p)$ and $level_A(q, k+1) = level_C(q)$.*

PROOF. Assume they are similar. Therefore, using the definitions of similarity and gap , we have that $gap_A((p, k + 1), (q, k + 1)) + \max(level_A(p, k + 1), level_A(q, k + 1)) > l$. Since the two states did not change their levels ($level_A(p, k + 1) = level_C(p)$ and $level_A(q, k + 1) = level_C(q)$) and the function gap does not change for pairs of states with $k + 1$ on the second component, we have that $gap_C(p, q) + \max(level_C(p), level_C(q)) > l$, which means $p \sim_C q$, contradicting our assumption of minimality for C . \square

Following the result in Lemma 15 and using Remark 9 and Remark 13 our algorithm needs to identify only the similarities between the states of the type (s_i, i) or $(s_i, k + 1)$ whose levels might have increased and all the other states in the automaton (including similarities between these states).

To achieve this goal we will store in memory all the computed values of gap between any two states (from the previous step) and after adding the new word to the language, use this information at the current step to compute/update the similarities between any states.

Let us count how many similarities between “old states” may occur. Any two states q and p are dissimilar in C , but the states $(p, k + 1)$ and $(q, k + 1)$, may become similar. This can happen only if at least one of them changes its level and $gap_A((p, k + 1), (q, k + 1)) + \max\{level_A((p, k + 1)), level_A((q, k + 1))\} > l$, i.e., only when one of them, say p , is equal to s_i , for some $0 \leq i \leq k$, according to Corollary 12.

Looking only at the $gap_A((s_i, k + 1), (q, k + 1))$ and at the new level(s) for $(s_i, k + 1)$ and $(q, k + 1)$, one can decide immediately whether $(p, k + 1) \sim_A (q, k + 1)$ or not. To compute the (new) levels in A we need to do this just for states $(s_i, k + 1)$, which takes at most $O(n)$ steps. To decide all the similarities between states of the type $(s_i, k + 1)$ and $(q, k + 1)$, one needs exactly n checks for each state $(s_i, k + 1)$, thus we have a total of $O(kn)$ such comparisons.

We now proceed to detect similarities between the states of the form (s_i, i) with $i \leq k$ and all the other states. We start with the last final state newly introduced, the state (s_k, k) . It is obvious that (s_k, k) is of level k and that using a transition labeled with a letter $a \in \Sigma$, the state (s_k, k) will go into a state of the form $(p, k + 1)$: $\delta_A((s_k, k), a) = (p, k + 1)$. To compute the gap between (s_k, k) and any “old state”, assuming that we have the gap table for all pairs of old states (see Remark 13), we use the following lemma.

Lemma 16 *For all $q \in F_C$, we have that*

$$gap_A((s_k, k), (q, k + 1)) = 1 + \min_{a \in \Sigma} \{gap_C(\delta_C(s_k, a), \delta_C(q, a))\}.$$

For all $q \in Q_C - F_C$, we have then that $gap_A((s_k, k), (q, k + 1)) = 0$.

PROOF. Since the gap between states with $k+1$ on the second component is not changed as observed in Remark 13, and $(\delta_C(s_k, a), \delta_C(q, a)) \in Q_C \times \{k+1\}$, by the definition of gap we obtain:

$$\begin{aligned} gap_A((s_k, k), (q, k+1)) &= 1 + \min_{a \in \Sigma} \{gap_A(\delta_A((s_k, k), a), \delta_A((q, k+1), a))\} \\ &= 1 + \min_{a \in \Sigma} \{gap_C(\delta_C(s_k, a), \delta_C(p, a))\}. \end{aligned}$$

The second part of the lemma is obvious. \square

Once the gap between (s_k, k) and all the original states is computed, then the gap between the state $(s_{k-1}, k-1)$ and all the old states plus (s_k, k) can be computed using a similar observation to Lemma 16. Denote by $S_m = \{(s_i, i) \mid i \geq m\} \cup \{(p, k+1) \mid p \in Q_C\}$, where $1 \leq m \leq k$.

Lemma 17 *Assume that the gap_A was computed between all pairs of states in S_m . Then one can compute the gap_A for all pairs of states from S_{m-1} .*

PROOF. Since $S_m \subset S_{m-1}$, we already have most of the values of gap_A computed; we only need to determine gap_A for $(s_{m-1}, m-1)$ and all the states from S_m . We notice that in one step the states $(s_{m-1}, m-1)$ and $(p, j) \in S_m$ will go in states from S_m , i.e., $\delta_A((s_{m-1}, m-1), a), \delta_A((p, j), a) \in S_m$, for all $a \in \Sigma$, thus the gap_A for S_{m-1} can be computed. \square

The exact formula to “extend” the gap_A from S_m to S_{m-1} is given by the following.

Remark 18 (1) *For any state $q \in Q_C$ and for any $0 \leq i < k$ we have:*

- (a) *if $(s_i, i) \in F_A$ and $(q, k+1) \in Q_A - F_A$ or vice versa, then*
 $gap_A((s_i, i), (q, k+1)) = 0;$
- (b) *if $((s_i, i) \in F_A$ and $(q, k+1) \in F_A)$ or $((s_i, i) \notin F_A$ and $(q, k+1) \notin F_A)$, then the gap is:*
 $gap_A((s_i, i), (q, k+1)) = 1 + \min_{a \in \Sigma} \{gap_A(\delta_A((s_i, i), a), \delta_A((q, k+1), a))\}.$

(2) *For any $0 \leq i < j \leq k$ we have:*

- (a) *if $(s_i, i) \in F_A$ and $(s_j, j) \in Q_A - F_A$ or vice versa, then*
 $gap_A((s_i, i), (s_j, j)) = 0;$
- (b) *if $((s_i, i) \in F_A$ and $(s_j, j) \in F_A)$ or $((s_i, i) \notin F_A$ and $(s_j, j) \notin F_A)$, then the gap is computed by: $gap_A((s_i, i), (s_j, j)) =$
 $1 + \min_{a \in \Sigma} \{gap_A(\delta_A((s_i, i), a), \delta_A((s_j, j), a))\}.$*

We can have a small speedup for the gap computation in the implementation of the algorithm by noticing that $gap_A((s_i, i), (p, k+1)) = gap_C(s_i, p)$ if $gap_C(s_i, p) < k - i$ for all $0 \leq i \leq k - 1$.

