Incremental Construction and Maintenance of Minimal Finite-State Automata

Rafael C. Carrasco*
Universitat d'Alacant

Mikel L. Forcada†
Universitat d'Alacant

Daciuk et al. [Computational Linguistics **26***:1, 3–16 (2000)] describe a method for constructing incrementally minimal, deterministic, acyclic finite-state automata (dictionaries) from sets of strings. But acyclic finite-state automata have limitations: for instance, if one wants a linguistic application to accept all possible integer numbers or internet addresses, the corresponding finite-state automaton has to be cyclic. In this paper, we describe a simple and equally efficient method to modify any minimal finite-state automaton (be it acyclic or not) so that a string is added to or removed from the language accepted by it; both operations are very important when performing dictionary maintenance and solve the dictionary construction problem addressed by Daciuk et al. as a special case. The algorithms proposed here may be straightforwardly derived from the customary textbook constructions for the intersection and the complementation of finite-state automata; the algorithms exploit the special properties of the automata resulting from the intersection operation when one of the finite-state automata accepts a single string.*

1 Introduction

In a recent paper in this journal, Daciuk et al. (2000) describe two methods for constructing incrementally minimal, deterministic, acyclic finite-state automata (dictionaries) from sets of strings: the first method adds strings in dictionary order and the sec-

* Departament de Llenguatges i Sistemes Informàtics, Universitat d'Alacant, E-03071 Alacant (Spain)
† Departament de Llenguatges i Sistemes Informàtics, Universitat d'Alacant, E-03071 Alacant (Spain)

ond one is for unsorted data. Adding an entry is an important dictionary maintenance operation; but so is removing an entry from the dictionary, for example, if it is found to be incorrect. Since ordering cannot obviously be expected in the removal case, we will focus on the second, more general, problem (a solution for which has already been sketched by Revuz (2000)).

But dictionaries or acyclic finite automata have limitations: for instance, if one wants an application to accept all possible integer numbers or internet addresses, the corresponding finite-state automaton has to be cyclic. In this article, we show a simple and equally efficient method to modify *any* minimal finite-state automaton (be it acyclic or not) so that a string is added to or removed from the language accepted by it. The algorithm may be straightforwardly derived from customary textbook constructions for the intersection and the complementation of finite-state automata; the resulting algorithm solves the dictionary construction problem addressed by Daciuk et al.'s (2000) second algorithm as a special case.

This paper has the following parts: in section 2, we give some necessary mathematical preliminaries; the minimal automata resulting from adding or removing a string are described in detail in section 3; the algorithms are described in section 4; in section 5, one addition and one removal example are explained in detail; and, finally, some closing remarks are given in section 6.

2 Mathematical preliminaries

2.1 Finite-state automata and languages

As in (Daciuk et al., 2000), we will define a deterministic finite-state automaton as $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, $q_0 \in Q$ is the start state, $F \subseteq Q$ is a set of accepting states, Σ is a finite set of symbols called the alphabet, and $\delta : Q \times \Sigma \to Q$ is the next-state mapping. In this paper, we will define δ as a *total* mapping; the corresponding finite-state automaton will be called **complete** (Revuz, 2000). This involves no loss of generality, as any finite-state automaton may be made complete by adding a new **absorption** state $\perp$ to Q, so that all undefined transitions point to it and $\delta(\perp, a) = \perp$ for all $a \in \Sigma$. Using complete finite-state automata is convenient for the theoretical discussion in this paper; real implementations of automata and the corresponding algorithms need not contain an explicit representation of the absorption state and its incoming and outgoing transitions.

For complete finite-state automata, the extended mapping $\delta^* : Q \times \Sigma^* \to Q$ (the extension of δ for strings) is defined simply as

$$\begin{array}{rcl} \delta^*(q, \epsilon) & = & q \\ \delta^*(q, ax) & = & \delta^*(\delta(q, a), x) \end{array} \tag{1}$$

for all $a \in \Sigma$ and $x \in \Sigma^*$, with ϵ the empty or null string. The language accepted by automaton M

$$\mathcal{L}(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}, \tag{2}$$

and the right language of state q

$$\vec{\mathcal{L}}(q) = \{x \in \Sigma^* : \delta^*(q, x) \in F\}, \tag{3}$$

are defined as in (Daciuk et al., 2000).

