Discrete Applied Mathematics 163 (2014) 238–246

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Cycle-aware minimization of acyclic deterministic finite-state automata

Johannes Bubenzer*

University of Potsdam, Department of Linguistics, Karl-Liebknecht-Strasse 24-25, 14476 Potsdam, Germany

ARTICLE INFO

Article history:
Received 20 January 2012
Received in revised form 1 July 2013
Accepted 11 August 2013
Available online 18 September 2013

Keywords:
Minimization
Deterministic finite state automata
Algorithmic

ABSTRACT

In this paper a linear-time algorithm for the minimization of acyclic deterministic finite-state automata is presented. The algorithm runs significantly faster than previous algorithms for the same task. This is shown by a comparison of the running times of both algorithms. Additionally, a variation of the new algorithm is presented which handles cyclic automata as input. The new cycle-aware algorithm minimizes acyclic automata in the desired way. In case of cyclic input, the algorithm minimizes all acyclic suffixes of the input automaton.

1. Introduction

Minimization of deterministic finite state automata (DFA) dates back to the 1950s [8,10]. The asymptotically fastest algorithm for the general case of DFA is due to Hopcroft [7] and runs in $O(n \log n)$. There exists a linear time algorithm for the minimization of acyclic deterministic finite-state automata (ADFA) by Revuz [14]. In this paper we present a new algorithm for the minimization of ADFA [2] and compare it to Revuz's well-established algorithm.

The new minimization algorithm cannot properly handle cyclic DFA as input. We therefore present an adapted algorithm which does not hang upon cyclic input. The adapted algorithm leaves all states in or before cycles unchanged, but minimizes all states in acyclic suffix paths.

2. Related work

Moore's famous minimization algorithm [10] determines non-equivalence for pairs of states of an input DFA. It thus runs with quadratic time and memory complexity. Starting with final and non-final states marked as non-equivalent, the algorithm proceeds by marking states reaching non-equivalent states (with the same symbol) as non-equivalent. In the end, all states that cannot be marked as non-equivalent are joined into one.

Hopcroft's algorithm [7] is the asymptotically fastest minimization algorithm in the general case. It runs in $O(n \log n)$. Due to the complexity of the algorithm and its proof, a few papers were published that explain and re-explain the Hopcroft algorithm [5,9].

The Brzozowski-algorithm [1] is quite special in that it computes the minimal automaton on the basis of the determinization (det) and the inversion (inv) operations. The minimal automaton $\mathcal{A}_m$ is computed from the source automaton $\mathcal{A}$ by:

$$\mathcal{A}_m \longleftarrow \det(\mathrm{inv}(\det(\mathrm{inv}(\mathcal{A})))). \tag{1}$$

* Fax: +49 3055876526.
E-mail addresses: johannes@bubenzer.com, bubenzer@uni-potsdam.de.

0166-218X/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.dam.2013.08.003

The algorithm first joins all equivalent suffix-states together using the determinization in between the double-inversion. The final determinization then joins the equivalent prefix states. The algorithm shows exponential worst case complexity, since the complexity of determinization is exponential but it performs exceptionally well in practice [13].

For certain subsets of the deterministic finite-state automata, one can achieve minimization in linear time. The best known algorithm is due to Revuz [14]—it minimizes acyclic DFAs in linear time. This approach is discussed in more detail in Section 4.

A taxonomy of the most important finite state minimization algorithms can be found in [16,18] and in [17].

3. Preliminaries

In this section we introduce the basic definitions and theorems regarding languages and automata needed in this paper.

3.1. Alphabets, words and languages

An *alphabet* Σ is a finite set of symbols. A word w over Σ is a finite concatenation of symbols from Σ, its length $|w|$ is the number of concatenated symbols. The symbol $w[i] = a \in \Sigma$ of a word w is the symbol at the i'th position. By ε we denote the empty word ($|\varepsilon| = 0$). The set Σ^* is the set of all words over Σ including the empty word, whereas Σ^+ is the set of all non-empty words. Any subset $\mathcal{L} \subseteq \Sigma^*$ of Σ^* is called a *language*. For convenience we assume each alphabet to be a total order together with some operation $<$.

3.2. Automata and languages

A *deterministic finite-state automaton* (DFA) $\mathcal{A} = \langle Q, q_0, \Sigma, \delta, F\rangle$ consists of a finite set of states Q, a designated start-state $q_0 \in Q$, an alphabet Σ, a set of final states $F \subseteq Q$ and a transition function $\delta : Q \times \Sigma \mapsto Q$. The transition function is extended to the acceptance of words in the usual way:

$$\delta^*(q, \varepsilon) = q$$
$$\delta^*(q, aw) = \delta^*(\delta(q, a), w)$$

for all states $q \in Q$ and $a \in \Sigma$ is a symbol, $w \in \Sigma^*$ is a word. A word w is said to be *accepted* by the automaton, if $\delta^*(q_0, w) \in F$. The language $\mathcal{L}(\mathcal{A})$ accepted by a DFA $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$:

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \in F\}.$$

Each language accepted by some DFAs is a called a *regular language*. The *right-language* $\overrightarrow{\mathcal{L}}(q)$ of a state $q \in Q$ is defined as the set of words accepted by an automaton started in q:

$$\overrightarrow{\mathcal{L}}(q) = \{w \in \Sigma^* \mid \delta^*(q, w) \in F\}.$$

We denote by $\delta'(q)$ the set of outgoing transitions from state q:

$$\delta'(q) = \{\langle a, p\rangle \mid a \in \Sigma,\ p \in Q,\ \delta(q, a) = p\}.$$

The destination state of a transition $t \in \delta'(q)$ is denoted by t_{next} and the transition symbol by t_{sym}. A typical way to implement the set of outgoing transitions is as an array of transitions. We use the term *transition array* in the following to refer to the actual implementation of the set of outgoing transitions. We will assume that transition arrays are sorted by their transition symbols. We define the *signature* $\overrightarrow{q}$ of a state q, to be q's set of outgoing transitions unified with ε iff the state is final:

$$\overrightarrow{q} = \begin{cases} \delta'(q) \cup \{\varepsilon\} & \texttt{if } q \in F \\ \delta'(q) & \texttt{else}. \end{cases}$$

The signature of final states gets an additional $\{\varepsilon\}$ element in the formula above. This element will be called the finality attribute in the following text. In the latter we will often refer to the signature as being a sequence of finality attribute and states rather than a set. A sequential signature holds the transitions ordered by the transition symbol, the finality attribute being the first element if present.

A state q is called *accessible* if it is reachable from the start-state:

$$\exists w \in \Sigma^* \mid \delta^*(q_0, w) = q$$

and the state is said to be *co-accessible* if there is a path from q to a final state.

$$\exists w \in \Sigma^* \mid \delta^*(q, w) \in F.$$

A DFA is *connected* iff each of its states is both accessible and co-accessible. We assume all automata to be connected in the following text. A DFA is *acyclic* if there is no state which can reach itself over a non-empty chain of transitions:

$$\forall q \in Q, \quad \forall w \in \Sigma^+ : \delta^*(q, w) = p \rightarrow q \neq p.$$

Otherwise the DFA is *cyclic*.