Marco Almeida

Equivalence of regular languages: an algorithmic approach and complexity analysis

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto

2010

Marco Almeida

Equivalence of regular languages: an algorithmic approach and complexity analysis

Tese submetida à Faculdade de Ciências da Universidade do Porto

para obtenção do grau de Doutor em Ciência de Computadores

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto

2010

Abstract

Over the last few decades, several alternative algorithms addressing the problems of minimisation and equivalence for finite automata and regular expressions have been proposed. Authors typically present the worst-case time complexity analysis of their algorithms, but that does not provide enough information on the practical behaviour. The lack of existing implementations of several algorithms in one single framework, especially of some of the older algorithms, makes it difficult to determine the running time characteristics.

Using the C and Python programming languages, we implemented random generators of finite automata and regular expressions, several automata minimisation algorithms (including an original, incremental one), some functional variants of a rewrite system to test the equivalence of two regular expressions without explicitly converting them to equivalent automata, and a set of algorithms to test the equivalence of two finite automata without resorting to a minimisation process.

After integrating these tools in the **FAdo** toolkit, and using more that 300 GB of random samples, we experimentally compare the relative performance of these algorithms. The size of the data sets is sufficient to ensure confidence intervals of $95.7\% - 99.5\%$, within a 1% error margin.

Resumo

Ao longo das últimas décadas, foram propostos vários algoritmos relativos aos problemas de minimização e equivalência de autómatos finitos e expressões regulares. Os autores apresentam tipicamente os seus resultados considerando análise de complexidade para o pior caso, mas esse estudo não é suficiente para tirar conclusões acerca do comportamento prático do algoritmo. Mais ainda, a falta de uma biblioteca única que reúna vários destes algoritmos numa implementação comum, dificulta bastante a realização de testes experimentais.

Utilizando as linguagens de programação C e Python, implementamos geradores aleatórios de autómatos finitos e expressões regulares, vários algoritmos de minimização de autómatos, algumas variantes funcionais de um sistema de re-escrita para testar a equivalência de duas expressões regulares sem obter explicitamente os autómatos equivalentes e um conjunto de algoritmos (que não envolvem processos de minimização) para verificar a equivalência de dois autómatos finitos.

Depois de integrar estes algoritmos no projecto **FAdo**, e recorrendo aos cerca de 300 GB de amostras aleatórias que gerámos, comparamos experimentalmente o seu desempenho. O tamanho das amostras é suficiente para garantir intervalos de confiança de $95.7\% - 99.5\%$, com uma margem de erro de 1%.

Résumé

Au cours des dernières décennies, plusieurs algorithmes ont été proposés sur les problèmes de minimisation et d'équivalence des automates finis et des expressions régulières. Généralement, les auteurs présentent les résultats considérant la complexité dans le pire des cas, mais cette étude n'est pas suffisante pour tirer des conclusions sur le comportement pratique des algorithmes. Par ailleurs, l'absence d'un seule bibliothèque qui combine plusieurs de ces algorithmes dans une mise en oeuvre commune, rend très difficile d'effectuer des tests expérimentaux.

En utilisant les langages de programmation C et Python, nous avons implémenter générateurs aléatoires d'automates finis et d'expressions régulières, plusieurs algorithmes de minimisation de automates, certaines variantes fonctionnelles d'un système de réécriture pour tester l'équivalence de deux expressions régulières sans explicitement générer les automates équivalents, et un ensemble d'algorithmes (qui n'impliquent pas de minimisation) pour vérifier l'équivalence de deux automates finis.

Après l'intégration de ces algorithmes dans le projet **FAdo**, et en utilisant environ 300 GB d'échantillons aléatoires, nous avons expérimentalement comparer ses performances. La taille des échantillons est suffisamment grand pour assurer intervalles de confiance à 95.7%−99.5%, avec une marge d'erreur de 1% .

Para os meu pais, que sempre acreditaram.

Acknowledgements

First and foremost, I have to thank my advisers, Nelma Moreira and Rogério Reis, for their support and suggestions over the last four years, their help and contributions writing the papers that form the core of my PhD and are central to this dissertation, which they also thoroughly commented and proofread.

I would also like to thank Sheng Yu for receiving me in Western Ontario. Although only for a short period of time, it was an important part of my academic education that made a difference in the way I see a number of things.

A special note to Mara, that double-checked the proper English grammar on every single sentence of every single chapter.

I must also thank all my friends and family for their permanent support, specially during those (unfortunately frequent) periods of not-so-good humour. I will not mention any names, because, quoting someone who also holds my gratitude: "You know who you are". Needless to say that I am forever grateful for all our nice dinners, late night coffees, long talks over the most varied subjects, quick escapades to watch a movie, music concerts, etc.

Regarding financial support, I thank Fundação para a Ciência e a Tecnologia for the PhD grant, and all other sources that contributed to fund conference participations, summer schools, and research visits namely: LIACC, Fundação Luso-Americana, and projects ASA, RESCUE, and CANTE.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Quoting Rozenberg and Salomaa [62],

> The theory of formal languages constitutes the stem or backbone of the field of science now generally known as theoretical computer science.

In fact, the application of language theory tools, such as finite automata, regular expressions and context-free grammars to the design of fundamental software such as compilers, pattern match tools, text processors, and XML processing to name a few, cannot be neglected and has been of paramount importance over the last *decades*. Just considering the *World Wide Web*, technologies such as CSS, HTML, and XHTML, and the importance of a browser nowadays should be enough to understand the fundamental role that this field of research currently plays.

The advantages and benefits of using computers and software to aid research is another aspect that can no longer be overlooked on formal languages' theory. Random generators, symbolic manipulation tools, and graphical editors simplify the study of finite automata and regular expressions. They facilitate the interchange of ideas among researchers, and may provide important insights on its characterisation. There are already a number of software systems that manipulate automata, regular expressions, grammars, and related structures, and more will certainly be developed. Examples of such systems are AGL, AMoRE, ASTL,

Automate, FADELA, FAdo, FinITE, FIRE Station, FSM, Grail+, INR, JFLAP, MERLin, MONA, Nooj, TESTAS, Turing's World, Unitex, Vaucanson, WFSC, and Whale Calf.

Over the last few decades, several alternative (and increasingly efficient) algorithms addressing the problems of minimisation, equivalence, containment, etc. for automata and regular expressions have been proposed. Authors typically present the worst-case time complexity analysis of their algorithms, but that does not provide enough information on the practical behaviour. The lack of existing implementations of several algorithms in one single framework, especially of some of the older algorithms, makes it difficult to determine the running time characteristics.

Using the C and Python programming languages, we implemented:

- random generators for deterministic finite automata, non-deterministic finite automata, and regular expressions;
- several well-known automata minimisation algorithms, as well as a new incremental quadratic algorithm;
- some functional variants of a rewrite system to test the equivalence of two regular expressions without explicitly converting them to equivalent automata;
- several algorithms to test the equivalence of two finite automata without resorting to any minimisation process.

After integrating these tools in the **FAdo** toolkit, and using the random samples of finite automata and regular expressions, we experimentally compared those algorithms relative performance when:

- minimising both deterministic and non-deterministic finite automata, comparing the performance of the new incremental algorithm with the older ones;
- testing the equivalence of samples of random regular expressions, comparing the new direct method with the more usual approach that explicitly converts the regular expressions into finite automata;

- testing the equivalence of both deterministic and non-deterministic finite automata, comparing the new direct method with the more usual approach that explicitly minimises the input finite automata;

This required almost 21 000 hours (875 days) of CPU usage, testing several algorithms on more than 300 GB of data. The initial setup of the system consumed nearly 800 GB of hard disk space.

1.1 Main contributions

The following are the main contributions of this work.

- Implementation of a uniform random generator of deterministic finite automata.
- Development and implementation of a parametrised generator of random non-deterministic finite automata.
- Design and implementation of three relational databases, publicly available, to store data sets of random regular objects: deterministic finite automata, non-deterministic finite automata, and regular expressions. These contain over 300 GB of easily accessible random samples, generated over the course of several months.
- Implementation, experimental study, and comparison of the performance of several finite automata minimisation algorithms. This study was conducted using samples obtained from the uniform random generator (and stored in the database) so that we could draw statistically significant conclusions.
- Development and implementation of a functional variant of a rewrite system to test the equivalence of two regular expressions using the notion of *derivative*. This method was also experimentally compared with the classical approach, which typically uses the equivalent minimal automata.

- Generalisation of the algebraic method described in the previous item to *partial derivatives*.

- Best-case improvement and generalisation to the non-deterministic case of an algorithm to test the equivalence of two deterministic finite automata, without minimising them. These algorithms were also experimentally compared with the classical approach, which typically uses the equivalent minimal automata.

- Relationship between the methods described in the previous three items by means of the notion of derivative of a regular expression and the associated automaton.

- Proof, exemplified by a regular expression, that the exponential worst-case complexity of the equivalence-testing algorithm is actually tight.

- Development and implementation of an original, quadratic, incremental finite automata minimisation algorithm.

- All the source code was integrated on the **FAdo** project, and is freely available from `http://www.ncc.up.pt/FAdo/`.

Publications list

- "On the representation of finite automata". Rogério Reis, Nelma Moreira, and Marco Almeida. In G. Pighizzini, C. Mereghetti, B. Palano, and D. Wotschkes, editors, *Proceedings of the 7th Workshop on Descriptional Complexity of Formal Systems*, pages 269–276, Como, Italy, June 30 – July 2, 2005.

- "Aspects of enumeration and generation with a string automata representation". Marco Almeida, Nelma Moreira, and Rogério Reis. In H. Leung and G.Pighizzini, editors, *Proceedings of the 8th Workshop on Descriptional Complexity of Formal Systems*, pages 58–69, Las Cruces, New Mexico, June 2006.

- "Enumeration and generation with a string automata representation". Marco Almeida, Nelma Moreira, and Rogério Reis. *Theoretical Computer Science*, 387(2):93–102,

2007.

- "On the performance of automata minimization algorithms". Marco Almeida, Nelma Moreira, and Rogério Reis. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Local Proceedings of the 4th Conference on Computability in Europe*, Athens, Greece, July 2008.

- "Exact generation of minimal acyclic deterministic finite automata". Marco Almeida, Nelma Moreira, and Rogério Reis. *International Journal of Foundations of Computer Science*, 19(4):751–765, August 2008.

- "Antimirov and Mosses's rewrite system revisited". Marco Almeida, Nelma Moreira, and Rogério Reis. In O. Ibarra and B. Ravikumar, editors, *Implementation and Application of Automata*, volume 5448 of *Lecture Notes on Computer Science*, pages 46–56. Springer-Verlag, 2008.

- "Testing the Equivalence of Regular Languages". Marco Almeida, Nelma Moreira, and Rogério Reis. In *Journal of Automata, Languages and Combinatorics*, to appear.

- "FAdo and GUItar: tools for automata manipulation and visualization". André Almeida, Marco Almeida, José Alves, Nelma Moreira, and Rogério Reis. In S. Maneth, editor, *Implementation and Application of Automata*, volume 5642 of *Lecture Notes on Computer Science*, pages 65–74. Springer-Verlag, 2009.

- "Incremental DFA minimisation". Marco Almeida, Nelma Moreira, and Rogério Reis. In *Proceedings of the 15th International Conference on Implementation and Application of Automata*, to appear.

1.2 Structure of this dissertation

The following three chapters contain the mathematical preliminaries. Chapter 2 presents some basic mathematical notions and definitions, including notation, topics from discrete

mathematics, language theory, algorithms and data structures, and complexity analysis. In Chapter 3 we introduce finite automata (both deterministic and non-deterministic), and in Chapter 4 we define and expose the notion of regular expression, while relating it to finite automata.

Whenever possible, we present experimental comparative results on the performance of the considered algorithms. Thus being, in Chapter 5, we expose and justify the conditions on which the experimental data was collected, used, and the results analysed. We also thoroughly describe the scenario on which all tests took place.

In Chapter 6 we describe the relational databases of samples of randomly generated deterministic finite automata, non-deterministic finite automata, and regular expressions.

Chapter 7 contains an improved functional approach to a terminating rewrite system for deciding the equivalence of two regular expressions. We prove its correctness, present some experimental comparative results, and also propose an alternative method based on partial derivatives.

Chapters 8 and 9 deal with finite automata minimisation and equivalence, respectively. In Chapter 8 we present our results on implementing and experimentally comparing the relative performance of several finite automata minimisation algorithms. We also present a new incremental minimisation algorithm. In Chapter 9 we consider the problem of deciding the equivalence of two finite automata (deterministic or non-deterministic) without minimising them, presenting and experimentally comparing the implementations.

We finally conclude with some final remarks and possible future work on Chapter 10.

Chapter 2

Preliminaries

In this chapter, we present some basic mathematical notions and definitions which will be repeatedly used throughout this dissertation. These include topics from discrete mathematics (sets, lists, relations), language theory (symbols, words, languages), and computer science (algorithms, data structures, complexity analysis). We will also introduce the notation used in the next chapters, namely, the pseudocode used to describe algorithms.

While we have omitted some proofs and examples of well-known results, we refer to the work of the following authors for further details. Grimaldi [30] was our main reference on topics of discrete mathematics. For disjoint-set algorithms and complexity analysis, we heavily used the work of Cormen et al. [20]. Knuth's "The Art of Computer Programming" [40, 42, 41] is an omnipresent work. We have resorted to one or more of its volumes while searching for information on topics that span from random number generation and statistical tests, to algorithm design and implementation and complexity analysis. Concerning regular languages, automata theory, and regular expressions, we have based several of the results, definitions, and notation presented in this chapter on the works of Hopcroft et al. [35], Kozen [45], Salomaa [63], Shallit [66], Wood [75], and Yu [76].

2.1 Notation, conventions, and basic definitions

Although completely uniform naming conventions are impossible to adopt, we have tried to name variables, functions, and algorithms in a way that corresponds to their usual designation in the literature. We have also adopted the commonly used notation for standard concepts, such as the "big-O" notation. The following conventions will be used throughout the rest of the chapters:

- capital letters ($A, B, C, \ldots$) for arbitrary sets;
- i, j, k, l, m, and n are usually integer variables;
- Σ for alphabets; when more than one is required, we add indexes such as Σ_1 and Σ_2;
- lower case letters from the beginning of the Latin alphabet ($a, b, c, \ldots$) for alphabet symbols;
- lower case letters from the end of the Latin alphabet ($u, v, w, \ldots$) for words;
- L for languages; if more than one is required, we include indexes: $L_1, L_2, \ldots$; when referring to the language of a specific object, such as a regular expression α for example, we may use L_α.
- D for deterministic finite automata and N for non-deterministic finite automata; when more than one is required, we use indexes ($D_1, D_2, \ldots$, or $N_1, N_2, \ldots$);
- usually, n and k for the number of states and size of the alphabet of a finite automata, respectively; n is also usually used for the length of a regular expression;
- p and q (possibly indexed by integers) for states of finite automata;
- lower case Greek letters, such as α, β, and γ, for regular expressions;
- the symbols for logical conjunction, disjunction, implication, equivalence, and exclusive disjunction are, respectively, $\wedge$, $\vee$, $\rightarrow$, $\Leftrightarrow$, and $\dot{\vee}$.

Algorithms and functions are presented with names longer than one letter, chosen to be suggestive of their use. They are typeset with small caps, using hyphens to separate words, for example QUICK-SORT. Algorithms' variables, although being a single letter, sometimes use subscripts, superscripts, hats, overbars, or prime symbols.

Unless otherwise stated, the implicit base for any logarithm is 2, so $\log(1024) = 10$ for example.

2.2 Sets, Lists, and Tuples

Throughout this dissertation, sets are heavily referred to and used on proofs, definitions, algorithms, etc. More often than not, they are sets of words (see Section 2.6). We will also use multisets, lists, and tuples quite extensively. On this Section, we will define these objects, the operations upon them, and describe the notation used.

2.2.1 Sets

A *set* is a collection of *elements* chosen from some predefined *universe* $\mathcal{U}$. A set is said to be *finite* if it contains a finite number of elements; otherwise, it is said to be *infinite*.

We use capital letters, such as A, B, C, ..., to represent sets and lowercase letters to represent elements. For some set A, we write $x \in A$ if x is an element of A, and $x \notin A$ if it is not.

An *empty set*, i.e., a set with no elements, is denoted by $\emptyset$. If S is a finite set, $|S|$ denotes the number of elements or *cardinality* of the set.

A finite set can be designated by listing its elements within braces. For example, $A = \{1, 3, 5, 7, 9\}$ is the set of the first five odd positive integers. The set $B = \{\}$ has no elements thus, $\{\} = \emptyset$. The order in which the members are listed is not important, so $\{5, 3, 9, 1, 7\}$ specifies the same set as $\{1, 3, 5, 7, 9\}$. Moreover, the number of times that

an element is listed is irrelevant, and we usually remove repeated elements for the sake of simplicity. For example, $\{1, 3, 1, 1, 5, 7, 3, 9\}$ also specifies the set of the first five odd positive integers.

Not all sets can be described by simple enumeration of its members (infinite sets, for example). Such sets may be specified by stating the properties that its members must satisfy. In such cases, we use another standard notation: $A' = \{x \mid 1 \leq x \leq 9 \text{ and } x \text{ is odd}\}$. The vertical bar | within the set braces is read "such that", and the properties following | help us determine which elements belong to the set. In this case, A' defines the set of all x, such that

- x is greater than or equal to 1, and
- x is lesser than or equal to 9, and
- x is an odd positive integer.

The sets A and A' contain the same elements, although described in different ways.

The general format of this specification is

$$\{x \mid P_1(x), P_2(x), \ldots, P_n(x)\}$$

meaning the set of all x, from the given universe, satisfying *all* properties P_i, $1 \leq i \leq n$.

If A and B are sets, we say that A is a *subset* of (or is *contained* in) B and write $A \subseteq B$ — or $B \supseteq A$ — if every element of A is an element of B. Moreover, if B contains some element that is not in A, then A is called a *proper subset* of B and this is denoted by $A \subset B$ or $B \supset A$. If A is not contained in B we write $A \not\subseteq B$.

Two sets, A and B, are said to be *equal* if $A \subseteq B$ and $B \subseteq A$. We denote this by $A = B$.

Given two sets A and B, we define the following operations:

- the *union* of A and B: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$,
- the *intersection* of A and B: $A \cap B = \{x \mid x \in A \text{ and } x \in B\}$,

- the *difference* of A and B: $A - B = \{ x \mid x \in A \text{ and } x \notin B \}$.

We say that A and B are *disjoint* when $A \cap B = \emptyset$.

For a given set $A \subseteq \mathcal{U}$, we denote the *complement* of A by $\overline{A}$ and define it by

$$\overline{A} = \mathcal{U} - A = \{ x \mid x \in \mathcal{U}, x \notin A \}.$$

Sometimes we need to refer to the union or intersection of an arbitrary number of sets. We extend the previous notation as follows. Let $A_1, \ldots, A_n$ be sets ($n \geq 1$). The union of the sets is denoted by

$$\bigcup_{i=1}^{n} A_i = \{ x \mid x \in A_i \text{ for some } 1 \leq i \leq n \}.$$

Similarly, their intersection is denoted by

$$\bigcap_{i=1}^{n} A_i = \{ x \mid x \in A_i \text{ for all } 1 \leq i \leq n \}.$$

The *subset* of a set A, denoted by 2^A, is the collection (or set) of all subsets of A. For the set $A = \{ 1, 3, 5 \}$, for example, $2^A = \{ \emptyset, \{1\}, \{3\}, \{5\}, \{1, 3\}, \{1, 5\}, \{3, 5\}, \{1, 3, 5\} \}$. In general, for any finite set A such that $|A| = n \geq 0$, A has 2^n subsets, and thus $|2^A| = 2^n$.

2.2.2 Multisets

A *multiset*, just like a set, is a collection of elements from some universe $\mathcal{U}$. Unlike a set, however, repeated elements are not ignored. The number of times each element appears is called its *multiplicity*. A set can be considered a multiset in which all multiplicities are equal to one. Given a multiset $M = \{ 1, 1, 3, 5, 7, 7, 9 \}$ we can refer to its underlying set; in this case, the underlying set of M is the set $\{ 1, 3, 5, 7, 9 \}$.

The operations on sets can be trivially extended to multisets in a natural way.

2.2.3 Lists

A *list* is an *ordered* multiset. Because it is ordered by position, we can refer to the first, second, and i^{th} elements. Finite lists are enclosed in square brackets, for example,

$$L = [\,000, 010, 011, 101, 111\,].$$

We always use *zero-based indexing*, selecting elements by an index which starts at 0. Thus, the first element of the list L, with index 0, is 000. The third element, 011, is indexed by 2.

An *empty list* is represented by [ ]. The *length* or *size* of a list L is the number of elements it contains and we denote it by $|L|$. Naturally, $|[\,]| = 0$.

2.2.4 Tuples

Let $n \geq 0$ be an integer. An *n-tuple* over some universe $\mathcal{U}$ is a list of fixed length n over that universe. It is represented by enclosing the elements of the list in parentheses. For example, $(\,010, 011, 101, 111)$ is a 4-tuple containing the first four prime numbers in binary. A 0-tuple is written as $(\,)$.

2.3 Asymptotic notation

The order of growth of the running time of an algorithm gives a simple characterisation of it's efficiency and allows us to compare the relative performance of alternative algorithms. For large enough inputs, the multiplicative constants and lower-order terms of the exact running-time (or space) are dominated by the effects of the input size itself. Thus, we consider input sizes large enough to assure that only the order of growth of the running-time (or space) is relevant.

2.3.1 O-notation

This is the notation we use when we have only an *asymptotic upper bound*. It allows us to say that a function of n is less than or equal to another function, up to a constant factor, and in the asymptotic sense as n grows toward infinity. For a given function $f(n)$, we write $f(n) = O(g(n))$ to give an upper bound to $f(n)$, within a constant factor. For all values of n greater than a certain point n_0, the value of $f(n)$ is smaller or equal to $cg(n)$, where c is a positive constant. Although $O(g(n))$ is in fact a set, it is more convenient to abuse the notation and write $f(n) = O(g(n))$ to indicate that $f(n)$ a member of the set $O(g(n))$. This simplifies its usage in equations and inequalities. Figure 2.1, where $f(n) = O(g(n))$, shows the intuition behind the O-notation.

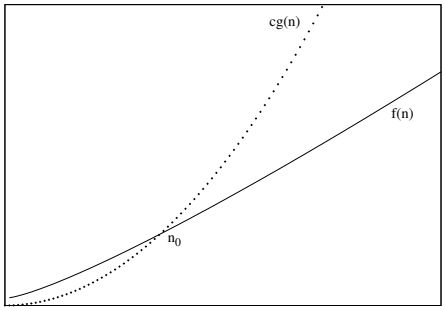


Figure 2.1: Intuition behind the O-notation: $f(n) = O(g(n))$.

Note that when we write $f(n) = O(g(n))$ we are merely stating that some constant multiple of $g(n)$ is an asymptotic upper bound of $f(n)$, with no claim on how tight that upper bound might be. In this context, any *linear* function is $O(n^2)$, for example. This is why we use O-notation to bound the *worst-case* running-time (or space) of an algorithm. As it only describes an upper bound, we get a (possibly pessimistic) limit on *every* input.

2.3.2 Ω-notation

Just like O-notation provides an asymptotic upper bound on a function, Ω-notation provides an *asymptotic lower bound*. For a given function $f(n)$, we write $f(n) = \Omega(g(n))$ to give

a lower bound to $f(n)$, within a constant factor. For all values of n greater than a certain point n_0, the value of $f(n)$ is greater than or equal to $cg(n)$, where c is a positive constant.

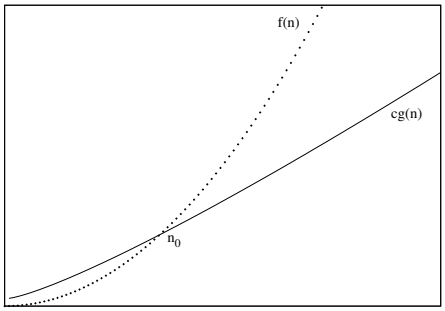


Figure 2.2: Intuition behind the Ω-notation: $f(n) = \Omega(g(n))$.

An example of the Ω-notation is shown in Figure 2.2: for all values of n to the right of n_0, the value of $f(n)$ is above $cg(n)$.

2.3.3 Θ-notation

Sometimes we are able to impose both an upper and a lower asymptotic bound on some function $f(n)$. We call this an *asymptotic tight bound* and we use the Θ-notation to denote it. We say that $f(n)$ is $\Theta(g(n))$, and write $f(n) = \Theta(g(n))$, if there exist two positive constants, c_1 and c_2, such that $f(n)$ can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for a sufficiently large value of n.

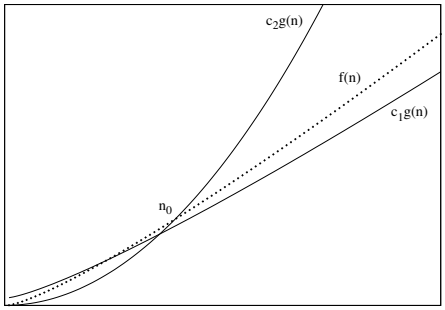


Figure 2.3: Intuition behind the Θ-notation: $f(n) = \Theta(g(n))$.

An example is shown in Figure 2.3. For all values of n to the right of n_0, the value of $f(n)$

lies above $c_1 g(n)$ and bellow $c_2 g(n)$. In other words, the two functions are asymptotically equal up to a constant factor.

2.4 Pseudocode

The syntax of our pseudocode is based on elements of the Python [26] programming language combined with standard mathematical notation (as described on Section 2.1). The functions MERGE and QUICK-SORT, presented bellow, are used as examples to illustrate the following conventions.

- Functions are defined with the keyword **def**. Function's arguments follow its name and are placed inside parentheses, separated by a comma. The function MERGE, for example, is defined at line 1 and takes two arguments: S_1 and S_2.
- Indentation indicates a block structure. The body of the **for** loop on MERGE, for example, begins on line 2 and contains lines 3–5 (line 6 is *outside* the loop). Using indentation instead of the conventional indicators of block structure, such as brackets or **begin** and **end** statements, reduces clutter and enhances readability.
- The looping keywords **while**, **for**, **continue**, and **break** have the same meaning as the ones in Python. Unlike Python, however, we use the usual math symbol $\in$ instead of the **in** keyword when iterating through the elements of a list or set. The **for** loop on MERGE illustrates this syntax.

```
1  def MERGE(S1, S2):
2      for x ∈ S2:
3          if x ∈ S1:
4              continue
5          S1 := S1 ∪ {x}
6      return S1
```

- Variables (such as x, L_1, and L_2 on QUICK-SORT) are local to the given procedure. We do not use global variables without explicit indication.

- We use, rather extensively, four Python built-in data types: dictionaries (associative arrays), lists, tuples, and sets. Dictionaries, lists, and tuples have a syntax similar to that of Python. Given a dictionary d, we use $d[k]$ to access or assign the value of the key k. In the same way, given a list L, $L[0]$ represents the first element, $L[-1]$ the last, and $L[i : j]$ is a *slice* containing the elements $L[i]$, $L[i+1]$, ..., $L[j-1]$. Just like in Python, tuples are immutable sequences that consist of a number of values, separated by commas, enclosed in parentheses. When considering sets, however, we stay away from Python and use standard mathematical notation: $\in$ for membership (lines 2 and 3 of MERGE), $|S|$ for the size of set S (line 2 of QUICK-SORT), set comprehensions (lines 5 and 6 of QUICK-SORT), etc.

- A variable that has not yet been defined has the special value NIL (equivalent to the **None** keyword in Python).

- The Boolean operators $\wedge$, $\vee$, $\Leftrightarrow$, and $\dot{\vee}$ use *minimal evaluation* semantics. Given an expression such as "**if** $x \wedge y$" we start by evaluating x, and the expression y is evaluated if and only if x evaluates to TRUE. Similarly, in the expression "**if** $x \vee y$", we evaluate the expression y if and only if x evaluates to FALSE. This semantic allows us to write expressions such as "$x \neq \text{NIL} \wedge d[x] = y$" without worrying about what happens when we try to access $d[x]$ and x is not defined.

- The assignment and comparison operators are, respectively, $:=$ and $=$. On line 5 of the MERGE procedure, for example, a new value is assigned to the S_1 variable, and on line 2 of QUICK-SORT there is a comparison involving the size of the list L.

```
1  def QUICK-SORT(L):
2      if |L| = 1:
3          return L
4      x := L[0]
5      L1 := { y | y ∈ L, y < x }
```

```
6       L_2 := { z | z ∈ L, z > x }
7       return QUICK-SORT(L_1) ∪ { x } ∪ QUICK-SORT(L_2)
```

- Parameters are passed to a procedure *by value*. The called procedure receives its own copy of the variable, and, should it assign a value to it, the change is *not* reflected on the calling procedure. This means that after an instruction such as MERGE(A, B), for example, the set A will not have been changed, although the procedure MERGE itself updates the argument S_1 several times.

2.5 Disjoint-set data structures and operations

The problem of creating and updating equivalence classes — common to several algorithms presented throughout this dissertation — may be solved in a simple and elegant manner by using a *disjoint-set* data structure. Such a data structure maintains a collection $C = \{ S_1, S_2, \ldots, S_k \}$ of disjoint dynamic sets and allows to efficiently perform two basic operations on it: finding which set a given element belongs to, and uniting two sets. These operations are implemented by what is often called a UNION-FIND algorithm.

Each set is identified by some member chosen to be its *representative*. Testing if two elements are on the same equivalence class corresponds to checking if their representatives are the same. Making two elements equivalent, whilst keeping the transitive closure, is achieved by merging the corresponding sets.

Let x, y, and z denote elements of a set. Following Cormen et al. [20, page 508], the implementation of the UNION-FIND method, used by several algorithms presented in the next chapters, is based on the following disjoint-set operations:

- MAKE(x): creates a new set (singleton) for one element x (the identifier); since the sets are disjoint, x may not already be in another set;
- FIND(x): returns the identifier S_x of the set which contains the element x;

- UNION(x, y): combines the sets identified by x and y into a new set formed by the union of those two sets; S_x and S_y are destroyed and removed from the collection.

```
1  def MAKE(x):
2      p[x] := x
3      rank[x] := 0
4
5  def UNION(x, y):
6      x' := FIND(x)
7      y' := FIND(y)
8      if rank[x'] > rank[y']:
9          p[y'] := x'
10     else:
11         p[x'] := y'
12         if rank[x'] = rank[y']:
13             rank[y'] := rank[y'] + 1
14
15 def FIND(x):
16     if x ∈ p:
17         if x ≠ p[x]:
18             x' := p[x]
19             p[x] := FIND(x')
20     else:
21         MAKE(x)
22     return p[x]
```

Variables p and $rank$ are global associative arrays (dictionaries). The parent of a node x is stored at $p[x]$. Since this implementation includes the *union by rank* heuristic, for each node x we maintain the integer value $rank[x]$ which is an upper bound on the height of x (the number of edges in the longest path between x and a descendant leaf). When a singleton is created, the initial rank of the single node is 0.

The UNION operation must consider two cases. If the roots of the two trees have equal rank, we may arbitrarily choose one of the roots as the parent and increment its rank. If, on the

other hand, the ranks are different, the root with the higher rank becomes the parent but the ranks themselves remain unchanged.

The FIND procedure is a two-pass method which implements the *path compression* heuristic, making each node on the find path point directly to the root. At each call to FIND(x), if x is not already the root, lines 19–20 are executed and it is updated to point directly to the root. The FIND operation does not alter the ranks.

Notice that the FIND procedure presented here is modified so that the set being looked for is created if it does not exist. Whenever FIND(x) fails, MAKE(x) is called and the set $S_x = \{ x \}$ is created. This variant will be useful on some applications of the UNION-FIND procedure as described on later chapters.

2.5.1 Complexity analysis

A UNION-FIND algorithm is useful in many contexts and several approaches have been proposed. Supposing a sequence of $m \geq n$ FIND instructions intermixed with $n - 1$ UNION operations, Hopcroft and Ullman [33] developed an algorithm whose worst-case running time is bounded by the *iterated logarithm*: $O(m \log^{\star}(n))$. The iterated logarithm of n is defined as follows:

$$\log^{\star}(n) = \min \left\{ i \mid \log^{i}(n) \leq 1 \right\}.$$

The following worst-case running time analysis is due to Tarjan [70] who proved a tighter upper bound using the inverse Ackermann function, which grows even slower than the iterated logarithm.

Let $A(x, y)$ denote a faster growing variant of Ackermann's function and $\alpha(x, y)$ its func-

tional inverse. These functions are defined by:

$$A(i,x) = \begin{cases} 2x & \text{if } i = 0, \\ 0 & \text{for } x = 0, \\ 2 & \text{for } x = 1, \\ A(i-1, A(i,x-1)) & \text{for } i \geq 1, x \geq 2; \end{cases}$$

$$\alpha(m,n) = \min\{\, z \geq 1 \mid A\,(z, 4\lceil m/n \rceil) > \log_2(n)\,\}.$$

The function $A(x,y)$ grows so fast that

$$A(4,3) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \quad \overbrace{\qquad\qquad}^{65\,536 \text{ two's}}.$$

Conversely, $\alpha(m,n)$ is so slow-growing that, for all values of n such that $\log(n) < A(4,3)$, $\alpha(m,n) \leq 3$ — notice that the function $\alpha(m,n)$ is maximised when $m = n$. Thus, for all *practical* purposes, $\alpha(m,n) \leq 3$.

Theorem 1. *An arbitrary sequence of $m > n$* FIND *instructions intermixed with $n-1$* UNION *instructions can be performed in $O(m\alpha(m,n))$.*

Proof. Tarjan [70] proves this result considering two extensions to the implementation: the *collapsing rule* and the *weighted union rule*.

Cormen et al. [20, pages 512–517] prove an equivalent result using the potential method of amortised analysis. Two functions are defined, one similar to Ackermann's function and another one similar to its inverse, in order to achieve the same slow growing effect. Two additional heuristics are also considered: *union by rank* and *path compression*. □

2.6 Symbols, Words, and Languages

An *alphabet* is a finite, non empty, set of elements. The elements of an alphabet are called *symbols* or *letters*. Two examples are $A = \{0,1\}$ and $B = \{a,b,c\}$. When referring to an

arbitrary finite alphabet abstractly, we usually denote it by the Greek letter Σ.

Words A *word* (or *string*) over some alphabet Σ is a finite sequence of symbols from Σ. Using the previous examples, 00, 010, and 1101 are words over A, while aba, $aabb$, and a are words over B. The *length* of a word w, denoted by $|w|$, is the number of symbols in w. The word 010, for example, has length 3.

We write a^n to denote a word of a's with length n. For example, $a^3 = aaa$, $a^1 = a$, and $a^0 = \epsilon$. Formally, it is defined inductively:

$$a^0 = \epsilon,$$
$$a^{n+1} = a^n a.$$

The *empty word* (or *null string*) is an empty sequence of symbols, denoted by ϵ. Naturally, $|\epsilon| = 0$. The set of all words over an alphabet Σ, is denoted by $\Sigma^\star$. This is an infinite (enumerable) set of finite-length words. For example,

$$\{a, b\}^\star = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \ldots\},$$
$$\{a\}^\star = \{\epsilon, a, aa, aaa, aaaa, \ldots\}$$
$$= \{a^n \mid n \geq 0\}.$$

We can define $\Sigma^\star$ recursively as follows.

- $\epsilon \in \Sigma^\star$,
- if $w \in \Sigma^\star$, then $aw \in \Sigma^\star$, for all $a \in \Sigma$.

Given two words w and v over an alphabet Σ, the *concatenation* of w with v, denoted by $w \cdot v$ or simply wv, is the word obtained by appending v to w. If, for example $w = abc$ and $v = aba$, $wv = abcaba$ and $vw = abaabc$. The concatenation is an associative operation, so we have that $(xy)z = x(yz)$. Note that $\epsilon\epsilon = \epsilon$, since appending an empty sequence to another empty sequence results in an empty sequence. Moreover, $w\epsilon = \epsilon w = w$ for any word w. This is so because ϵ is not a symbol of the alphabet: it simply denotes

a word with no symbols. By these definitions, the set $\Sigma^\star$, with word concatenation as the associative binary operation and ϵ as the *identity*, is a monoid.

We write w^n for the word obtained by concatenating n copies of w. For example, $(ab)^2 = abab$ and $(ab)^0 = \epsilon$. Formally, it is defined inductively:

$$w^0 = \epsilon,$$
$$w^{n+1} = w^n w.$$

We say that two words w and v are *equal*, and write $w = v$, if they have the same length and the same symbols in the same order.

A *prefix* of a word w is an initial sub-word of w, i.e., u is a prefix of w if there is a word v such that $w = uv$. The word ab, for example, is a prefix of $abcba$. The empty word is a prefix of every word and every word is a prefix of itself. A prefix v of a word w is called *proper* if $v \neq \epsilon$ and $v \neq w$. We define *suffix* and *proper suffix* in an analogous manner.

The *reversal* of a word $w = a_0 a_1 \dots a_n$, denoted w^R, is the word $a_n a_{n-1} \dots a_0$. It is defined inductively by

$$\epsilon^R = \epsilon,$$
$$w^R = v^R a,$$

for $w = av$, $a \in \Sigma$, and $v \in \Sigma^\star$.

Languages A *language* L over an alphabet Σ is a set of words over Σ, i.e., a set $L \subseteq \Sigma^\star$. We denote its cardinality by $|L|$. The *empty language*, denoted by $\emptyset$ or $\{\}$, is a language with no words. The set of all words over Σ, denoted by $\Sigma^\star$, is the *universal language*. Note that $\emptyset$ and $\{\epsilon\}$ are languages over every alphabet. We say that two languages L_1 and L_2 are *equal*, and write $L_1 = L_2$, if they contain the same words.

Apart from the usual Boolean operations on sets — union, intersection, and complement — there are some operations specific to languages. Given two languages $L_1 \subseteq \Sigma_1^\star$ and

$L_2 \subseteq \Sigma_2^\star$ their *concatenation* (or *product*), denoted by $L_1 \cdot L_2$ — or simply $L_1 L_2$ as we usually omit the $\cdot$ operator — is defined by

$$L_1 L_2 = \{ w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2 \}.$$

The concatenation is an associative operation, i.e., for all languages A, B, and C we have that $A(BC) = (AB)C$. It also has an *identity* and a *zero* since $\{\epsilon\}A = A\{\epsilon\} = A$ and $\emptyset A = A\emptyset = \emptyset$. This implies that the set of all languages over some alphabet Σ, $2^{\Sigma^\star}$, is a monoid with respect to concatenation.

Set concatenation *distributes* over union. Thus, for any languages A, B, and C:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

and

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

De Morgan's laws also hold:

$$\overline{A \cup B} = \bar{A} \cap \bar{B},$$
$$\overline{A \cap B} = \bar{A} \cup \bar{B}.$$

The *powers* L^n of a language $L \subseteq \Sigma^\star$ are defined inductively as follows:

$$L^0 = \{\epsilon\},$$
$$L^{n+1} = L L^n,$$

where n is a non-negative integer.

The *star* (or *Kleene closure*) of a language L, denoted by $L^\star$, is the set of all finite powers of L:

$$\begin{aligned} L^\star &= L^0 \cup L^1 \cup L^3 \cup \cdots \\ &= \bigcup_{i=0}^{\infty} L^i. \end{aligned}$$

Similarly, we define L^+ to be the union of all *nonzero powers* of L:

$$L^+ = LL^\star = \bigcup_{i=1}^{\infty} L^i.$$

Given an arbitrary language L, the star operation satisfies the following properties which are often quite useful:

$$\begin{aligned} L^\star L^\star &= L^\star; \\ (L^\star)^\star &= L^\star; \\ L^\star &= \{\epsilon\} \cup LL^\star \\ &= \{\epsilon\} \cup L^\star L; \\ \emptyset^\star &= \{\epsilon\}. \end{aligned}$$

The *reversal* of a language L, denoted by L^R, is the language consisting of the reversals of all its words. It is defined as

$$L^R = \left\{ w^R \mid w \in L \right\}.$$

Chapter 3

Finite automata

In this chapter we will introduce one of the major mechanisms for defining regular languages: the finite automaton. It is one of the simplest and most fundamental computing models with applications in pattern matching, lexical analysis, communication protocols, hardware circuit minimisation, and XML processing, just to name a few examples.

We will define two types of finite automata: deterministic finite automata and non-deterministic finite automata. Both types accept the exact same family of (regular) languages. We will also describe the basic operations — union, intersection, concatenation, and complementation — closure properties, an efficient computational representation, and a uniform random generation model.

Fundamental to finite automata is the concept of state. Intuitively, a state is an instantaneous description of some system. It "contains" all the information necessary to determine how the system can evolve from that point onward. Changes of state are called transitions, which can happen either spontaneously or in response to an external input.

3.1 Deterministic finite automata

Formally, a *deterministic finite automaton* (DFA) is a quintuple $D = (Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of *states*;
- Σ is a non-empty finite set of *input symbols* (alphabet);
- $\delta : Q \times \Sigma \to Q$ is the *transition function*, not necessarily total;
- $q_0 \in Q$ is the *initial state* (or *start state*);
- $F \subseteq Q$ is the set of *final states*.

The *structure* (or *skeleton*) of a deterministic finite automaton, denoted by $\mathrm{DFA}_\emptyset$, is a tuple $A = (Q, \Sigma, \delta, q_0)$. It defines a DFA without its final state information. Each structure is shared by $2^{|Q|}$ DFAs — corresponding to the possible configurations of the final states.

The *size* of a DFA $D = (Q, \Sigma, \delta, q_0, F)$, denoted by $|D|$, is its number of states. Hence $|D| = |Q|$.

We say that two DFAs $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ are *isomorphic*, denoted by $D_1 \simeq D_2$, if there exists a bijection $f : Q_1 \to Q_2$ such that

$$f(q_1) = q_2,$$
$$f(\delta(q, a)) = \delta_2(f(q), a),$$
$$q \in F_1 \Leftrightarrow f(q) \in F_2,$$

for all $q \in Q_1, a \in \Sigma$.

A DFA can also be represented by a *transition diagram*[1]. A transition diagram is a directed graph where:

- each node corresponds to a state;

[1] All transition diagrams in this dissertation were drawn with the VAUCANSON-G [49] package.

- a transition such as $\delta(p, a) = q$ is represented by an arc from node p to node q, labelled by a;
- the initial state is signalled by the unlabelled input arrow;
- final states are represented by a double circle.

As an example, let $D = (\{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_0, q_2\})$ be a DFA such that the transition function is defined as follows:

$$\delta(q_0, a) = q_1,$$
$$\delta(q_0, b) = q_2,$$
$$\delta(q_1, a) = q_2,$$
$$\delta(q_2, a) = q_2.$$

Figure 3.1 shows the transition diagram for the DFA D.

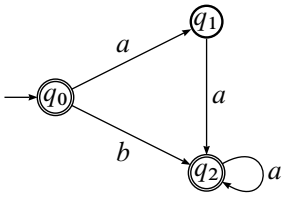


Figure 3.1: Transition diagram of an incomplete DFA.

When the transition function is total, we say that the DFA is *complete*. Note, however, that any incomplete DFA can easily be transformed into a complete one that accepts the same language. On DFA D, for example, states q_1 and q_2 are missing transitions by b. In order to make it complete, we simply add a new state — which we call a *sink state* — and redirect all the missing transitions there. The DFA on Figure 3.2 is complete and equivalent to the finite automaton D.

However, this transition function accepts only single symbols as input. In order to describe what happens when we process a word, i.e., follow a sequence of symbols, we need to define

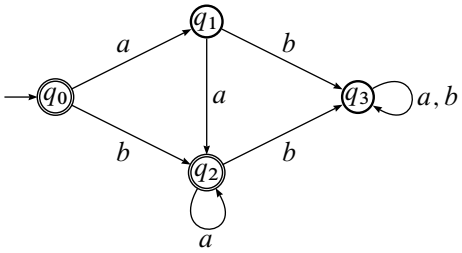


Figure 3.2: Transition diagram of a complete DFA.

an *extended transition function*. This function, denoted by $\hat{\delta}$, returns the state an automaton reaches after processing some sequence of symbols. It is defined inductively as follows:

$$\hat{\delta}(q, \epsilon) = q,$$
$$\hat{\delta}(q, w) = \hat{\delta}(\delta(q, a), u),$$

where $w = au$, for $a \in \Sigma$, $u \in \Sigma^\star$. We say that the words w such that $\hat{\delta}(q_0, w) \in F$ are *accepted* by the DFA.

Lemma 2. *Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Given two words $u, v \in \Sigma^\star$, we have that*

$$\hat{\delta}(\hat{\delta}(q, u), v) = \hat{\delta}(q, uv),$$

for any $q \in Q$.

Proof. We wil proceed by induction on the length of u.

Base: $|u| = 0$.

$$\begin{aligned} \hat{\delta}(\hat{\delta}(q, \epsilon), v) &= \hat{\delta}(q, v) && \text{by definition of } \hat{\delta}, \\ &= \hat{\delta}(q, \epsilon v) && \text{concatenation with the empty word.} \end{aligned}$$

Induction: suppose that $\hat{\delta}(\hat{\delta}(q, u), v) = \hat{\delta}(q, uv)$ for a word u such that $|u| = n$, and let $a \in \Sigma$. We will show that $\hat{\delta}(\hat{\delta}(q, au), v) = \hat{\delta}(q, auv)$.

$$\begin{aligned} \hat{\delta}(\hat{\delta}(q, au), v) &= \hat{\delta}(\hat{\delta}(\delta(q, a)u), v) && \text{by definition of } \hat{\delta}, \\ &= \hat{\delta}(\delta(q, a), uv) && \text{by induction hypothesis,} \\ &= \hat{\delta}(q, auv) && \text{by definition of } \hat{\delta}. \end{aligned}$$

□

On any DFA, a state is called *accessible* if it is reachable from the initial state by some sequence of transitions. Similarly, all states that reach a final state are called *useful*. If all states of a DFA are accessible, we say it is *initially connected* (ICDFA). When all states of an initially connected DFA are useful we say it is *trim*. We denote the skeleton of an ICDFA by $\mathrm{ICDFA}_{\emptyset}$.

The *language* accepted by a DFA $D = (Q, \Sigma, \delta, q_0, F)$, denoted by $L(D)$, is defined by

$$L(D) = \left\{ w \in \Sigma^\star \mid \hat{\delta}(q_0, w) \in F \right\},$$

i.e., the set of words w that result in a sequence of transitions from the start state to any accepting state. Two DFAs D_1 and D_2 are *equivalent*, and we write $D_1 \sim D_2$, if they accept the same language, i.e., $L(D_1) = L(D_2)$.

A DFA is *minimal* if there is no equivalent DFA with fewer states.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Purely for notation simplification reasons, we define the following operator on every state $q \in Q$:

$$\hat{\epsilon}(q) = \begin{cases} 1 & \text{if } q \in F, \\ 0 & \text{if } q \notin F. \end{cases}$$

For two distinct states q_1 and q_2 in Q, we say that q_1 is *distinguishable* from q_2, and write $q_1 \not\sim q_2$, if there exists a word $w \in \Sigma^\star$ such that $\hat{\epsilon}\big(\hat{\delta}(q_1, w)\big) \neq \hat{\epsilon}\big(\hat{\delta}(q_2, w)\big)$. When no such word exists, the states q_1 and q_2 and *equivalent* (or *indistinguishable*) and we write $q_1 \sim q_2$. Formally, we define the equivalence relation $\sim$ on the set of states Q by

$$q_1 \sim q_2 \Leftrightarrow \forall w \in \Sigma^\star \; \hat{\epsilon}\big(\hat{\delta}(q_1, w)\big) = \hat{\epsilon}\big(\hat{\delta}(q_2, w)\big).$$

It is easy to see that the relation $\sim$ is indeed an equivalence relation since it is

- reflexive: $q \sim q$;
- symmetric: if $q_1 \sim q_2$, then clearly $q_2 \sim q_1$;

- transitive: $q_1 \sim q_2$ and $q_2 \sim q_3$ naturally implies that $q_1 \sim q_3$.

As with all equivalence relations, $\sim$ *partitions* the set on which it is defined into disjoint *equivalence classes*: $[\,q\,] = \{\, p \mid p \sim q \,\}$. Every state $q \in Q$ is contained in exactly one equivalence class $[\,q\,]$ and $p \sim q \Leftrightarrow [\,p\,] = [\,q\,]$.

An equivalence relation $\sim$ over Q is called a *right-invariant equivalence relation* if, for all $q_1, q_2 \in Q$, and all $a \in \Sigma$, $q_1 \sim q_2$ implies $\delta(q_1, a) \sim \delta(q_2, a)$.

Lemma 3. *The relation $\sim$ is right-invariant.*

Proof. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA, $q_1, q_2 \in Q$, $a \in \Sigma$, and $u \in \Sigma^\star$.

$$\begin{aligned}
q_1 \sim q_2 &\Leftrightarrow \hat{\epsilon}\big(\hat{\delta}(q_1, au)\big) = \hat{\epsilon}\big(\hat{\delta}(q_2, au)\big) && \text{by definition of } \sim, \\
&\Leftrightarrow \hat{\epsilon}\big(\hat{\delta}(\delta(q_1, a), u)\big) = \hat{\epsilon}\big(\hat{\delta}(\delta(q_2, a), u)\big) && \text{by definition of } \hat{\delta}, \\
&\Leftrightarrow \delta(q_1, a) \sim \delta(q_2, a) && \text{by definition of } \sim.
\end{aligned}$$

□

In any DFA $D = (Q, \Sigma, \delta, q_0, F)$, equivalent states are redundant and can be safely collapsed without changing the language accepted. In the resulting DFA, called the *quotient automaton* and denoted by $D/\!\sim$, each state corresponds to an equivalence class of $\sim$. The formal definition is as follows. Let

$$D/\!\sim = (Q', \Sigma, \delta', q_0', F'),$$

where

$$\begin{aligned}
Q' &= \{\, [q] \mid q \in Q \,\}, \\
\delta'([q], a) &= [\,\delta(q, a)\,], \\
q_0' &= [q_0], \\
F' &= \{\, [q] \mid q \in F \,\}.
\end{aligned}$$

Lemma 4. *For all $q \in Q$ and $w \in \Sigma^\star$, $\hat{\delta}'([q], w) = \big[\,\hat{\delta}(q, w)\,\big]$.*

Proof. The proof follows by induction on the length of w.

Basis: $|w| = 0$.

$$\begin{aligned}\hat{\delta}'([q], \epsilon) &= [q] && \text{by definition of } \hat{\delta}',\\ &= \left[\, \hat{\delta}(q, \epsilon) \,\right] && \text{by definition of } \hat{\delta}.\end{aligned}$$

Induction: let $a \in \Sigma$ and $|w| = n$. Supposing that $\hat{\delta}'([q], w) = \left[\, \hat{\delta}(q, w) \,\right]$, we have to show that $\hat{\delta}'([q], aw) = \left[\, \hat{\delta}(q, aw) \,\right]$.

$$\begin{aligned}\hat{\delta}'([q], aw) &= \hat{\delta}'(\delta'([q], a), w) && \text{by definition of } \hat{\delta}',\\ &= \hat{\delta}'([\,\delta(q, a)\,], w) && \text{by definition of } \delta',\\ &= \left[\, \hat{\delta}(\delta(q, a), w) \,\right] && \text{by induction hypothesis,}\\ &= \left[\, \hat{\delta}(q, aw) \,\right] && \text{by definition of } \hat{\delta}.\end{aligned}$$

□

Theorem 5. *A quotient automaton recognises the same language as the DFA from which it is constructed, i.e., given a DFA* $D = (Q, \Sigma, \delta, q_0, F)$, $L(D/\!\sim) = L(D)$.

Proof. For any $w \in \Sigma^\star$,

$$\begin{aligned}w \in L(D/\!\sim) &\Leftrightarrow \hat{\delta}'(q_0', w) \in F' && \text{by definition,}\\ &\Leftrightarrow \hat{\delta}'([q_0], w) \in F' && \text{by definition,}\\ &\Leftrightarrow \left[\, \hat{\delta}(q_0, w) \,\right] \in F' && \text{by Lemma 4,}\\ &\Leftrightarrow \hat{\delta}(q_0, w) \in F && \text{because } q \in F \Leftrightarrow [q] \in F',\\ &\Leftrightarrow w \in L(D) && \text{by definition.}\end{aligned}$$

□

Lemma 6. *Let* $D_a = (Q_a, \Sigma, \delta_a, s_a, F_a)$ *and* $D_b = (Q_b, \Sigma, \delta_b, s_b, F_b)$ *be two DFAs such that* $D_a \sim D_b$. *We have that*

$$\forall q_a \in Q_a\ \exists q_b \in Q_b\ \forall w \in \Sigma^\star : \hat{\epsilon}\big(\hat{\delta}_a(q_a, w)\big) = \hat{\epsilon}\big(\hat{\delta}_b(q_b, w)\big).$$

Proof. For each $q_a \in Q_a$, let $\sigma(q_a)$ denote the smallest word $v \in \Sigma^\star$ such that $\hat{\delta}_a(s_a, v) = q_a$, and let $q_b = \hat{\delta}_b(s_b, \sigma(q_a))$. We will show that

$$\hat{\epsilon}\big(\hat{\delta}_a(q_a, w)\big) = \hat{\epsilon}\big(\hat{\delta}_b(q_b, w)\big),$$

for $w \in \Sigma^\star$.

By definition, $\hat{\delta}_a(q_a, w) = \hat{\delta}_a(\hat{\delta}(s_a, \sigma(q_a)), w)$ and $\hat{\delta}_b(q_b, w) = \hat{\delta}_b(\hat{\delta}(s_b, \sigma(q_a)), w)$. From Lemma 2, $\hat{\delta}_a(\hat{\delta}(s_a, \sigma(q_a)), w) = \hat{\delta}(s_a, \sigma(q_a)w)$ and $\hat{\delta}_b(\hat{\delta}(s_b, \sigma(q_a)), w) = \hat{\delta}(s_b, \sigma(q_a)w)$. Since $D_a \sim D_b$, clearly $\hat{\epsilon}\big(\hat{\delta}(s_a, \sigma(q_a)w)\big) = \hat{\epsilon}\big(\hat{\delta}(s_b, \sigma(q_a)w)\big)$. □

Theorem 7. *The number of states of a quotient automaton cannot be reduced.*

Proof. Let $D/\!\sim = (Q', \Sigma, \delta', q_0', F')$ be a quotient DFA as described above. We will show, by contradiction, that a DFA $D_m = (Q_m, \Sigma, \delta_m, q_{0_m}, F_m)$ such that $L(D_m) = L(D/\!\sim)$ and $|D_m| < |D/\!\sim|$ does not exist.