2.2 Single-string automaton

We find it also convenient to define the (complete) **single-string automaton** for string w, denoted $M_w = (Q_w, \Sigma, \delta_w, q_{0w}, F_w)$, such that $\mathcal{L}(M_w) = \{w\}$. This automaton has $Q_w = \mathrm{Pr}(w) \cup \{\perp_w\}$, where $\mathrm{Pr}(w)$ is the set of all prefixes of w and $\perp_w$ is the absorption state, $F_w = \{w\}$, and $q_{0w} = \epsilon$ (note that non-absorption states in Q_w will be named after the corresponding prefix of w). The next-state function is defined as follows

$$\delta(x, a) = \begin{cases} xa & \text{if } xa \in \mathrm{Pr}(w) \\ \perp_w & \text{otherwise} \end{cases} . \tag{4}$$

Note that the single-string automaton for a string w has $|Q_w| = |w| + 2$ states.

2.3 Operations with finite-state automata

2.3.1 Intersection automaton Given two finite-state automata M_1 and M_2, it is easy to build an automaton M so that $\mathcal{L}(M) = \mathcal{L}(M_1) \cap \mathcal{L}(M_2)$. This construction is found in formal language theory textbooks (Hopcroft and Ullman, 1979, p. 59) and is referred to as *standard* in papers (Karakostas, Viglas, and Lipton, 2000). The (complete) **intersection automaton** has $Q = Q_1 \times Q_2$, $q_0 = (q_{01}, q_{02})$, $F = F_1 \times F_2$, and $\delta((q_1, q_2), a) = (\delta(q_1, a), \delta(q_2, a))$.

2.3.2 Complementary automaton Given a complete finite-state automaton M, it is easy to build its **complementary automaton** $\bar{M}$ so that $\mathcal{L}(\bar{M}) = \Sigma^* - \mathcal{L}(M)$: the only change is the set of final states: $\bar{F} = Q - F$ (Hopcroft and Ullman, 1979, p. 59). In particular, the **complementary single-string automaton** M_{-w} accepting $\Sigma^* - \{w\}$ is identical to M_w except that $F_{-w} = Q - \{w\}$.

2.3.3 Union automaton The above constructions may be easily combined to obtain a construction to build, from two complete automata M_1 and M_2, the (complete) **union automaton** M such that $\mathcal{L}(M) = \mathcal{L}(M_1) \cup \mathcal{L}(M_2)$. It suffices to consider that, for any two languages on Σ^*, $L_1 \cup L_2 = \Sigma^* - (\Sigma^* - L_1) \cap (\Sigma^* - L_2)$. The resulting automaton M is identical to the intersection automaton defined above except for the fact that $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.

3 Adding and removing a string

3.1 Adding a string

Given a (possibly cyclic) minimal complete finite-state automaton M, it is easy to build a new complete automaton M' accepting $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$ by applying the union construct defined above to M and the single-string automaton M_w. The resulting automaton $M' = (Q', \Sigma, \delta', q_0', F')$, which may be very easily minimized (see below) has four kinds of states in Q':

- States of the form $(q, \perp_w)$ with $q \in Q - \{\perp\}$, equivalent to those non-absorption states of M which are not reached by any prefix of w; they will be called **intact** states because they have the same transition structure as their counterparts in M (that is, if $\delta(q, a) = q'$, then $\delta'((q, \perp_w), a) = (q', \perp_w)$), and belong to F' if $q \in F$. As a result, they have exactly the same right languages, $\vec{\mathcal{L}}((q, \perp_w)) = \vec{\mathcal{L}}(q)$, because all of their outgoing transitions go to other intact states; furthermore, each state $(q, \perp_w)$ has a different right language; therefore, no two intact states will ever be merged into one by minimization (intact states may however be eliminated, if they become unreachable, as we will describe below). For large automata (dictionaries) M, these are the great majority of states (the number of intact states ranges between $|Q| - |w| - 1$ and $|Q|$); therefore, it will be convenient in practice to consider M' as a modified version of M and will be treated as such in the algorithms found in this paper.