Lemma 16 and Remark 18 suggest the work of the algorithm. We first compute the gap between (s_k, k) and the old states; we now have the gap computed between all the states in S_k . At the second step we can compute the gap between $(s_{k-1}, k-1)$ and the states from S_k obtaining gap for S_{k-1} . At the next step we can compute the gap between $(s_{k-2}, k-2)$ and states from S_{k-1} using values of gap computed for S_{k-1} obtaining gap for S_{k-2} . The process can be iterated up until we have computed all the gap function for S_0 , which is actually the gap for all pairs of states in Q_A .

Once the gap matrix is fully computed, the similarities between any two states $p, q \in Q_A$ can be determined easily by checking the levels of p, q and the $gap_A(p, q)$ using Remark 14. We do this just for the “cloned” states and “original” states that change their levels, since all the other pairs of states are dissimilar by Lemma 15.

3.1.0.1 The Incremental Algorithm: We give now a sketch for the incremental algorithm proposed; a more detailed pseudocode can be found in the appendix along with the C++ source code (implementation in the Grail+ package).

Input $C, gap_C, Number, w, k, l$

Output $A, gap_A, Number$ such that $L(A) \cap \Sigma^{\leq l} = L(C) \cap \Sigma^{\leq l} \cup \{w\}$

Build A as, described in subsection 3.1

$Number[k] = Number[k] + 1$ /* We updated $Number[k]$ */

Do a breath first search traversal starting in $(0, 0)$ to compute the levels of states in Q_A .

For all $q \in Q_C$

 Compute $gap_A((s_k, k), (q, k+1))$ (cf. Lemma 16)

For $i=k-1$ down to 0

 For all $q \in Q_C$

 Compute $gap_A((s_i, i), (q, k+1))$ (cf. Remark 18)

 For $j=k$ downto $i+1$

 Compute $gap_A((s_i, i), (s_j, j))$ (cf. Remark 18)

The gap table is now updated. We now find the similarities.

For $i=k$ down to 0

 For all $q \in Q_C$

 Compute similarity for the pairs $(s_i, i), (q, k+1)$.

 Compute similarity for the pairs $(s_i, k+1), (q, k+1)$.

For $i=k-1$ down to 0

 For $j=k$ down to $i+1$

 Compute similarity for the pairs $(s_i, i), (s_j, j)$,

Reduce the automaton by merging the similar states.

We can improved performance if we consider applying Lemma 10 in our implementation, i.e., maintaining a list for which similarity should not be checked, since *level* and *gap* information may not change. The algorithm has been implemented in Grail+ and was tested against the algorithm presented in [10]. The source code of the implementation as well as the updated version of Grail+ will be made available by e-mail request and also can be downloaded from the following address <http://www.latech.edu/~apaun/cover.html>. The test language chosen was $L_k = \{w \mid |w| = k\}$ since it was expected that this language will provide good compression results for the DFCA. We obtained the following results¹ that show an excellent performance of our method for the chosen test language. He have been running both algorithms on the same computer (CPU: Pentium 4 3.4 GHz; Memory: 1GB DDR400; OS: Linux 2.6.8.1 kernel (Slackware 10.0)). In the table we give the name of the algorithm, the maximum number of states in the memory during the execution of the algorithm, the maximum memory space needed, and the time required for the algorithm to finish (for the Körner algorithm we give the time required without and then with the trie building).

| Algorithm | States | Memory req. | Time/time with trie | 1 | # Σ |
|-----------|--------|-------------|---------------------------|---|------------|
| Körner | 3905 | 70k | 1.512s/1.961s | 5 | 5 |
| Incremnt. | 18 | 1.8k | 0.461s | 5 | 5 |
| Körner | 19530 | 1.4M | 40.52s/52.706s | 6 | 5 |
| Incremnt. | 21 | 2.2k | 3.196s | 6 | 5 |
| Körner | 97655 | 7.0M | 24min 49.26s/34min 6.944s | 7 | 5 |
| Incremnt. | 24 | 2.7k | 22.420s | 7 | 5 |

3.2 Adding a word of length $|w| = k > l$

We start the discussion in this case by noting that if there exists $x \in L$, such that $l < |xu| \leq k$ and $\delta_C(0, x) = \delta_C(0, xu) = p$ (where $p \in Q_C$), then we need to “split” the state p , otherwise the word xu of length less than or equal to k will be also considered in the new cover language. In other words, to make sure that no other words are in the language accepted by the new DFCA A , all loops in C must be expanded to chains of length at least l . However, any chain of length greater than l should go in a “sink” state with the last transition of that chain because we accept only one word of length greater than l . We will

¹ It is worth mentioning that the words were inserted incrementally in the standard lexicographical order to the cover automaton. We do not believe that this fact had a significant influence on the time/space efficiency of the proposed algorithm.

construct the new automaton as having the level encoded in the state; thus, the states will be of the form (p, i) , where $p \in Q_C$ and $level_A((p, i)) = i$ (the level information will be attached to each state by construction).

We construct the following DFCA: $A = (\Sigma, Q_A, \delta_A, (n, 0), F_A)$, where:

$$\delta_A((p, i), a) = \begin{cases} (\delta_C(p, a), i + 1) & \text{if } 0 \leq i \leq l - 1, p \neq n \\ (\delta_C(s_i, a), i + 1) & \text{if } 0 \leq i \leq l - 1, p = n \text{ and } a \neq w_{i+1}, \\ (n, i + 1) & \text{if } 0 \leq i \leq k, p = n \text{ and } a = w_{i+1}, \\ (n, k + 1) & \text{in all other cases.} \end{cases}$$

$$F_A = \{(q, s) \mid q \in F_C, s \leq l\} \cup \{(n, i) \mid i \leq l, s_i \in F_C\} \cup \{(n, k)\}.$$

Of course, $Q_A \subset Q_C \times \{0, 1, \dots, k + 1\}$.

Lemma 19 *The DFA A constructed above is accepting $L(A) = L \cup \{w\}$.*

PROOF. Let x be a word in L . If x is not a prefix of w , then $\delta_A((n, 0), x) = (\delta_C(0, x), |x|)$. Since $x \in L$, we have that $\delta_C(0, x) \in F_C$, thus by the definition of F_A and because $|x| \leq l$, we also have that $(\delta_C(0, x), |x|) \in F_A$. This means that if x is not a prefix of w then $x \in L(A)$.

