UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Constructing Minimal Acyclic Deterministic Finite Automata

Bruce William Watson

bruce@bruce-watson.com
FASTAR Research Group
www.fastar.org

Professor Extraordinary
Computer Science
University of Pretoria

Full Professor
Information Systems
Stellenbosch University

Submitted in partial fulfillment of the requirements for the degreee Philosophiae Doctor (Computer Science) in the Faculty of Engineering, Built Environment and Information Technology

University of Pretoria, Pretoria, South Africa

November 2010

© University of Pretoria

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Copyright © 2010 by the University of Pretoria, South Africa.

All rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced, in any form or by any means, including but not limited to photocopy, photograph, magnetic or other record, without prior agreement and written permission of the copyright holder.

FASTAR logo designed by STARLIT — `www.star-lit.co.za`

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Constructing Minimal Acyclic Deterministic Finite Automata

Author:
Prof.Dr. Bruce W. WATSON

Supervisor:
Prof.Dr. Derrick G. KOURIE

University of Pretoria, Pretoria, South Africa

To my family, especially Liam and Keira (I almost finished this thesis as you arrived)

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Abstract

This thesis is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Ph.D) in the FASTAR group of the Department of Computer Science, University of Pretoria, South Africa. I present a number of algorithms for constructing minimal acyclic deterministic finite automata (MADFAs), most of which I originally derived/designed or co-discovered. Being acyclic, such automata represent finite languages and have proven useful in applications such as spell-checking, virus-searching and text indexing. In many of those applications, the automata grow to billions of states, making them difficult to store without using various compression techniques — the most important of which is *minimization*. Results from the late 1950's show that minimization yields a unique automaton (for a given language), and later results show that minimization of acyclic automata is possible in time linear in the number of states. These two results make for a rich area of algorithmics research; automata and algorithmics research are relatively old fields of computing science and the discovery/invention of new algorithms in the field is an exciting result.

I present both incremental and nonincremental algorithms. With nonincremental techniques, the unminimized acyclic deterministic finite automaton (ADFA) is first constructed and then minimized. As mentioned above, the unminimized ADFA can be very large indeed — often even too large to fit within the virtual memory space of the computer. As a result, incremental techniques for minimization (i.e. the ADFA is minimized *during* its construction) become interesting. Incremental algorithms frequently have *some* overhead: if the unminimized ADFA fits easily within physical memory, it may still be faster to use nonincremental techniques.

The presentation used in this thesis has a few unusual characteristics:

- Few other presentations follow a correctness-by-construction style for presenting and deriving algorithms. The presentations given here include correctness arguments or sketches thereof.
- The presentation is *taxonomic* — emphasizing the similarities and differences between the algorithms at a fundamental level.
- While it is possible to present these algorithms in a formal-language-theoretic setting, this thesis remains somewhat closer to the actual implementation issues.
- In several chapters, new algorithms and interesting new variants of existing algorithms are presented.
- It gives new presentations of many existing algorithms — all in a common format with common examples.
- There are extensive links to the existing literature.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Contents

Abstract **i**

Preface **vi**

1 Introduction **1**
1.1 Problem statement . . . 1
1.2 To the reader . . . 2
1.3 Related work and a short history . . . 2
1.4 Links to the literature . . . 4
1.5 Future work . . . 4

2 Preliminaries **7**
2.1 General definitions . . . 7
2.2 Algorithm presentation . . . 8
2.3 Strings and languages . . . 8
2.4 Automata . . . 10
2.5 Minimality of automata . . . 17
2.5.1 Computing E . . . 21
2.5.2 Computing $\neg$E in ADFAs . . . 21
2.5.3 Computing $E(p, q)$ pointwise . . . 21
2.5.4 A more efficient computation of E . . . 21

3 A MADFA-construction skeleton **23**
3.1 Specific instantiations . . . 24
3.1.1 Choosing a structural invariant . . . 24
3.1.2 Function add_word . . . 25
3.1.3 Function cleanup . . . 25
3.2 Commentary . . . 25

4 Trie intermediate ADFA **27**
4.1 Procedure add_word_T . . . 27
4.1.1 Adding only prefix words . . . 28
4.1.2 Adding a nonprefix word in a trie . . . 29
4.2 Procedure cleanup_T . . . 30
4.2.1 Selecting $N : N \subseteq T \wedge N \neq \emptyset \wedge \text{Inequiv}(N)$ . . . 31
4.2.1.1 Selecting a single state . . . 32
4.2.1.2 Selecting a path of states . . . 34
4.2.2 Selecting $N : N \subseteq T \wedge N \neq \emptyset \wedge \text{Pairwise_inequiv}(D, N)$ . . . 35
4.3 An example . . . 38

4.4 Time and space performance . . . 41
4.4.1 Improvements . . . 41
4.5 Commentary . . . 41

5 Arbitrary intermediate ADFA **43**
5.1 Procedure add_word_N . . . 43
5.2 Procedure cleanup_N . . . 46
5.3 Time and space performance . . . 46
5.4 Commentary . . . 46

6 Minimal intermediate ADFA **49**
6.1 Procedure add_word_I . . . 49
6.1.1 Recursive helper procedure visit_min . . . 51
6.2 Procedure cleanup_I . . . 54
6.3 An example . . . 54
6.4 Time and space performance . . . 59
6.5 Commentary . . . 59

7 Reversed trie intermediate ADFA **61**
7.1 Procedure add_word_R . . . 61
7.2 Procedure cleanup_R . . . 61
7.3 An example . . . 62
7.4 Time and space performance . . . 63
7.5 Commentary . . . 63