- States of the form (q, x) with $q \in Q - \{\perp\}$ and $x \in \mathrm{Pr}(w)$, such that $\delta^*(q_0, x) = q$; they will be called **cloned** states, inspired by the terminology in (Daciuk et al., 2000); the remaining states in $(Q - \{\perp\}) \times \mathrm{Pr}(w)$ —the great majority of states in $Q \times Q_w$— may safely be discarded because they are unreachable from the new start state $q_0' = (q_0, \epsilon)$, which itself is a cloned state. Cloned states are modified versions of the original states $q \in Q - \{\perp\}$: all of their outgoing transitions point to the corresponding intact states in Q', that is, $(\delta(q, a), \perp_w)$, except for the transition with symbol $a : xa \in \mathrm{Pr}(w)$, which now points to the corresponding cloned state $(\delta(q, a), xa)$, that is,

$$\delta'((q, x), a) = \begin{cases} (\delta(q, a), xa) & \text{if } xa \in \mathrm{Pr}(w) \\ (\delta(q, a), \perp_w) & \text{otherwise} \end{cases} \quad (5)$$

Cloned states are in F' if the original states are in F; in addition, if there is a cloned state of the form (q, w), then it is in F'. There are at most $|w| + 1$ cloned states.

- States of the form $(\perp, x)$, with $x \in \mathrm{Pr}(w)$. These states will be called **queue** states; states of this form appear when the string w is not in $\mathcal{L}(M)$ (the pertinent case, because we are adding it) and only if in the original automaton $\delta^*(q_0, x) = \perp$ for some $x \in \mathrm{Pr}(w)$. Only the final queue state $(\perp, w)$ —if it exists— is in F'. There are at most $|w|$ queue states.

- The new absorption state $\perp' = (\perp, \perp_w) \notin F$.

This automaton has to be minimized; however, due to the nature of the construction algorithm, minimization may be accomplished in a small number of operations. It is not difficult to show that minimization may be performed by initializing a list R called the **register** (Daciuk et al., 2000) with all of the intact states and then testing, one by one, queue and cloned states (starting with the last queue state $(\perp, w)$ or, if it does not exist, the last clone state (q, w), and descending in $\mathrm{Pr}(w)$) against states in the register and adding them to the register if they are not found to be equivalent to a state in R (performing this check backwards avoids having to test the equivalence of states by visiting their descendants recursively: see the end of section 4.1). Minimization (including the elimination of unreachable states in M') appears in section 4 as part of the string addition and removal algorithms.

3.2 Removing a string

Again, given a (possibly cyclic) minimal complete finite-state automaton M, it is easy to build a new complete automaton M' accepting $\mathcal{L}(M') = \mathcal{L}(M) - \{w\} = \mathcal{L}(M) \cap (\Sigma^* - \{w\})$ applying the intersection construct defined above to M and M_{-w}. The resulting automaton has the same sets of reachable states in Q' as in the case of adding string w, and, therefore, the same close-to-minimality properties; however, since w is supposed to be in $\mathcal{L}(M)$, no queue states will be formed (note that, if $w \notin \mathcal{L}(M)$, a nonaccepting queue with all states eventually equivalent to $\perp' = (\perp, \perp_w)$ may be formed). The accepting states in F' are: intact states $(q, \perp_w)$ and cloned states (q, x) with $q \in F$, except for state (q, w). Minimization may be performed analogously to the string addition case.

4 Algorithms

4.1 Adding a string

Figure 3 shows the algorithm that may be used to add a string to an existing automaton, which follows the construction in section 3.1. The resulting automaton is viewed as a modification of the original one: therefore, intact states are not created; instead, unreachable intact states are eliminated later. The register R of states not needing minimization is initialized with Q. The algorithm has three parts:

- First, the cloned and queue states are built and added to Q by using function $\mathrm{clone}()$ for all prefixes of w. The function returns a cloned state (with all transitions created) if the argument is a nonabsorption state in $Q - \{\perp\}$ or a queue state if it operates on the absorption state $\perp \in Q$.