If x is a prefix of w , i.e., $x = w_1 \dots w_i$ we also have that $\delta_A((n, 0), x) = (n, |x|) = (n, i)$. But $(n, i) \in F_A$ if and only if $s_i \in F_C$, which is true if $x = w_1 \dots w_i$. Therefore, $L \subseteq L(A)$. We also have that $w \in L(A)$, since $\delta_A((n, 0), w) = (n, k) \in F_A$.

We will now prove that $L(A) \subseteq L \cup \{w\}$: in other words, for the automaton A , if $x \in L(A)$ then $x \in L \cup \{w\}$. Let $x \in L(A)$, i.e., $\delta_A((n, 0), x) \in F_A$. We distinguish two cases: when x is not a prefix of w and when x is a prefix of w .

In the first case, $\delta_A((n, 0), x) = (\delta_C(0, x), |x|)$, which implies by the definition of F_A that $|x| \leq l$ and $\delta_C(0, x) \in F_C$ thus it follows that $x \in L$.

In the second case, x is a prefix of w , so $\delta_A((n, 0), x) = (n, |x|)$. Since $x \in L(A)$, $(n, |x|)$ needs to be final in A , thus either $|x| = i$ and $i \leq l$, where $s_i \in F_C$ or $|x| = k$. In other words, x is a prefix of w that is in L or $x = w$, i.e., $x \in L \cup \{w\}$. \square

We now describe the properties of the automaton A : we can easily see that $level_A(p, i) \geq level_C(p)$ for all possible values of p and i . Also, one can note that the state $(n, k + 1)$ is a sink state and $level_A((n, k + 1)) = l + 1$.

Since this automaton has a particular form, we can speed up the process of completing the gap table for the new automaton by giving some formulae for particular pairs of states.

Remark 20 1. All states (p, l) , with $p \in F_C$ are final and they are equivalent to (n, k) . Therefore, they can be merged together; the gap between these states is $k + 1$.

2. The sink state $(n, k + 1)$, is similar with all non-final states which cannot reach a final state with a word of length at most $k - l - 1$.

3. The gap between the sink state and (n, k) (final state) is 0.

Using the above remarks and a technique similar to the one in [5,12] we can now compute the gap function for all states of A . For our algorithm we only need to compute the gap function for the sink state $(n, k + 1)$ and all other states. This is done using a BFS traversal for each final state of the graph associated with the new DFA while considering the arrows reversed.

Once we have the gap computed for the sink state and all other states we can proceed to the next step.

Let us compute $gap_A((p, i), (q, j))$ for states $p, q \in Q_C$, and $i, j \leq l$.

Remark 21 If $gap_C(p, q) + \max(i, j) \leq l$, $gap_A((p, i), (q, j)) = gap_C(p, q)$. For the states with higher levels, i.e., $gap_C(p, q) + \max(i, j) > l$, one can compute the gap_A table using the technique for computing gap for a (not necessarily minimal) DFA as in [12].

For computing the gap function between the states (n, i) and all other states we use the same technique used in Remark 18.

The minimization algorithm for this case is basically the same as for the case $|w| < l$, with the following differences:

- (1) the initial construction has to embed the level in the name of the state, and we do this up to level l ;
- (2) we first compute the gap between the sink state $(n, k + 1)$ and all other states;
- (3) the “old” states having several levels will inherit the gap table from C as described in Remark 21;
- (4) the next steps are the same as in [12], computing the gap function for the “newly introduced” states, using a formula as given below:

$$\text{gap}_A((n, i), (p, j)) = \begin{cases} 0, & \text{if } s_i \in F_C \text{ and } p \notin F_C \text{ or } s_i \notin F_C \text{ and } p \in F_C \\ 1 + \min_{a \in \Sigma} \{\text{gap}_A(\delta_A((n, i), a), \delta_A((p, j), a))\}, & \text{otherwise.} \end{cases}$$

Remark 22 *The time complexity for adding a word of length k greater than l increases significantly, since each time we “expand” the DFCA for l to a DFA for $L \cup \{w\}$ we can have an explosion in the number of states. This behavior is expected mostly in the case when n is much smaller than k .*

To avoid such explosions, the best choice is to start the incremental algorithm with the longest word in the language. When this cannot be done due to specific restrictions imposed by the problem/language considered, one should try to add it as soon as possible.

3.3 Deleting a word in the DFCA

When we delete a word from a finite language, we distinguish two cases:

- after deletion the length of the longest word in $L \setminus \{w\}$ has changed from l to $l' < l$ or
- the length of the longest word in $L \setminus \{w\}$ has not changed.

In the first case, by deleting the word, the length of the longest word will decrease, therefore the new minimal cover automaton must be constructed using the new length, thus more similarities could appear between states because of the change in l . In order to speed up the algorithm we keep track of the number of words of a given length in an array *Number* of length l . If $\text{Number}[l - 1] = 0$, the new l becomes the largest i less than l for which $\text{Number}[i - 1] \neq 0$. We will use the same standard Cartesian product construction as for adding a word of length less than or equal to l modifying the automaton C such that the new automaton A will accept a cover language for $L - \{w\}$ (only the final states are computed different).

We construct the DFA $A = (\Sigma, Q_A, \delta_A, 0_A, F_A)$, with $Q_A = Q_C \cup Q_W$; each state $q \in Q_C$ is denoted in Q_A by $(q, k + 1)$ and each state $i \in Q_W$ is denoted by (s_i, i) . The initial state is $(s_0, 0) = (0, 0)$, the set of final states is $F_A = \{(q, k + 1) \mid q \in F_C\} \cup \{(s_i, i) \mid s_i \in F_C, 0 \leq i < k\}$, and the transition function δ_A is given by the the following:

$$\delta_A((p, i), a) = \begin{cases} (s_{i+1}, i + 1) & \text{if } 0 \leq i \leq k, p = s_i, \text{ and } w_{i+1} = a \\ (\delta_C(p, a), k + 1) & \text{otherwise.} \end{cases}$$

Similar to the case of adding a word, we can now use a standard breadth first search (BFS) algorithm to compute/update the levels of the states in Q_A , as

well as detecting any unreachable states of the form $(p, k + 1)$.

The next step is to minimize this DFCA; we are now interested in detecting all the similarities in the automaton A . To do this we will use the *level* and *gap* information computed for the automaton A .