8 Avoiding cloning while adding words **65**

9 Words in lexicographic order **67**
9.1 Procedure add_word_S . . . 67
9.1.1 A minor problem in using visit_min . . . 70
9.2 Procedure cleanup_S . . . 71
9.3 An example . . . 71
9.4 Time and space performance . . . 73
9.4.1 Improvements . . . 73
9.5 Commentary . . . 74

10 Minimizing depth layers **75**
10.1 Procedure add_word_D . . . 75
10.2 Procedure cleanup_D . . . 77
10.3 An example . . . 77
10.4 Time and space performance . . . 81
10.5 Commentary . . . 81

11 Minimizing semi-incrementally **83**
11.1 Procedure add_word_W . . . 83
11.1.1 Procedure semi_min . . . 85
11.1.1.1 Refining $S'_{11.1.1}$ . . . 86
11.1.1.2 Refining $S''_{11.1.1}$ . . . 86
11.1.1.3 A final version of semi_min . . . 87

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

11.2 Procedure $\mathsf{cleanup}_W$ . . . 88
11.3 An example . . . 88
11.4 Time and space performance . . . 92
11.5 Commentary . . . 92

Bibliography **93**

Index **99**

Colophon **101**

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Ph.D) in the FASTAR group of the Department of Computer Science, University of Pretoria, South Africa. I present a number of algorithms for constructing minimal acyclic deterministic finite automata, most of which I personally originally derived/designed or discovered. In certain applications, these automata are a compact representation of a large finite number of words — for example in a spelling checker or a computer virus detector. As such, efficient new algorithms also have commercial value in addition to their intrinsic scientific value.

In the time leading up to the presentation of my first Ph.D [Wat95] in Eindhoven, I was actively implementing and using finite automata and transducers in industry (primarily in the fields of compilation and text indexing). During and subsequent to my Eindhoven Ph.D research, I have remained very active in the field in several ways:

- I have served as a consultant on automata construction, minimization, and implementation, particularly for applications in computational linguistics, network security and text indexing.

- From 1996, I served on the program committee of the *International Workshop on Implementing Automata*. In 2000, the workshop was upgraded to the international *Conference on Implementations and Applications of Automata*. During 23–25 July 2001, the sixth such event was chaired by me and hosted at the University of Pretoria, in South Africa [WW01a, WW01b]. As a side-effect of that conference, I co-edited a special issue of the journal *Theoretical Computer Science* [WW04]. In 2007, I presented one of the keynote talks at the conference, in Prague.

- From its origins in 1996 onwards, I have been involved (served in the program committee) of the annual *Prague Stringology Conference* (formerly a workshop).

- Also since 1996, I have actively been involved in *Finite State Methods in Natural Language Processing* workshop (not held annually). It serves as a venue for work at the intersection of computing science and linguistics — including the work presented in this thesis.

- The first annual FASTAR symposium (on algorithmics, data-structures and applications of finite state techniques) was held in Eindhoven in 2004. Having an algorithmics and data-structures slant, the FASTAR conference complemented the Conference on Implementations and Applications of Automata.

- I have actively published work in this field. For an overview, see the reference list at the end of this thesis.

Although the common thread in my research relates to automata, a more recent specific thread is the work on constructing acyclic deterministic finite automata. It is this work that is reported in this thesis.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Acknowledgements Particular thanks go to Derrick Kourie for being an outstanding promotor, supervisor and motivator, and to Loek Cleophas for his detailed feedback. For their technical inputs over several years, I am grateful to (in random order)

- The FASTAR and ESPRESSO research groups, especially Fritz Venter, Ernest Ketcha, Tinus Strauss, Lorraine Liang, ...
- The original *Program Implementation* (later *Software Construction*) group in Eindhoven, especially Frans Kruseman Aretz (my first promotor), Kees Hemerik (my first copromotor), Gerard Zwaan, Michiel Frishert, Tom Verhoeff, Rik van Geldrop, ...
- The Prague Stringology group, particularly Bořivoj Melichar, Jan Holub, František Franěk, Jan Janoušek, Jan Žďárek, ...
- The StringMasters, including Lynette van Zijl, Bill Smyth, Costas Iliopoulos, Maxime Crochemore, Jackie Daykin, Brink van der Merwe, Laurent Mouchard, ...
- Computational linguists, especially Andre Kempe, Thomas Hanneforth, Johannes Bubenzer, Lauri Karttunen, Anssi Yli-Jyrä, Kimmo Koskenniemi, Jan Daciuk, Krister Lindén, Tamás Gaál, Stoyan Mihov, ...
- Researchers in related fields, such as Blaine Kubesh, Jeffrey Shallit, John Brzozowski, Richard Watson, Stefan Gruner, Judith Bishop, Roelf van den Heever, ...

(My apologies for anyone I have overlooked.) Last but certainly not least, thanks go to my entire family for their support — in particular to Nanette who (as always) *also* proofread this thesis.

Chapter 1

Introduction

In this thesis, we present algorithms for building minimal acyclic deterministic finite automata, also known as MADFAs. By their acyclic nature, they represent finite languages and are therefore useful in applications such as storing words for spell-checking (among other computational linguistics applications), computer and biological virus searching, program verification, text indexing and searching, and XML tag lookup. In each of these applications, the automata can grow extremely large (sometimes having more than 10^9 states) and are difficult to store without first applying a minimization procedure. The specific applications are not discussed further here, but can be found in literature on stringology [Smy03, CR03, CR94], computational linguistics [JM00], information retrieval [BYRN99], data-structures [GBY91], computational biology [Gus97, Pev00] and compilers [ALSU07].

