Operation L-INSERT on Factor Automaton*

Bořivoj Melichar and Milan Šimánek

Department of Computer Science & Engineering
Faculty of Electrical Engineering
Czech Technical University Prague

e-mail: melichar@fel.cvut.cz, simanek@fel.cvut.cz

Abstract. The factor automaton is used for time-optimal searching for substrings in text. In general, if the text is changed the new factor automaton has to be constructed. When the text change is simple enough we can change the original factor automaton to reflect the changes of the text and save the time of the new factor automaton construction.

This paper deals with operation L-INSERT and describes the algorithm modifying the factor automaton when a new symbol is prepended to the text. This algorithm can be also used for on-line backward construction of factor automaton.

Keywords: factor automaton, DAWG, operation on factor automaton, construction of factor automaton, finite automaton.

1 Introduction

The factor automaton is a finite automaton accepting the set of all factors (substrings) of the given text (string) T. The factor automaton can be constructed for arbitrary text by one of the common construction algorithms. The time complexity of the construction is linear to the size of the text T, while pattern matching for pattern P is linear to the size of the pattern P and is independent of the size of text T. So, in the most common case the factor automaton is once constructed and many time used for pattern matching. However, when we change the text T the factor automaton must be dropped and new factor automaton has to be constructed.

If the changes in the text are simple enough then we can find an algorithm modifying the original factor automaton according text T. The time complexity of this algorithm is often better then the complete construction of the new factor automaton for the changed text.

A nice example of such algorithm is the APPEND algorithm described in [1, Chapter 6.3], which can modify given factor automaton when a new symbol is appended to the text T. The authors use this algorithm as a part of their on-line factor automaton construction algorithm for text $T = t_1 t_2 \cdots t_n$: they start with one-node factor

*This research has been partialy supported by the Ministry of Education, Youth, and Sports of the Czech Republic under research program No. J04/98:212300014 (Research in the area of information technologies and cummunications) and by Grant Agency of Czech Republic grant No. 201/01/1433.

automaton for empty text ε and compute successively factor automata for texts t_1, t_1t_2, $t_1t_2t_3$, $\cdots$, $t_1t_2\cdots t_n$.

Another known factor automaton modifying algorithm is the L-DELETE algorithm [2]. It can make desired changes to the factor automaton when the text T is reduced by deleting the leftmost symbol. The L-DELETE algorithm can be used in conjunction with the APPEND algorithm to implement fast substring matching in sliding window data compression method.

This paper describes an L-INSERT algorithm modifying the factor automaton when the text T is prepended by a new symbol. Like the APPEND algorithm, this algorithm can also be used for the construction of the factor automaton. The well-known construction using operation APPEND creates the factor automaton by appending symbols of the text T from left to right. On the contrary, the construction based on L-INSERT creates the factor automaton starting with the rightmost symbol to the left.

2 Basic Definitions

The factor automaton for text T is defined as a finite automaton M accepting the language $L(M) = Fac(T)$ of all factors of T. There is an infinite number of such automata, hence we select one with very regular structure of its transition diagram (Figure 1). All its states are both initial and final.

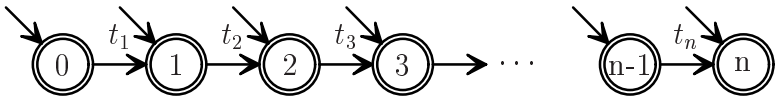


Figure 1: Canonical nondeterministic factor automaton (CNFA)

Definition 2.1 — Canonical nondeterministic factor automaton (CNFA) Canonical nondeterministic factor automaton CNFA for text $T = t_1t_2t_3\cdots t_n$ is a nondeterministic finite automaton $M = (Q, \mathcal{A}, \delta, I, F)$ which satisfies:

1. $Q = \{q_0, q_1, q_2, \cdots q_n\}$
2. $\forall q_i \in Q, a \in \mathcal{A}: \quad \delta(q_i, a) = \begin{cases} \{q_{i+1}\} & \forall i < n, a = t_{i+1} \\ \emptyset & \text{in other cases} \end{cases}$
3. $I = Q$
4. $F = Q$

We cannot directly use CNFA because of a nondeterminism. Each nondeterministic finite automaton can be transformed to deterministic one accepting the same language. The transformation can be done by subset construction [3]. We use the variant of the transformation which does not insert inaccesible states into the resulting DFA [4, algorithm 3.6] and we denote it as the standard determinization method.

The standard determinization method is based on the following state-sets construction: For each nondeterministic finite automaton $M = (Q, \mathcal{A}, \delta, I, F)$ we can

find a deterministic finite automaton $\hat{M} = (\hat{Q}, \mathcal{A}, \hat{\delta}, \hat{q}_0, \hat{F})$ accepting the same language satisfying the following conditions:

- $\hat{Q} \subseteq \mathcal{P}(Q)$ such that $\hat{Q} = \{\hat{q} : \hat{q} = \delta^*(I, w); w \in \mathcal{A}^*\}$
- $\hat{\delta}$ is a mapping $\hat{\delta} : \hat{Q} \times \mathcal{A} \mapsto \hat{Q}$
  $\forall \hat{q} \in Q, a \in \mathcal{A}: \quad \hat{\delta}(\hat{q}, a) = \bigcup_{q \in \hat{q}} \delta(q, a)$,
- $\hat{q}_0 \in \hat{Q} \quad \hat{q}_0 = I$,
- $\hat{F} \subset \hat{Q} \quad \hat{F} = \{\hat{q} \in \hat{Q} : \hat{q} \cap F \neq \emptyset\}$.

We use the hat accent to denote deterministic automaton, its states and transition function. States of CDFA are sets of CNFA. Note, that that CDFA contains only reachable states.

Definition 2.2 — Canonical deterministic factor automaton (CDFA) Canonical deterministic factor automaton CDFA for text T is a deterministic automaton given as the result of the standard determinization of the canonical nondeterministic factor automaton for the same text T.

