Implementation and Application of Automata

14th International Conference, CIAA 2009
Sydney, Australia, July 14-17, 2009
Proceedings
Volume Editor

Sebastian Maneth
NICTA and University of New South Wales
Sydney, Australia
E-mail: sebastian.maneth@nicta.com.au

Library of Congress Control Number: Applied for

CR Subject Classification (1998): F.1, F.4, F.2, D.2

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-02978-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-02978-3 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
springer.com
© Springer-Verlag Berlin Heidelberg 2009
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper   SPIN: 12715176   06/3180   5 4 3 2 1 0
Preface

The 14th International Conference on Implementation and Application of Automata (CIAA 2009) was held in NICTA’s Neville Roach Laboratory at the University of New South Wales, Sydney, Australia during July 14–17, 2009.

This volume of Lecture Notes in Computer Science contains the papers that were presented at CIAA 2009, as well as abstracts of the posters and short papers that were presented at the conference. The volume also includes papers or extended abstracts of the three invited talks presented by Gonzalo Navarro on Implementation and Application of Automata in String Processing, by Christoph Koch on Applications of Automata in XML Processing, and by Helmut Seidl on Program Analysis Through Finite Tree Automata.

The 23 regular papers were selected from 42 submissions covering various fields in the application, implementation, and theory of automata and related structures. This year, six additional papers were selected as “short papers”; at the conference these were allocated the same presentation length as regular papers. Each paper was reviewed by at least three Program Committee members, with the assistance of external referees. Papers were submitted by authors from the following countries: Australia, Austria, Belgium, Brazil, Canada, China, Czech Republic, Finland, France, Germany, India, Italy, Republic of Korea, Japan, Latvia, The Netherlands, Portugal, Russian Federation, Spain, South Africa, Turkey, United Arab Emirates, and the USA.

I wish to thank all who made this conference possible: the authors for submitting their papers, the Program Committee members and external referees (listed on the next pages) for giving their valuable opinions and writing reports about the submitted papers, and the three invited speakers for giving presentations related to the implementation and application of automata. Finally, I would like to express my gratitude to the sponsors (listed on the next pages), local organizers, and to the editors of Lecture Notes in Computer Science, in particular to Alfred Hofmann, for their help in publishing this volume in a timely manner.

July 2009

Sebastian Maneth
CIAA 2009 was organized in cooperation with NICTA, Australia’s ICT Research Center of Excellence. It was held in NICTA’s Neville Roach Laboratory at the University of New South Wales, Sydney, Australia.

### Program Committee

<table>
<thead>
<tr>
<th>Name</th>
<th>Institution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mikolaj Bojańczyk</td>
<td>Warsaw University, Poland</td>
</tr>
<tr>
<td>Ahmed Bouajjani</td>
<td>Université Paris Diderot, France</td>
</tr>
<tr>
<td>Cristian S. Calude</td>
<td>University of Auckland, New Zealand</td>
</tr>
<tr>
<td>Jean-Marc Champarnaud</td>
<td>Université de Rouen, France</td>
</tr>
<tr>
<td>Hubert Comon-Lundh</td>
<td>École Normale Supérieure de Cachan, France</td>
</tr>
<tr>
<td>Maxime Crochemore</td>
<td>Université Paris-Est Marne-la-Vallée, France</td>
</tr>
<tr>
<td>Michael Domaratzki</td>
<td>University of Manitoba, Canada</td>
</tr>
<tr>
<td>Frank Drewes</td>
<td>Umeå University, Sweden</td>
</tr>
<tr>
<td>Jan Holub</td>
<td>Czech Technical University in Prague, Czech Republic</td>
</tr>
<tr>
<td>Hendrik Jan Hoogeboom</td>
<td>Universiteit Leiden, The Netherlands</td>
</tr>
<tr>
<td>Juraj Hromkovic</td>
<td>ETH Zürich, Switzerland</td>
</tr>
<tr>
<td>Oscar H. Ibarra</td>
<td>University of California, Santa Barbara, USA</td>
</tr>
<tr>
<td>Lucian Ilie</td>
<td>University of Western Ontario, Canada</td>
</tr>
<tr>
<td>Masami Ito</td>
<td>Kyoto Sangyo University, Japan</td>
</tr>
<tr>
<td>Juhani Karhumäki</td>
<td>University of Turku, Finland</td>
</tr>
<tr>
<td>Markus Lohrey</td>
<td>Universität Leipzig, Germany</td>
</tr>
<tr>
<td>Sebastian Maneth (Chair)</td>
<td>NICTA and University of New South Wales, Australia</td>
</tr>
<tr>
<td>Denis Maurel</td>
<td>Université François Rabelais de Tours, France</td>
</tr>
<tr>
<td>Filippo Mignosi</td>
<td>Università Degli Studi - L’Aquila, Italy</td>
</tr>
<tr>
<td>Mehryar Mohri</td>
<td>Courant Institute of Mathematical Sciences, USA</td>
</tr>
<tr>
<td>Anca Muscholl</td>
<td>Université Bordeaux 1, France</td>
</tr>
<tr>
<td>Joachim Niehren</td>
<td>INRIA Lille - Nord Europe, France</td>
</tr>
<tr>
<td>Dirk Nowotka</td>
<td>Universität Stuttgart, Germany</td>
</tr>
<tr>
<td>Bala Ravikumar</td>
<td>Sonoma State University, USA</td>
</tr>
<tr>
<td>Wojciech Rytter</td>
<td>Warsaw University, Poland</td>
</tr>
<tr>
<td>Kai Salomaa</td>
<td>Queen’s University, Canada</td>
</tr>
<tr>
<td>Thomas Schwentick</td>
<td>Technische Universität Dortmund, Germany</td>
</tr>
<tr>
<td>Stefan Schwoon</td>
<td>Technische Universität München, Germany</td>
</tr>
<tr>
<td>Colin Stirling</td>
<td>University of Edinburgh, UK</td>
</tr>
<tr>
<td>Hsu-Chun Yen</td>
<td>National Taiwan University, Taiwan</td>
</tr>
<tr>
<td>Sheng Yu</td>
<td>University of Western Ontario, Canada</td>
</tr>
</tbody>
</table>
Steering Committee

Jean-Marc Champarnaud, Université de Rouen, France
Oscar H. Ibarra, University of California, Santa Barbara, USA
Denis Maurel, Université François Rabelais de Tours, France
Kai Salomaa, Queen’s University, Canada
Sheng Yu (Chair), University of Western Ontario, Canada

External Referees

Cyril Allauzen, Alessio Langiu
Pavlos Antoniou, Jürgen Laun
Miroslav Balík, Aurélien Lemay
Hans-Joachim Böckenhauer, Peter Leupold
Iovka Boneva, Christof Löding
Béatrice Bouchou, Alexander Meduna
Arnaud Carayol, Tobias Mömke
Giusi Castiglione, Marius Nagy
Loek Cleophas, Cyril Nicaud
Maxime Crochemore, Alexander Okhotin
Ömer Egecioglu, Pawel Parys
Yuan Gao, Andrei Păun
Stefan Göller, Narad Rampersad
Giovanna Guaiana, Antonia Restivo
Peter Habermehl, Giuseppina Rindone
Mika Hirvensalo, Adam Rogalewicz
Benjamin Hoffmann, Jacques Sakarovitch
Johanna Högbberg, Sven Schewe
Jan Janoušek, Olivier Serre
Jarkko Kari, Fabien Torre
Dennis Komm, Szymon Toruńczyk
Jörg Kreiker, Nicholas Tran
Manfred Kufleitner, Stavros Tripakis
Yoshiyuki Kunimochi, Tomás Vojnar
Dietrich Kuske, Bruce Watson

Local Organization

Jane Gillis, NICTA
Sebastian Maneth, NICTA and University of New South Wales

Sponsors

National ICT Australia Limited (NICTA)
The University of New South Wales, Sydney, Australia
Yahoo! Research Latin America, Santiago, Chile
## Table of Contents

### Invited Lectures

Implementation and Application of Automata in String Processing .......................... 1  

*Gonzalo Navarro*

Applications of Automata in XML Processing ......................................................... 2  

*Christoph Koch*

Program Analysis through Finite Tree Automata .................................................... 3  