We consider both incremental and nonincremental algorithms. With nonincremental techniques, the unminimized acyclic deterministic finite automaton (ADFA) is first constructed and then minimized. As mentioned above, the unminimized ADFA can be very large indeed — often even too large to fit within the virtual memory space of the computer. As a result, incremental techniques for minimization (i.e. the ADFA is minimized *during* its construction) become interesting. Incremental algorithms frequently have *some* overhead: if the unminimized ADFA fits easily within physical memory, it may still be faster to use nonincremental techniques. On the other hand, with very large ADFAs, using an incremental technique may be the *only* option.

Although our main computational problem in this thesis is 'building MADFAs', we often refer to a particular subproblem: modify a MADFA to additionally accept a specific word w. By default, when we refer to an 'algorithm', we assume it to solve one of these two problems.

1.1 Problem statement

Although a precise definition will be given later, we can informally state the problem:

> Given an alphabet Σ and some finite set of words $W \subset \Sigma^*$, construct a minimal acyclic deterministic finite automaton M which accepts exactly the words in W.

Although all of the algorithms in this thesis are presented as imperative programs, it is clear that they could also have been derived using other paradigms — for example as functional programs. Imperative programs are used to ease the translation directly to C++.

In this thesis, we are primarily interested in acyclic deterministic finite automata. The algorithms can be extended to work with acyclic deterministic *transducers* (automata with outputs on transitions).

1.2 To the reader

This thesis presents several original research contributions. As a result, there are several compelling reasons to read it:

- Very few other presentations follow a correctness-by-construction style for presenting and deriving algorithms. The presentations given here include correctness arguments or sketches thereof.
- The presentation is *taxonomic* — emphasizing the similarities and differences between the algorithms at a fundamental level.
- While it is possible to present these algorithms in a formal-language-theoretic setting (as was done in [Wat02b]), this thesis remains somewhat closer to the actual implementation issues.
- In several chapters, new algorithms and interesting new variants of existing algorithms are presented.
- It gives new presentations of many existing algorithms — all in a common format with common examples.
- There are extensive links to the existing literature.

The structure and style of this thesis deserves some explanation:

- In several cases, the simplest algorithms are presented in the main part of the chapter, while refinements for a practical implementation are only mentioned at the end of the chapter.
- For most of the algorithms presented in this thesis, we only mention the running times and memory requirements. No attempt is made to rigorously derive them, as this is presented elsewhere in the literature.
- There is no single chapter with global conclusions. In a sense, the algorithms (and their corresponding derivations) are themselves the 'conclusions' of this work. In each chapter, there are some closing comments.

1.3 Related work and a short history

A great deal of practical work on constructing and minimizing ADFAs has been done. Unfortunately, much of the research is of a proprietary nature and thus forms part of the folklore of automata algorithmics. Some of the algorithms may even have been known for some years and remained unpublished.

In the early-1990s, Dominique Revuz derived one of the first known efficient (linear in time and space) ADFA minimization algorithms [Rev91, Rev92]. The primary algorithm presented by Revuz uses an ordering of the words to quickly compress the endings of the words within the dictionary. Further work by Revuz has also yielded algorithms which correspond rather closely to some of the algorithms in this thesis[1]. A version of Revuz's algorithm appears in §4.2. Recent derivations by Johannes Bubenzer (in Thomas Hanneforth's group at Universität Potsdam) have yielded efficient

[1]All minimization algorithms show strong similarities, as can be seen from the taxonomy in [Wat95]. The subtle differences between the algorithms can lead to domain-specific performance advantages for each algorithm.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

new algorithms bearing a resemblance to Revuz's [Bub11]; that work is not explicitly included in this thesis.

By the mid-1990s, several groups were working independently on incremental algorithms — most of which are the same or very similar. In Greece, Sgarbas *et al* derived an algorithm[2] and presented it in [SFK95]. In Japan, Park *et al* were also deriving a related generalized algorithm [PAMS94]. At Marne-la-Valée in France, a group (including Revuz) was continuing work on related algorithms. In 1996, Richard E. Watson and I (both working at Ribbit Software Systems Inc. in Canada) completed work on the implementation of a generalized incremental algorithm for a division of Novell Corporation. (The Novell group using this particular algorithm was later acquired by Lernout & Hauspie, the now-defunct Belgian speech technology company.) Unlike many of the other derivations of related algorithms, our implementation also provides facilities for removing words from the language accepted by the automaton, while maintaining minimality[3]. Owing to its commercial value, the algorithm was not published at that time. Also in 1996–1997, Jan Daciuk was completing his Ph.D. research (independently) involving the generalized algorithm. In addition, Daciuk derived a new incremental algorithm which adds the words in lexicographic order. (This is known as the *sorted* algorithm.) Daciuk approached us during his literature search and we decided to combine efforts, publishing the generalized and sorted algorithms at the *First Workshop on Finite State Machines in Natural Language Processing* [DWW98] in Ankara, Turkey. Several papers (including ours) at that workshop were invited for submission to the *Journal of Computational Linguistics*. While typesetting that paper, we discovered the work of Stoyan Mihov, then a Ph.D. student in Bulgaria who had also derived the sorted algorithm, publishing it as [Mih99a]. Again, we combined efforts — with Daciuk, Mihov, Watson & Watson publishing the journal article [DMWW00]. Daciuk and Mihov also published the algorithms in their dissertations as [Dac98] and [Mih99b] respectively. Independently, in the field of program verification, Gerard Holzmann and Anuj Puri [HP98] discovered a restricted form of the algorithm, in which all words accepted by the automaton are the same length. Also independently, Marcin Ciura and Sebastian Deorowicz discovered the sorted algorithm, benchmarked it by building automata for several dictionaries and published the results as a technical report [CD99]. In early 2000, Daciuk re-examined the literature, discovering the work of Sgarbas *et al* and Ciura & Deorowicz. At the 2000 *Conference on Implementations and Applications of Automata* (CIAA[4]), Dominique Revuz presented essentially the generalized algorithm [Rev00] — though he also sketched word deletion algorithms similar to those previously derived by Watson & Watson for Novell. At the 2001 CIAA, Jorge Graña *et al* summarized some of the current results and made improvements to several of the algorithms [GBA01]. Recently, the generalized algorithm has been straightforwardly extended by Rafael Carrasco and Mikel Forcada to handle *cyclic* automata [CF02]. In this thesis, the generalized algorithm is given in Chapter 6 while the sorted algorithm is given in Chapter 9.