- Second, those intact states that have become unreachable as a result of assigning the cloned state q_0' as the new start state are removed from Q and R and the start state is replaced by its clone. Unreachable states are simply those having no incoming transitions as constructed by the algorithm or as a consequence of the removal of other unreachable states; therefore, function $\mathrm{unreachable}()$ simply has to check for the absence of incoming transitions. Note that only intact states $(q, \perp_w)$ corresponding to q such that $\delta^*(q_0, x) = q$ for some $x \in \mathrm{Pr}(w)$ may become unreachable as a result of having been cloned.

- third, the queue and cloned states are checked (starting with the last state) against the register using function $\mathrm{replace\_or\_register}()$, which is essentially the same as the non-recursive version in Algorithm 2 in (Daciuk et al., 2000), and is shown in figure 1. If the argument state q is found to be equivalent to a state p in the register R, function $\mathrm{merge}(p, q)$ is called to redirect into p those transitions coming into q; if not, it is simply added to the register. Equivalence is checked by function $\mathrm{equiv}()$, shown in fig. 2), which checks for the equivalence of states by comparing (i) whether both states are accepting or not, and (ii) whether the corresponding outgoing transitions lead to the same state in R. Note that outgoing transitions cannot lead to equivalent states as there are no pairs of different equivalent states in the register ($\forall p, q \in R, \mathrm{equiv}(p, q) \Rightarrow p = q$) and backwards minimization guarantees that the state has no transitions to non-register states.

Finally, the new (minimal) automaton is returned. In real implementations, absorption states are not explicitly stored; this results in small differences in the implementations of the functions $\mathrm{clone}()$ and $\mathrm{equiv}()$.

4.2 Removing a string

The algorithm for removing a string from the language accepted by an automaton M' only differs from the previous algorithm in that the line

$$F \leftarrow F - \{q_{\mathrm{last}}\}$$

has to be added after the first end_for. Since the string removal algorithm will usually be asked to remove a string which was in $\mathcal{L}(M)$, function $\mathrm{clone}()$ will usually generate only cloned states and no queue states (see section 3.2 for the special case $w \notin L(M)$).

5 Examples

5.1 Adding a string

Assume we want to add the string bra to the automaton in fig. 4, which accepts the set of strings $(\mathtt{ba})^{+} \cup \{\mathtt{bar}\}$ (in all automata, the absorption state and all transitions leading to it will not be drawn for clarity). The single-string automaton for string bra is shown in fig 5. Application of the first stages of the string addition algorithm leads to the (unminimized) automaton in fig. 6. The automaton has, in addition to the set of intact states $\{(0, \bot_w), \ldots, (5, \bot_w)\}$, two cloned states ($(0, \epsilon)$ and $(1, \mathtt{b})$) and two queue states ($(\bot, \mathtt{br})$ and $(\bot, \mathtt{bra})$). As a consequence of the designation of $(0, \epsilon)$ as the new start state, shadowed states $(0, \bot_w)$ and $(1, \bot_w)$ become unreachable (have no incoming transitions) and are eliminated precisely in that order in the second stage of the algorithm. The final stage of the algorithm puts intact states in the register and tests queue and cloned states for equivalence with states in the register. The first state tested is $(\bot, \mathtt{bra})$, which is found to be equivalent to $(3, \bot_w)$: therefore, transitions coming into $(\bot, \mathtt{bra})$ are made to point to $(3, \bot_w)$. Then, states $(\bot, \mathtt{br})$, $(1, \mathtt{b})$ and $(0, \epsilon)$ are tested in order, found to have no equivalent in the register, and added to it. The resulting minimal automaton, after a convenient renumbering of states, is shown in fig. 7.