*Helmut Seidl*

### Technical Contributions

An $n \log n$ Algorithm for Hyper-minimizing States in a (Minimized) Deterministic Automaton ................................................................. 4  

*Markus Holzer and Andreas Maletti*

On Extremal Cases of Hopcroft’s Algorithm ......................................................... 14  

*Giusi Castiglione, Antonio Restivo, and Marinella Sciortino*

Compact Normal Form for Regular Languages as Xor Automata ............................ 24  

*Jean Vuillemin and Nicolas Gama*

Cellular Automata with Sparse Communication .................................................. 34  

*Martin Kutrib and Andreas Malcher*

A Cellular Automaton Model for Car Traffic with a Slow-to-Stop Rule ..................... 44  

*Adam Clarridge and Kai Salomaa*

On Parallel Implementations of Deterministic Finite Automata ............................. 54  

*Jan Holub and Stanislav Štekr*

FAdo and GUltar: Tools for Automata Manipulation and Visualization ........................ 65  

*André Almeida, Marco Almeida, José Alves, Nelma Moreira, and Rogério Reis*

A Testing Framework for Finite-State Morphology .............................................. 75  

*François Barthélémy*

A Table Compression Method for Extended Aho-Corasick Automaton ........................ 84  

*Yanbing Liu, Yifu Yang, Ping Liu, and Jianlong Tan*
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compact Representation for Answer Sets of $n$-ary Regular Queries</td>
<td>94</td>
</tr>
<tr>
<td>Kazuhiro Inaba and Haruo Hosoya</td>
<td></td>
</tr>
<tr>
<td>Recognition of a Spanning Tree of Directed Acyclic Graphs by Tree</td>
<td>105</td>
</tr>
<tr>
<td>Automata</td>
<td></td>
</tr>
<tr>
<td>Akio Fujiyoshi</td>
<td></td>
</tr>
<tr>
<td>Random Generation of Deterministic Tree (Walking) Automata</td>
<td>115</td>
</tr>
<tr>
<td>Pierre-Cyrille Héam, Cyril Nicaud, and Sylvain Schmitz</td>
<td></td>
</tr>
<tr>
<td>Hedge Pattern Partial Derivative</td>
<td>125</td>
</tr>
<tr>
<td>Taro Suzuki and Satoshi Okui</td>
<td></td>
</tr>
<tr>
<td>TAGED Approximations for Temporal Properties Model-Checking</td>
<td>135</td>
</tr>
<tr>
<td>Roméo Courbis, Pierre-Cyrille Héam, and Olga Kouchnarenko</td>
<td></td>
</tr>
<tr>
<td>Verifying Parallel Programs with Dynamic Communication Structures</td>
<td>145</td>
</tr>
<tr>
<td>Mohamed Faouzi Atig and Tayssir Touili</td>
<td></td>
</tr>
<tr>
<td>Fixpoint Guided Abstraction Refinement for Alternating Automata</td>
<td>155</td>
</tr>
<tr>
<td>Pierre Ganty, Nicolas Maquet, and Jean-François Raskin</td>
<td></td>
</tr>
<tr>
<td>Automata-Based Termination Proofs</td>
<td>165</td>
</tr>
<tr>
<td>Radu Iosif and Adam Rogalewicz</td>
<td></td>
</tr>
<tr>
<td>Implementation of State Elimination Using Heuristics</td>
<td>178</td>
</tr>
<tr>
<td>Jae-Hee Ahn and Yo-Sub Han</td>
<td></td>
</tr>
<tr>
<td>Short Regular Expressions from Finite Automata: Empirical Results</td>
<td>188</td>
</tr>
<tr>
<td>Hermann Gruber, Markus Holzer, and Michael Tautschnig</td>
<td></td>
</tr>
<tr>
<td>Small Extended Expressions for Acyclic Automata</td>
<td>198</td>
</tr>
<tr>
<td>Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot</td>
<td></td>
</tr>
<tr>
<td>Quantum Queries on Permutations with a Promise</td>
<td>208</td>
</tr>
<tr>
<td>Rūsiņš Freivalds and Kazuo Iwama</td>
<td></td>
</tr>
<tr>
<td>Time-Optimal Winning Strategies for Poset Games</td>
<td>217</td>
</tr>
<tr>
<td>Martin Zimmermann</td>
<td></td>
</tr>
<tr>
<td>Amount of Nonconstructivity in Finite Automata</td>
<td>227</td>
</tr>
<tr>
<td>Rūsiņš Freivalds</td>
<td></td>
</tr>
</tbody>
</table>

**Short Papers and Poster Abstracts**

Multiflex: A Multilingual Finite-State Tool for Multi-Word Units     | 237  |
| Agata Savary                                                          |      |
# Table of Contents

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Efficient Parsing Using Filtered-Popping Recursive Transition Networks</td>
<td>241</td>
</tr>
<tr>
<td>Javier M. Sastre-Martínez</td>
<td></td>
</tr>
<tr>
<td>Forest FIRE: A Taxonomy-Based Toolkit of Tree Automata and Regular Tree Algorithms</td>
<td>245</td>
</tr>
<tr>
<td>Loek Cleophas and Kees Hemerik</td>
<td></td>
</tr>
<tr>
<td>Formally Synthesising a Protocol Converter: A Case Study</td>
<td>249</td>
</tr>
<tr>
<td>Jing Cao and Albert Nymeyer</td>
<td></td>
</tr>
<tr>
<td>Compiler Generator Based on Restarting Automata</td>
<td>253</td>
</tr>
<tr>
<td>Jan Procházka</td>
<td></td>
</tr>
<tr>
<td>Are Statecharts Finite Automata?</td>
<td>258</td>
</tr>
<tr>
<td>Hanlin Lu and Sheng Yu</td>
<td></td>
</tr>
<tr>
<td><strong>Author Index</strong></td>
<td>263</td>
</tr>
</tbody>
</table>
Implementation and Application of Automata in String Processing*

Gonzalo Navarro

Department of Computer Science
University of Chile
gnavarro@dcc.uchile.cl

Automata have been enormously successful in matching different types of complex patterns on sequences, with applications in many areas, from text retrieval to bioinformatics, from multimedia databases to signal processing. In general terms, the process to match a complex pattern is (1) design a NFA that recognizes the pattern; (2) slightly modify it to recognize any string ending with the pattern; (3) convert it into a DFA; (4) feed it with the sequence, signaling the endpoints of a pattern occurrence each time the DFA reaches a final state. Alternatively one can omit step (2) and backtrack with the DFA on the suffix tree of the sequence, which leads to sublinear-time complex pattern matching in many relevant cases. This process, as it is well-known, has a potential problem in stage (3), because the DFA can be of exponential size. Rather than being a theoretical reservation, the problem does arise in a number of real-life situations.

*Bit-parallelism* is a technique that helps circumvent this problem in many practical cases. It allows carrying out several operations in parallel on the bits of a computer word. By mapping NFA states to bits, bit-parallelism allows one to simulate the NFA behavior efficiently without converting it to deterministic. We show how bit-parallelism can be applied in many problems where the NFA has a regular structure, which allows us simulating it using typical processor instructions on machine words. Moreover, we show that even on general regular expressions, without any particular structure, bit-parallelism allows one to reduce the space requirement of the DFA. In general, the bit-parallel algorithm on the NFA is simpler to program and more space and time efficient than the one based on the DFA.

We show the use of bit-parallelism for exact pattern matching, for allowing optional and repeatable characters, classes of characters and bounded-length gaps, and for general regular expressions. The paradigm is flexible enough to permit combining any of those searches with approximate matching, where the occurrence can be at a limited edit distance to a string of the language denoted by the automaton. We then show applications of these ideas to natural language processing, where the text is seen as a sequence of words, and bit-parallelism allows flexibility in the matching at the word level, for example allowing missing or spurious words.