In early 1998, I presented the generalized incremental algorithm from memory in a seminar at Sheng Yu's group at the University of Western Ontario, in Canada. During the presentation, I made some alternative derivation choices, arriving at a semi-incremental algorithm[5], which was then presented at the 1998 *Workshop on Implementing Automata* held in Rouen, France [Wat98b]. That paper was subsequently revised as a journal article in *Science of Computer Programming* [Wat03a] and a simplified version is given in this thesis as Chapter 11. While preparing this thesis, I derived

[2]which we call the *generalized* incremental algorithm, since it can add words to the automaton in any order whatsoever.

[3]In [Rev00], Revuz sketches algorithms related to the word deletion ones presented in this thesis.

[4]Formerly the Workshop on Implementations and Applications of automata.

[5]A *semi-incremental* algorithm in this context is one which does much of the minimization work incrementally (as words are added), but still requires a final 'cleanup' phase.

a simplified version of the semi-incremental algorithm — also based on adding words in *any* order of decreasing length. That simplified algorithm is not previously known from the literature and is presented in Chapter 10.

By 1999, I had started work on a taxonomy of the known algorithms. It was subsequently presented at the 1999 *Workshop on Implementing Automata* (the predecessor of CIAA) in Potsdam, Germany [Wat99c] and as a journal article in the *South African Computer Journal* [Wat01e]. One of the algorithms presented in that taxonomy can be derived by combining an automata construction and a minimization algorithm — both by Brzozowski [Wat00a, Wat02a]. An alternative derivation of the same algorithm is given in [Wat02b]. The resulting algorithm appears in this thesis as Chapter 7.

Also in 2000, I wrote a book chapter covering an elegant new recursive algorithm [Wat03b]. An elaborated version is presented as [Wat01d]. That algorithm can be viewed as a recursive rendition of the one in Chapter 6.

1.4 Links to the literature

The relationships between literature and the algorithms presented here is given in the following table:

Chapter	*Algorithm*	*Literature*
4	Revuz	[Rev91, Rev92] and *new*. Recent work by Bubenzer [Bub11] is not explicitly included in this thesis
5	Naïve	*New*
6	Fully incremental	[SFK95, PAMS94, DWW98, Mih99a, DMWW00, Dac98, Mih99b, HP98, Rev00, GBA01]
7	Double reversal (Brzozowski)	[Wat99c, Wat01e, Wat00a, Wat02a, Wat02b]
9	Words in lexicographic order	[Mih99a, Dac98, Mih99b, CD99]
10	Decreasing length (depth layers)	*New*
11	Decreasing length (semi-incremental)	[Wat98b, Wat03a]

Typically, the algorithms presented here include more detailed correctness arguments than their previous presentations in the literature.

1.5 Future work

Like all work in algorithmics, the results presented here are only the beginning. There are several promising directions for future work:

- Construct a toolkit (C++ library) of the algorithms and benchmark them in the style of [Wat95, Cle08]. This work has already begun, though is not advanced enough to report here.
- Derive and present the algorithms here in another paradigm, for example, functionally. This may lead to cleaner derivations and further insights.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- Explore parallelization of these algorithms, or the derivation of entirely new types of parallel algorithms solving the same problem. Recent related work in [SKW08] indicates promising directions.

Chapter 2

Preliminaries

In this chapter, we present the necessary mathematical preliminaries (including many properties not often given in the literature), assuming readers have a working knowledge of automata and formal languages. For such a background, see [HU79, Wat95, Smy03, CR03].

§2.1 and §2.2 lay notational foundations, including the style of algorithm presentation; §2.3 gives numerous definitions related to strings and languages, while §2.4 introduces automata and their key properties, and finally §2.5 details the minimality of automata and algorithms for minimizing them.

2.1 General definitions

In this section, we present some general notational definitions.

Notation 2.1 (Quantification) *We assume a basic understanding of the meaning of* quantification. *We use the notation*

$$\langle \oplus\ a : R(a) : f(a) \rangle$$

where $\oplus$ *is an associative and commutative operator (to be quantified) with unit* $1_\oplus$*,* a *is a* dummy variable*,* R *is a* range predicate*, and* $f(a)$ *is the* quantification expression*. By definition, when the range is empty (the predicate is* false*), the entire quantification evaluates to the unit* $1_\oplus$*. For example, a* $\forall$ *quantification over an empty range evaluates to* true *because the quantified operator is* $\wedge$ *which has unit* true*.*

Notation 2.2 (Conditional Boolean operators) *We use* ***cand*** *and* ***cor*** *to refer to the* conditional *(also known as 'short circuit') equivalents of* $\wedge$ *and* $\vee$ *(respectively).*

Notation 2.3 (Powerset) *For any set* A*, we use* $\mathcal{P}(A)$ *to denote the set of all subsets (including the empty set,* $\emptyset$*) of* A*.*

Notation 2.4 (Set difference) *For any sets* A, B*, we use* $A - B$ *to denote set difference instead of the* $A \setminus B$ *sometimes seen in the literature.*

Notation 2.5 (Derivation) *We adopt the 'Eindhoven' style of derivational proof, in which each derivation step appears on its own line, separated by a derivation operator (typically* $\equiv$*,* $\Rightarrow$*,* $=$*, etc.) and 'hint' of why the step is valid. For example*

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

P_0

$\equiv$ *"some reason why P_1 should obviously be equivalent to P_0"*

P_1

Note of course that this step could have involved implication, or any of the other derivation operators.