Assume that $L(D_m) = L(D/\!\sim)$. We know, from Lemma 6, that for each state $q' \in Q'$ there is at least one state, say $q_m \in Q_m$, such that $\hat{\epsilon}\big(\hat{\delta}'(q', w)\big) = \hat{\epsilon}(\delta_m(q_m, w))$, for every $w \in \Sigma^\star$. Since $|D_m| < |D/\!\sim|$, there must be at least two distinct states, q_1' and q_2', such that

$$\hat{\epsilon}\big(\hat{\delta}'(q_1', w)\big) = \hat{\epsilon}\big(\hat{\delta}_m(q_m, w)\big)$$

and

$$\hat{\epsilon}\big(\hat{\delta}'(q_2', w)\big) = \hat{\epsilon}\big(\hat{\delta}_m(q_m, w)\big).$$

But this would mean that $\hat{\epsilon}\big(\hat{\delta}'(q_1', w)\big) = \hat{\epsilon}\big(\hat{\delta}'(q_2', w)\big)$, which is impossible because, by construction, a quotient automaton has no equivalent states. □

Corollary 8. *A quotient DFA is minimal.*

Proof. The proof follows directly from the definition of minimal DFA, Theorem 5, and Theorem 7. □

Theorem 9. *A minimal DFA is unique up to isomorphism.*

Proof. Given an arbitrary DFA D, we know, from Theorem 7, that any minimal DFA equivalent to D will have the same number of states as the quotient automaton $D/\sim$. We need to show that for any two DFAs $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ such that

$$L(D_1) = L(D_2) = L(D)$$

and

$$|D_1| = |D_2| = |D/\sim|$$

there exists a bijection $f : Q_1 \to Q_2$ such that

$$\begin{aligned} f(q_1) &= q_2, \\ f(\delta(q, a)) &= \delta_2(f(q), a), \\ q \in F_1 &\Leftrightarrow f(q) \in F_2, \end{aligned}$$

for all $q \in Q_1, a \in \Sigma$.

Let $\sigma(q)$ denote the smallest word $w \in \Sigma^\star$ such that $\hat{\delta}_1(q_1, w) = q$, for some $q \in Q_1$. We will show that D_1 and D_2 are isomorphic under the map

$$\begin{aligned} f &: Q_2 \to Q_2 \\ f(q) &= \hat{\delta}_2(q_2, \sigma(q)). \end{aligned}$$

Since both D_1 and D_2 are deterministic, the map f is one-to-one. Because D_2 has no inaccessible states, f is also onto.

The three previous conditions that show that f is an isomorphism of automata (i.e., preserves the initial state, the transition function, and the final states) are argued as follows.

$$\begin{aligned} f(q_1) &= \hat{\delta}_2(q_2, \sigma(q_1)) && \text{by definition of } f, \\ &= \hat{\delta}_2(q_2, \epsilon) && \hat{\delta}_1(q_1, w) = q_1 \Rightarrow w = \epsilon, \\ &= q_2 && \text{by definition of } \hat{\delta}_2. \end{aligned}$$

Let $\sigma(\delta_1(q, a)) = wa$, and consequently $\sigma(q) = w$, for some $q \in Q_1$, $w \in \Sigma^\star$, and

$a \in \Sigma$.

$$\begin{aligned} f(\delta_1(q,a)) &= \hat{\delta}_2(q_2, \sigma(\delta_1(q,a))) && \text{by definition of } f, \\ &= \hat{\delta}_2(q_2, wa) && \text{by definition of } \sigma(\delta_1(q,a)), \\ &= \delta(\hat{\delta}_2(q_2, w), a) && \text{by definition of } \hat{\delta}_2, \\ &= \delta_2(f(q), a) && \text{by definition of } f. \end{aligned}$$

Clearly $q \in F_1 \Leftrightarrow f(q) \in F_2$ for all $q \in Q_1$, because $L(D_1) = L(D_2)$. □

Corollary 10. *A regular language can be univocally identified by the minimal DFA that accepts it.*

Proof. It is a direct consequence of Theorem 9. □

3.1.1 String representation for ICDFAs

The computational representation of a DFA has a significant impact on the amount of resources needed to manipulate that information. A compact representation may help to maintain the data on memory (possibly even in the local caches), avoiding the need of constant page swapping and dramatically improving random I/O access times, which consequently improves the overall performance of an algorithm.

Let us consider only skeletons of DFAs. Following Reis et al. [60, 61], a simple representation of a $\text{DFA}_\emptyset$ can be obtained by the enumeration of its states, keeping, for each state, a set of its transitions for every symbol. Using such a representation for the $\text{DFA}_\emptyset$ of Figure 3.3, for example, we have:

$$\begin{gathered} \{\{q_0, \{(a, q_0), (b, q_1)\}\}, \\ \{q_1, \{(a, q_0), (b, q_4)\}\}, \\ \{q_2, \{(a, q_1), (b, q_4)\}\}, \\ \{q_3, \{(a, q_3), (b, q_2)\}\}, \\ \{q_4, \{(a, q_0), (b, q_4)\}\}\}. \end{gathered}$$

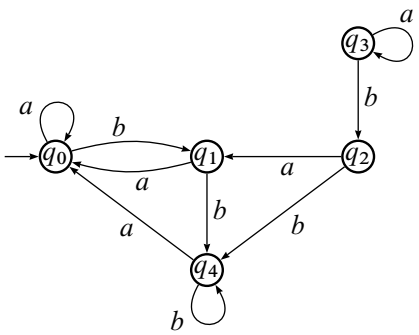


Figure 3.3: An example of a $\text{DFA}_\emptyset$.

If we assume that the $\text{DFA}_\emptyset$ is complete and consider a total order over the alphabet, this representation can be simplified by omitting the alphabetic symbols. For our example, we would have:

$$\begin{aligned}
&\{\{q_0, [q_0, q_1]\},\\
&\ \{q_1, [q_0, q_4]\},\\
&\ \{q_2, [q_1, q_4]\},\\
&\ \{q_3, [q_3, q_2]\},\\
&\ \{q_4, [q_0, q_4]\}\}.
\end{aligned}$$

If we also consider some order over the the labels used for naming the states, we can simplify the representation even further by using it to identify each state. In the example above, it is the natural order over the indexes of the states' names:

$$[[q_0, q_1], [q_0, q_4], [q_1, q_4], [q_3, q_2], [q_0, q_4]].$$

If we consider only complete $\text{DFA}_\emptyset$s, there is exactly one transition for each symbol from a given state and each inner block has a fixed length: the size of the alphabet. This allows us to use a flat list instead of the previous list-of-lists structure:

$$[q_0, q_1, q_0, q_4, q_1, q_4, q_3, q_2, q_0, q_4].$$

It is possible to define a canonical order over the set of the states by exploring the automaton in a breadth-first search, and choosing, at each node, the outgoing edges in alphabetical

order. The procedure is as follows. Using integers to represent the states, let the first state, 0, be the initial state q_0; the second state, 1, is the first state to be accessible from 0 (except for 0 itself); the third state, 2, is the next state to be accessible from 0 (except for 0 and 1), and so on.

This ordered representation is unique and allows to unequivocally identify a complete $\text{ICDFA}_\emptyset$ up to isomorphism. It can not, however, be extended to general $\text{DFA}_\emptyset$s as the example in Figure 3.3 shows. If we consider only the $\text{ICDFA}_\emptyset$ obtained from the first three states (q_0, q_1, and q_4), this method induces in fact an unique order, and the corresponding string representation is

$$[0, 1, 0, 2, 0, 2].$$

Considering the remaining inaccessible states q_2 and q_3, however, the representation is no longer unique and we can arbitrate an order:

$$[0, 1, 0, 2, 0, 2, \mathbf{3, 4, 1, 2}] \text{ or } [0, 1, 0, 2, 0, 2, \mathbf{1, 2, 4, 3}].$$

For each canonical string representing an $\text{ICDFA}_\emptyset$, we can obtain a canonical form for ICDFAs simply by adding a sequence of final states.

Given an arbitrary ICDFA D as input, the function ICDFA-TO-STRING — publicly available from the **FAdo** [22] project — returns a tuple (S, F_S) where S is the unique string representation of D, and F_S the corresponding list of final states. We denote the i^{th} state in the string S by S_i.

```
1  def ICDFA-To-String(D := (Q, Σ, δ, q0, F)):
2  Tf := { }
3  Tr := { }
4  Tf[q0] := 0
5  Tr[0] := q0
6  S := [ ]
7  i := 0
8  j := 0
9  while i ≤ j :
```

```
10        L := [ ]
11        for a ∈ Σ :
12            s := δ(T_r[i], a)
13            if s ∉ T_f :
14                j := j + 1
15                T_f[s] := j
16                T_r[j] := s
17            L := L ∪ [T_f[s]]
18        S := S ∪ [ L ]
19        i := i + 1
20    F_s := [ ]
21    for s ∈ F :
22        F_s := F_s ∪ [T_f[s]]
23    return (S, F_s)
```

This canonical representation can be extended to non-complete ICDFA$_\emptyset$s by naming all missing transitions with an unused value, such as -1. We can assume that those are transitions to an unspecified sink state.

3.1.2 Random generation

Given a canonical string representation of an ICDFA$_\emptyset$ with n states over an alphabet of k symbols, let f_j denote the index of the first occurrence of the state j, for $j \in [1, n-1]$. We call these indexes *flags*. It is easy to see that, in a well formed string,

$$(\forall\, j \in [2, n-1])(f_{j-1} < f_j), \tag{G1}$$

and

$$(\forall m \in [1, n-1])(f_m < km), \tag{G2}$$

which means that $f_1 \in [0, k-1]$ and $f_{j-1} < f_j < kj$, for $j \in [2, n-1]$.

Exact enumeration We can take advantage of this representation to sequentially enumerate all ICDFA$_\emptyset$s with a given number of states n and alphabet size k. Let S be the string

representation of some ICDFA$_\emptyset$ being enumerated. We start by fixating the sequence of flags. The remaining states S_i, such that $i \notin \{ f_j \mid j \in [1, n-1] \}$, can be inserted in the string according to the following rules:

$$i < f_1 \;\Rightarrow\; S_i = 0; \tag{G3}$$

$$(\forall\, j \in [1, n-2])(f_j < i < f_{j+1} \;\Rightarrow S_i \in [0, j]); \tag{G4}$$

$$i > f_{n-1} \;\Rightarrow\; S_i \in [0, n-1]. \tag{G5}$$

Uniform random generation The canonical string representation and the previous set of rules can also be used to develop a strategy to generate uniform random ICDFA$_\emptyset$s (and consequently ICDFAs). The method is as follows:

1. randomly generate a valid sequence of flags, according to G1 and G2;
2. randomly generate the remaining elements of the string, following the rules G3–G5;
3. randomly generate a set of final states.

Step 1 is actually the only one that requires a careful implementation. Consider a random string S representing an ICDFA$_\emptyset$ with 5 states over an alphabet of 2 symbols. According to rule G1, the flag f_1 can be either 0 or 1. There are, however, 14 0450 ICDFA$_\emptyset$s such that $f_1 = 0$ and only 20 225 with $f_1 = 1$. Hence, in order to be uniform, the random generation of flags must consider this and make the first case more probable than the second. Step 2 can be implemented by repeatedly using a uniform random number generator for values in the range $[0, i]$ for $0 \leq i < n$, according to rules G3–G5. Step 3 is also easily implemented with a uniform random integer generator: it suffices to randomly generate a value $i \in [0, 2^n]$, which can be used to identify a block of the subset of the set of states, whose members are set to be the final states.

This uniform random generation process is implemented in a library written in the C programming language. The library, as well as some auxiliary tools, is freely available as part of the **FAdo** project, from the project's website: `http://www.ncc.up.pt/FAdo/`. There

is also an online demonstration web page[2] that uses the library to generate random ICDFAs, outputting the result in several formats: XML, DOT [67], LaTeX, etc.

3.2 Non-deterministic finite automata

A *non-deterministic finite automaton* (NFA) is one for which the next state is not necessarily uniquely determined by the current state and input symbol. While in a DFA there is at most one transition out of each state for each symbol of the alphabet, in a non-deterministic automaton there may be as many as the number of states for each symbol. This means that an NFA has the power to be in several states at once. It does not add, however, any expressive power to the formalism. In fact, any language that can be described by some NFA can also be described by a DFA.

Formally, an NFA is a quintuple $N = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q, and F are defined exactly the same way as for a DFA. The transition function, however, is defined as $\delta : Q \times \Sigma \to 2^Q$, and returns a set of states instead of a single state.

Just like in the deterministic case, we need to extend the transition function δ to a function $\hat{\delta}$ that describes the processing of a word. Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, we define $\hat{\delta}$ by:

$$\begin{aligned} \hat{\delta} &: Q \times \Sigma^\star \to 2^Q \\ \hat{\delta}(p, \epsilon) &= \{ p \} \\ \hat{\delta}(p, w) &= \bigcup_{p' \in \hat{\delta}(p,a)} \hat{\delta}(p', v). \end{aligned}$$

where $w = av$ such that $v \in \Sigma^\star$ and $a \in \Sigma$.

Sometimes it is useful to extend the transition function even further so that it will take a set of states $P \subseteq Q$ and a symbol $a \in \Sigma$, and returns the set of all states accessible from each

[2] `http://www.dcc.fc.up.pt/~mfa/automata/`

$p' \in P$ when reading a. We define this new transition function in the following way:

$$\Delta : 2^Q \times \Sigma \to 2^Q$$
$$\Delta(P, a) = \bigcup_{p' \in P} \delta(p', a).$$

An NFA *accepts* a word $w \in \Sigma^\star$ if, while reading the symbols of w, there is at least a sequence of choices which reaches a final state. Formally, we define the *language* accepted by an NFA $N = (Q, \Sigma, \delta, q_0, F)$, by:

$$L(N) = \left\{ w \mid \hat{\delta}(q, w) \cap F \neq \emptyset \right\}$$

Again, similarly to the deterministic case, we say that two NFAs N_1 and N_2 are *equivalent*, and write $N_1 \sim N_2$, if they accept the same language.

A common extension to NFAs is the use of transitions labelled by ϵ, the empty word. Formally, this special kind of automata (ϵ-NFA) is defined by the same quintuple $N_\epsilon = (Q, \Sigma, \delta, q_0, F)$ as the previous NFAs but the domain of the transition function, defined as $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$, is equipped with the extra symbol. This "feature" allows an NFA to make a transition spontaneously, without reading any input. Although it does not add any expressive power (a language is accepted by an NFA if and only if it is accepted by some ϵ-NFA), it does sometimes simplify the construction of an automaton.

Another generalisation we may consider is a *non-deterministic initial state*. An NFA with such an initial state is defined with the same quintuple as a standard NFA, except that there is a set of initial states rather than exactly one. Thus, the computation of the acceptance of a given word starts from a non-deterministically chosen initial state. We can trivially convert such an NFA to an ϵ-NFA by adding a new initial state i, creating an ϵ-transition from i to each of the other initial states, and then defining i as the only initial state. Clearly, the generalisation does not add any power to the formalism, and this kind of NFA accepts the same family of languages as a standard NFA (or an ϵ-NFA, or a DFA).

3.2.1 Equivalence of non-deterministic and deterministic finite automata

Although, for some languages, an NFA is easier to construct than a DFA, the expressive power of both formalisms is equivalent and any language that can be described by some NFA can also be described by a DFA. Moreover, there is an algorithm for making any NFA N deterministic, i.e., computing an equivalent DFA. This is achieved with the *subset construction* which, in the worst case, constructs all the subsets of the set of states of N. The notion of NFA, its equivalence to DFAs, and the subset construction are results due to Rabin and Scott [59].

Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$ such that $|Q| = n$, we can construct a DFA $D = (Q', \Sigma, \delta', q_0', F')$ in the following way:

$$\begin{aligned} Q' &= 2^Q; \\ \delta'(p, a) &= \Delta(p, a), \quad \text{for } p \in Q', a \in \Sigma; \\ q_0' &= \{ q_0 \}; \\ F' &= \{ p \in Q' \mid p \cap F \neq \emptyset \}. \end{aligned}$$

Notice that this construction implies that $|Q'| = 2^n$ even when the transition function is not total. Consequently, the size of the DFA D will be exponentially large even when some (or most) of the states in Q' are not actually used/necessary. This results in a huge, possibly disconnected DFA. There is a more constructive approach, as demonstrated by the procedure FA-DETERMINISTIC. The blowup in the number of states may sometimes be avoided by assuring that only reachable states are added to the set Q'. Such an implementation of the algorithm follows the transition function δ and incrementally builds the set of states Q'. This assures that a state q will be one of the states of the resulting DFA D if and only if it is used by the transition function δ'.

```
1  def FA-DETERMINISTIC(N := (Q, Σ, δ, q0, F)):
2      Q' := ∅
3      δ' := NIL
```

```
4     V := ∅
5     V := PUSH(V, q0)
6     while V ≠ ∅:
7         p := POP(V)
8         for a ∈ Σ:
9             T := Δ(p, a)
10            δ'(p, a) := T
11            if T ∉ Q':
12                V := PUSH(V, T)
13            Q' := Q' ∪ {T}
14    F' = ∅
15    for p ∈ Q':
16        if p ∩ F ≠ ∅:
17            F' = F' ∪ {p}
18    D := (Q', Σ, δ', {q0}, F')
19    return D
```

3.2.2 Random generation

Lacking a uniform random generator for NFAs, we implemented one which combines the van Zijl bit-stream method, as presented by Champarnaud et al. [14], with one of Leslie's approaches [47]. This allows us to both generate initially connected NFAs and control its transition density. Although Leslie presents a "generate-and-test" method which may never stop, our implementation adds some minor changes that assure the desired properties by construction. A brief explanation of the random NFA generator follows.

Suppose we want to generate a random NFA with n states over an alphabet of k symbols and a transition density d. Let the states (respectively the symbols) be named by the integers $0, \ldots, n-1$ (respectively $0, \ldots, k-1$). A sequence of n^2k bits describes the transition function in the following way: the occurrence of a non-zero bit at the position $ink + jk + a$ denotes the existence of a transition from state i to state j labelled by the symbol a.

Starting with a sequence of all-0 bits, the first step of the algorithm is to create a connected

structure and thus ensure that all the states of the final NFA will be accessible. In order to do so, we define the first state as 0, mark it as visited, generate a transition from 0 to any not-visited state i, and mark i as visited. Next, until all states are marked as visited, randomly choose an already visited state q_1, randomly choose a not-visited state q_2, add a transition from q_1 to q_2 (by a random symbol), and mark q_2 as visited. At this point we have an initially connected NFA and proceed by adding random transitions. Until the desired density is achieved, we simply select one of the bitstream's 0 bits and set it to 1. By maintaining a list of visited states on the first step and keeping record of the 0 bits on the second step, we avoid generating either a disconnected NFA or a repeated transition, and assure that the algorithm will always halt. The set of final states can be easily obtained by generating an equiprobable bitstream of size n and considering final all the states that correspond to a non-zero position in the bitstream.

As an example, consider the bitstream on Figure 3.4 which represents the NFA of Figure 3.5.

$$\underbrace{\overbrace{00}^{q_0}\,|\,\overbrace{11}^{q_1}\,|\,\overbrace{10}^{q_2}}_{q_0}\;\|\;\underbrace{\overbrace{00}^{q_0}\,|\,\overbrace{00}^{q_1}\,|\,\overbrace{00}^{q_2}}_{q_1}\;\|\;\underbrace{\overbrace{01}^{q_0}\,|\,\overbrace{10}^{q_1}\,|\,\overbrace{01}^{q_2}}_{q_2}$$

Figure 3.4: Bitstream representation of the NFA on Figure 3.5.

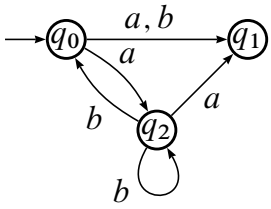


Figure 3.5: The NFA built from the bitstream on Figure 3.4.

Chapter 4

Regular expressions

Just like finite automata, regular expressions are used to recognise regular languages. The formalism was introduced by Kleene [39], who also proved its equivalence in expressive power to finite automata. While finite automata tend to be used in computational implementations — mainly for efficiency reasons — regular expressions provide a good human-readable representation. When defining a token or specifying a string to be matched, for example, finite automata quickly become too cumbersome.

4.1 Basic definition

A *regular expression* over an alphabet Σ is defined inductively in the following way:

- the constants $\emptyset$ and ϵ are regular expressions;
- any symbol $a \in \Sigma$ is a regular expression;
- if α and β are regular expressions, then the *disjunction* (or *union*) $\alpha + \beta$ is a regular expression;
- if α and β are regular expressions, then the *concatenation* $\alpha \cdot \beta$ is a regular expression;

- if α is a regular expression, then $\alpha^\star$ is also a regular expression, denoting the *Kleene closure* or *star* of α.

We usually omit unnecessary parentheses and the concatenation operator $\cdot$, adopting the following conventions:

1. the star operator is of highest precedence (it applies only to the leftmost well-formed regular expression);
2. the concatenation operator comes next in precedence and is left-associative;
3. with the lowest precedence, the disjunction operator, is also left-associative.

Thus, a regular expression such as

$$\alpha_1\beta_1^\star\alpha_2 + \gamma_1 + \gamma_2\beta_2$$

should be read as

$$((((\alpha_1 \cdot \beta_1^\star) \cdot \alpha_2) + \gamma_1) + (\gamma_2 \cdot \beta_2)).$$

The *language* recognised by a regular expression α, denoted by $L(\alpha)$, is defined in the following way. Let β and γ be arbitrary regular expressions.

- $L(\emptyset) = \emptyset$ and $L(\epsilon) = \{\epsilon\}$;
- $L(a) = \{a\}$, for $a \in \Sigma$;
- $L(\beta + \gamma) = L(\beta) \cup L(\gamma)$;
- $L(\beta\gamma) = L(\beta)L(\gamma)$;
- $L(\beta^\star) = L(\beta)^\star$.

We say that two regular expressions α and β are *equivalent*, and write $\alpha \sim \beta$, if they recognise the same language, i.e., $L(\alpha) = L(\beta)$.

We use the *ordinary length*, denoted by $|\alpha|$, as the measure of *size* for a regular expression α. This measure counts the total number of symbols in α, including parentheses and operators. The regular expression $(a + b)^\star(a + \epsilon)$, for example, has size 12. There are other possible measures, like the *alphabetic size*, denoted by $|\alpha|_\Sigma$, which counts only the number of alphabetic symbols in α. Since we will only be considering irreducible regular expressions as defined in Section 4.3, however, there will be no syntactic redundancy and the ordinary length should be enough.

A regular expression α possesses the *empty word property* (e.w.p.) [63] if and only if $\epsilon \in L(\alpha)$, i.e., any of the following conditions holds:

- $\alpha = \epsilon$;
- $\alpha = \beta^\star$ (where β is an arbitrary regular expression);
- α is a disjunction of regular expressions, where at least one possesses the e.w.p.;
- α is a concatenation of regular expressions, all of which possess the e.w.p.

The *constant part* of a regular expression α, denoted by $\hat{\epsilon}(\alpha)$, is ϵ if α has the e.w.p., and $\emptyset$ otherwise.

4.2 Axiomatic system

Let $\mathrm{R_E}$ denote the set of all regular expressions. The algebraic structure $(\mathrm{R_E}, +, \cdot, \emptyset, \epsilon)$, constitutes an idempotent semiring, and, with the unary operator $\star$, a Kleene algebra. There are several well-known complete (non purely equational) axiomatisations of Kleene algebras [64, 44], but we will essentially consider Salomaa's axiom system F_1 [64, 63],

which is defined as follows:

$$\alpha + \emptyset \sim \alpha, \tag{A_1}$$

$$\alpha + \alpha \sim \alpha, \tag{A_2}$$

$$\alpha + \beta \sim \beta + \alpha, \tag{A_3}$$

$$\alpha + (\beta + \gamma) \sim (\alpha + \beta) + \gamma, \tag{A_4}$$

$$\epsilon\alpha \sim \alpha, \tag{A_5}$$

$$\emptyset\alpha \sim \emptyset, \tag{A_6}$$

$$\alpha(\beta\gamma) \sim (\alpha\beta)\gamma, \tag{A_7}$$

$$\alpha(\beta + \gamma) \sim \alpha\beta + \alpha\gamma, \tag{A_8}$$

$$(\alpha + \beta)\gamma \sim \alpha\gamma + \beta\gamma, \tag{A_9}$$

$$\alpha^\star \sim \epsilon + \alpha\alpha^\star, \tag{A_{10}}$$

$$\alpha^\star \sim (\epsilon + \alpha)^\star. \tag{A_{11}}$$

The system also includes the following two rules of inference:

- *Substitution*

$$\frac{\gamma' \sim \gamma[\alpha/\beta], \quad \alpha \sim \beta, \quad \gamma \sim \delta}{\gamma' \sim \delta, \quad \gamma' \sim \gamma}; \tag{R_1}$$

- *Solution of equations*

$$\frac{\alpha \sim \alpha\beta + \gamma, \quad \hat{\epsilon}(\beta) = \emptyset}{\alpha \sim \beta^\star\gamma}. \tag{R_2}$$

We say that two regular expressions are *similar* [11] if one can be transformed into the other using only the Axioms A_1, A_2, A_3, and A_4, and the following identities:

- $\alpha\emptyset \sim \emptyset\alpha \sim \emptyset$,
- $\alpha\epsilon \sim \epsilon\alpha \sim \alpha$.

When not similar, regular expressions are called *dissimilar*.

Unlike Salomaa, we use ϵ instead of $\emptyset^\star$, but $\emptyset^\star \sim \epsilon$ may actually be derived with the following sequence of equations:

$$\begin{aligned} \emptyset^\star &\sim \epsilon + \emptyset\emptyset^\star && \text{by Axiom } A_{10}; \\ &\sim \epsilon + \emptyset && \text{by Axiom } A_6; \\ &\sim \epsilon && \text{by Axiom } A_1. \end{aligned}$$

The disjunction of regular expressions is associative, commutative and idempotent (Axioms A_4, A_3 and A_2, respectively). The concatenation is associative (Axiom A_7), and the star is idempotent. We denote by ACI the set of axioms that includes the associativity, commutativity and idempotence of disjunction, and by $ACIA$ the set ACI plus the associativity of concatenation. Throughout this thesis, when referring to an arbitrary regular expression — unless otherwise stated — we consider all regular operations modulo these properties. This means, for example, that we do not distinguish between the following regular expressions:

$$\begin{gathered} (\alpha + (\beta^\star)^\star) + \gamma(\alpha'\beta') + \alpha, \\ \alpha + (\beta^\star + (\gamma\alpha')\beta' + \alpha), \\ \alpha + \beta^\star + \gamma\alpha'\beta'. \end{gathered}$$

4.3 Succinct regular expressions

We say that a regular expression α is *uncollapsible* [21] if none of the following conditions holds:

- α contains the proper sub-expression $\emptyset$, and $|\alpha| > 1$;
- α contains a sub-expression of the form $\beta\gamma$ or $\gamma\beta$ where $L(\beta) = \{\epsilon\}$;
- α contains a sub-expression of the form $\beta + \gamma$ or $\gamma + \beta$ where $L(\beta) = \{\epsilon\}$ and $\hat{\epsilon}(\gamma) = \epsilon$.

A regular expression α is *irreducible* [21] if it is uncollapsible and the following conditions are true:

- α does not contain superfluous parentheses (we adopt the usual operator precedence conventions and omit outer parentheses);
- α does not contain a sub-expression of the form $(\beta^\star)^\star$.

These reductions allow us to avoid considering that two trivially equivalent regular expressions, such as $\alpha = \emptyset + \epsilon\epsilon a \epsilon\epsilon + \emptyset$ and $\beta = a$, have sizes $|\alpha| = 9$ and $|\beta| = 1$.

Notice that these transformations rely solely on taking regular expressions modulo $ACIA$ and the idempotence of the star operator. Although simple, they allow for a more succinct representation of regular expressions.

4.4 Conversion to finite automata

There are several possible approaches to the problem of transforming a regular expression into a finite automaton which accepts the same language. One of the simplest and possibly more intuitive methods, due to Thompson, transforms a regular expression into an ϵ-NFA. It may, however, generate a considerable number of ϵ-transitions, resulting in an unnecessarily large automaton.

Other methods transform a regular expression into an NFA without ϵ-transitions or even directly into a DFA. The direct conversion to a DFA can be both time and space consuming, but the same result is easily achieved by splitting the process in two separate steps:

1. obtain an NFA from the regular expression;
2. convert it into an equivalent DFA using the subset construction.

In the following, we give a brief description of Thompson's method and of an approach due to Glushkov, based on marked regular expressions.

4.4.1 Thompson's method

The following construction, due to Thompson [71], can be found in several introductory books on automata theory and formal languages, such as Hopcroft et al. [35] or Wood [75], and is implemented on some pattern match tools such as `GNU grep` [23]. Our definition follows the presentation by Hopcroft et al. [35, pages 101–104].

Let α and β be regular expressions. A ϵ-NFA N_α, such that $L(N_\alpha) = L(\alpha)$, is recursively defined as follows.

- $N_\emptyset = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$, where $\delta(q_0, a) = \emptyset$ for all $a \in \Sigma$.

- $N_\epsilon = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$, where $\delta(q_0, \epsilon) = \{q_1\}$ and $\delta(q_0, a) = \emptyset$ for all $a \in \Sigma$.

- $N_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$, where $\delta(q_0, a) = \{q_1\}$ is the only transition.

- Let $N_\alpha = (Q_\alpha, \Sigma, \delta_\alpha, q_\alpha, F_\alpha)$ and $N_\beta = (Q_\beta, \Sigma, \delta_\beta, q_\beta, F_\beta)$ such that $Q_\alpha \cap Q_\beta = \emptyset$.

  - $N_{\alpha+\beta} = (Q, \Sigma, \delta, q_0, \{q_1\})$, where q_0 and q_1 are new states, and

$$
\begin{aligned}
Q &= Q_\alpha \cup Q_\beta \cup \{q_0, q_1\};\\
\delta(q_0, \epsilon) &= \{q_\alpha, q_\beta\};\\
\delta(q, \epsilon) &= \{q_1\}, \text{ for all } q \in F_\alpha \cup F_\beta;\\
\delta(q, a) &= \begin{cases} \delta_\alpha(q, a) & \text{if } q \in Q_\alpha, \\ \delta_\beta(q, a) & \text{if } q \in Q_\beta; \end{cases}
\end{aligned}
$$

  for $a \in \Sigma$.

– $N_{\alpha\beta} = (Q, \Sigma, \delta, q_\alpha, F_\beta)$, where

$$
\begin{aligned}
Q &= Q_\alpha \cup Q_\beta; \\
\delta(q, \epsilon) &= \{ q_\beta \}, \text{ for all } q \in F_\alpha; \\
\delta(q, a) &= \begin{cases} \delta_\alpha(q, a) & \text{if } q \in Q_\alpha, \\ \delta_\beta(q, a) & \text{if } q \in Q_\beta; \end{cases}
\end{aligned}
$$

for $a \in \Sigma$.

– $N_{\alpha^\star} = (Q, \Sigma, \delta, q_0, \{ q_1 \})$, where q_0 and q_1 are new states, and

$$
\begin{aligned}
Q &= Q_\alpha \cup \{ q_0, q_1 \}; \\
\delta(q_0, \epsilon) &= \{ q_\alpha, q_1 \}; \\
\delta(q, \epsilon) &= \{ q_\alpha, q_1 \}, \text{ for all } q \in F_\alpha; \\
\delta(q, a) &= \delta_\alpha(q, a), \text{ for all } q \in Q_\alpha \text{ and } a \in \Sigma.
\end{aligned}
$$

The ϵ-NFA obtained from this construction presents the following properties:

- there is exactly one accepting state;
- there are no arcs going into the initial state;
- there are no arcs coming out of the accepting state.

For a regular expression α such that $|\alpha| = n$, Thompson's method takes $\theta(n)$ time and space to construct an ϵ-NFA N_α with $\theta(n)$ states and $\theta(n)$ transitions [16, 17]. More specifically, N_α will have between n and $2n$ states, and between n and $4n$ transitions.

4.4.2 Glushkov automata

A different approach transforms a regular expression α into a particular NFA without ϵ-transitions. Paraphrasing a description by Brüggemann-Klein and Wood [13], if a word is

defined by some regular expression α, then it must be possible to spell out that word by tracing the appropriate "path" through α. As an example, consider the word $w = abba$ and the regular expression $\alpha = (a + b)^\star a + \epsilon$. Using subscripts to mark each symbol in α, we can rewrite α as $(a_1 + b_2)^\star a_3 + \epsilon$, and determine that $w \in L(\alpha)$ because it corresponds to the path that starts at a_1, visits b_2 twice, and finally arrives at a_3. Naturally, the structure of the regular expression restricts the marked subscripts that can be used to match adjacent symbols of a word. Taking the previous example, for instance, if some symbol in a word is matched by a_3, no further symbol of that word can be matched in α. These restrictions were first formalised by Glushkov [28], and independently, by McNaughton and Yamada [51]. The NFAs obtained through this construction are usually called *Glushkov automata*. It has been claimed [10] that this NFA is the canonical representation because of its natural connection with the derivatives of the original regular expression.

Following Yu's [76] construction, which in turn is based on the presentation by Brüggemann-Klein and Wood [13], we will describe this regular-expression-to-NFA conversion method.

Let α be a regular expression. The Glushkov NFA N_α, such that $L(N_\alpha) = L(\alpha)$, is recursively defined as follows.

- $N_\emptyset = (\{ q_0 \}, \Sigma, \delta, q_0, \emptyset)$, where $\delta(q_0, a) = \emptyset$ for all $a \in \Sigma$.
- $N_\epsilon = (\{ q_0 \}, \Sigma, \delta, q_0, \{ q_0 \})$, where $\delta(q_0, a) = \emptyset$ for all $a \in \Sigma$.
- $N_a = (\{ q_0, q_1 \}, \Sigma, \delta, q_0, \{ q_1 \})$, where $\delta(q_0, a) = \{ q_1 \}$ is the only transition for some $a \in \Sigma$.
- Let $N_\alpha = (Q_\alpha, \Sigma, \delta_\alpha, q_\alpha, F_\alpha)$ and $N_\beta = (Q_\beta, \Sigma, \delta_\beta, q_\beta, F_\beta)$, such that $Q_\alpha \cap Q_\beta = \emptyset$.
  - $N_{\alpha+\beta} = (Q, \Sigma, \delta, q_\alpha, F)$, where

$$Q = Q_\alpha \cup (Q_\beta - \{ q_\beta \}) \text{ (after merging } q_\alpha \text{ and } q_\beta \text{ into } q_\alpha);$$

$$F = \begin{cases} F_\alpha \cup F_\beta & \text{if } q_\beta \notin F_\beta, \\ F_\alpha \cup (F_\beta - \{ q_\beta \}) \cup \{ q_\alpha \} & \text{otherwise;} \end{cases}$$

$$\delta(q,a) = \begin{cases} \delta_\alpha(q_\alpha,a) \cup \delta_\beta(q_\beta,a) & \text{if } q = q_\alpha, \\ \delta_\alpha(q,a) & \text{if } q \in Q_\alpha, \\ \delta_\beta(q,a) & \text{if } q \in Q_\beta; \end{cases}$$

for $a \in \Sigma$.

– $N_{\alpha\beta} = (Q, \Sigma, \delta, q_\alpha, F)$, where

$$Q = Q_\alpha \cup (Q_\beta - \{ q_\beta \}) \text{ (merging each } q \in F_\alpha \text{ with a copy of } q_\beta);$$

$$F = \begin{cases} F_\beta & \text{if } q_\beta \notin F_\beta, \\ F_\alpha \cup (F_\beta - \{ q_\beta \}) & \text{if } q_\beta \in F_\beta; \end{cases}$$

$$\delta(q,a) = \begin{cases} \delta_\alpha(q,a) & \text{if } q \in Q_\alpha - F_\alpha, \\ \delta_\alpha(q,a) \cup \delta_\beta(q_\beta,a) & \text{if } q \in F_\alpha, \\ \delta_\beta(q,a) & \text{if } q \in Q_\beta - \{ q_\beta \}; \end{cases}$$

for $a \in \Sigma$.

– $N_{\alpha^\star} = (Q, \Sigma, \delta, q_\alpha, F)$, where

$$Q = Q_\alpha;$$

$$F = F_\alpha \cup \{ q_\alpha \};$$

$$\delta(q,a) = \begin{cases} \delta_\alpha(q,a) & \text{if } q \in Q_\alpha - F_\alpha, \\ \delta_\alpha(q,a) \cup \delta_\alpha(q_\alpha,a) & \text{if } q \in F_\alpha; \end{cases}$$

for $a \in \Sigma$.

Glushkov automata present the following known properties:

- the starting state has no incoming transitions;
- for a given state, all incoming transitions are labelled by the same symbol;
- given a regular expression α, the number of states of the resulting NFA is always $|\alpha|_\Sigma + 1$.

4.5 Extended regular expressions

As defined on Section 4.1, regular expressions allow only three operations: union, concatenation, and iteration. Complement, intersection, and difference, however, are often quite useful. Because regular languages are closed for complement, intersection, and difference, introducing these operators does not affect the expressive power of regular expressions. A regular expression containing any of these operators is called an *extended regular expression*.

Let α and β be regular expressions. We define extended regular expressions by taking the recursive specification of a regular expression (page 45) and adding the following:

- $\overline{\alpha}$ is an extended regular expression denoting the *complement*;
- the *intersection*, $\alpha \cap \beta$, is an extended regular expression;
- the *difference*, denoted by $\alpha - \beta$, is also an extended regular expression.

Just like with non-extended regular expressions, we omit unnecessary parentheses and the concatenation operator, adopting the same conventions.

The language recognised by an extended regular expression α, denoted $L(\alpha)$, is defined in the following way. Let β and γ be arbitrary extended regular expressions.

- $L(\emptyset) = \emptyset$ and $L(\epsilon) = \{\epsilon\}$;
- $L(a) = \{a\}$, for $a \in \Sigma$;
- $L(\beta + \gamma) = L(\beta) \cup L(\gamma)$;
- $L(\beta\gamma) = L(\beta)L(\gamma)$;
- $L(\beta^\star) = L(\beta)^\star$;
- $L\left(\overline{\beta}\right) = \Sigma^\star - L(\beta)$;

- $L(\beta \cap \gamma) = L(\beta) \cap L(\gamma)$;

- $L(\beta - \gamma) = L(\beta) - L(\gamma)$.

To illustrate the descriptive power of extended regular expressions, consider the problem of specifying a regular expression to recognise C-style comments. Recall that a C comment is a sequence of the form /*...*/, where ... represents any string of characters except */. Let Σ be the alphabet of all valid characters in C and γ represent the disjunction of all symbols $a \in (\Sigma - \{/, *\})$. Moreover, let $\alpha = /*$ and $\beta = */$. A possible regular expression to recognise this language is

$$/*((/ + \gamma) + **^{\star}\gamma)**^{\star}/.$$

Using extended regular expressions, however, we can use the complement operator and obtain a solution which is both shorter and easier to read:

$$\alpha\overline{\Sigma^{\star}\beta\Sigma^{\star}}\beta.$$

Because of this ability to be reduced, however, some problems require even more time to be solved (assuming that time is measured as a function of the length of the regular expression). The *emptiness-of-complement problem*, for example, which consists on determining whether the complement of a given regular expression denotes the empty set,

- is in P-SPACE for ordinary regular expressions [1, 395–399];

- at least of exponential space (and therefore time, since NP-TIME is contained in P-SPACE) complexity[1] $c'c^{\sqrt{n/\log(n)}}$, for regular expressions extended with intersection but not with complement [1, 411–418];

- not decidable by any elementary space-bounded Turing machine for the full class of extended regular expressions [1, 419–423].

[1] For a regular expression of size n, and constants $c' > 0$, $c > 1$.

4.6 Linear forms

A regular expression α is said to be *linear* if it is generated by the following context free grammar:

$$\begin{aligned} L &\to S \mid S \cdot R \mid L + L \\ R &\to S \mid R + R \mid R \cdot R \mid R^\star \\ S &\to a \in \Sigma, \end{aligned} \tag{G_1}$$

where L is the initial symbol. Hence, a linear regular expression α is of the form

$$a_1\alpha_1 + \cdots + a_n\alpha_n$$

where α_i is an arbitrary regular expression and $a_i \in \Sigma$.

We say that an expression $a_i\alpha_i$ has *head* a_i and *tail* α_i and use $\textsc{Head}(\alpha)$ and $\textsc{Tail}(\alpha)$ to denote, respectively, the multiset of all heads and the multiset of all tails in α. If each element in $\textsc{Head}(\alpha)$ occurs exactly once, the linear regular expression α is called *deterministic*.

A regular expression is called *pre-linear* if it is generated by following context free grammar:

$$\begin{aligned} P &\to \emptyset \mid Q \\ Q &\to L \mid Q \cdot R \mid Q + Q \\ L &\to S \mid S \cdot R \mid L + L \\ R &\to S \mid R + R \mid R \cdot R \mid R^\star \\ S &\to a \in \Sigma, \end{aligned} \tag{G_2}$$

where P is the initial symbol. Less formally, a pre-linear regular expression is either $\emptyset$, an already linear regular expression or a disjunction of concatenations where the first argument of each concatenation is a pre-linear regular expression. Just like with linear regular expressions, we say that $a_i\alpha_i$ has head a_i and tail α_i.

4.7 Derivatives

The *derivative* [12] of a regular expression α with respect to a *symbol* $a \in \Sigma$, written $a^{-1}(\alpha)$, is a regular expression β such that:

$$L(\beta) = \{ w \mid aw \in L(\alpha) \}.$$

The formal recursive definition is the following:

$$\begin{aligned}
a^{-1}(\emptyset) &= \emptyset; \\
a^{-1}(\epsilon) &= \emptyset; \\
a^{-1}(\alpha) &= \begin{cases} \epsilon & \text{if } \alpha = a, \\ \emptyset & \text{if } \alpha \neq a\ ; \end{cases} \\
a^{-1}(\alpha + \beta) &= a^{-1}(\alpha) + a^{-1}(\beta); \\
a^{-1}(\alpha\beta) &= a^{-1}(\alpha)\beta + \hat{\epsilon}(\alpha)a^{-1}(\beta); \\
a^{-1}(\alpha^\star) &= a^{-1}(\alpha)\alpha^\star.
\end{aligned}$$

where α, β are arbitrary, not necessarily irreducible, regular expressions.

Notice that in the particular case of a deterministic linear regular expression, the computation of the derivative of a regular expression α with respect to a symbol a can be simplified as follows:

$$a^{-1}(\alpha) = \begin{cases} \beta & \text{if } a\beta \text{ is an operand of } \alpha, \\ \epsilon & \text{if } \alpha = a, \\ \emptyset & \text{otherwise.} \end{cases}$$

The notion of derivative can easily be extended to words in a natural way. Let $w^{-1}(\alpha)$ denote the derivative of the regular expression α with respect to a word $w \in \Sigma^\star$, such that

$w = ua$ for $a \in \Sigma$. It is recursively computed in the structure of w as follows:

$$\epsilon^{-1}(\alpha) = \alpha;$$
$$w^{-1}(\alpha) = (ua)^{-1}(\alpha) = a^{-1}(u^{-1}(\alpha)).$$

4.7.1 Partial derivatives

Partial derivatives [5] are a generalisation to the non-deterministic case of the notion of derivative. Let α be a regular expression and $\alpha' = a^{-1}(\alpha)$. The set of *partial derivatives* of a regular expression α with respect to a symbol $a \in \Sigma$, denoted by $\partial_a(\alpha)$, can be seen as the set of the operands of the disjunction α'. The following recursive definition computes the set of partial derivatives of an arbitrary regular expression with respect to a symbol $a \in \Sigma$.

$$\partial_a(\emptyset) = \{\};$$
$$\partial_a(\epsilon) = \{\};$$
$$\partial_a(\alpha) = \begin{cases} \{\epsilon\}, & \text{if } \alpha = a; \\ \{\}, & \text{otherwise;} \end{cases}$$
$$\partial_a(\alpha + \beta) = \partial_a(\alpha) \cup \partial_a(\beta);$$
$$\partial_a(\alpha\beta) = \partial_a(\alpha)\beta \cup \hat{\epsilon}(\alpha)\partial_a(\beta);$$
$$\partial_a(\alpha^\star) = \partial_a(\alpha)\alpha^\star.$$

Let $A = \{\alpha_1, \ldots, \alpha_n\}$ be a set of regular expressions and $\sum A$ denote the disjunction constructed with the elements of A, i.e.,

$$\sum A = \alpha_1 + \cdots + \alpha_n.$$

The notion of partial derivatives of a regular expression α can trivially be extended to words

as follows. Let $w \in \Sigma^\star$ such that $w = ua$ and $a \in \Sigma$.

$$\partial_\epsilon(\alpha) = \{\alpha\};$$
$$\partial_{ua}(\alpha) = \partial_a\Big(\sum \partial_u(\alpha)\Big).$$

Clearly,

$$L(a^{-1}(\alpha)) = L\Big(\sum \partial_a(\alpha)\Big) \quad \text{and} \quad L(\partial_w(\alpha)) = L\Big(\sum w^{-1}(\alpha)\Big).$$

The set of all partial derivatives of a regular expression α with regard to all words $w \in \Sigma^\star$, denoted by $P_\partial(\alpha)$, is finite and bounded by the number of alphabetic symbols in α.

4.8 Representation and implementation

We always consider irreducible regular expressions modulo associativity of the concatenation and disjunction, commutativity of the disjunction, and idempotence of both disjunction and star operations. For each operator, we use a data structure that enforces these properties and simplifies the processes which assure that the regular expressions are kept irreducible.

Disjunctions are represented as a set of regular expressions. Concatenated regular expressions are kept in an ordered list. The idempotence of the Kleene star is assured by not allowing double stared regular expressions. The implementation is object-oriented, using a different class for each operator.

4.8.1 Disjunctions

A disjunction is represented as a set of regular expressions. This presents us with a natural way to enforce the ACI properties:

- *associativity*: there is no pairwise association of any two arguments, so

$$(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma);$$

- *commutativity*: by definition, the order of the elements in a set is irrelevant, thus

$$\alpha + \beta = \beta + \alpha;$$

- *idempotence*: also by definition, a set contains no repeated elements, hence

$$\alpha + \alpha = \alpha.$$

As for making a disjunction irreducible, only two conditions must hold:

- the sub-expression $\emptyset$ may not occur;
- if $\hat{\epsilon}(\alpha) = \epsilon$, ϵ is not allowed as a sub-expression.

An algorithm to maintain a disjunction irreducible is linear in the number of operands. As an example, consider the regular expression

$$\alpha + \emptyset + \beta^\star\gamma + \alpha + \epsilon,$$

represented by the set

$$\{\epsilon, \alpha, \beta^\star\gamma\}.$$

4.8.2 Concatenation

Concatenations of regular expressions are kept in an ordered list. This allows us to take advantage of the associative property and easily apply transformations to any pair of adjacent regular expressions. This representation also simplifies the following transformations:

$$\alpha\epsilon \to \alpha,$$
$$\epsilon\alpha \to \alpha,$$
$$\alpha\emptyset \to \emptyset,$$
$$\emptyset\alpha \to \emptyset.$$

These are necessary to make the regular expressions irreducible, as we can simply iterate through the list and remove each occurrence of ϵ. If $\emptyset$ is found, we may safely return it as the equivalent irreducible regular expression. Consider the following examples:

$$\alpha\epsilon\beta \to [\alpha, \beta],$$
$$\alpha\emptyset\beta \to \emptyset,$$
$$\alpha\beta^\star\gamma \to [\alpha, \beta^\star, \gamma].$$

4.8.3 Kleene star

In order to keep starred regular expressions irreducible, the constructor of the class does not create regular expressions of the form $\alpha^{\star\star}$. This is done by checking if the regular expression passed as an argument is already of the same type. When this is the case, only the argument is kept, thus avoiding the double star.

We also add the following two simplifications to the star operator representation:

$$\emptyset^\star \to \epsilon \quad \text{and} \quad \epsilon^\star \to \epsilon.$$

Although these are not necessary to make the regular expressions irreducible, they allow for significant simplifications which greatly increasing the overall efficiency of some algorithms.

Chapter 5

Experimental tests

Apart from the theoretical worst-case run-time analysis, which authors typically present, not much is known about the practical performance of finite automata and regular expression manipulation algorithms. Such information is essential, however, when for example, one has to choose an appropriate algorithm for a given application. Throughout this dissertation, whenever possible, we present experimental comparative results on the performance of the considered algorithms.

Collecting and presenting experimental data in a fair way is never an easy task. The quality of the input data, hardware choice, running operating system, file system used, general configuration of the machine, etc., may impact on an algorithm's performance, skewing the results. In this Chapter we carefully describe and justify the scenario on which all the experimental tests were performed.

Since we are primarily interested in the relative performance of the algorithms, all tests were executed on the same machine, under the exact same conditions. Depending on the test-case, we collected a number of statistics in order to classify each algorithm's behaviour: running time, memory usage, and recursion depth.

5.1 Related work

Concerning finite automata minimisation algorithms, Bruce Watson [74] presented some experimental results, but based on rather small and biased data sets. Using an ad-hoc random automata generator, Tabakov and Vardi [69] experimentally compared Hopcroft's and Brzozowski's algorithms. Baclet and Pagetti [7] analysed different implementations of the equivalence class refinement process in Hopcroft's algorithm. More recently, Bassino et al. [9, 8] presented some experimental comparative results of Moore and Hopcroft's DFA minimisation algorithms (using samples obtained from a uniform random generator) and studied the average complexity of Moore's minimisation algorithm

5.2 Sampling

Even considering only non-isomorphic ICDFAs, the total number of automata with n states over an alphabet of k symbols grows so fast (cf. Appendix A) that trying to apply an algorithm to every single ICDFA is simply not feasible — even for rather small values of n and k. The same applies to NFAs and regular expressions.

In order to evaluate the performance of a given algorithm, we must therefore take some *random samples* of these huge universes and use them as the input data. It is important, however, that these samples are of a *reasonable* size. Too large a sample implies a waste of the resources which we were trying to save in the first place; a sample too small diminishes the utility of the results. By fixing a confidence interval and a confidence level (margin of error), we can calculate the correct sample size and thus assure that the results are suitable for drawing statistically significant conclusions. According to Cochran [18, page 75], given a margin of error ϵ and an estimated proportion p, we may use the formula

$$N = \frac{z^2 p(1-p)}{\epsilon^2} \tag{5.1}$$

to determine the appropriate sample size N. By application of the central limit theorem, we can assume that the data is normally distributed and take z from the abscissa of the normal

curve that cuts off an area of the confidence level γ at the tails, i.e.,

$$P(-z < Z < z) = \gamma.$$

As we know that the population is large (cf. Appendix A) but do not have a proper estimate for the proportion p, we assume $p = \frac{1}{2}$ for maximum variability. We may then simplify Equation 5.1 in the following way:

$$N = \frac{z^2}{4\epsilon^2} = \left(\frac{z}{2\epsilon}\right)^2. \tag{5.2}$$

5.3 Random input

When gathering random samples, it is important to know the model followed by the random data, i.e., the probability distribution used by the generator. Statistically supported conclusions may only be drawn from tests on "good" randomly generated objects. Biased samples will either produce wrongful results or make the estimation of population parameters impossible. A *uniform random generator* produces unbiased outputs by assuring that any finite number of values are equally likely to be generated.

All our tests on ICDFAs were performed on samples obtained from the uniform random generator described on Subsection 3.1.2. Thus, the results are conclusive, statistically significant, and reflect the trends of the population.

The samples of regular expressions were also obtained from a uniform random generator. We implemented the method described by Mairson [50] for the generation of context-free languages using a grammar of almost irreducible regular expressions proposed by Shallit [65].

The random generator of NFAs, however, does not follow a uniform probability distribution. In fact, it is rather a kind of ad-hoc random generator that enforces some properties on the output (cf. Subsection 3.2.2). Hence, the results of the experimental tests on the NFA samples may, or may not, describe the actual behaviour of the population.

5.4 The environment

Each process was allowed to run during a maximum period of 24 hours and use up to 2 GB of RAM. This time limit was enforced by the `psmon` tool [4] which was setup to issue a `SIGTERM` signal to any process that exceeded it. All code controlling the benchmarks was instrumented in order to catch `kill` signals. This allowed us to save the statistics on the amount of work that was completed at the time of a forced halt — elapsed time, number of tests performed, memory usage, recursive calls count, etc.

5.4.1 Hardware and Software

All algorithms were implemented in the Python [26] programming language using similar data structures whenever possible. Efforts were made to implement the algorithms as efficiently as possible, while preserving readability.

The tests were performed with version 2.6 of the Python interpreter [25] on the same computer: two Intel Xeon 5140 2.33 GHz dual-core processors with 4 GB of fully buffered RAM, running a minimal 64 bit Debian GNU/Linux [58] system[1]. The amount of physical memory was sufficient to avoid page swapping effects and only essential operating system processes were allowed to run concurrently with the benchmarks.

5.4.2 Sample sizes

Using the random generators described on Subsections 3.2.2 and 3.1.2, we produced samples of random NFAs and ICDFAs, respectively, with the following characteristics.

[1] Debian *squeeze* with a stock kernel (version 2.6.32)

ICDFAs	
States	$\{5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$
Symbols	$\{k \mid 2 \leq k \leq 16\} \cup \{18, 20, 25, 30, 35, 40, 45, 50\}$
States	$\{1000\}$
Symbols	$\{2, 3, 5\}$

NFAs	
States	$\{5, 10, 20, 40, 60, 80, 100\}$
Symbols	$\{k \mid 2 \leq k \leq 16\} \cup \{18, 20, 25, 30, 35, 40, 45, 50\}$
Transition density	$\{0.1, 0.5, 0.8\}$

The samples of regular expressions were also obtained with a uniform random generator according to the following criteria.

Regular expressions	
Length	$\{10, 20, 30, 40, 50, 75, 100, 125, 150, 175, 200, 250, 300\}$
Alphabet size	$\{k \mid 2 \leq k \leq 16\} \cup \{18, 20, 25, 30, 35, 40, 45, 50\}$

Each sample contains 20 000 elements. By direct application of Equation 5.2, we can easily see that this more than suffices to ensure results statistically significant with a 99.5% confidence level and a 1% error margin:

$$N = \left(\frac{z}{2\epsilon}\right)^2 = \left(\frac{2.576}{0.02}\right)^2 = 16590.$$

Some of the tests were applied to pairs of automata or regular expressions. On such cases we can only consider data sets with 10 000 objects, but these are still quite enough for a 95.7% confidence interval with a 1% error margin:

$$N = \left(\frac{z}{2\epsilon}\right)^2 = \left(\frac{1.960}{0.02}\right)^2 = 9604.$$

5.4.3 Statistics collection

During all tests, only typical Linux processes ran in the background. Running times were obtained using the GNU `time` utility [24]. Since this tool relies on the `getrusage` system call to read the resource usage from the `/proc` filesystem (at least on Linux systems), we have access to the individual time measures: total elapsed time, actual CPU assigned time, and the time devoted to the operating system during the execution of the program. By considering only the CPU time, we assure that no other process (including the background ones) skew the algorithms' performance data.