* Partially funded by the Millennium Institute for Cell Dynamics and Biotechnology (ICDB), Grant ICM P05-001-F, Mideplan, Chile.
Applications of Automata in XML Processing

Christoph Koch
Cornell University
Ithaca, NY, USA
koch@cs.cornell.edu

XML is at once a document format and a semistructured data model, and has become a de-facto standard for exchanging data on the Internet. XML documents can alternatively be viewed as labeled trees, and tree automata are natural mechanisms for a wide range of processing tasks on XML documents. In this talk, I survey applications of automata in XML processing with an emphasis on those directions of work that so far have had the greatest practical impact. The talk will consist of three parts. In the first, I will discuss XML validation. The standard schema formalisms for XML, Document Type Definitions and XML Schema, are regular tree grammars at their core. These official standards of the World Wide Web Consortium are well-founded in automata theory and formal language theory, and are designed to incorporate special restrictions to facilitate the creation of automata for document validation. The second part will cover XML stream processing techniques and XML publish-subscribe systems, an area in which a number of exciting automata-based systems have been built. The third and final part covers XML query processing using automata, and applications in Web information extraction.
Program Analysis through Finite Tree Automata

Helmut Seidl

Lehrstuhl für Informatik II, Technische Universität München
Boltzmannstraße 3, D-85748 Garching b. München, Germany
seidl@in.tum.de

Dynamic Pushdown Networks (dpn’s) have recently been introduced as a convenient abstraction of systems which provide recursive procedure calls and spawning of concurrent tasks such as Java programs [1, 4, 6]. We show how the executions of dpn’s can naturally be represented through ranked trees. The configuration reached by a program execution then can be read off from the sequence of leaves of this execution tree. This observation allows us to reduce decision problems such as reachability of configurations within a regular set for dpn’s to standard decision problems for finite tree automata.

Our reduction does not only shed fresh light onto dpn’s but also provides us with new efficient algorithms which can be implemented through standard libraries for finite tree automata. Finite tree automata on the other hand, can be nicely represented by specific Horn clauses. In the presentation, we therefore indicate how these algorithms can be realized by means of generic solvers for a particular decidable class of Horn clauses [2, 3].

Bibliography

An $n \log n$ Algorithm for Hyper-minimizing States in a (Minimized) Deterministic Automaton

Markus Holzer$^1,\star$ and Andreas Maletti$^2,\star\star$

$^1$ Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
holzer@informatik.uni-giessen.de

$^2$ Departament de Filologies Romàniques, Universitat Rovira i Virgili
Av. Catalunya 35, 43002 Tarragona, Spain
andreas.maletti@urv.cat

Abstract. We improve a recent result [A. Badr: Hyper-Minimization in $O(n^2)$. In Proc. CIAA, LNCS 5148, 2008] for hyper-minimized finite automata. Namely, we present an $O(n \log n)$ algorithm that computes for a given finite deterministic automaton (dfa) an almost equivalent dfa that is as small as possible—such an automaton is called hyper-minimal. Here two finite automata are almost equivalent if and only if the symmetric difference of their languages is finite. In other words, two almost-equivalent automata disagree on acceptance on finitely many inputs. In this way, we solve an open problem stated in [A. Badr, V. Geffert, I. Shipman: Hyper-minimizing minimized deterministic finite state automata. RAIRO Theor. Inf. Appl. 43(1), 2009] and by Badr. Moreover, we show that minimization linearly reduces to hyper-minimization, which shows that the time-bound $O(n \log n)$ is optimal for hyper-minimization.

1 Introduction

Early studies in automata theory revealed that nondeterministic and deterministic finite automata are equivalent [1]. However, nondeterministic automata can be exponentially more succinct w.r.t. the number of states [23]. In fact, finite automata are probably best known for being equivalent to right-linear context-free grammars and, thus, for capturing the lowest level of the Chomsky-hierarchy, which is the family of regular languages. Over the last 50 years, a vast literature documenting the importance of finite automata as an enormously valuable concept has been developed. Although, there are a lot of similarities between nondeterministic and deterministic finite automata, one important difference is that of the minimization problem. The study of this problem also dates back to the early beginnings of automata theory. It is of practical relevance because regular languages are used in many applications, and one may like to represent the languages succinctly. While for nondeterministic automata the computation of an equivalent minimal automaton is PSPACE-complete [4] and thus highly

* Part of the work was done while the author was at Institut für Informatik, Technische Universität München, Boltzmannstraße 3, D-85748 Garching bei München, Germany.

** Supported by the Ministerio de Educación y Ciencia (MEC) grant JDCI-2007-760.
Intractable, the corresponding problem for deterministic automata is known to be effectively solvable in polynomial time [5]. An automaton is minimal if every other automaton with fewer states disagrees on acceptance for at least one input.

Minimizing deterministic finite automata (DFA) is based on computing an equivalence relation on the states of the machine and collapsing states that are equivalent. Here two states \( p, q \in Q \), where \( Q \) is the set of states of the automaton under consideration, are equivalent, if the automaton starting its computation in state \( p \) accepts the same language as the automaton if \( q \) is taken as a start state. Minimization of two equivalent DFA leads to minimal DFA that are isomorphic up to the renaming of states. Hence, minimal DFA are unique. This allows one to give a nice characterization: A DFA \( M \) is minimal if and only if in \( M \): (i) there are no unreachable states and (ii) there is no pair of different but equivalent states.

The computation of this equivalence can be implemented in a straightforward fashion by repeatedly refining the relation starting with a partition that groups accepting and rejecting states together yielding a polynomial-time algorithm of \( O(n^2) \); compare with [5]. Hopcroft’s algorithm [6] for minimization slightly improves the naive implementation to a running time of \( O(m \log n) \) where \( m = |Q \times \Sigma| \) and \( n = |Q| \), where \( \Sigma \) is alphabet of input symbols of the finite automaton, and is up to now the best known minimization algorithm. Recent developments have shown that this bound is tight for Hopcroft’s algorithm [7,8]. Thus, minimization can be seen as a form of lossless compression that can be done effectively while preserving the accepted language exactly.

Recently, a new form of minimization, namely hyper-minimization was studied in the literature [9,10]. There the minimization or compression is done while giving up the preservation of the semantics of finite automata, i.e., the accepted language. It is clear that the semantics cannot vary arbitrarily. A related minimization method based on cover automata is presented in [11,12]. Hyperminimization [9,10] allows the accepted language to differ in acceptance on a finite number of inputs, which is called almost-equivalence. Thus, hyper-minimization aims to find an almost-equivalent DFA that is as small as possible. Here an automaton is hyper-minimal if every other automaton with fewer states disagrees on acceptance for an infinite number of inputs. In [9] basic properties of hyper-minimization and hyper-minimal DFA are investigated. Most importantly, a characterization of hyper-minimal DFA is given, which is similar to the characterization of minimal DFA mentioned above. Namely, a DFA \( M \) is hyper-minimal if and only if in \( M \): (i) there are no unreachable states, (ii) there is no pair of different but equivalent states, and (iii) there is no pair of different but almost-equivalent states, such that at least one of them is a preamble state. Here a state is called a preamble state if it is reachable from the start state by a finite number of inputs, only; otherwise the state is called a kernel state. These properties allow a structural characterization of hyper-minimal DFA. Roughly speaking, the kernels (all states that are kernel states) of two almost-equivalent hyper-minimized automata are isomorphic in the standard sense, and their preambles are also isomorphic, except for acceptance values. Thus, it turns out that hyper-minimal DFA are not necessarily unique. Nevertheless, it was shown in [9] that hyper-minimization can be done in time \( O(m \cdot n^3) \), where
Let \( m = |\Sigma| \) and \( n = |Q| \); for constant alphabet size this gives an \( O(n^3) \) algorithm. Later, the bound was improved to \( O(n^2) \) in \cite{10}. In this paper we improve this upper bound further to \( O(n \log n) \), and argue that it is reasonably well because any upper bound \( t(n) = \Omega(n) \) for hyper-minimization implies that minimization can be done within \( t(n) \). To this end, we linearly reduce minimization to hyper-minimization.