2.2 Algorithm presentation

In this thesis, the abstract algorithms are presented in Dijkstra's guarded command language [Dij76, Gri80], while the concrete implementation details are presented in C++. For ease of presentation, we take a number of notational shortcuts in the abstract algorithm presentations:

- We use the **as-sa** statement: many **if-fi** statements involve one branch guarded by a predicate R and another **skip** branch guarded by $\neg R$; in that case, we simplify to an **as-sa** statement with the single R-guarded statement.
- Since all of the algorithms presented in this thesis operate on one finite automaton (constructing it from a finite set of words), we simply assume the automaton and set of words are *global variables*. This is in contrast to explicitly passing them to program functions, procedures, predicates (as used in pre- and post-conditions and invariants), etc. The aim of this shortcut is to reduce typesetting and presentation clutter. Naturally, a real implementation would limit the scope of such variables for software engineering reasons.
- We often use *shadow* variables in predicates (preconditions, postconditions, invariants, etc.) to capture an old value of some variable or property, allowing us to later reason about the 'old' value. Shadow variables are typeset the same as program variables.
- The notation $x : S$ designates a statement S to be refined in such a way that it may modify only variable x. Since the majority of the statements may modify the global automaton (and its subcomponents), we do not explicitly mention it in this 'frame'.

Work on a C++ toolkit of the algorithms is ongoing; it will be available via `www.fastar.org` when complete.

2.3 Strings and languages

In this section, we present definitions and properties related to strings and languages.

Definition 2.6 (Alphabet) *An alphabet is a finite nonempty set of* symbols *— also known as* letters.

Throughout this thesis, we assume a fixed alphabet Σ.

Notation 2.7 *Σ^* denotes the set of all words over Σ — including the empty string, written as ε. Furthermore, we define $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.*

Definition 2.8 (String operators) *Define string head and tail operators* $\mathsf{head} \in \Sigma^+ \longrightarrow \Sigma$ *and* $\mathsf{tail} \in \Sigma^+ \longrightarrow \Sigma^*$ *(for $a \in \Sigma, v \in \Sigma^*$) as*

$$\mathsf{head}(av) = a$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

and

$$\mathsf{tail}(av) = v$$

Without explicitly defining them, we can extend these two operators as needed to sequences of other types.

Definition 2.9 (String and language reversal) *Given string w, define w^R to be the reversal of w, i.e. in which the letters appear in reverse order. Inductively (for $a \in \Sigma$), $\varepsilon^R = \varepsilon$ and $(aw)^R = w^R a$. For a language L, define $L^R = \{ w^R \mid w \in L \}$.*

Definition 2.10 (Alphabet ordering) *We assume a total ordering $\leqslant$ on alphabet Σ. (This is typically the ASCII ordering.) By extension, we also have the ordering $<$ on Σ.*

Some of the algorithms in this thesis require the *lexicographic* ordering (also known as the 'telephone book' ordering) on words in Σ^*.

Definition 2.11 (Lexicographic ordering of Σ^*) *For simplicity, we begin with the ordering $\sqsubset_l$ on Σ^*. For all $a, b \in \Sigma$ and $v, w \in \Sigma^*$*

$$\varepsilon \sqsubset_l av$$

and

$$av \sqsubset_l bw \equiv \begin{cases} v \sqsubset_l w & \textit{if } a = b \\ a < b & \textit{otherwise} \end{cases}$$

The lexicographic ordering $\sqsubseteq_l$ is a total ordering on Σ^*, defined as

$$v \sqsubseteq_l w \equiv (v = w \vee v \sqsubset_l w)$$

Example 2.12 (Lexicographic order) *The words* `had`, `hard`, `he`, `head`, `heard`, `her`, `herd`, *and* `here` *are in lexicographic order.*

Definition 2.13 (Longest common prefix) *Given two words $v, w \in \Sigma^*$, define $v \overset{\triangle}{\scriptscriptstyle p} w$ as the longest common prefix of v and w. This can also be given inductively (where additionally $a, b \in \Sigma : a \neq b$)*

$$\varepsilon \overset{\triangle}{\scriptscriptstyle p} v = v \overset{\triangle}{\scriptscriptstyle p} \varepsilon = \varepsilon$$

and

$$av \overset{\triangle}{\scriptscriptstyle p} bw = \varepsilon$$

and

$$av \overset{\triangle}{\scriptscriptstyle p} aw = a(v \overset{\triangle}{\scriptscriptstyle p} w)$$

Example 2.14 (Longest common prefix)

$$\mathtt{head} \overset{\triangle}{\scriptscriptstyle p} \mathtt{heard} = \mathtt{hea}$$

Definition 2.15 (Left derivative) *Given two strings v, w such that v is a prefix of w, we define the left v-derivative of w, written $v^{-1}w$, as the unique string such that*

$$w = v(v^{-1}w)$$

Example 2.16 (Left derivative)

$$\mathtt{her}^{-1}\mathtt{herd} = \mathtt{d}$$

Derivatives were introduced by Brzozowski in [Brz62b]. There is a symmetrical notion of *right* derivatives, though we have no need for them in this thesis.

2.4 Automata

In this section, we present automata and related properties.