Memory usage was measured with a Python module included in the **FAdo** [22] toolkit. It reads the memory usage information directly from `/proc/<pid>/status`, providing instant information about any running process. This allows us to get the memory usage details at several execution points of a benchmark and account only for the memory consumed by the actual algorithms, subtracting the space used by auxiliary code such as database connection modules.

We did not use any methods to disable the cache memory. Any frequently accessed data structures that were not too large to fit in a first level cache (usually from the less complex algorithms) certainly benefited from it.

Chapter 6

Random objects database

Both the quality and ease of availability of the input data have a significant impact on the elaboration of experimental tests. Having some consistent, reasonable sized, random samples readily available greatly simplifies the tasks of preparing or even repeating some experimental test.

Using the random generators described on the previous chapters, and the PostgreSQL relational database system [57], we have designed and implemented three relational databases to store datasets of random ICDFAs, NFAs, and regular expressions. In order to assure that we could draw statistically significant conclusions when using the datasets, all databases were developed according to the requirements described in Chapter 5, using the random generators described in Section 5.3. The number of objects stored in each database, as well as the respective space they consume on disk, are presented on Table 6.1.

6.1 Database of random ICDFAs

The random ICDFAs' database stores random samples of 20 000 automata with the following characteristics:

- $n \in \{5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ states, each over an alphabet of $k \in$

Object type	**Number**	**Size**
ICDFAs	4 600 000	22 GB
NFAs	9 660 000	279 GB
Regular Expressions	5 980 000	31 GB
Total	20 240 000	332 GB

Table 6.1: Summary of the random objects databases.

$\{x \mid 2 \leq x \leq 16\} \cup \{18, 20, 25, 30, 35, 40, 45, 50\}$ symbols;

- $n \in \{1000\}$ states, each over an alphabet of $k \in \{2, 3, 5\}$.

Besides the structure of each ICDFA, the database also stores some relevant complementary information such as minimality, being trimmed, acyclic, etc. This allows to obtain, with a simple SQL query, some random automata with specific properties. The entity-relationship model of this database is presented on Figure 6.1. Although, at the moment, it is composed by a single table, future plans include a relation for equivalent ICDFAs, so that one can easily relate non-minimal equivalent DFAs with different sizes.

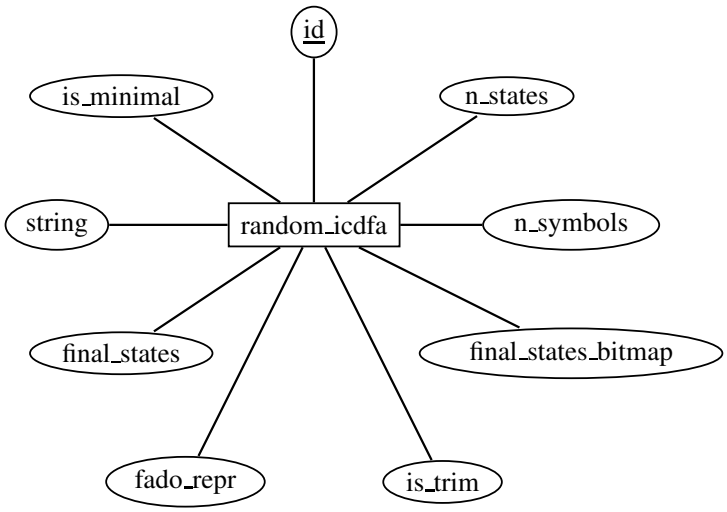


Figure 6.1: Entity-relationship diagram of the database of random ICDFAs.

For efficiency reasons, besides its unique string representation [60], the database stores the pre-parsed internal representation of the **FAdo** toolkit for each ICDFA. This avoids the need to parse an automaton's description every time we need to manipulate it. For similar reasons, each automaton's final states set is stored in two different ways: as a comma separated list of integers, and as a bitmap. It is important to note that this replication of data does not contradict any of the five normal forms of the relational database modelling. Indeed, it is simply a trade-off between time and space.

Since, by its own nature, this database will be primarily used with read-only requests (`SELECT` queries), we used the table partitioning features of PostgreSQL to split the large automata table into smaller physical pieces. We implemented "range partitioning", using the number of states of the DFAs as the delimiter. This resulted in a database with 11 sub-tables: the (empty) single parent table which exists just to represent the entire data set, and a child table that contains all ICDFA with a given number of states. This division of the search space dramatically improves the performance, sometimes by a factor of 10.

Still considering fast `SELECT` queries, we explicitly created an index in the `n_states` column, another in the `n_symbols` column, and a compound one on both these columns. The main purpose is to speed-up the typical queries, which tend to involve both these parameters, as the following example — which requests a sample of 100 ICDFAs, each with 10 states over an alphabet of 2 symbols — shows:

```
SELECT fado_repr FROM random_icdfa WHERE
    n_states=10 AND
    n_symbols=2
    LIMIT 100;
```

6.2 Database of random NFAs

The database of random NFAs follows a structure quite similar to the one of random ICDFAs, described on the previous section.

It stores samples of 20 000 random NFAs with

$$n \in \{5, 10, 20, 40, 60, 80, 100\}$$

states, each over an alphabet of

$$k \in \{2, 3, 4, \ldots, 16, 18, 20, 25, 30, 35, 40, 45, 50\}$$

symbols, and with a transition density,

$$d \in \{0.1, 0.5, 0.8\}.$$

The entity-relationship model is presented on Figure 6.2.

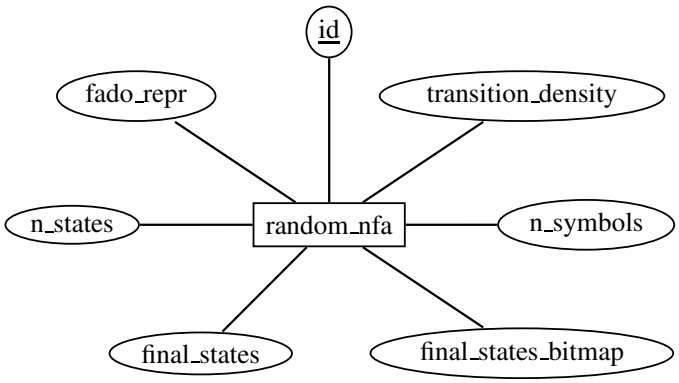


Figure 6.2: Entity-relationship diagram of the random NFAs' database.

In this database, we do not store any additional information besides the number of states, size of the alphabet, set of final states, and the internal representation of each NFA within the **FAdo** toolkit. We do favour performance, and maintain the trade-off between time and space by storing the set of final states in two different ways: the comma separated list of integers, and the bitmap.

Just like in the ICDFA case, we used table partitioning to split the large automata table into smaller physical pieces, using the number of states of the NFAs as the delimiter. We also explicitly created four additional indexes:

- one in the `n_states` column;
- one in the `n_symbols` column;
- one in the `transition_density` column;
- one compound index on the three previous columns.

6.3 Database of random regular expressions

The database of random regular expressions follows a structure rather similar to that of finite automata, containing samples generated according to the context-free grammar G_R [46, 65].

$$
\begin{aligned}
S &\to RS \mid CC \mid EE \mid I \mid \epsilon \mid \emptyset \\
CC &\to CCR \mid RR \\
R &\to (RS) \mid EE \mid I \\
EE &\to (RS)^\star \mid (CC)^\star \mid I^\star \\
RS &\to \epsilon + X \mid Y + Z \\
X &\to T \mid T + X \\
T &\to CC \mid I \\
Y &\to Z \mid Y + Z \\
Z &\to CC \mid EE \mid I
\end{aligned}
\qquad (G_R)
$$

It stores random samples of 20 000 regular expressions of length

$$n \in \{\, 10, 20, 30, 40, 50, 75, 100, 125, 150, 175, 200, 250, 300\}$$

each over an alphabet of

$$k \in \{\, 2, 3, 4, \ldots, 16, 18, 20, 25, 30, 35, 40, 45, 50\}$$

symbols. The entity-relationship model is presented on Figure 6.3.

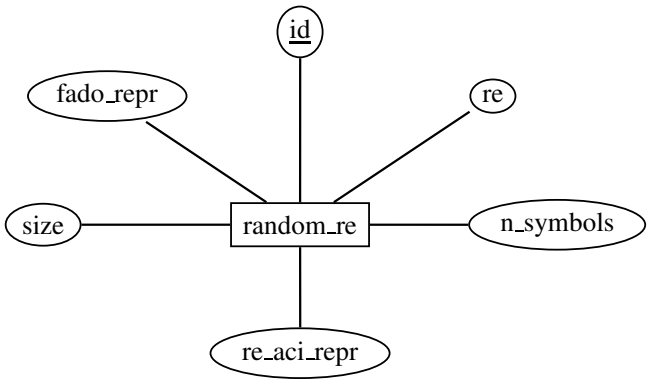


Figure 6.3: Entity-relationship diagram of the database of random regular expressions.

Besides the usual human-readable string, two pre-parsed representations of each regular expression are kept in the database: one in modulo ACI, and another one using the internal representation of the **FAdo** toolkit (`re_aci_repr` and `fado_repr`, respectively). Again, we do so to avoid the overhead of parsing a regular expression every time some algorithm needs to use it.

For efficiency reasons, the large table `random_re` is also split, by the size of the regular expressions, into smaller physical pieces. There are explicit indexes on columns `size` and `n_symbols`, as well as a compound index on both these columns.

6.4 Using the databases

The three databases are publicly available and may be freely used. The information required to access each of the databases may be found on Table 6.2.

Some usage examples

- Get 5 minimal ICDFAs with 10 states over an alphabet of 2 symbols:

```
marco@apollo:~$ psql --host khixote.ncc.up.pt --user guest --dbname icdfa_dataset
```

Database	PostgreSQL name	Host	User	Password
ICDFAs	icdfa_dataset	khixote.ncc.up.pt	guest	guest
NFAs	nfa_dataset	khixote.ncc.up.pt	guest	guest
RE	re_dataset	khixote.ncc.up.pt	guest	guest

Table 6.2: Connection details for the databases of random objects.

```
Password for user guest:
psql (8.4.3, server 8.4.2)
Type "help" for help.

icdfa_dataset=> SELECT string, final_states FROM random_icdfa WHERE
                n_states=10 AND
                n_symbols=2 AND
                is_minimal=true
                LIMIT 5;
                 string                  |  final_states
-----------------------------------------+-----------------
 1,2,3,4,3,5,0,6,2,3,4,1,7,8,9,9,9,4,9,0 | 0,1,7
 1,2,3,0,3,3,4,2,5,6,3,2,6,7,8,8,1,9,0,4 | 2,3,6
 1,2,3,0,4,0,2,5,1,1,6,7,4,8,6,9,9,1,0,9 | 4,6,9
 1,0,2,1,1,3,4,5,6,7,8,9,7,7,4,3,0,6,9,0 | 0,1,2,4,5,6,7,8
 1,0,2,3,0,4,5,0,4,6,3,7,3,8,7,1,1,9,2,7 | 2,3,6,9
(5 rows)
```

- Get the unique identifier of 5 random NFAs with 10 states, over an alphabet of 2 symbols, and transition density $d = 0.8$:

```
marco@apollo:~$ psql --host khixote.ncc.up.pt --user guest --dbname nfa_dataset
Password for user guest:
psql (8.4.3, server 8.4.2)
Type "help" for help.

nfa_dataset=> SELECT id FROM random_nfa WHERE
```

```
              n_states=10 AND
              n_symbols=2 AND
              transition_density='0.8'
              LIMIT 5;
   id
---------
 1239401
 1239399
 1239398
 1239395
 1239392
(5 rows)
```

- Get 5 random regular expressions of size 50, over an alphabet of 7 symbols:

```
marco@apollo:~$psql --host khixote.ncc.up.pt --user guest --dbname re_dataset
Password for user guest:
psql (8.4.3, server 8.4.2)
Type "help" for help.

re_dataset=> SELECT re FROM random_re WHERE
              size=50 AND
              n_symbols=7
              LIMIT 5;
                            re
------------------------------------------------------------
 cecfd*bcd+(a+e(ff(gc+bgfed*e+gbdc*c)c+f*)a*b)f+g+b
 d+(bbc(dcgccdaecdb+fgdagf)ffbb*aabecdbecafegga)*fd
 gaeb(ef*b+(@epsilon+ac*ad*bd)cf+ag)(bgcgee*dagg*+g)dafafc
 aecbfb*ffefed*+(db+g)fcfefb(f+adb*ecb*e)edbcbd*fbc
 bda*b*gcff*bdd+cdfc*d*bdc+eega*bca*edce+bbdeecfee*
(5 rows)
```

Chapter 7

Equivalence of regular expressions

Although, mainly for efficiency reasons, finite automata are normally used to manipulate regular languages, regular expressions provide a particularly good notation for its representation. Deciding whether two regular expressions are equivalent, however, is a PSPACE-complete problem [68]. If we consider regular expressions extended with intersection, the problem becomes even harder [1, page 419], requiring $c'c^{\sqrt{n/\log n}}$ space and time for a regular expression of size n and some constants $c' > 0$ and $c > 1$.

The usual approach to this decision problem starts by transforming each regular expression into an equivalent NFA, making each of the NFAs deterministic, and either minimising both DFAs and testing if the minimal automata are isomorphic, or applying one of the equivalence-testing algorithms described in Chapter 9. In the worst case, because the NFA-to-DFA conversion process may result in an exponential number of states (cf. Subsection 3.2.1), this process has an exponential worst-case running-time.

Ginzburg [27] has described a procedure that checks the equality of regular expressions using derivatives — as introduced by Brzozowski [12] — and transition graphs. For each regular expression α, a corresponding transition graph is used to generate a finite set of left-linear equations that characterise α. Two regular expressions, α and β are equivalent if and only if the pairwise constant terms in the set of left-linear equations are equal.

Also based on the algebraic properties of regular expressions and the notion of derivative Antimirov and Mosses [6] proposed a complete, terminating rewrite system for deciding their equivalence. This rewrite system is a refutation method that normalises regular expressions in a way such that testing their equivalence corresponds to an iterated process of testing the equivalence of their derivatives. In this chapter we present an improved functional approach to that method, prove its correctness, and present some experimental comparative results. Besides this variant of Antimirov and Mosses' algorithm, we also present an alternative based on partial derivatives.

Antimirov and Mosses suggested that their approach could obtain better average-case algorithm than those based on the comparison of minimal DFAs. Our results lead to the conclusion that indeed these algebraic methods are feasible and, quite often, more efficient than the usual approach.

Throughout this chapter there is some abuse of the Python-like pseudocode syntax described in Section 2.4, namely on the **if-then-else** chains used to indicate a pattern match. We believe, however, that this is not only more consistent with the style of the algorithms presented in other Chapters, but that it also reduces clutter and enhances readability by not being too "strict".

7.1 Classical method

The classical approach to the problem of testing the equivalence of two regular expressions α and β, i.e., deciding if $L(\alpha) = L(\beta)$, typically consists of:

1. obtain an equivalent pair of NFAs, N_α and N_β;

2. convert the NFAs to equivalent DFAs, $D_\alpha \sim N_\alpha$ and $D_\beta \sim N_\beta$;

3. $\begin{cases} \text{minimise}^{1} \text{ both DFAs, } D_\alpha \text{ and } D_\beta, \text{ and test if they are isomorphic,} \\ \quad or \\ \text{apply a direct equivalence-testing algorithm}^{2} \text{ to } D_\alpha \text{ and } D_\beta. \end{cases}$

The canonical string described in Subsection 3.1.1 provides a computationally efficient representation that may be used to simplify the isomorphism test.

7.2 Avoiding finite automata

In 1994, Antimirov and Mosses [6] presented a rewrite system for deciding the equivalence of two extended (with intersection) regular expressions based on a new Horn-equational axiomatisation of the extended algebra of regular sets. Based on the same algebraic calculus for proving/disproving equations of extended regular expressions, we propose a functional variant of that rewrite system. We argue that this functional approach allows us to simplify the original rewrite system (discarding most of the rules), is computationally efficient, and still avoids the construction of the minimal finite automata.

7.2.1 Antimirov and Mosses' rewrite system

The term-rewriting techniques proposed by Antimirov and Mosses provide an algebraic calculus for testing the equality of extended regular expressions while avoiding the construction of finite automata.

The axiomatisation of the algebra of the extended regular expressions is based on Salomaa's system [64], replacing the inference rule that depends on the negation of the empty word property by an equational implication. For finite alphabets, this axiomatisation is finite.

The rewrite system, LF, applies the set of extended axioms — modulo associativity of the

[1] Automata minimisation is thoroughly discussed on Chapter 8.

[2] Direct DFA equivalence-testing algorithms are described on Chapter 9.

concatenation and associativity and commutativity of the disjunction — reducing the regular expressions to a specific form. At the same time, it obtains the derivatives of each regular expression. This system includes an auxiliary function f, used to calculate the non-constant part of a regular expression.

The main component of the inference system is the set of transformation rules TR, which also involves the rewrite system LF for computing linear forms. The four rules are referred to as DIS, SIM, IND, and SPL. The rule SIM itself is a more or less arbitrary rewrite system which simplifies regular expressions. It needs only to be sufficiently strong in order to make the set of derivatives finite, i.e., assure disjunctions modulo ACI. The non-equivalence of two regular expressions is proved when DIS is reached. The rule IND removes an equation from the set of equalities to be proved, whilst SPL adds a new equation to the set of "inductive hypotheses".

7.2.2 Functional approach

Unlike Antimirov and Mosses, we do not consider extended regular expressions. We do, however, assume (without any loss of generality) that all regular expressions are irreducible and represented modulo ACI. This allows us to avoid the simplification step of Antimirov and Mosses' system (SIM) with little overhead.

We start by defining and proving the correctness of some auxiliary functions. These are used by RE-EQUIVALENT-P, the procedure that decides if two regular expressions recognise the same language.

Given two arbitrary regular expressions α and β, the first step of the equivalence-testing procedure is to compute two pre-linear regular expressions α' and β' such that $\alpha' \sim \alpha$ and $\beta' \sim \beta$. This is implemented by RE-PRE-LINEAR.

```
1 def RE-PRE-LINEAR(α):
2     if α = ∅:
3         return ∅
```

```
4       if α = ϵ:
5           return ∅
6       if α = a:
7           return a
8       if α = (β + γ):
9           return RE-PRE-LINEAR(β) + RE-PRE-LINEAR(γ)
10      if α = (β⋆):
11          return RE-PRE-LINEAR(β)β⋆
12      if α = (aβ):
13          return aβ
14      if α = (β⋆γ):
15          return RE-PRE-LINEAR(β)β⋆γ + RE-PRE-LINEAR(γ)
16      if α = ((β + γ)ψ):
17          return RE-PRE-LINEAR(βψ) + RE-PRE-LINEAR(γψ)
```

Since the equivalence-testing procedure needs deterministic linear regular expressions, the next step is to linearise the pre-linear regular expressions α' and β'. A call such as RE-LINEAR(α', β') returns two linear regular expressions, α'' and β'', such that $\alpha'' \sim \alpha'$ and $\beta'' \sim \beta'$.

```
1   def RE-LINEAR(α):
2       if α = (β + γ):
3           return RE-LINEAR(β) + RE-LINEAR(γ)
4       if α = ((β + γ)ψ):
5           return RE-LINEAR(βψ) + RE-LINEAR(γψ)
6       return α
```

Given a linear regular expression α'', the function RE-DETERMINISTIC computes and returns a new deterministic linear regular expression α_d such that $\alpha_d \sim \alpha''$.

```
1   def RE-DETERMINISTIC(α):
2       if α = (aβ + a):
3           return a(β + ϵ)
4       if α = (aβ + aγ):
5           return a(β + γ)
6       if α = (aβ + aγ + ψ):
```

```
7          return RE-DETERMINISTIC(a(β + γ) + ψ)
8      return α
```

These functions implement a variant of Antimirov and Mosses' LF rewrite system. The procedure RE-PRE-LINEAR corresponds to LF's system function f which, contrary to what is claimed by Antimirov and Mosses, returns a pre-linear regular expression, not a linear one.

Theorem 11. *The function* RE-PRE-LINEAR *is well defined.*

Proof. Let Σ be an alphabet, $a \in \Sigma$ and α, β, γ, and ψ be arbitrary regular expressions.

It is clear that for

$$\alpha = \begin{cases} \emptyset, \\ \epsilon, \\ a, \\ \beta + \gamma, \\ \beta^\star, \end{cases}$$

RE-PRE-LINEAR(α) is well defined. We need only to show that RE-PRE-LINEAR(α) is well defined when α is a concatenation. These are all the possible cases:

$$\alpha = \begin{cases} \emptyset\beta, \\ \beta\emptyset, \\ \epsilon\beta, \\ \beta\epsilon, \\ a\beta, \\ (\beta + \gamma)\psi, \\ \beta^\star\gamma. \end{cases}$$

Because we are only considering irreducible regular expressions modulo ACI,

$$\emptyset\beta \sim \beta\emptyset \sim \emptyset$$

and

$$\epsilon\beta \sim \beta\epsilon \sim \beta.$$

Thus, $\emptyset\beta$ and $\beta\emptyset$ are explicitly handled by RE-PRE-LINEAR, and concatenations with the empty word ($\epsilon\beta$ and $\beta\epsilon$) do not need to be considered. The three remaining cases are explicitly handled by RE-PRE-LINEAR. □

Theorem 12. *The function* RE-PRE-LINEAR *returns a pre-linear regular expression.*

Proof. Recall that a regular expression is pre-linear if it is generated by the context free grammar G_2 (cf. Chapter 4). We will show that for an arbitrary regular expression α,

$$\text{RE-PRE-LINEAR}(\alpha) \in L(G_2).$$

The proof follows by induction on the structure of α. Let $a \in \Sigma$ and β, γ, and ψ be regular expressions.

Base:

$$\text{RE-PRE-LINEAR}(\emptyset) = \emptyset,$$
$$\text{RE-PRE-LINEAR}(\epsilon) = \emptyset,$$
$$\text{RE-PRE-LINEAR}(a) = a,$$

Every one of these regular expressions is clearly generated by the grammar G_2.

Induction:

- $\text{RE-PRE-LINEAR}(\beta + \gamma) = \text{RE-PRE-LINEAR}(\beta) + \text{RE-PRE-LINEAR}(\gamma)$

  By induction hypothesis,

  $$\text{RE-PRE-LINEAR}(\beta) \in L(G_2)$$

and

$$\text{RE-PRE-LINEAR}(\gamma) \in L(G_2).$$

The disjunction of two pre-linear regular expressions is derived by the production

$$Q \to Q + Q$$

and therefore, $\text{RE-PRE-LINEAR}(\beta + \gamma) \in L(G_2)$;

- $\text{RE-PRE-LINEAR}(\beta^\star) = \text{RE-PRE-LINEAR}(\beta)\beta^\star$

  By induction hypothesis, $\text{RE-PRE-LINEAR}(\beta) \in L(G_2)$. Using the production

  $$Q \to Q \cdot R$$

  we can derive $\text{RE-PRE-LINEAR}(\beta)\beta^\star$, thus $\text{RE-PRE-LINEAR}(\beta^\star) \in L(G_2)$;

- $\text{RE-PRE-LINEAR}(a\beta) = a\beta$

  Clearly $a\beta$ is generated by the grammar G_2;

- $\text{RE-PRE-LINEAR}(\beta^\star\gamma) = \text{RE-PRE-LINEAR}(\beta)\beta^\star\gamma + \text{RE-PRE-LINEAR}(\gamma)$

  By induction hypothesis, both $\text{RE-PRE-LINEAR}(\beta)$ and $\text{RE-PRE-LINEAR}(\gamma)$ are generated by G_2. The concatenation $\text{RE-PRE-LINEAR}(\beta)\beta^\star\gamma$ is clearly pre-linear and may be generated by the rule

  $$Q \to Q \cdot R.$$

  Just like with the previous cases, the disjunction of two pre-linear regular expressions is also a pre-linear regular expression and therefore $\text{RE-PRE-LINEAR}(\beta^\star\gamma) \in L(G_2)$;

- $\text{RE-PRE-LINEAR}((\beta + \gamma)\psi) = \text{RE-PRE-LINEAR}(\beta\psi) + \text{RE-PRE-LINEAR}(\gamma\psi)$

  By induction hypothesis,

  $$\text{RE-PRE-LINEAR}(\beta\psi) \in L(G_2)$$

  and

  $$\text{RE-PRE-LINEAR}(\gamma\psi) \in L(G_2).$$

  Again, the disjunction of two pre-linear regular expressions is also a pre-linear regular expression. □

Theorem 13. *Given a pre-linear regular expression $\alpha \in L(G_2)$ or $\emptyset$,* RE-LINEAR(α) *returns either a linear regular expression or $\emptyset$.*

Proof. Recall that a pre-linear regular expression is either $\emptyset$, an already linear regular expression or a disjunction of concatenations where the first argument of each concatenation is a pre-linear regular expression (cf. Chapter 4).

Notice that RE-LINEAR recursively breaks disjunctions at line 2. Each element of a disjunction is either already linear, or of the form $\alpha = (\beta + \gamma)\psi$, where $\beta + \gamma$ is pre-linear. In this case, the distributive property is applied, α is expanded into $\beta\psi + \gamma\psi$, and RE-LINEAR is recursively called on both $\beta\psi$ and $\gamma\psi$ at line 5. The next recursive calls will repeat the procedure until the expressions are not disjunctions or of the form $\alpha = (\beta + \gamma)\psi$. Since we are expanding pre-linear regular expressions, this will result on a linear regular expression $\alpha' \in L(G_1)$.

Regular expressions of any other form ($\emptyset$, already linear regular expressions such as simple concatenations with symbols, etc.), are clearly generated by the grammar G_2 and returned without any modification at line 6. □

Lemma 14. *Let α be an arbitrary regular expression. We have that*

$$L(\text{RE-PRE-LINEAR}(\alpha)) = L(\alpha) - \{\epsilon\}.$$

Proof. The proof follows by induction on the structure of α. Let $a \in \Sigma$ and β, γ, and ψ be regular expressions.

Base:

$$\text{RE-PRE-LINEAR}(\emptyset) = \emptyset;$$
$$\text{RE-PRE-LINEAR}(\epsilon) = \emptyset;$$
$$\text{RE-PRE-LINEAR}(a) = a.$$

Induction:

- $\alpha = \text{RE-PRE-LINEAR}(\beta + \gamma)$

$$\begin{aligned} L(\alpha) &= L(\text{RE-PRE-LINEAR}(\beta) + \text{RE-PRE-LINEAR}(\gamma)) \\ &= L(\text{RE-PRE-LINEAR}(\beta)) \cup L(\text{RE-PRE-LINEAR}(\gamma)) \\ &= (L(\beta) - \{\epsilon\}) \cup (L(\gamma) - \{\epsilon\}) \\ &= (L(\beta) \cup L(\gamma)) - \{\epsilon\} \\ &= L(\beta + \gamma) - \{\epsilon\} \end{aligned}$$

- $\alpha = \text{RE-PRE-LINEAR}(\beta^\star)$

$$\begin{aligned} L(\alpha) &= L(\text{RE-PRE-LINEAR}(\beta)\beta^\star) \\ &= L(\text{RE-PRE-LINEAR}(\beta))L(\beta^\star) \\ &= (L(\beta) - \{\epsilon\})L(\beta^\star) \\ &= (L(\beta) \cap \overline{\{\epsilon\}})L(\beta^\star) \\ &= L(\beta\beta^\star) \cap \overline{\{\epsilon\}}L(\beta^\star) \\ &= L(\beta\beta^\star) \cap \overline{\{\epsilon\}} \\ &= (L(\beta\beta^\star) \cap \overline{\{\epsilon\}}) \cup (\{\epsilon\} \cap \overline{\{\epsilon\}}) \\ &= (L(\beta\beta^\star) \cup \{\epsilon\}) \cap \overline{\{\epsilon\}} \\ &= L(\beta\beta^\star) - \{\epsilon\} \\ &= L(\beta^\star) - \{\epsilon\}; \end{aligned}$$

- $\alpha = \text{RE-PRE-LINEAR}(a\beta)$

$$\begin{aligned} L(\alpha) &= L(a\beta) \\ &= L(a\beta) - \{\epsilon\}; \end{aligned}$$

- $\alpha = \text{RE-PRE-LINEAR}(\beta^\star\gamma)$

$$\begin{aligned}
L(\alpha) &= L(\text{RE-PRE-LINEAR}(\beta)\beta^\star\gamma + \text{RE-PRE-LINEAR}(\gamma)) \\
&= L(\text{RE-PRE-LINEAR}(\beta)\beta^\star\gamma) \cup L(\text{RE-PRE-LINEAR}(\gamma)) \\
&= L(\text{RE-PRE-LINEAR}(\beta))L(\beta^\star\gamma) \cup L(\text{RE-PRE-LINEAR}(\gamma)) \\
&= (L(\beta) - \{\epsilon\})L(\beta^\star\gamma) \cup (L(\gamma) - \{\epsilon\}) \\
&= (L(\beta) \cap \overline{\{\epsilon\}})L(\beta^\star\gamma) \cup (L(\gamma) \cap \overline{\{\epsilon\}}) \\
&= (L(\beta)L(\beta^\star\gamma) \cap \overline{\{\epsilon\}}L(\beta^\star\gamma)) \cup (L(\gamma) \cap \overline{\{\epsilon\}}) \\
&= (L(\beta)L(\beta^\star\gamma) \cap \overline{\{\epsilon\}}) \cup (L(\gamma) \cap \overline{\{\epsilon\}}) \\
&= (L(\beta)L(\beta^\star)L(\gamma) \cap \overline{\{\epsilon\}}) \cup (L(\gamma) \cap \overline{\{\epsilon\}}) \\
&= (L(\beta)L(\beta^\star)L(\gamma) \cup L(\gamma)) \cap \overline{\{\epsilon\}} \\
&= (L(\beta)L(\beta^\star) \cup \{\epsilon\})L(\gamma) \cap \overline{\{\epsilon\}} \\
&= L(\beta\beta^\star + \epsilon)L(\gamma) \cap \overline{\{\epsilon\}} \\
&= L(\beta^\star)L(\gamma) \cap \overline{\{\epsilon\}} \\
&= L(\beta^\star\gamma) \cap \overline{\{\epsilon\}} \\
&= L(\beta^\star\gamma) - \{\epsilon\}.
\end{aligned}$$

- $\alpha = \text{RE-PRE-LINEAR}((\beta + \gamma)\psi)$

$$\begin{aligned}
L(\alpha) &= L(\text{RE-PRE-LINEAR}(\beta\psi) + \text{RE-PRE-LINEAR}(\gamma\psi)) \\
&= L(\text{RE-PRE-LINEAR}(\beta\psi)) \cup L(\text{RE-PRE-LINEAR}(\gamma\psi)) \\
&= (L(\beta\psi) - \{\epsilon\}) \cup (L(\gamma\psi) - \{\epsilon\}) \\
&= (L(\beta\psi) \cup L(\gamma\psi)) - \{\epsilon\} \\
&= L(\beta\psi + \gamma\psi) - \{\epsilon\} \\
&= L((\beta + \gamma)\psi) - \{\epsilon\};
\end{aligned}$$

□

Lemma 15. *Given an arbitrary regular expression α we have that*

$$\textsc{Re-Linear}(\alpha) \sim \alpha.$$

Proof. The proof follows by induction on the structure of α. Let $a \in \Sigma$ and β, γ, and ψ be regular expressions.

Base:

$$\textsc{Re-Linear}(\emptyset) = \emptyset \sim \emptyset;$$

$$\textsc{Re-Linear}(\epsilon) = \epsilon \sim \epsilon;$$

$$\textsc{Re-Linear}(a) = a \sim a;$$

Induction:

- $\textsc{Re-Linear}(\beta + \gamma) = \textsc{Re-Linear}(\beta) + \textsc{Re-Linear}(\gamma)$

  By induction hypothesis,

  $$\textsc{Re-Linear}(\beta) \sim \beta$$

  and

  $$\textsc{Re-Linear}(\gamma) \sim \gamma,$$

  hence, $\textsc{Re-Linear}(\beta + \gamma) \sim \beta + \gamma$.

- $\textsc{Re-Linear}(\beta^\star) = \beta^\star \sim \beta^\star$.

- $\textsc{Re-Linear}(a\beta) = a\beta \sim a\beta$.

- $\textsc{Re-Linear}(\beta^\star\gamma) = \beta^\star\gamma \sim \beta^\star\gamma$.

- $\textsc{Re-Linear}((\beta + \gamma)\psi) = \textsc{Re-Linear}(\beta\psi) + \textsc{Re-Linear}(\gamma\psi)$

  By induction hypothesis,

  $$\textsc{Re-Linear}(\beta\psi) \sim \beta\psi$$

  and

  $$\textsc{Re-Linear}(\gamma\psi) \sim \gamma\psi,$$

therefore, RE-LINEAR$((\beta + \gamma)\psi) \sim \beta\psi + \gamma\psi$.

By Axiom A_9,

$$\beta\psi + \gamma\psi \sim (\beta + \gamma)\psi,$$

thus

$$\text{RE-LINEAR}((\beta + \gamma)\psi) \sim (\beta + \gamma)\psi.$$

□

Theorem 16. *Let α be an arbitrary regular expression. We have that*

$$L(\text{RE-LINEAR}(\text{RE-PRE-LINEAR}(\alpha))) = L(\alpha) - \{\epsilon\}$$

That is to say, RE-LINEAR(RE-PRE-LINEAR(α)) *returns a regular expression that recognises the same language as α, except for the empty word.*

Proof. It follows directly from the definition of RE-LINEAR and Lemmas 14 and 15. □

Theorem 17. *Given a linear regular expression as argument, the function* RE-DETERMINISTIC *returns a deterministic linear regular expression.*

Proof. Let α be a linear regular expression and $\alpha_d = \text{RE-DETERMINISTIC}(\alpha)$. We have to show that HEAD(α_d) does not have repeated elements. Without loss of generality, let us consider only one alphabet symbol $a \in \Sigma$ — we can repeat the process for each remaining symbol.

When there is only one sub-expression of the form $a\alpha'$ in α, we have that $\alpha_d = \alpha$ and, considering only the symbol a, HEAD(α_d) is a singleton. With two sub-expressions of the same form, we have that either $\alpha_d = a(\beta + \gamma)$ or $\alpha_d = a(\beta + \epsilon)$ and again, HEAD(α_d) contains no repeated elements.

Suppose now that there are n sub-expressions with the prefix a. By definition, RE-DETERMINISTIC will apply the distributive property, reduce the number of sub-expressions of the form $a\alpha'$ to $n - 1$, and recursively call itself. After $n - 1$ applications, the resulting

regular expression α_d has only one element of the form $a\alpha'$, and $\text{HEAD}(\alpha_d)$ is a multiset with multiplicity 1, therefore it contains no repeated elements. □

Theorem 18. *Given a linear regular expression α, we have that*

$$\text{RE-DETERMINISTIC}(\alpha) \sim \alpha.$$

Proof. Except for the following two cases, RE-DETERMINISTIC returns its argument, so clearly

$$\text{RE-DETERMINISTIC}(\alpha) \sim \alpha.$$

- $\alpha = a\beta + a$ or $\alpha = a\beta + a\gamma$

  if the argument is of the form $\alpha = a\beta + a$ or $\alpha = a\beta + a\gamma$ the distributive property is applied and $a(\alpha + \epsilon)$ or $a(\beta + \gamma)$ are returned (respectively). Trivially

  $$\text{RE-DETERMINISTIC}(\alpha) \sim \alpha$$

  in both cases.

- $\alpha = a\beta + a\gamma + \psi$

  if the argument has the form $\alpha = a\beta + a\gamma + \psi$, again, the distributive property is applied, and the next recursive call is invoked with the trivially equivalent regular expression $a(\beta + \gamma) + \psi$. Notice that at some point further ahead, RE-DETERMINISTIC will return its argument (or an equivalent variant) and thus,

  $$\text{RE-DETERMINISTIC}(\alpha) \sim \alpha.$$

□

Theorem 19. *For any regular expression α,*

$$\alpha \sim \hat{\epsilon}(\alpha) + \text{RE-LINEAR}(\text{RE-PRE-LINEAR}(\alpha)) \tag{7.1}$$

and

$$\alpha \sim \hat{\epsilon}(\alpha) + \text{RE-DETERMINISTIC}(\text{RE-LINEAR}(\text{RE-PRE-LINEAR}(\alpha))). \tag{7.2}$$

Proof. By Theorem 18, $\alpha \sim \text{RE-DETERMINISTIC}(\alpha)$ for every regular expression α, therefore, we need to prove only Equation 7.1. There are two cases to consider.

- If $\epsilon \in L(\alpha)$,

$$\begin{aligned} L(\hat{\epsilon}(\alpha) + \text{RE-LINEAR}(\alpha)) &= L(\hat{\epsilon}(\alpha)) \cup L(\text{RE-LINEAR}(\alpha)) && \text{by definition,} \\ &= L(\epsilon) \cup L(\text{RE-LINEAR}(\alpha)) && \epsilon \in L(\alpha), \\ &= \{\epsilon\} \cup (L(\alpha) - \{\epsilon\}) && \text{by Theorem 16,} \\ &= L(\alpha) && \epsilon \in L(\alpha). \end{aligned}$$

- If $\epsilon \notin L(\alpha)$, then $\hat{\epsilon}(\alpha) = \emptyset$ and, by Theorem 16, the equivalence is obvious.

□

Lemma 20. *Let α be an arbitrary regular expression and $a \in \Sigma$. We have that*

$$a^{-1}(\alpha) = a^{-1}(\text{RE-PRE-LINEAR}(\alpha)).$$

Proof. The proof follows by induction on the structure of α. Let β, γ, and ψ be regular expressions.

Base:

$$a^{-1}(\emptyset) = \emptyset = a^{-1}(\emptyset) = a^{-1}(\text{RE-PRE-LINEAR}(\emptyset))$$

$$a^{-1}(\epsilon) = \emptyset = a^{-1}(\emptyset) = a^{-1}(\text{RE-PRE-LINEAR}(\epsilon))$$

$$a^{-1}(b) = \left\{ \begin{array}{ll} \epsilon & \text{if } a = b \\ \emptyset & \text{if } a \neq b \end{array} \right\} = a^{-1}(b) = a^{-1}(\text{RE-PRE-LINEAR}(b))$$

Induction: Suppose now, by induction hypothesis, that for every regular expression α,

$$a^{-1}(\alpha) = a^{-1}(\text{RE-PRE-LINEAR}(\alpha)).$$

We have the following three cases.

- Disjunction:

$$\begin{aligned} a^{-1}(\beta+\gamma) &= a^{-1}(\beta) + a^{-1}(\gamma) \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta)) + a^{-1}(\textsc{Re-Pre-Linear}(\gamma)) \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta) + \textsc{Re-Pre-Linear}(\gamma)) \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta+\gamma)). \end{aligned}$$

- Star closure:

$$\begin{aligned} a^{-1}(\beta^\star) &= a^{-1}(\beta)\beta^\star \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta))\beta^\star \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta))\beta^\star + \emptyset\beta^\star \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta))\beta^\star + \hat{\epsilon}(\textsc{Re-Pre-Linear}(\beta))\beta^\star \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta)\beta^\star) \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta^\star)). \end{aligned}$$

- As for the concatenation, there are three cases to consider:

$$\begin{aligned} a^{-1}(a\beta) &= a^{-1}(\textsc{Re-Pre-Linear}(a\beta)); \\ a^{-1}((\beta+\gamma)\psi) &= a^{-1}(\beta\psi + \gamma\psi) \\ &= a^{-1}(\beta\psi) + a^{-1}(\gamma\psi) \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta\psi)) + a^{-1}(\textsc{Re-Pre-Linear}(\gamma\psi)) \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta\psi) + \textsc{Re-Pre-Linear}(\gamma\psi)) \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta\psi + \gamma\psi)) \\ &= a^{-1}(\textsc{Re-Pre-Linear}(\beta+\gamma)\psi); \\ a^{-1}(\beta^\star\gamma) &= a^{-1}(\beta^\star)\gamma + \hat{\epsilon}(\beta^\star)a^{-1}(\gamma) \\ &= a^{-1}(\beta)\beta^\star\gamma + \epsilon a^{-1}(\gamma) \\ &= a^{-1}(\beta)\beta^\star\gamma + a^{-1}(\gamma) \end{aligned}$$

$$
\begin{aligned}
&= a^{-1}(\text{RE-PRE-LINEAR}(\beta))\beta^\star\gamma + a^{-1}(\text{RE-PRE-LINEAR}(\gamma)) \\
&= a^{-1}(\text{RE-PRE-LINEAR}(\beta))\beta^\star\gamma + \emptyset\beta^\star a^{-1}(\gamma) + \\
&\quad + a^{-1}(\text{RE-PRE-LINEAR}(\gamma)) \\
&= a^{-1}(\text{RE-PRE-LINEAR}(\beta))\beta^\star\gamma + \\
&\quad + \hat{\epsilon}(\text{RE-PRE-LINEAR}(\beta))\beta^\star a^{-1}(\gamma) + \\
&\quad + a^{-1}(\text{RE-PRE-LINEAR}(\gamma)) \\
&= a^{-1}(\text{RE-PRE-LINEAR}(\beta)\beta^\star\gamma) + a^{-1}(\text{RE-PRE-LINEAR}(\gamma)) \\
&= a^{-1}(\text{RE-PRE-LINEAR}(\beta)\beta^\star\gamma + \text{RE-PRE-LINEAR}(\gamma)) \\
&= a^{-1}(\text{RE-PRE-LINEAR}(\beta^\star\gamma)).
\end{aligned}
$$

□

Lemma 21. *Let α and β be two arbitrary regular expressions and $a \in \Sigma$. We have that*

$$\alpha \sim \beta \Rightarrow a^{-1}(\alpha) \sim a^{-1}(\beta).$$

Proof. Suppose that $\alpha \sim \beta$, then $L(\alpha) = L(\beta)$. We need to show that $L(a^{-1}(\alpha)) = L(a^{-1}(\beta))$ for any $a \in \Sigma$.

$$
\begin{aligned}
L(a^{-1}(\alpha)) &= \{\, w \mid aw \in L(\alpha) \,\} && \text{by definition,} \\
&= \{\, w \mid aw \in L(\beta) \,\} && \text{because } L(\alpha) = L(\beta), \\
&= L(a^{-1}(\beta)) && \text{by definition.}
\end{aligned}
$$

□

Theorem 22. *Let $a \in \Sigma$ and α be an arbitrary regular expression. The following holds:*

$$a^{-1}(\alpha) \sim a^{-1}(\alpha_d),$$

where $\alpha_d = \text{RE-DETERMINISTIC}(\text{RE-LINEAR}(\text{RE-PRE-LINEAR}(\alpha)))$.

Proof.

$$
\begin{aligned}
a^{-1}(\alpha) &= a^{-1}(\text{RE-PRE-LINEAR}(\alpha)) && \text{by Lemma 20,}\\
&\sim a^{-1}(\text{RE-LINEAR}(\text{RE-PRE-LINEAR}(\alpha))) && \text{by Lemmas 15 and 21,}\\
&\sim a^{-1}(\alpha_d) && \text{by Theorem 18.}
\end{aligned}
$$

□

The two main functions of the equivalence-testing process are RE-DERIVATIVES and RE-EQUIVALENT-P. The first one, RE-DERIVATIVES, computes the set of the derivatives of a pair of deterministic linear regular expressions, (α, β), with respect to every symbol of the alphabet. It is enough to consider only the symbols in $\text{HEAD}(\alpha) \cup \text{HEAD}(\beta)$, and we do so for efficiency reasons.

```
1  def RE-DERIVATIVES(α, β):
2      return { (a^{-1}(α), a^{-1}(β)) | a ∈ (HEAD(α) ∪ HEAD(β)) }
```

When applied to two arbitrary regular expressions α and β, RE-EQUIVALENT-P returns TRUE or FALSE according to whether $\alpha \sim \beta$ or not.

```
1  def RE-EQUIVALENT-P(α, β):
2      S := {(α, β)}
3      H := ∅
4      while S ≠ ∅:
5          (α, β) := POP(S)
6          if ε̂(α) ≠ ε̂(β):
11             return FALSE
12         H := PUSH(H, (α, β))
13         α' := RE-DETERMINISTIC(RE-LINEAR(RE-PRE-LINEAR(α)))
14         β' := RE-DETERMINISTIC(RE-LINEAR(RE-PRE-LINEAR(β)))
15         S' := { d | d ∈ RE-DERIVATIVES(α', β'), d ∉ H }
16         S := S ∪ S'
17     return TRUE
```

At each step of the main loop (lines 4–16), RE-EQUIVALENT-P proceeds by replacing a pair of regular expressions by a set S' of pairs of derivatives. When either a pair of regular expressions such that their constant parts are different or the set S is empty, RE-EQUIVALENT-P returns. If $\alpha \sim \beta$ the call RE-EQUIVALENT-P(α, β) returns TRUE, otherwise it returns FALSE. Comparing with Antimirov and Mosses' rewrite system TR, we note that in each call to RE-EQUIVALENT-P, the set S contains only pairs of regular expressions which are not already in H; this renders the rule IND of TR unnecessary. Additionally, our data structures avoid the usage of the SIM rule by assuring that the regular expressions are always irreducible and represented modulo ACI.

Theorem 23. *The function* RE-EQUIVALENT-P *is terminating.*

Proof. It is clear that the function terminates when the set S is empty. The only new elements added to this set (at line 16) are the derivatives (computed at line 15) of the current pair of regular expressions — which was previously removed from S at line 5.

Because all regular expressions are considered modulo ACI, there is only a finite number of derivatives [12], and, from a given point on, $S \cup S' = S$. Since that at each iteration of the main loop one element is removed from S (line 5), after a finite number of iterations, $S = \emptyset$ and RE-EQUIVALENT-P terminates.

In order to assure that the same pair of regular expressions is not considered more than once, and thus prevent a possible infinite loop, the history of all pairs of regular expressions already processed is kept on the set H. The condition on line 15 assures that no repeated regular expressions are ever added to the set S. □

Lemma 24. *Let α and β be two deterministic linear regular expressions. We have that*

$$\forall (\alpha', \beta') \in \text{RE-DERIVATIVES}(\alpha, \beta) \quad \alpha \sim \beta \Rightarrow \alpha' \sim \beta'.$$

Proof. It is a direct consequence of Lemma 21 and the definition of RE-DERIVATIVES. □

Lemma 25. *Given two regular expressions α and β such that $\alpha \sim \beta$,*

$$\text{RE-EQUIVALENT-P}(\alpha, \beta) = \text{TRUE}.$$

Proof. If α and β are deterministic linear regular expressions and $\alpha \sim \beta$ we know, by Lemma 24, that

$$\forall(\alpha', \beta') \in \text{RE-DERIVATIVES}(\alpha, \beta) \quad \hat{\epsilon}(\alpha') = \hat{\epsilon}(\beta').$$

Thus being, the condition on line 6 will always be false, and no iteration of the main loop of RE-EQUIVALENT-P will return FALSE. □

Lemma 26. *Given two regular expressions, α and β, such that $\alpha \nsim \beta$,*

$$\text{RE-EQUIVALENT-P}(\alpha, \beta) = \text{FALSE}.$$

Proof. Let $w \in \Sigma^\star$. If $\alpha \nsim \beta$, either

$$w \in L(\alpha) \wedge w \notin L(\beta)$$

or

$$w \notin L(\alpha) \wedge w \in L(\beta).$$

Without loss of generality, let us consider only the first case. At some point of the main loop, the pair $(w^{-1}(\alpha), w^{-1}(\beta))$ will be popped (line 5) out of the set S. It is a well-known result by Brzozowski [12] that if $w \in L(\alpha)$, then $\hat{\epsilon}(w^{-1}(\alpha)) = \epsilon$. Since, by hypothesis, $w \in L(\alpha)$ and $w \notin L(\beta)$, we have that $\hat{\epsilon}(w^{-1}(\alpha)) = \epsilon$ and $\hat{\epsilon}(w^{-1}(\beta)) = \emptyset$. Therefore, the condition on line 6 of RE-EQUIVALENT-P will be true and RE-EQUIVALENT-P will return FALSE at line 7. □

Theorem 27. *Let α and β be arbitrary regular expressions. The call*

$$\text{RE-EQUIVALENT-P}(\alpha, \beta)$$

returns TRUE *if and only if $\alpha \sim \beta$.*

Proof. By direct application of Lemmas 25 and 26. □

7.3 An alternative using partial derivatives

Given an arbitrary regular expression, the set of its partial derivatives can be directly obtained from an alternative linearization process. Since we already represent disjunctions as sets (cf. Section 4.8), this alternative linearization can be easily implemented as a variant of RE-LINEAR.

A linear regular expression $\alpha = a_1\alpha_1 + \cdots + a_n\alpha_n$ can be represented by a finite set of monomials $S_\alpha = \{(a_1, \alpha_1), \ldots, (a_n, \alpha_n)\}$, called a *linear set*. The concatenation of such a set with an arbitrary regular expression β is defined in a natural way:

$$S_\alpha \cdot \beta = \{(a_1, \alpha_1\beta), \ldots, (a_n, \alpha_n\beta)\}.$$

Let α be a regular expression. The call RE-LINEAR-SET(α) returns the corresponding linear set.

```
1  def RE-LINEAR-SET(α):
2      if α = ∅:
3          return {}
4      if α = ϵ:
5          return {}
6      if α = a:
7          return {(a, ϵ)}
8      if α = (β + γ):
9          return RE-LINEAR-SET(β) ∪ RE-LINEAR-SET(γ)
10     if α = (β⋆):
11         return RE-LINEAR-SET(β)β⋆
12     if α = {(a, β)}:
13         return aβ
14     if α = (β⋆γ):
15         return RE-LINEAR-SET(β)β⋆γ ∪ RE-LINEAR-SET(γ)
16     if α = ((β + γ)ψ):
17         return RE-LINEAR-SET(βψ) ∪ RE-LINEAR-SET(γψ)
```

The two linearization methods, as implements by RE-LINEAR and RE-LINEAR-SET, are

related in the following way. Let

$$\alpha_\ell = \text{RE-LINEAR}(\alpha) = a_1\alpha_1 + \cdots + a_n\alpha_n$$

and

$$S_\alpha = \text{RE-LINEAR-SET}(\alpha) = \{ (a_1, \alpha_1), \ldots, (a_n, \alpha_n) \}.$$

It is clear that

$$\alpha_\ell \sim \sum_{(a_i,\alpha_i)\in S_\alpha} a_i\alpha_i = a_1\alpha_1 + \cdots + a_n\alpha_n.$$

Notice that due to the associativity of the disjunction and the unordered nature of sets, it is only possible to assure the equivalence of the languages, not syntactic equality of the regular expressions.

Let us now consider a process to make linear sets deterministic. We say that a linear set S_α is deterministic if, for each symbol $a \in \Sigma$, there is at most one element of the form $(a, \alpha) \in S_\alpha$. The procedure RE-LINEAR-SET-DETERMINISTIC takes a linear set as argument and makes it deterministic by merging all elements (a_i, α_i) and (a_j, α_j), such that $a_i = a_j$.

```
1  def RE-LINEAR-SET-DETERMINISTIC(α):
2      S_α := RE-LINEAR-SET(α)
3      S'_α := ∅
4      for a ∈ Σ:
5          S'_α := S'_α ∪ (a, Σ_{(a,α_i)∈S_α} α_i)
6      return S'_α
```

Given an arbitrary regular expression α, we can use this linearization process to compute the set of its partial derivatives with regard to to all symbols $a \in \Sigma$. The sets of partial derivatives of two regular expressions can be used to test their equivalence in a way similar to the one described on Section 7.2. In fact, only two slight modifications to RE-EQUIVALENT-P— which are implemented by RE-EQUIVALENT-PARTIAL-P— are necessary.

```
1  def RE-PARTIAL-DERIVATIVES(S_α, S_β):
2      return {(α', β') | (a, α') ∈ S_α, (a, β') ∈ S_β, a ∈ Σ}
```

The function RE-PARTIAL-DERIVATIVES computes the sets of partial derivatives of a pair of regular expressions (α, β) — with respect to every symbol of the alphabet — using the respective deterministic linear sets, S_α and S_β.

```
1   def RE-EQUIVALENT-PARTIAL-P(α, β):
2       S := {(α, β)}
3       H := ∅
4       while S ≠ ∅:
5           (α, β) := POP(S)
6           if ε̂(α) ≠ ε̂(β):
11              return FALSE
12          H := PUSH(H, (α, β))
13          S_α := RE-LINEAR-SET-DETERMINISTIC(α)
14          S_β := RE-LINEAR-SET-DETERMINISTIC(β)
15          S' := { d | d ∈ RE-PARTIAL-DERIVATIVES(S_α, S_β), d ∉ H }
16          S := S ∪ S'
17      return TRUE
```

7.4 Efficient implementation with disjoint-sets

Instead of a simple stack keeping a list of pairs of regular expressions which have already been tested for equivalence — such as the one implemented by variable H in RE-EQUIVALENT-P and RE-EQUIVALENT-PARTIAL-P— we can use a collection of disjoint sets to avoid infinite loops. Although this modification does not lower the overall worst-case running-time of the algorithm[1], it does reduce the complexity of a frequent internal operation — membership testing — and thus allows for a more efficient computational implementation.

At line 15 of both RE-EQUIVALENT-P and RE-EQUIVALENT-PARTIAL-P each derivative is tested for membership on H. By using a collection of disjoint sets where the regular

[1] Subsection 9.5.3 contains a proof that the exponential upper bound is tight.

expressions already tested for equivalence are merged on the same set, we can use UNION-FIND and reduce the search time on a set with n elements from $O(\log n)$ to $O(1)$.

```
1  def RE-EQUIVALENT-UNION-FIND-P(α, β):
2      S := {(α, β)}
3      UNION(h(α), h(β))
4      while S ≠ ∅:
5          (α, β) := POP(S)
6          if ε̂(α) ≠ ε̂(β):
11             return FALSE
12         S_α := RE-LINEAR-SET-DETERMINISTIC(α)
13         S_β := RE-LINEAR-SET-DETERMINISTIC(β)
14         for a ∈ (HEAD(α) ∪ HEAD(β)):
15             α' := ({d | (a, d) ∈ S_α}, 1)
16             β' := ({d | (a, d) ∈ S_β}, 2)
17             if FIND(h(α')) ≠ FIND(h(β')):
18                 UNION(h(α'), h(β'))
19                 S := S ∪ {α', β'}
20     return TRUE
```

When implementing this variation, however, some caution must be taken. Let α and β be the regular expressions tested for equivalence.

- We must ensure that $w^{-1}(\alpha) \neq u^{-1}(\beta)$ for all $w, u \in \Sigma^\star$. We do so by keeping tuples instead of simple derivatives where the second element is used to identify the regular expression from which the first element was obtained from. Thus, the derivative $\alpha' = w^{-1}(\alpha)$ is represented by $(\alpha', 1)$, whilst a derivative $\beta' = u^{-1}(\beta)$ is represented by $(\beta', 2)$.