We can see that most results in subsection 3.1 for adding a word are also valid for word deletion; more precisely Lemma 10, Lemma 11, Corollary 12, and Remark 13 are also valid.

Remark 23 *The Remark 14 is also valid but in this case we have to consider l the length of the longest word in $L \setminus \{w\}$.*

Since the length may decrease, Lemma 15 is valid only for the case when l is unchanged, for all other cases we have to check for similarity for all pairs of states $(p, k + 1)$, $(q, k + 1)$, using *gap* and *level* information.

For updating the *gap* table Lemma 16 is no longer valid for this case and will be replaced by the following lemma:

Lemma 24 *For all $q \in F_C$, we have that $gap_A((s_k, k), (q, k + 1)) = 0$. For all $q \in Q_C - F_C$, we have that $gap_A((s_k, k), (q, k + 1)) = 1 + \min_{a \in \Sigma} \{gap_C(\delta_C(s_k, a), \delta_C(q, a))\}$.*

The proof is similar to the proof of Lemma 16, considering the fact that in Lemma 24 (s_k, k) is not a final state. Lemma 17 is also a valid for this case and therefore the sequence for updating *gap* table/similarity will be the same. The exact formula for computing gap_A for S_{m-1} once we know gap for S_m is given in the Remark 18.

Lemma 24, Remark 18, and Remark 23 suggest the work of the algorithm in the case of word deletion:

3.3.0.2 The Incremental Algorithm: We give now a sketch for the incremental algorithm proposed; a more detailed pseudocode can be found in the appendix along with the C++ source code (implementation in the Grail+ package).

Input C , gap_C , $Number$, w , k , l

Output A , gap_A , $Number$

Build A as, described in subsection 3.3

Do a breath first search to compute $level_A(p, i)$ for all $(p, i) \in Q_A$.
 $Number[k] = Number[k] - 1$ /* We updated $Number[k]$ */

```

For all  $q \in Q_C$ 
  Compute  $gap_A((s_k, k), (q, k + 1))$  (cf. Lemma 24)
For  $i=k-1$  down to 0
  For all  $q \in Q_C$ 
    Compute  $gap_A((s_i, i), (q, k + 1))$  (cf. Remark 18)
  For  $j=k$  downto  $i+1$ 
    Compute  $gap_A((s_i, i), (s_j, j))$  (cf. Remark 18)
The gap table is now updated. We now find the similarities.
If  $k = l$  and  $Number[k] = 0$  then
  update  $l$ 
  For all  $p, q \in Q_C$ 
    Compute similarity for all the pairs  $(p, k + 1), (q, k + 1)$ .
For  $i=k$  down to 0
  For all  $q \in Q_C$ 
    Compute similarity for the pairs  $(s_i, i), (q, k + 1)$ .
    Compute similarity for the pairs  $(s_i, k + 1), (q, k + 1)$ .
For  $i=k-1$  down to 0
  For  $j=k$  down to  $i + 1$ 
    Compute similarity for the pairs  $(s_i, i), (s_j, j)$ ,
Reduce the automaton by merging the similar states.

```

The same optimization based on Lemma 10 can also be implemented in this case.

4 Final remarks

Our incremental algorithm described in section 3.1.0.1 is fast, but it was observed (see e.g.[20]) that such an incremental algorithm could be modified to run even faster if one can perform a preprocessing (which is in fact sorting) of the input set of strings. We already have good results in this direction and submission of another paper describing an incremental algorithm for sorted input data is expected. The string subtraction has a similar algorithm to the string addition, and the complexity is similar. We also plan to conduct more experiments using real dictionaries and compare the difference in the memory requirements between our incremental algorithms and other DFCA minimization algorithms. It is also worth noting that the string addition algorithm for the case when $|w| > l$ can produce a high number of states, thus it is now efficient to first scan the words in the language and find the longest word (requires $O(n)$ time), and then to start the algorithm with the longest word in L . The discussion for the case $|w| > l$ is valuable if one needs the ability to update (maybe later) the language; for example, adding a new word to a spellchecker should permit also the addition of longer words. It is open whether a faster algorithm for the case $|w| > l$ can be devised, or whether one can design an

incremental algorithm of linear space for the string addition into the language of a DFCA.

5 Acknowledgments

We would like to mention the DFCA fruitful discussions with S. Yu and the insightful suggestions received from the anonymous referees. C. Câmpeanu gratefully acknowledges the support received from NSERC DGP-I249600; A. Păun gratefully acknowledges the support in part by LA BoR RSC grant LEQSF (2004-07)-RD-A-23, NSF IMR-0414903 and NSF CCF-0523572.

References

- [1] Rafael C. Carrasco, Mikel L. Forcada “Incremental Construction and Maintenance of Minimal Finite-State Automata”, *Computational Linguistics*, 28, 2, (2002), 207 – 216.
- [2] Cezar Câmpeanu, Andrei Păun, “Counting The Number of Minimal DFCA Obtained by Merging States”, *International Journal of Foundations of Computer Science*, 14, 6, (2003), 995 – 1006.
- [3] Cezar Câmpeanu, Andrei Păun, “Lower Bounds for NFA to DFCA Transformations”, *Proceedings of DFCS 2004*, London ON, Canada, 121 – 130.
- [4] Cezar Câmpeanu, Andrei Păun, Lila Kari, “Results on Transforming NFA into DFCA”, *Fundamenta Informaticae* 64, 1-4, (2005), 53 – 63.
- [5] Cezar Câmpeanu, Andrei Păun and Sheng Yu, “An Efficient Algorithm for Constructing Minimal Cover Automata for Finite Languages”, *International Journal of Foundations of Computer Science*, 13, 1 (2002) 83 – 97.
- [6] Cezar Câmpeanu, Nicolae Sântean, Sheng Yu, “Finite Languages and Cover Automata”, *Theoretical Computer Science*, 267, 1-2, (2001), 3 – 16.
- [7] Jan Daciuk, “Comparison of Construction Algorithms for Minimal, Acyclic, Deterministic, Finite-State Automata from Sets of Strings”, *Proceedings of Seventh International Conference on Implementation and Application of Automata CIAA 2002*, Tours, France, (2002) and LNCS Series 2608, (2003), 255 – 261.
- [8] Jan Daciuk, Stoyan Mihov, Bruce Watson, Richard E. Watson “Incremental Construction of Minimal Acyclic Finite State Automata”, *Computational Linguistics*, 26, 1, (2000), 3 – 16.
- [9] John E. Hopcroft, Jeffrey D. Ullman, *Introduction to Automata Theory, Languages and Computation* Addison-Wesley, 1979.