The L-INSERT algorithm modifying CNFA is very simple (it just inserts a new state and one transition). We use that algorithm and the standard determinization to find L-INSERT algorithm modifying CDFA. To keep the relationship between states of CNFA and CDFA automata we use several adjacent data structures.

3 Adjacent Data Structures

To enable efficient algorithm modifying CDFA we extend CDFA by following additional information:

- suffix links,
- text pointers,
- in-degree of nodes.

3.1 Suffix Links

Each state $\hat{q}$ of the CDFA represents a set of active states of the CNFA – after accepting any string w the active state $\hat{q}_w = \hat{\delta}^*(\hat{q}_0, w)$ of CDFA represents a set of active states $Q_w = \delta^*(I, w)$ of CNFA, formally $\hat{q}_w = Q_w$.

Lemma 3.1 If two states $\hat{q}_u, \hat{q}_w \in \hat{Q}$ have nonempty intersection, $\hat{q}_u \cap \hat{q}_w \neq \emptyset$, then one of them is a subset of the other ($\hat{q}_w \subset \hat{q}_u$).

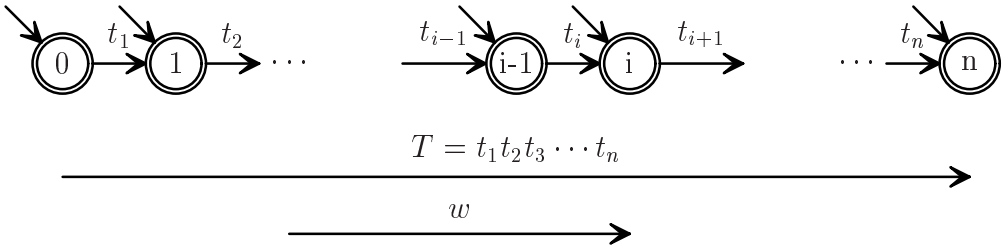


Figure 2: If state $\hat{q}_w = \hat{\delta}^*(\hat{q}_0, w)$ contains a state q_i then string w ends at position i

Proof:

If both two states $\hat{q}_u$ and $\hat{q}_w$ contain state q_i then both represent the CNFA active state q_i. Because of very regular structure of CNFA the state q_i becomes active only if the accepted string is a factor of the text T ending at position i (see Figure 2). It means that both strings u and w (leading to states $\hat{q}_u$ and $\hat{q}_w$) are factors of the text T ending on the same position i. Therefore one of them must be a suffix of the other (Figure 3). Let

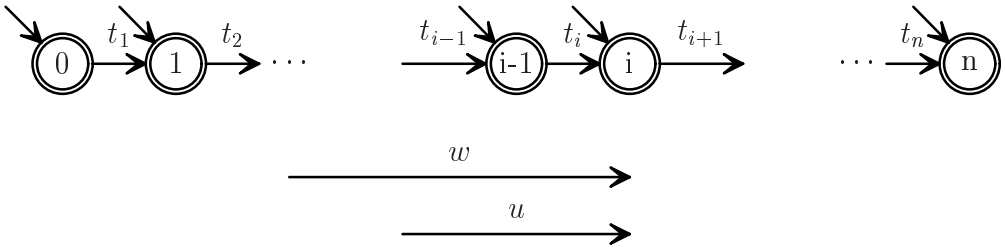


Figure 3: Strings u and w end in the same position.

u be a suffix of w. The state $\hat{q}_w$ represents states $\hat{q}_w = \{q_{j_1}, q_{j_2}, q_{j_3}, \cdots\}$ where j_k are ending positions of all occurrences of the string w in the text. The string u is a suffix of w so that it occurs at least on the same ending positions, therefore $\hat{q}_w \subset \hat{q}_u$ (Figure 4). ▫

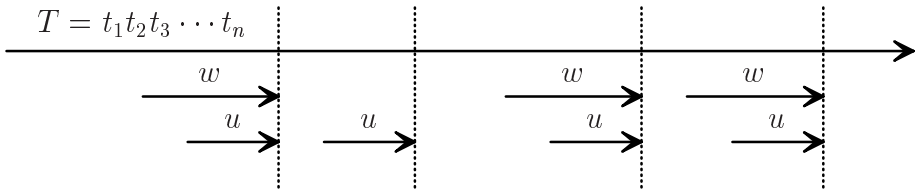
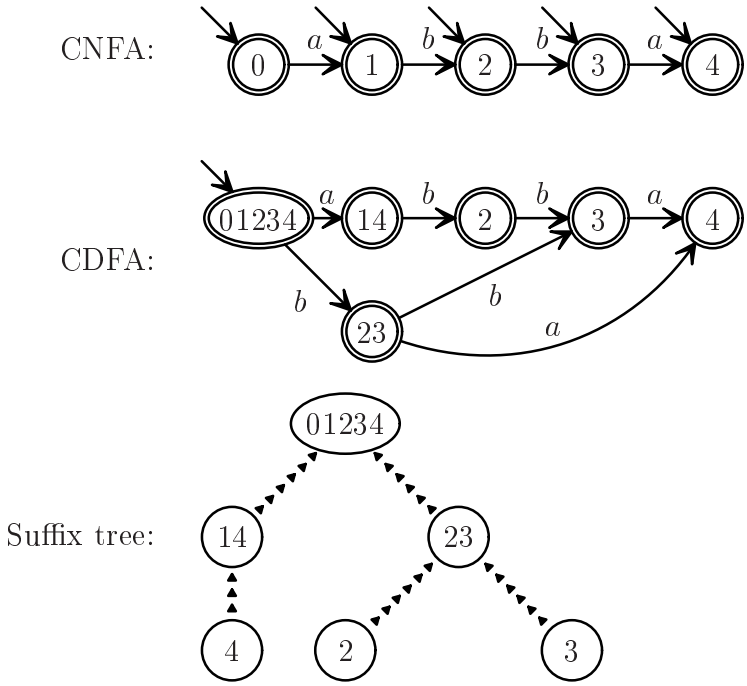


Figure 4: String u ends at least on the same ending positions as string w.