- The FIND operation needs an equality test on the elements of the set. Testing the equality of two regular expressions — even syntactic equality — is a computationally expensive operation, and tuple comparison is even more costly. Since we know that each element of the set is unique, we may consider some hash function which assures that the probability of collision for these elements is extremely low. This allows us to

safely use the hash values (integers) as the elements of the set, and thus, arguments to the FIND operation, instead of the regular expressions themselves.

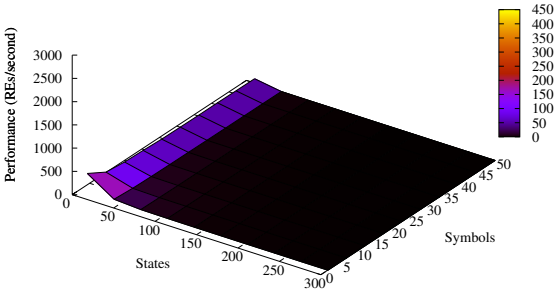
The predicate RE-EQUIVALENT-UNION-FIND-P tests the equivalence of two regular expressions α and β using these optimisations. The hash function, although not explicitly defined, is named h. Our Python implementation relies on the default hashing used for sets and dictionaries.

7.5 Experimental results

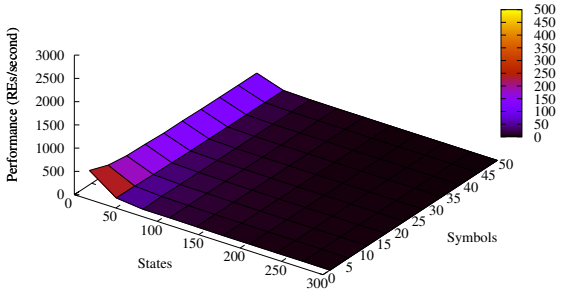
This section contains some experimental comparative results of the previously described algorithms for testing the equivalence of regular expressions: RE-EQUIVALENT-P, RE-EQUIVALENT-PARTIAL-P, and RE-EQUIVALENT-UNION-FIND-P. Since the main goal of these benchmarks is to compare the performance of the direct equivalence-testing methods with the more usual approach, based on DFA minimisation, they also include some results using Hopcroft's and Brzozowski's algorithms — recall that while Hopcroft's algorithm has the lowest known worst-case running time complexity, Brzozowski's presented a better practical performance when minimising NFAs, cf. Section 8.4. In order to obtain the NFAs from the regular expressions we used Glushkov's algorithm, as described by Yu [76, pages 72–74].

The conditions on which the experimental tests were conducted are described in detail on Chapter 5.

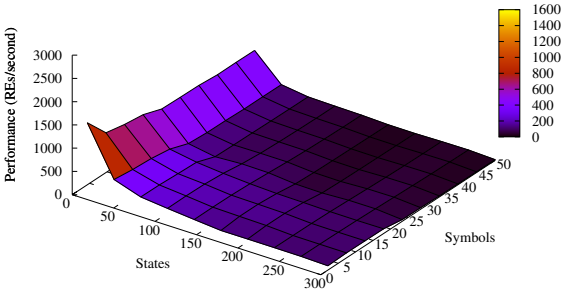
Here we include only a set of three-dimensional graphs. These are intended to be a bird's-eye view on the overall performance of the algorithms previously discussed on this Chapter. Recall that we define *performance* as the number of finite automata tested for equivalence per second. Appendix B includes complete tables with the exact values of the running time, memory usage, average number of recursive calls, etc. for each algorithm. The rule of thumb while reading the graphs is as follows: a darker area means lower values, and therefore, poorer performance.



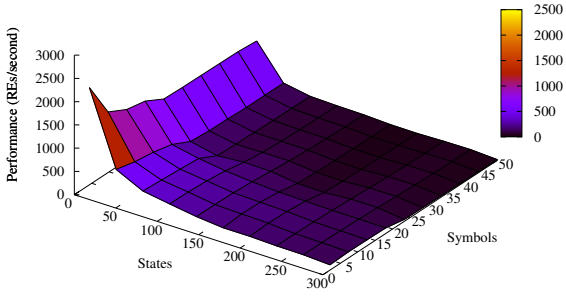
(a) DFA-MINIMISE-HOPCROFT



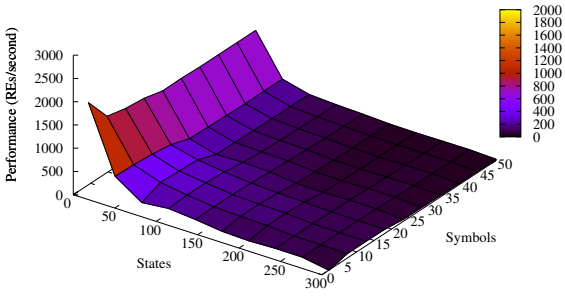
(b) DFA-MINIMISE-BRZOZOWSKI



(c) RE-EQUIVALENT-P



(d) RE-EQUIVALENT-PARTIAL-P



(e) RE-EQUIVALENT-UNION-FIND-P

Figure 7.1: Benchmarks of regular expressions equivalence-testing algorithms.

Clearly, any of the direct comparison methods — RE-EQUIVALENT-P, RE-EQUIVALENT-PARTIAL-P, or RE-EQUIVALENT-UNION-FIND-P— outperforms the classical approach,

based on DFA minimisation. As conjectured, Brzozowski's algorithm performs faster than Hopcroft's on several cases, which, without exception, presents the worst results.

Considering the direct comparison methods, which are frequently ten times faster than Hopcroft's algorithm, RE-EQUIVALENT-P is always the slowest of the implementations. Although RE-EQUIVALENT-PARTIAL-P and RE-EQUIVALENT-UNION-FIND-P present very similar results, RE-EQUIVALENT-PARTIAL-P is slightly faster with small alphabets. When the number of symbols increases, however, RE-EQUIVALENT-UNION-FIND-P becomes faster — actually, the fastest of the algorithms. The number of symbols required for RE-EQUIVALENT-UNION-FIND-P to outperform RE-EQUIVALENT-PARTIAL-P depends on the length of the regular expressions: while for regular expressions of length 5 RE-EQUIVALENT-UNION-FIND-P is faster than RE-EQUIVALENT-PARTIAL-P when the alphabet contains 10 or more symbols, considering regular expressions of length 150, the same performance results are only seen for regular expressions with 20 or more alphabetic symbols.

Chapter 8

Finite automata minimisation

The problem of finding the minimal DFA equivalent to a given automaton can be traced back to the 1950's with the works of Huffman [37, 38] and Moore [52]. Having applications on compiler construction, pattern matching, hardware circuit minimisation, and XML processing to name a few, over the years several alternative (and increasingly efficient) algorithms have been proposed. Authors typically present the worst-case time complexity analysis of their algorithms, but that does not provide enough information on the the practical behaviour. Moreover, little is known about the average-case time complexity. Two exceptions may be found in the works of Nicaud [56], where it is proved that the average-case complexity of Brzozowski's algorithm is exponential for group automata, and Bassino et al. [8], where it is shown that Moore's state minimisation algorithm is log-linear on average.

Using the Python programming language, we implemented the automata minimisation algorithms due to Hopcroft [34], Brzozowski [11], Watson [74, 72], and the new incremental method described on Section 8.3. Using this implementation, we experimentally compared the algorithms' relative performance when minimising random samples of ICDFAs.

The choice of the algorithms is justified by the disparate worst-case complexities (presented on Table 8.1) and doubts about the practical behaviour of each algorithm. Moreover, since sometimes one is only interested in knowing if a given finite automaton is already minimal

Algorithm	Worst-case	Average-case	Experience
Moore	$O(k\,n^2)$	$O(k\,n\log(n))$	Good
Hopcroft	$O(k\,n\log(n))$	–	Good
Brzozowski	$O(2^n)$	$O(2^n)$	Bad
Watson	$O(k^{\max(0,n-2)})$	–	Bad
Watson & Daciuk	$O(k\,n^2\alpha(n^2))$	–	–
Incremental	$O(k\,n^2\alpha(n^2))$	–	Good

Table 8.1: Running time complexity of DFA minimisation algorithms.

(and not actually minimise it), the incremental algorithms are of particular interest since they may be halted as soon as the first pair of equivalent states is found.

Because DFA-MINIMISE-BRZOZOWSKI can be directly applied to NFAs, our experimental tests include benchmarks with non-deterministic finite automata. However, we do not consider NFA minimisation in its literal meaning and the output of our tests is always the equivalent minimal DFA.

We do not include any experimental results of the fully memoized and more efficient version of Watson's algorithm, proposed by Watson and Daciuk [73], because during the initial performance tests a bug was found and one of the authors is currently trying to fix it.

8.1 Related work

Lhoták [48] proposed a general data structure for DFA minimisation algorithms to run in $O(k\,n\log n)$, where n is the number of states of the DFA and k is the size of the alphabet. In his taxonomy, Watson [74] presented some experimental results on the performance of DFA minimisation algorithms, but the samples were rather small and biased. Tabakov and Vardi [69] experimentally compared Hopcroft and Brzozowski's algorithms. Baclet and Pagetti [7] analysed different implementations of Hopcroft's algorithm refinement process,

and Bassino et al. [9] experimentally compared Moore and Hopcroft's algorithms.

8.2 Algorithms

Over the years, several different finite algorithms automata minimisation algorithms have been developed and published. Although they all depend upon computing an equivalence relation on the set of states, several approaches are possible: explicitly computing the equivalence relation, computing the partition (of states) that it induces, or computing the complement of the equivalence relation (finding all pairs of distinguishable states).

Aho et al. [1, pages 157–162], for example, present the minimisation of finite automata as an application of a partitioning algorithm. The problem of minimising states in a finite automaton $D = (Q, \Sigma, \delta, q_0, F)$ is equivalent to the problem of finding the coarsest (having fewest blocks) partition P of Q such that:

- P is consistent with the the initial partition $\{ F, Q - F \}$, i.e., each block in P is a subset either F or $Q - F$;
- if the states p and q are in the same block, then the states $\delta(p, a)$ and $\delta(q, a)$ are also in the same block, for each $a \in \Sigma$.

The algorithm which authors typically present as "Moore's algorithm", based on the works of Huffman and Moore, finds the pairs of distinguishable states and uses this information to create the equivalence classes.

Hopcroft's algorithm presents a more efficient approach to the set partitioning problem, resulting in the fastest (in terms of worst-case running time complexity) known DFA minimisation algorithm.

Brzozowski's simple and elegant algorithm computes the equivalence classes in a somewhat mysterious manner. A result due to Champarnaud et al. [15] shows that given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, two states p and q are equivalent if and only if after the two first operations

(reversal and subset construction) they both belong to the same set $S \subseteq 2^Q$.

8.2.1 Moore's Algorithm

One way to minimise a DFA, usually credited to Huffman [37, 38] and Moore [52], is to determine all distinguishable states. Having been found, a minimal equivalent automaton can be constructed by collapsing any maximal set of mutually indistinguishable (equivalent) states into a single state.

Although neither Huffman nor Moore explicitly present an algorithm for minimising finite automata, they do define state equivalence, the distinguishability relation (and associated set partition), prove that a minimal finite automata has no equivalent states and that a minimal DFA is in fact unique.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Our version of the algorithm, DFA-MINIMISE-MOORE, runs in $O\big(|\Sigma||Q|^2\big)$ time. It is based on the algorithms given by Hopcroft and Ullman [36, page 70] — attributed to Huffman [37] and Moore [52] — and Shallit [66, page 87].

In their presentation, Hopcroft and Ullman use the variable L to map each pair of states to a list of related pairs of states. We implement it as an associative array (dictionary).

Given a DFA D, DFA-MINIMISE-MOORE returns the minimal DFA D' such that $D \sim D'$.

```
1  def DFA-MINIMISE-MOORE(Q, Σ, δ, q_0, F):
2      for p ∈ Q:
3          for q ∈ Q:
4              L[(p, q)] := ∅
5              if (p ∈ F) ∨̇ (q ∈ F):
6                  M[(p, q)] := TRUE
7              else:
8                  M[(p, q)] := FALSE
9      for p ∈ Q:
10         for q ∈ Q:
11             if (p ∈ F) ⇔ (q ∈ F):
```

```
12                    m := FALSE
13                    for a ∈ Σ:
14                        if M[(δ(p,a), δ(q,a))]:
15                            m := TRUE
16                            break
17                    if m:
18                        M[(p,q)] := TRUE
19                        MARK-SET(p,q)
20                    else:
21                        for a ∈ Σ:
22                            if δ(p,a) ≠ δ(q,a):
23                                x := (δ(p,a), δ(q,a))
24                                L[x] := L[x] ∪ {(p,q)}
25    D' := (Q, Σ, δ, q0, F)
26    for p ∈ Q:
27        for q ∈ Q:
28            if M[(p,q)] = FALSE
29                JOIN-STATES(D', {{p,q}})
30    return D'
```

The algorithm DFA-MINIMISE-MOORE proceeds by finding the smallest word which distinguishes a pair of states. Every pair of states (p, q) is assumed to be equivalent until some word which distinguishes p from q is found.

The two global variables, L and M, are used to keep a set associated to each pair of states, and mark a pair of states as either equivalent or distinguishable, respectively. They are global only to simplify the description of DFA-MINIMISE-MOORE and its integration with the auxiliary procedure MARK-SET. Given two states p and q, $L[(p, q)]$ contains a set of pairs of states which are distinguishable if and only if p is distinguishable from q. The states p and q are marked as distinguishable by making $M[(p, q)] = \text{TRUE}$.

The loop in lines 2–8 separates the trivially distinguishable pairs of states (those such that one state is final and the other is not) from the remaining pairs, assumed to be equivalent until proven otherwise.

The loop in lines 9–24 iterates through all possible pairs of states. For each pair of states (p, q) not yet marked as distinguishable (line 11), if an already marked pair of states reachable from p and q is found (line 17), (p, q) and all pairs on its set are also marked as distinguishable (line 19). If, on the other hand, no two states directly accessible from the pair (p, q) are marked, it is added to the set of each pair directly accessible from it (lines 21–24).

In lines 25–29 a copy of the original DFA is created, and each pair of equivalent states is merged by the call to JOIN-STATES. The minimal DFA D' is returned at line 30.

```
1  def MARK-SET(p, q):
2      for (p', q') ∈ L[(p, q)]:
3          if M[(p', q')] = FALSE:
4              M[(p', q')] := TRUE
5              MARK-SET(p', q')
```

The procedure MARK-SET recursively marks all unmarked pairs (p', q') on the list of (p, q), and all pairs on the list of (p', q'), etc.

8.2.2 Hopcroft's algorithm

In terms of worst-case complexity analysis, Hopcroft [34] presented, in 1971, the best known algorithm for deterministic finite automata minimisation. It runs on $O(k\,n \log n)$ time when applied to a DFA with n states over an alphabet of k symbols. It is presented as DFA-MINIMISE-HOPCROFT.

Although widely known and cited, the original algorithm's proofs of correctness and running time are quite difficult to understand. Gries [29] and Knuutila [43] present detailed and easy to follow tutorial reconstructions of the original algorithm, including time complexity analysis and correctness proofs.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Unlike Moore's algorithm (and several of its variations [35, pages 154–157]), DFA-MINIMISE-HOPCROFT does not identify pairs of distin-

guishable states. Instead, it proceeds by *refining* the coarsest partition until reaching a stable partition. The initial partition is $P = \{ F, Q - F \}$. At each step of the algorithm, a block $B \in P$ and a symbol $a \in \Sigma$ are selected to refine the partition. This refinement process *splits* every other block $B' \in P$ according to whether a state of B', when consuming a symbol $a \in \Sigma$, reaches a state which is in B or not. Formally, we call this procedure SPLIT and define it in the following way.

Definition 28.

$$\textsc{Split}(B', B, a) = \left(B' \cap \check{\delta}(B, a), B' \cap \overline{\check{\delta}(B, a)} \right)$$

where

$$\check{\delta}(Q', a) = \bigcup_{q \in Q'} \delta^{-1}(q, a), \quad Q' \subseteq Q.$$

DFA-MINIMISE-HOPCROFT terminates when there are no more blocks to refine. In the end, each block of the partition is a set of equivalent states and represents a single state of the minimal DFA.

```
1  def DFA-Minimise-Hopcroft(D := (Q, Σ, δ, q, F)):
2      L := ∅
3      if |F| < |Q − F|:
4          P := { Q − F, F }
5          L := Push(L, F)
6      else:
7          P := { F, Q − F }
8          L := Push(L, Q − F)
9      while L ≠ ∅:
10         C := Pop(L)
11         for a ∈ Σ:
12             for B ∈ P:
13                 (B1, B2) = Split(B, C, a)
14                 P := P − { B }
15                 P := P ∪ { B1, B2 }
16                 if |B1| < |B2|:
17                     L := Push(L, B1)
```

```
18                  else:
19                      L := PUSH(L, B_2)
20      D' := D
21      JOIN-STATES(D', P)
22      return D'
```

The data structure L, implemented as a stack, contains the blocks of P which are yet to be visited. At each iteration of the main loop (lines 9–19), one element (the *splitter*) is removed (line 10) from L and used in the splitting process (line 13). Although the choice of the element does not influence the correctness of the algorithm, Baclet and Pagetti [7] presented some experimental results stating that a *last-in-first-out* (stacked) policy yields better practical results.

Next, having selected a splitter set C, all elements of P (lines 12–19) are refined and used to update P and L. At this point, the original algorithm by Hopcroft applies the splitting process to and refines only blocks $B \in P$ such that there exists $t \in B$ with $\delta(t, a) \in \check{\delta}(C, a)$ for some $a \in \Sigma$. We omit this selection and iterate through all blocks in P because, for a given $B \in P$, if $\delta(t, a) \notin \check{\delta}(C, a)$ for all $t \in B$, the call SPLIT(B, C, a) will return $(\emptyset, B)$. This will change neither P nor L because:

- B is removed from P at line 14, but $B_2 = B$ is added at line 15;
- at line 16, $B_1 = \emptyset$; thus $|B_1| < |B_2|$ and $\emptyset$ is added to L at line 17.

```
1   def SPLIT(B, C, a):
2       X := ∅
3       for q ∈ C:
4           X := X ∪ {δ^{-1}(q, a)}
5       Y := Q − X
6       return (B ∩ X, B ∩ Y)
```

A call such as SPLIT(B, C, a) refines the set B into two smaller sets according to the splitter C and the transitions labelled by the symbol a (cf. Definition 28). The sets $\check{\delta}(C, a)$ and its

complement are computed at lines 2–4 and 5 (respectively) and the refined classes, obtained from the intersection with B, are returned at line 6.

8.2.3 Brzozowski's algorithm

Given a (possibly non-deterministic) finite automaton A without ϵ-transitions Brzozowski's algorithm builds the minimal DFA A_m, such that A_m is equivalent to A, simply by applying two successive reverse and subset construction operations. This is a deep result which relates two basic automata constructions (reversal and subset) with finite automata minimisation.

We present our implementation of Brzozowski's minimisation algorithm [11] as DFA-MINIMISE-BRZOZOWSKI. Although it assumes a DFA as input, it can very easily be generalised to handle NFAs, simply by using NFA-REVERSE instead of DFA-REVERSE in line 2.

```
1  def DFA-MINIMISE-BRZOZOWSKI(D):
2      D^R := FA-DETERMINISTIC(DFA-REVERSE(D))
3      D_m := FA-DETERMINISTIC(DFA-REVERSE(D^R))
4      return D_m
```

Let N be a non-deterministic finite automaton. The procedure NFA-REVERSE returns an NFA N_r such that the language recognized by N is the reversal of the one recognized by N_r, i.e., $L(N_r) = L^R(N)$. DFA-REVERSE performs the equivalent transformation to DFAs. The procedure FA-DETERMINISTIC, already presented in Subsection 3.2.1, takes an arbitrary NFA N as argument and returns an equivalent DFA D such that $L(N) = L(D)$.

```
1  def DFA-REVERSE(Q, Σ, δ, q_0, F):
2      return NFA-REVERSE(Q, Σ, δ, {q_0}, F)
```

```
1  def NFA-REVERSE(Q, Σ, δ, I, F):
2      I_r := F
3      F_r := I
```

```
4       for q ∈ Q :
5           for a ∈ Σ :
6               δ_r(q, a) := ∅
7       for q ∈ Q :
8           for a ∈ Σ :
9               for t ∈ δ(q, a) :
10                  δ_r(t, a) := δ_r(t, a) ∪ { q }
11      N_r := (Q, Σ, δ_r, I, F_r)
12      return N_r
```

Because of the peculiar way by which Brzozowski's algorithm computes the minimal DFA, Watson, in his taxonomy [74, page 193], assumed it to be unique and placed it apart all other algorithms. Later however, after having analysed how the sequential determinizations perform the minimisation, Champarnaud et al. [15] showed that DFA-MINIMISE-BRZOZOWSKI *does* compute state equivalences.

Since it invokes the subset construction algorithm FA-DETERMINISTIC twice — in order to make the two reversed automata deterministic — DFA-MINIMISE-BRZOZOWSKI is exponential in the worst case (cf. Subsection 3.2.1). Moreover, Nicaud [56] has proved that, for the specific case of group automata, the algorithm is also exponential on the average case. Nonetheless, some authors (e.g. Watson [74, page 333]) have stated that it does present very good practical results, sometimes even outperforming Hopcroft's $O(k\,n \log n)$ algorithm.

8.2.4 Watson's incremental algorithm

In 2001, Watson presented an *incremental* DFA minimisation algorithm [72]. Unlike other known finite automata minimisation algorithms (e.g. Moore or Hopcroft), this one may be halted at any time yielding a partially minimised DFA that recognises the same language as the input DFA. This is the first (known) algorithm with usable intermediate results. Later, Watson and Daciuk proposed an improved version of the same algorithm [73], this time making use of full memoization in order to obtain a lower worst-case running time.

Given a DFA $D = (Q, \Sigma, \delta, q_0, F)$, the original algorithm, DFA-MINIMISE-WATSON, presents a worst-case exponential running time: $O\big(|\Sigma|^{\max(0,|Q|-2)}\big)$. The memoized version, on the other hand, yields an almost quadratic algorithm, $O\big(|\Sigma||Q|^2\alpha\big(|Q|^2\big)\big)$, where α denotes an inverse of the Ackermann function. Since $\alpha(x) \leq 4$ for any $x \leq 16^{512}$ (cf. Subsection 2.5.1), it can be considered a constant for all "practical" values of x. During our initial tests, however, a rather serious problem was found and one of the authors is currently trying to fix it. Therefore, we were not able to include this algorithm in the benchmarks.

Exemplifying the problem The DFA on Figure 8.1 illustrates the bug on Watson and Daciuk's minimisation algorithm. Although the ICDFA on the left is already minimal, the algorithm fails to compute the correct equivalence classes and returns the DFA on the right, with only 3 states. Clearly, the DFAs do not recognise the same language.

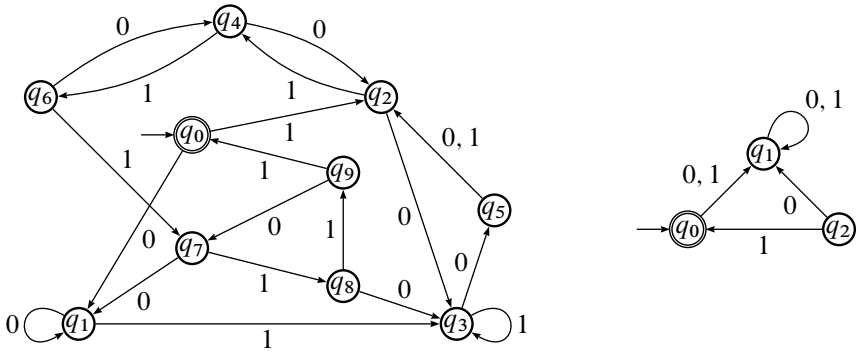


Figure 8.1: An ICDFA that Watson & Daciuk's algorithm fails to minimise.

```
1  def DFA-MINIMISE-WATSON(Q, Σ, δ, q0, F):
2      E := { (q, q) | q ∈ Q }
3      Ē := ((Q − F) × F) ∪ (F × (Q − F))
4      k := max(0, |Q| − 2)
5      S := ∅
6      for p ∈ Q:
7          for q ∈ Q:
8              if WATSON-EQUIV-P(p, q, k):
```

```
9                   E := E ∪ {(p, q), (q, p)}
10              else:
11                  Ē := Ē ∪ {(p, q), (q, p)}
12      D′ := (Q, Σ, δ, q, F)
13      for (p, q) ∈ E:
14          JOIN-STATES(D′, (p, q))
15      return D′
```

DFA-MINIMISE-WATSON is a straightforward implementation of Watson's original algorithm, but specialised for ICDFAs. At each point of the main loop (lines 6–11), variables E and $\bar{E}$ maintain the sets of pairs of states already known to be equivalent and not-equivalent, respectively. This loop can be interrupted at any time, and the partially computed set of equivalent pairs of states E can be used to merge states. Variable k is used to limit the recursion depth of the auxiliary procedure WATSON-EQUIV-P, only for matters of efficiency. The proof that a word of size $|Q| - 2$ suffices to distinguish any pair of states may be found in the works of Conway [19, page 11] or Wood [75, page 129]. Also for matters of efficiency, a variable S, containing a set of presumably equivalent pairs of states, is made global. Lines 12–15 are not part of the original algorithm, but we include them in order to make the construction of the minimal DFA explicit.

DFA-MINIMISE-WATSON makes use of an auxiliary function, WATSON-EQUIV-P, which tests the equivalence of any two states, p and q. The third argument, k, is the afore mentioned parameter used to impose a limit on the recursion depth.

```
1  def WATSON-EQUIV-P(p, q0, k):
2      if k = 0:
3          return (p ∈ F ⇔ q ∈ F)
4      elif (p, q) ∈ S:
5          return TRUE
6      else:
7          eq := (p ∈ F ⇔ q ∈ F)
8          S := S ∪ {(p, q)}
9          for a ∈ Σ:
```

```
10                if eq = FALSE:
11                    return FALSE
12                eq := eq ∧ WATSON-EQUIV-P(δ(p, a), δ(q, a), k − 1)
13            S := S − {(p, q)}
14        return eq
```

WATSON-EQUIV-P follows the transition function and recursively calls itself with increasingly longer words (line 12) until one of the following two conditions occurs:

- the recursion limit was reached (line 2), which means that the two states used in the first call to WATSON-EQUIV-P (line 8 of DFA-MINIMISE-WATSON) are equivalent if and only if they are either both final or both not final, since the longest word necessary to distinguish them has been computed;
- a loop was found (line 4), and clearly the states are equivalent since no word that distinguishes the states has been, or will be, found.

The ability to interrupt the algorithm presents an excellent opportunity to save computational resources when, given a DFA, the goal is not to obtain the equivalent minimal automaton, but solely to check if it is already minimal. The procedure DFA-MINIMAL-WATSON-P implements a simplified version of Watson's algorithm which halts, returning FALSE, when the first pair of equivalent states is found. Naturally, if no pair of states is found to be equivalent, the algorithm returns TRUE, indicating that the input DFA is in fact already minimal.

```
1  def DFA-MINIMAL-WATSON-P(Q, Σ, δ, q0, F):
2      k := max(0, |Q| − 2)
3      S := ∅
4      for p ∈ Q:
5          for q ∈ Q:
6              if WATSON-EQUIV-P(p, q, k):
7                  return FALSE
8      return TRUE
```

8.3 A new incremental method

We will now present a new quadratic incremental DFA minimisation algorithm. Given an arbitrary DFA D as input, this algorithm may be halted at any time returning a partially minimised DFA that has no more states than D and recognises the same language. Whenever the minimisation process is interrupted, calling the incremental minimisation algorithm with the output of the halted process resumes the minimisation process. Being incremental also allows for the algorithm to be applied to an automaton D at the same time as D is being used to process a word for acceptance.

The algorithm uses a disjoint-set data structure to represent the DFA's states. UNION-FIND is used to mark pairs of equivalent states and to keep and update the equivalence classes. This approach maintains the transitive closure in a very concise and elegant manner. The pairs of states marked as distinguishable are stored in an auxiliary data structure in order to avoid repeated computations.

Unlike the usual technique, which computes the equivalence classes of the set of states, this algorithm proceeds by testing the equivalence of pairs of states. The intermediate results are stored for the speedup of future computations in order to assure quadratic running time and memory usage.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA such that $n = |Q|$ and $k = |\Sigma|$. We assume that the states are represented by integers, and thus it is possible to order them. This ordering is used to normalise pairs of states.

```
1 def NORMALISE-PAIR(p, q):
2     if p < q:
3         return (p, q)
4     else:
5         return (q, p)
```

The normalisation step allows us to improve the behaviour of the minimisation algorithm by ensuring that only $\frac{n^2-n}{2}$ pairs of states are considered.

The quadratic time bound of the minimisation procedure DFA-MINIMISE-INCREMENTAL is achieved by testing each pair of states for equivalence exactly once. We assure this by storing the intermediate results of all calls to the pairwise equivalence-testing function INCREMENTAL-EQUIV-P. Some auxiliary data structures, designed specifically to improve the worst-case running time, are presented on Subsection 8.3.1.

```
1  def DFA-MINIMISE-INCREMENTAL((Q, Σ, δ, q0, F)):
2      for q ∈ Q:
3          MAKE(q)
4      Ē := { NORMALISE-PAIR(p, q) | p ∈ F, q ∈ Q − F }
5      for p ∈ Q:
6          for q ∈ { x | x ∈ Q, x > p }:
7              if (p, q) ∈ Ē:
8                  continue
9              if FIND(p) = FIND(q):
10                 continue
11             E := SET-MAKE(|Q|²)
12             H := SET-MAKE(|Q|²)
13             if INCREMENTAL-EQUIV-P(p, q):
14                 for (p′, q′) ∈ SET-ELEMENTS(E):
15                     UNION(p′, q′)
16             else:
17                 for (p′, q′) ∈ SET-ELEMENTS(H):
18                     Ē := Ē ∪ { (p′, q′) }
19     C := {}
20     for p ∈ Q:
21         r := FIND(p)
22         C[r] := C[r] ∪ { p }
23     D′ := D
24     JOIN-STATES(D′, C)
25     return D′
```

The algorithm DFA-MINIMISE-INCREMENTAL starts by creating the initial equivalence classes (lines 2–3); these are singletons as no states are yet marked as equivalent. The

global variable $\overline{E}$, used to store the distinguishable pairs of states, is also initialised (line 4) with the trivial identifications. Variables H and E, also global and reset before each call to INCREMENTAL-EQUIV-P, maintain the history of calls to the transition function and the set of potentially equivalent pairs of states, respectively.

The main loop of DFA-MINIMISE-INCREMENTAL (lines 5–18) iterates through all the normalised pairs of states and, for those not yet known to be either distinguishable or equivalent, calls the pairwise equivalence test INCREMENTAL-EQUIV-P. Every call to INCREMENTAL-EQUIV-P is conclusive and the result is stored either by merging the corresponding equivalence classes (lines 13–15), or updating $\overline{E}$ (lines 16–18). Thus, each recursive call to INCREMENTAL-EQUIV-P will avoid one iteration on the main loop of DFA-MINIMISE-INCREMENTAL by skipping (lines 7–10) that pair of states.

Finally, at lines 19–22, the set partition of the corresponding equivalence classes is created. Next, the DFA D is copied to D' (line 23) and the equivalent states are merged by the call to JOIN-STATES. The last instruction, at line 25, returns the minimal DFA D' equivalent to D.

```
1  def INCREMENTAL-EQUIV-P(p, q):
2      if (p, q) ∈ Ē:
3          return FALSE
4      if SET-SEARCH((p, q), H) ≠ NIL:
5          return TRUE
6      H := SET-INSERT((p, q), H)
7      for a ∈ Σ:
8          (p', q') := NORMALISE-PAIR(FIND(δ(p, a)), FIND(δ(q, a)))
9          if p' ≠ q' and SET-SEARCH((p', q'), E) = NIL:
10             E := SET-INSERT((p', q'), E)
11             if not INCREMENTAL-EQUIV-P(p', q'):
12                 return FALSE
13             else:
14                 H := SET-REMOVE((p', q'), H)
15     E := SET-INSERT((p, q), E)
16     return TRUE
```

Algorithm INCREMENTAL-EQUIV-P is used to test the equivalence of the two states, p and q, passed as arguments.

The global variables E and H are updated with the pair (p, q) during each nested recursive call. As there is no recursion limit, INCREMENTAL-EQUIV-P will only return when p is distinguishable from q (line 3) or when a cycle is found (line 5). If a call to INCREMENTAL-EQUIV-P returns FALSE, then all pairs of states recursively tested are distinguishable and variable H — used to store the sequence of calls to the transition function — will contain a set of distinguishable pairs of states. If it returns TRUE, no pair of distinguishable states was found within the cycle and variable E will contain a set of equivalent states. This is the strategy which assures that each pair of states is tested for equivalence exactly once: every call to INCREMENTAL-EQUIV-P is conclusive and the result stored for future use. It does, however, lead to an increased usage of memory.

Theorem 29. *The algorithm* DFA-MINIMISE-INCREMENTAL *is terminating.*

Proof. It should suffice to notice the following facts:

- all the loops in DFA-MINIMISE-INCREMENTAL are finite;
- the variable H on INCREMENTAL-EQUIV-P assures that the number of recursive calls is finite.

□

Lemma 30. *The algorithm* INCREMENTAL-EQUIV-P *runs in* $O(k\,n^2)$ *time.*

Proof. The number of recursive calls to INCREMENTAL-EQUIV-P is controlled by the local variable H. This variable keeps the history of calls to the transition function (line 8). In the worst case, all possible pairs of states are used: $\frac{n^2-n}{2}$, due to the normalisation step. Since each call may reach line 7, we need to consider k additional recursive calls for each pair of states, hence $O(k\,n^2)$. □

Lemma 31. *The algorithm* INCREMENTAL-EQUIV-P *returns* TRUE *if and only if the two states passed as arguments are equivalent.*

Proof. INCREMENTAL-EQUIV-P returns FALSE if the two states, p and q, used as arguments are such that $(p, q) \in \bar{E}$ (lines 2–3). This is correct because the global variable $\bar{E}$ contains all the pairs of states already proven to be distinguishable. Conversely, INCREMENTAL-EQUIV-P returns TRUE only if $(p, q) \in H$ (lines 4–5) or a recursive call returned TRUE (line 16). In both cases this means that a cycle with no distinguishable elements was detected, which implies that all the recursively visited pairs of states are equivalent. □

Theorem 32. *Given a DFA* $D = (Q, \Sigma, \delta, q_0, F)$*,* DFA-MINIMISE-INCREMENTAL *computes the minimal DFA* D' *such that* $D \sim D'$*.*

Proof. The procedure DFA-MINIMISE-INCREMENTAL finds pairs of equivalent states by exhaustive enumeration. The loop in lines 5–18 enumerates all possible pairs of states, and, for those not yet proven to be either distinguishable or equivalent, INCREMENTAL-EQUIV-P is called. When line 19 is reached, all pairs of states have been enumerated and the equivalent ones have been found (cf. Lemma 31). The loop in lines 20–22 creates the equivalence classes and the procedure JOIN-STATES, at line 24, merges the equivalent states, updating the corresponding transitions. Since the new DFA D' does not have any equivalent states, it is minimal. □

Lemma 33. *At line 13 of* DFA-MINIMISE-INCREMENTAL*, when* INCREMENTAL-EQUIV-P *returns* TRUE*, all the pairs of states stored in the global variable* E *are equivalent.*

Proof. By Lemma 31, if INCREMENTAL-EQUIV-P returns TRUE then the two states, p and q, used as arguments are equivalent. Since there is no depth recursion control, INCREMENTAL-EQUIV-P only returns TRUE when a cycle is detected. Thus being, all the pairs of states used as arguments in the recursive calls must also be equivalent. These pairs of states are stored in the global variable E at line 10 of INCREMENTAL-EQUIV-P. □

Lemma 34. *At line 13 of* DFA-MINIMISE-INCREMENTAL*, if* INCREMENTAL-EQUIV-P *returns* FALSE*, all the pairs of states stored in the global variable* H *are distinguishable.*

Proof. Given a pair of distinguishable states (p, q), clearly all pairs of states (p', q') such that $\hat{\delta}(p', w) = p$ and $\hat{\delta}(q', w) = q$ are also distinguishable, for $w \in \Sigma^\star$. By Lemma 31, INCREMENTAL-EQUIV-P returns FALSE only when the two states, p and q, used as arguments are distinguishable. Throughout the successive recursive calls to INCREMENTAL-EQUIV-P, the global variable H is used to store the history of calls to the transition function (line 6) and thus contains only pairs of states with a path to (p, q). All of these pairs of states are therefore distinguishable. □

Lemma 35. *Each time that* INCREMENTAL-EQUIV-P *calls itself recursively, the two states used as arguments will not be considered in the main loop of* DFA-MINIMISE-INCREMENTAL.

Proof. The arguments of every call of INCREMENTAL-EQUIV-P are kept in two global variables: E and H.

By Lemma 33, whenever INCREMENTAL-EQUIV-P returns TRUE, all the pairs of states stored in E are equivalent. Immediately after being called from DFA-MINIMISE-INCREMENTAL (line 13), if INCREMENTAL-EQUIV-P returns TRUE, the equivalence classes of all the pairs of states in E are merged (lines 14–15). Future references to any of these pairs will be skipped at lines 9–10.

In the same way, by Lemma 34, if INCREMENTAL-EQUIV-P returns FALSE, all the pairs of states stored in H are distinguishable. Lines 17–18 of DFA-MINIMISE-INCREMENTAL update the global variable $\bar{E}$ with this new information and future references to any of these pairs of states will be skipped at lines 7–8 of DFA-MINIMISE-INCREMENTAL. □

Theorem 36. *Algorithm* DFA-MINIMISE-INCREMENTAL *is incremental.*

Proof. Halting the main loop of DFA-MINIMISE-INCREMENTAL at any point within the lines 5–18 only prevents the finding of *all* the equivalent pairs of states. Merging the known equivalent states on D', a copy of the input DFA D, assures that the size of D' is not greater than that of D and thus, is closer to the minimal equivalent DFA. Calling DFA-MINIMISE-

INCREMENTAL with D' as the argument would resume the minimisation process, finding the remaining equivalent states. □

Theorem 37. *Algorithm* DFA-MINIMISE-INCREMENTAL *runs in* $O(kn^2\alpha(n))$ *time.*

Proof. The number of iterations of the main loop in lines 5–18 of DFA-MINIMISE-INCREMENTAL is bounded by $\frac{n^2-n}{2}$. Each iteration may call INCREMENTAL-EQUIV-P, which, by Lemma 30, is $O(kn^2)$. By Lemma 35 every recursive call to INCREMENTAL-EQUIV-P avoids one iteration on the main loop. Therefore, disregarding the UNION-FIND calls, and because all operations on variables $\overline{E}$, E, and H are $O(1)$, the $O(kn^2)$ bound holds. Since there are $O(kn^2)$ FIND and UNION intermixed calls, and exactly n MAKE calls, the time spent on all the UNION-FIND operations is bounded by $O(kn^2\alpha(n))$ — cf. Section 2.5. All things considered, DFA-MINIMISE-INCREMENTAL runs in $O(kn^2 + kn^2\alpha(n)) = O(kn^2\alpha(n))$. □

Corollary 38. *Algorithm* DFA-MINIMISE-INCREMENTAL *runs in* $O(kn^2)$ *time for all* practical *values of* n.

Proof. Function α is related to an inverse of Ackermann's function. It grows so slowly (cf. Subsection 2.5.1) that we may consider it a constant. □

8.3.1 Efficient set implementation

The variables E and H are heavily used in INCREMENTAL-EQUIV-P as several insert, remove, and membership-test operations are executed throughout the algorithm. In order to achieve the desired quadratic upper bound, all these operations must be performed in $O(1)$. Therefore, in the following paragraphs, we describe some efficient set representation and manipulation procedures.

```
1  def SET-MAKE(n) :
2      T := HASH-TABLE(n)
3      L := LINKED-LIST()
```

```
4       return (T, L)
5
6   def SET-INSERT(v, (T, L)) :
7       p := LIST-INSERT(v, L)
8       T[v] := p
9       return (T, L)
10
11  def SET-REMOVE(v, (T, L)) :
12      p := T[v]
13      LIST-REMOVE(p, L)
14      T[v] := NIL
15      return (T, L)
16
17  def SET-SEARCH(v, (T, L)) :
18      if T[v] ≠ NIL :
19          p := T[v]
20          return LIST-ELEMENT(p, L)
21      else :
22          return NIL
23
24  def SET-ELEMENTS((T, L)) :
25      return L
```

These set-manipulation procedures combine a hash-table with a doubly-linked list. This is another space-time trade-off that allows us to assure the desired complexity on all operations. The hash-table maps a given value (state of the DFA) to the address on which it is stored in the linked list. Since we know the size of the hash-table in advance — n^2 elements for a DFA with n states — searching, inserting, and removing elements is $O(1)$. The linked list assures that, at lines 14–15 and 17–18 of DFA-MINIMISE-INCREMENTAL, the loop is repeated only on the elements that were actually used in the calls to INCREMENTAL-EQUIV-P, instead of iterating through the entire hash-table.

The procedure SET-MAKE creates a new set represented by a tuple (T, L) where T is a hash-table and L a linked list. Its only argument is an integer defining the maximum size

of the set. This information is necessary when the set is represented by a *direct-address table* [20, pages 222–223]. If the number of states becomes too large for a single table, another option, somewhat more complicated, is *perfect hashing* [20, pages 245–249]. Both representations assure $O(1)$ search, insert, and remove operations.

SET-INSERT adds a new element to the set. The call to LIST-INSERT (line 7) adds the element v to the linked list L, returning a pointer p to the memory address where v was stored. Next, at line 8, p is used as the value to the key v on the hash-table T. All these operations are performed in constant time. The updated set, represented by the tuple (T, L), is returned at the end of the procedure (line 9).

Removing an element v from a set is implemented by the procedure SET-REMOVE. Since the hash-table stores memory addresses, SET-REMOVE starts by obtaining, at line 12, the value p (memory address) of the key v. After using p to remove v from the linked list (line 13), p is replaced by the special value NIL as the pointer to the value v, meaning that v is no longer on the set. The updated set, without the element v, is returned at line 15.

The procedure SET-SEARCH tries to locate an element v on a set (T, L). If the memory address of the element v (stored in the hash-table T) is valid, the pointer p is retrieved from T (line 19) and the corresponding value stored in the linked list is returned (line 20). If, on the other hand, there is no valid memory address for v (line 21), it is not an element of the set, and NIL is returned (line 22).

8.3.2 An example

Let us walk through the minimisation process of the DFA presented on Figure 8.2, exemplifying the incremental minimisation algorithm in action. It is a simple ICDFA with 5 states, named sequentially from 0 to 4, over an alphabet of two symbols, a and b. The only final state (and also the initial state) is 0.

After creating the UNION-FIND data structures with lines 2–3, DFA-MINIMISE-INCREMENTAL will initialise the set of distinguishable pairs of states (variable $\overline{E}$ at line 4) with

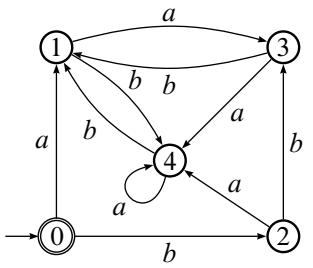


Figure 8.2: Incremental minimisation example: input DFA.

the trivial identifications: $\{(0,1),(0,3),(0,2),(0,4)\}$.

Following the order induced by the states' names, the first pair to be tested for equivalence is $(0,1)$. These states are already known to be distinguishable and will be ignored at lines 7–8. The same is true for the pairs $(0,2)$, $(0,3)$, and $(0,4)$.

The next pair to be tested for equivalence is $(1,2)$. Nothing is known about this pair so INCREMENTAL-EQUIV-P is called. Following the transitions by a, the pair $(3,4)$ is reached. From $(3,4)$, regardless of the symbol used, a loop is reached: $(4,4)$ by a, and $(1,1)$ by b. Another loop is found when trying to follow the transitions of the states $(1,2)$ by the symbol b. This results on INCREMENTAL-EQUIV-P returning TRUE with $E = \{(1,2),(3,4)\}$, and DFA-MINIMISE-INCREMENTAL making state 1 equivalent to state 2, and state 3 equivalent to state 4 by merging the corresponding UNION-FIND sets at lines 13–15. The partially minimised DFA obtained at this point is presented in Figure 8.3.

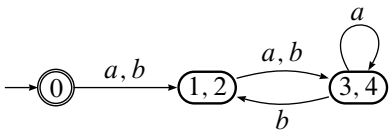


Figure 8.3: Incremental minimisation example: partially minimised DFA.

Continuing with the minimisation process, again, nothing is yet known about the equivalence of the states 1 and 3. The call to INCREMENTAL-EQUIV-P returns TRUE and sets the global variable $E = \{(1,3),(2,4)\}$. After merging the equivalence classes of the states $(1,3)$ and $(2,4)$, the remaining pairs of states — $(1,4)$, $(2,3)$, $(2,4)$, and $(3,4)$ — are

known to be equivalent and are ignored at lines 9–10 of DFA-MINIMISE-INCREMENTAL.

Since there are no more states to test, line 19 of DFA-MINIMISE-INCREMENTAL is reached. The set partition corresponding to the equivalence classes is created (lines 20–22), and the equivalent states are merged (line 24). The resulting minimal DFA is presented on Figure 8.4.

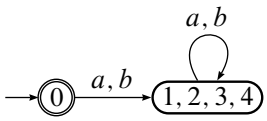


Figure 8.4: Incremental minimisation example: minimal DFA.

8.4 Experimental results

In the following subsections, we present some comparative experimental results. Here we include only a few three-dimensional graphs which represent the *performance* of a minimisation algorithm: the number of minimised finite automata per second. These are intended to be a bird's-eye view of the overall performance of the algorithms discussed previously on this Chapter. Appendix C includes complete tables with the exact values of the running time, memory usage, etc. for each algorithm.

The rule of thumb for reading the graphics is as follows: a darker area means lower values, and therefore, poorer performance.

8.4.1 ICDFAs

Looking at the results of the two classical polynomial algorithms, DFA-MINIMISE-MOORE and DFA-MINIMISE-HOPCROFT, we can see that, although the difference is not huge, DFA-MINIMISE-MOORE performs slightly better.

As for the two exponential algorithms, DFA-MINIMISE-BRZOZOWSKI and DFA-MINI-

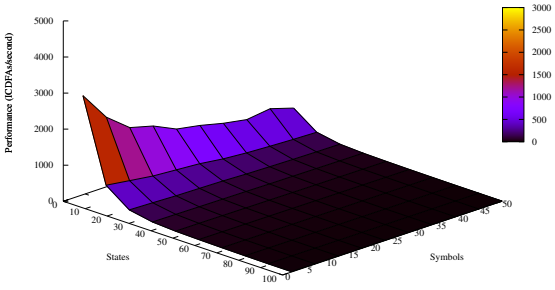
MISE-WATSON, while both present very similar results, DFA-MINIMISE-WATSON displays a slightly superior performance especially when dealing with small alphabets (25 symbols or less). For larger alphabets (30 or more symbols, in our tests), DFA-MINIMISE-BRZOZOWSKI outperforms DFA-MINIMISE-WATSON.

The clear winner is, without any doubt, the new incremental algorithm DFA-MINIMISE-INCREMENTAL. While presenting a quadratic worst-case running-time, it outperforms even Hopcroft's $O(k\,n\log(n))$ approach — at least in the average case, for a uniform distribution — on all the tested ICDFAs.

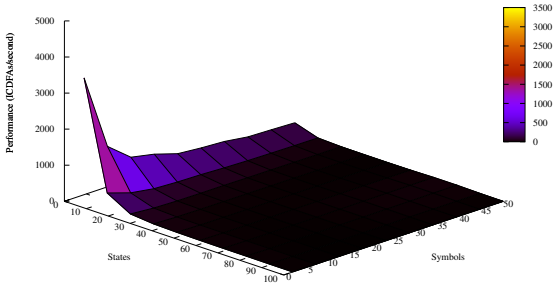
8.4.2 NFAs

Considering the benchmarks on the samples of NFAs with transition density $d = 0.1$, apart from the performance peak (common to all algorithms) when the number of states equals 5, Brzozowski's algorithm is usually the fastest. Sporadically, it is outmatched only by the quadratic incremental algorithm. Considering the combinatorial explosion in the NFA-to-DFA conversion (cf. Appendix F), it is not very surprising that the algorithms which explicitly invoke the subset method present rather poor results. Probably contrary to what one would expect, however, Brzozowski's algorithm is relatively moderate in terms of memory usage. In fact, the two quadratic algorithms (Moore's and the new incremental one) clearly present the more considerable expenditure of memory (cf. Appendix C, Section C.2).

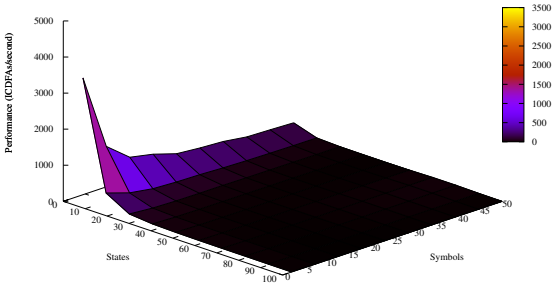
When the transition density increases to $d = 0.5$ or $d = 0.8$, except for some very rare cases, Brzozowski's algorithm is clearly the fastest approach to NFA minimisation. Regardless of the number of states or size of the alphabet, no other algorithm minimises a greater number of NFAs per second.



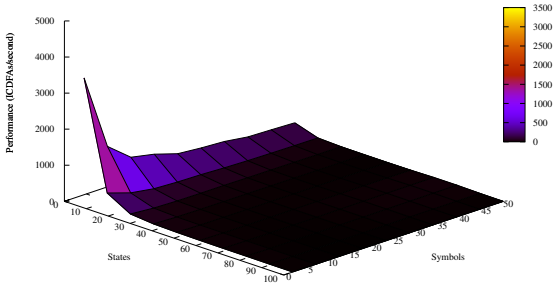
(a) DFA-MINIMISE-MOORE



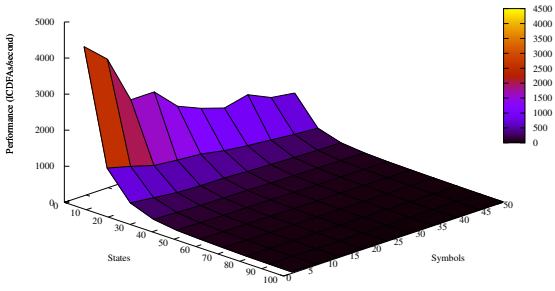
(b) DFA-MINIMISE-HOPCROFT



(c) DFA-MINIMISE-BRZOZOWSKI

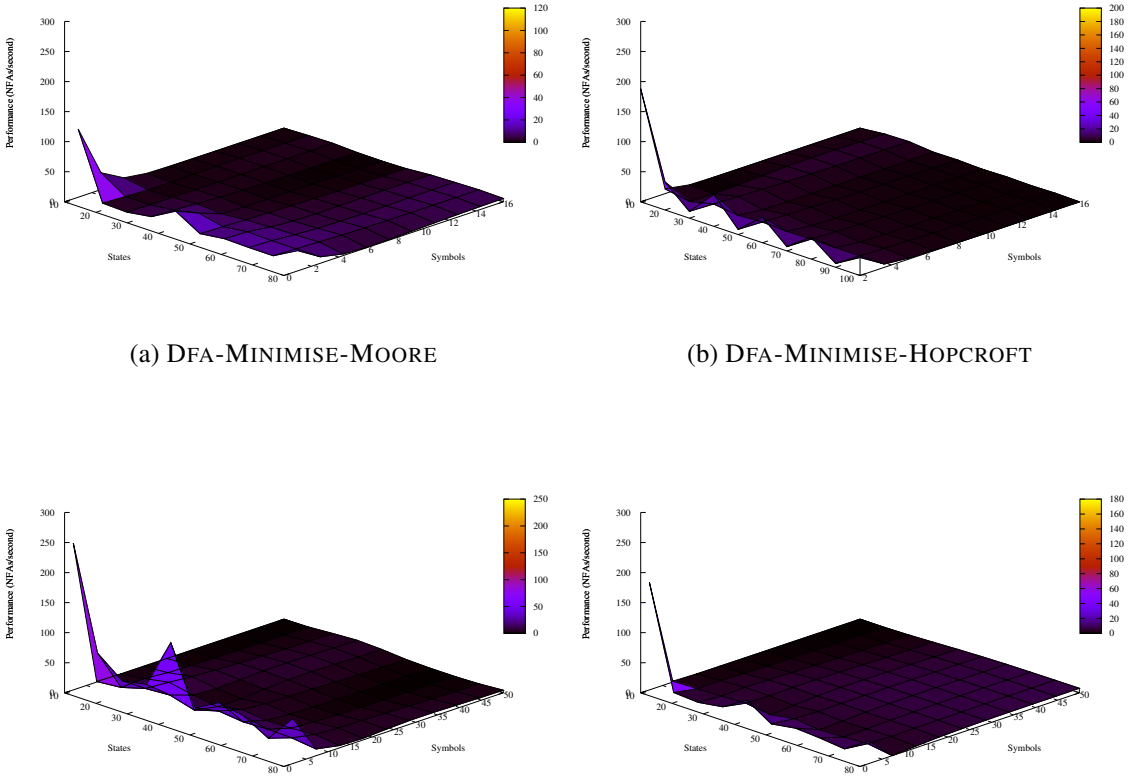


(d) DFA-MINIMISE-WATSON



(e) DFA-MINIMISE-INCREMENTAL

Figure 8.5: Performance graphics: minimisation of ICDFAs.



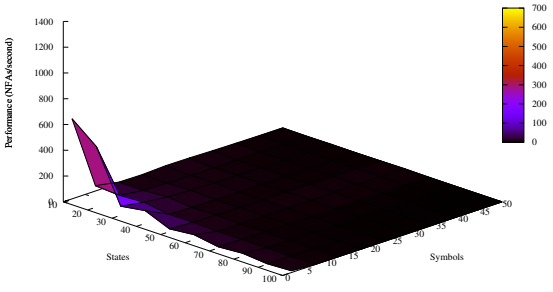
(a) DFA-MINIMISE-MOORE

(b) DFA-MINIMISE-HOPCROFT

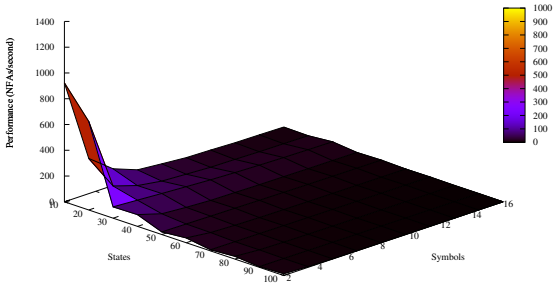
(c) DFA-MINIMISE-BRZOZOWSKI

(d) DFA-MINIMISE-INCREMENTAL

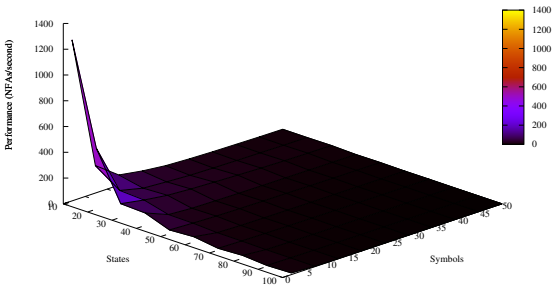
Figure 8.6: Performance graphics: minimisation of NFAs with transition density $d = 0.1$.