- [10] Heiko Körner, “A Time and Space Efficient Algorithm for Minimizing Cover Automata for Finite Languages”, *International Journal of Foundations of Computer Science*, 14, 6, (2003), 1071 – 1086.
- [11] Stoyan Mihov, “Direct Construction of Minimal Acyclic Finite States Automata”, *Ann. de l’Université de Sofia “St. Kl. Ohridski”, Faculté de Mathématique et Informatique*, Sofia, Bulgaria, Vol. 92, 1. 2, (1998).
- [12] Andrei Păun, Nicolae Sântean, Sheng Yu, “An $O(n^2)$ algorithm for Minimal Cover-Automata for Finite Languages”, *Proceedings of the 5th International Conference on Implementation and Application of Automata Implementing Automata CIAA’00*, (2000), 243 – 251.
- [13] Nicolae Sântean, *Towards a Minimal Representation for Finite Languages: Theory and Practice*, MSc Thesis, Department of Computer Science, The University of Western Ontario, 2000.
- [14] Arto Salomaa, *Formal Languages* Academic Press, 1973.
- [15] Kyriakos N. Sgarbas, Nikos D. Fakotakis, George K. Kokkinakis, “Two Algorithms for incremental construction of Directed Acyclic Word Graphs”, *International Journal on Artificial Intelligence Tools*, World Scientific, 4(3), (1995), 369 – 381.
- [16] Kyriakos N. Sgarbas, Nikos D. Fakotakis, George K. Kokkinakis, “Optimal insertion in deterministic DAWGs”, *Theoretical Computer Science*, 301, 1(3), (2003), 103 – 117.
- [17] Bruce W. Watson, “A Taxonomy of finite automata minimization algorithms”, *Eindhoven University of Technology, The Netherlands, Computing Science Note*, 93/44, (1993).
- [18] Bruce W. Watson, *Taxonomies and Toolkits of Regular Language Algorithms*, Ph.D. thesis, Eindhoven University of Technology, the Netherlands, 1995.
- [19] Bruce W. Watson, “An incremental DFA minimization algorithm”, *Finite State Methods in Natural Language Processing*, ESSLLI Workshop, Helsinki, Finland, August (2001), 20 – 24.
- [20] Bruce W. Watson, Jan Daciuk, “An efficient incremental DFA minimization algorithm”, *Natural Language Engineering*, 9, 1, (2003), 49 – 64.
- [21] Sheng Yu, “Regular languages”, in *Handbook of Formal Languages, Vol I*, eds. G. Rozenberg and A. Salomaa, Springer-Verlag, (1997), 41 – 110.
- [22] The Grail + Project. A symbolic computation environment for finite state machines, regular expressions, and finite languages. Available online at the following address: <http://www.csd.uwo.ca/research/grail/>

A The pseudocode for the algorithm

Input: A DFCA $C = (\Sigma, Q, 0, \delta, F)$ for the language L with the length of the largest word l . We also have as input the new word to be introduced or deleted w with $|w| \leq l$, and the precomputed arrays *Number*, *level* and *gap* which store the number of words of each length in the language, levels of the states and the gaps between states, respectively.

Output: A DFCA $A = (\Sigma, Q', 0', \delta', F')$ for the language $L \cup \{w\}$ (or $L \setminus \{w\}$) with the arrays *Number*, *level* and *gap* updated.

if we are: adding w and $w \in L$, removing w and $w \notin L$, or if $|w| > l$ return
/* do nothing in these cases*/

let $k = |w|$

create $k + 1$ new states with the labels $n, n + 1, \dots, n + k - 1, n + k$

if we are adding a word, the state $n + k$ is final; otherwise, it is not final.

create the arrays *old* of size $k + 1$ initialized with -1 and *merged* of size $n + k + 1$ initialized with 0. $old[0] = 0$

for $i = 1$ to k do $old[i] = \delta(old[i - 1], w[i])$

$new = n$

for $i = 0$ to $k - 1$ do

 for all $a \in \Sigma$ do $\delta(new, a) = \delta(old[i], a)$

$\delta(new, w[i]) = new + 1$; $new = new + 1$

for all $a \in \Sigma$ do $\delta(n + k, a) = \delta(old[k], a)$

Apply the Breadth First algorithm starting in n and compute/update the levels of all states $old[i]$ with $0 \leq i \leq k$. If such a state p becomes unreachable, then $merged[p] = 1$

for $j = 0$ to $n - 1$ do /*we compute the gaps between old states and $n + k$ */