From the lemma above, any pair of CDFA states containing any common CNFA state q_i are ordered by set inclusion. Therefore all CDFA states representing any CNFA state q_i create ordered set (chain of states). The initial state $\hat{q}_0 = I = Q = \{q_0, q_1, \cdots q_n\}$ containing all CNFA states is a superset of any set of CNFA states and it is the biggest set of any chain of sets. We can say that all states of CDFA are

ordered in a rooted tree with the root $\hat{q}_0$. The common name for such tree is **suffix tree**.



Positions in text T: ${}_0a_1b_2b_3a_4$

state	words	ending pos.
$\hat{q}_{\{q_0,q_1,q_2,q_3,q_4\}}$	ε	0, 1, 2, 3, 4
$\hat{q}_{\{q_1,q_4\}}$	a	1, 4
$\hat{q}_{\{q_2,q_3\}}$	b	2, 3
$\hat{q}_{\{q_2\}}$	ab	2
$\hat{q}_{\{q_3\}}$	bb abb	3
$\hat{q}_{\{q_4\}}$	ba bba $abba$	4

Figure 5: An example of suffix tree for $T = abba$

This suffix tree (as a data structure) can be implemented by pointers from each state $\hat{q} \in \hat{Q}$ to its parent $\hat{p}$ in the suffix tree. We call such pointer as **suffix link** and denote $\hat{p} = suf[\hat{q}]$. The state $suf^k[\hat{q}]$ means $k^{\underline{th}}$ iteration of suffix link and $suf^*[\hat{q}]$ (transitive closure) denotes a set of all iterations of suffix link of the state $\hat{q}$.

$$suf^*[\hat{q}] = \{\hat{q}, suf[\hat{q}], suf^2[\hat{q}], suf^3[\hat{q}], \cdots\}$$

Lemma 3.2 If two nonequal states $\hat{p}, \hat{q} \in \hat{Q}$ differ by a one state $q \in Q$ i.e. $\hat{p} = \hat{q} \cup \{q\}$ then there exists a direct suffix link between them: $\hat{p} = suf[\hat{q}]$.

Proof:

Any two states $\hat{p}, \hat{q} \in \hat{Q}$ where $\hat{q}$ is a proper subset of $\hat{p}$ ($\emptyset \subset \hat{q} \subset \hat{p}$) are connected by a suffix link iff there does not exist another state r such that $\hat{q} \subset \hat{r} \subset \hat{p}$. As states $\hat{p}$ and $\hat{q}$ differ only by one state, no such state $\hat{r}$ may exist. □

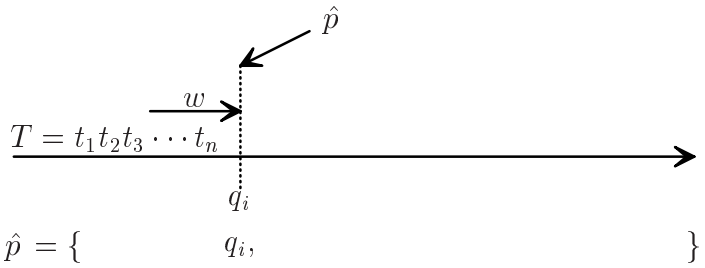


Figure 6: The state $\hat{p}$ has no incoming suffix link iff it contain only one state

Lemma 3.3 State $\hat{p} \in \hat{Q}$ has no incoming suffix link if and only if the set $\hat{q}$ contains exactly one state $q \in Q$.

Proof:

We divide the proof of equivalence to proofs of the both implications. The proof of the first implication (the state $\hat{p}$ has no incoming suffix link $\Longrightarrow$ the set $\hat{p}$ contains only one state) follows from this contradiction:

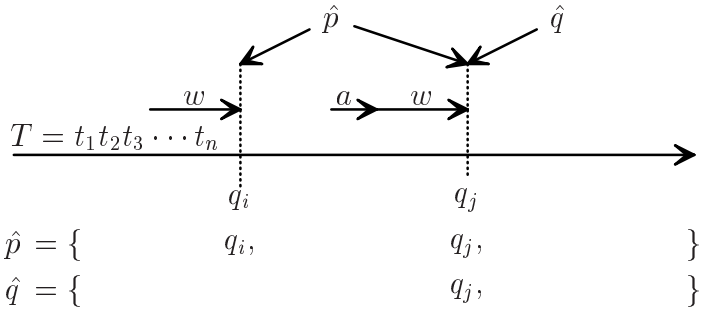


Figure 7: If the state $\hat{p}$ contains two states then it has incoming suffix link.

If the set $\hat{p}$ would contain more than one state (see Figure 7) then there would exist the longest factor w of the text T, which would end at ending positions represented by members of $\hat{p}$. Not all occurrences of string w are preceded by the same symbol (because w is the longest string with these endings) and therefore there would exist a string aw which is a factor of the text T and would end at positions $\hat{q}$ where $\hat{q} \subset \hat{p}$. Due to this inclusion both states $\hat{p}$ and $\hat{q}$ would share the same branch of suffix tree which would lead from $\hat{q}$ to $\hat{p}$. The state $\hat{p}$ would have at least one incoming suffix link, which gives the contradiction.

The second part, the proof of backward implication (the set $\hat{p}$ contains only one state $\Longrightarrow$ the state $\hat{p}$ has no incoming suffix link) is trivial because a suffix link can lead only from a subset to a superset and a set with just only one state has no regular subsets. □

Lemma 3.4 If a state $\hat{p} \in \hat{Q}$ has just one incoming suffix link and w is the longest string leading to this state $\hat{p} = \hat{\delta}^*(\hat{q}_0, w)$ (see Figure 8) then

a) there are at least two occurrences of the string w in the text T,

b) the string w is a prefix of the text T,

c) all occurrences of w in T except the very first one (the prefix of T) are preceded by the same symbol.