5.2 Removing a string

Now, let us consider we want to remove string baba from the language accepted by the automaton in fig. 7 (the single-string automaton for baba is shown in fig. 8). The automaton resulting from the application of the initial (construction) stages of the automaton is shown in fig. 9. Note that state $(6, \mathtt{baba})$ is marked as nonaccepting because we are removing a string. Again, as a consequence of the designation of $(0, \epsilon)$ as the new start state, shadowed states $(0, \bot_w)$, $(1, \bot_w)$, and $(2, \bot_w)$ become unreachable (have no incoming transitions) and are eliminated precisely in that order in the second stage of the algorithm. The last stage of the algorithm puts all intact states in the register and checks cloned states $(6, \texttt{baba})$, $(4, \texttt{bab})$, $(2, \texttt{ba})$, $(1, \texttt{b})$ and $(0, \epsilon)$ (no queue states since baba is accepted by the automaton in fig. 7), and finds none of them to be equivalent to those in the register, to which they are added. The resulting minimal automaton is shown in figure 10.

6 Concluding remarks

We have derived, from basic results of language and automata theory, a simple method to modify a minimal (possibly cyclic) finite-state automaton so that it recognizes one more string or one string less, while keeping the finite-state automaton minimal. These two operations may be applied to dictionary construction and maintenance and generalize the result in Daciuk et al.'s (2000) second algorithm (incremental construction of acyclic finite-state automata from unsorted strings) in two respects, with interesting practical implications:

- The method described here allows for the addition of strings to the languages of cyclic automata (in practice, it may be convenient to have cycles in dictionaries if we want them to accept, for example, all possible integer numbers or internet addresses). In this respect, the algorithm presented also generalizes the string removal method sketched by Revuz (2000) for acyclic automata.
- Removal of strings is as easy as addition. This means that, for example, the detection of an erroneous entry in the dictionary does not imply having to rebuild it completely.

The asymptotic time complexity of the algorithms is in the same class ($O(|Q||w|)$) as that in (Daciuk et al., 2000). This is because the slowest part of the algorithm (the last one) checks all queue and cloned states ($O(|w|)$) against all states the register ($O(|Q|)$). As suggested by one of the referees, an improvement in efficiency may be obtained by realizing that, in many cases, cloned states corresponding to the shortest prefixes of string w are not affected by minimization because their intact equivalents have become unreachable and therefore have been removed from the register; the solution lies in identifying these states and not cloning them (for example, Daciuk et al's (2000) and Revuz's (2000) algorithms do not clone them).

As for the future, we are working on an adaptation of this algorithm to the maintenance of morphological analysers and generators using finite-state nondeterministic letter transducers (Roche and Schabes, 1997; Garrido et al., 1999).

Acknowledgments

This work has been funded by the Spanish Comisión Interministerial de Ciencia y Tecnología through grant TIC2000-1599. We thank the two referees for their suggestions and Prof. Colin de la Higuera (Univ. Jean Monnet, St. Etienne, France) for his comments on the manuscript.

References

Daciuk, Jan, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. 2000. Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics*, 26(1):3–16.

Garrido, Alicia, Amaia Iturraspe, Sandra Montserrat, Hermínia Pastor, and Mikel L. Forcada. 1999. A compiler for morphological analysers and generators based on finite-state transducers. *Procesamiento del Lenguaje Natural*, (25):93–98.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to automata theory, languages, and computation*. Addison–Wesley, Reading, MA.

Karakostas, George, Anastasios Viglas, and Richard J. Lipton. 2000. On the complexity of intersecting finite state automata. In *Proceedings of the 15th Annual IEEE Conference on Computational Complexity (CoCo'00)*.

Revuz, Dominique. 2000. Dynamic acyclic minimal automaton. In *Preproceedings of CIAA 2000, Fifth International Conference on Implementation and Application of Automata*, pages 226–232, London, Canada, July 24–25.