## 2 Preliminaries

Let \( S \) and \( T \) be sets. Their symmetric difference \( S \ominus T \) is \((S \setminus T) \cup (T \setminus S)\). The sets \( S \) and \( T \) are almost-equal if \( S \ominus T \) is finite. A finite set \( \Sigma \) is an alphabet. By \( \Sigma^* \) we denote the set of all strings over \( \Sigma \). The empty string is denoted by \( \varepsilon \). Concatenation of strings is denoted by juxtaposition and \( |w| \) denotes the length of the word \( w \in \Sigma^* \). A deterministic finite automaton (DFA) is a tuple \( (Q, \Sigma, q_0, \delta, F) \) where \( Q \) is a finite set of states, \( \Sigma \) is an alphabet of input symbols, \( q_0 \in Q \) is the initial state, \( \delta : Q \times \Sigma \rightarrow Q \) is a transition function, and \( F \subseteq Q \) is a set of final states. The transition function \( \delta \) extends to \( \delta : Q \times \Sigma^* \rightarrow Q \) as follows: \( \delta(q, \varepsilon) = q \) and \( \delta(q, \sigma w) = \delta(\delta(q, \sigma), w) \) for every \( q \in Q \), \( \sigma \in \Sigma \), and \( w \in \Sigma^* \). The DFA \( M \) recognizes the language \( L(M) = \{ w \in \Sigma^* \mid \delta(q_0, w) \in F \} \).

Two states \( p, q \in Q \) are equivalent, denoted by \( p \equiv q \), if \( \delta(p, w) \in F \) if and only if \( \delta(q, w) \in F \) for every \( w \in \Sigma^* \). The DFA \( M \) is minimal if it does not have equivalent states. The name ‘minimal’ stems from the fact that no DFA with less states also recognizes \( L(M) \) if \( M \) is minimal. It is known that for \( M \) an equivalent minimal DFA can efficiently be computed using Hopcroft’s algorithm \cite{6}, which runs in time \( O(m \log n) \) where \( m = |Q \times \Sigma| \) and \( n = |Q| \).

In the following, let \( M = (Q, \Sigma, q_0, \delta, F) \) be a minimal DFA. Let us recall some notions from \cite{9}. A state \( q \in Q \) is a kernel state if \( q = \delta(q_0, w) \) for infinitely many \( w \in \Sigma^* \). Otherwise \( q \) is a preamble state. We denote the set of kernel states by \( \text{Ker}(M) \) and the set of preamble states by \( \text{Pre}(M) \). For states \( p, q \in Q \) we write \( p \rightarrow q \) if there exists \( w \in \Sigma^+ \) such that \( \delta(p, w) = q \). The states \( p \) and \( q \) are strongly connected, denoted by \( p \leftrightarrow q \), if \( p \rightarrow q \) and \( q \rightarrow p \). Note that strongly connected states are also a kernel states since both are reachable by the minimality of \( M \). Finally, \( q \in Q \) is a center state if \( q \leftrightarrow q \).

## 3 Hyper-minimization

As already remarked, minimization yields an equivalent DFA that is as small as possible. It can thus be considered a form of lossless compression. Sometimes the compression rate is more important than the preservation of the semantics. This leads to the area of lossy compression where the goal is to compress even further at the expense of errors (typically with respect to some error profile). Our error profile is very simple: We allow a finite number of errors. Consequently, we call two DFA \( M_1 \) and \( M_2 \) almost-equivalent if \( L(M_1) \) and \( L(M_2) \) are almost-equal. A DFA that admits no smaller almost-equivalent DFA is called hyper-minimal. Hyper-minimization \cite{9,10} aims to find an almost-equivalent hyper-minimal DFA.
Algorithm 1. Overall structure of the hyper-minimization algorithm

Require: a dfa \( M \)

\[
\begin{aligned}
    M &\leftarrow \text{MINIMIZE}(M) & \text{// Hopcroft’s algorithm; } O(m \log n) \\
    K &\leftarrow \text{COMPUTEKERNEL}(M) & \text{// compute the kernel states; see Section 3.1} \\
    \sim &\leftarrow \text{AEQUIVALENTSTATES}(M) & \text{// compute almost-equivalence; see Section 3.2} \\
    M &\leftarrow \text{MERGESTATES}(M, K, \sim) & \text{// merge almost-equivalent states; } O(m) \\
\end{aligned}
\]

return \( M \)

The contributions [9, 10] report hyper-minimization algorithms for \( M \) that run in time \( O(n^3) \) and \( O(n^2) \), respectively. Note that \(|\Sigma|\) is assumed to be constant in those contributions. Our aim here is to develop a hyper-minimization algorithm that runs in time \( O(n \log n) \) under the same assumptions.

Roughly speaking, minimization aims to identify equivalent states and hyper-minimization aims to identify almost-equivalent states, which we define next. Recall that \( M = (Q, \Sigma, q_0, \delta, F) \) is a minimal dfa. Let \( m = |Q \times \Sigma| \) and \( n = |Q| \).

Definition 1 (cf. [9, Definition 2.2]). For all states \( p, q \in Q \), we say that \( p \) and \( q \) are almost-equivalent, denoted by \( p \sim q \), if there exists \( k \geq 0 \) such that \( \delta(p, w) = \delta(q, w) \) for every \( w \in \Sigma^* \) with \(|w| \geq k\).

Let us present the overall structure of the hyper-minimization algorithm of [10] in Algorithm 1. Note that compared to [10], we exchanged lines 2 and 3. MINIMIZE refers to classical minimization. Hopcroft’s algorithm implements it and runs in time \( O(m \log n) \) [6]. The procedure MERGESTATES is described in [9, 10], where it is also proved that it runs in time \( O(m) \). To make the paper self-contained, we present their algorithm (see Algorithm 2) and the corresponding results next. Note that merging a state \( p \) into another state \( q \) denotes the usual procedure of redirecting (in \( M \)) all incoming transitions of \( p \) to \( q \). If \( p \) was the initial state, then \( q \) is the new initial state. Clearly, the state \( p \) can be deleted.

Theorem 1 ([9, Section 4]). If the requirements of Algorithm 2 are met, then it returns in time \( O(m) \) a hyper-minimal dfa that is almost-equivalent to \( M \).

Consequently, if we can implement: (i) COMPUTEKERNEL and (ii) AEQUIVALENTSTATES in time \( O(m \log n) \), then we obtain a hyper-minimization algorithm that runs in time \( O(m \log n) \). The next two sections will show suitable implementations for both procedures.

3.1 Identify Kernel States

As we have seen in Algorithm 2, kernel states play a special role because we never merge two kernel states. It was already shown in [9, 10], how to identify the kernel states in time \( O(mn) \). It turns out that the kernel states can easily be computed using a well-known algorithm due to Tarjan [13] (see Algorithm 3).

Theorem 2. Ker\((M)\) can be computed in time \( O(m) \).
Algorithm 2. Merging almost-equivalent states

Require: a minimal dfa $M$, its kernel states $K$, and its almost-equivalent states $\sim$

for all $B \in (Q/\sim)$ do
  2: $S \leftarrow B \cap K$  // $S$ contains the kernel states of the block $B$
  if $S \neq \emptyset$ then
    4: select $q \in S$  // select an arbitrary kernel state $q$ from $B$
  else
    6: select $q \in B$  // if no such kernel state exists, pick any state $q$ of $B$
  for all $p \in B \setminus S$ do
    8: merge $p$ into $q$  // merge all preamble states of the block into $q$
return $M$

Proof. Using TARJAN’s algorithm \[13\] (or the algorithms by GABOW \[14,15\] or KOSARAJU \[16,17\]) we can identify the strongly connected components in time $O(m + n)$. Algorithm \[3\] presents a simplified formulation because all states of $M$ are reachable from $q_0$. The initial call is TARJAN($M, q_0$). Thus, we identified states $q$ such that $q \rightarrow q$ because such a state is part of a strongly connected component of at least two states or has a self-loop (i.e., $\delta(q, \sigma) = q$ for some $\sigma \in \Sigma$). Another depth-first search can then mark all states $q$ such that $p \rightarrow p \rightarrow q$ for some state $p$ in time $O(m)$. Clearly, such a marked state is a kernel state and each kernel state is marked because for each $q \in \ker(M)$ there exists a state $p \in Q$ such that $p \rightarrow p \rightarrow q$ by \[9, Lemma 2.12\]. \qed

3.2 Identify Almost-Equivalent States

The identification of almost-equivalent states will be slightly more difficult. We improve the strategy of \[9\], which runs in time $O(mn^2)$, by avoiding pairwise comparisons, which yields a factor $n$, and by merging states with a specific strategy, which reduces a factor $n$ to $\log n$. Since $M$ is a minimal dfa, the relation $\sim$ coincides with the relation defined in \[9, Definition 2.2\]. Thus, we know that $\sim$ is a congruence relation by \[9, Facts 2.5–2.7\].