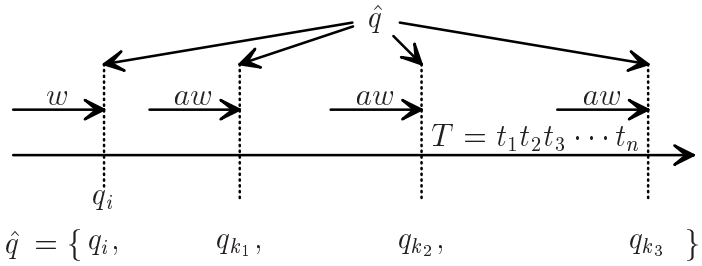


Figure 8: The only one incoming suffix link leads to a state $\hat{p}$.

Proof:

The proof of part a) follows from the Lemma 3.3.

There are no couple of occurrences of string w following two different symbols. If two strings aw and bw (where $a \neq b$) would occur in text T then both states $\hat{q}_{aw}$ and $\hat{q}_{bw}$ would be disjunct subsets of $\hat{p}$ and their suffix links would lead to state $\hat{p}$. At least one occurrence of w must not be preceded by the same symbol as others because w is the longest string leading to state $\hat{p}$. Therefore w occurs at the beginning of T and all next occurrences are preceded by the same symbol. w is a prefix of T. This proves parts b) and c). □

Lemma 3.5 If a suffix link $suf[\hat{q}] = \hat{p}$ is the only suffix link leading to state $\hat{p}$ then set $\hat{p}$ is larger then $\hat{q}$ by just one state q_i (i.e. $\hat{p} = \{q_i\} \cup \hat{q}$).

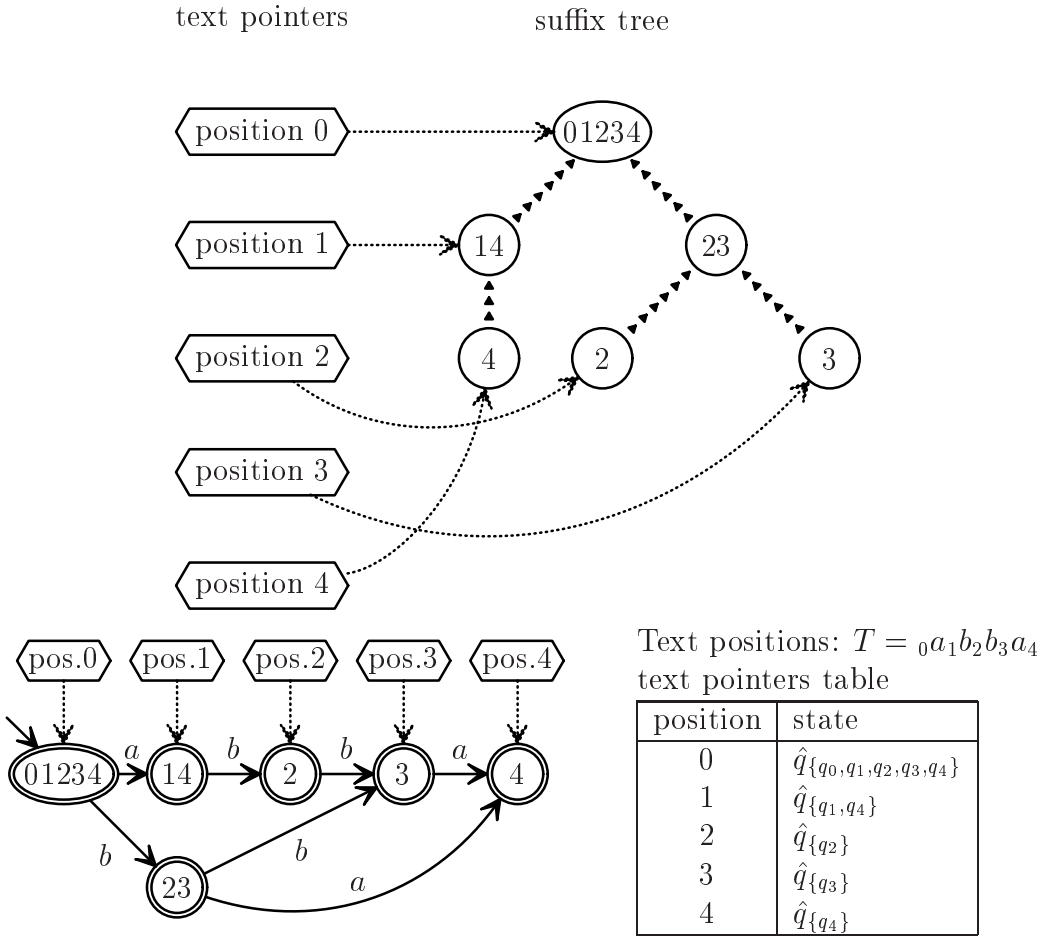
Proof:

Let w be a string leading to the state $\hat{p} = \hat{\delta}^*(\hat{q}_0, w)$ (see Figure 8). Due to Lemma 3.4, string w is a prefix of the text T and all other occurrences of w in the text T are preceded by the same symbol a. The string aw occurs at the same ending positions as string w except the very first one (w is a prefix of T). We can divide the set $\hat{p}$ into the first occurrence (the state q_i) and the rest (occurrences of aw): $\hat{p} = \{q_i\} \cup \hat{\delta}^*(\hat{q}_0,\text{aw})$. Due to Lemma 3.2 it holds $\hat{p} = suf[\hat{\delta}^*(\hat{q}_0,\text{aw})]$. There is only one suffix link leading to $\hat{p}$ so that states $\hat{\delta}^*(\hat{q}_0,\text{aw})$ and $\hat{q}$ are identical and we can write $\hat{p} = \{q_i\} \cup \hat{q}$.