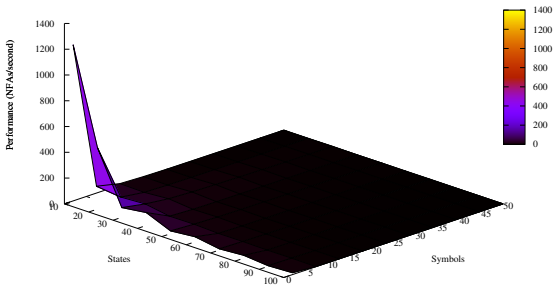
(a) DFA-MINIMISE-MOORE



(b) DFA-MINIMISE-HOPCROFT

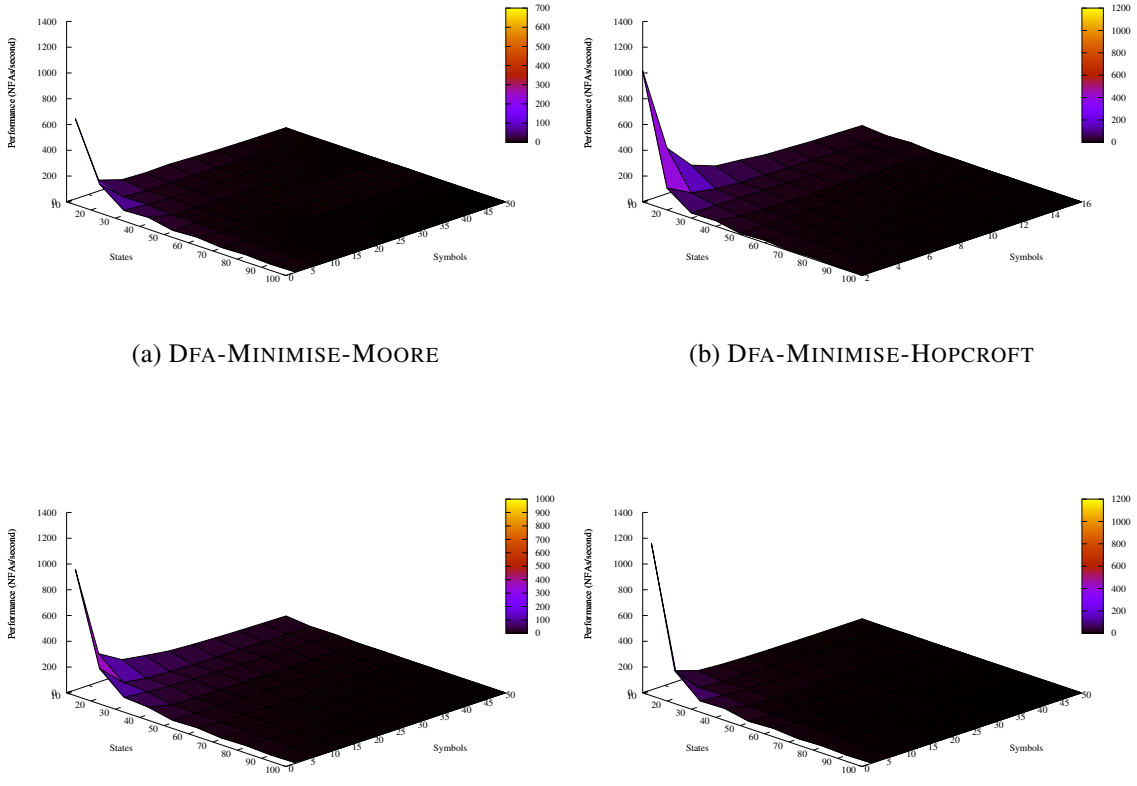


(c) DFA-MINIMISE-BRZOZOWSKI



(d) DFA-MINIMISE-INCREMENTAL

Figure 8.7: Performance graphics: minimisation of NFAs with transition density $d = 0.5$.



(a) DFA-MINIMISE-MOORE

(b) DFA-MINIMISE-HOPCROFT

(c) DFA-MINIMISE-BRZOZOWSKI

(d) DFA-MINIMISE-INCREMENTAL

Figure 8.8: Performance graphics: minimisation of NFAs with transition density $d = 0.8$.

Chapter 9

Finite automata equivalence

When trying to determine the equivalence of two regular languages, whether these are represented by finite automata or regular expressions, we tend to use the corresponding minimal DFAs. We know, from Corollary 10 (page 34), that for each regular language there is a unique (up to renaming of states) minimal DFA that recognises it, and use this canonical representation as the means to identify regular languages in a straightforward manner. Several minimisation algorithms, with different approaches and worst-case running-time complexity results, were already presented in Chapter 8.

There are, however, alternative approaches to the decidability of regular objects' equivalence which do not resort to minimisation. Some of them date already to the early 1970's.

In this Chapter, we will discuss different approaches to this problem, presenting some algorithms which are able to decide the equivalence of finite automata (deterministic and non-deterministic) *without* resorting to the minimal DFA.

9.1 Classical approach

When comparing the language recognised by two finite automata, F_1 and F_2, the usual approach relies on the uniqueness of the minimal equivalent DFAs. Since only two known

minimisation algorithms may be directly applied to non-deterministic finite automata — Brzozowski's [11], and the generalisation we present on Section 9.4 — if either F_1 or F_2 are non-deterministic, it may be necessary to obtain equivalent deterministic ones before using the minimisation procedure. This may be achieved with the subset construction method (cf. Subsection 3.2.1).

The predicate DFA-EQUIVALENT-P implements the simple minimisation-based equivalence test returning TRUE if and only if the finite automata F_1 and F_2 recognise the same language. It assumes that the minimisation algorithm can only handle DFAs and calls the MAKE-DETERMINISTIC procedure on the input automata.

```
1  def DFA-EQUIVALENT-P(F1, F2):
2      D1 := MAKE-DETERMINISTIC(F1)
3      D2 := MAKE-DETERMINISTIC(F2)
4      D1' := MINIMISE(D1)
5      D2' := MINIMISE(D2)
6      if DFA-ISOMORPHIC-P(D1', D2'):
7          return TRUE
8      else
9          return FALSE
10
11 def DFA-ISOMORPHIC-P(D1, D2):
12     S1 := ICDFA-TO-STRING(D1)
13     S2 := ICDFA-TO-STRING(D2)
14     if S1 = S2:
15         return TRUE
16     else
17         return FALSE
```

Because the minimal DFA is unique only up to renaming of states, it is necessary to check if the resulting minimal DFAs are isomorphic after completing the minimisation processes. This check, as implemented in DFA-ISOMORPHIC-P, can be performed in linear time when using the canonical string representation proposed by Reis et al. [60], already described on Subsection 3.1.1. This string is obtained with the call to ICDFA-TO-STRING.

In terms of worst-case complexity analysis, the best known DFA minimisation algorithm, by Hopcroft [34], is log–linear as presented in Chapter 8. The running time of this algorithm is $O(k\,n\log(n))$ when applied to a DFA with n states over an alphabet of k symbols. Because the isomorphism check is linear, the minimisation time dominates the equivalence test algorithm. Thus, when resorting to DFA minimisation to check the equivalence of two DFAs, $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, we can not expect a running time better than $O(k\,n\log(n))$, where $k = |\Sigma|$ and $n = \max(|Q_1|, |Q_2|)$.

As for NFAs, and since that at one point or another they use the subset construction, both Brzozowski's algorithm and the one we propose on Section 9.4 present an exponential running time.

9.2 An almost linear algorithm

Aho, Hopcroft, and Ullman presented [1, pages 143–145], in 1974, an algorithm for testing the equivalence of DFAs without resorting to any minimisation process. This algorithm makes use of a disjoint-set data structure to represent the states of the automata and the UNION-FIND algorithm [1, pages 124–129] to create and locate the equivalence classes. Our implementation follows the one described on Section 2.5.

9.2.1 Historical note

The first reference to this work is a 1971 technical report Hopcroft and Karp [31]. It cites another technical report, by Hopcroft and Ullman [32], describing the set-merging algorithm on which the equivalence testing algorithm is based. Hopcroft and Ullman [33] later improved the complexity analysis of the set-merging algorithm by showing it was bounded by a very slowly growing function: the iterated logarithm. As already stated on Subsection 2.5.1, Tarjan [70] took a step further and used the inverse Ackermann function — which grows even slower than the iterated logarithm — to prove a tighter upper bound

and a restricted version of the lower bound. This is the upper bound we will be considering throughout this Chapter. Although not directly related to the original algorithm on finite automata equivalence, the result affects all work relying on disjoint-set forest data structures, such as the case of this equivalence-testing method.

9.2.2 The original algorithm

Let $D_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $D_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be two DFAs, with $|Q_1| = n_1$, $|Q_2| = n_2$, such that Q_1 and Q_2 are disjoint, i.e., $Q_1 \cap Q_2 = \emptyset$. In order to simplify notation, we assume $Q = Q_1 \cup Q_2$, $F = F_1 \cup F_2$, and $\delta(p, a) = \delta_i(p, a)$ for $p \in Q_i$, $a \in \Sigma$, $i \in \{1, 2\}$.

The predicate DFA-EQUIVALENT-HK-P implements the method proposed by Aho et al. [1, pages 143–145] to determine if two finite automata are equivalent without using minimisation algorithms. It is based on the following observations. Suppose D_1 and D_2 are equivalent, then:

- q_1 and q_2 must be equivalent;
- if $p, q \in Q$ are equivalent, then $\delta(p, a)$ must be equivalent to $\delta(q, a)$, for all $a \in \Sigma$.

```
1  def DFA-EQUIVALENT-HK-P(D1, D2):
2      for q ∈ Q:
3          MAKE(q)
4      S := ∅
5      UNION(q1, q2)
6      S := PUSH(S, (q1, q2))
7      while (p, q) := POP(S):
8          for a ∈ Σ:
9              p' := FIND(δ(p, a))
10             q' := FIND(δ(q, a))
11             if p' ≠ q':
12                 S := PUSH(S, (p', q'))
```

```
13                UNION(p', q')
14        for  p ∈ Q :
15            p' := FIND(p)
16            if  ε̂(p) ≠ ε̂(p') :
21                return FALSE
22        return TRUE
```

Assuming that Q_1 and Q_2 are disjoint, DFA-EQUIVALENT-HK-P starts by creating a set for each state $q \in Q$ (lines 2–3). Two equivalence classes $[p]$ and $[q]$ are merged (lines 5 and 13) whenever it is discovered that, if D_1 is to be equivalent to D_2, the states p and q must also be equivalent. Every time two states are merged, the identifiers of the respective equivalence classes are selected (line 12), and for each $a \in \Sigma$, the sets containing the pair of successor states are also merged (line 13). When the point is reached where every pair of states in the same set has its successors pair in a single set (for all $a \in \Sigma$), the process terminates. The automata D_1 and D_2 are equivalent if and only if, at this point, no set contains both a final and a non-final state (lines 14–18).

If D_1 and D_2 are equivalent DFAs, DFA-EQUIVALENT-HK-P computes the finest right-invariant equivalence relation over Q that identifies the initial states, q_1 and q_2.

The associated set partition is built using the UNION-FIND algorithm for disjoint-set forests as described on Section 2.5.

9.2.3 Complexity analysis

Lemma 39. *Disregarding set operations, the worst-case running time of the procedure* DFA-EQUIVALENT-HK-P *is* $O(kn)$*, where* $k = |\Sigma|$ *and* $n = |Q_1| + |Q_2|$.

Proof. Lines 2–3 are executed exactly $n_1 + n_2$ times. The number of times that the **while** loop in the lines 8–14 is executed is limited by the total number of elements pushed to the stack S. Each time a pair of states is pushed into the stack, two sets are merged (lines 12–14) and thus the total number of sets is decreased by one. As initially there are $n_1 + n_2$ sets,

only $n_1 + n_2 - 1$ calls to the UNION instruction are possible and at most $n_1 + n_2 - 1$ pairs of states are placed in the stack. Since the main **while** loop is executed once for each symbol in the alphabet, disregarding the set operations, the total running time of the algorithm is $O(k(n_1 + n_2))$. □

Theorem 40. *Let $n = n_1 + n_2$ and $k = |\Sigma|$. The worst-case time complexity of* DFA-EQUIVALENT-HK-P *is $O(kn\alpha(kn, n))$.*

Proof. By Lemma 39 and Theorem 1, DFA-EQUIVALENT-HK-P presents a worst-case time complexity of $O(kn + m\alpha(m, n))$, where m is the number of FIND instructions. Since the number of FIND calls is bounded by $2nk + 1$ (one isolated call at line 7, and 2 calls in the **while** loop at lines 10 and 11) the running-time as a function of k and n is $O(kn + kn\alpha(kn, n)) = O(kn\alpha(kn, n))$.

Moreover, considering $\alpha(kn, n)$ constant for any conceivable application, we can view the running time as linear in kn. □

Theorem 41. *The upper bound from Theorem 40 is tight.*

Proof. We present on Table 9.1 an example of two DFAs, one with a single state and another with $n + 1$ states, such that the length of the shortest word which distinguishes the initial states is exactly n.

This shows that the upper bound proved on Theorem 40 may actually be achieved and, thus, cannot be reduced in the general case.

It also proves that the result by Conway [19, page 11] — on the length of the largest word required to distinguish two states of different DFAs— is tight. □

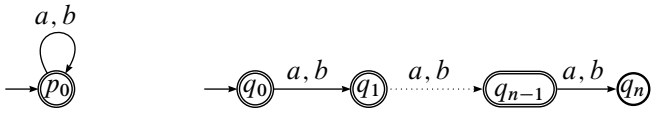


Figure 9.1: Two DFAs not distinguishable with a word of length smaller than n.

9.3 Improved best-case running time

By altering FIND so that the set being looked for is created if it does not exist, i.e., whenever FIND(x) fails MAKE(x) is called and the set $S_x = \{x\}$ is created[1], we may add a *refutation* procedure earlier in the algorithm. This allows the algorithm to return as soon as it finds a pair of distinguishable states, i.e., such that one is final and the other is not.

This alteration to the FIND procedure avoids the initialisation of $|Q_1| + |Q_2|$ sets which may actually never be used. Although it does not change the worst-case complexity, the overall best-case running-time is considerably improved, going from $\Omega(kn\alpha(kn, n))$ to $\Omega(1)$ — where k is the size of the alphabet and n is the sum of the number of states in both automata. This is because with the refutation condition, not only it is possible to distinguish the automata by the first pair of states, but it is also possible to avoid the linear check in the lines 14–18.

The increasingly high number of minimal ICDFAs observed by Almeida et al. [2] (cf. Appendix E), also suggests that, when dealing with random ICDFAs, the probability of having two equivalent automata is very low, and a refutation method will be very useful (for some experimental results, see Section 9.6).

We will now prove the correctness of DFA-EQUIVALENT-HKI-P. It is a variant of DFA-EQUIVALENT-HK-P, altered to accommodate the refutation method which improves the best-case running-time complexity of the algorithm.

```
1  def DFA-EQUIVALENT-HKI-P(D_1, D_2):
2      MAKE(p_0)
3      MAKE(q_0)
4      S := ∅
5      UNION(p_0, q_0)
6      S := PUSH(S, (p_0, q_0))
7      while (p, q) := POP(S):
8          if ε̂(p) ≠ ε̂(q):
```

[1] cf. the implementation details in Section 2.5

```
13            return FALSE
14        for a ∈ Σ:
15            p' := FIND(δ(p, a))
16            q' := FIND(δ(q, a))
17            if p' ≠ q':
18                UNION(p', q')
19                S := PUSH(S, (p', q'))
20    return TRUE
```

Definition 42 (Homogeneous set)**.** *A set of states A is* homogeneous *if and only if all its elements are either final or not-final, i.e., $\hat{\epsilon}(p) = \hat{\epsilon}(q)$ or $\hat{\epsilon}(p) \neq \hat{\epsilon}(q)$ for all $p, q \in A$.*

Lemma 43. *At line 8 of* DFA-EQUIVALENT-HK-P*, the sets S_i of the* UNION-FIND *structure are homogeneous if and only if all the pairs of states (p, q) already pushed into the stack S are such that $\hat{\epsilon}(p) = \hat{\epsilon}(q)$.*

Proof. Let us proceed by induction on the number of times l that line 8 is executed. For $l = 1$, it is trivial. Suppose now that all the sets are homogeneous up to the l^{th} time that line 8 is executed. If for all $a \in \Sigma$ the condition of line 11 is false, the homogeneous character of the sets will remain unaltered in the $(l + 1)^{th}$ run of the loop. Otherwise, it is clear that in lines 12–13, $S_{p'} \cup S_{q'}$ is homogeneous if and only if $\hat{\epsilon}(p) = \hat{\epsilon}(q)$. □

Theorem 44. *The procedures* DFA-EQUIVALENT-HK-P *and* DFA-EQUIVALENT-HKI-P *are equivalent.*

Proof. By Lemma 43, if there is a pair of states (p, q) pushed into the stack such that $\hat{\epsilon}(p) \neq \hat{\epsilon}(q)$, then the algorithm may terminate and return FALSE. That is exactly what DFA-EQUIVALENT-HKI-P implements at lines 8–9. □

Theorem 45. *The predicate* DFA-EQUIVALENT-HKI-P *returns* TRUE *if and only if D_1 and D_2 are equivalent.*

Proof. Directly from the proof of correctness of DFA-EQUIVALENT-HK-P (by Aho et al. [1, pages 143–145]) and Theorem 44. □

9.4 Generalisation to NFAs

We can embed the subset construction directly into DFA-EQUIVALENT-HKI-P, generalising the procedure so that it can be used to test the equivalence of non-deterministic finite automata.

NFA-EQUIVALENT-HKE-P does this by treating the set returned by each call to the transition function as a single state. It allows for an incremental construction of the equivalent deterministic automata, so that we may test the equivalence of the NFAs without actually computing the equivalent minimal automata.

This must be done with some caution, however, if we wish to avoid unnecessary combinatorial blowups in the number of states. Here, it is essential to use the idea described in Section 9.3 and implement the FIND procedure in a way such that a call to FIND(i) creates the set S_i if it does not exist. This way we avoid calling MAKE for each of the $2^{|Q|}$ sets — worst case of the subset construction — as some of those states may not be accessible in the equivalent DFA and thus, are not necessary.

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ be two NFAs. We assume that Q_1 and Q_2 disjoint, and we make $Q = Q_1 \cup Q_2$, $F = F_1 \cup F_2$, and $\Delta(P, a) = \Delta_i(P, a)$ for $P \subseteq Q_i, a \in \Sigma, i \in \{1, 2\}$.

```
1  def NFA-EQUIVALENT-HKE-P(N1, N2):
2      MAKE({q1})
3      MAKE({q2})
4      S := ∅
5      UNION({q1}, {q2})
6      S := PUSH(S, ({q1}, {q2}))
7      while (P1, P2) := POP(S):
8          if ε̂(P1) ≠ ε̂(P2):
13             return FALSE
14         for a ∈ Σ:
15             P1' := FIND(Δ(P1, a))
16             P2' := FIND(Δ(P2, a))
```

```
17              if P'_1 ≠ P'_2:
18                  UNION(P'_1, P'_2)
19                  S := PUSH(S, (P'_1, P'_2))
20      return TRUE
```

Lemma 46. NFA-EQUIVALENT-HKE-P *is a generalisation of* DFA-EQUIVALENT-HKI-P *which embeds the subset construction and thus, may be applied to NFAs.*

Proof. As DFA-EQUIVALENT-HKI-P has already been proven correct, it suffices to show that the elements P_1 and P_2 (popped from the stack S at line 7 of NFA-EQUIVALENT-HKE-P) are subsets of 2^Q which correspond to a single state in the associated DFA, just like in the subset construction method. The proof follows by induction on the number of operations on the stack S.

Base: The sets $\{ q_1 \}$ and $\{ q_2 \}$ are pushed onto the stack. These are the initial states of the DFAs equivalent to N_1 and N_2, respectively.

Induction: By induction hypothesis, we have that at the n^{th} call to POP(S), both P_1 and P_2 are subsets of Q which correspond to a single state in the deterministic automaton equivalent to N_1 or N_2 (denoted by D_1 and D_2, respectively). Without loss of generality, let us consider only P_1. Notice that, by definition, Δ corresponds to the transition function for the deterministic automaton in the subset construction method. Thus the call $\Delta(P_1, a)$ returns the subset of 2^Q reachable from P_1 by consuming the symbol a. This corresponds to the next "deterministic" state of either D_1 or D_2, and so NFA-EQUIVALENT-HKE-P is a variant of DFA-EQUIVALENT-HKI-P which embeds the subset construction. □

Theorem 47. NFA-EQUIVALENT-HKE-P *returns* TRUE *if and only if N_1 and N_2 are equivalent NFAs.*

Proof. It is a direct consequence of Theorem 44, which proves the correctness of DFA-EQUIVALENT-HKI-P, and Lemma 46. □

9.5 Relationship with Antimirov and Mosses' method

We presented, on Section 7.2, a method for testing the equivalence of two regular expressions based on a rewrite system due to Antimirov and Mosses.

Throughout this Section, we will establish a relationship between that approach to the regular expression equivalence problem and the previously described methods to test the equivalence of finite automata.

9.5.1 A naïve DFA-EQUIVALENT-HK-P

The procedure DFA-EQUIVALENT-HKN-P is a simplified, naïve version of DFA-EQUIVALENT-HK-P. It will be useful to establish a relationship with Antimirov and Mosses' approach to test regular expressions' equivalence. We will proceed by showing some important properties and proving its correctness.

Let $D_1 = (Q_1, \Sigma, q_1, \delta_1, F_1)$ and $D_2 = (Q_2, \Sigma, q_2, \delta_2, F_2)$ be DFAs, such that Q_1 and Q_2 are disjoint.

Definition 48. *We define the relation $\mathcal{R}$ as follows.*

$$\mathcal{R} = \left\{ (p, q) \in Q_1 \times Q_2 \mid \exists x \in \Sigma^\star : \hat{\delta}_1(q_1, x) = p \wedge \hat{\delta}_2(q_2, x) = q \right\}.$$

```
1  def DFA-EQUIVALENT-HKN-P(D1, D2):
2      S := {(q1, q2)}
3      H := ∅
4      while (p, q) := POP(S):
5          H := PUSH(H, (p, q))
6          for a ∈ Σ:
7              p' := δ1(p, a)
8              q' := δ2(q, a)
9              if (p', q') ∉ H:
10                 S := PUSH(S, (p', q'))
11     for (p, q) ∈ H:
```

```
12        if  ε̂(p) ≠ ε̂(q):
17            return FALSE
18    return TRUE
```

Lemma 49. *In* DFA-EQUIVALENT-HKN-P *no pair of states is ever removed from* H.

Proof. Obvious, as only PUSH operations are performed on H throughout the algorithm. □

Lemma 50. *In* DFA-EQUIVALENT-HKN-P*, the sets* S *and* H *are disjoint.*

Proof. Clearly, during the variables' initialisation at lines 2–3, the sets are disjoint as S contains one pair of states and H is the empty set.

Pairs of states are only pushed into H at line 5, but these are popped from S immediately before, at line 4.

Only on line 10 are any elements, say (p', q'), pushed into S, but this happens only if $(p', q') \notin H$. □

Lemma 51. *Let* $k = |\Sigma|$*,* $n_1 = |Q_1|$*, and* $n_2 = |Q_2|$*.* DFA-EQUIVALENT-HKN-P *is terminating in* $O(k\,n_1 n_2)$ *time.*

Proof. The elements of S are pairs of states (p, q), such that $p \in Q_1$ and $q \in Q_2$. This results in, at most, $n_1 n_2$ elements being pushed into S. The only PUSH operation on H — line 5 — is performed with elements popped from S and thus, H will also have at most $n_1 n_2$ elements. This assures termination.

For each element in S, lines 6–10 are executed once for each element of Σ. As the loop at lines 11–13 is executed at most $n_1 n_2$ times, this results in a running time complexity of $O(k\,n_1 n_2)$. □

Lemma 52. *In* DFA-EQUIVALENT-HKN-P*, for all* $(p, q) \in Q_1 \times Q_2$*, we have that*

$$(p, q) \in S \text{ at some step } k > 0 \;\Leftrightarrow\; (p, q) \in H \text{ at some step } k' > k.$$

Proof. By Lemma 50, we have that S and H are disjoint. It is obvious that if $(p, q) \in S$ in a step k of DFA-EQUIVALENT-HKN-P, then $(p, q) \in H$ for any $k' > k$. Simply observe that elements are only pushed into H after being popped from S — lines 4–5. For the same reason, if some element $(p, q) \in H$ at step k', it had to be in S at some step $k < k'$. □

Lemma 53. *For all* $(p, q) \in Q_1 \times Q_2$, $(p, q) \in S$ *at some step of* DFA-EQUIVALENT-HKN-P*, if and only if* $(p, q) \in \mathcal{R}$.

Proof. Let $(p, q) \in \mathcal{R}$, i.e., $\exists w : \hat{\delta}_1(q_1, w) = p \wedge \hat{\delta}_2(q_2, w) = q$. The proof follows by induction on the length of the word w.

Base: $\hat{\delta}_1(q_1, \epsilon) = q_1$, $\hat{\delta}_2(q_2, \epsilon) = q_2$, and $(q_1, q_2) \in S$ already at line 2.

Induction: Let $w = ua$ such that $a \in \Sigma$, $\hat{\delta}_1(q_1, u) = p$, and $\hat{\delta}_2(q_2, u) = q$. By induction hypothesis, we know that $(p, q) \in S$. On lines 7–8, $p' = \delta_1(p, a)$ and $q' = \delta_2(q, a)$ will be computed and added to S if $(p', q') \notin H$. When $(p', q') \in H$, however, we know by Lemma 52 that (p', q') was already placed in S at some previous step of the algorithm.

Conversely, and because new elements are only added to S on line 10, $(p, q) \in S$ only if there is some word w such that $\hat{\delta}_1(q_1, w) = p \wedge \hat{\delta}_2(q_2, w) = q$. □

Lemma 54. *At line 11 of* DFA-EQUIVALENT-HKN-P*, for all* $(p, q) \in Q_1 \times Q_2$, $(p, q) \in H$ *if and only if* $(p, q) \in \mathcal{R}$.

Proof.

$\Rightarrow$: If, at line 11 of DFA-EQUIVALENT-HKN-P, $(p, q) \in H$, by Lemma 52 and Lemma 53 we know that $(p, q) \in \mathcal{R}$.

$\Leftarrow$: Suppose $(p, q) \in \mathcal{R}$. We know by Lemma 53 that $(p, q) \in S$ at some step k of DFA-EQUIVALENT-HKN-P. By Lemma 52 there exists a step $k' > k$ such that $(p, q) \in H$. Because S is empty at line 11, this step has surely been matched. □

Theorem 55. $D_1 \sim D_2$ *if and only if*

$$\forall (p, q) \in \mathcal{R},\ \hat{\epsilon}(p) = \hat{\epsilon}(q).$$

Proof.

$\Rightarrow$: Suppose, by absurd, that $D_1 \not\sim D_2$ and that the condition holds. If $D_1 \not\sim D_2$, there exists $w \in \Sigma^\star$ such that $\hat{\delta}_1(q_1, w) = p'$, $\hat{\delta}_2(q_2, w) = q'$, and $\hat{\epsilon}(p) \neq \hat{\epsilon}(q)$. As clearly $(p', q') \in \mathcal{R}$, this is a contradiction.

$\Leftarrow$: If $\forall (p, q) \in \mathcal{R}, p \in F_1 \Leftrightarrow q \in F_2$, clearly there is no word $w \in \Sigma^\star$ such that $\hat{\epsilon}\big(\hat{\delta}_1(q_1, w)\big) \neq \hat{\epsilon}\big(\hat{\delta}_2(q_2, w)\big)$, therefore $D_1 \sim D_2$. □

Theorem 56. DFA-EQUIVALENT-HKN-P *returns* TRUE *if and only if* D_1 *is equivalent to* D_2.

Proof. Follows directly from Lemma 54 and Theorem 55. □

Corollary 57. DFA-EQUIVALENT-HKN-P *and* DFA-EQUIVALENT-HK-P *are equivalent.*

The relation $\mathcal{R}$ can be seen as a reflexive and symmetric relation on $Q_1 \cup Q_2$. Its transitive closure, $\mathcal{R}^\star$, is an equivalence relation.

Theorem 58. *For all* $(p, q) \in \mathcal{R}$

$$\hat{\epsilon}(q) = \hat{\epsilon}(q) \Leftrightarrow \forall (p', q') \in \mathcal{R}^\star,\ \hat{\epsilon}(p') = \hat{\epsilon}(q').$$

Proof. Let $(p, q), (q, r) \in \mathcal{R}$. Since $\mathcal{R}^\star$ is the transitive closure of $\mathcal{R}$, $(p, r) \in \mathcal{R}^\star$ and if $\hat{\epsilon}(p) = \hat{\epsilon}(q)$, then $\hat{\epsilon}(p) = \hat{\epsilon}(r)$. On the other hand, as $\mathcal{R} \subseteq \mathcal{R}^\star$, if $\hat{\epsilon}(p) = \hat{\epsilon}(q)\ \forall (p, q) \in \mathcal{R}^\star$, the same will be true for every $(p, q) \in \mathcal{R}$. □

Corollary 59. $D_1 \sim D_2$ *if and only if* $\forall (p, q) \in \mathcal{R}^\star,\ \hat{\epsilon}(p) = \hat{\epsilon}(q)$.

Proof. It is a direct consequence of Theorem 58. □

DFA-EQUIVALENT-HK-P computes $\mathcal{R}^\star$ by starting with the finest partition in $Q_1 \cup Q_2$ (the identity). Moreover, if $D_1 \sim D_2$, $\mathcal{R}^\star$ is a right-invariant relation.

9.5.2 Equivalence of the two methods

The function RE-EQUIVALENT-S-P implements a simplified version of the regular expressions' equivalence-testing method described on Section 7.2 — where further details about the notation, implementation, and comparison with the original rewrite system may be found.

```
1  def RE-EQUIVALENT-S-P(α, β):
2      S := {(α, β)}
3      H := ∅
4      while (α, β) := POP(S):
5          if ε̂(α) ≠ ε̂(β):
10             return FALSE
11         H := PUSH(H, (α, β))
12         for a ∈ Σ:
13             α' := a^{-1}(α)
14             β' := a^{-1}(β)
15             if (α', β') ∉ H:
16                 S := PUSH(S, (α', β'))
17     return TRUE
```

It is possible to use RE-EQUIVALENT-S-P to obtain a DFA from each of the regular expressions, α and β. Let $D_\alpha = (Q_\alpha, \Sigma, \delta_\alpha, q_\alpha, F_\alpha)$ and $D_\beta = (Q_\beta, \Sigma, \delta_\beta, q_\beta, F_\beta)$ be the DFAs equivalent to α e β, respectively. They are constructed in the following way:

- initialise $Q_\alpha = \{\alpha\}$, $Q_\beta = \{\beta\}$;
- $q_\alpha = \alpha$, $q_\beta = \beta$;
- for each instruction $\alpha' = a^{-1}(\alpha)$, add the transition $\delta_\alpha(\alpha, a) = \alpha'$ and make $Q_\alpha = Q_\alpha \cup \{\alpha'\}$ (same for β and β');
- within the loop on lines 4–12, whenever $\hat{\epsilon}(\alpha) = \epsilon$, make $F_\alpha = F_\alpha \cup \{\alpha\}$ (same for β).

The *Brzozowski's automata* of the regular expressions α and β are Q_α and Q_β, respectively.

We will now show that the regular expressions equivalence test RE-EQUIVALENT-S-P actually embeds Hopcroft and Karp's method while constructing the equivalent DFAs. In order to do so, we will first apply Theorem 44 to DFA-EQUIVALENT-HKN-P, transforming it into the refutation procedure DFA-EQUIVALENT-HKR-P.

```
1  def DFA-EQUIVALENT-HKR-P(D_1 := (Q_1, Σ, δ_1, q_1, F_1), D_2 := (Q_2, Σ, δ_2, q_2, F_2)):
2      S := {(q_1, q_2)}
3      H := ∅
4      while (p, q) := POP(S):
5          if ε̂(p) ≠ ε̂(q):
10             return FALSE
11         H := PUSH(H, (p, q))
12         for a ∈ Σ:
13             p' := δ_1(p, a)
14             q' := δ_2(q, a)
15             if (p', q') ∉ H:
16                 S := PUSH(S, (p', q'))
17     return TRUE
```

Lemma 60. RE-EQUIVALENT-S-P *embeds* DFA-EQUIVALENT-HKN-P *while constructing the Brzozowski's DFAs.*

Proof. By Theorem 44, DFA-EQUIVALENT-HKR-P is equivalent to DFA-EQUIVALENT-HKN-P, but includes a refutation step. To verify that RE-EQUIVALENT-S-P actually embeds DFA-EQUIVALENT-HKR-P while constructing the Brzozowski's DFAs, the following observations should be enough. The instructions

$$\alpha' = a^{-1}(\alpha) \text{ and } \beta' = a^{-1}(\beta)$$

from RE-EQUIVALENT-S-P are trivially equivalent to

$$p' = \delta_1(p, a) \text{ and } q' = \delta_2(q, a)$$

in DFA-EQUIVALENT-HKR-P, by the very definitions of regular expression derivative and the method which constructs the equivalent DFAs.

The halting conditions are also equivalent. Since $p \in F_\alpha$ if and only if $\hat{\epsilon}(\alpha) = \epsilon$, we know that $\hat{\epsilon}(\alpha') \neq \hat{\epsilon}(\beta')$ if and only if $\hat{\epsilon}(p') \neq \hat{\epsilon}(q')$ when we consider the DFAs associated to each of the regular expressions, where $p' \in Q_\alpha$, $q' \in Q_\beta$. □

Theorem 61. RE-EQUIVALENT-S-P *corresponds to* DFA-EQUIVALENT-HK-P *applied to the Brzozowski's automata of the two regular expressions.*

Proof. By Corollary 57, DFA-EQUIVALENT-HKN-P and DFA-EQUIVALENT-HK-P are equivalent. By Lemma 60 we have that RE-EQUIVALENT-S-P embeds DFA-EQUIVALENT-HKN-P while constructing the Brzozowski's DFAs. Thus, applying RE-EQUIVALENT-S-P to two regular expressions α and β is equivalent to applying DFA-EQUIVALENT-HK-P to the Brzozowski's automata of α and β. □

9.5.3 Worst-case complexity

We can use the equivalence between RE-EQUIVALENT-S-P and DFA-EQUIVALENT-HK-P— as established on Subsection 9.5.2, and, in particular, by Theorem 61 — to prove an upper bound on the running-time of RE-EQUIVALENT-PARTIAL-P. Recall that RE-EQUIVALENT-PARTIAL-P is a non-deterministic version of RE-EQUIVALENT-S-P which uses partial derivatives and that NFAs are related to partial derivatives in the same natural way as DFAs are related to derivatives.

We proceed by showing that the Brzozowski NFA N_α, obtained while applying RE-EQUIVALENT-PARTIAL-P to a regular expression α as described on Section 7.3, is such that $|N_\alpha| \in O(|\alpha|_\Sigma)$ and the number of states of the smallest equivalent DFA is $O\left(2^{|N_\alpha|}\right)$.

Figure 9.2 presents a classical example of a bad behaved case (with $n + 1$ states) of the subset construction, by Hopcroft et al. [35, pages 154–157]. Although this example does not reach the 2^{n+1} states bound, the smallest equivalent DFA has exactly 2^n states.

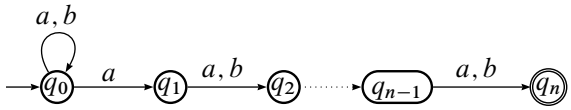


Figure 9.2: NFA that has no equivalent DFA with less than 2^n states.

Consider the regular expression family $\alpha_\ell = (a+b)^\star(a(a+b)^\ell)$, where $|\alpha_\ell|_\Sigma = 3 + 2\ell$. It is easy to see that the NFA in Figure 9.2 is obtained directly from the application of RE-EQUIVALENT-PARTIAL-P to α_ℓ. The corresponding partial derivatives are presented on Figure 9.3.

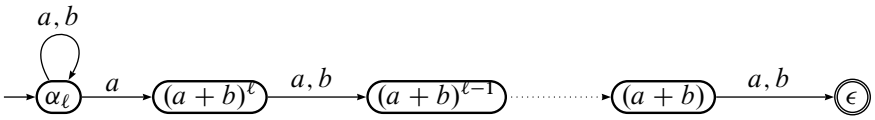


Figure 9.3: Brzozowski NFA obtained with RE-EQUIVALENT-PARTIAL-P.

The set of the partial derivatives $PD(\alpha_\ell) = \left\{\alpha_\ell, (a+b)^\ell, \ldots, (a+b), \epsilon\right\}$ has $\ell + 2 = n + 1$ elements, which corresponds to the size of the obtained NFA. The equivalent minimal DFA has $2^n = 2^{\ell+1}$ states.

9.6 Experimental results

This section contains some experimental comparative results of the previously described algorithms: DFA-EQUIVALENT-HK-P and NFA-EQUIVALENT-HKE-P. We also include results for the algorithm DFA-EQUIVALENT-HKS-P, a variant of NFA-EQUIVALENT-HKE-P which makes use of the string representation described on Subsection 3.1.1. The simplicity of the representation seems to be quite suitable for this type of algorithm, and actually exhibits quite good practical results turning out to be the fastest implementation. This is a good example of the impact that a good data structure may have on the overall performance of an algorithm.

Since the main goal of these benchmarks is to compare the performance of the the di-

rect equivalence tests DFA-EQUIVALENT-HK-P and NFA-EQUIVALENT-HKE-P with the classical approach, they include results for minimisation-based equivalence tests using algorithms by Moore, Brzozowski, and Hopcroft. The incremental DFA minimisation algorithm DFA-MINIMISE-INCREMENTAL, introduced on Subsection 8.3, is also considered mainly because it presents the best practical performance (cf. Section 8.4).

The conditions on which the experimental tests were conducted are described in detail on Chapter 5.

Here we include only a set of three-dimensional graphs. There are intended to be a bird's-eye view on the overall performance of the algorithms discussed previously on this chapter. We define *performance* as the number of finite automata tested for equivalence per second. Appendix D includes complete tables with the exact values of the running time, memory usage, average number of recursive calls, etc. for each algorithm.

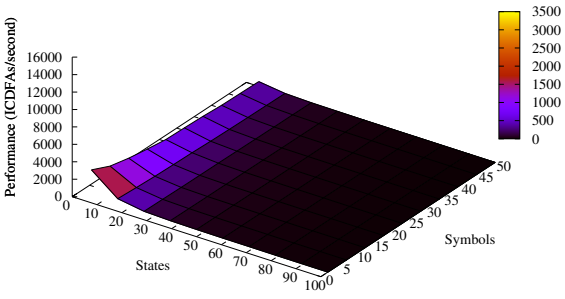
The rule of thumb for reading the graphics is as follows: a darker area means lower values, and therefore, poorer performance.

9.6.1 ICDFAs equivalence

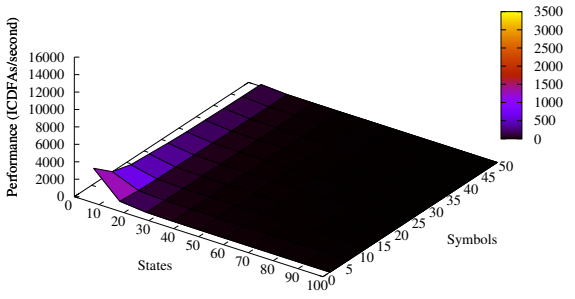
Clearly, the methods which do not rely in minimisation processes are *significantly* faster.

Considering the minimisation-based equivalence testing algorithms, DFA-MINIMISE-INCREMENTAL presents the best results (Figure 9.4c). It is closely followed by DFA-MINIMISE-MOORE, which still performs noticeably better than DFA-MINIMISE-HOPCROFT (Figure 9.4a and Figure 9.4b, respectively). This should come as no surprise, given the results from Section 8.4.

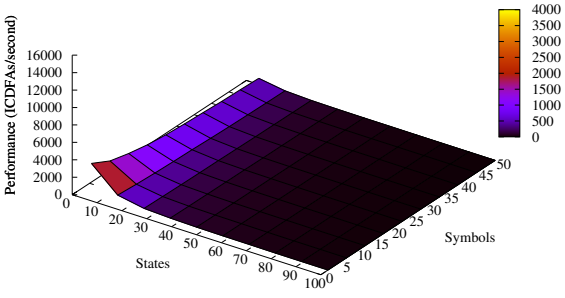
As for the methods that do not resort to the minimal DFA, DFA-EQUIVALENT-HKS-P (Figure 9.4f) is notoriously the fastest. Naturally, a consequence of the simple data structures involved. It is followed by NFA-EQUIVALENT-HKE-P (Figure 9.4e), our generalisation of DFA-EQUIVALENT-HK-P to NFAs which includes the refutation procedure.



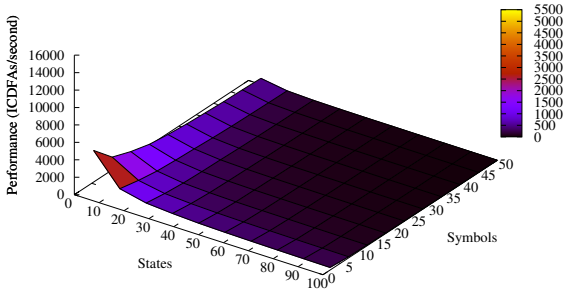
(a) DFA-MINIMISE-MOORE



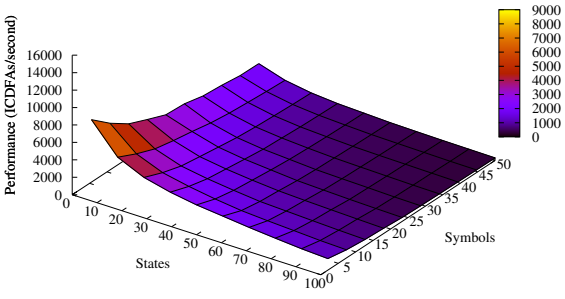
(b) DFA-MINIMISE-HOPCROFT



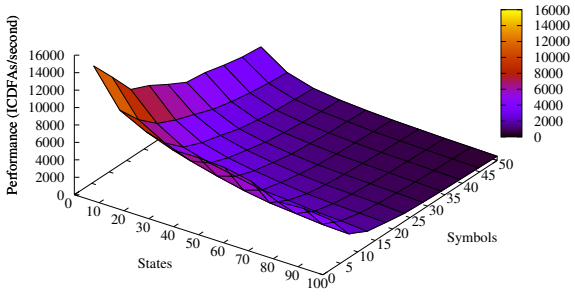
(c) DFA-MINIMISE-INCREMENTAL



(d) DFA-EQUIVALENT-HK-P



(e) NFA-EQUIVALENT-HKE-P



(f) DFA-EQUIVALENT-HKS-P

Figure 9.4: Performance graphics: ICDFAs' equivalence-testing algorithms.

As one can easily tell from the performance difference between DFA-EQUIVALENT-HK-P (Figure 9.4d) and its non-deterministic generalisation NFA-EQUIVALENT-HKE-P (Fig-

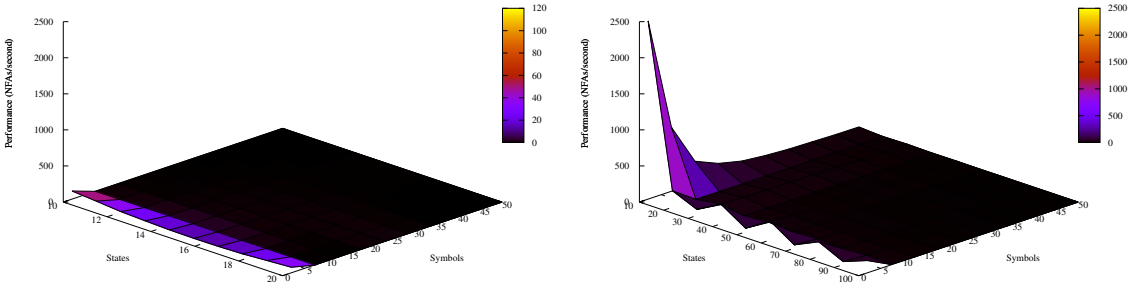
ure 9.4e), the refutation process is certainly an enhancement. This is somewhat natural since we have already observed [2] that, when dealing with uniformly generated random ICDFAs, the asymptotic probability of an automaton being minimal is nearly 1. Hence, the probability of finding two uniformly randomly generated equivalent ICDFAs is extremely low.

Some exact numbers on the amount of minimal ICDFAs with n states over an alphabet of k symbols, as well as some statistically significant probabilities are presented on Appendix E.

9.6.2 NFAs equivalence

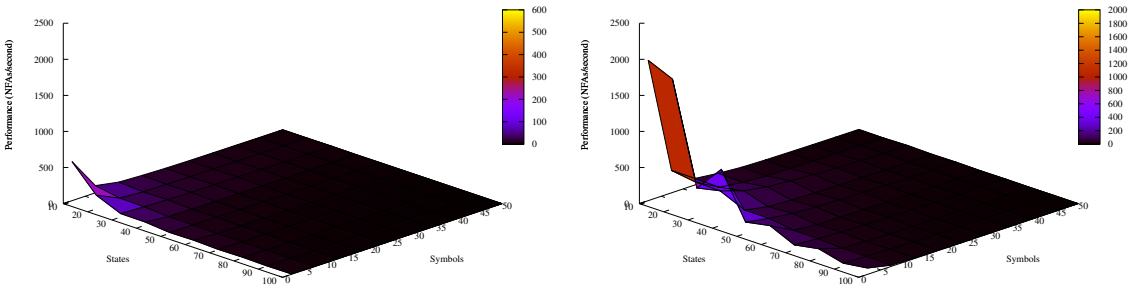
Since we already know, from the experimental results presented on Chapter 8, that when minimising NFAs Brzozowski's algorithm is the fastest we have available, here we present only experimental tests relating it to the new method NFA-EQUIVALENT-HKE-P, which does not resort to any minimisation process.

Clearly, NFA-EQUIVALENT-HKE-P performs faster than Brzozowski's algorithm, regardless of the transition density. Considering the experimental resuls on Section 8.4, Brzozowski's algorithm should perform better than the usual minimisation-based methods, which implies that this new direct comparison method is the fastest approach to test NFAs' equivalence.



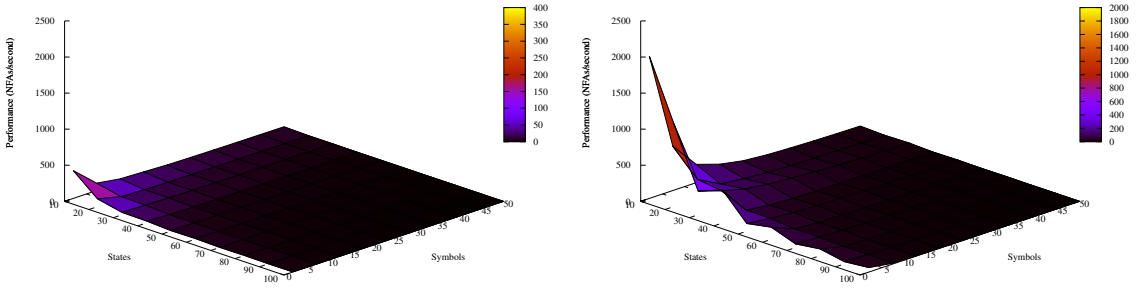
(a) DFA-MINIMISE-BRZOZOWSKI (b) NFA-EQUIVALENT-HKE-P

Figure 9.5: Performance graphics: NFAs' equivalence, with transition density $d = 0.1$.



(a) DFA-MINIMISE-BRZOZOWSKI (b) NFA-EQUIVALENT-HKE-P

Figure 9.6: Performance graphics: NFAs' equivalence, with transition density $d = 0.5$.



(a) DFA-MINIMISE-BRZOZOWSKI (b) NFA-EQUIVALENT-HKE-P

Figure 9.7: Performance graphics: NFAs' equivalence, with transition density $d = 0.8$.

Chapter 10

Conclusions

Using data sets of randomly generated automata (ICDFAs and NFAs), we have experimentally compared several automata minimisation algorithms, namely: Moore's, Hopcroft's, Watson's, an optimised version by Daciuk, Brzozowski's, and a new incremental quadratic algorithm. The ICDFAs' data set was obtained with a uniform random generator and is large enough to ensure a 99.5% confidence level and a 1% error margin. Not being uniform, the random generator of NFAs does not present the same statistical accuracy, but the results in this case are still interesting. With samples whose size spans from 10 to 1 000 states, we showed that Moore and Hopcroft's algorithms, although having different worst-case running-times, present similar results in the average case and both are outperformed by the new quadratic incremental algorithm. The algorithms due to Brzozowski and Watson, do reveal their exponential character and also present very similar results. Considering NFAs' minimisation, and using the ad-hoc random generator, we have showed that Brzozowski's algorithm is usually the fastest of the considered algorithms. Probably contrary to what one would expect, it is also relatively moderate in terms of memory usage.

We have presented several variants of a method by Aho et al. (from an original technical report by Hopcroft and Karp) to compare the language recognised by two DFAs without resorting to any minimisation processes. We extended this approach to NFAs and concluded that it does provide much more time-efficient methods for checking the equivalence of finite

automata. One of our modifications consists on placing a refutation condition earlier in the algorithm, which allows for better running times to be achieved in the average case. This is sustained by the experimental results presented in Appendix D. Using Brzozowski's automata, we also showed that a modified version of Antimirov and Mosses method translates directly to Hopcroft and Karp's algorithm.

Concerning the problem of testing the equivalence of regular expressions, we have presented a functional approach, based on a rewrite system by Antimirov and Mosses, that attempts to refute the equivalence by finding a pair of derivatives such that their constant parts are different. The experimental results, which are statistically significant up to a 95.7% confidence interval with a 1% error margin, point to a good average-case performance, not only for non-equivalent regular expressions — which would be expected, since it is a refutation procedure — but also when considering syntactically equal (and thus equivalent) regular expressions [3]. Extending this refutation procedure to partial derivatives, we took advantage of the simpler computational representation and added a memoization mechanism, which allowed for further performance gains. Our final improvement, in terms of implementation, consisted on replacing the naïve stack representation of the equivalence classes by a UNION-FIND data structure. This allowed to reduce the worst-case time complexity of searching a set with n elements from $O(\log n)$ to $O(1)$.

Given the spread of multi-core processors and grid computer systems, a parallel execution of the classic method and these direct comparison approaches can lead to an optimised framework for testing the equivalence of regular languages.

Finally, we presented a new incremental minimisation algorithm. Unlike other non-incremental minimisation algorithms, the intermediate results are usable and reduce the size of the input DFA. This property can be used to minimise a DFA when it is simultaneously processing a string or, for example, to reduce the size of a DFA when the running-time of the minimisation process must be restricted for some reason. This new approach, while presenting a quadratic worst-case running-time, is quite simple and easy to understand and to implement. According to the experimental results, this minimisation algorithm outperforms Hopcroft's $O(k\,n\log(n))$ approach and other incremental algorithms, at least

in the average case, for automata with 1 000 states or less.

10.1 Future work

Concerning the representation of regular expressions, future research work may be directed to the problem of eliminating redundant information. We have used the notion of "succinct regular expression", but, except for some approximate context-free grammars by Lee and Shallit [46, 65], there is no known formal study. Such a characterisation could be useful both in the implementation of the algorithms we have presented, and in symbolic manipulation frameworks in a general way. It could also lead to some regular expressions' simplification process, based, for example, on a rewrite system.

Future work may also consider the generalisation of the equivalence algorithms we have proposed to extended regular expressions — with intersection or complement, for example — or maybe even a specialisation of the algorithms, to test inclusion instead of equivalence.

We also believe that it is possible to improve of the finite automata equivalence-testing algorithms by using some ideas from Navarro's bit-parallel implementations [53, 55, 54], taking advantage of the L1 cache and the extremely fast bit-wise operations at the CPU registers' level.

All of these tasks could be integrated into the **FAdo** toolkit, further expanding this symbolic manipulation framework.

Appendix A

Number of non-isomorphic ICDFAs

The following tables present the number of ICDFAs (in scientific notation, due to the considerable size of the values) for a given number of states n over an alphabet of k symbols.