Roche, Emmanuel and Yves Schabes. 1997. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, Mass., pages 1–65.

```
function replace_or_register(q)
    if ∃p ∈ R : equiv(p, q) then
        merge(p, q)
    else
        R ← R ∪ {q}
    end_if
end_function
```

Figure 1
The function replace_or_register()

```
function equiv(p, q)
    if (p ∈ F ∧ q ∉ F) ∨ (p ∉ F ∧ q ∈ F) return false
    for all symbols a ∈ Σ
        if δ(p, a) ≠ δ(q, a) return false
    end_for
    return true
end_function
```

Figure 2
The function $\mathrm{equiv}(p, q)$

algorithm **addstring**

Input: $M = (Q, \Sigma, \delta, q_0, F)$ (minimal, complete), $w \in \Sigma^*$

Output: $M' = (Q', \Sigma, \delta', q_0', F')$ minimal, complete, and such that $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$

$R \leftarrow Q$ [initialize register]

$q_0' \leftarrow \mathrm{clone}(q_0)$ [clone start state]

$q_{\mathrm{last}} \leftarrow q_0'$

for $i = 1$ to $|w|$

$q \leftarrow \mathrm{clone}(\delta^*(q_0, w_1 \cdots w_i))$ [create cloned and queue states;
add clones of accepting states to F]

$\delta(q_{\mathrm{last}}, w_i) \leftarrow q$

$q_{\mathrm{last}} \leftarrow q$

end_for

$i \leftarrow 1$

$q_{\mathrm{current}} \leftarrow q_0$

while($i \leq |w|$ and unreachable(q_{current}))

$q_{\mathrm{next}} \leftarrow \delta(q_{\mathrm{current}}, w_i)$

$Q \leftarrow Q - \{q_{\mathrm{current}}\}$ [remove unreachable state from Q
and update transitions in δ]

$R \leftarrow R - \{q_{\mathrm{current}}\}$ [remove also from register]

$q_{\mathrm{current}} \leftarrow q_{\mathrm{next}}$

$i \leftarrow i + 1$

end_while

if unreachable(q_{current})

$Q \leftarrow Q - \{q_{\mathrm{current}}\}$

$R \leftarrow R - \{q_{\mathrm{current}}\}$

end_if

$q_0 \leftarrow q_0'$ [replace start state]

for $i = |w|$ downto 1

replace_or_register($\delta^*(q_0, w_1 \cdots w_i)$) [check queue and cloned states one by one]

end_for

return $M = (Q, \Sigma, \delta, q_0, F)$

end_algorithm

Figure 3
Algorithm to add a string w to the language accepted by a finite-state automaton while keeping it minimal.

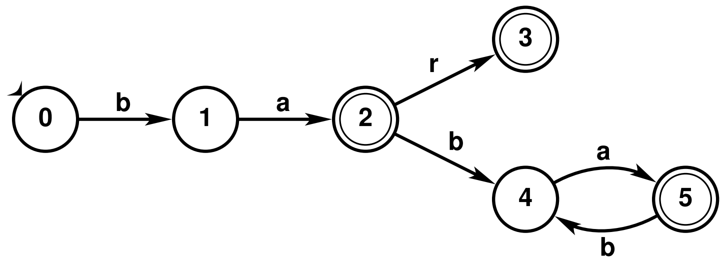

Figure 4
Minimal automaton accepting the set of strings $(\texttt{ba})^+ \cup \{\texttt{bar}\}$.

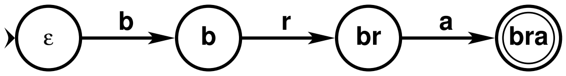

Figure 5
Single-string automaton accepting string `bra`.

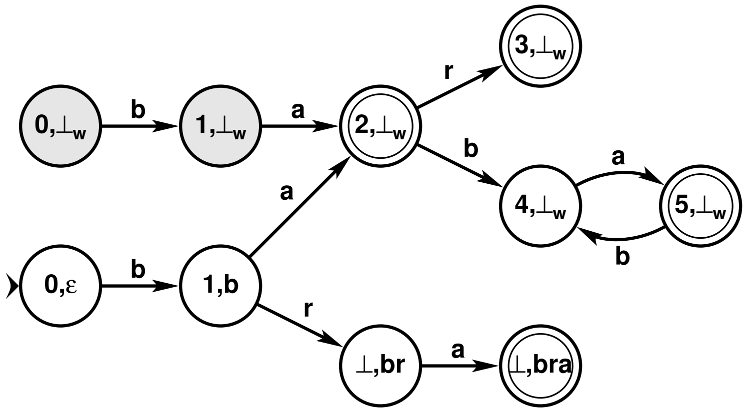

Figure 6
Unminimized automaton accepting the set $(\texttt{ba})^+ \cup \{\texttt{bar}\} \cup \{\texttt{bra}\}$. Shadowed states $(0, \perp_w)$ and $(1, \perp_w)$ have become unreachable (have no incoming transitions) and are eliminated precisely in that order.