 if $j \notin F$ then $gap[n + k, j] = 0$; $gap[j, n + k] = 0$

 else $min = l + 1$

 for all $a \in \Sigma$ do

$currentgap = gap[\delta(j, a), \delta(old[k], a)]$

 if $min > currentgap$ then $min = currentgap$

$gap[n + k, j] = min + 1$; $gap[j, n + k] = min + 1$

for $i = n + k - 1$ down to n do

 for $j = 0$ to $n - 1$ do /*find the gaps between i and the old states*/

$min = l + 1$

 for all $a \in \Sigma$ do

$currentgap = gap[\delta(j, a), \delta(i, a)]$

 if $min > currentgap$ then $min = currentgap$

$gap[i, j] = min + 1$; $gap[j, i] = min + 1$

 for $j = i + 1$ to $n + k$ do /*find the gaps between the new states*/

$min = l + 1$

 for all $a \in \Sigma$ do

```

    currentgap = gap[ $\delta(i, a), \delta(j, a)$ ]
    if  $min > currentgap$  then  $min = currentgap$ 
    gap[ $i, j$ ] =  $min + 1$ ; gap[ $j, i$ ] =  $min + 1$ 
/* We have the gap matrix completely computed, thus the similarities between
the new states and the old states can now be detected */
    for  $j=0$  to  $k$  do
        if  $old[j] \neq i$  and  $merged[i] + merged[j] = 0$  then
            lev = max(level[old[j]], level[i])
            if gap[old[j], i] + lev > l then merge(i, old[j])
If we add the word  $w$  then  $Number[k] = Number[k] + 1$ 
else /* we erase the word  $w$  */
    Number[k] = Number[k] - 1
    if  $k = l$  and  $Number[k] = 0$  then
        temp = k - 1
        while ( $Number[temp] = 0$  and  $temp > 1$ ) do  $temp = temp - 1$ 
        l = temp
        for  $i = 1$  to  $n + k$ 
            for  $j = 1$  to  $n + k$ 
                lev = max(level[i], level[j])
                if gap[i, j] + lev > l then merge(i, j)
for  $i = n + k$  down to  $n$  do /* if l was reduced,  $i = n + k$  down to 0 */
    for  $j = i - 1$  down to 0 do
        if merged[j] = 0 then lev = max(level[i], level[j])
        if gap[i, j] + lev > l then merge(i, j)
/* We now delete the rows and columns from the matrix gap for the states
that were deleted (either merged or unreachable)*/
swap(0, n) /* the new start state n is swapped with the old start state*/
 $i = 1$ ;  $j = n + k$ 
while ( $i < j$ ) do
    while ( $merged[j] = 1$ ) do  $j = j - 1$  /*j points to the last state that
does not disappear*/
    while ( $merged[i] = 0$ ) do  $i = i + 1$  /* i points to the first state that
disappears */
    swap(i, j) /* we update the gap level and merged arrays*/
     $i = i + 1$ ;  $j = j - 1$ 

```

B The implementation in GRAIL + of the algorithms presented.

The source code of the implementation as well as the updated version of Grail+ will be made available by e-mail request and also can be downloaded from the following address <http://www.latech.edu/~apaun/cover.html>.

```

/*****
File: classes/fcm/fltofcmsrc
-----
Description:
-----
This file contains the definition of the following functions:
void fcm<Label>::fltofcmsrc(const fl<Label>& lang)
Revision History:
-----
Jason Smith Initial version of source code
*****/
/*****
void fcm<Label>::fltofcmsrc(const fl<Label>& lang)
Description:
This function incrementally builds a minimal cover automaton from
a finite language. It requires n-squared space with respect to
the maximum number of states during its execution. If the input
is sorted, the maximum number of states is often much lower.
This version of the code is modified to include word subtractions.
Any word starting with - will be removed from the cover automata.
Parameters:
fl<Label>& - a finite language
Return Value: none
*****/

template <class Label>
void
fcm<Label>::fltofcmsrc(const fl<Label>& lang)
{
    // Temporary variables
    int a, i, j, k, n, x, y;
    // words contains the finite language we are building an
    // automaton for and current_word is the word being added
    // during each iteration of the main loop.
    set<string<Label> > words;
    string<Label> current_word;
    lang.get_words(words);
    // The attach() function is used to represent each state
    // by an array of transitions, rather than the default
    // representation.
    // all_states contains our representation of the automata.
    attach();
    // These are temporary values used for construction of the
    // automata.
    atstate<Label> *temp_state;
    atstate<Label> complete_state;
    inst<Label> temp_inst;
    // This loop finds all of the symbols used in the finite
    // language. It is necessary to know all of these symbols
    // beforehand. Note that - is ignored because it is used
    // to indicate word deletion.
    set<Label> alpha;
    for (i = 0; i < words.size(); i++)
    {
        j = 0;
        // Ignore the first letter if it is -
        if (words[i][0] == '-')
            j++;
        while(j < words[i].size())
        {
            alpha += words[i][j];
            j++;
        }
    }
    // start contains the index of the start state
    int start;
    // level contains an integer value for each state
    // corresponding to its level (the minimum number of
    // transitions that need to be taken from the start state
    // to reach this state).
    array<int> level;
    // The following are temporary variables used in computation
    // of the gap function and merging
    int lev;
    int temp;
    int jPrime, kPrime;
    bool tempb;
    int min;
    // deleting is true if we are deleting a word, and false
    // otherwise
    bool deleting;
    // l_reduced is true when l has been decreased and similarities
    // must be considered for the entire automata.
    bool l_reduced = true;
    // When adding a new word, the old states are the ones
    // that are cloned.
    array<int> old;
    // These variables are used to perform the breadth first
    // traversal which finds the level of each state.
    array<int> current_wave;
    array<int> next_wave;
    int level_count;

```

```

// newindex is used when reordering the states after some
// states have been merged.
array<int> newindex;
// merged contains a boolean value for each state
// corresponding to whether or not it has been merged.
array<bool> merged;
// The gap structure is used during minimization. It
// represents the minimum length of a word necessary to
// show that two states are not equivalent.
array<array<short> * > gap;
array<short> *gap_temp;
int currentgap;
if (words.size() == 0)
    return;
// First, find the longest word
// Ignore word subtractions
int max = 0;
for (i = 1; i < words.size(); i++)
    if ((words[i].size() > words[max].size()) && (words[i][0] != '-'))
        max = i;
maxlen = words[max].size();
// Initialize the sizes array. It contains the number of words
// of each possible length accepted by the automata. We keep track
// of this to know when to decrease the maximum length if a word
// is deleted.
array<int> sizes;
for (i = 0; i <= maxlen; i++)
    sizes += 0;
// Initialize the old array.
// There can only be as many cloned states as there are letters
// in a new word + 1
for (i = 0; i <= maxlen; i++)
    old += -1;
// complete_state is an array of transitions which only
// points to its own state. It is used when adding the
// first state.
for (i = 0; i < alpha.size(); i++)
{
    temp_inst.assign(0, alpha[i], 0);
    complete_state += temp_inst;
}
// Create a start state with transitions leading to itself.
// This is used as a starting point for the algorithm.
start = 0;
fm<Label>::start_states += 0;
all_states += new atstate<Label>();
(*all_states[all_states.size()-1]) = complete_state;
gap += new array<short>;
level += 0;
// wc is the location of the current word being added
int wc = 0;
while(wc < words.size())
// This is the main loop in which all of the words are added.
{
    current_word = words[wc];
    // Check to see if we are adding or deleting a word. Since
    // the algorithm is similar, we use the same code for both.
    // We also need to check if we are adding a word that is
    // already present or deleting a word that doesn't exist.
    // 0 is the correct start state at this point in the algorithm.
    fm<Label>::start_states[0] = 0;
    if ((current_word.size() > 0) && (current_word[0] == '-'))
    // We are deleting word
    {
        deleting = true;
        current_word.remove(0); // remove the -
        // If the word is not a member of the language,
        // move to the next word.
        if (member_of_language(current_word, 0) == 0)
        {
            wc++;
            continue;
        }
    }
    else // We are adding a word
    {
        deleting = false;
        // If the word is already a member of the language,
        // move to the next word.
        if (member_of_language(current_word, 0) != 0)
        {
            wc++;
            continue;
        }
    }
}
//*****
// This section of code modifies the automaton to accept
// the new word or erase the input word.
int current, prev;
current = all_states.size();
prev = start; // first state to be cloned
start = current; // the new start state (0, 0)