Table A.1: Number of ICDFAs with $\{2, \ldots, 8\}$ states.

k	n						
	2	3	4	5	6	7	8
2	4.80×10^{1}	1.73×10^{3}	8.40×10^{4}	5.14×10^{6}	3.80×10^{8}	3.28×10^{10}	3.24×10^{12}
3	2.24×10^{2}	6.37×10^{4}	3.40×10^{7}	2.93×10^{10}	3.69×10^{13}	6.41×10^{16}	1.47×10^{20}
4	9.60×10^{2}	1.94×10^{6}	1.03×10^{10}	1.12×10^{14}	2.19×10^{18}	6.97×10^{22}	3.37×10^{27}
5	3.97×10^{3}	5.52×10^{7}	2.81×10^{12}	3.78×10^{17}	1.12×10^{23}	6.34×10^{28}	6.31×10^{34}
6	1.61×10^{4}	1.52×10^{9}	7.37×10^{14}	1.22×10^{21}	5.38×10^{27}	5.42×10^{34}	1.10×10^{42}
7	6.50×10^{4}	4.15×10^{10}	1.91×10^{17}	3.85×10^{24}	2.54×10^{32}	4.53×10^{40}	1.88×10^{49}
8	2.61×10^{5}	1.13×10^{12}	4.90×10^{19}	1.21×10^{28}	1.19×10^{37}	3.75×10^{46}	3.18×10^{56}
9	1.05×10^{6}	3.05×10^{13}	1.26×10^{22}	3.78×10^{31}	5.58×10^{41}	3.09×10^{52}	5.34×10^{63}
10	4.19×10^{6}	8.23×10^{14}	3.22×10^{24}	1.18×10^{35}	2.61×10^{46}	2.55×10^{58}	8.97×10^{70}
11	1.68×10^{7}	2.22×10^{16}	8.25×10^{26}	3.70×10^{38}	1.22×10^{51}	2.10×10^{64}	1.51×10^{78}
12	6.71×10^{7}	6.00×10^{17}	2.11×10^{29}	1.16×10^{42}	5.67×10^{55}	1.73×10^{70}	2.53×10^{85}
13	2.68×10^{8}	1.62×10^{19}	5.41×10^{31}	3.61×10^{45}	2.65×10^{60}	1.42×10^{76}	4.24×10^{92}
14	1.07×10^{9}	4.38×10^{20}	1.38×10^{34}	1.13×10^{49}	1.24×10^{65}	1.17×10^{82}	7.11×10^{99}
15	4.29×10^{9}	1.18×10^{22}	3.54×10^{36}	3.53×10^{52}	5.76×10^{69}	9.66×10^{87}	1.19×10^{107}
16	1.72×10^{10}	3.19×10^{23}	9.07×10^{38}	1.10×10^{56}	2.69×10^{74}	7.96×10^{93}	2.00×10^{114}
17	6.87×10^{10}	8.61×10^{24}	2.32×10^{41}	3.45×10^{59}	1.25×10^{79}	6.55×10^{99}	3.36×10^{121}
18	2.75×10^{11}	2.33×10^{26}	5.95×10^{43}	1.08×10^{63}	5.85×10^{83}	5.40×10^{105}	5.63×10^{128}
19	1.10×10^{12}	6.28×10^{27}	1.52×10^{46}	3.37×10^{66}	2.73×10^{88}	4.45×10^{111}	9.45×10^{135}
20	4.40×10^{12}	1.70×10^{29}	3.90×10^{48}	1.05×10^{70}	1.27×10^{93}	3.66×10^{117}	1.59×10^{143}

continued on next page

continued from previous page

k	n						
	2	3	4	5	6	7	8
21	1.76×10^{13}	4.58×10^{30}	9.98×10^{50}	3.29×10^{73}	5.94×10^{97}	3.02×10^{123}	2.66×10^{150}
22	7.04×10^{13}	1.24×10^{32}	2.55×10^{53}	1.03×10^{77}	2.77×10^{102}	2.48×10^{129}	4.46×10^{157}
23	2.81×10^{14}	3.34×10^{33}	6.54×10^{55}	3.21×10^{80}	1.29×10^{107}	2.04×10^{135}	7.49×10^{164}
24	1.13×10^{15}	9.01×10^{34}	1.67×10^{58}	1.00×10^{84}	6.04×10^{111}	1.68×10^{141}	1.26×10^{172}
25	4.50×10^{15}	2.43×10^{36}	4.29×10^{60}	3.13×10^{87}	2.82×10^{116}	1.39×10^{147}	2.11×10^{179}
26	1.80×10^{16}	6.57×10^{37}	1.10×10^{63}	9.80×10^{90}	1.31×10^{121}	1.14×10^{153}	3.54×10^{186}
27	7.21×10^{16}	1.77×10^{39}	2.81×10^{65}	3.06×10^{94}	6.13×10^{125}	9.41×10^{158}	5.93×10^{193}
28	2.88×10^{17}	4.79×10^{40}	7.19×10^{67}	9.57×10^{97}	2.86×10^{130}	7.75×10^{164}	9.95×10^{200}
29	1.15×10^{18}	1.29×10^{42}	1.84×10^{70}	2.99×10^{101}	1.33×10^{135}	6.38×10^{170}	1.67×10^{208}
30	4.61×10^{18}	3.49×10^{43}	4.71×10^{72}	9.34×10^{104}	6.23×10^{139}	5.25×10^{176}	2.80×10^{215}
31	1.84×10^{19}	9.43×10^{44}	1.21×10^{75}	2.92×10^{108}	2.90×10^{144}	4.33×10^{182}	4.70×10^{222}
32	7.38×10^{19}	2.55×10^{46}	3.09×10^{77}	9.12×10^{111}	1.36×10^{149}	3.56×10^{188}	7.89×10^{229}
33	2.95×10^{20}	6.87×10^{47}	7.90×10^{79}	2.85×10^{115}	6.32×10^{153}	2.93×10^{194}	1.32×10^{237}
34	1.18×10^{21}	1.86×10^{49}	2.02×10^{82}	8.91×10^{118}	2.95×10^{158}	2.42×10^{200}	2.22×10^{244}
35	4.72×10^{21}	5.01×10^{50}	5.18×10^{84}	2.78×10^{122}	1.38×10^{163}	1.99×10^{206}	3.72×10^{251}
36	1.89×10^{22}	1.35×10^{52}	1.33×10^{87}	8.70×10^{125}	6.42×10^{167}	1.64×10^{212}	6.25×10^{258}
37	7.56×10^{22}	3.65×10^{53}	3.40×10^{89}	2.72×10^{129}	3.00×10^{172}	1.35×10^{218}	1.05×10^{266}
38	3.02×10^{23}	9.86×10^{54}	8.69×10^{91}	8.50×10^{132}	1.40×10^{177}	1.11×10^{224}	1.76×10^{273}
39	1.21×10^{24}	2.66×10^{56}	2.22×10^{94}	2.66×10^{136}	6.52×10^{181}	9.15×10^{229}	2.95×10^{280}
40	4.84×10^{24}	7.19×10^{57}	5.70×10^{96}	8.30×10^{139}	3.04×10^{186}	7.54×10^{235}	4.95×10^{287}
41	1.93×10^{25}	1.94×10^{59}	1.46×10^{99}	2.59×10^{143}	1.42×10^{191}	6.21×10^{241}	8.30×10^{294}
42	7.74×10^{25}	5.24×10^{60}	3.73×10^{101}	8.10×10^{146}	6.62×10^{195}	5.11×10^{247}	1.39×10^{302}
43	3.09×10^{26}	1.41×10^{62}	9.56×10^{103}	2.53×10^{150}	3.09×10^{200}	4.21×10^{253}	2.34×10^{309}
44	1.24×10^{27}	3.82×10^{63}	2.45×10^{106}	7.91×10^{153}	1.44×10^{205}	3.47×10^{259}	3.92×10^{316}
45	4.95×10^{27}	1.03×10^{65}	6.26×10^{108}	2.47×10^{157}	6.73×10^{209}	2.86×10^{265}	6.58×10^{323}
46	1.98×10^{28}	2.78×10^{66}	1.60×10^{111}	7.73×10^{160}	3.14×10^{214}	2.35×10^{271}	1.10×10^{331}
47	7.92×10^{28}	7.52×10^{67}	4.10×10^{113}	2.41×10^{164}	1.46×10^{219}	1.94×10^{277}	1.85×10^{338}
48	3.17×10^{29}	2.03×10^{69}	1.05×10^{116}	7.55×10^{167}	6.83×10^{223}	1.60×10^{283}	3.11×10^{345}
49	1.27×10^{30}	5.48×10^{70}	2.69×10^{118}	2.36×10^{171}	3.19×10^{228}	1.31×10^{289}	5.21×10^{352}
50	5.07×10^{30}	1.48×10^{72}	6.89×10^{120}	7.37×10^{174}	1.49×10^{233}	1.08×10^{295}	8.74×10^{359}

Table A.2: Number of ICDFAs with $\{9, \ldots, 15\}$ states.

k	n						
	9	10	11	12	13	14	15
2	3.61×10^{14}	4.47×10^{16}	6.08×10^{18}	9.04×10^{20}	1.45×10^{23}	2.52×10^{25}	4.68×10^{27}
3	4.28×10^{23}	1.55×10^{27}	6.80×10^{30}	3.57×10^{34}	2.21×10^{38}	1.59×10^{42}	1.31×10^{46}
4	2.35×10^{32}	2.27×10^{37}	2.96×10^{42}	5.03×10^{47}	1.10×10^{53}	3.00×10^{58}	1.01×10^{64}
5	1.03×10^{41}	2.60×10^{47}	9.77×10^{53}	5.26×10^{60}	3.94×10^{67}	4.00×10^{74}	5.40×10^{81}
6	4.17×10^{49}	2.74×10^{57}	2.94×10^{65}	4.97×10^{73}	1.27×10^{82}	4.76×10^{90}	2.54×10^{99}
7	1.65×10^{58}	2.79×10^{67}	8.57×10^{76}	4.53×10^{86}	3.94×10^{96}	5.42×10^{106}	1.14×10^{117}
8	6.42×10^{66}	2.81×10^{77}	2.47×10^{88}	4.08×10^{99}	1.20×10^{111}	6.08×10^{122}	5.05×10^{134}
9	2.49×10^{75}	2.82×10^{87}	7.06×10^{99}	3.65×10^{112}	3.66×10^{125}	6.78×10^{138}	2.22×10^{152}
10	9.67×10^{83}	2.82×10^{97}	2.02×10^{111}	3.26×10^{125}	1.11×10^{140}	7.55×10^{154}	9.74×10^{169}
11	3.75×10^{92}	2.82×10^{107}	5.75×10^{122}	2.90×10^{138}	3.36×10^{154}	8.39×10^{170}	4.26×10^{187}
12	1.45×10^{101}	2.82×10^{117}	1.64×10^{134}	2.59×10^{151}	1.02×10^{169}	9.32×10^{186}	1.87×10^{205}

continued on next page

continued from previous page

k	n						
	9	10	11	12	13	14	15
13	5.62×10^{109}	2.82×10^{127}	4.68×10^{145}	2.31×10^{164}	3.09×10^{183}	1.04×10^{203}	8.18×10^{222}
14	2.18×10^{118}	2.82×10^{137}	1.34×10^{157}	2.06×10^{177}	9.35×10^{197}	1.15×10^{219}	3.58×10^{240}
15	8.44×10^{126}	2.82×10^{147}	3.81×10^{168}	1.84×10^{190}	2.83×10^{212}	1.28×10^{235}	1.57×10^{258}
16	3.27×10^{135}	2.82×10^{157}	1.09×10^{180}	1.64×10^{203}	8.57×10^{226}	1.42×10^{251}	6.87×10^{275}
17	1.27×10^{144}	2.82×10^{167}	3.10×10^{191}	1.46×10^{216}	2.60×10^{241}	1.58×10^{267}	3.01×10^{293}
18	4.91×10^{152}	2.82×10^{177}	8.86×10^{202}	1.30×10^{229}	7.87×10^{255}	1.76×10^{283}	1.32×10^{311}
19	1.90×10^{161}	2.82×10^{187}	2.53×10^{214}	1.16×10^{242}	2.38×10^{270}	1.95×10^{299}	5.77×10^{328}
20	7.37×10^{169}	2.82×10^{197}	7.21×10^{225}	1.03×10^{255}	7.22×10^{284}	2.17×10^{315}	2.53×10^{346}
21	2.85×10^{178}	2.82×10^{207}	2.06×10^{237}	9.22×10^{267}	2.19×10^{299}	2.41×10^{331}	1.11×10^{364}
22	1.11×10^{187}	2.82×10^{217}	5.87×10^{248}	8.22×10^{280}	6.62×10^{313}	2.68×10^{347}	4.84×10^{381}
23	4.28×10^{195}	2.82×10^{227}	1.67×10^{260}	7.33×10^{293}	2.01×10^{328}	2.97×10^{363}	2.12×10^{399}
24	1.66×10^{204}	2.82×10^{237}	4.78×10^{271}	6.54×10^{306}	6.07×10^{342}	3.30×10^{379}	9.29×10^{416}
25	6.43×10^{212}	2.82×10^{247}	1.36×10^{283}	5.83×10^{319}	1.84×10^{357}	3.67×10^{395}	4.07×10^{434}
26	2.49×10^{221}	2.82×10^{257}	3.89×10^{294}	5.20×10^{332}	5.57×10^{371}	4.08×10^{411}	1.78×10^{452}
27	9.65×10^{229}	2.82×10^{267}	1.11×10^{306}	4.63×10^{345}	1.69×10^{386}	4.53×10^{427}	7.80×10^{469}
28	3.74×10^{238}	2.82×10^{277}	3.17×10^{317}	4.13×10^{358}	5.11×10^{400}	5.04×10^{443}	3.41×10^{487}
29	1.45×10^{247}	2.82×10^{287}	9.03×10^{328}	3.68×10^{371}	1.55×10^{415}	5.60×10^{459}	1.50×10^{505}
30	5.61×10^{255}	2.82×10^{297}	2.58×10^{340}	3.28×10^{384}	4.69×10^{429}	6.22×10^{475}	6.55×10^{522}
31	2.17×10^{264}	2.82×10^{307}	7.35×10^{351}	2.93×10^{397}	1.42×10^{444}	6.91×10^{491}	2.87×10^{540}
32	8.43×10^{272}	2.82×10^{317}	2.10×10^{363}	2.61×10^{410}	4.30×10^{458}	7.68×10^{507}	1.25×10^{558}
33	3.26×10^{281}	2.82×10^{327}	5.99×10^{374}	2.33×10^{423}	1.30×10^{473}	8.54×10^{523}	5.50×10^{575}
34	1.26×10^{290}	2.82×10^{337}	1.71×10^{386}	2.08×10^{436}	3.94×10^{487}	9.49×10^{539}	2.41×10^{593}
35	4.90×10^{298}	2.82×10^{347}	4.87×10^{397}	1.85×10^{449}	1.19×10^{502}	1.05×10^{556}	1.05×10^{611}
36	1.90×10^{307}	2.82×10^{357}	1.39×10^{409}	1.65×10^{462}	3.62×10^{516}	1.17×10^{572}	4.62×10^{628}
37	7.35×10^{315}	2.82×10^{367}	3.97×10^{420}	1.47×10^{475}	1.10×10^{531}	1.30×10^{588}	2.02×10^{646}
38	2.85×10^{324}	2.82×10^{377}	1.13×10^{432}	1.31×10^{488}	3.32×10^{545}	1.45×10^{604}	8.85×10^{663}
39	1.10×10^{333}	2.82×10^{387}	3.23×10^{443}	1.17×10^{501}	1.00×10^{560}	1.61×10^{620}	3.88×10^{681}
40	4.28×10^{341}	2.82×10^{397}	9.21×10^{454}	1.04×10^{514}	3.05×10^{574}	1.79×10^{636}	1.70×10^{699}
41	1.66×10^{350}	2.82×10^{407}	2.63×10^{466}	9.30×10^{526}	9.22×10^{588}	1.98×10^{652}	7.43×10^{716}
42	6.42×10^{358}	2.82×10^{417}	7.50×10^{477}	8.29×10^{539}	2.79×10^{603}	2.21×10^{668}	3.25×10^{734}
43	2.49×10^{367}	2.82×10^{427}	2.14×10^{489}	7.39×10^{552}	8.46×10^{617}	2.45×10^{684}	1.43×10^{752}
44	9.63×10^{375}	2.82×10^{437}	6.10×10^{500}	6.59×10^{565}	2.56×10^{632}	2.72×10^{700}	6.24×10^{769}
45	3.73×10^{384}	2.82×10^{447}	1.74×10^{512}	5.88×10^{578}	7.76×10^{646}	3.03×10^{716}	2.73×10^{787}
46	1.45×10^{393}	2.82×10^{457}	4.97×10^{523}	5.24×10^{591}	2.35×10^{661}	3.36×10^{732}	1.20×10^{805}
47	5.60×10^{401}	2.82×10^{467}	1.42×10^{535}	4.67×10^{604}	7.12×10^{675}	3.74×10^{748}	5.24×10^{822}
48	2.17×10^{410}	2.82×10^{477}	4.04×10^{546}	4.16×10^{617}	2.16×10^{690}	4.15×10^{764}	2.29×10^{840}
49	8.41×10^{418}	2.82×10^{487}	1.15×10^{558}	3.71×10^{630}	6.53×10^{704}	4.61×10^{780}	1.00×10^{858}
50	3.26×10^{427}	2.82×10^{497}	3.29×10^{569}	3.31×10^{643}	1.98×10^{719}	5.13×10^{796}	4.40×10^{875}

Table A.3: Number of ICDFAs with $\{18, 20, 25, 50, 75, 100, 1000\}$ states.

k	n						
	18	20	25	50	75	100	1000
2	4.34×10^{34}	2.52×10^{39}	4.64×10^{51}	1.81×10^{118}	2.91×10^{190}	2.16×10^{266}	3.70×10^{3658}
3	1.58×10^{58}	3.13×10^{66}	9.03×10^{87}	7.66×10^{205}	1.29×10^{335}	4.92×10^{471}	1.45×10^{6712}
5	6.07×10^{103}	9.41×10^{118}	2.49×10^{158}	7.17×10^{376}	9.48×10^{617}	6.75×10^{873}	2.71×10^{12733}

Appendix B

Equivalence of regular expressions

The following tables present experimental comparative results of several regular expression equivalence-testing algorithms (cf. Chapter 7). The sampling and experimental study were conducted as described in Chapter 5, using datasets with 20 000 pairs of regular expressions of size n over an alphabet of k symbols.

The full names of the algorithms are not used for matters of space economy. Instead, the following mapping applies.

Algorithm Name	Column Id.
DFA-MINIMISE-HOPCROFT	**H**
RE-EQUIVALENT-P	**E**
RE-EQUIVALENT-PARTIAL-P	$\mathbf{E}_{\mathrm{p}}$
RE-EQUIVALENT-UNION-FIND-P	$\mathbf{E}_{\mathrm{UF}}$

Column *Perf.* refers to the performance of the considered algorithm, i.e., the number of pairs of regular expressions tested for equivalence per second. The memory usage, in *kilobytes*, is shown on column *Sp.*, and column *Iter.* presents the average number of recursive calls (meaningful only on some algorithms) necessary to decide the equivalence of two regular expressions.

Table B.1: Benchmarks of regular expressions equivalence-testing algorithms (size 10).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	415.8	4	1506.0	4	27493	2262.4	4	27497	1941.7	4	28527
3	339.2	4	1298.7	136	36604	1968.5	136	37043	1709.4	4	37206
4	256.7	136	1189.0	136	46191	1715.2	136	46795	1538.4	4	45557
5	220.5	4	1113.5	4	53267	1605.1	4	54576	1497.0	4	51603
6	203.0	136	1061.5	4	58813	1569.8	4	60270	1412.4	4	56391
7	183.7	4	1030.9	136	65584	1479.2	4	67036	1385.0	4	61195
8	170.0	136	1008.0	136	71667	1424.5	4	73188	1394.7	136	64740
9	168.7	136	1017.2	4	71764	1424.5	4	73346	1369.8	4	65060
10	158.7	136	998.0	136	77162	1364.2	136	78974	1392.7	4	67277
11	150.9	136	944.2	4	83340	1317.5	4	85512	1322.7	4	71122
12	147.2	136	946.9	136	86743	1298.7	136	88893	1324.5	136	72826
13	136.9	136	946.9	136	89727	1265.8	136	91720	1290.3	136	74561
14	134.3	136	919.1	136	93170	1256.2	136	94996	1272.2	136	77337
15	126.5	136	923.3	136	96885	1216.5	4	99050	1257.8	136	79093
16	126.4	136	881.0	136	100809	1193.3	4	103032	1256.2	136	80374
18	125.6	136	893.6	136	103297	1186.2	136	105128	1248.4	136	81481
20	109.7	4	723.0	136	158634	932.8	4	160049	1142.8	136	105742
25	105.0	136	755.2	136	153052	968.0	136	154316	1146.7	136	103892
30	101.7	136	742.9	136	157430	954.1	136	158499	1153.4	136	105141
35	98.7	4	736.9	136	160976	950.5	4	161774	1136.3	136	106638
40	99.3	136	703.7	4	163084	945.1	4	163964	1136.3	136	107396
45	95.1	136	707.2	4	164752	938.9	4	165329	1133.7	4	107634
50	93.2	4	701.7	136	166674	905.7	4	167288	1133.7	4	108818

Table B.2: Benchmarks of regular expressions equivalence-testing algorithms (size 20).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	187.7	136	923.3	136	30260	1303.7	136	30530	1131.2	136	33053
3	133.2	4	780.6	136	43164	1144.1	136	43696	1021.4	136	44968
4	113.8	136	738.5	136	54049	1096.4	136	54934	988.1	136	56010
5	86.2	4	713.7	136	62132	1028.8	136	63493	951.4	136	64510
6	77.7	136	699.3	136	70530	992.0	136	72383	909.0	136	73248
7	63.0	268	671.5	136	77658	964.3	136	79379	859.1	136	79622
8	55.8	136	673.8	4	84839	931.0	136	86811	866.5	136	85404
9	55.8	268	661.3	136	84870	939.8	4	87331	859.1	136	85070
10	50.5	268	669.7	4	90216	908.2	4	92689	851.7	4	91223
11	45.4	268	632.9	4	99026	871.8	4	101550	813.6	4	96373
12	42.1	136	609.3	136	105342	862.8	136	107984	804.5	136	100756
13	39.5	136	618.0	136	112036	822.3	136	115000	788.6	136	103416
14	38.9	136	606.4	4	116829	800.0	4	120168	775.7	136	108044
15	37.2	136	598.8	4	121828	786.1	4	124932	773.3	4	110656
16	34.9	136	574.3	4	129221	761.0	4	132623	761.6	4	114640
18	33.8	268	562.4	136	133536	739.6	136	136978	758.1	136	117470
20	25.5	268	341.2	4	306478	416.1	4	310050	568.8	4	192400
25	23.6	268	363.2	4	284431	449.6	4	287057	599.5	4	184471

continued on next page

continued from previous page

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
30	22.4	268	347.3	136	297664	443.4	136	300066	586.5	136	189764
35	20.3	268	346.1	4	307636	435.5	4	309935	578.7	4	193046
40	19.0	136	336.2	4	314053	429.1	4	316003	578.3	4	195174
45	18.2	136	341.2	4	320598	413.3	4	322276	575.7	4	198266
50	17.8	136	333.1	136	325227	423.9	136	326866	571.1	136	199476

Table B.3: Benchmarks of regular expressions equivalence-testing algorithms (size 30).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	105.8	136	646.8	136	32392	939.8	268	32752	728.8	276	36789
3	77.5	136	574.7	136	48475	838.2	136	49206	736.3	136	52092
4	60.7	144	545.2	136	62080	802.5	136	63215	714.7	136	65090
5	47.9	136	544.9	136	71203	779.4	4	73519	703.2	4	74950
6	39.1	136	525.2	136	79546	765.1	140	82398	682.1	4	84644
7	33.3	268	512.5	4	88801	737.4	140	91674	663.1	136	93912
8	28.0	136	511.5	136	97662	713.7	136	100669	632.1	136	102670
9	28.8	268	524.6	4	95379	721.5	4	98325	642.6	4	101879
10	24.9	268	510.7	4	103796	701.2	4	107266	627.3	4	108204
11	22.0	268	490.6	136	112216	670.6	136	115671	627.7	136	113313
12	19.8	268	489.7	136	116399	668.8	136	119635	606.0	136	119809
13	18.4	268	497.5	4	121866	652.7	136	125359	606.0	136	121586
14	16.7	268	486.1	136	129922	639.3	136	133561	588.2	136	129712
15	15.7	268	477.5	136	138363	619.1	136	141857	576.7	136	134555
16	14.6	268	467.7	4	143172	595.2	4	146655	580.3	136	137617
18	14.0	268	427.1	4	153169	575.7	4	157161	564.3	4	144147
20	10.6	136	211.9	4	440890	260.0	4	447077	370.0	144	275080
25	9.8	276	235.6	4	392870	299.8	4	396928	392.6	136	257845
30	8.6	400	228.5	4	417110	286.2	136	420633	385.0	136	266415
35	7.9	532	223.3	4	436907	269.4	4	440108	377.9	136	273854
40	7.2	532	212.7	136	452818	265.4	136	455764	372.4	136	279299
45	6.9	532	215.2	136	465789	262.8	4	468490	367.1	136	282890
50	6.5	532	206.3	4	474868	258.5	4	477032	364.6	136	287040

Table B.4: Benchmarks of regular expressions equivalence-testing algorithms (size 40).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	56.3	272	510.2	268	33122	736.9	136	33620	514.1	400	40112
3	48.5	136	450.0	268	51533	679.8	268	52112	572.7	272	56909
4	38.3	136	447.2	140	65456	641.4	144	67465	564.9	144	74268
5	30.6	136	426.4	136	79241	610.5	136	81627	543.7	136	87332
6	24.9	268	433.6	4	88728	605.6	140	91325	527.9	272	99669
7	20.6	268	428.6	4	98035	596.3	140	100976	533.6	144	107131
8	17.8	400	424.6	136	106156	582.7	144	109831	524.3	140	114584
9	18.0	408	421.2	4	105402	583.4	4	109183	522.4	140	115571

continued on next page

continued from previous page

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
10	15.3	268	411.6	140	113995	569.4	140	117927	513.8	140	123869
11	13.5	268	405.3	136	124790	555.5	136	129211	515.7	136	128701
12	11.9	404	398.4	136	132139	541.4	136	136313	503.5	136	135826
13	10.9	400	398.8	136	137076	530.5	136	141105	488.7	136	141750
14	10.1	268	394.3	140	144379	519.7	136	148436	469.0	144	147552
15	9.1	268	387.7	136	149824	504.7	144	154824	471.0	136	153031
16	8.5	268	382.9	4	157967	502.5	136	162246	473.2	140	157735
18	8.0	400	370.2	136	164778	476.6	4	169147	465.3	136	161567
20	5.8	408	155.1	136	557763	190.7	136	565253	269.1	136	350799
25	5.2	400	178.4	144	485241	214.3	136	490537	285.6	136	324658
30	4.5	804	162.7	136	521705	201.0	136	527449	288.6	136	338162
35	3.9	664	164.4	4	545654	198.8	144	549925	274.4	144	347213
40	3.5	664	151.1	136	577612	184.0	140	581640	267.0	136	356443
45	3.3	664	153.9	4	595833	185.8	136	599450	273.6	136	363596
50	3.2	796	150.9	136	605413	184.8	136	608612	270.5	136	369192

Table B.5: Benchmarks of regular expressions equivalence-testing algorithms (size 50).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	34.8	404	421.9	272	34653	611.2	536	34886	369.5	932	45880
3	32.5	276	383.5	408	54274	560.8	412	55061	455.1	352	63217
4	27.1	136	358.0	400	71157	537.6	276	73233	466.4	408	80914
5	21.7	268	370.9	144	84247	536.4	136	86692	480.0	136	94957
6	17.4	268	373.6	136	93186	524.6	272	96896	469.7	136	108296
7	14.1	268	360.1	136	104423	509.4	136	109190	454.1	144	121106
8	12.0	408	351.0	136	116286	498.0	136	120274	440.1	144	132018
9	12.1	404	364.4	136	114407	501.7	136	118369	447.6	136	129568
10	10.5	268	350.5	136	123145	489.9	144	127608	439.9	140	138706
11	9.0	400	352.7	136	131942	478.4	136	135733	435.9	140	144509
12	8.0	400	332.6	136	142616	461.6	140	148498	427.8	144	152740
13	7.1	400	335.5	136	150314	450.8	268	156005	420.1	272	158903
14	6.5	400	347.2	136	157049	439.5	136	161886	414.4	268	163768
15	6.0	268	335.6	136	163795	432.5	136	169088	413.0	136	168019
16	5.4	400	321.3	136	175190	419.1	136	180793	404.6	136	175305
18	5.1	408	311.2	136	179117	411.3	136	185641	408.8	136	178479
20	3.6	400	119.1	140	664459	144.4	268	672527	210.0	272	420973
25	3.2	400	144.2	140	557700	175.1	136	564308	230.2	136	382022
30	2.7	596	132.0	136	604416	165.6	280	610398	225.3	268	398977
35	2.3	884	127.4	136	641647	153.0	140	646662	223.9	276	416138
40	2.1	904	121.2	136	676299	146.8	136	681388	212.6	136	428080
45	1.9	940	121.8	144	707076	145.7	136	711484	205.1	136	441194
50	1.8	864	119.1	136	722950	140.7	136	727517	208.3	136	444108

Table B.6: Benchmarks of regular expressions equivalence-testing algorithms (size 75).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	9.3	1432	283.6	1192	36063	410.0	1720	37086	153.3	3524	61644
3	13.9	400	266.5	484	59559	416.3	804	61380	302.2	1196	74411
4	13.2	276	272.0	272	80638	395.4	408	82708	337.8	676	95956
5	10.6	268	266.9	272	95530	391.8	408	100374	343.4	276	115800
6	8.5	532	273.7	268	107157	398.2	276	111108	341.4	268	131119
7	6.9	400	278.6	272	122893	380.0	268	128586	333.6	136	148278
8	5.7	400	277.7	140	131493	388.9	268	136093	326.6	268	161075
9	5.8	536	267.0	136	133965	377.5	268	139824	328.9	268	159229
10	4.8	400	263.5	136	146084	367.3	276	150963	326.2	276	168565
11	4.2	532	267.6	140	153628	372.1	272	158385	327.3	268	175533
12	3.6	536	254.5	268	163684	362.8	280	169455	315.6	140	187537
13	3.2	532	258.1	136	170562	352.4	136	176285	315.1	272	192396
14	2.8	532	260.0	268	179788	351.9	268	186690	310.8	280	200219
15	2.6	532	253.1	268	191042	329.5	272	196896	307.5	272	208580
16	2.4	532	241.4	136	200343	324.5	272	206708	310.2	268	211842
18	2.3	532	243.1	136	207576	324.3	268	214129	294.7	136	223518
20	1.5	532	79.5	136	870056	93.2	268	881631	132.9	276	575524
25	1.3	532	96.8	136	689223	118.9	268	696377	147.1	136	509297
30	1.0	1384	88.8	136	762116	106.3	268	768618	143.7	140	537931
35	0.8	1384	84.9	276	848089	96.9	268	854698	134.2	136	569495
40	0.8	1376	79.8	136	907536	92.3	268	914885	132.5	136	592157
45	0.7	1336	75.2	136	955183	87.3	268	961794	127.7	268	614150
50	0.6	1344	71.8	136	1006344	84.2	268	1012035	126.0	268	631549

Table B.7: Benchmarks of regular expressions equivalence-testing algorithms (size 100).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	2.2	2800	209.9	2388	38550	309.5	4280	39214	71.4	15884	84770
3	6.5	568	224.5	800	63806	336.5	536	65479	217.4	2292	85415
4	7.4	408	219.5	664	86498	332.1	404	89267	256.6	2164	110977
5	6.2	404	219.6	796	108573	328.0	536	110968	268.4	268	136044
6	4.9	556	238.7	400	119679	322.3	404	123954	264.6	540	157673
7	4.0	664	231.5	272	134613	322.2	272	139378	259.0	272	174327
8	3.2	544	235.4	276	145154	320.4	268	151342	261.2	268	188487
9	3.2	536	235.4	276	145386	316.4	268	150470	263.9	268	183894
10	2.7	804	233.6	276	157916	304.8	268	163663	261.5	268	193594
11	2.3	672	240.0	268	167414	312.0	268	173490	252.4	268	210254
12	1.9	672	231.5	268	179416	292.4	276	185913	259.4	268	215162
13	1.6	700	220.8	140	189870	288.9	272	195911	250.1	268	229126
14	1.4	772	219.1	136	203402	279.4	272	208432	255.6	272	230490
15	1.2	664	221.5	136	206895	283.5	272	214074	253.4	268	237923
16	1.1	588	217.0	136	221282	275.1	400	228103	245.7	276	249826
18	1.0	684	202.9	268	229560	266.6	268	237276	235.7	268	259386
20	0.6	940	58.1	272	1018768	69.0	404	1033832	95.6	276	711213
25	0.5	976	79.3	144	767322	91.0	268	776254	111.7	400	599025

continued on next page

continued from previous page

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
30	0.4	1856	72.2	276	875694	81.0	404	884382	108.0	268	643636
35	0.3	1068	63.7	268	987560	72.1	404	996289	97.4	268	696808
40	0.3	2040	57.4	268	1081206	66.4	268	1088191	93.0	400	734267
45	0.2	2144	56.2	272	1159856	62.7	400	1166568	88.4	408	769868
50	0.2	2160	54.2	268	1211789	58.8	268	1218459	90.1	416	782974

Table B.8: Benchmarks of regular expressions equivalence-testing algorithms (size 125).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	0.5	5384	100.3	9504	42496	207.7	13992	45665	41.3	25892	112773
3	3.4	1336	183.3	664	68383	278.6	804	70014	147.3	3412	101248
4	4.4	532	183.3	476	93711	284.3	928	96023	207.5	1348	126676
5	3.5	432	191.6	392	111134	288.4	796	115880	226.0	532	151127
6	2.7	680	198.0	408	126714	257.8	400	131243	225.1	532	172786
7	2.1	724	199.4	276	141135	263.9	404	148230	224.8	400	193946
8	1.7	772	200.7	268	154325	265.8	400	160686	218.5	268	209234
9	1.7	792	201.6	412	153130	259.3	404	159142	225.7	408	206481
10	1.4	732	194.8	272	170536	252.2	280	175804	221.4	412	228157
11	1.2	816	191.9	272	183560	245.3	268	189818	214.3	400	232710
12	1.0	744	189.8	272	198136	242.5	400	206238	207.9	408	249463
13	0.9	808	190.4	268	202083	236.9	276	210364	216.4	400	254350
14	0.8	792	185.4	276	211380	236.7	400	218677	219.0	400	260095
15	0.7	928	183.7	268	223636	233.0	400	231731	207.5	268	270647
16	0.6	940	180.9	272	231277	232.3	400	238654	204.7	400	278949
18	0.6	884	175.2	268	244239	220.7	268	253977	201.4	400	283888
20	0.4	948	48.3	404	1142839	52.2	532	1158279	73.3	704	824089
25	0.3	1044	67.9	400	817440	74.8	400	827499	92.4	560	673053
30	0.2	2636	57.9	400	936558	66.0	532	946277	86.7	556	740150
35	0.2	2520	53.1	268	1074694	57.9	536	1084748	78.0	576	802048
40	0.1	2140	45.3	408	1193285	51.8	540	1201834	75.9	400	848259
45	0.1	2720	43.3	268	1289318	48.5	404	1297328	72.6	548	891808
50	0.1	2872	41.2	268	1386283	45.5	532	1395621	69.8	720	927760

Table B.9: Benchmarks of regular expressions equivalence-testing algorithms (size 150).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	0.1	18140	110.1	10672	42649	178.8	17796	45389	18.5	90880	176581
3	1.7	2704	156.6	2124	71768	220.1	3492	73620	119.7	3916	111922
4	2.7	696	165.7	840	97367	234.7	984	100575	180.0	1864	137453
5	2.2	664	177.6	580	115961	239.4	564	120928	185.1	788	169526
6	1.8	1004	175.2	532	130548	243.2	572	134674	191.9	548	194343
7	1.4	824	177.7	556	149459	237.5	436	154828	193.5	436	215126
8	1.2	940	179.8	448	165596	233.6	484	171861	188.1	432	233495
9	1.1	984	173.8	476	164114	234.3	592	169814	189.7	440	230857

continued on next page

continued from previous page

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
10	1.0	996	176.9	408	180392	226.0	452	187584	189.5	400	242611
11	0.8	888	179.4	440	186518	230.2	448	193354	186.9	416	258971
12	0.7	960	175.4	412	200466	223.2	404	207536	183.7	448	272522
13	0.6	988	168.8	448	216149	217.7	420	223967	185.3	208	281551
14	0.5	848	166.5	276	223317	214.1	436	230065	184.6	576	283034
15	0.5	1012	162.7	400	239892	206.5	400	250987	184.6	596	301016
16	0.4	944	159.9	268	253370	201.4	400	263335	175.1	540	312864
18	0.4	996	154.9	404	255690	195.8	416	264277	172.7	560	314611
20	0.2	1472	38.0	400	1247809	42.6	664	1265720	60.7	708	933809
25	0.2	1408	58.1	400	847247	64.2	532	857742	82.7	688	728230
30	0.1	2852	48.8	400	1002738	56.1	540	1012038	74.0	696	810543
35	0.1	3172	42.5	408	1162337	48.8	668	1172209	64.1	708	892154
40	0.1	3236	38.5	532	1297939	42.1	532	1307021	60.2	692	952220
45	0.1	3448	35.1	400	1410737	40.4	668	1419482	59.3	708	1008542
50	0	3544	33.1	532	1499886	37.5	672	1508680	56.6	840	1040679

Table B.10: Benchmarks of regular expressions equivalence-testing algorithms (size 175).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	0	20360	36.3	43464	49213	93.9	59016	51810	10.7	93956	224312
3	0.9	2316	134.0	2440	75421	195.4	2532	78295	86.1	16488	124301
4	1.9	696	144.5	1136	102270	206.1	1652	106461	153.9	1208	149708
5	1.7	804	153.3	972	122265	215.0	1152	125030	163.5	1072	181255
6	1.3	1012	162.0	932	136267	220.7	616	140914	160.3	556	216998
7	1.0	1048	166.8	652	155894	215.3	1092	161818	162.1	604	240164
8	0.8	896	162.1	460	166758	216.3	588	172239	165.9	588	257578
9	0.8	984	159.2	584	169826	211.6	464	175339	169.2	600	255944
10	0.7	1044	166.0	608	191649	204.6	588	197890	164.3	608	271062
11	0.6	1104	166.0	568	195871	204.5	428	204745	157.5	608	293876
12	0.5	1080	157.0	568	208877	205.0	580	214210	163.4	652	300855
13	0.4	704	152.8	440	220109	200.6	496	227637	161.4	728	313476
14	0.4	1180	151.8	580	234162	196.0	480	242429	163.2	616	319313
15	0.3	1136	159.6	400	247980	189.4	460	258135	159.3	600	330202
16	0.3	1548	143.2	548	263616	183.8	592	273944	154.2	756	343480
18	0.3	1440	139.1	432	269405	178.4	708	279400	157.9	588	338987
20	0.1	1748	33.5	556	1294152	36.8	868	1314558	52.6	740	1016705
25	0.1	1580	53.9	536	863518	60.6	852	873614	67.6	852	801106
30	0.1	3612	44.8	532	1047866	48.5	712	1058668	63.2	840	877462
35	0.1	3172	39.5	600	1189415	43.8	844	1199185	55.9	732	962231
40	0	3964	34.8	588	1340826	37.8	816	1346892	53.0	784	1036137
45	0	3604	30.2	692	1495367	33.5	836	1505255	46.4	876	1106379
50	0	4296	27.8	616	1617031	30.8	828	1625854	46.3	760	1159709

Table B.11: Benchmarks of regular expressions equivalence-testing algorithms (size 200).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	0	15124	82.8	28212	42703	147.6	45376	43055	5.8	147444	340449
3	0.5	3240	120.7	7248	78047	181.9	2380	78772	75.3	12736	134746
4	1.4	888	135.2	1448	104775	189.0	1656	109103	125.6	2308	164426
5	1.3	676	138.1	844	129101	195.7	872	135019	143.3	1888	198829
6	1.0	960	147.2	560	143377	203.5	608	146674	150.4	700	228369
7	0.8	1076	157.1	444	159429	199.4	548	165187	150.4	724	257117
8	0.6	1212	145.7	564	173049	195.9	724	182148	149.4	728	278413
9	0.6	1184	142.7	584	179280	194.5	548	185550	146.0	708	277206
10	0.5	1260	159.7	560	190318	191.7	576	200540	147.4	688	296465
11	0.4	1276	156.1	404	202446	191.6	536	209971	149.0	732	308262
12	0.3	1284	144.2	412	210664	188.3	556	218561	144.5	712	325513
13	0.3	1564	140.8	544	229821	182.4	692	240311	143.9	676	334345
14	0.3	1684	139.0	552	240568	179.5	692	249624	150.1	748	336444
15	0.2	1632	142.8	552	259941	172.8	548	271513	147.1	716	347805
16	0.2	1712	133.9	472	267847	170.9	696	278484	146.5	696	355410
18	0.2	1692	136.5	572	283459	166.7	712	295451	138.8	820	371750
20	0.1	2156	30.9	720	1314052	35.2	876	1332485	44.9	840	1071435
25	0.1	2144	48.3	772	880051	54.5	848	892230	64.0	848	829027
30	0	3924	41.4	676	1056679	46.3	872	1066406	53.5	488	933755
35	0	4320	34.4	716	1258270	38.6	880	1267779	48.4	896	1044817
40	0	4276	32.0	704	1416601	32.4	984	1427309	45.4	548	1113537
45	0	4624	28.7	684	1567187	29.4	836	1578048	43.4	928	1189996
50	0	4968	24.6	696	1718814	27.7	968	1728790	38.7	920	1261864

Table B.12: Benchmarks of regular expressions equivalence-testing algorithms (size 250).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	0	24280	89.1	33100	42101	135.3	53756	43483	1.5	1628960	792275
3	0.1	5740	110.3	4096	79936	153.9	3628	83571	55.7	19092	155960
4	0.8	2224	117.7	2884	107967	166.6	1088	111315	101.5	4044	180802
5	0.7	920	121.8	988	134096	167.6	1020	138514	117.3	2952	224329
6	0.6	1276	122.9	1128	151983	165.3	1140	158603	118.7	1000	265352
7	0.4	1028	128.3	744	165643	172.9	756	172708	123.4	680	287596
8	0.3	1744	128.3	688	180438	166.0	596	188777	121.8	888	322425
9	0.3	1684	128.2	564	180048	171.9	608	185094	122.9	772	316961
10	0.3	1792	126.6	588	195486	164.2	632	202945	119.6	864	336736
11	0.2	1792	122.6	708	216601	161.3	764	228650	119.1	792	368731
12	0.2	1800	121.7	612	233450	160.0	768	240076	122.5	712	367055
13	0.1	1800	121.4	628	242104	154.0	740	252342	121.9	840	378420
14	0.1	1808	118.3	616	260904	150.9	676	269141	119.4	876	394693
15	0.1	1448	118.4	672	274833	147.2	680	283761	115.4	696	411041
16	0.1	2432	114.7	616	289316	144.8	724	297642	123.3	692	404055
18	0.1	2256	113.3	716	303098	145.3	672	311610	114.0	792	426284
20	0	2364	25.9	980	1425369	29.0	1264	1449403	37.6	1052	1193842
25	0	2612	43.3	424	922688	49.5	1228	934423	53.8	1088	892580

continued on next page

continued from previous page

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
30	0	5004	35.4	916	1067968	41.9	768	1079795	46.3	924	1017910
35	0	5388	28.9	532	1272790	32.9	1196	1284407	39.6	1052	1129233
40	0	5616	26.3	756	1466441	29.2	1276	1480302	35.0	684	1253924
45	0	5712	23.2	1068	1656180	25.5	1336	1668491	32.0	904	1355850
50	0	6040	20.0	928	1829066	22.7	1280	1840555	31.7	952	1416892

Table B.13: Benchmarks of regular expressions equivalence-testing algorithms (size 300).

k	**H**		**E**			$\mathbf{E}_{\mathrm{p}}$			$\mathbf{E}_{\mathrm{UF}}$		
	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	0	45764	97.5	7272	40706	118.2	11924	41661	0.8	735804	1359307
3	0	7756	85.7	13440	80145	125.7	4072	81518	37.0	63908	185316
4	0.5	2612	100.3	2164	112993	132.3	1888	117499	76.6	5068	215123
5	0.5	1636	108.3	1476	136348	139.9	972	140333	94.1	2176	254318
6	0.4	1960	115.0	1068	150291	140.7	836	153659	99.9	1316	283922
7	0.3	1936	111.9	696	178019	136.5	868	184100	97.5	908	334956
8	0.2	1968	112.3	868	191040	135.8	856	199192	99.3	880	361854
9	0.2	1996	113.8	708	187431	136.3	772	195004	98.5	1248	359315
10	0.2	1524	112.2	780	208644	132.8	764	217145	96.4	1104	392960
11	0.1	1456	111.0	704	223752	131.7	384	235319	95.9	932	399759
12	0.1	2344	109.5	712	237247	129.2	820	247835	100.6	896	405017
13	0.1	2292	109.0	848	251086	131.4	868	258626	97.8	1016	435618
14	0.1	2448	106.7	820	270982	127.9	836	280175	95.5	848	451183
15	0.1	2376	106.6	816	280009	124.2	740	290853	95.1	828	459198
16	0	2452	104.1	712	299581	120.5	756	310690	94.2	800	466710
18	0	2680	105.4	736	306700	123.1	744	315978	93.1	840	479033
20	0	2992	24.1	1364	1435185	24.8	1620	1461986	31.0	1612	1305703
25	0	3308	40.7	724	929140	43.0	1524	934820	44.6	1380	941082
30	0	5992	34.1	792	1116738	36.0	1564	1127015	39.5	1052	1051237
35	0	6344	26.1	1144	1329941	30.0	1544	1343321	33.9	1560	1226768
40	0	6692	21.8	1276	1537374	24.5	1636	1548112	30.1	1560	1359841
45	0	7100	19.2	1300	1730990	21.1	1644	1741320	26.7	1484	1462583
50	0	7480	17.2	1236	1905520	19.6	1624	1917470	26.1	1504	1550359

Appendix C

Finite automata minimisation

The following tables present experimental comparative results of several finite automata minimisation algorithms (cf. Chapter 8). The sampling and experimental study were conducted as described in Chapter 5, using datasets with 20 000 pairs of automata with n states over an alphabet of k symbols. Column *Perf.* refers to the performance of the given algorithm, i.e., the number of automata minimised per second. The memory usage, in *kilobytes*, is shown on column *Sp*. For matters of space economy, the names of the algorithms are not used. Instead, the following mapping applies.

Algorithm Name	Column Id.
DFA-MINIMISE-MOORE	**M**
DFA-MINIMISE-HOPCROFT	**H**
DFA-MINIMISE-BRZOZOWSKI	**B**
DFA-MINIMISE-INCREMENTAL	**I**

C.1 ICDFAs

Table C.1: Benchmark of ICDFA minimisation algorithms (5 states).

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	2949.85	4	3442.34	4	1424.50	4	3616.63	4	4338.39	4
3	2754.82	4	2577.31	4	853.97	4	3205.12	4	5405.40	4
4	2604.16	4	2169.19	4	436.96	4	2617.80	4	4273.50	4
5	2436.05	4	1801.80	4	290.02	4	2312.13	4	3629.76	4
6	2207.50	4	1522.07	4	217.69	4	1865.67	4	4255.31	4
7	2145.92	4	1330.67	4	189.55	4	1277.13	4	3787.87	4
8	1966.56	4	1236.09	4	141.94	4	923.36	4	3527.33	4
9	1901.14	4	1089.32	4	123.15	4	755.00	4	3016.59	4
10	1790.51	4	980.39	4	119.71	4	592.24	4	2762.43	4
11	1623.37	4	919.54	4	101.92	4	482.16	4	3384.09	4
12	1654.25	4	855.43	4	90.30	4	367.17	4	2785.51	4
13	1628.66	4	805.47	4	84.38	4	293.68	4	2406.73	4
14	1441.96	4	735.83	4	79.25	4	241.16	4	2341.92	4
15	1405.48	4	691.80	4	71.40	4	222.64	4	2378.12	4
16	1380.26	4	674.08	4	70.31	4	184.94	4	2463.05	4
18	1456.66	4	674.53	4	63.16	4	149.95	4	2430.13	4
20	1317.52	4	586.16	4	57.53	4	107.93	4	2008.03	4
25	1153.40	4	469.37	4	45.24	4	52.91	4	1814.88	4
30	1038.96	4	413.99	4	37.36	4	32.77	4	1539.64	4
35	878.34	4	374.53	4	31.73	4	21.27	4	1331.55	4
40	762.77	4	282.52	4	27.82	4	15.05	4	1480.38	4
45	845.30	4	262.70	4	24.74	4	9.90	4	1186.23	4
50	647.03	4	238.46	4	22.66	4	8.03	4	1091.70	4

Table C.2: Benchmark of ICDFA minimisation algorithms (10 states).

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	1336.89	4	1757.46	4	73.45	4	691.80	4	2358.49	4
3	1175.08	4	922.08	4	26.29	4	81.63	4	1958.86	4
4	1148.10	4	609.94	4	12.93	4	11.72	4	2047.08	4
5	1183.43	4	518.80	4	7.83	4	2.32	4	1661.12	4
6	1007.04	4	472.47	4	5.34	4	0.57	4	1712.32	4
7	956.02	4	508.38	4	3.95	4	0.16	4	1516.30	4
8	942.06	4	445.73	4	3.05	4	0.06	4	1703.57	4
9	847.09	4	278.04	4	2.45	4	0.02	4	1567.39	4
10	828.15	4	250.46	4	1.89	4	0.01	4	1484.78	4
11	789.88	4	234.10	4	1.71	4	0	4	1282.05	4
12	813.33	4	223.28	4	1.48	4	–	–	1301.23	4
13	723.32	4	205.90	4	1.29	4	–	–	1375.51	4
14	720.72	4	189.46	4	1.12	4	–	–	1272.26	4
15	662.25	4	167.74	4	1.03	4	–	–	1124.22	4
16	641.43	4	159.57	4	0.93	148	–	–	1070.09	4
18	630.31	4	137.35	4	0.78	216	–	–	1165.50	4

continued on next page

continued from previous page

k	M		H		B		W		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
20	624.60	4	123.44	4	0.68	240	–	–	1160.09	4
25	532.62	4	100.95	4	0.50	1912	–	–	885.73	4
30	417.79	4	82.52	4	0.40	1640	–	–	799.68	4
35	381.60	4	70.98	4	0.33	2788	–	–	693.48	4
40	341.35	4	60.16	4	0.28	4552	–	–	622.47	4
45	313.03	4	54.55	4	0.24	3980	–	–	539.95	4
50	281.84	4	49.74	4	0.21	3248	–	–	488.75	4

Table C.3: Benchmark of ICDFA minimisation algorithms (20 states).

k	M		H		B		W		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	412.37	4	441.69	4	0.31	110260	2.15	4	842.45	4
3	385.13	4	216.66	4	0.03	63284	0	4	704.97	4
4	390.93	4	154.59	4	0	112116	–	–	759.30	4
5	357.52	4	116.80	4	–	–	–	–	721.76	4
6	363.70	4	95.64	4	–	–	–	–	660.93	4
7	396.66	4	77.94	4	–	–	–	–	625.00	4
8	326.31	4	71.25	4	–	–	–	–	627.15	4
9	311.38	4	61.91	4	–	–	–	–	568.34	4
10	297.35	4	56.39	4	–	–	–	–	490.07	4
11	319.23	4	50.74	4	–	–	–	–	442.38	4
12	300.66	4	46.46	4	–	–	–	–	417.36	4
13	288.60	4	41.80	4	–	–	–	–	423.54	4
14	285.75	4	40.15	4	–	–	–	–	420.43	4
15	286.69	4	37.20	4	–	–	–	–	388.04	4
16	321.02	4	33.77	4	–	–	–	–	385.35	4
18	240.61	4	30.95	4	–	–	–	–	422.74	4
20	224.69	4	27.62	4	–	–	–	–	370.78	4
25	176.35	4	21.77	4	–	–	–	–	338.58	4
30	161.25	4	17.89	4	–	–	–	–	315.30	4
35	151.66	4	15.05	4	–	–	–	–	306.51	8
40	135.34	4	13.38	4	–	–	–	–	275.63	8
45	121.06	4	11.58	4	–	–	–	–	230.81	4
50	109.82	4	10.69	4	–	–	–	–	194.42	8

Table C.4: Benchmark of ICDFA minimisation algorithms (30 states).

k	M		H		B		W		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	178.55	4	127.64	4	–	–	–	–	423.01	40
3	167.98	4	75.46	4	–	–	–	–	373.76	56
4	162.98	4	52.63	4	–	–	–	–	348.06	72
5	168.25	4	40.51	4	–	–	–	–	326.47	88
6	153.90	4	34.88	4	–	–	–	–	296.91	104
7	160.64	4	28.00	4	–	–	–	–	307.40	112

continued on next page

continued from previous page

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
8	150.10	4	25.04	4	–	–	–	–	296.38	116
9	149.46	4	22.35	4	–	–	–	–	272.85	120
10	142.97	4	19.61	4	–	–	–	–	249.28	128
11	134.22	4	17.22	4	–	–	–	–	238.46	128
12	133.56	4	16.36	4	–	–	–	–	229.77	132
13	126.51	4	15.25	4	–	–	–	–	222.22	136
14	119.82	4	14.07	4	–	–	–	–	212.26	144
15	123.15	4	13.01	4	–	–	–	–	204.14	144
16	117.19	4	11.98	4	–	–	–	–	196.79	136
18	112.37	4	10.93	4	–	–	–	–	191.33	140
20	105.97	4	9.67	4	–	–	–	–	182.54	144
25	89.50	4	7.68	4	–	–	–	–	167.05	148
30	81.08	4	6.33	4	–	–	–	–	153.16	152
35	76.30	4	5.21	4	–	–	–	–	141.57	152
40	69.09	4	4.78	4	–	–	–	–	128.39	152
45	64.65	4	4.23	4	–	–	–	–	122.02	156
50	62.17	172	3.88	4	–	–	–	–	117.52	160

Table C.5: Benchmark of ICDFA minimisation algorithms (40 states).

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	90.72	4	65.93	4	–	–	–	–	191.44	116
3	90.36	4	39.96	4	–	–	–	–	196.36	184
4	86.85	4	28.27	4	–	–	–	–	169.88	200
5	85.75	4	21.45	4	–	–	–	–	163.43	240
6	86.42	4	17.64	4	–	–	–	–	159.66	256
7	81.15	4	15.29	4	–	–	–	–	153.49	280
8	82.04	4	13.07	4	–	–	–	–	149.85	284
9	81.92	4	11.49	4	–	–	–	–	145.67	300
10	76.99	4	10.25	4	–	–	–	–	143.07	304
11	76.28	4	9.43	4	–	–	–	–	143.88	312
12	75.72	4	8.61	4	–	–	–	–	132.14	320
13	74.50	4	7.90	4	–	–	–	–	134.29	320
14	69.46	4	7.38	4	–	–	–	–	132.57	332
15	68.47	4	6.76	4	–	–	–	–	129.14	332
16	68.25	4	6.41	4	–	–	–	–	128.46	332
18	66.44	4	5.74	4	–	–	–	–	120.38	340
20	62.79	4	5.00	4	–	–	–	–	116.82	348
25	56.10	276	4.04	4	–	–	–	–	103.85	356
30	51.40	1236	3.40	4	–	–	–	–	96.06	360
35	48.39	2096	2.88	4	–	–	–	–	88.73	364
40	46.31	2652	2.55	4	–	–	–	–	84.28	368
45	43.21	3288	2.24	4	–	–	–	–	79.34	364
50	41.06	3708	1.99	4	–	–	–	–	75.66	368

Table C.6: Benchmark of ICDFA minimisation algorithms (50 states).