Let us attempt to explain the algorithm. The vector $(\delta(q, \sigma) \mid \sigma \in \Sigma)$ is called the follow-vector of $q$. The algorithm keeps a set $I$ of states that need to be processed and a set $P$ of states that are still useful. Both sets are initially $Q$ and the hash map $h$ is initially empty. The algorithm then iteratively processes states of $I$ and computes their follow-vector. Since $h$ is initially empty, the first follow-vector will simply be stored in $h$. The algorithm proceeds in this fashion until it finds a state, whose follow-vector is already stored in $h$. It then extracts the state with the same vector from $h$ and compares the sizes of the blocks in $\pi$ that the two states belong to. Suppose that $p$ is the state that belongs to the smaller block and $q$ is the state that belongs to the larger block. Then we merge $p$ into $q$ and remove $p$ from $P$ because it is now useless. In addition, we update the block of $q$ to include the block of $p$ and add all states that have transitions leading to $p$ to $I$ because their follow-vectors have changed due to the merge. The algorithm repeats this process until the set $I$ is empty.
Algorithm 3. Tarjan’s algorithm $\text{TARJAN}(M, q)$ computing the strongly connected components of $M$

Require: a DFA $M = (Q, \Sigma, q_0, \delta, F)$ and a state $q \in Q$

Global: index, low: $Q \rightarrow \mathbb{N}$ initially undefined, $i \in \mathbb{N}$ initially $0$, $S$ stack of states initially empty

2: \text{index}(q) \leftarrow i \quad \text{// set index of } q \text{ to } i; \text{ } q \text{ is thus explored}

low(q) \leftarrow i \quad \text{// set lowest index (of a state) reachable from } q \text{ to the index of } q

4: $i \leftarrow i + 1 \quad \text{// increase current index}$

Push($S, q$) \quad \text{// push state } q \text{ to the stack } S

6: \text{for all } \sigma \in \Sigma \text{ do}

if $\text{index}(\delta(q, \sigma))$ is undefined then

8: $\text{TARJAN}(M, \delta(q, \sigma)) \quad \text{// if successor not yet explored, then explore it}$

low(q) \leftarrow \min(\text{low}(q), \text{low}(\delta(q, \sigma))) \quad \text{// update lowest reachable index for } q

10: else

if $\delta(q, \sigma) \in S$ then

12: low(q) \leftarrow \min(\text{low}(q), \text{index}(\delta(q, \sigma))) \quad \text{// update lowest reachable index}$

if low(q) = index(q) then

14: repeat

$p \leftarrow \text{Pop}(S) \quad \text{// found component; remove all states of it from stack } S$

16: \ldots \quad \text{// store strongly connected components}$

until $p = q$

Example 1. Consider the minimal DFA of Figure 1(left) (see [9, Figure 2]). Let us show the run of Algorithm 4 on it. We present a protocol (for line 10) in Table 1.

At the end of the algorithm the hash map contains the following entries:

<table>
<thead>
<tr>
<th>$B$</th>
<th>$C$</th>
<th>$D$</th>
<th>$E$</th>
<th>$F$</th>
<th>$G$</th>
<th>$H$</th>
<th>$I$</th>
<th>$J$</th>
<th>$K$</th>
<th>$L$</th>
<th>$M$</th>
<th>$N$</th>
<th>$O$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>$D$</td>
<td>$E$</td>
<td>$I$</td>
<td>$M$</td>
<td>$N$</td>
<td>$O$</td>
<td>$P$</td>
<td>$Q$</td>
<td>$R$</td>
<td>$S$</td>
<td>$T$</td>
<td>$U$</td>
<td>$V$</td>
</tr>
</tbody>
</table>

From Table 1, we obtain the partition induced by $\sim$, which is

$$\{\{A\}, \{B\}, \{C, D\}, \{E\}, \{F\}, \{G, H, I, J\}, \{L, M\}, \{P, Q\}, \{R\}\}.$$

This coincides with the partition obtained in [9, Figure 2]. Since $E, F, I, J, L, M, P, Q, \text{ and } R$ are kernel states, we can only merge $C$ into $D$ and merge $G$ and $H$ into $I$. The result of those merges is shown in Figure 1(right). The obtained DFA coincides with the one of [9, Figure 3].

Next, let us look at the time complexity before we turn to correctness. In this respect, line 14 is particularly interesting because it might add to the set $I$, which controls the loop. Our strategy that determines which states to merge will realize the reduction of a factor $n$ to just $\log n$. To simplify the argument,
Algorithm 4. Algorithm computing ~

Require: minimal dfa $M = (Q, \Sigma, q_0, \delta, F)$

for all $q \in Q$ do
2: $\pi(q) \leftarrow \{q\}$ // initial block of $q$ contains just $q$ itself
3: $h \leftarrow \emptyset$ // hash map of type $h : Q^{|\Sigma|} \rightarrow Q$
4: $I \leftarrow Q$ // states that need to be considered
5: $P \leftarrow Q$ // set of current states
6: while $I \neq \emptyset$ do
7: $q \leftarrow \text{RemoveHead}(I)$ // remove state from $I$
8: $\text{succ} \leftarrow (\delta(q, \sigma) \mid \sigma \in \Sigma)$ // compute vector of successors using current $\delta$
9: if $\text{HasValue}(h, \text{succ})$ then
10: $p \leftarrow \text{Get}(h, \text{succ})$ // retrieve state in bucket succ of $h$
11: if $|\pi(p)| \geq |\pi(q)|$ then
12: $\text{Swap}(p, q)$ // exchange roles of $p$ and $q$
13: $P \leftarrow P \setminus \{p\}$ // state $p$ will be merged into $q$
14: $I \leftarrow (I \setminus \{p\}) \cup \{r \in P \mid \exists \sigma : \delta(r, \sigma) = p\}$ // add predecessors of $p$ in $P$ to $I$
15: $\delta \leftarrow \text{MergeState}(\delta, p, q)$ // merge states $p$ and $q$ in $\delta$; $q$ survives
16: $\pi(q) \leftarrow \pi(q) \cup \pi(p)$ // $p$ and $q$ are almost-equivalent
17: $h \leftarrow \text{Put}(h, \text{succ}, q)$ // store $q$ in $h$ under key succ
18: return $\pi$

Fig. 1. An example automaton and the resulting hyper-minimal automaton with $a$-transitions (straight lines) and $b$-transitions (dashed lines). The initial state is $A$.

we will call $\delta(q, \sigma)$ a transition and we consider it the same transition even if the value of $\delta(q, \sigma)$ changes in the course of the algorithm.

Proposition 1. The following properties of Algorithm 4 hold whenever line 7 is executed: (i) $I \subseteq P$ and (ii) $\{\pi(p) \mid p \in P\}$ is a partition of $Q$.

Moreover, let us consider $p$ and $q$ after the execution of line 10. In essence, we would like to show that $p \neq q$. We thus need to show that $(\delta(q, \sigma) \mid \sigma \in \Sigma) \neq \alpha$ for every $\alpha \in h^{-1}(q)$ whenever line 8 is executed. Clearly, when line 8 is first executed with our particular $q$, then $h^{-1}(q) = \emptyset$ and thus the property trivially holds. Moreover, $q$ is then no longer in $I$. It can be added to $I$ in line 14, but only
if \( \delta(q, \sigma) \notin P \) for some \( \sigma \in \Sigma \). Then it is changed in line 15 such that \( \delta(q, \sigma) \in P \) after its execution. Thus, all stored values \( h^{-1}(q) \) have at least one component that is not in \( P \), whereas \( \delta(q, \sigma) \in P \) for every \( \sigma \in \Sigma \) after execution of line 15. Consequently, in line 10 the retrieved state \( p \) cannot be \( q \) itself.

**Lemma 1.** For every \( r \in Q \) and \( \sigma \in \Sigma \), the transition \( \delta(r, \sigma) \) is considered at most \((\log n)\) times in lines 14 and 15 during the full execution of Algorithm 4.