□

3.2 Text Pointers

Most of algorithms operating on factor automaton need to resolve which states of CDFA represent given state q of CNFA. Since all relevant CDFA states contain q they create a separate branch in the suffix tree. We can store only the starting state of the branch and continue over the suffix tree to its root. Text pointers is a data structure which keeps the information about the starting state. It can be implemented as an array $TextPos[i]$ of CDFA states indexed by position i in text. In factor automata it holds $TextPos[i] = \hat{\delta}^*(\hat{q}_0, t_1 t_2 \cdots t_i)$. An example of text pointers array for $T = abba$ is on Figure 9.



Text positions: $T = {}_0a_1b_2b_3a_4$
text pointers table

position	state
0	$\hat{q}_{\{q_0,q_1,q_2,q_3,q_4\}}$
1	$\hat{q}_{\{q_1,q_4\}}$
2	$\hat{q}_{\{q_2\}}$
3	$\hat{q}_{\{q_3\}}$
4	$\hat{q}_{\{q_4\}}$

Figure 9: An example of the suffix tree and the automaton with text pointers for $T = abba$.

Note that the number of states is often larger then the number of positions in the text. Therefore, there exist states which are not the value of any $TextPos$. An example of that is on Figure 9. Although the state $\hat{q}_{\{q_2,q_3\}}$ represents ending positions 2 and 3 for string b, it is neither a value of $TextPos[2]$ nor $TextPos[3]$. We can get all states representing the ending position 2 by inspecting the whole branch of suffix tree (a sequence of suffix links) from the state $\hat{q}_{\{q_2\}} = TextPos[2]$.

3.3 Node In-degree

We use the number of transitions leading to this state (incoming transitions) as a reference counter for detecting unreachable states. If the automaton has unreachable states then one of them must have in-degree equal to zero because the CDFA has no loops. After its removing it holds that either another unreachable state becomes zero in-degree or we are sure there are no unreachable states in the automaton.

3.4 Operation L-INSERT

The canonical nondeterministic factor automata (CNFA) for the texts $T = t_1t_2t_3 \cdots t_n$ and $aT = at_1t_2t_3 \cdots t_n$ are shown on the Figure 10.

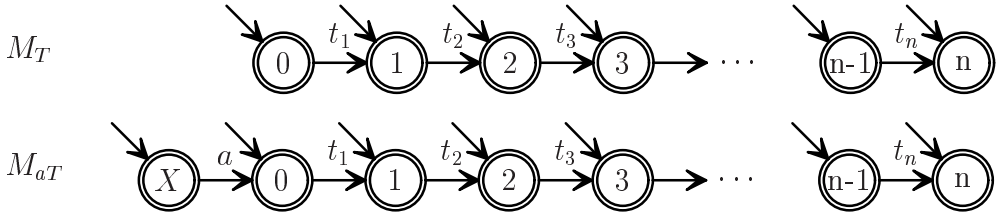


Figure 10: The change in CNFA when a new symbol is prepended.

The operation L-INSERT creates a new state q_X, which is both initial and final and a new transition from the state q_X into the state q_0.

The algorithm modifying CDFA follows from the relationship between nondeterministic and deterministic factor automaton.

When the new initial state q_X is created, CDFA's initial state $\hat{q}_0$ – see Figure 11 (step 1) – is changed to the new state $\hat{q}'_0 = \hat{q}_0 \cup \{q_X\}$. The outgoing transitions from this state are still the same as from $\hat{q}_0$ (step 2). Now, we create a new transition in CNFA leading from q_X to q_0 for symbol a. In the CDFA, we should redirect the transition leading from $\hat{q}'_0$ labeled by a symbol a to another state which contains similar set of states extended by the state q_0, because $q_0 = \delta(q_X, a)$ is the new transition (step 3).

The algorithm is based on the recursive function `GetExtendedState`$(\hat{q}, i)$, which takes the set of states $\hat{q}$ and integer i as arguments, and finds a state $\hat{q}' = \hat{q} \cup \{q_i\}$. If there is no such state in the automaton, it is created by the function. The value of the function is the state $\hat{q}'$ (Figure 12).

Using this function the whole algorithm can be written in five steps:

1. create a new state $\hat{q}'_0$ with the same outgoing transitions as $\hat{q}_0$,
2. get the old target of the first transition: $\hat{q} = \hat{\delta}(\hat{q}'_0, a)$,
3. compute new state for that transition: $\hat{q}' =$ `GetExtendedState`$(\hat{q}, 0)$,
4. redirect the transition: $\hat{\delta}(\hat{q}'_0, a) = \hat{q}'$,
5. change the initial state to $\hat{q}'_0$.

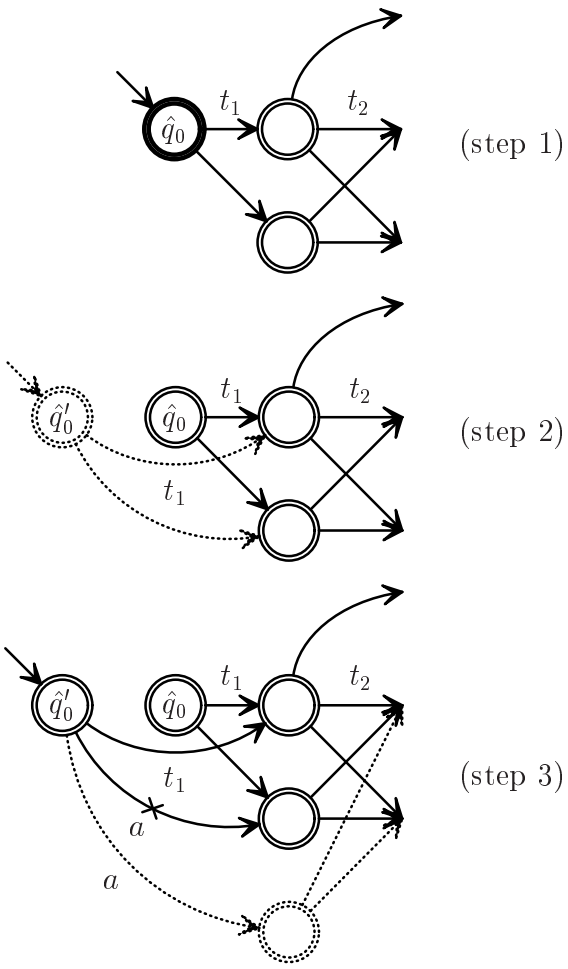


Figure 11: The change in CDFA when symbol a is inserted.