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	56.51	4	39.48	4	–	–	–	–	117.21	288
3	57.09	4	23.26	4	–	–	–	–	109.15	312
4	54.23	4	17.07	4	–	–	–	–	104.99	404
5	57.72	4	13.16	4	–	–	–	–	106.75	424
6	56.09	4	10.63	4	–	–	–	–	102.93	452
7	55.14	4	9.03	4	–	–	–	–	98.73	484
8	55.62	4	7.61	4	–	–	–	–	97.13	512
9	53.14	4	7.00	4	–	–	–	–	94.84	528
10	51.57	4	6.31	4	–	–	–	–	94.33	540
11	51.28	4	5.66	4	–	–	–	–	93.44	552
12	49.63	172	5.02	4	–	–	–	–	89.80	560
13	48.89	172	4.75	4	–	–	–	–	89.21	576
14	49.53	172	4.25	4	–	–	–	–	87.09	576
15	47.48	688	4.14	4	–	–	–	–	85.22	580
16	47.52	548	3.85	4	–	–	–	–	84.85	584
18	45.13	940	3.41	4	–	–	–	–	80.90	592
20	43.15	940	3.08	4	–	–	–	–	79.43	600
25	40.79	2496	2.45	4	–	–	–	–	73.94	612
30	37.44	3668	2.06	4	–	–	–	–	67.54	620
35	35.11	4556	1.74	4	–	–	–	–	63.74	628
40	33.45	5016	1.50	4	–	–	–	–	59.02	632
45	31.68	6584	1.36	4	–	–	–	–	57.20	632
50	29.80	7184	1.21	4	–	–	–	–	53.31	636

Table C.7: Benchmark of ICDFA minimisation algorithms (60 states).

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	42.08	4	25.81	4	–	–	–	–	84.15	364
3	40.69	4	15.16	4	–	–	–	–	79.28	492
4	39.15	4	10.99	4	–	–	–	–	79.40	600
5	38.37	4	8.76	4	–	–	–	–	76.73	656
6	38.67	4	7.01	4	–	–	–	–	73.08	708
7	37.50	4	6.05	4	–	–	–	–	72.08	740
8	37.66	4	5.17	4	–	–	–	–	70.95	772
9	36.72	172	4.56	4	–	–	–	–	68.87	804
10	36.86	428	4.04	4	–	–	–	–	68.13	824
11	35.43	768	3.73	4	–	–	–	–	65.58	836
12	35.02	940	3.34	4	–	–	–	–	65.69	860
13	35.20	1452	3.11	4	–	–	–	–	64.35	860
14	34.64	1712	2.89	4	–	–	–	–	62.64	876
15	33.93	1988	2.69	4	–	–	–	–	61.72	880
16	33.45	2300	2.48	4	–	–	–	–	62.19	888
18	33.39	2860	2.31	4	–	–	–	–	59.99	908
20	32.04	3444	2.07	4	–	–	–	–	57.47	912
25	29.37	5184	1.65	4	–	–	–	–	54.31	932

continued on next page

continued from previous page

k	M		H		B		W		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
30	27.43	6640	1.29	4	–	–	–	–	50.55	944
35	26.50	8088	1.15	4	–	–	–	–	49.39	952
40	24.69	9628	1.02	4	–	–	–	–	45.85	964
45	23.12	10980	0.90	4	–	–	–	–	43.09	968
50	21.86	12464	0.81	4	–	–	–	–	41.38	968

Table C.8: Benchmark of ICDFA minimisation algorithms (70 states).

k	M		H		B		W		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	29.15	4	18.13	4	–	–	–	–	61.45	484
3	29.21	4	10.81	4	–	–	–	–	58.18	660
4	28.24	4	7.61	4	–	–	–	–	56.93	796
5	28.81	4	6.03	4	–	–	–	–	55.44	912
6	27.86	172	4.94	4	–	–	–	–	53.95	968
7	27.89	684	4.11	4	–	–	–	–	53.20	1036
8	27.38	940	3.55	4	–	–	–	–	53.04	1100
9	26.97	1964	3.12	4	–	–	–	–	51.90	1120
10	26.67	2004	2.70	4	–	–	–	–	49.89	1148
11	26.57	2344	2.52	4	–	–	–	–	49.70	1196
12	25.89	2868	2.33	4	–	–	–	–	49.60	1192
13	25.76	3168	2.15	4	–	–	–	–	48.87	1212
14	25.77	3472	2.02	4	–	–	–	–	45.50	1236
15	25.68	4148	1.86	4	–	–	–	–	47.14	1236
16	25.43	4348	1.73	4	–	–	–	–	46.71	1244
18	24.40	5472	1.57	4	–	–	–	–	45.22	1268
20	24.33	5828	1.43	4	–	–	–	–	44.37	1280
25	22.41	8172	1.12	4	–	–	–	–	41.89	1308
30	21.27	10204	0.93	4	–	–	–	–	39.53	1328
35	19.40	12340	0.80	4	–	–	–	–	36.93	1340
40	18.96	14056	0.71	4	–	–	–	–	35.17	1352
45	17.98	16072	0.63	4	–	–	–	–	34.16	1352
50	16.68	18084	0.55	4	–	–	–	–	32.06	1356

Table C.9: Benchmark of ICDFA minimisation algorithms (80 states).

k	M		H		B		W		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	19.99	684	13.18	4	–	–	–	–	44.91	712
3	20.34	1196	7.78	4	–	–	–	–	43.14	952
4	20.36	684	5.61	4	–	–	–	–	41.95	1088
5	20.78	684	4.36	4	–	–	–	–	41.73	1216
6	20.02	2580	3.54	4	–	–	–	–	40.39	1312
7	19.96	3244	2.99	4	–	–	–	–	39.34	1388
8	19.71	3504	2.62	4	–	–	–	–	39.49	1456
9	19.34	4332	2.28	4	–	–	–	–	38.23	1496

continued on next page

continued from previous page

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
10	19.31	4656	2.07	4	–	–	–	–	38.08	1528
11	19.10	4972	1.85	4	–	–	–	–	38.34	1572
12	19.06	5880	1.68	4	–	–	–	–	37.03	1596
13	19.01	6372	1.57	4	–	–	–	–	36.82	1612
14	18.24	6704	1.46	4	–	–	–	–	36.58	1636
15	18.59	6988	1.37	4	–	–	–	–	36.31	1648
16	18.18	7876	1.27	4	–	–	–	–	35.48	1664
18	18.34	9020	1.16	4	–	–	–	–	35.05	1688
20	18.12	9908	1.02	4	–	–	–	–	34.81	1708
25	16.51	12856	0.81	4	–	–	–	–	32.04	1748
30	15.88	15492	0.69	4	–	–	–	–	30.33	1764
35	15.05	18104	0.58	4	–	–	–	–	29.07	1780
40	14.33	20472	0.51	4	–	–	–	–	27.88	1796
45	13.58	23028	0.45	4	–	–	–	–	26.13	1804
50	13.04	25644	0.41	4	–	–	–	–	25.46	1804

Table C.10: Benchmark of ICDFA minimisation algorithms (90 states).

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	15.91	1496	7.89	4	–	–	–	–	35.87	892
3	16.16	2220	4.81	4	–	–	–	–	34.64	1164
4	18.61	80	3.44	4	–	–	–	–	33.78	1376
5	19.65	84	2.66	4	–	–	–	–	32.23	1592
6	15.65	4044	2.15	4	–	–	–	–	32.07	1668
7	15.53	4624	1.86	4	–	–	–	–	31.27	1780
8	15.59	5304	1.59	4	–	–	–	–	30.52	1864
9	15.41	5584	1.43	4	–	–	–	–	30.25	1908
10	15.24	6716	1.27	4	–	–	–	–	30.33	1968
11	15.14	7292	1.17	4	–	–	–	–	29.81	1996
12	15.11	7880	1.05	4	–	–	–	–	30.07	2036
13	14.99	8216	0.96	4	–	–	–	–	29.55	2060
14	14.72	9068	0.90	4	–	–	–	–	29.24	2092
15	14.77	9944	0.83	4	–	–	–	–	28.75	2104
16	14.34	10628	0.79	4	–	–	–	–	28.32	2132
18	14.56	11980	0.71	4	–	–	–	–	27.18	2164
20	14.40	13152	0.64	4	–	–	–	–	27.86	2192
25	13.25	16752	0.51	4	–	–	–	–	25.46	2500
30	12.68	19968	0.42	4	–	–	–	–	24.52	2528
35	12.06	23384	0.36	4	–	–	–	–	23.46	2540
40	11.58	26652	0.31	4	–	–	–	–	22.48	2564
45	11.01	15088	0.28	4	–	–	–	–	21.25	2576
50	10.32	19832	0.25	4	–	–	–	–	20.48	2720

Table C.11: Benchmark of ICDFA minimisation algorithms (100 states).

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	12.66	2424	6.25	4	–	–	–	–	28.23	1028
3	12.95	2732	3.74	4	–	–	–	–	27.73	1444
4	12.89	84	2.70	4	–	–	–	–	27.13	1708
5	12.90	84	2.09	4	–	–	–	–	26.43	1928
6	12.53	5212	1.68	4	–	–	–	–	25.87	2136
7	12.47	6104	1.42	4	–	–	–	–	25.53	2224
8	12.51	7012	1.23	4	–	–	–	–	25.25	2320
9	12.72	7888	1.10	4	–	–	–	–	25.12	2412
10	12.22	5460	0.97	4	–	–	–	–	24.94	2452
11	12.47	9372	0.88	4	–	–	–	–	24.31	2516
12	12.08	10224	0.81	4	–	–	–	–	24.43	2536
13	12.05	10836	0.75	4	–	–	–	–	24.10	2564
14	11.97	12016	0.70	4	–	–	–	–	24.35	2748
15	11.97	12336	0.64	4	–	–	–	–	23.84	2772
16	11.72	13512	0.60	4	–	–	–	–	23.61	2796
18	11.69	15200	0.55	4	–	–	–	–	23.49	2828
20	11.65	16668	0.49	4	–	–	–	–	22.53	2872
25	10.75	21176	0.37	4	–	–	–	–	21.58	3444
30	10.43	25240	0.33	4	–	–	–	–	20.58	3748
35	9.72	29328	0.28	4	–	–	–	–	19.47	3640
40	9.69	33380	0.24	4	–	–	–	–	18.98	3656
45	9.31	37212	0.22	4	–	–	–	–	18.49	3816
50	8.69	33184	0.18	4	–	–	–	–	17.17	3824

Table C.12: Benchmark of ICDFA minimisation algorithms (1000 states).

k	**M**		**H**		**B**		**W**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	0.02	407100	0.01	4	–	–	–	–	0.23	33920
3	0.02	401108	0	4	–	–	–	–	0.23	33964
5	0.02	396672	0	4	–	–	–	–	0.23	34224

C.2 NFAs

C.2.1 Transition density 0.1

Table C.13: Benchmark of NFA minimisation algorithms (5 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	1472.75	4	1841.62	4	2758.62	4	2469.13	4
3	936.32	4	1132.50	4	1665.27	4	1486.98	4
4	539.95	4	640.40	4	1002.00	4	739.64	4
5	389.18	4	426.62	4	718.64	4	540.39	4
6	245.82	4	230.54	4	442.67	4	307.64	4
7	206.33	4	159.43	4	381.31	4	223.96	4
8	149.20	4	100.28	4	227.89	4	154.48	4
9	127.13	4	77.40	4	180.39	4	130.82	4
10	98.29	4	55.57	4	134.39	4	98.59	4
11	87.97	4	45.44	4	109.28	4	86.04	4
12	75.64	4	35.39	4	83.30	4	70.62	4
13	68.19	4	30.39	4	71.82	4	64.40	4
14	58.40	4	25.40	4	61.17	4	54.41	4
15	54.64	4	22.55	4	54.34	4	55.15	4
16	46.50	4	18.99	4	46.77	4	47.85	4
18	38.60	4	15.38	4	38.04	4	41.70	4
20	33.04	4	12.13	4	31.40	4	35.23	4
25	24.58	4	8.39	4	21.29	4	26.22	4
30	20.04	4	6.07	4	15.86	4	20.59	4
35	16.97	4	4.86	4	12.30	4	17.52	4
40	14.44	4	3.98	4	9.92	4	14.83	4
45	12.73	4	3.35	4	8.40	4	13.05	4
50	11.37	4	2.83	4	7.20	4	11.57	4

Table C.14: Benchmark of NFA minimisation algorithms (10 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	121.29	4388	188.19	4	244.17	4	178.68	940
3	22.24	50352	17.48	4	33.47	136	29.26	13944
4	7.04	112268	3.39	4	9.26	84	10.14	21372
5	3.11	137640	1.14	4	3.80	84	4.60	32076
6	1.61	167044	0.47	56	1.87	184	2.56	27936
7	0.95	186900	0.22	752	1.09	152	1.59	42644
8	0.59	236764	0.11	192	0.69	276	1.08	64432
9	0.40	317660	0.06	208	0.46	5620	0.76	84360
10	0.28	394508	0.04	1984	0.33	8456	0.59	97536
11	0.21	411468	0.03	1852	0.25	8968	0.47	102772
12	0.16	341628	0.02	1280	0.19	8484	0.38	91084
13	0.13	469312	0.01	2044	0.15	9084	0.32	111716
14	0.10	424680	0.01	1992	0.12	8892	0.27	108192

continued on next page

continued from previous page

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
15	0.08	454912	0.01	2488	0.10	9956	0.23	110484
16	0.07	412864	0	2732	0.08	8984	0.20	122136
18	–	–	–	–	0.06	9928	0.16	135208
20	–	–	–	–	0.05	14124	0.13	147228
25	–	–	–	–	0.02	24296	0.08	166112
30	–	–	–	–	0.01	25540	0.06	184780
35	–	–	–	–	0.01	28292	0.05	208896
40	–	–	–	–	0.01	28228	0.04	196908
45	–	–	–	–	0	30328	0.03	224508
50	–	–	–	–	–	–	0.03	246756

Table C.15: Benchmark of NFA minimisation algorithms (20 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	5.62	120428	46.58	4	159.22	4	18.41	3756
3	0.01	2046828	1.44	3348	6.51	48	0.39	196892
4	0	3127380	0.01	15380	0.18	65832	0.02	2019372
5	0	2144692	0	14600	0.01	678420	0	2011584
6	0	2151212	–	–	0	1357920	–	–
7	–	–	–	–	–	–	–	–
8	–	–	–	–	–	–	–	–
9	–	–	–	–	–	–	–	–
10	–	–	–	–	–	–	–	–
11	–	–	–	–	–	–	–	–
12	–	–	–	–	–	–	–	–
13	–	–	–	–	–	–	–	–
14	–	–	–	–	–	–	–	–
15	–	–	–	–	–	–	–	–
16	–	–	–	–	–	–	–	–
18	–	–	–	–	–	–	–	–
20	–	–	–	–	–	–	–	–
25	–	–	–	–	–	–	–	–
30	–	–	–	–	–	–	–	–
35	–	–	–	–	–	–	–	–
40	–	–	–	–	–	–	–	–
45	–	–	–	–	–	–	–	–
50	–	–	–	–	–	–	–	–

Table C.16: Benchmark of NFA minimisation algorithms (40 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	38.74	80	53.31	4	159.97	4	49.38	4
3	4.92	55148	9.71	4	70.72	4	8.26	4
4	0.58	892300	2.69	708	36.15	4	1.91	1268
5	0.03	2041204	0.91	4092	21.40	4	0.53	36472

continued on next page

continued from previous page

k	M		H		B		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
6	0	2051812	0.36	30232	13.65	1668	0.15	58832
7	0	1996880	0.14	63800	7.59	11872	0.04	160160
8	0	1960764	0.05	131532	5.82	18624	0.01	613596
9	0	1935436	0.02	195008	4.95	9560	0	2002856
10	0	2192876	0	310940	2.57	41136	–	–
11	0	1960160	–	–	2.85	8244	–	–
12	0	2005308	–	–	0.17	2048536	–	–
13	0	2200008	–	–	0.26	2113540	–	–
14	0	2216932	–	–	1.13	2011504	–	–
15	0	2262660	–	–	0.19	1991260	–	–
16	0	4048120	–	–	0.22	2059956	–	–
18	–	–	–	–	–	–	–	–
20	–	–	–	–	–	–	–	–
25	–	–	–	–	–	–	–	–
30	–	–	–	–	–	–	–	–
35	–	–	–	–	–	–	–	–
40	–	–	–	–	–	–	–	–
45	–	–	–	–	–	–	–	–
50	–	–	–	–	–	–	–	–

Table C.17: Benchmark of NFA minimisation algorithms (60 states).

k	M		H		B		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	38.40	4	42.73	4	110.02	4	42.82	4
3	8.73	940	9.76	4	57.97	4	9.67	4
4	2.56	30932	3.41	4	34.40	4	3.34	4
5	0.90	169436	1.44	4	23.58	4	1.36	4
6	0.34	533612	0.69	3356	17.21	4	0.66	2044
7	0.12	1103324	0.39	6204	12.89	4	0.34	4460
8	0.04	2019804	0.23	11176	9.62	4	0.19	9944
9	–	–	0.14	15480	7.77	2688	0.11	33088
10	–	–	0.09	29140	6.16	1772	0.06	16176
11	–	–	0.06	39704	5.19	156	0.04	35688
12	–	–	0.04	54628	4.31	4348	0.02	40664
13	–	–	0.03	63812	3.62	84	–	–
14	–	–	0.02	117552	2.95	18276	–	–
15	–	–	0.01	87536	2.61	9144	–	–
16	–	–	0.01	117532	2.23	13488	–	–
18	–	–	–	–	1.56	40728	–	–
20	–	–	–	–	1.34	5852	–	–
25	–	–	–	–	0.61	200264	–	–
30	–	–	–	–	0.51	138948	–	–
35	–	–	–	–	0.25	2313528	–	–
40	–	–	–	–	0.03	2231108	–	–
45	–	–	–	–	0.12	2115632	–	–

continued on next page

continued from previous page

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
50	–	–	–	–	0.13	2019680	–	–

Table C.18: Benchmark of NFA minimisation algorithms (80 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	35.91	4	36.85	4	74.82	4	37.84	4
3	9.21	4	9.52	4	42.92	4	9.79	4
4	2.94	2170524	3.56	13580	28.40	4	3.28	67304
5	1.41	6828	1.60	4	19.81	4	1.60	4
6	0.68	42588	0.82	4	14.38	4	0.83	4
7	–	–	0.49	52	11.55	4	0.46	224
8	–	–	0.30	2240	8.63	4	0.28	3268
9	–	–	0.20	3660	7.41	4	0.19	3640
10	–	–	0.13	5764	6.19	300	0.12	4120
11	–	–	0.08	7968	5.21	84	0.09	7612
12	–	–	0.06	10836	4.38	4	0.06	8064
13	–	–	0.05	13032	3.88	4	–	–
14	–	–	0.03	19864	3.41	2032	–	–
15	–	–	0.03	21160	3.01	104	–	–
16	–	–	0.02	26856	2.54	4696	–	–
18	–	–	–	–	2.06	1948	–	–
20	–	–	–	–	1.70	2276	–	–
25	–	–	–	–	1.09	28044	–	–
30	–	–	–	–	0.76	45932	–	–
35	–	–	–	–	0.55	84072	–	–
40	–	–	–	–	0.36	113956	–	–
45	–	–	–	–	0.33	27088	–	–
50	–	–	–	–	0.24	152020	–	–

Table C.19: Benchmark of NFA minimisation algorithms (100 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	31.03	4	30.04	4	52.97	4	31.19	4
3	9.62	4	9.90	4	32.30	4	10.06	4
4	3.79	4	3.86	4	21.83	4	3.98	4
5	1.83	364	1.92	4	16.34	4	1.95	4
6	0.96	6448	1.04	4	12.39	4	1.05	4
7	0.58	24904	0.59	4	9.81	4	0.62	4
8	0.35	45864	0.40	180	8.04	4	0.39	4
9	0.22	75688	0.27	84	6.62	4	0.27	260
10	0.15	134316	0.18	344	5.67	4	0.18	272
11	0.11	199668	0.14	3244	4.81	108	–	–
12	0.07	340004	0.10	3896	4.15	136	–	–

continued on next page

continued from previous page

k	M		H		B		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
13	0.05	601192	0.08	7164	3.72	108	–	–
14	0.04	488056	0.06	7128	3.25	256	–	–
15	0.02	641520	0.05	9724	2.87	224	–	–
16	0.02	810172	0.04	9740	2.58	3628	–	–
18	–	–	–	–	2.00	4532	–	–
20	–	–	–	–	1.73	5460	–	–
25	–	–	–	–	1.14	13856	–	–
30	–	–	–	–	0.80	11840	–	–
35	–	–	–	–	0.58	27708	–	–
40	–	–	–	–	0.44	41200	–	–
45	–	–	–	–	0.36	68044	–	–
50	–	–	–	–	0.30	29704	–	–

C.2.2 Transition density 0.5

Table C.20: Benchmark of NFA minimisation algorithms (5 states).

k	M		H		B		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	944.73	4	2057.61	4	1860.46	4	1499.25	4
3	390.47	4	1104.97	4	1060.44	4	637.75	4
4	211.79	4	443.75	4	725.42	4	322.52	4
5	110.76	4	292.39	4	460.08	4	166.66	4
6	74.39	4	155.42	4	309.31	4	96.21	4
7	44.34	4	108.73	4	217.84	4	59.88	4
8	36.10	4	78.65	4	150.42	4	47.31	4
9	27.05	172	59.83	4	107.93	4	34.45	4
10	22.31	84	45.70	4	79.89	4	26.51	4
11	17.87	84	35.59	4	59.40	4	20.93	4
12	15.82	428	28.04	4	48.03	4	18.52	4
13	13.65	172	22.58	4	36.86	4	15.58	4
14	12.67	624	19.69	4	31.58	4	14.08	4
15	11.25	656	16.30	4	25.30	4	12.44	4
16	10.57	684	14.15	4	21.88	4	11.80	4
18	–	–	–	–	16.23	4	10.34	4
20	–	–	–	–	12.84	4	9.90	4
25	–	–	–	–	8.39	4	8.93	4
30	–	–	–	–	6.45	4	8.33	4
35	–	–	–	–	5.31	4	8.06	4
40	–	–	–	–	4.67	4	7.38	4
45	–	–	–	–	4.00	4	6.88	4
50	–	–	–	–	3.59	4	6.29	4

Table C.21: Benchmark of NFA minimisation algorithms (10 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	622.47	4	924.21	4	1250.00	4	1212.12	4
3	314.36	4	445.23	4	799.04	4	498.38	4
4	157.17	4	200.92	4	546.29	4	199.02	4
5	94.88	4	129.80	4	393.23	4	118.00	4
6	59.86	4	79.19	4	293.77	4	72.74	4
7	43.12	4	56.95	4	215.91	4	54.21	4
8	30.80	4	41.76	4	173.02	4	38.08	4
9	22.47	1196	31.90	4	143.98	4	29.83	4
10	17.18	236	25.52	4	118.51	4	22.95	4
11	13.13	84	21.33	4	105.46	4	18.73	4
12	11.02	940	17.43	4	90.38	4	15.89	4
13	8.54	6508	14.40	4	79.43	4	12.84	4
14	7.00	2732	12.53	4	71.57	4	10.92	4
15	5.73	25196	10.58	4	64.54	4	9.24	4
16	4.80	8616	9.24	4	58.63	4	7.99	4
18	–	–	–	–	49.81	4	5.91	4
20	–	–	–	–	41.61	4	4.72	4
25	–	–	–	–	28.76	4	2.66	252
30	–	–	–	–	21.29	4	1.65	1848
35	–	–	–	–	15.55	4	1.11	5672
40	–	–	–	–	12.00	84	0.76	5500
45	–	–	–	–	9.23	84	0.55	19624
50	–	–	–	–	7.43	100	0.40	30052

Table C.22: Benchmark of NFA minimisation algorithms (20 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	469.92	4	687.52	4	471.92	4	477.32	4
3	166.01	4	182.46	4	212.24	4	171.04	4
4	97.05	4	96.39	4	167.00	4	102.31	4
5	60.81	4	63.12	4	130.40	4	65.08	4
6	43.25	4	44.64	4	100.92	4	45.86	4
7	30.61	4	33.51	4	84.23	4	32.73	4
8	23.12	4	24.55	4	74.33	4	24.70	4
9	18.10	4	18.79	4	64.66	4	19.48	4
10	14.52	4	14.78	4	57.02	4	15.04	4
11	10.91	4	12.42	4	49.54	4	12.28	4
12	9.42	4	10.05	4	45.80	4	10.13	4
13	7.88	4	8.45	4	40.61	4	8.39	4
14	6.69	4	7.35	4	37.19	4	7.13	4
15	5.57	4	6.18	4	35.31	4	6.04	4
16	4.46	4	5.43	4	32.04	4	5.33	4
18	3.68	4	–	–	28.88	4	4.08	4
20	2.97	1204	–	–	24.42	4	3.31	4
25	1.76	428	–	–	18.42	4	2.07	4

continued on next page

continued from previous page

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
30	1.19	1388	–	–	14.18	4	1.30	4
35	0.86	2264	–	–	11.84	4	1.01	4
40	0.64	6784	–	–	9.94	4	0.78	4
45	0.50	9532	–	–	8.16	4	0.63	140
50	0.40	10944	–	–	7.22	240	0.51	180

Table C.23: Benchmark of NFA minimisation algorithms (40 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	100.05	4	91.94	4	97.42	4	100.62	4
3	49.72	4	48.30	4	59.75	4	50.01	4
4	30.83	4	29.71	4	44.39	4	31.38	4
5	21.19	4	21.25	4	34.29	4	21.46	4
6	15.47	4	15.32	4	28.09	4	15.78	4
7	12.18	4	11.91	4	24.08	4	12.09	4
8	9.48	4	9.41	4	21.10	4	9.17	4
9	7.65	4	7.87	4	18.46	4	7.64	4
10	6.42	4	6.33	4	16.44	4	6.35	4
11	5.35	4	5.30	4	15.07	4	5.38	4
12	4.45	4	4.58	4	13.72	4	4.49	4
13	3.81	4	3.89	4	12.38	4	3.89	4
14	3.33	4	3.40	4	11.63	4	3.36	176
15	3.01	4	2.94	4	10.65	180	3.02	140
16	2.62	4	2.62	188	10.00	4	2.76	184
18	2.08	4	–	–	8.88	288	2.17	328
20	1.73	212	–	–	8.02	504	1.79	320
25	1.15	316	–	–	6.14	1292	1.15	772
30	0.77	2172	–	–	4.95	1920	0.78	1536
35	0.57	2772	–	–	4.16	2464	0.58	1872
40	0.43	2832	–	–	3.63	2336	0.45	2096
45	0.34	5452	–	–	3.10	2836	0.36	2556
50	0.27	6192	–	–	2.73	4488	0.29	4800

Table C.24: Benchmark of NFA minimisation algorithms (60 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	40.77	4	38.16	4	39.99	4	40.40	4
3	22.06	4	22.29	4	25.97	4	22.13	4
4	13.65	4	13.55	4	19.33	4	13.83	4
5	9.41	4	9.43	4	15.49	4	9.62	4
6	7.09	4	7.07	4	12.76	4	7.04	4
7	5.27	4	5.40	228	10.87	272	5.28	348
8	4.25	332	4.23	568	9.36	576	4.17	576
9	3.42	664	3.55	548	8.25	796	3.48	988

continued on next page

continued from previous page

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
10	2.87	888	2.80	1112	7.40	1120	2.85	1300
11	2.44	1300	2.37	1512	6.77	1408	2.35	1700
12	2.05	1564	2.06	1856	6.07	1392	2.11	2016
13	1.76	1996	1.75	2120	5.69	1704	1.77	2448
14	1.53	2428	1.53	2100	5.29	1824	1.54	2748
15	1.32	2732	1.35	2540	4.89	2728	1.36	3040
16	1.20	3032	1.19	3036	4.60	2392	1.19	3352
18	–	–	–	–	4.03	3184	0.94	3964
20	–	–	–	–	3.61	4020	0.77	4140
25	–	–	–	–	2.86	5596	0.51	6344
30	–	–	–	–	2.35	7404	0.37	7104
35	–	–	–	–	1.99	8104	0.27	10612
40	–	–	–	–	1.72	11680	0.21	11564
45	–	–	–	–	1.51	13336	0.16	14000
50	–	–	–	–	1.36	15784	0.13	14388

Table C.25: Benchmark of NFA minimisation algorithms (80 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	23.38	4	22.72	4	22.28	4	23.23	4
3	12.62	4	12.16	4	14.92	4	12.57	4
4	7.99	4	7.70	4	11.07	4	8.04	4
5	5.46	4	5.49	484	9.11	268	5.45	472
6	4.05	64	3.95	712	7.22	260	4.04	616
7	3.02	344	3.06	1160	6.12	640	3.09	716
8	2.45	892	2.41	1124	5.46	1472	2.40	1964
9	1.96	1200	1.97	1604	4.70	1692	1.98	1720
10	1.62	1648	1.62	2132	4.27	1984	1.59	1992
11	1.36	2616	1.35	2936	3.88	2540	1.34	2496
12	1.17	3016	1.16	3168	3.53	3076	1.17	3060
13	1.00	3440	0.98	3152	3.24	3296	1.01	3744
14	0.90	3184	0.85	3388	3.04	3580	0.88	3760
15	0.78	3768	0.76	4124	2.85	4468	0.77	4116
16	0.67	4132	0.68	4660	2.68	4724	0.68	4352
18	0.53	4672	–	–	2.30	5764	0.55	5552
20	0.44	5712	–	–	2.09	5888	0.45	6188
25	0.28	8480	–	–	1.63	8496	0.29	9384
30	0.20	14012	–	–	1.35	12932	0.20	13648
35	0.15	16428	–	–	1.16	15300	0.15	16252
40	0.11	18660	–	–	1.02	17596	0.12	17796
45	0.09	22040	–	–	0.88	20428	0.09	20420
50	0.07	24020	–	–	0.78	21712	0.07	23312

Table C.26: Benchmark of NFA minimisation algorithms (100 states).

k	M		H		B		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	14.94	4	14.79	4	13.41	4	15.07	4
3	8.21	4	7.96	4	9.68	4	8.10	4
4	5.08	4	4.94	84	7.26	80	4.95	224
5	3.52	688	3.48	720	5.62	552	3.43	732
6	2.58	768	2.51	1164	4.67	1640	2.55	1308
7	1.95	1760	1.97	1656	4.04	1500	1.94	2384
8	1.56	2108	1.55	1996	3.49	2748	1.60	2812
9	1.27	2640	1.24	3028	3.09	3172	1.27	3208
10	1.05	2860	1.04	4132	2.84	3512	1.07	4100
11	0.89	3820	0.84	3740	2.52	4140	0.86	4048
12	0.75	4316	0.72	4888	2.29	4412	0.76	5160
13	0.63	5092	0.64	4808	2.13	4740	0.64	4980
14	0.56	4680	0.56	5080	1.98	6080	0.55	5292
15	0.48	6356	0.50	6088	1.83	6176	0.50	5800
16	0.44	5980	0.44	6676	1.71	6252	0.43	7300
18	0.34	7592	–	–	1.53	8220	0.34	6828
20	0.30	8400	–	–	1.36	9576	0.28	9408
25	0.18	14216	–	–	1.06	12672	0.19	14872
30	0.12	18144	–	–	0.89	15728	0.13	17072
35	0.09	20240	–	–	0.75	20296	0.09	22160
40	0.07	25864	–	–	0.65	23560	0.07	24620
45	0.05	29904	–	–	0.57	26036	0.06	28488
50	0.05	32012	–	–	0.52	28212	0.05	30968

C.2.3 Transition density 0.8

Table C.27: Benchmark of NFA minimisation algorithms (5 states).

k	M		H		B		I	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	1518.60	4	2453.98	4	1972.38	4	2242.15	4
3	1011.12	4	1633.98	4	1423.48	4	1465.20	4
4	705.21	4	1391.78	4	1102.53	4	966.65	4
5	518.67	4	816.65	4	913.24	4	730.19	4
6	383.06	4	673.62	4	733.40	4	521.51	4
7	307.97	4	478.24	4	777.90	4	433.46	4
8	262.19	4	480.65	4	642.67	4	389.18	4
9	218.69	4	294.55	4	665.33	4	265.85	4
10	174.84	4	251.25	4	486.14	4	227.89	4
11	158.32	4	219.82	4	415.11	4	181.70	4
12	136.57	4	185.56	4	424.17	4	156.27	4
13	122.91	4	164.05	4	378.14	4	135.48	4
14	102.65	4	140.82	4	333.33	4	118.14	4
15	96.24	4	126.55	4	295.90	4	113.35	4
16	87.59	4	116.99	4	275.90	4	104.95	4

continued on next page

continued from previous page

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
18	–	–	–	–	271.37	4	90.37	4
20	–	–	–	–	197.60	4	73.32	4
25	–	–	–	–	143.68	4	51.32	4
30	–	–	–	–	116.83	4	40.09	4
35	–	–	–	–	91.75	4	31.55	4
40	–	–	–	–	77.20	4	26.25	4
45	–	–	–	–	64.24	4	21.98	4
50	–	–	–	–	56.51	4	18.47	4

Table C.28: Benchmark of NFA minimisation algorithms (10 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	623.44	4	1017.81	4	937.64	4	1137.00	4
3	339.84	4	647.03	4	554.17	4	544.95	4
4	221.06	4	231.10	4	446.92	4	259.33	4
5	142.61	4	157.72	4	475.85	4	161.00	4
6	109.05	4	112.60	4	319.69	4	113.47	4
7	85.74	4	86.43	4	221.11	4	90.15	4
8	71.13	4	70.05	4	199.40	4	68.46	4
9	54.91	4	57.41	4	176.05	4	54.07	4
10	46.26	4	46.89	4	149.86	4	47.61	4
11	37.62	4	39.11	4	133.48	4	39.51	4
12	31.22	4	33.91	4	127.35	4	33.16	4
13	27.83	4	29.10	4	112.86	4	28.97	4
14	24.23	4	26.78	4	104.37	4	25.00	4
15	21.10	4	23.17	4	94.77	4	22.67	4
16	18.92	4	20.06	4	90.16	4	19.50	4
18	–	–	–	–	84.83	4	16.19	4
20	–	–	–	–	74.12	4	13.43	4
25	–	–	–	–	57.37	4	9.07	4
30	–	–	–	–	47.00	4	6.41	4
35	–	–	–	–	39.24	4	4.72	4
40	–	–	–	–	33.15	4	3.80	4
45	–	–	–	–	29.21	4	3.05	4
50	–	–	–	–	25.30	4	2.54	4

Table C.29: Benchmark of NFA minimisation algorithms (20 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	179.09	4	170.98	4	229.62	4	204.98	4
3	93.17	4	97.83	4	155.64	4	98.75	4
4	57.33	4	59.15	4	107.71	4	56.12	4
5	39.14	4	38.29	4	89.05	4	38.63	4
6	27.38	4	28.09	4	72.51	4	27.72	4
7	20.54	4	20.83	4	62.12	4	21.01	4

continued on next page

continued from previous page

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
8	16.14	4	16.62	4	54.51	4	16.61	4
9	13.06	4	13.35	4	48.67	4	13.50	4
10	10.74	4	11.05	4	44.37	4	11.00	4
11	8.83	4	9.05	4	38.71	4	9.31	4
12	7.53	4	8.01	4	35.14	4	7.80	4
13	6.37	4	6.68	4	32.84	4	6.79	4
14	5.67	4	5.86	4	30.02	4	5.78	4
15	4.96	4	5.10	4	28.26	4	5.18	4
16	4.40	4	4.57	4	26.61	4	4.60	4
18	3.47	4	–	–	23.66	4	3.66	4
20	2.84	4	–	–	21.42	4	2.94	4
25	1.83	4	–	–	16.41	4	1.90	4
30	1.25	4	–	–	13.28	4	1.32	4
35	0.92	4	–	–	11.63	4	1.00	4
40	0.70	4	–	–	9.98	4	0.76	4
45	0.56	192	–	–	8.84	204	0.60	196
50	0.45	556	–	–	7.47	212	0.49	208

Table C.30: Benchmark of NFA minimisation algorithms (40 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	47.72	4	45.48	4	57.27	4	47.67	4
3	24.17	4	22.76	4	37.22	4	23.74	4
4	14.29	4	14.21	4	28.07	4	14.53	4
5	9.74	4	9.60	4	21.86	4	9.74	4
6	6.98	4	7.00	4	18.53	4	6.88	4
7	5.38	4	5.30	4	15.60	4	5.17	4
8	4.06	4	4.09	4	13.57	4	4.12	4
9	3.34	4	3.35	4	12.27	4	3.30	148
10	2.68	4	2.75	140	10.89	324	2.65	420
11	2.29	4	2.26	340	9.81	304	2.28	560
12	1.96	356	1.90	616	8.96	612	1.96	844
13	1.67	488	1.67	432	8.20	336	1.66	964
14	1.42	748	1.48	968	7.66	844	1.46	1068
15	1.25	1024	1.27	728	7.28	1432	1.25	1336
16	1.12	1052	1.12	1384	6.77	1208	1.13	1628
18	0.91	1772	–	–	5.91	1932	0.88	2248
20	0.71	2224	–	–	5.32	2204	0.72	2704
25	0.46	3392	–	–	4.28	3464	0.48	3848
30	0.33	4076	–	–	3.58	4052	0.33	4848
35	0.24	5556	–	–	2.99	5340	0.24	5572
40	0.18	7212	–	–	2.65	7060	0.19	7092
45	0.14	8684	–	–	2.36	7264	0.15	8104
50	0.11	10972	–	–	2.02	9300	0.12	9276

Table C.31: Benchmark of NFA minimisation algorithms (60 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	20.84	4	20.65	4	25.50	4	20.54	4
3	10.62	4	10.30	4	16.94	4	10.68	4
4	6.50	4	6.47	4	12.44	4	6.45	4
5	4.35	4	4.34	4	10.22	4	4.34	4
6	3.16	4	3.06	288	8.32	284	3.07	284
7	2.37	156	2.42	572	7.09	432	2.38	576
8	1.83	612	1.85	908	6.17	752	1.86	932
9	1.50	944	1.48	1084	5.43	1172	1.46	1268
10	1.19	1280	1.22	984	4.87	1512	1.24	1592
11	1.02	1780	1.04	2076	4.40	1436	1.02	2108
12	0.87	2092	0.84	1676	4.03	2200	0.86	2404
13	0.75	2024	0.76	2692	3.80	2224	0.75	2528
14	0.65	2556	0.64	2048	3.48	2364	0.65	2856
15	0.57	2828	0.57	3100	3.29	3276	0.56	2952
16	0.50	3196	0.50	3544	3.08	3064	0.50	3516
18	–	–	–	–	2.66	3784	0.41	3932
20	–	–	–	–	2.42	4700	0.32	5008
25	–	–	–	–	1.92	6324	0.21	5896
30	–	–	–	–	1.60	7240	0.15	7400
35	–	–	–	–	1.37	9236	0.11	11968
40	–	–	–	–	1.18	11588	0.08	14148
45	–	–	–	–	1.03	13408	0.06	15564
50	–	–	–	–	0.93	15276	0.05	17344

Table C.32: Benchmark of NFA minimisation algorithms (80 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	11.72	4	11.66	4	14.12	4	11.55	4
3	5.93	4	6.04	4	9.63	4	6.01	4
4	3.71	4	3.60	4	7.16	4	3.69	136
5	2.43	84	2.46	748	5.77	472	2.45	576
6	1.83	776	1.79	1404	4.66	1012	1.78	948
7	1.35	820	1.33	1476	3.99	1592	1.34	1920
8	1.04	1016	1.04	1792	3.46	1376	1.05	1408
9	0.84	2192	0.82	1628	3.14	1796	0.83	2556
10	0.71	2084	0.68	2652	2.76	2688	0.69	2624
11	0.58	2860	0.58	3464	2.52	2816	0.57	3192
12	0.49	2952	0.47	3428	2.33	4008	0.49	3440
13	0.41	3224	0.42	4452	2.15	3836	0.42	4524
14	0.37	3912	0.36	3884	2.01	4436	0.36	4904
15	0.31	4544	0.32	4416	1.87	5312	0.32	4288
16	0.28	4780	0.27	4460	1.73	5952	0.28	4584
18	0.23	6560	–	–	1.55	6112	0.23	5652
20	0.18	8828	–	–	1.37	7448	0.18	7864
25	0.12	10720	–	–	1.09	10916	0.12	11392

continued on next page

continued from previous page

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
30	0.08	15120	–	–	0.91	13304	0.08	13980
35	0.06	16836	–	–	0.77	14108	0.06	16532
40	0.04	20416	–	–	0.68	18824	0.04	21000
45	0.03	24240	–	–	0.59	20952	0.03	23700
50	0.03	28388	–	–	0.55	22312	0.02	25184

Table C.33: Benchmark of NFA minimisation algorithms (100 states).

k	**M**		**H**		**B**		**I**	
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.
2	7.54	4	7.28	4	9.20	4	7.46	4
3	3.84	164	3.88	672	6.13	324	3.75	384
4	2.37	692	2.31	1252	4.59	708	2.38	632
5	1.59	932	1.56	1268	3.57	1344	1.57	1508
6	1.14	2160	1.13	2088	2.98	2648	1.14	2496
7	0.87	2744	0.86	3452	2.44	3840	0.88	2888
8	0.68	3192	0.67	3432	2.27	3488	0.67	3500
9	0.54	3600	0.53	4940	2.00	3900	0.54	4928
10	0.45	4568	0.43	4512	1.77	4816	0.44	4572
11	0.37	5364	0.36	5160	1.65	5964	0.37	5508
12	0.31	6512	0.31	5752	1.51	7188	0.31	5872
13	0.27	7592	0.27	6712	1.36	7388	0.27	6788
14	0.23	7236	0.23	7772	1.29	8460	0.23	8132
15	0.20	10088	0.20	9216	1.19	8068	0.20	9964
16	0.18	10744	0.18	10996	1.10	8948	0.18	10916
18	0.14	12660	–	–	0.97	9848	0.14	12072
20	0.11	14304	–	–	0.87	11840	0.11	12700
25	0.07	20120	–	–	0.68	17692	0.07	18648
30	0.05	24512	–	–	0.58	21520	0.05	25716
35	0.04	27896	–	–	0.49	26824	0.03	26828
40	0.03	31060	–	–	0.43	27860	0.03	34508
45	0.02	37336	–	–	0.38	34160	0.02	36040
50	0.01	42280	–	–	0.35	39916	0.02	39924

Appendix D

Finite automata equivalence-testing

The tables on the following sections present experimental comparative results of several equivalence-testing algorithms for finite automata (cf. Chapter 9). The sampling and experimental study were conducted as described in Chapter 5, using datasets of 10 000 pairs of automata with n states over an alphabet of k symbols.

We present the results of only two algorithms which employ the usual minimisation-based method of testing DFAs' equivalence (cf. Section 9.1): DFA-MINIMISE-MOORE and DFA-MINIMISE-INCREMENTAL. We already know, from the experimental results presented on Chapter 8, that the fastest DFA minimisation algorithm is DFA-MINIMISE-INCREMENTAL, followed by DFA-MINIMISE-MOORE. Therefore, adding results for DFA-MINIMISE-HOPCROFT, DFA-MINIMISE-BRZOZOWSKI, or DFA-MINIMISE-WATSON would bring no surprise. For matters of space economy, the names of the algorithms are not used. Instead, the following mapping applies.

Algorithm Name	Column Id.
DFA-MINIMISE-MOORE	**M**
DFA-MINIMISE-BRZOZOWSKI	**B**
DFA-MINIMISE-INCREMENTAL	**I**
DFA-EQUIVALENT-HK-P	**HK**
NFA-EQUIVALENT-HKE-P	**HKe**
DFA-EQUIVALENT-HKS-P	**HKs**

Column *Perf.* refers to the performance of the given algorithm, i.e., the number of pairs of automata tested for equivalence per second. The memory usage, in *kilobytes*, is shown on column *Sp.*, and column *Iter.* presents the average number of recursive calls (meaningful only on some algorithms) necessary to decide the

equivalence of two automata.

D.1 ICDFAs

Table D.1: Benchmark of ICDFA equivalence-testing algorithms (10 states).

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	1526.7	4	1913.8	4	2989.5	4	18.9	7633.5	4	2.5	15384.6	4	2.5
3	1303.7	4	1527.8	4	2600.7	4	18.9	7168.4	4	2.6	13986.0	4	2.6
4	1256.2	4	1314.0	4	2247.1	4	19.0	6451.6	4	2.6	12195.1	4	2.6
5	1120.4	4	1298.7	4	2028.3	4	19.0	5698.0	4	2.6	10989.0	4	2.6
6	971.3	4	1197.6	4	1742.1	4	19.0	5494.5	4	2.7	9900.9	4	2.7
7	916.5	4	1074.1	4	1610.3	4	19.0	5050.5	4	2.7	9803.9	4	2.7
8	914.0	4	1189.7	4	1464.1	4	19.0	4566.2	4	2.7	8510.6	4	2.7
9	789.5	4	991.0	4	1347.7	4	19.0	4587.1	4	2.7	8968.6	4	2.7
10	813.3	4	988.1	4	1265.8	4	19.0	4504.5	4	2.7	8474.5	4	2.7
11	780.6	4	942.0	4	1196.1	4	19.0	4291.8	4	2.7	7751.9	4	2.7
12	763.3	4	913.6	4	1172.3	4	19.0	4201.6	4	2.7	7968.1	4	2.7
13	782.4	4	944.7	4	1066.0	4	19.0	3846.1	4	2.7	7299.2	4	2.7
14	747.1	4	834.0	4	1002.0	4	19.0	3656.3	4	2.7	7067.1	4	2.7
15	644.5	4	790.8	4	943.8	4	19.0	3642.9	4	2.7	6872.8	4	2.7
16	605.5	4	763.6	4	886.5	4	19.0	3565.0	4	2.7	6369.4	4	2.7
18	516.6	4	700.2	4	828.5	4	19.0	3333.3	4	2.7	6172.8	4	2.7
20	508.0	4	664.0	4	811.3	4	19.0	3262.6	4	2.6	5333.3	4	2.6
25	412.8	4	550.8	4	649.1	4	19.0	2656.0	4	2.7	4914.0	4	2.7
30	364.6	4	520.4	4	548.3	4	19.0	2607.5	4	2.6	4395.6	4	2.6
35	326.1	4	447.7	4	469.3	4	19.0	2185.7	4	2.6	4115.2	4	2.6
40	296.2	4	402.1	4	414.8	4	19.0	2070.3	4	2.6	3436.4	4	2.6
45	271.1	4	361.0	4	395.8	4	19.0	1937.9	4	2.6	3257.3	4	2.6
50	241.6	4	325.7	4	337.4	4	19.0	1853.5	4	2.6	3030.3	4	2.6

Table D.2: Benchmark of ICDFA equivalence-testing algorithms (20 states).

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	384.8	4	577.0	4	1628.6	4	38.9	5555.5	4	2.7	12269.9	4	2.7
3	374.1	4	512.4	4	1356.8	4	38.9	5012.5	4	2.8	10989.0	4	2.8
4	356.2	4	501.8	4	1156.0	4	38.9	4555.8	4	2.8	9569.3	4	2.8
5	349.2	4	475.8	4	1043.8	4	39.0	4484.3	4	2.9	8733.6	4	2.9
6	326.6	4	467.3	4	925.9	4	39.0	3913.8	4	2.9	7692.3	4	2.9
7	299.8	4	425.8	4	825.0	4	39.0	3642.9	4	2.9	7407.4	4	2.9
8	288.3	4	405.7	4	744.0	4	39.0	3430.5	4	3.0	6329.1	4	3.0
9	262.2	4	408.6	4	671.8	4	39.0	3361.3	4	2.9	6369.4	4	2.9
10	273.4	4	388.5	4	656.3	4	39.0	3164.5	4	2.9	5747.1	4	2.9
11	247.5	4	381.3	4	673.6	4	39.0	2808.9	4	3.0	5571.0	4	3.0
12	232.3	4	366.5	4	566.0	4	39.0	2923.9	4	3.0	5128.2	4	3.0
13	221.2	4	347.7	4	503.1	4	39.0	2721.0	4	3.0	4987.5	4	3.0

continued on next page

continued from previous page

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
14	217.1	4	343.2	4	487.6	4	39.0	2583.9	4	3.1	4773.2	4	3.1
15	206.8	4	333.8	4	447.2	4	39.0	2522.0	4	3.0	4566.2	4	3.0
16	197.6	4	313.0	4	433.1	4	39.0	2380.9	4	3.0	4405.2	4	3.0
18	198.8	4	289.9	4	396.6	4	39.0	2262.4	4	3.0	3913.8	4	3.0
20	181.3	4	273.5	4	363.1	4	39.0	2232.1	4	3.0	3430.5	4	3.0
25	160.6	4	237.1	4	291.0	4	39.0	1788.9	4	3.0	3058.1	4	3.0
30	143.0	4	207.5	4	245.7	4	39.0	1623.3	4	3.1	2610.9	4	3.1
35	128.8	4	193.1	4	211.9	4	39.0	1473.8	4	3.0	2328.2	4	3.0
40	116.0	4	176.4	4	185.9	4	39.0	1405.4	4	2.9	2162.1	4	2.9
45	102.6	4	159.9	4	166.2	4	39.0	1265.8	4	2.9	1926.7	4	2.9
50	95.5	4	147.5	4	147.1	4	39.0	1157.4	4	2.9	1795.3	4	2.9

Table D.3: Benchmark of ICDFA equivalence-testing algorithms (30 states).

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	164.0	4	288.6	28	1196.1	4	58.9	4273.5	4	2.8	11695.9	4	2.8
3	152.9	4	271.7	44	907.4	4	58.9	4016.0	4	2.9	9708.7	4	2.9
4	154.5	4	247.8	60	787.7	4	58.9	3603.6	4	2.8	8695.6	4	2.8
5	156.5	4	237.9	72	672.9	4	58.9	3508.7	4	2.9	7092.1	4	2.9
6	154.2	4	230.9	88	611.2	4	59.0	3062.7	4	2.9	7142.8	4	2.9
7	137.3	4	225.3	100	546.4	4	59.0	2849.0	4	3.1	6079.0	4	3.1
8	131.7	4	216.7	108	490.0	4	59.0	2635.0	4	3.0	5780.3	4	3.0
9	130.0	4	200.7	108	460.6	4	59.0	2531.6	4	3.1	5291.0	4	3.1
10	125.5	4	202.6	116	421.7	4	59.0	2386.6	4	3.0	4901.9	4	3.0
11	122.5	4	190.3	116	395.8	4	59.0	2195.3	4	3.1	4184.1	4	3.1
12	117.1	4	180.4	120	375.1	4	59.0	2227.1	4	3.1	4219.4	4	3.1
13	113.5	4	179.0	124	348.0	4	59.0	2148.2	4	3.1	3976.1	4	3.1
14	111.9	4	177.7	124	305.8	4	59.0	2040.8	4	3.1	3809.5	4	3.1
15	107.4	4	162.2	128	296.0	4	59.0	2081.1	4	3.1	3649.6	4	3.1
16	103.5	4	165.1	124	278.7	4	59.0	1921.2	4	3.2	3389.8	4	3.2
18	102.3	4	157.7	128	266.0	4	59.0	1776.1	4	3.2	3003.0	4	3.2
20	96.8	4	149.6	132	238.4	4	59.0	1666.6	4	3.2	2773.9	4	3.2
25	84.7	80	130.6	136	186.1	4	59.0	1336.0	4	3.3	2415.4	4	3.3
30	77.1	84	118.0	136	160.3	4	59.0	1240.6	4	3.2	2004.0	4	3.2
35	70.4	84	109.1	140	135.5	4	59.0	1116.0	4	3.2	1768.3	4	3.2
40	63.6	172	98.7	144	120.8	4	59.0	1036.2	4	3.2	1571.0	4	3.2
45	60.6	740	92.8	144	107.8	4	59.0	947.4	4	3.2	1451.3	4	3.2
50	57.3	3524	86.7	148	98.9	4	59.0	879.5	4	3.1	1333.3	4	3.1

Table D.4: Benchmark of ICDFA equivalence-testing algorithms (40 states).

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	83.7	4	164.6	100	885.3	4	78.9	3565.0	4	2.7	9900.9	4	2.7
3	93.3	4	162.0	172	715.5	4	78.9	3236.2	4	2.8	8298.7	4	2.8
4	82.2	4	148.0	188	667.7	4	78.9	2998.5	4	2.9	7352.9	4	2.9

continued on next page

continued from previous page

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
5	81.6	4	151.5	228	556.6	4	78.9	2785.5	4	3.0	6622.5	4	3.0
6	77.5	4	142.0	244	462.2	4	79.0	2383.7	4	2.8	5763.6	4	2.8
7	75.5	4	133.3	268	404.7	4	79.0	2336.4	4	3.0	5221.9	4	3.0
8	74.4	180	126.2	272	385.1	4	79.0	2259.8	4	3.1	4672.8	4	3.1
9	72.6	84	123.1	292	338.9	4	79.0	2185.7	4	3.2	4237.2	4	3.2
10	69.5	156	119.8	292	305.0	4	79.0	2016.1	4	3.1	3976.1	4	3.1
11	70.2	84	119.0	300	277.6	4	79.0	1937.9	4	3.1	3831.4	4	3.1
12	67.9	84	114.1	308	258.3	4	79.0	1785.7	4	3.2	3571.4	4	3.2
13	66.6	84	108.9	308	245.7	4	79.0	1760.5	4	3.3	3412.9	4	3.3
14	64.5	172	109.7	320	233.0	4	79.0	1755.9	4	3.1	3236.2	4	3.1
15	63.3	172	104.5	316	216.7	4	79.0	1597.4	4	3.2	2962.9	4	3.2
16	63.5	432	107.2	320	197.8	4	79.0	1545.5	4	3.2	2865.3	4	3.2
18	60.7	300	101.0	332	191.3	4	79.0	1495.8	4	3.3	2534.8	4	3.3
20	59.8	620	96.1	332	178.1	4	79.0	1398.6	4	3.3	2267.5	4	3.3
25	52.0	1900	84.2	340	138.5	4	79.0	1088.7	4	3.4	1978.2	4	3.4
30	47.1	2308	76.9	344	118.3	4	79.0	956.0	4	3.4	1626.0	4	3.4
35	44.2	3044	71.3	348	103.5	4	79.0	902.9	4	3.4	1445.0	4	3.4
40	41.8	3908	66.4	352	93.1	4	79.0	787.0	4	3.4	1270.6	4	3.4
45	37.9	4492	60.6	352	84.9	4	79.0	733.1	4	3.4	1145.4	4	3.4
50	36.7	5028	58.4	356	75.7	4	79.0	673.1	4	3.4	1065.5	4	3.4

Table D.5: Benchmark of ICDFA equivalence-testing algorithms (50 states).