Definition 2.17 (Deterministic finite automata) *A deterministic finite automaton (a* DFA *— also used to denote the set of all such automata) is a quadruple* (Q, δ, s, F) *where*

- Q *is a finite nonempty set of* states.
- $\delta \in Q \times \Sigma \overset{partial}{\longrightarrow} Q$ *is the (possibly partial) transition function. We use* $\perp$ *to denote the invalid destination state of a transition, thus the signature could have been written as a total function*[1] $\delta \in Q \times \Sigma \longrightarrow Q \cup \{\perp\}$.
- $s \in Q$ *is the start state.*
- $F \subseteq Q$ *is the set of final states.*

In the literature, a common interpretation is to view $\perp$ as a *special* state, such as a sink state. In this thesis, $\perp$ rather indicates an undefined part of δ. The sink state interpretation would lead to cyclic automata — undesirable in this thesis.

Throughout this thesis, we assume a specific DFA $M = (Q, \delta, s, F)$. Many functions and predicates take (Q, δ, s, F) as their only argument. To avoid notational clutter, where the meaning is clear we will omit the argument and assume it implicitly.

In this thesis, we will not need *nondeterministic* finite automata.

Definition 2.18 (Abstract program procedures and types for automata) *Program type* STATE *is a universe of states. We also assume a number of program functions/procedures*[2]*:*

- create() : STATE

  Create a new state (without out-transitions) in M *(taken from* STATE*).*

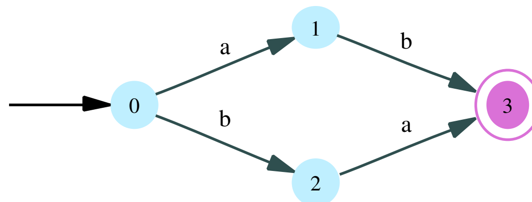
- clone(p : STATE) : STATE

  Create and return a new state with the same out-transitions and 'finality' as p.

- merge(p, q : STATE)

  Assume $p, q : p \neq q$ *are equivalent (discussed later); redirect all of* p*'s in-transitions to* q*, and delete* p.

Notation 2.19 (Drawing DFAs) *We draw the automata in the standard way, depicting states as ellipses, start states having an in-edge from nowhere and final states being two concentric ellipses. Transitions are depicted as labeled directed edges, as in*



[1] Note that numerous other isomorphic signatures are possible, for example $\delta \in Q \overset{\text{partial}}{\longrightarrow} \mathcal{P}(\Sigma \times Q)$.

[2] Note that these procedures implicitly take our DFA M as an additional argument, and we assume that they update the components of $M = (Q, \delta, s, F)$ as needed.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Definition 2.20 (ADFA) ADFA *is the set of all* DFAs *with acyclic transition graphs.*

Definition 2.21 (Size of a DFA) *The size of* M*, written* $|M|$*, is defined as* $|Q|$.

Other notions of size are possible, for example, involving the total number of transitions. We do not consider them here.

Notation 2.22 *For a state* p*,* Σ_p *denotes the subset of* Σ *on which* p *has out-transitions. That is,*

$$\Sigma_p = \{ a \mid a \in \Sigma \wedge \delta(p, a) \neq \perp \}$$

Definition 2.23 (Confluence state) *A state* p *is a* confluence *state, written* $\mathsf{Is_confl}(p)$*, iff it has more than one in-transition. In the literature, these are sometimes also known as 're-entrant' states, though we avoid that term. In Notation 2.19, state 3 is a confluence.*

Definition 2.24 *A set of states* X *is confluence-free, written* $\mathsf{Confl_free}(X)$*, iff*

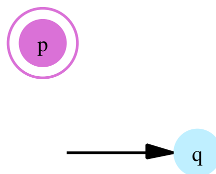
$$\langle \forall\ p : p \in X : \neg \mathsf{Is_confl}(p) \rangle$$

Property 2.25 (Merging states) *For states* $p, q : p \neq q$ *where both have in-transitions,* $\mathsf{merge}(p, q)$ *leaves* q *as a confluence state. (Recall that* p *is deleted.)*

Definition 2.26 (Useless state) *A state* p *is useless if there is no path from the start state to* p*, or there is no path from* p *to a final state.*

In this thesis, most algorithms will be designed to not introduce useless states; similarly, most of our definitions and properties will require the assumption of no useless states. We have, however, taken one minor short-cut: in a DFA accepting the empty language, the start state s is not final and is therefore, strictly speaking, useless. To keep the algorithms simple, we ignore this corner case.