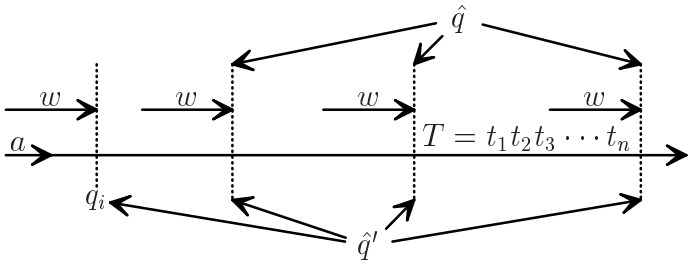


Figure 12: The state $\hat{q}'$ contains state q_i and all states from $\hat{q}$

We assume any unreachable state is removed as soon as it looses the last incoming transition (or the last reference).

Let us concern the function `GetExtendedState`$(\hat{q}, i)$. It assumes that the string $w = at_1t_2t_3\cdots t_i$ leads to the state $\hat{q}$ (i.e. $\hat{q} = \hat{\delta}^*(\hat{q}_0, w)$). It is the shortest string leading to this state because the text shorter by the first symbol a would be a prefix of T an would occur in advance at ending position i.

Note that the string $w = at_1t_2t_3\cdots t_i$ may not be a factor of the text T. In this case the state $\hat{q}$ may be $\hat{q} = \{\} = \emptyset$. In such case, the solution is a state $\hat{q}' = \{q_i\}$. Of course, this state may or may not be present in the current automaton. We can find it by inspecting the text pointer at position i. The value of $TextPos[i]$ may be the required state $\hat{q}' = \{q_i\}$ or its superset. According to Lemma 3.3: if there is no suffix link leading to this state then it contains only one CNFA state $\{q_i\}$ and it is the result value of the function `GetExtendedState` (Figure 13). If there exists a

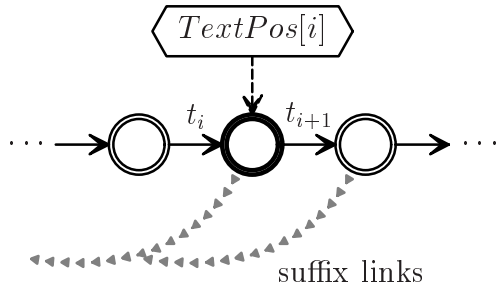


Figure 13: The focused state has no incoming suffix links therefore it contains only one state q_i

suffix link leading to this state then we must create a new state $\hat{q}' = \{q_i\}$ and set its outgoing transitions. In this case the state $\hat{q}'$ will have only one outgoing transition for the symbol t_i leading to state $\{q_{i+1}\}$ (which can be obtained by recursive calling the function `GetExtendedState`$(nil, i+1)$). In addition, we should set up the suffix link of this state to lead to $TextPos[i]$ and update $TextPos[i]$ to new value – state $\hat{q}'$. (See Figure 14).

Now, we concern the case when $\hat{q}$ is an already existing state of CDFA. The function `GetExtendedState` should locate the state representing the set $\hat{q} \cup \{q_i\}$. If there is no such state, it should be created. Due to the Lemma 3.2 if there exists such state it must be the target of the suffix link from state $\hat{q}$. But the suffix parent $\hat{p} = suf(\hat{q})$ of the state $\hat{q}$ may not be the required state in any case, of course. We can test it by inspecting the number of suffix links leading to it. There are two disjunct cases:

- only one suffix link leads to state $\hat{p}$,
- the state $\hat{p}$ is a target of more suffix links.

At first we assume the suffix link from the state $\hat{q}$ to the state $\hat{p}$ is the only link leading to $\hat{p}$ (Figure 15). As the string $w = at_1t_2t_3\cdots t_i$ is the shortest string leading to $\hat{q}$ then the first suffix – string $u = t_1t_2t_3\cdots t_i$ leads to state $suf(\hat{q}) = \hat{p}$. We are sure that string $t_1t_2t_3\cdots t_i$ occurs at position i and therefore $\hat{p}$ contain the required

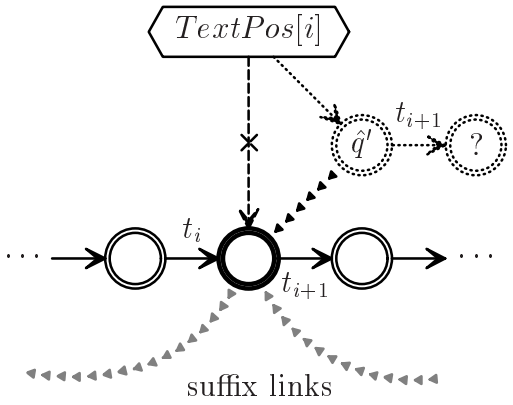


Figure 14: If any suffix link leads to the state found by $TextPtr[i]$ then we have to create a new state $\hat{q}'$, connect its suffix link, outgoing transition and redirect $TextPtr[i]$