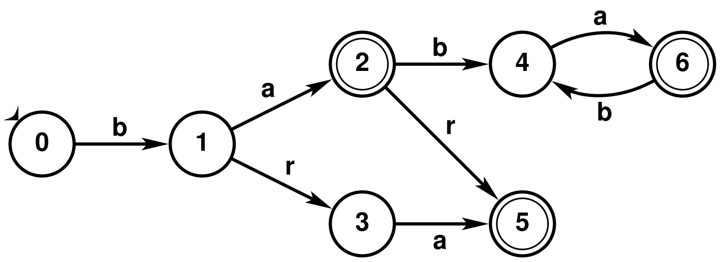

Figure 7

Minimal automaton accepting the set $(\texttt{ba})^+ \cup \{\texttt{bar}\} \cup \{\texttt{bra}\}$.

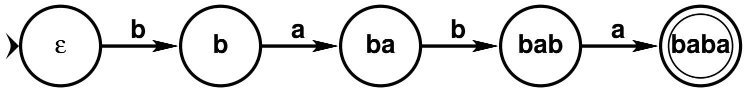

Figure 8
Single-string automaton accepting the string `baba`.

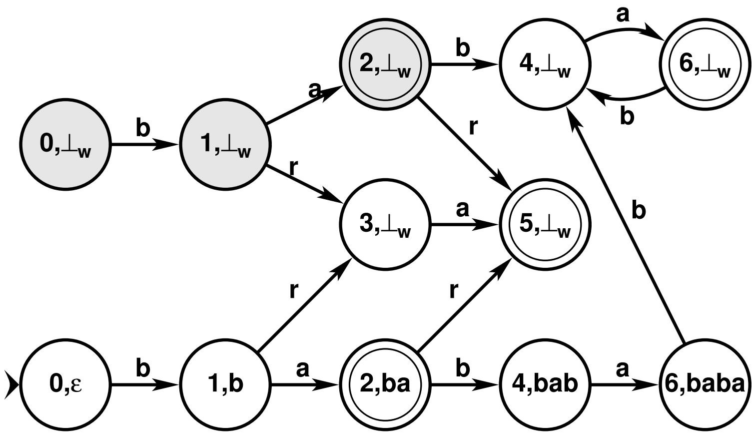

Figure 9
Unminimized automaton accepting the set $(\mathtt{ba})^+ \cup \{\mathtt{bar}\} \cup \{\mathtt{bra}\} - \{\mathtt{baba}\}$. Shadowed states $(0, \perp_w)$, $(1, \perp_w)$ and $(2, \perp_w)$ have become unreachable (have no incoming transitions) and are eliminated precisely in that order.

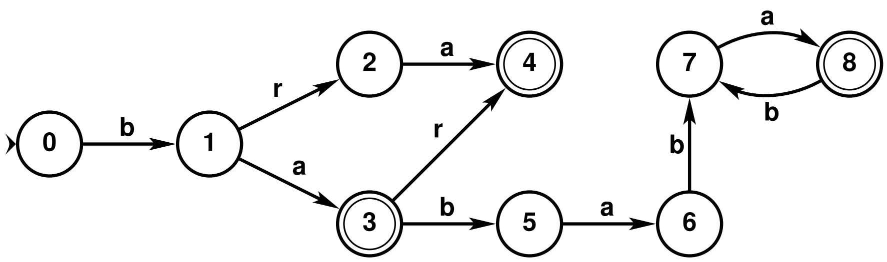

Figure 10
Minimal automaton accepting the set $(\texttt{ba})^+ \cup \{\texttt{bar}\} \cup \{\texttt{bra}\} - \{\texttt{baba}\}$.