Example 2.27 *In the following (unconnected)* DFA*, both states are useless.*

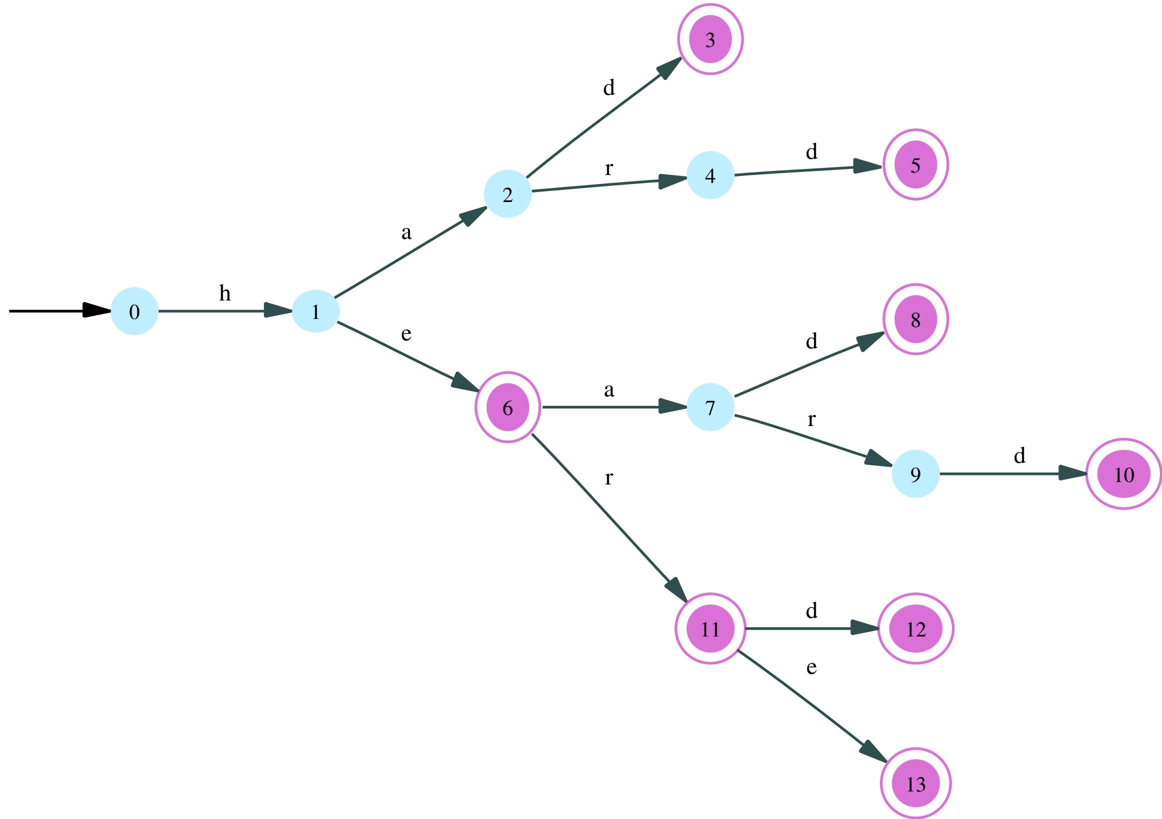


Property 2.28 *Since we have no useless states, in an* ADFA *the start state* s *is never a confluence state.*

Definition 2.29 (Trie) M *is a* trie*, written* $\mathsf{Is_trie}$ *(we assume* M *in our global scope), iff its transition graph is a tree rooted at start state* s.

Property 2.30 (Tries) *Tries have no confluence states.*

Example 2.31 *The following* DFA *is a trie:*



Property 2.32 *If a trie has no useless states then all leaves are final states.*

Definition 2.33 (Extending δ) *We extend δ to $\delta^* \in Q \times \Sigma^* \stackrel{partial}{\longrightarrow} Q$ as*

$$\delta^*(p, \varepsilon) = p$$

and (for $a \in \Sigma, v \in \Sigma^*$)

$$\delta^*(p, av) = \begin{cases} \delta^*(\delta(p, a), v) & \text{if } a \in \Sigma_p \\ \perp & \text{otherwise} \end{cases}$$

Definition 2.34 (Right language of a state) *The right language of a state p, denoted $\overrightarrow{\mathcal{L}}(p)$, is defined by*

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

That is, $\overrightarrow{\mathcal{L}}(p)$ is the set of strings on paths from p to any final state.

Property 2.35 $q \in F \equiv \varepsilon \in \overrightarrow{\mathcal{L}}(q)$.

Definition 2.36 (Left language of a state) *The left language of a state p, denoted $\overleftarrow{\mathcal{L}}(p)$, is defined by*

$$\overleftarrow{\mathcal{L}}(p) = \{ w \mid \delta^*(s, w) = p \}$$

That is, $\overleftarrow{\mathcal{L}}(p)$ is the set of strings on paths from start state s to state p.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Property 2.37 *For a state* p *which is not useless, we have* $\overleftarrow{\mathcal{L}}(p) \neq \emptyset$ *and* $\overrightarrow{\mathcal{L}}(p) \neq \emptyset$.

Example 2.38 *In Notation 2.19,* $\overrightarrow{\mathcal{L}}(0) = \overleftarrow{\mathcal{L}}(3) = \{ab, ba\}$.

Property 2.39 (Recursive definition of $\overrightarrow{\mathcal{L}}$) *The recursive definition of* δ^* *can be used to give a recursive definition for* $\overrightarrow{\mathcal{L}}$ *as follows:*

$$\overrightarrow{\mathcal{L}}(q) = \left(\bigcup_{a \in \Sigma_q} \{a\} \overrightarrow{\mathcal{L}}(\delta(q,a)) \right) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{if } q \notin F \end{cases}$$

Phrased differently, a string v is in $\overrightarrow{\mathcal{L}}(q)$ iff

- v is of the form aw where $a \in \Sigma$ is a label of an out-transition from q to $\delta(q,a)$ (i.e. $a \in \Sigma_q$) and w is in the right language of $\delta(q,a)$, or
- $v = \varepsilon$ and q is a final state.

A recursive definition of $\overleftarrow{\mathcal{L}}$ is not required in this thesis.

Definition 2.40 (Language of a DFA) *The language accepted by* DFA M*, denoted* $\mathcal{L}$*, is defined by*

$$\mathcal{L} = \overrightarrow{\mathcal{L}}(s)$$

Note that we could also have defined $\mathcal{L}$ using left languages as

$$\langle \cup\, f : f \in F : \overleftarrow{\mathcal{L}}(f) \rangle$$

Definition 2.41 (Path through a DFA) *For state* p *and* $w \in \Sigma^*$,

$$[p \overset{w}{\rightsquigarrow}]$$

is the sequence of states $p, \ldots, \delta^*(p,v)$ *where* v *is the longest prefix of* w *such that* $\delta^*(p,v) \neq \perp$*. We refer to this as the single "*w*-path from state* p*."*