```

```

fm<Label>::start_states[0] = start;
n = all_states.size(); // 0 .. n-1 are the original states
// Initialize the old array
for(i = 0; i < old.size(); i++)
    old[i] = -1;
// Start by adding clone/queue states
for(i = 0; i < current_word.size(); i++)
{
    old[i] = prev;
    all_states += new atstate<Label>();
    (*all_states[all_states.size()-1]) = (*all_states[prev]);
    // fix the transitions in the cloned state
    for (j = 0; j < (*all_states[current]).size(); j++)
        (*all_states[current])[j].assign(current, (*all_states[current])[j].get_label(),
            (*all_states[current])[j].get_sink());
    level += i;
    gap_temp = new array<short>((*gap[all_states.size()-1]));
    (*gap_temp) += 0;
    gap += gap_temp;
    // Add (or modify) the transition to the next state
    prev = -1;
    for (j = 0; j < (*all_states[current]).size(); j++)
        if ((*all_states[current])[j].get_label() == current_word[i])
            {
                prev = (*all_states[current])[j].get_sink().value();
                (*all_states[current])[j].assign(current, current_word[i],
                    current + 1);
            }
        current++;
} // End clone/queue loop
// Add the final state accepting the new word (or rejecting it,
// if we are deleting a word).
all_states += new atstate<Label>();
(*all_states[all_states.size()-1]) = (*all_states[prev]);
if (deleting == false)
    (*all_states[all_states.size()-1]).set_final(true);
else
    (*all_states[all_states.size()-1]).set_final(false);
level += i;
old[i] = prev;
gap_temp = new array<short>((*gap[all_states.size()-1]));
(*gap_temp) += 0;
gap += gap_temp;
// fix the transitions in the cloned state
for (j = 0; j < (*all_states[current]).size(); j++)
    (*all_states[current])[j].assign(current, (*all_states[current])[j].get_label(),
        (*all_states[current])[j].get_sink());
/*****/
// Find the levels of all states and remove unreachable states
level = find_levels();
// Initialize the merged and newindex arrays
for (j = 0; j < merged.size(); j++)
    merged[j] = false;
while (merged.size() < all_states.size())
    merged += false;
for (j = 0; j < newindex.size(); j++)
    newindex[j] = j;
while (newindex.size() < all_states.size())
    newindex += newindex.size();
/*****/
// The following section of code performs the minimization
// of the automaton. It follows closely to the psuedocode.
// Merge old states with intact states
for (j = 0; j < n; j++)
    for (k = 0; k <= current_word.size(); k++)
        if ( (level[j] != -1) && (level[old[k]] != -1) && (old[k] != j)
            && (merged[j] == false) && (merged[old[k]] == false)
            && ((*all_states[old[k]].final() == (*all_states[j].final()) ) )
        {
            lev = (level[old[k]] > level[j] ? level[old[k]] : level[j]);
            if (old[k] > j)
            {
                if ((*gap[old[k]])[j] + lev > maxlen)
                {
                    if (level[old[k]] > level[j])
                    {
                        newindex[old[k]] = j;
                        merged[old[k]] = true;
                    }
                    else
                    {
                        newindex[j] = old[k];
                        merged[j] = true;
                    }
                }
            }
        }
    else
        if ((*gap[j])[old[k]] + lev > maxlen)
        {
            if (level[old[k]] > level[j])
            {
                newindex[old[k]] = j;
            }
        }
}

```

```

        merged[old[k]] = true;
    }
    else
    {
        newindex[j] = old[k];
        merged[j] = true;
    }
}
for (j = all_states.size() - 1; j > n-1; j--)
{
    for (k = 0; k < n; k++) // Begin computing gap between new and old states
    {
        if ((*all_states[j]).final() != (*all_states[k]).final())
            (*gap[j])[k] = 0;
        else
        {
            min = maxlen;
            for (a = 0; a < alpha.size(); a++)
            {
                jPrime = -1;
                kPrime = -1;
                for (x = 0; x < (*all_states[k]).size(); x++)
                    if ((*all_states[k])[x].get_label() == alpha[a])
                        kPrime = (*all_states[k])[x].get_sink().value();
                for (x = 0; x < (*all_states[j]).size(); x++)
                    if ((*all_states[j])[x].get_label() == alpha[a])
                        jPrime = (*all_states[j])[x].get_sink().value();
                if (jPrime != kPrime)
                {
                    if (jPrime > kPrime)
                        currentgap = (*gap[jPrime])[kPrime];
                    else
                        currentgap = (*gap[kPrime])[jPrime];
                    if (min > currentgap)
                        min = currentgap;
                }
            }
            (*gap[j])[k] = min + 1;
        }
    } // End computing gap between new and old states
}
for (j = all_states.size() - 1; j > n-1; j--)
{
    for (k = n; k < j; k++) // Begin creating gap between newly created states
    {
        if ((*all_states[j]).final() == (*all_states[k]).final())
            min = (*gap[j])[old[k-n]];
        else
            min = 0;
        jPrime = -1;
        kPrime = -1;
        for (x = 0; x < (*all_states[k]).size(); x++)
            if ((*all_states[k])[x].get_label() == current_word[j-n-1])
                kPrime = (*all_states[k])[x].get_sink().value();
        for (x = 0; x < (*all_states[j]).size(); x++)
            if ((*all_states[j])[x].get_label() == current_word[j-n-1])
                jPrime = (*all_states[j])[x].get_sink().value();
        if (jPrime != kPrime)
        {
            if (jPrime > kPrime)
                currentgap = (*gap[jPrime])[kPrime];
            else
                currentgap = (*gap[kPrime])[jPrime];
        }
        currentgap++;
        if (min > currentgap)
            min = currentgap;
        (*gap[j])[k] = min;
    } // End computing gap between newly created states
} // End computing gap
// Now we must update the sizes array, and recompute similarities
// if l changes.
if (deleting)
{
    sizes[current_word.size()]--;
    // If we have deleted the last word of length l, then the automaton
    // must be adjusted. It's possible that l must decrease by more than
    // one, so a loop checks for the new longest length
    if (sizes[maxlen] == 0)
    {
        i = 1;
        while ((i <= maxlen) && (sizes[maxlen-i] == 0))
            i++;
        // l_reduced must be set to true so that the similarities will
        // be computed on the entire automata.
        l_reduced = true;
        maxlen = maxlen-i;
    }
}
}
else // If we have added a word
{