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	55.4	4	107.1	276	677.0	4	98.9	3003.0	4	2.7	10101.0	4	2.7
3	54.8	4	103.4	296	565.7	4	98.9	2617.8	4	2.9	8333.3	4	2.9
4	54.5	256	100.2	388	468.7	4	98.9	2544.5	4	3.0	6825.9	4	3.0
5	53.0	4	95.6	412	413.3	4	99.0	2252.2	4	2.9	5665.7	4	2.9
6	51.5	172	89.9	440	367.4	4	99.0	2016.1	4	3.2	5050.5	4	3.2
7	50.1	176	87.1	468	324.8	4	99.0	1876.1	4	3.1	4347.8	4	3.1
8	49.8	428	86.1	500	282.5	4	99.0	1858.7	4	3.0	4008.0	4	3.0
9	47.9	624	83.3	512	261.0	4	99.0	1693.4	4	3.2	3816.7	4	3.2
10	48.0	84	82.0	524	232.0	4	99.0	1661.1	4	3.1	3590.6	4	3.1
11	47.8	816	80.0	540	211.3	4	99.0	1554.0	4	3.2	3294.8	4	3.2
12	46.1	1132	77.7	548	204.3	4	99.0	1540.8	4	3.2	3164.5	4	3.2
13	45.5	876	75.2	564	187.8	4	99.0	1492.5	4	3.5	2865.3	4	3.5
14	44.6	1708	74.2	564	184.1	4	99.0	1375.5	4	3.3	2832.8	4	3.3
15	43.8	4176	72.7	568	168.2	4	99.0	1386.0	4	3.3	2673.7	4	3.3
16	41.7	3920	73.5	572	160.3	4	99.0	1334.2	4	3.2	2518.8	4	3.2
18	42.0	2116	69.3	580	151.4	4	99.0	1208.4	4	3.5	2134.4	4	3.5
20	41.2	2524	67.4	588	140.9	4	99.0	1152.7	4	3.4	1962.7	4	3.4
25	36.2	4140	60.8	600	110.6	4	99.0	932.8	4	3.3	1662.5	4	3.3
30	34.2	4908	56.3	612	95.0	4	99.0	825.7	4	3.4	1356.8	4	3.4
35	31.7	6092	52.4	616	83.6	4	99.0	731.5	4	3.5	1240.6	4	3.5
40	29.5	13580	48.0	620	74.0	4	99.0	659.8	4	3.5	1090.5	4	3.5

continued on next page

continued from previous page

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
45	28.4	13556	45.3	620	67.6	4	99.0	608.0	4	3.5	1000.0	4	3.5
50	25.8	13536	41.8	624	59.8	4	99.0	583.6	4	3.4	885.7	4	3.4

Table D.6: Benchmark of ICDFA equivalence-testing algorithms (60 states).

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	37.9	428	77.7	352	564.0	4	118.9	2604.1	4	2.7	9661.8	4	2.7
3	38.5	172	74.3	476	464.2	4	118.9	2395.2	4	2.9	8000.0	4	2.9
4	38.2	684	71.6	588	382.9	4	118.9	1869.1	4	2.9	5128.2	4	2.9
5	37.1	876	68.3	640	338.0	4	118.9	1839.9	4	3.1	4545.4	4	3.1
6	36.1	1136	64.8	696	299.4	4	119.0	1632.6	4	3.0	4255.3	4	3.0
7	35.2	1132	63.5	724	273.5	4	119.0	1555.2	4	3.1	3937.0	4	3.1
8	36.6	84	63.2	756	242.0	4	119.0	1525.5	4	3.1	3629.7	4	3.1
9	36.0	328	61.8	792	230.1	4	119.0	1444.0	4	3.1	3289.4	4	3.1
10	35.8	2224	60.2	816	208.7	4	119.0	1335.1	4	3.2	2873.5	4	3.2
11	33.7	2664	59.0	820	196.0	4	119.0	1302.9	4	3.2	2836.8	4	3.2
12	33.0	2968	57.8	848	181.9	4	119.0	1221.7	4	3.2	2567.3	4	3.2
13	32.6	3608	56.7	848	167.1	4	119.0	1162.1	4	3.2	2333.7	4	3.2
14	32.5	3904	55.7	864	160.2	4	119.0	1155.4	4	3.3	2262.4	4	3.3
15	31.6	3896	54.3	868	152.5	4	119.0	1122.3	4	3.3	2127.6	4	3.3
16	30.7	3544	53.2	876	144.4	4	119.0	1079.9	4	3.3	2169.1	4	3.3
18	30.8	4284	50.4	896	132.7	4	119.0	1029.3	4	3.3	1897.5	4	3.3
20	29.5	4832	48.8	900	117.4	4	119.0	907.4	4	3.4	1707.9	4	3.4
25	26.9	6984	43.8	1056	95.2	4	119.0	729.9	4	3.4	1382.1	4	3.4
30	24.7	13624	40.9	1060	83.7	4	119.0	660.2	4	3.6	1164.1	4	3.6
35	23.9	9648	39.5	1108	71.7	4	119.0	601.3	4	3.4	1050.4	4	3.4
40	22.3	13592	36.0	1116	62.8	4	119.0	541.8	4	3.5	942.0	4	3.5
45	20.5	12540	33.5	1120	56.0	4	119.0	490.4	4	3.5	834.7	4	3.5
50	19.6	14104	32.3	1124	51.7	4	119.0	466.3	4	3.5	759.0	4	3.5

Table D.7: Benchmark of ICDFA equivalence-testing algorithms (70 states).

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	28.4	880	58.1	472	486.1	4	138.9	2237.1	4	2.7	8928.5	4	2.7
3	28.7	940	54.8	644	387.6	4	138.9	1796.9	4	2.8	5681.8	4	2.8
4	28.1	1392	52.1	780	336.9	4	138.9	1721.1	4	3.0	5012.5	4	3.0
5	27.4	2160	50.6	896	291.0	4	138.9	1662.5	4	3.0	4291.8	4	3.0
6	26.5	2220	49.2	960	262.7	4	138.9	1461.9	4	3.0	3690.0	4	3.0
7	26.0	3900	48.6	1024	233.5	4	139.0	1400.5	4	3.1	3466.2	4	3.1
8	26.3	3244	47.8	1088	210.8	4	139.0	1303.7	4	3.0	3194.8	4	3.0
9	25.5	3700	46.8	1108	191.5	4	139.0	1273.8	4	3.1	2994.0	4	3.1
10	25.0	3540	45.0	1136	181.9	4	139.0	1189.7	4	3.0	2840.9	4	3.0
11	24.6	4576	44.3	1180	168.8	4	139.0	1097.0	4	3.1	2567.3	4	3.1

continued on next page

continued from previous page

k	M		I		HK			HKe			HKs		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
12	24.4	3976	43.2	1180	157.0	4	139.0	1104.3	4	3.2	2352.9	4	3.2
13	23.4	5080	42.4	1328	146.3	4	139.0	1071.8	4	3.1	2309.4	4	3.1
14	23.9	10640	41.8	1224	139.2	4	139.0	1063.8	4	3.2	2100.8	4	3.2
15	23.3	5672	41.6	1224	124.9	4	139.0	952.3	4	3.2	2018.1	4	3.2
16	22.7	6264	40.6	1228	122.1	4	139.0	931.5	4	3.3	1851.8	4	3.3
18	22.6	13560	38.9	1256	108.2	4	139.0	907.0	4	3.4	1614.2	4	3.4
20	22.4	13272	37.6	1268	100.4	4	139.0	817.3	4	3.6	1532.5	4	3.6
25	20.5	13700	34.6	1868	81.4	4	139.0	657.4	4	3.5	1237.6	4	3.5
30	18.6	23584	31.7	1908	71.5	4	139.0	589.7	4	3.6	1079.3	4	3.6
35	18.0	14144	30.0	1856	59.8	4	139.0	533.0	4	3.6	953.2	4	3.6
40	16.8	16248	27.9	1988	54.7	4	139.0	478.3	4	3.7	824.4	4	3.7
45	16.0	32320	27.0	2000	49.8	4	139.0	439.5	4	3.7	744.3	4	3.7
50	14.6	32288	25.0	2008	43.3	4	139.0	406.5	4	3.7	679.5	4	3.7

Table D.8: Benchmark of ICDFA equivalence-testing algorithms (80 states).

k	M		I		HK			HKe			HKs		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	19.8	4780	42.3	696	407.6	4	158.9	1792.1	4	2.8	6309.1	4	2.8
3	20.0	3860	41.2	940	338.6	4	158.9	1639.3	4	2.9	5154.6	4	2.9
4	19.6	5548	39.6	1080	293.9	4	158.9	1550.3	4	2.9	4484.3	4	2.9
5	20.4	6572	39.3	1200	257.0	4	158.9	1499.2	4	3.0	4115.2	4	3.0
6	19.9	6828	38.5	1296	224.5	4	159.0	1282.0	4	3.0	3533.5	4	3.0
7	19.5	7264	37.8	1372	204.9	4	159.0	1243.7	4	3.0	3289.4	4	3.0
8	19.0	7260	36.7	1576	185.0	4	159.0	1153.4	4	3.1	2915.4	4	3.1
9	18.6	6768	35.9	1752	165.9	4	159.0	1121.0	4	3.2	2680.9	4	3.2
10	18.4	8800	35.3	1652	157.1	4	159.0	1040.5	4	3.3	2433.0	4	3.3
11	18.2	7500	34.9	1952	147.8	4	159.0	1025.6	4	3.3	2309.4	4	3.3
12	17.9	9864	33.9	1712	136.2	4	159.0	981.3	4	3.2	2173.9	4	3.2
13	17.6	16184	33.3	1860	129.4	4	159.0	932.8	4	3.2	1956.9	4	3.2
14	17.3	15664	32.0	1888	122.2	4	159.0	913.2	4	3.3	1876.1	4	3.3
15	17.6	15680	32.3	1900	112.3	4	159.0	879.1	4	3.3	1760.5	4	3.3
16	16.9	14968	31.8	2180	108.1	4	159.0	871.4	4	3.3	1652.8	4	3.3
18	17.4	16204	30.7	1940	96.2	4	159.0	798.4	4	3.5	1490.3	4	3.5
20	17.1	15524	30.4	2088	89.5	4	159.0	745.7	4	3.4	1373.6	4	3.4
25	15.5	17124	27.2	2916	73.1	4	159.0	569.9	4	3.5	1165.5	4	3.5
30	14.7	18932	26.0	2924	58.0	4	159.0	524.2	4	3.6	985.2	4	3.6
35	13.7	35460	24.0	3012	53.6	4	159.0	473.2	4	3.6	852.8	4	3.6
40	12.9	34656	22.8	3008	48.8	4	159.0	433.4	4	3.6	764.8	4	3.6
45	12.2	49468	21.8	3016	41.9	4	159.0	389.9	4	3.6	672.9	4	3.6
50	11.7	34608	20.5	3080	39.5	4	159.0	361.2	4	3.8	602.9	4	3.8

Table D.9: Benchmark of ICDFA equivalence-testing algorithms (90 states).

k	M		I		HK			HKe			HKs		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	17.0	5580	34.6	880	357.9	4	178.9	1510.5	4	2.7	6191.9	4	2.7

continued on next page

continued from previous page

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
3	16.6	6448	34.1	1152	303.9	4	178.9	1355.0	4	2.8	5154.6	4	2.8
4	16.1	7024	32.5	1364	261.4	4	178.9	1329.7	4	2.9	4166.6	4	2.9
5	15.9	7644	31.6	1580	228.1	4	179.0	1260.2	4	3.0	3663.0	4	3.0
6	15.4	7624	30.2	2048	197.8	4	179.0	1086.9	4	3.0	3273.3	4	3.0
7	15.1	8732	29.3	2296	180.4	4	179.0	1064.9	4	3.1	2949.8	4	3.1
8	15.0	9708	29.2	2380	161.5	4	179.0	1006.5	4	3.1	2688.1	4	3.1
9	14.8	9724	28.9	2416	149.6	4	179.0	978.4	4	3.1	2469.1	4	3.1
10	14.8	16500	28.0	2612	134.9	4	179.0	926.3	4	3.1	2347.4	4	3.1
11	14.7	11380	27.2	2908	125.5	4	179.0	878.3	4	3.1	2132.1	4	3.1
12	14.4	12188	27.2	2944	117.4	4	179.0	846.7	4	3.2	1990.0	4	3.2
13	14.2	12740	26.6	2840	109.6	4	179.0	831.9	4	3.2	1892.1	4	3.2
14	13.8	12948	26.2	2868	104.9	4	179.0	804.1	4	3.3	1745.2	4	3.3
15	14.0	11004	25.9	2884	98.0	4	179.0	779.4	4	3.3	1629.9	4	3.3
16	13.5	14656	25.3	2912	93.7	4	179.0	751.3	4	3.4	1526.7	4	3.4
18	13.4	23908	24.6	2948	87.4	4	179.0	713.7	4	3.3	1391.7	4	3.3
20	13.3	17016	24.2	3104	77.9	4	179.0	654.2	4	3.5	1261.8	4	3.5
25	12.4	34832	21.7	4020	64.3	4	179.0	509.2	4	3.5	1072.9	4	3.5
30	11.6	34820	20.3	4100	53.7	4	179.0	444.6	4	3.7	909.5	4	3.7
35	10.8	34748	19.4	4136	45.7	4	179.0	408.7	4	3.6	759.8	4	3.6
40	10.3	28752	18.4	4152	42.4	4	179.0	380.3	4	3.6	683.2	4	3.6
45	9.8	34156	17.0	4208	36.3	4	179.0	345.5	4	3.8	615.5	4	3.8
50	9.1	37292	16.3	4272	34.5	4	179.0	326.3	4	3.9	548.6	4	3.9

Table D.10: Benchmark of ICDFA equivalence-testing algorithms (100 states).

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	11.9	6828	26.9	1016	327.4	4	198.9	1337.7	4	2.8	5813.9	4	2.8
3	12.6	6760	27.0	1428	271.0	4	198.9	1319.2	4	2.9	5076.1	4	2.9
4	12.6	10876	25.9	2092	231.9	4	198.9	1207.7	4	3.0	4192.8	4	3.0
5	12.5	9324	26.0	2180	202.5	4	199.0	1132.5	4	3.0	3636.3	4	3.0
6	12.2	5680	24.4	2916	177.2	4	199.0	993.0	4	3.0	3115.2	4	3.0
7	12.0	10560	24.2	3136	159.8	4	199.0	956.0	4	3.1	2805.0	4	3.1
8	12.1	15636	23.7	3228	149.8	4	199.0	925.9	4	3.1	2531.6	4	3.1
9	11.9	12036	23.4	3760	135.0	4	199.0	872.2	4	3.2	2306.8	4	3.2
10	11.8	15588	23.2	3500	125.4	4	199.0	827.8	4	3.2	2053.3	4	3.2
11	11.9	6400	22.2	3696	114.0	4	199.0	793.6	4	3.2	1919.3	4	3.2
12	11.7	14380	22.2	3712	108.1	4	199.0	773.0	4	3.3	1853.5	4	3.3
13	11.4	15496	21.9	3744	102.3	4	199.0	738.8	4	3.3	1675.0	4	3.3
14	11.4	16756	21.6	4092	96.1	4	199.0	736.9	4	3.3	1603.8	4	3.3
15	11.2	17500	21.6	4208	89.3	4	199.0	710.2	4	3.3	1539.6	4	3.3
16	11.2	17636	21.2	3976	85.2	4	199.0	679.3	4	3.4	1409.4	4	3.4
18	11.1	19840	20.2	4004	75.2	4	199.0	624.2	4	3.5	1287.0	4	3.5
20	10.6	20804	19.7	4176	69.3	4	199.0	576.3	4	3.4	1104.3	4	3.4
25	9.9	34860	18.0	5148	56.8	4	199.0	451.6	4	3.5	913.6	4	3.5
30	9.4	34764	17.2	5196	47.1	4	199.0	406.3	4	3.6	799.0	4	3.6

continued on next page

continued from previous page

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
35	9.0	33860	16.0	5260	41.9	4	199.0	374.0	4	3.7	697.5	4	3.7
40	8.5	37796	15.5	4916	37.8	4	199.0	346.8	4	3.7	625.1	4	3.7
45	8.2	41772	14.7	4980	33.7	4	199.0	319.7	4	3.7	560.6	4	3.7
50	7.7	72696	13.6	5484	30.7	4	199.0	296.2	4	4.0	493.7	4	4.0

Table D.11: Benchmark of ICDFA equivalence-testing algorithms (1000 states).

k	**M**		**I**		**HK**			**HKe**			**HKs**		
	Perf.	Sp.	Perf.	Sp.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.	Perf.	Sp.	Iter.
2	0	722008	0.1	106048	19.8	4	1998.9	38.4	4	1.9	1003.0	4	1.9
3	0	709360	0.1	89628	17.2	4	1998.9	33.0	4	2.0	742.6	4	2.0
5	0	750956	0.1	106156	14.1	4	1998.9	34.3	4	2.0	471.1	4	2.0

D.2 NFAs

D.2.1 Transition density 0.1

Table D.12: Benchmarks of NFA equivalence-testing algorithms (10 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	113.25	4	2469.13	4	2.65
3	16.83	156	1996.00	4	2.87
4	4.53	268	1647.44	4	3.27
5	1.82	80	1336.89	4	3.87
6	0.92	940	1113.58	4	4.27
7	0.53	96	896.05	4	4.95
8	0.34	9672	708.71	4	6.30
9	0.23	8900	653.16	4	6.52
10	0.17	14772	491.40	4	8.47
11	0.12	13364	453.30	4	8.84
12	0.09	14896	353.73	4	11.10
13	0.08	14580	305.34	4	12.03
14	0.06	17444	279.87	4	12.25
15	0.05	16864	247.64	4	12.83
16	0.04	18524	223.26	4	13.76
18	0.03	19232	168.71	4	17.86
20	0.02	26580	144.21	4	18.64
25	0.01	35560	86.72	4	25.80
30	0	42660	59.74	4	32.48

continued on next page

continued from previous page

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
35	–	–	39.72	4	43.22
40	–	–	31.49	4	47.65
45	–	–	23.40	4	57.10
50	–	–	20.38	4	61.40

Table D.13: Benchmarks of NFA equivalence-testing algorithms (20 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	77.89	4	230.52	4	35.86
3	3.36	16	50.57	4	123.15
4	0.09	79936	18.15	4	274.71
5	0	1191140	8.72	684	482.91
6	–	–	5.06	88	717.31
7	–	–	3.22	2476	966.78
8	–	–	2.41	10348	1155.18
9	–	–	1.69	15468	1468.38
10	–	–	1.31	6572	1713.98
11	–	–	0.97	8620	2082.52
12	–	–	0.83	20336	2283.89
13	–	–	0.70	20012	2474.38
14	–	–	0.57	24492	2785.20
15	–	–	0.48	23508	3114.74
16	–	–	0.42	28272	3310.10
18	–	–	0.32	29104	3882.34
20	–	–	0.25	51052	4518.78
25	–	–	0.16	75124	5613.25
30	–	–	0.11	100684	7112.44
35	–	–	0.09	172056	7614.04
40	–	–	0.06	120372	9103.66
45	–	–	0.05	152428	10382.44
50	–	–	0.03	146560	12245.75

Table D.14: Benchmarks of NFA equivalence-testing algorithms (40 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	–	–	266.52	4	13.82
3	–	–	68.39	4	43.58
4	–	–	23.41	4	101.38
5	–	–	9.69	4	202.95
6	–	–	4.77	4	349.39
7	–	–	2.62	84	535.15
8	–	–	1.54	48	811.78
9	–	–	1.01	5736	1109.62
10	–	–	0.62	5872	1517.80

continued on next page

continued from previous page

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
11	–	–	0.45	3544	2001.46
12	–	–	0.33	29436	2541.30
13	–	–	0.25	53512	3132.36
14	–	–	0.20	32172	3616.60
15	–	–	0.15	66804	4418.74
16	–	–	0.11	87092	5521.91
18	–	–	0.07	135696	7287.14
20	–	–	0.04	153936	9727.51
25	–	–	0.02	391644	18368.53
30	–	–	0.01	502552	26557.71
35	–	–	0	1446000	71860.49
40	–	–	0	3368664	120343.03
45	–	–	0	1097796	82113.93
50	–	–	0	522788	90174.10

Table D.15: Benchmarks of NFA equivalence-testing algorithms (60 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	–	–	265.60	4	8.68
3	–	–	87.94	4	21.01
4	–	–	34.05	4	44.80
5	–	–	15.66	4	78.65
6	–	–	7.97	4	127.22
7	–	–	4.51	4	195.49
8	–	–	2.89	4	272.07
9	–	–	1.89	268	377.61
10	–	–	1.28	148	495.55
11	–	–	0.91	1240	632.84
12	–	–	0.67	760	792.35
13	–	–	0.51	1328	957.59
14	–	–	0.40	1448	1163.66
15	–	–	0.32	1584	1361.83
16	–	–	0.25	1620	1622.38
18	–	–	0.14	108	2232.30
20	–	–	0.09	1092	2919.56
25	–	–	0.04	6088	4733.13
30	–	–	0.02	6696	8086.07
35	–	–	0.01	65480	11338.04
40	–	–	0	181164	16446.70
45	–	–	0	192196	22649.48
50	–	–	0	66524	25910.24

Table D.16: Benchmarks of NFA equivalence-testing algorithms (80 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	–	–	202.79	4	6.58

continued on next page

continued from previous page

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
3	–	–	78.75	4	15.33
4	–	–	33.91	4	29.85
5	–	–	16.34	4	51.31
6	–	–	8.35	4	81.92
7	–	–	4.95	4	118.32
8	–	–	3.08	4	164.64
9	–	–	2.08	4	217.10
10	–	–	1.45	80	278.06
11	–	–	1.00	108	359.78
12	–	–	0.76	240	433.89
13	–	–	0.57	180	526.17
14	–	–	0.44	248	635.16
15	–	–	0.34	84	769.42
16	–	–	0.29	1604	881.25
18	–	–	0.18	1672	1145.70
20	–	–	0.12	1392	1520.26
25	–	–	0.05	3344	2604.95
30	–	–	0.02	5508	4381.41
35	–	–	0.01	28108	5615.67
40	–	–	0.01	26392	8421.14
45	–	–	0	32748	10223.66
50	–	–	0	14964	14861.83

Table D.17: Benchmarks of NFA equivalence-testing algorithms (100 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	–	–	170.21	4	5.19
3	–	–	83.62	4	10.11
4	–	–	41.03	4	18.15
5	–	–	20.93	4	29.11
6	–	–	11.48	4	44.18
7	–	–	7.12	4	62.12
8	–	–	4.56	172	85.53
9	–	–	3.13	384	110.08
10	–	–	2.22	64	140.00
11	–	–	1.62	696	173.13
12	–	–	1.23	932	206.52
13	–	–	0.96	864	244.11
14	–	–	0.75	1256	289.14
15	–	–	0.61	1356	332.79
16	–	–	0.49	1636	381.38
18	–	–	0.33	2132	487.25
20	–	–	0.24	2732	618.20
25	–	–	0.11	6616	1013.27
30	–	–	0.06	7652	1505.29

continued on next page

continued from previous page

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
35	–	–	0.03	9940	2229.77
40	–	–	0.02	14524	3066.48
45	–	–	0.01	16020	3989.93
50	–	–	0.01	15020	5296.07

D.2.2 Transition density 0.5

Table D.18: Benchmarks of NFA equivalence-testing algorithms (10 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	542.88	4	1945.52	4	3.26
3	381.97	4	1275.51	4	4.78
4	220.84	4	873.36	4	6.66
5	159.69	4	610.50	4	8.73
6	121.16	4	430.84	4	10.92
7	102.25	4	315.05	4	12.96
8	83.74	4	239.75	4	14.98
9	68.39	4	183.62	4	17.50
10	60.43	4	153.18	4	19.52
11	51.68	4	128.13	4	21.53
12	46.16	4	108.02	4	23.21
13	40.26	4	91.34	4	25.72
14	36.23	4	83.61	4	27.09
15	32.72	4	70.35	4	30.26
16	29.29	4	65.01	4	31.19
18	25.04	4	53.81	4	34.45
20	20.63	4	42.35	4	39.56
25	14.29	4	27.74	4	47.77
30	10.56	4	20.05	4	54.90
35	7.66	4	15.39	4	62.10
40	6.07	96	11.78	4	70.68
45	4.52	68	9.59	4	76.01
50	3.66	240	7.96	4	81.83

Table D.19: Benchmarks of NFA equivalence-testing algorithms (20 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	190.25	4	1798.56	4	2.58
3	105.87	4	942.50	4	3.35
4	77.53	4	664.01	4	4.10
5	62.79	4	544.66	4	4.95
6	49.90	4	341.64	4	6.08
7	42.07	4	264.48	4	6.90

continued on next page

continued from previous page

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
8	36.24	4	223.81	4	8.02
9	31.45	4	181.48	4	8.89
10	28.34	4	138.14	4	10.20
11	25.16	4	113.97	4	11.40
12	22.87	4	93.15	4	12.87
13	20.49	4	80.93	4	14.01
14	18.66	4	65.18	4	15.22
15	17.16	4	56.51	4	16.83
16	15.86	4	49.82	4	17.95
18	13.81	4	42.52	4	20.18
20	12.10	4	32.87	4	23.54
25	8.94	4	20.10	4	29.73
30	7.03	4	14.07	4	36.18
35	5.81	164	10.29	4	42.06
40	4.75	172	8.16	4	46.00
45	3.89	496	6.73	268	51.99
50	3.36	708	5.49	84	56.44

Table D.20: Benchmarks of NFA equivalence-testing algorithms (40 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	45.45	4	773.99	4	2.49
3	29.09	4	370.23	4	2.99
4	21.96	4	212.85	4	3.50
5	17.52	4	159.92	4	4.04
6	14.39	4	126.53	4	4.55
7	12.38	4	104.43	4	5.11
8	11.01	4	87.10	4	5.57
9	9.59	4	73.64	4	6.05
10	8.56	4	64.69	4	6.51
11	7.62	4	52.10	4	6.92
12	6.90	4	45.87	4	7.48
13	6.12	168	41.37	172	8.07
14	5.71	16	36.87	340	8.63
15	5.40	532	33.63	712	9.03
16	5.21	652	30.63	908	9.57
18	4.38	624	25.24	1208	10.52
20	3.92	884	21.09	1752	11.68
25	3.05	1852	14.75	2648	14.16
30	2.54	3040	10.80	3260	16.77
35	2.10	3564	8.54	4208	18.84
40	1.79	4444	6.54	4512	21.85
45	1.57	6268	5.51	5552	23.75
50	1.39	5756	4.45	6128	27.18

Table D.21: Benchmarks of NFA equivalence-testing algorithms (60 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	19.65	4	222.17	4	2.50
3	13.25	4	142.53	4	3.00
4	9.71	4	99.47	4	3.51
5	7.74	160	78.36	168	4.02
6	6.45	1008	61.84	856	4.48
7	5.45	1372	50.84	1264	5.03
8	4.72	1768	37.37	1884	5.54
9	4.14	2060	31.34	2328	6.04
10	3.67	2512	27.39	2676	6.41
11	3.35	2352	23.92	3220	7.02
12	3.04	3044	21.54	3728	7.49
13	2.82	3392	18.82	4224	8.11
14	2.62	3744	17.19	4636	8.42
15	2.45	4260	15.73	5064	8.98
16	2.27	4488	14.25	5516	9.62
18	1.95	5472	12.09	6624	10.48
20	1.82	6200	10.20	7028	11.52
25	1.44	8680	7.23	9996	13.93
30	1.13	11196	5.57	11216	16.34
35	0.99	16048	4.30	16152	19.10
40	0.85	16760	3.52	18536	21.06
45	0.74	18828	2.86	20940	23.41
50	0.68	24080	2.42	23848	26.10

Table D.22: Benchmarks of NFA equivalence-testing algorithms (80 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	11.04	4	129.08	4	2.52
3	7.31	4	85.82	4	2.99
4	5.52	596	62.51	576	3.49
5	4.44	1164	48.76	1284	4.01
6	3.60	2176	33.73	1848	4.49
7	3.16	1956	28.26	2596	5.00
8	2.66	3140	23.63	3240	5.53
9	2.41	3284	20.71	4016	5.94
10	2.16	3828	17.54	4600	6.58
11	1.93	4740	15.71	5192	6.96
12	1.78	5344	14.11	5580	7.55
13	1.61	5700	12.59	5924	8.07
14	1.51	5904	11.44	6488	8.55
15	1.40	7616	10.04	7516	9.13
16	1.33	8380	9.28	7600	9.47
18	1.16	8568	7.86	8960	10.54
20	1.04	10116	6.80	10416	11.31
25	0.83	14560	4.85	14188	14.02

continued on next page

continued from previous page

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
30	0.67	18188	3.66	18396	16.49
35	0.56	23308	2.92	20820	19.09
40	0.50	24356	2.34	24560	21.61
45	0.44	28892	1.92	28256	24.34
50	0.39	31812	1.63	32956	26.64

Table D.23: Benchmarks of NFA equivalence-testing algorithms (100 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	7.34	4	89.03	4	2.53
3	4.88	632	59.09	628	3.02
4	3.64	1372	43.04	1032	3.50
5	2.82	1872	29.62	2056	4.00
6	2.33	3392	23.33	3192	4.51
7	2.02	4036	19.23	3228	5.06
8	1.71	4752	16.30	4956	5.49
9	1.53	5664	14.31	5460	6.04
10	1.40	6036	12.61	5692	6.50
11	–	–	11.17	6360	6.99
12	–	–	9.80	6812	7.55
13	–	–	8.73	8496	8.04
14	1.00	10524	8.05	8656	8.57
15	0.91	9540	7.11	8816	9.16
16	0.87	11612	6.65	9880	9.54
18	0.75	13196	5.54	11548	10.60
20	0.67	13412	4.81	13424	11.43
25	0.52	20232	3.47	20288	13.96
30	0.44	23988	2.67	22612	16.48
35	0.38	29604	2.13	29352	18.88
40	0.32	33748	1.71	32644	21.61
45	0.29	40292	1.42	38108	23.97
50	0.25	41936	1.19	43544	26.77

D.2.3 Transition density 0.8

Table D.24: Benchmarks of NFA equivalence-testing algorithms (10 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	385.50	4	1968.50	4	2.45
3	243.66	4	1449.27	4	2.97
4	227.01	4	1089.32	4	3.43
5	159.71	4	904.15	4	3.99
6	118.52	4	712.75	4	4.45

continued on next page

continued from previous page

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
7	103.79	4	620.73	4	4.98
8	93.43	4	521.64	4	5.45
9	83.64	4	430.66	4	5.88
10	76.01	4	372.16	4	6.36
11	65.73	4	328.19	4	6.82
12	61.75	4	284.25	4	7.28
13	53.97	4	253.10	4	7.73
14	51.03	4	213.94	4	8.25
15	48.49	4	191.57	4	8.77
16	45.50	4	168.63	4	9.31
18	41.45	4	148.58	4	10.16
20	36.64	4	123.77	4	11.13
25	27.73	4	82.31	4	13.47
30	22.75	4	58.74	4	15.80
35	19.01	4	43.83	4	18.16
40	16.07	4	33.95	4	20.56
45	14.54	4	27.47	4	22.84
50	12.67	4	22.20	4	25.07

Table D.25: Benchmarks of NFA equivalence-testing algorithms (20 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	106.80	4	1150.74	4	2.51
3	73.71	4	880.28	4	2.98
4	52.31	4	616.52	4	3.48
5	43.56	4	516.52	4	3.95
6	34.99	4	337.60	4	4.45
7	30.15	4	271.73	4	4.99
8	26.95	4	259.20	4	5.51
9	23.38	4	196.61	4	5.98
10	21.20	4	169.89	4	6.43
11	–	–	131.01	4	6.91
12	17.47	4	108.94	4	7.53
13	–	–	95.61	4	7.94
14	–	–	87.98	4	8.48
15	13.85	4	78.50	4	8.88
16	–	–	72.85	4	9.41
18	11.48	4	61.92	4	10.59
20	10.38	4	52.60	4	11.42
25	8.11	4	33.24	4	13.99
30	6.41	4	24.08	4	16.45
35	5.50	308	18.74	180	18.62
40	4.82	480	14.08	272	21.58
45	4.14	232	11.46	84	24.19
50	3.80	808	9.57	520	26.26

Table D.26: Benchmarks of NFA equivalence-testing algorithms (40 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	28.95	4	457.87	4	2.53
3	18.31	4	222.22	4	3.01
4	13.81	4	147.66	4	3.49
5	10.72	4	115.59	4	3.96
6	9.40	4	84.40	4	4.52
7	7.88	148	71.12	136	5.04
8	6.82	504	58.11	440	5.50
9	6.04	720	49.30	676	6.03
10	5.62	1188	43.59	1236	6.63
11	4.85	1372	35.07	1400	7.07
12	4.50	1268	31.21	1696	7.50
13	4.19	1920	27.97	1992	7.98
14	3.94	2428	25.00	2448	8.42
15	3.56	1924	22.70	2820	8.98
16	3.35	2476	20.80	3120	9.56
18	2.95	3200	16.52	3828	10.53
20	2.63	4112	13.72	4664	11.64
25	2.09	5884	9.83	6096	14.01
30	1.75	6952	7.17	7384	16.62
35	1.50	8248	5.57	8552	19.02
40	1.29	10480	4.38	10188	21.76
45	1.15	11440	3.60	11816	24.05
50	1.00	14664	2.98	12900	26.31

Table D.27: Benchmarks of NFA equivalence-testing algorithms (60 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	12.93	4	169.11	4	2.50
3	8.33	4	104.78	4	3.04
4	6.36	288	74.14	4	3.49
5	4.98	468	57.88	572	4.00
6	4.20	660	44.68	1180	4.56
7	3.50	1524	38.03	1664	4.97
8	3.04	1492	28.43	2112	5.51
9	2.66	2144	24.07	2304	5.98
10	2.46	3044	20.97	3088	6.58
11	2.23	3500	18.31	3708	7.07
12	2.06	3112	16.52	4144	7.44
13	1.87	3816	14.24	4212	7.99
14	1.78	4080	12.95	4844	8.42
15	1.61	4772	11.84	5312	9.00
16	1.52	5712	10.86	5780	9.46
18	1.36	6100	9.05	6356	10.45
20	1.19	7844	7.64	7212	11.50
25	0.95	10512	5.43	10848	13.88

continued on next page

continued from previous page

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
30	0.78	12912	4.07	12440	16.92
35	0.67	17604	3.09	17668	19.32
40	0.59	18264	2.56	20140	21.34
45	0.52	21444	2.09	21968	24.01
50	0.46	23228	1.77	22088	26.41

Table D.28: Benchmarks of NFA equivalence-testing algorithms (80 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	7.08	4	95.44	4	2.50
3	4.76	240	63.68	348	3.02
4	3.51	384	45.76	784	3.52
5	2.84	1136	35.93	1600	4.02
6	2.34	1936	24.87	2376	4.53
7	2.01	3080	20.94	2532	4.98
8	1.76	3200	17.90	3160	5.50
9	1.54	3876	15.41	3612	5.99
10	1.40	4444	13.15	4520	6.46
11	1.28	5660	11.76	5052	6.92
12	1.17	5844	10.31	5644	7.47
13	1.07	6348	9.33	5620	8.05
14	0.99	6620	8.58	6444	8.44
15	0.94	8228	7.66	7724	9.00
16	0.86	9244	7.00	7352	9.45
18	0.75	8636	5.88	8976	10.56
20	0.66	11080	4.93	12432	11.58
25	0.54	16940	3.58	14296	13.95
30	0.45	18420	2.67	18656	16.55
35	0.38	23144	2.13	22052	18.84
40	0.32	28588	1.69	28112	21.73
45	0.29	30648	1.38	32112	23.91
50	0.26	34552	1.19	34796	26.05

Table D.29: Benchmarks of NFA equivalence-testing algorithms (100 states).

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
2	4.62	84	64.27	252	2.46
3	3.07	1052	42.01	892	3.03
4	2.26	2400	30.87	2256	3.49
5	1.83	3076	20.71	3100	3.98
6	1.49	4396	16.89	4000	4.49
7	1.28	6436	13.89	4824	5.10
8	1.09	7348	11.60	5872	5.56
9	0.99	7516	10.22	6616	5.98

continued on next page

continued from previous page

k	**B**		**HKe**		
	Perf.	Sp.	Perf.	Sp.	Iter.
10	0.89	8544	8.96	8008	6.44
11	–	–	7.96	8452	6.98
12	–	–	6.91	9960	7.61
13	–	–	6.25	10168	8.14
14	0.65	13104	5.51	12636	8.56
15	0.58	13288	5.07	11972	9.20
16	0.56	15532	4.67	14532	9.51
18	0.49	17508	3.92	15908	10.40
20	0.44	18956	3.31	18024	11.83
25	0.35	26408	2.38	25320	14.07
30	0.28	31236	1.86	31508	16.06
35	0.24	36904	1.44	35496	19.16
40	0.21	41860	1.16	40472	21.65
45	0.19	46796	0.93	47192	24.05
50	0.17	56212	0.79	54524	26.73

Appendix E

Minimal ICDFA density

The values presented on Table E.1 are the *exact percentages* of minimal ICDFAs for the given number of states n and alphabet size k.

k/n	2	3	4	5	6	7
2	50%	59%	66%	72%	75%	77%
3	50%	65%	78%	86%	–	–
4	50%	69%	75%	–	–	–
5	50%	66%	–	–	–	–

Table E.1: Exact percentages of minimal ICDFAs.

Table E.2 and Table E.3 show the *probabilities* of obtaining a minimal ICDFA when using a uniform random generator. These were calculated while performing the experimental tests for Chapter 8. The sampling and experimental study were conducted as described in Chapter 5, using datasets of 20 000 elements.

Table E.2: Probability of randomly generating a minimal ICDFA.

k/n	5	10	20	30	40	50
2	0.59750	0.69035	0.75055	0.77290	0.77725	0.79095
3	0.73145	0.86240	0.93405	0.95430	0.96610	0.97335
4	0.77310	0.89505	0.94830	0.96595	0.97480	0.97890
5	0.79545	0.90135	0.95160	0.96525	0.97405	0.98030
6	0.80085	0.90325	0.94960	0.96635	0.97710	0.98075
7	0.80475	0.89945	0.94945	0.96475	0.97515	0.97960
8	0.79815	0.89990	0.94955	0.96685	0.97465	0.98130

continued on next page

continued from previous page

k/n	5	10	20	30	40	50
9	0.79385	0.89995	0.95105	0.96620	0.97480	0.97910
10	0.80060	0.90155	0.95115	0.96490	0.97455	0.98100
11	0.79720	0.89970	0.94890	0.96780	0.97625	0.98010
12	0.80125	0.90000	0.94920	0.96585	0.97615	0.97940
13	0.80250	0.90295	0.94790	0.96590	0.97625	0.97820
14	0.79715	0.90195	0.95220	0.96785	0.97545	0.98045
15	0.80865	0.89565	0.94985	0.96450	0.97395	0.97965
16	0.80000	0.89910	0.94985	0.96660	0.97560	0.98030
18	0.80120	0.89885	0.94940	0.96655	0.97400	0.97805
20	0.80210	0.90120	0.94800	0.96455	0.97585	0.97960
25	0.79890	0.89910	0.95060	0.96610	0.97355	0.97945
30	0.80280	0.90030	0.94875	0.96745	0.97280	0.98035
35	0.80330	0.90210	0.95195	0.96790	0.97550	0.98060
40	0.80205	0.90125	0.95140	0.96540	0.97605	0.98015
45	0.79610	0.90230	0.94950	0.96645	0.97465	0.98070
50	0.80065	0.89730	0.94805	0.96935	0.97665	0.97945

Table E.3: Probability of randomly generating a minimal ICDFA (cont.).

k/n	60	70	80	90	100	1 000
2	0.78465	0.79775	0.79280	0.80155	0.79640	0.85590
3	0.97690	0.98285	0.98480	0.98670	0.98730	0.99980
4	0.98470	0.98520	0.98615	0.98915	0.99000	–
5	0.98275	0.98630	0.98785	0.98905	0.98980	0.97230
6	0.98305	0.98540	0.98580	0.98840	0.98925	–
7	0.98285	0.98700	0.98750	0.98900	0.98915	–
8	0.98215	0.98645	0.98710	0.98740	0.99045	–
9	0.98505	0.98540	0.98640	0.98940	0.99035	–
10	0.98295	0.98640	0.98625	0.98905	0.98940	–
11	0.98125	0.98515	0.98770	0.98825	0.99070	–
12	0.98470	0.98540	0.98735	0.98900	0.98985	–
13	0.98340	0.98615	0.98730	0.98945	0.98935	–
14	0.98360	0.98520	0.98585	0.98910	0.99045	–
15	0.98250	0.98700	0.98715	0.99055	0.99010	–
16	0.98390	0.98735	0.98580	0.98730	0.98970	–
18	0.98305	0.98410	0.98785	0.98825	0.99005	–
20	0.98270	0.98535	0.98845	0.98955	0.99030	–
25	0.98335	0.98670	0.98765	0.98885	0.98990	–
30	0.98260	0.98670	0.98855	0.98930	0.98985	–
35	0.98530	0.98585	0.98780	0.98900	0.98930	–
40	0.98520	0.98625	0.98760	0.98920	0.99045	–
45	0.98350	0.98655	0.98755	0.98795	0.99055	–
50	0.98420	0.98490	0.98755	0.98760	0.98810	–

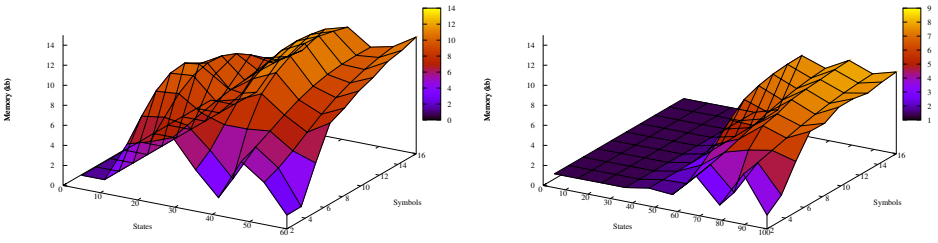
Appendix F

Subset construction

The following graphs, on Figures F.1 and F.2, reflect the behaviour of the subset construction when applied to samples of 20 000 NFAs as described on Chapter 5. We collected experimental data for two parameters: space usage of the method, and the average size of the resulting DFAs.

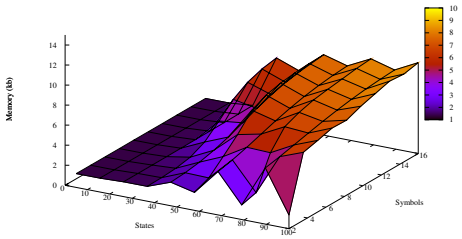
Due to the large range of values — applying the subset construction to some samples with 40 states and a transition density $t = 0.1$ results in DFAs with almost 10 000 states, while no equivalent sample with transition density $t = 0.8$ produces DFAs with more than 51 states — the graphs are presented in a logarithmic scale.

It is clear, on all cases, that the samples with transition density $t = 0.1$ easily produce combinatorial explosions. Datasets of NFAs with 20 states, for example, sometimes result in DFAs with more than 13 000 states, consuming more than 100 MB of memory. NFAs with transition densities $t = 0.5$ or $t = 0.8$ produce rather similar results, and nothing that suggests the possibility of a combinatorial explosion.



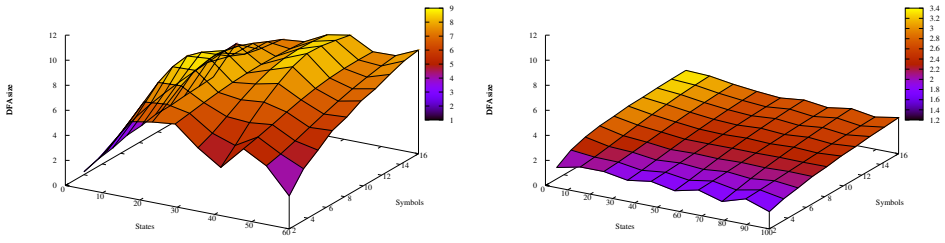
(a) Transition density $t = 0.1$

(b) Transition density $t = 0.5$

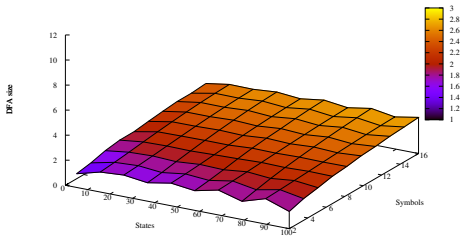


(c) Transition density $t = 0.8$

Figure F.1: Subset construction: space used to obtain an equivalent DFA.



(a) Transition density $t = 0.1$ (b) Transition density $t = 0.5$



(c) Transition density $t = 0.8$

Figure F.2: Sub set construction: average size of the equivalent DFAs.

Bibliography

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison Wesley, 1974.

[2] M. Almeida, N. Moreira, and R. Reis. Enumeration and generation with a string automata representation. *Theoretical Computer Science*, 387(2):93–102, 2007. Special issue "Selected papers of DCFS 2006".

[3] M. Almeida, N. Moreira, and R. Reis. Antimirov and Mosses's rewrite system revisited. In O. Ibarra and B. Ravikumar, editors, *Proceedings of the 13th International Conference on Implementation and Application of Automata, CIAA 2008*, number 5448 in Lecture Notes on Computer Science, pages 46–56. Springer, 2008.

[4] Marco Almeida. `psmon`, 2009–2010. `http://savannah.nongnu.org/projects/psmon/`.

[5] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996.

[6] Valentin M. Antimirov and Peter D. Mosses. Rewriting extended regular expressions. In G. Rozenberg and A. Salomaa, editors, *Developments in Language Theory*, pages 195–209, 1993.

[7] Manuel Baclet and Claire Pagetti. Around hopcroft's algorithm. In Oscar H. Ibarra and Hsu-Chun Yen, editors, *Implementation and Application of Automata*, volume 4096 of *Lecture Notes on Computer Science*, pages 114–125. Springer, 2006.

[8] Frédérique Bassino, Julien David, and Cyril Nicaud. On the Average Complexity of Moore's State Minimization Algorithm. In Susanne Albers and Jean-Yves Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science STACS 2009 Proceedings of the 26th Annual Symposium on the Theoretical Aspects of Computer Science*, pages 123–134, Freiburg Germany, 2009. IBFI Schloss Dagstuhl.

[9] Frédérique Bassino, Julien David, and Cyril Nicaud. REGAL: a library to randomly and exhaustively generate automata. In Jan Holub and Jan Žd'árek, editors, *Implementation and Application of Automata*, volume 4783 of *Lecture Notes on Computer Science*, pages 303–305. Springer, 2007.

[10] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.

[11] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In J. Fox, editor, *Proceedings of the Symposium on Mathematical Theory of Automata*, volume 12 of *MRI Symposia Series*, pages 529–561, New York, NY, April 24–26 1962. Polytechnic Press of the Polytechnic Institute of Brooklyn, Brooklyn, NY.

[12] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the Association for Computing Machinery*, 11(4):481–494, October 1964.

[13] Anne Brüggemann-Klein and Derick Wood. Deterministic regular languages. In *STACS 92*, pages 173–184. Springer-Verlag, 1992.

[14] J.-M. Champarnaud, G. Hansel, T. Paranthoën, and D. Ziadi. Random generation models for NFAs. *J. Autom. Lang. Comb.*, 9(2-3):203–216, 2004.

[15] J.-M. Champarnaud, A. Khorsi, and T. Paranthoën. Split and join for minimizing: Brzozowski's algorithm. In M. Balík and M. Simánek, editors, *Proceedings of the Prague Stringology Conference, PSC'02*, Research report DC-2002-03, pages 96–104. Czech Technical University of Prague, 2002.

[16] Chia-Hsiang Chang. *From Regular Expressions to DFA's Using Compressed NFA's*. PhD thesis, Courant Institute of Mathematical Sciences, New York University, October 1992. `http://cs.nyu.edu/web/Research/Theses/chang_chia-hsiang.pdf`.

[17] Chia-Hsiang Chang and Robert Paige. From Regular Expressions to DFA's Using Compressed NFA's. In *CPM '92: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, pages 90–110. Springer-Verlag, 1992.

[18] William G. Cochran. *Sampling Techniques*. John Wiley & Sons, third edition, 1977.

[19] J. H. Conway. *Regular Algebra and Finite Machines*. Mathematical Series. Chapman and Hall, 11 New Fetter Lane, London, EC4, 1971.

[20] Thomas H. Cormen, Charles. E. Leiserson, Ronald. L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2003.

[21] Keith Ellul, Bryan Krawetz, Jeffrey Shallit, and Ming wei Wang. Regular expressions: New results and open problems. *J. Autom. Lang. Comb.*, 9:233–256, September 2004.

[22] **FAdo**: Tools for formal languages manipulation. `http://www.ncc.up.pt/FAdo`.

[23] Free Software Foundation. GNU `grep`. `http://www.gnu.org/software/grep/`.

[24] Free Software Foundation. GNU `time`. `http://www.gnu.org/software/time/`.

[25] Python Software Foundation. *Python 2.6*, 2006–2010. `http://docs.python.org/release/2.6.6/`.

[26] Python Software Foundation. `Python` programming language, 2006–2010. `http://www.python.org`.

[27] A. Ginzburg. A procedure for checking equality of regular expressions. *Journal of the Association for Computing Machinery*, 14(2):355–362, April 1967. `http://cs.simons-rock.edu/cmpt320/ginzburg.pdf`.

[28] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1, 1961. http://stacks.iop.org/0036-0279/16/i=5/a=A01.

[29] David Gries. Describing an algorithm by Hopcroft. Technical Report TR 72-151, Cornell University, Ithaca, New York 14850, December 1972.

[30] Ralph P. Grimaldi. *Discrete and Combinatorial Mathematics*. Addison Wesley, fourth edition, 1999.

[31] J. E. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report TR 71–114, University of California, Berkeley, California, November 1971.

[32] J. E. Hopcroft and J. D. Ullman. A linear list merging algorithm. Technical Report TR 71–111, Cornell University, November 1971.

[33] J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM J. Comput.*, 2(4):294–303, 1973.

[34] John Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, New York, 1971.

[35] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, second edition, 2000. International Edition.

[36] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, first edition, April 1979.

[37] D. A. Huffman. The synthesis of sequential switching circuits. Technical Report 274, Massachusetts Institute of Technology, January 1954. Reprinted from the Journal of the Franklin Institute Vol. 257, Nos. 3 and 4, March and April, 1954.

[38] D. A. Huffman. The synthesis of sequential switching circuits. *Journal of Symbolic Logic*, 20(1):69–70, 1955.

[39] S. C. Kleene. Representation of events in nerve nets and finite automata. Technical Report RM-704, U. S. Air Force, December 1951.

[40] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison Wesley, second edition, 1973.

[41] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison Wesley, first edition, 1973.

[42] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison Wesley, second edition, 1981.

[43] Timo Knuutila. Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 250(1–2):333–363, 2001.

[44] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Inf. Comput.*, 110:366–390, May 1994.

[45] Dexter C. Kozen. *Automata and Computability*. Undergraduate Texts in Computer Science. Springer, 1997.

[46] Jonathan Lee and Jeffrey Shallit. Enumerating regular expressions and their languages. In Michael Domaratzki, Alexander Okhotin, Kai Salomaa, and Sheng Yu, editors, *Implementation and Application of Automata*, volume 3317 of *Lecture Notes in Computer Science*, pages 2–22. Springer Berlin / Heidelberg, 2005.

[47] Ted Leslie. Efficient approaches to subset construction. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, 1995. `http://www.csd.uwo.ca/Research/grail/.papers/subset.ps`.

[48] Ondřej Lhoták. A general data structure for efficient minimization of deterministic finite automata. Course project, CS 662 Formal Languages and Parsing, December 2000.

[49] Sylvain Lombardy and Jacques Sakarovitch. VAUCANSON-G. `http://www-igm.univ-mlv.fr/~lombardy/Vaucanson-G/`.

[50] Harry G. Mairson. Generating words in a context-free language uniformly at random. *Information Processing Letters*, 49(2):95–99, 1994.

[51] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9(1):39–47, 1960.

[52] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, 34:129–153, 1956. Princeton University Press.

[53] Gonzalo Navarro. Nr-grep: A fast and flexible pattern matching tool. *Software Practice and Experience (SPE*, 31:2001, 2000.

[54] Gonzalo Navarro. Approximate regular expression searching with arbitrary integer weights. *Nordic J. of Computing*, 11:356–373, December 2004.

[55] Gonzalo Navarro and Mathieu Raffinot. Compact DFA representation for fast regular expression search. In *Proceedings of the 5th International Workshop on Algorithm Engineering*, WAE '01, pages 1–12, London, UK, 2001. Springer-Verlag.

[56] C. Nicaud. *Étude du comportement en moyenne des automates finis et des langages rationnels*. PhD thesis, Université de Paris 7, 2000.

[57] PostgreSQL object-relational DBMS, 2008–2010. `http://www.postgresql.org`.

[58] Debian Project. Debian gnu/linux, 2010. `http://www.debian.org`.

[59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[60] R. Reis, N. Moreira, and M. Almeida. On the representation of finite automata. In C. Mereghetti, B. Palano, G. Pighizzini, and D.Wotschke, editors, *Proceedings of the 7th International Workshop on Descriptional Complexity of Formal Systems, DCFS 2005*, pages 269–276, 2005.

[61] R. Reis, N. Moreira, and M. Almeida. On the representation of finite automata. Technical Report DCC-2005-04, DCC - FC & LIACC, Universidade do Porto, April 2005. `http://www.dcc.fc.up.pt/Pubs/TR05/dcc-2005-04.ps.gz`.

[62] G. Rozenberg and A. Salomaa, editors. *Word, Language, Grammar*, volume 1 of *Handbook of Formal Languages*. Springer, 1997.

[63] A. Salomaa. *Theory of Automata*, volume 100 of *International Series of Monographs in Pure and Applied Mathematics*. Pergamon Press, first edition, 1969.

[64] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966.

[65] Jeffrey Shallit. Regular expressions enumeration and state complexity. In *Proceedings of the 9th International Conference on Implementation and Application of Automata, CIAA 2004*, 2004. `http://www.cs.uwaterloo.ca/~shallit/Talks/ciaa7.ps`.

[66] Jeffrey Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2008.

[67] Graphviz — Graph Visualization Software. *The DOT Language*. `http://www.graphviz.org/doc/info/lang.html`.

[68] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 1–9, New York, NY, USA, 1973. ACM.

[69] Deian Tabakov and Moshe Y. Vardi. Experimental evaluation of classical automata constructions. In *In LPAR 2005, LNCS 3835*, pages 396–411. Springer, 2005.

[70] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the Association for Computing Machinery*, 22(2):215–225, April 1975.

[71] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[72] Bruce W. Watson. An incremental DFA minimization algorithm. In L. Karttunen, K. Koskenniemi, and G. van Noord, editors, *Proceedings of the Second International Workshop on Finite-State Methods in Natural Language Processing*, Helsinki, Finland, August 2001.

[73] Bruce W. Watson and Jan Daciuk. An efficient DFA minimization algorithm. *Natural Language Engineering*, 9(1):49–64, 2003.

[74] Bruce William Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* PhD thesis, Eindhoven University of Technology, 1995.

[75] Derick Wood. *Theory of Computation*. John Wiley & Sons, 1987.

[76] Sheng Yu. *Word, Language, Grammar*, volume 1 of *Handbook of Formal Languages*, chapter Regular Languages, pages 41–110. Springer, 1997.

Author Index

W

Y

Algorithm Index

Subject Index