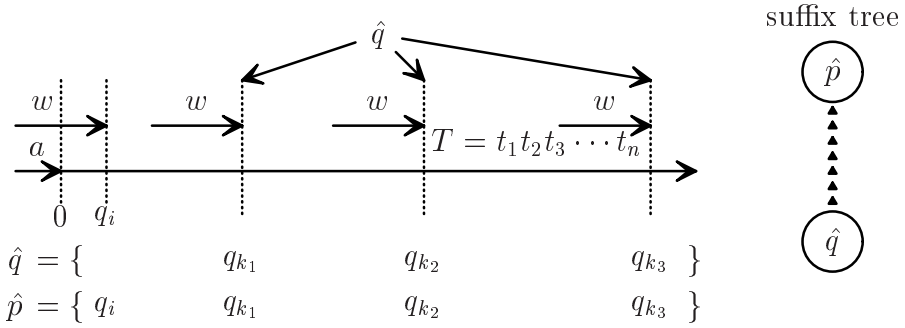


Figure 15: $\hat{q} \mapsto \hat{p}$ is the only suffix link leading to $\hat{p}$ therefore $\hat{p} = \hat{q} \cup \{q_i\} = \hat{p}'$

state $\hat{q}_i$. On the other side, the state $\hat{p}$ does not contain any other state then $\{q_i\}$ or $\hat{q}$ (see Lemma 3.5) therefore state $\hat{p}$ is the value of the function `GetExtendedState` .

Now, assume there exist at least two suffix links leading to the state $\hat{p}$. One of them is the link from $\hat{q}$ and let another one lead from a state $\hat{q}_q$ (Figure 16). The

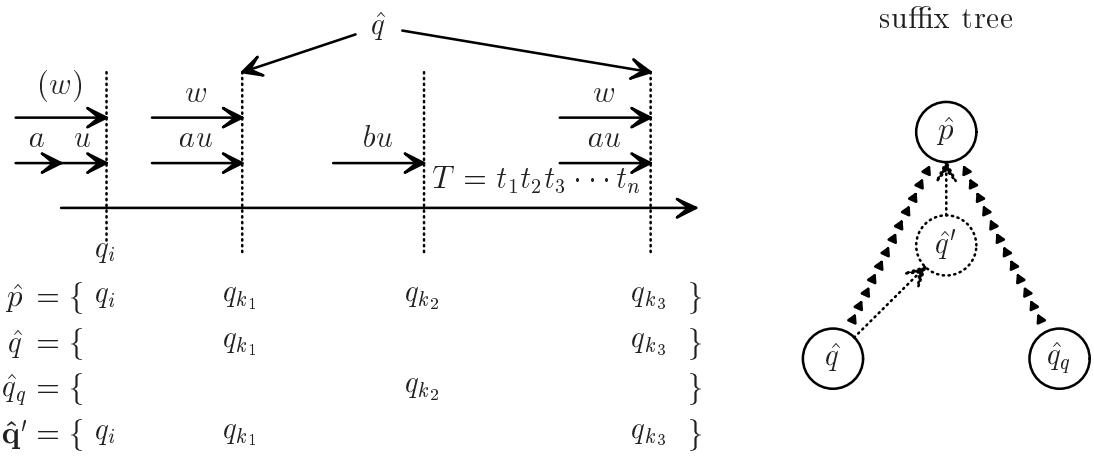


Figure 16: If the state $\hat{p}$ receives more suffix links then it is unusable. A new state $\hat{q}'$ has to be created.

sets $\hat{q}$ and $\hat{q}_q$ are disjunct because they are in the different branches of the suffix tree. The state $\hat{p}$ is the superset of both sets. Therefore, the set $\hat{p}$ contains more states then $\hat{q} \cup \{q_i\}$ and will be unusable for us. The resulting state is still not in the set of states of the automaton and we have to create it.

We create a new state $\hat{q}'$ which should represent the set $\hat{q} \cup \{q_i\}$ and therefore it inherits the same outgoing transition as $\hat{q}$. However the transition for the symbol t_{i+1} should be redirected to the state (the set of CNFA states) extended by the state q_{i+1}. We can lookup this state using the function `GetExtendedState` in recursion. The redirection is made by assigning $\hat{\delta}(\hat{q}', t_{i+1}) =$ `GetExtendedState`$(\hat{\delta}(\hat{q}, i), i+1)$. Finally, we should update suffix links. The new state $\hat{q}'$ is a subset of $\hat{p}$ and a superset of $\hat{q}$ therefore we include it between states $\hat{p}$ and $\hat{q}$: $suf[\hat{q}'] = \hat{p}$ and $suf[\hat{q}] = \hat{q}'$.

Algorithm 3.1 — Operation L-INSERT using function `GetExtendedState`

INPUT:	CDFA automaton $\hat{M} = (\hat{Q}, \mathcal{A}, \hat{\delta}, \hat{q}_0, \hat{F})$ with suffix links, text T and text pointers
	symbol a
OUTPUT:	CDFA automaton $\hat{M}$ with suffix links, text T and text pointers
LOCAL:	integer n
	state $\hat{p}$
	state $\hat{q}'$
	state $\hat{q}'_0$
	state $\hat{t}$
REQUIRE:	$\hat{M}$ accepts factors of $T = t_1t_2t_3\cdots t_n$
ENSURE:	$\hat{M}$ will accept factors of $T = at_1t_2t_3\cdots t_n$

1: function `GetExtendedState`(*state* $\hat{q}$, *integer* i)
2: **if** ($\hat{q}$ == nil) **then**