```

```

        sizes[current_word.size()]++;
    }
    // Merge states
    int lower_limit; // Contains the index of the lowest state
                    // that we can merge.
    // If l_reduced is not true, we only need to merge new states
    // Otherwise, we must consider all states.
    if (l_reduced == false)
        lower_limit = n+1;
    else
        lower_limit = 0;
    for (j = all_states.size() - 1; j >= lower_limit; j--)
        for (k = j - 1; k >= 0; k--)
            if ((level[k] != -1) && (level[j] != -1))
                {
                    lev = (level[k] > level[j] ? level[k] : level[j]);
                    if (((*gap[j])[k] + lev > maxlen) && (merged[j] == false) &&
                        (merged[k] == false))
                        {
                            // Always merge a higher level state into a lower level state.
                            // Also, we must check to see if a state has already been merged.
                            if (level[k] > level[j])
                                {
                                    newindex[k] = j;
                                    merged[k] = true;
                                }
                            else
                                {
                                    newindex[j] = k;
                                    merged[j] = true;
                                }
                        }
                }
    l_reduced = false; // If l was reduced, the changes have been made
                    // and l_reduced can be set back to false.
    /*****
    // This loop redirects any transitions to states that have been
    // merged. If a state has been merged, newindex will point to
    // the state it has been merged to.
    for (j = 0; j < all_states.size(); j++)
        {
            for (k = (*all_states[j]).size() - 1; k >= 0; k--)
                {
                    x = newindex[(all_states[j])[k].get_sink().value()];
                    while (x != newindex[x])
                        x = newindex[x];
                    (all_states[j])[k].assign(j, (all_states[j])[k].get_label(), x);
                }
        }
    /*****
    // This section of code removes the states that have been
    // merged to another or are unreachable. First, the start
    // state swapped with the state at position 0. Then, the
    // states that will be deleted are swapped with states at the end
    // Swap the old and new start states
    tempb = merged[0];
    merged[0] = merged[n];
    merged[n] = tempb;
    temp = level[0];
    level[0] = level[n];
    level[n] = temp;
    temp_state = all_states[0];
    all_states[0] = all_states[n];
    all_states[n] = temp_state;
    start = 0;
    for (k = n+1; k < gap.size(); k++) // Swap gap[k][0] with gap[k][n]
        {
            x = (*gap[k])[0];
            (*gap[k])[0] = (*gap[k])[n];
            (*gap[k])[n] = x;
        }
    for (k = 1; k < (*gap[n]).size(); k++) // Swap gap[n][k] with gap[k][0]
        {
            x = (*gap[k])[0];
            (*gap[k])[0] = (*gap[n])[k];
            (*gap[n])[k] = x;
        }
    // After the start state and state 0 have been swapped, transitions
    // pointing to these states will need to be modified.
    for (j = 0; j < all_states.size(); j++)
        {
            for (k = (*all_states[j]).size() - 1; k >= 0; k--)
                {
                    x = (*all_states[j])[k].get_sink().value();
                    if (x == n)
                        x = 0;
                    else if (x == 0)
                        x = n;
                    (all_states[j])[k].assign(j, (all_states[j])[k].get_label(), x);
                }
        }
    for (j = 0; j < newindex.size(); j++)

```

```

    newindex[j] = j;
    // j contains the location of the last state to be kept
    j = all_states.size() - 1;
    // n contains the location of the first state to be deleted
    n = 1;
    // x contains the new size of the automata
    x = all_states.size();
    while(n < j)
    {
        // Find the first state to be deleted
        while ((merged[n] == false) && (level[n] != -1) && (n < j))
            n++;
        // Find the last state to be kept and update the new
        // size of the automata
        while ((merged[j] == true) || (level[j] == -1) && (j > 0))
        {
            x = j;
            j--;
        }
        // Swap the state to be deleted with the state to be kept
        // and delete the unneeded state.
        if (n < j)
        {
            x = j;
            // Swap information about the state
            // Note that the information for the state which will be
            // deleted is simply lost.
            merged[n] = merged[j];
            level[n] = level[j];
            // Swap the state itself
            temp_state = all_states[j];
            all_states[j] = all_states[n];
            all_states[n] = temp_state;
            newindex[j] = n; // newindex keeps track of which states
                            // were swapped so that the transitions
                            // pointing to swapped states may be updated.
            // Update the gap function
            for (k = 0; k < n; k++)
                (*gap[n])[k] = (*gap[j])[k];
            k = n+1;
            while(k < j)
            {
                (*gap[k])[n] = (*gap[j])[k];
                k++;
            }
            j--;
            n++;
        }
    }
    for (j = all_states.size() - 1; j >= x; j--)
    {
        // Since some states have been deleted, arrays
        // which have information about these states need
        // to be deleted.
        merged.remove(j);
        level.remove(j);
        delete all_states[j];
        all_states.remove(j);
        delete gap[j];
        gap.remove(j);
    }
    /*****
    // After all of the states have been swapped, transitions
    // pointing to the swapped states need to be modified.
    // newindex[x] contains the location of the state swapped
    // with x.
    for (j = 0; j < all_states.size(); j++)
    {
        for (k = (*all_states[j]).size() - 1; k >= 0; k--)
        {
            x = newindex[(all_states[j])[k].get_sink().value()];
            while (x != newindex[x])
                x = newindex[x];
            (all_states[j])[k].assign(j, (all_states[j])[k].get_label(), x);
        }
    }
    // newindex may now return to the same size as the automata
    while(newindex.size() > all_states.size())
        newindex.remove(newindex.size()-1);
    wc++;
} // End main loop
// Return the cover automata to Grail's default format
fm<Label>::start_states[0] = start;
unattach();
fm<Label>::arcs.sort();
// Free all dynamically allocated memory
while (gap.size() > 0)
{
    delete gap[gap.size()-1];
    gap.remove(gap.size()-1);
}
}

```