**Proof.** Suppose that \( p = \delta(r, \sigma) \) in line 14. Moreover, \( |\pi(p)| < |\pi(q)| \) by lines 11–12. Then line 15 redirects the transition \( \delta(r, \sigma) \) to \( q \); i.e., \( \delta(r, \sigma) = q \) after line 15. Moreover, \( |\pi(q)| > 2 \cdot |\pi(p)| \) after the execution of line 16 because \( p \neq q \) as already argued, and thus, \( \pi(p) \cap \pi(q) = \emptyset \) by Proposition 1. Moreover, by the same proposition \( |\pi(q)| \leq n \) for every \( q \in Q \). Consequently, \( \delta(r, \sigma) \) can be considered at most \((\log n)\) times in lines 14 and 15. \( \square \)

**Theorem 3.** Algorithm 4 can be implemented to run in time \( O(m \log n) \).

**Proof.** Clearly, we assume that all operations except for those in lines 14 and 15 to execute in constant time. Then lines 1–5 execute in time \( O(n) \). Next we will prove that the loop in lines 6–17 executes at most \( O(m \log n) \) times. By Proposition 1 we have \( I \subseteq P \). Now let us consider a particular state \( q \in Q \). Then \( q \in I \) initially and it has \( |\Sigma| \) outgoing transitions. By Lemma 1 every such transition is considered at most \((\log n)\) times in line 14, which yields that \( q \) is added to \( I \). Consequently, the state \( q \) can be chosen in line 10 at most \((1 + |\Sigma| \cdot \log n)\) times. Summing over all states of \( Q \), we obtain that the loop in lines 6-17 can be executed at most \((n + m \cdot \log n)\) times. Since all lines apart from lines 14 and 15 are assumed to execute in constant time, this proves the statement for all lines apart from 14 and 15. By Lemma 1 every transition is considered at most \((\log n)\) times in those two lines. Since there are \( m \) transitions in \( M \) and each consideration of a transition can be assumed to run in constant time, we obtain that lines 14 and 15 globally (i.e., including all executions of those lines) execute in time \( O(m \log n) \), which proves the statement. \( \square \)

Finally, we need to prove that Algorithm 4 is correct. By Proposition 1 \( \{ \pi(p) \mid p \in P \} \) is a partition of \( Q \) whenever line 7 is executed. Let \( \simeq \) be the induced equivalence relation. Next we prove that \( \simeq \) is a congruence.

**Lemma 2.** Whenever line 7 is executed, \( \pi \) induces a congruence.

This proves that we compute a congruence. Now we can use [9, Lemma 2.10] to prove that all states in a block of the returned partition are almost-equivalent.

**Theorem 4.** The partition returned by Algorithm 4 induces \( \sim \).

**Proof.** Let \( \simeq \) be the congruence (see Lemma 2) returned by Algorithm 4. For every \( \sigma \in \Sigma \) and \( p, q \in Q \) that are merged in line 15 we have \( \delta(p, \sigma) \sim \delta(q, \sigma) \). Thus, \( p \sim q \) by [9, Lemma 2.10], which proves \( \simeq \subseteq \sim \). For the converse, let \( p \sim q \). Clearly, \( \delta \) is the transition function of \( M/\simeq \) at the end of the algorithm. Denote
the transition function of $M/\sim$ by $\delta'$ and the original transition function of $M$ by $\delta$. Since $p \sim q$, there exists $k \geq 0$ such that $\delta(p, w) = \delta(q, w)$ for every $w \in \Sigma^*$ with $|w| \geq k$. Clearly, this yields that $\delta'(p, w) = \delta'(q, w)$ for every such $w$. This implies the existence of $B, D \in (Q/\sim)$ such that $\delta'(B, \sigma) = \delta'(D, \sigma)$ for every $\sigma \in \Sigma$. However, an easy proof shows that the algorithm does not terminate as long as there are distinct states $B$ and $D$ such that $\delta'(B, \sigma) = \delta'(D, \sigma)$ for every $\sigma \in \Sigma$. Consequently, $p \sim q$, which proves the statement. 

**Theorem 5.** For every dfa we can obtain a almost-equivalent, hyper-minimal dfa in time $O(m \log n)$.

### 4 Conclusions

We have designed an $O(m \log n)$ algorithm, where $m = |Q \times \Sigma|$ and $n = |Q|$, that computes a hyper-minimized dfa from a given dfa, which may have fewer states than the classical minimized dfa. Its accepted language is almost-equivalent to the original one; i.e., differs in acceptance on a finite number of inputs only. Since hyper-minimization is a very new field of research, most of the standard questions related to descriptional complexity such as, e.g., nondeterministic automata to dfa conversion w.r.t. hyper-minimality, are problems of further research.

Finally, let’s argue that minimization linearly reduces to hyper-minimization. This is seen as follows: Let $M = (Q, \Sigma, q_0, \delta, F)$ be a dfa. If $L(M) = \emptyset$, which can be verified in time linear in the number of states, then we are already done since the single state hyper-minimal dfa accepting the emptyset is also minimal. Now let $L(M) \neq \emptyset$ and assume $\#$ to be a new input symbol not contained in $\Sigma$. We construct a dfa $M' = (Q, \Sigma \cup \{\#\}, q_0, \delta', F)$ by $\delta'(p, \sigma) = \delta(p, \sigma)$ for $p \in Q$ and $\sigma \in \Sigma$ and $\delta'(p, \#) = q_0$ for $p \in Q$. Observe, that by construction $M'$ consists of kernel states only. Thus, hyper-minimizing $M'$ leads to a dfa $M''$ that is unique because for two almost-equivalent hyper-minimized automata the kernels are isomorphic to each other [7, Theorem 3.5]—compare this with the characterization

<table>
<thead>
<tr>
<th>$I$</th>
<th>$Q \setminus P$</th>
<th>$q$</th>
<th>$p$</th>
<th>$\pi$ (singleton blocks not shown)</th>
</tr>
</thead>
<tbody>
<tr>
<td>${B, \ldots, R}$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\ldots$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>${M}$</td>
<td>${Q}$</td>
<td>$R$</td>
<td>${P, Q}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>${Q}$</td>
<td>$M$</td>
<td>$L$</td>
<td>${P, Q}$</td>
</tr>
<tr>
<td>${H}$</td>
<td>${M, Q}$</td>
<td>$J$</td>
<td>$I$</td>
<td>${L, M}, {P, Q}$</td>
</tr>
<tr>
<td>${F, I}$</td>
<td>${J, M, Q}$</td>
<td>$H$</td>
<td>${I, J}, {L, M}, {P, Q}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>${I}$</td>
<td>${J, M, Q}$</td>
<td>$F$</td>
<td>${I, J}, {L, M}, {P, Q}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>${C, D, G}$</td>
<td>${J, M, Q}$</td>
<td>$I$</td>
<td>$H$</td>
<td>${I, J}, {L, M}, {P, Q}$</td>
</tr>
<tr>
<td>${D, G}$</td>
<td>${H, J, M, Q}$</td>
<td>$C$</td>
<td>${H, I, J}, {L, M}, {P, Q}$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>${G}$</td>
<td>${H, J, M, Q}$</td>
<td>$D$</td>
<td>$C$</td>
<td>${H, I, J}, {L, M}, {P, Q}$</td>
</tr>
<tr>
<td>${B}$</td>
<td>${D, H, J, M, Q}$</td>
<td>$G$</td>
<td>$I$</td>
<td>${C, D}, {H, I, J}, {L, M}, {P, Q}$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>${D, G, H, J, M, Q}$</td>
<td>$B$</td>
<td>$C$</td>
<td>${C, D}, {G, H, I, J}, {L, M}, {P, Q}$</td>
</tr>
</tbody>
</table>
of minimal and hyper-minimal dfa mentioned in the Introduction. Thus, $M''$ is a minimal dfa accepting $L(M')$. Then it is easy to see that taking $M''$ and deleting the $\#$-transitions yields a minimal dfa accepting $L(M)$. Hence, minimization linearly reduces to hyper-minimization. Thus, our algorithm achieves the optimal worst-case complexity in the light of the recent developments for Hopcroft’s state minimization algorithm, which show that the $O(n \log n)$ bound is tight for that algorithm [7] even under any possible implementation [8].