3: $\hat{t} = TextPtr[i]$
4: $n = |suf^{-1}(\hat{t})|$ { *the number of suffix links incomming to* $\hat{t}$ }
5: **if** $(n == 0)$ **then**
6: $\hat{q}' = \hat{t}$
7: return $\hat{q}'$
8: **else**
9: $\hat{q}'$ =new state
10: $\hat{\delta}(\hat{q}', a) =$ `GetExtendedState`$(nil, i+1)$
11: $suf[\hat{q}'] = \hat{t}$
12: return $\hat{q}'$
13: **end if**
14: **else**
15: $\hat{p} = suf[\hat{q}]$
16: $n = |suf^{-1}(\hat{p})|$
17: **if** $(n == 1)$ **then**
18: $\hat{q}' = \hat{p}$
19: return $\hat{q}'$
20: **else**
21: $\hat{q}' = duplicate(\hat{q})$
22: $\hat{\delta}(\hat{q}', t_{i+1}) =$ `GetExtendedState`$(\hat{\delta}(\hat{q}, t_{i+1}), i+1)$
23: $suf[\hat{q}'] = \hat{p}$
24: $suf[\hat{q}] = \hat{q}'$
25: return $\hat{q}'$
26: **end if**
27: **end if**
28: endfunction
29: $\hat{q}'_0 = duplicate(\hat{q}_0)$
30: $\hat{\delta}(\hat{q}'_0, a) =$ `GetExtendedState`$(\hat{\delta}(\hat{q}_0, a), 0)$
31: $SetInitialState(\hat{q}'_0)$

4 Efficiency of the Algorithm

4.1 Time Complexity

The best case from the time complexity point of view appears when the new inserted symbol a is equal to each symbol in the text: $T = a^n$. In such case, the recursive function `GetExtendedState` is called only once. Neither this function nor the main algorithm contain loop, therefore the time complexity is constant $O(1)$ – independent on the size of the text T.

The worst case occurs if all symbols in text T are the same but different from the new inserted symbol a: $T = b^n$. In such case, the original automaton has $n+1$ states and the new automaton will have $2n-1$ states, and so the algorithm have to create $n-2$ states and it has asymptotically time complexity linear $O(n)$ with respect to the size of the text T.

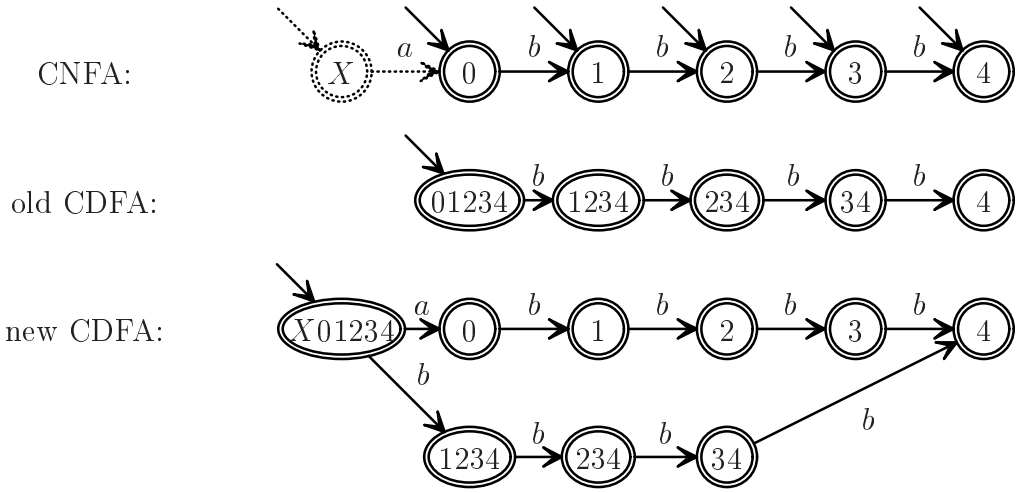


Figure 17: The worst case

4.2 Space Complexity

The algorithm requires extra space for following data structures:

- text pointers,
- suffix links,
- states,
- transitions,
- stack for recursion.

Text pointers is an array indexed by the position in text T. The size of the array is linear to the size of text T. Text pointers are more useful for other operations with factor automata. In the case of L-INSERT algorithm, text pointers can be substituted by text T, because we need successively the values $TextPos[0], TextPos[1], TextPos[2], ...$ and $TextPos[i] = \hat{\delta}(TextPos[i-1], t_i)$ while $TextPos[0] = \hat{q}_0$. So that we could compute the values of $TextPos$ during recursion of the function `GetExtendedState`.

Both suffix links and states take the same space complexity because there is just one outgoing suffix link per a state. The number of states is at most $2n$ (proved in [1]).

The number of transitions in the factor automaton is less than $3n$ (proved in [1]).

The size of the stack required for the recursion is limited by the number of recursive calls. As a new states is created before any recursive call, the total number of recursive calls is limited by the number of inserted states. Moreover, the recursion function `GetExtendedState` can be transformed into an iteration loop without a need of an extra data space.

As the all data structures require space at most linear to the size of the automaton, we can say the L-INSERT algorithm is space-linear.

5 Conclusion

This paper deals with the factor automaton and its modifications when the text often changes. We discuss several operations on the text and cite algorithms reflecting these operations into the factor automaton. Moreover we describe some adjacent data structures (suffix links and text pointers) used in algorithms modifying the factor automaton. We present a new algorithm of operation L-INSERT. The algorithm can efficiently modify a factor automaton when a new symbol is inserted before the first symbol of the text. This algorithm can be also used for on-line backward construction of the factor automata. This means that the text grows from right to left while constructing the automaton. Finally, the time and space complexity of the L-INSERT algorithm is also discussed.

References

[1] M. Crochemore and W. Rytter. *Text algorithms*. Oxford University Press, 1994.

[2] M. Šimánek. The factor automaton. In J. Holub and M. Šimánek, editors, *Proceedings of the Prague Stringologic Club Workshop '98*, pages 102–106, Czech Technical University, Prague, Czech Republic, 1998. Collaborative Report DC–98–06.

[3] J. E. Hopcroft and J. D. Ullman. *Introduction to automata, languages and computations*. Addison-Wesley, Reading, MA, 1979.

[4] J. Holub. Simulation of nondeterministic finite automata in approximate string and sequence matching. Technical Report DC–98–04, Department of Computer Science and Engineering, Czech Technical University, Prague, Czech Republic, 1998.