References

On Extremal Cases of Hopcroft’s Algorithm

Giusì Castiglione, Antonio Restivo, and Marinella Sciortino

Università di Palermo, Dipartimento di Matematica e Applicazioni,
via Archirafi, 34 - 90123 Palermo, Italy
{giusi,restivo,mari}@math.unipa.it

Abstract. In this paper we consider the problem of minimization of deterministic finite automata (DFA) with reference to Hopcroft’s algorithm. Hopcroft’s algorithm has several degrees of freedom, so there can exist different sequences of refinements of the set of the states that lead to the final partition. We find an infinite family of binary automata for which such a process is unique. Some recent papers (cf. [13]) have been devoted to find families of automata for which Hopcroft’s algorithm has its worst execution time. They are unary automata associated to circular words. However, automata minimization can be achieved also in linear time when the alphabet has only one letter (cf. [14]), so in this paper we face the tightness of the algorithm when the alphabet contains more than one letter. In particular we define an infinite family of binary automata representing the worst case of Hopcroft’s algorithm. They are automata associated to particular trees and we deepen the connection between the refinement process of Hopcroft’s algorithm and the combinatorial properties of such trees.

1 Introduction

A deterministic finite automaton (DFA) is a recognizer of a regular language and provides a compact representation of the language itself. Among the equivalent deterministic finite automata (i.e. recognizing the same regular language), there exists a unique one (up to isomorphism) with minimal number of states, called minimal automaton of the language. Describing a regular language by its minimal automaton is important in many applications, such as, for instance, text searching, lexical analysis or coding systems, where space considerations are prominent.

Finding the minimal automaton equivalent to a given DFA is a classical and largely studied problem in Theory of Automata and Formal Languages, also called automata minimization problem. Several methods have been developed to minimize a deterministic finite automaton. Some of them operate by successive refinements of a partition of the states. For instance, we recall the well known algorithm proposed by Moore in 1956 (cf. [13]) with time complexity $O(kn^2)$, where $n$ is the number of states of the DFA and $k$ is the cardinality of the alphabet. More efficient is the algorithm provided by Hopcroft in 1971 (cf. [2]) where the refinements are computed in $O(kn \log n)$. Besides, such an algorithm is the fastest known solution to the automata minimization problem.
A taxonomy of finite automata minimization algorithms is given in [15]. Very recently, many papers on experimental comparison of minimization algorithms has been published.

The general complexity of the automata minimization problem is still an open question but there are families of automata for which Hopcroft’s algorithm runs effectively in $O(n \log n)$ (cf. [37]). Such families are unary automata associated to circular words. However, automata minimization can be achieved also in linear time when the alphabet has only one letter (cf. [14]), but the solution does not seem to extend to larger alphabet. In this paper we are focus on finding families of automata defined on more that one letter alphabet representing the worst case of Hopcroft’s algorithm. Actually, we provide an infinite family of binary automata defined by binary labelled trees and relate the execution of Hopcroft’s algorithm on such automata with some combinatorial properties of the associated binary tree. Moreover, for such automata the refinement process leading from the initial partition of set of the states to the final one is uniquely determined. Recall that, in general, Hopcroft’s algorithm has several degrees of freedom since it leaves several choices to the programmer.

The paper is organized as follows. The Section 2 contains the description of Hopcroft’s algorithm by focusing on its degrees of freedom. The Section 3 introduces the notion of standard binary tree and standard tree-like automaton. The uniqueness of the execution of Hopcroft’s algorithm on standard tree-like automata is studied in Section 4. In Section 5 we deepen the problem of tightness of Hopcroft’s algorithm, by providing an infinite family of binary automata representing the worst case of the algorithm. Section 6 describes some sided research topics and future directions.

2 Hopcroft’s Algorithm

In 1971 Hopcroft proposed an algorithm for minimizing a deterministic finite state automaton with $n$ states, over an alphabet $\Sigma$, in $O(|\Sigma| n \log n)$ time (cf. [9]). This algorithm has been widely studied and described by many authors (see for example [10][12][15]) cause of the difficult to give its theoretical justification, to prove correctness and to compute running time.

In Figure 1 we give a brief description of the algorithm’s running.

Given an automaton $A = (Q, \Sigma, \delta, q_0, F)$, it computes the coarsest congruence that saturates $F$. Let us observe that the partition $\{F, Q \setminus F\}$, trivially, saturates $F$. Given a partition $\Pi = \{Q_1, Q_2, ..., Q_m\}$ of $Q$, we say that the pair $(Q_i, a)$, with $a \in \Sigma$, splits the class $Q_j$ if $\delta^{-1}_a(Q_i) \cap Q_j \neq \emptyset$ and $Q_j \not\subseteq \delta^{-1}_a(Q_i)$. In this case, the class $Q_j$ is split into $Q'_j = \delta^{-1}_a(Q_i) \cap Q_j$ and $Q''_j = Q_j \setminus \delta^{-1}_a(Q_i)$. Furthermore, we have that a partition $\Pi$ is a congruence if and only if for any $1 \leq i, j \leq m$ and any $a \in \Sigma$, the pair $(Q_i, a)$ does not splits $Q_j$.

Hopcroft’s algorithm operates by a sequence $\Pi_1, \Pi_2, ..., \Pi_t$ of successive refinements of a partition of the states and it is based on the so-called “smaller half” strategy. Actually, it starts from the partition $\Pi_1 = \{F, Q \setminus F\}$ and refines it by means of splitting operations until it obtains a congruence, i.e. until
Hopcroft Minimization ($A = (Q, \Sigma, \delta, q_0, F)$)

1. $\Pi \leftarrow \{F, Q \setminus F\}$
2. for all $a \in \Sigma$ do
3.     $W \leftarrow \{(\text{min}(F, Q \setminus F), a)\}$
4. while $W \neq \emptyset$ do
5.     choose and delete any $(C, a)$ from $W$
6.     for all $B \in \Pi$ do
7.         if $B$ is split from $(C, a)$ then
8.             $B' \leftarrow \delta_a^{-1}(C) \cap B$
9.             $B'' \leftarrow B \setminus \delta_a^{-1}(C)$
10.            $\Pi \leftarrow \Pi \setminus \{B\} \cup \{B', B''\}$
11.         for all $b \in \Sigma$ do
12.             if $(B, b) \in W$ then
13.                 $W \leftarrow W \setminus \{(B, b)\} \cup \{(B', b), (B'', b)\}$
14.             else
15.                 $W \leftarrow W \cup \{(\text{min}(B', B''), b)\}$

Fig. 1. Hopcroft’s algorithm

no split is possible. To do that it maintains the current partition $\Pi_i$ and a set $W \subseteq \Pi_i \times \Sigma$, called waiting set, that contains the pairs for which it has to check whether some classes of the current partition are split. The main loop of the algorithm takes and deletes one pair $(C, a)$ from $W$ and, for each class $B$ of $\Pi_i$, checks if it is split by $(C, a)$. If it is the case, the class $B$ in $\Pi_i$ is replaced by the two sets $B'$ and $B''$ obtained from the split. For each $b \in \Sigma$, if $(B, b) \in W$, it is replaced by $(B', b)$ and $(B'', b)$, otherwise the pair $(\text{min}(B', B''), b)$ is added to $W$ (with the notation $\text{min}(B', B'')$, we mean the set with minimum cardinality between $B'$ and $B''$). Let us observe that a class is split by $(B', b)$ if and only if it is split by $(B'', b)$, hence, the pair $(\text{min}(B', B''), b)$ is chosen for convenience.

We point out that the algorithm has a degree of freedom because the pair $(C, a)$ to be processed at each step is freely chosen. Another free choice intervenes when a set $B$ is split into $B'$ and $B''$ with the same size and it is not present in $W$. In this case, the algorithm can, indifferently, add to $W$ either $B'$ or $B''$.

Such considerations imply that there can be several sequences of successive refinements that starting from the initial partition $\Pi_1 = \{F, Q \setminus F\}$ lead to the coarsest congruence of the input automaton $A$.

As regards the running time of the algorithm we can observe that the splitting of classes of the partition, with respect to the pair $(C, a)$, takes a time proportional to the cardinality of the set $C$. Hence, the running time of the algorithm is proportional to the sum of the cardinality of all sets processed. Hopcroft proved that the running time is bounded by $O(|\Sigma||Q| \log |Q|)$. In [3] the authors proved that this bound is tight, in the sense that they provided a family of unary automata for which there exist a sequence of refinements such that the time complexity of the algorithm is $\Theta(|\Sigma||Q| \log |Q|)$. However, for the same automata there exist other sequences of refinements producing executions that run in linear time. In [7] we presented a family of unary automata for which
there is a unique sequence of refinements. Moreover we defined a subclass of such automata for which the running time is $\Theta(|\Sigma||Q|\log |Q|)$. Such a subclass of unary automata was extended in [1]. Actually, unary automata represent a very special case for the automata minimization problem. In fact, the minimisation can be achieved also in linear time when the alphabet has only one letter (cf. [14]). So, we are interested in facing both the problem of the uniqueness of the refinements and the tightness of the algorithm when the alphabet contains more than one letter.

In next sections we consider a family of binary automata having a unique sequence of refinements. Moreover we find a class of binary automata for which the running time of Hopcroft’s algorithm is $\Theta(|\Sigma||Q|\log |Q|)$.

3 Standard Trees and Tree-Like Automata

In this section we present a class of binary automata defined by using the notion of binary labelled tree.

Let $\Sigma = \{0, 1\}$ and $A = \{a, b\}$ be two binary alphabets. A binary labelled tree over $A$ is a map $\tau : \Sigma^* \rightarrow A$ whose domain $dom(\tau)$ is a prefix-closed subset of $\Sigma^*$. The elements of $dom(\tau)$ are called nodes, if $dom(\tau)$ has a finite (resp. infinite) number of elements we say that $\tau$ is finite (resp. infinite). The height of a finite tree $\tau$, denoted by $h(\tau)$, is defined as $\max\{|u| + 1, u \in dom(\tau)\}$. We say that a tree $\tilde{\tau}$ is a prefix of a tree $\tau$ if $dom(\tilde{\tau}) \subseteq dom(\tau)$ and $\tilde{\tau}$ is the restriction of $\tau$ to $dom(\tilde{\tau})$. A complete infinite tree is a tree whose domain is $\Sigma^*$. Besides, a complete finite tree of height $n$ is a tree whose domain is $\Sigma^{n-1}$. The empty tree is the tree whose domain is the empty set.

![Binary infinite labeled tree](image)

If $x, y \in dom(\tau)$ are nodes of $\tau$ such that $x = y_i$ for some $i \in \Sigma$, we say that $y$ is the father of $x$ and in particular, if $i = 0$ (resp. $i = 1$) $x$ is the left son (resp. right son) of $y$. A node without sons is called leaf and the node $\varepsilon$ is called the root of the tree. Given a tree $\tau$, the outer frontier of $\tau$ is the set $Fr(\tau) = \{x_i|x \in dom(\tau), i \in \Sigma, x_i \notin dom(\tau)\}$.

Example 1. In Fig[2] an example of an infinite tree $\tau$ is depicted. We have, for instance, that $0111, 1011 \in dom(\tau)$ and $0110, 1001, 1000 \in Fr(\tau)$. 

\[ Fig. 2. Binary infinite labeled tree \]
Let $\tau$ and $\tau'$ be two binary labelled trees. We have that $\tau$ is a subtree of $\tau'$ if there exist a node $v \in \text{dom}(\tau')$ such that:

i) $v \cdot \text{dom}(\tau) = \{vu | u \in \text{dom}(\tau)\} \subseteq \text{dom}(\tau')$

ii) $\tau(u) = \tau'(vu)$ for all $u \in \text{dom}(\tau)$.

In this case we say that $\tau$ is a subtree of $\tau'$ that occurs at node $v$.

In [11] operations among trees have been introduced. Here, we are interested in the concatenation among trees. Roughly speaking, we can say that in order to concatenate two trees $\tau_1$ and $\tau_2$ we attach the root of $\tau_2$ to one of the element of the outer frontier of $\tau_1$. Obviously, since the outer frontier of $\tau_1$ can have more than one element, by concatenating $\tau_1$ and $\tau_2$ we obtain a set of trees. In what follows we use the notion of simultaneous concatenation of $\tau_2$ to all the nodes of $Fr(\tau_1)$ i.e. the tree $\tau_1 \circ \tau_2$ defined as follows:

i) $\text{dom}(\tau_1 \circ \tau_2) = \text{dom}(\tau_1) \cup Fr(\tau_1)\text{dom}(\tau_2);$;

ii) $\forall x \in \text{dom}(\tau_1 \circ \tau_2)$, $\tau_1 \circ \tau_2(x) = \begin{cases} \tau_1(x) & \text{if } x \in \text{dom}(\tau_1) \\ \tau_2(y) & \text{if } x = zy, \ z \in Fr(\tau_1), \ y \in \text{dom}(\tau_2). \end{cases}$

Let $\tau$ be a tree, with $\tau^\omega$ we denote the infinite simultaneous concatenation $\tau \circ \tau \circ \tau \circ \ldots$. Notice that, by infinitely applying the simultaneous concatenation, we obtain a complete infinite tree.

We define factor of a tree a finite complete subtree of the tree, and in the following we are interested in particular factors we define by using some notations given in [5].

Let $\tau$ be a tree, $\sigma$ and $\bar{\sigma}$ two factors of $\tau$ such that $\bar{\sigma}$ is the complete prefix of $\sigma$ of height $h(\sigma) - 1$, then $\sigma$ is called an extension of $\bar{\sigma}$ in $\tau$. A factor $\sigma$ of a tree $\tau$ is extendable in $\tau$ if there exists at least one extension of $\sigma$ in $\tau$.

A factor $\sigma$ of $\tau$ is 2-special if there exist exactly two different extensions of $\sigma$ in $\tau$.

We say that $\gamma$ is a circular factor of $\tau$ if it is a factor of $\tau^\omega$ with $h(\gamma) \leq h(\tau)$. A circular factor $\gamma$ of $\tau$ is a 2-special circular factor if there exist exactly two different extensions of $\gamma$ in $\tau^\omega$ (that we can call circular extensions). The concept of circular factor can be easily understood by noting that in the case of unary tree it coincides with the well-known notion of circular factor of a word.

With reference to a characterization of the notion of circular standard word given in [4], we say that a finite tree $\tau$ is a standard tree if for each $0 \leq h \leq h(\tau) - 2$ it has only a 2-special circular factor of height $h$.

*Example 2.* An example of standard tree, called finite uniform tree, is a complete tree defined by labelling all the nodes at the same level with the same letter taken in the same order it occurs in a given standard word. In Figure 8 we give the uniform tree from the word *abaab*.

Let $A = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton (DFA) over the finite alphabet $\Sigma$, where $Q$ is a finite state set, $\delta$ is a transition function, $q_0 \in Q$ is the initial state and $F \subseteq Q$ the set of final states. Let $G = (V, E)$ be the
transitive directed graph associated to $\mathcal{A}$. We say that $\mathcal{A}$ is a tree-like automaton if $G = (V, E)$ has a subgraph $G_t = (V, E_t)$, containing all nodes $V$, which is a tree (called skeleton) with root $q_0$, and such that all edges of $E \setminus E_t$ are edges from a node to an ancestor.

Given a finite binary labelled tree $\tau$ we can uniquely associate a tree-like automaton $\mathcal{A}_\tau$ having $\tau$ as skeleton and such that for each missing edge we add a transition to the root of the tree. Moreover, the root is the initial state and the states corresponding to nodes labelled by $a$ (resp. $b$) are non-final (resp. final) states.

Example 3. In Fig. 4 a finite labelled tree and the corresponding tree-like automaton are depicted. In the automaton, the initial state labelled by 1 corresponds to the root of the tree.

4 Hopcroft’s Algorithm on Standard Tree-Like Automata

In this section we deepen the connection between the refinements process of Hopcroft’s algorithm when applied on a tree-like automaton associated to a standard tree and the combinatorial properties of the tree itself. By using such properties we prove that such a process is uniquely determined.