Notation 2.42 (Path through a DFA) *We use the standard parentheses notation to denote state sequences which are open at the beginning or end — for example* $(p \overset{w}{\rightsquigarrow}]$ *does not include* p *but does include the rest of* $[p \overset{w}{\rightsquigarrow}]$*. In some contexts, we may pass a path* $[p \overset{w}{\rightsquigarrow}]$ *as an argument to a predicate or function which expects a* set*, thereby implicitly treating the path as a set of states.*

Property 2.43 *We can give a recursive definition for* $[p \overset{w}{\rightsquigarrow}]$*:*

$$[p \overset{\varepsilon}{\rightsquigarrow}] = p$$

and, for all $a \in \Sigma, w \in \Sigma^*$ *(where* $\cdot$ *is sequence concatenation and* ε *is the empty sequence which some authors write as* $[\,]$*)*

$$[p \overset{aw}{\rightsquigarrow}] = p \cdot \begin{cases} [\delta(p,a) \overset{w}{\rightsquigarrow}] & \text{if } a \in \Sigma_p \\ \varepsilon & \text{otherwise} \end{cases}$$

Definition 2.44 (Reachability of states) *Succ is a binary relation on states defined as*

$$\mathsf{Succ}(p,q) \equiv \langle \exists\, a : a \in \Sigma_p : \delta(p,a) = q \rangle$$

Note that Succ is essentially δ with the Σ component projected away.

Definition 2.45 (Succ^+ and Succ^*) *Succ^* (respectively Succ^+) is the reflexive and transitive (respectively reflexive-only) closure of Succ. $\mathsf{Succ}^*(p,q)$ iff there is a path from p to q in the transition graph.*

It follows that Succ^* (respectively Succ^+) is essentially δ^* (respectively δ^+) with the Σ^* (respectively Σ^+) component projected away.

Notation 2.46 *We will also use Succ as a function, mapping a state to its successor states. In this context,*

$$\mathsf{Succ}(p) = \{\, \delta(p,a) \mid a \in \Sigma_p \,\}$$

We extend Succ to taking a set of states by distributing Succ over $\cup$. We similarly extend the signatures of Succ^* and Succ^+.

Property 2.47 (Recursive forms of Succ^+ and Succ^*) *For any state p, we have*

$$\begin{array}{ll}
 & \mathsf{Succ}^+(p) \\
= & \quad \text{“definition of reflexive closure”} \\
 & \mathsf{Succ}^*(\mathsf{Succ}(p)) \\
= & \quad \text{“Definition 2.44”} \\
 & \mathsf{Succ}^*(\langle \cup\, a : a \in \Sigma_p : \delta(p,a) \rangle) \\
= & \quad \text{“Succ and } \mathsf{Succ}^* \text{ distribute over } \cup \text{”} \\
 & \langle \cup\, a : a \in \Sigma_p : \mathsf{Succ}^*(\delta(p,a)) \rangle
\end{array}$$

Using the above derivation we have

$$\begin{array}{ll}
 & \mathsf{Succ}^*(p) \\
= & \quad \text{“definition of reflexive and transitive closure”} \\
 & \{p\} \cup \mathsf{Succ}^+(p) \\
= & \quad \text{“derivation above”} \\
 & \{p\} \cup \langle \cup\, a : a \in \Sigma_p : \mathsf{Succ}^*(\delta(p,a)) \rangle
\end{array}$$

(These are well-formed because our automata are acyclic and finite. Similar definitions are valid for cyclic automata, and are then based on a fixed-point of such recursive equations.)

Property 2.48 (Confluence-free state paths) *A confluence-free path $[s \overset{u}{\rightsquigarrow}]$ allows us to characterize the successors of the* remaining *states (not on that path):*

$$\begin{array}{ll}
 & \mathsf{Confl_free}([s \overset{u}{\rightsquigarrow}]) \\
\equiv & \quad \text{“definitions of } [s \overset{u}{\rightsquigarrow}] \text{ and confluence state; state } s \text{ has no in-transitions”} \\
 & \text{each state in } (s \overset{u}{\rightsquigarrow}] \text{ has a single in-transition from its predecessor, which is in } [s \overset{u}{\rightsquigarrow}) \\
\Rightarrow & \quad \text{“transitions from other states } Q - [s \overset{u}{\rightsquigarrow}) \text{ cannot go to } (s \overset{u}{\rightsquigarrow}] \text{”}
\end{array}$$

$$\begin{array}{ll} & \mathsf{Succ}(Q - [s \overset{u}{\leadsto})) \cap (s \overset{u}{\leadsto}] = \emptyset \\ \equiv & \quad \text{“\textit{set calculus}”} \\ & \mathsf{Succ}(Q - [s \overset{u}{\leadsto})) \subseteq Q - (s \overset{u}{\leadsto}] \\ \equiv & \quad \text{“\textit{state } s \textit{ has no in-transition (and is not a successor of any state)}”} \\ & \mathsf{Succ}(Q - [s \overset{u}{\leadsto})) \subseteq Q - [s \overset{u}{\leadsto}] \end{array}$$

Definition 2.49 (Longest right word length function) *For an* ADFA *only*[3]*, define* $\overrightarrow{\mathcal{L}}_{|max|} \in Q \longrightarrow \mathbb{N}$ *as*

$$\overrightarrow{\mathcal{L}}_{|max|}(p) = \langle \textbf{\textit{MAX}}\ w : w \in \overrightarrow{\mathcal{L}}(p) : |w| \rangle$$

$\overrightarrow{\mathcal{L}}_{|max|}(p)$ is the length of the longest path from p to any final state in an ADFA. In [Rev92], Revuz calls $\overrightarrow{\mathcal{L}}_{|max|}(p)$ the 'height' of p.

Property 2.50 (Function $\overrightarrow{\mathcal{L}}_{|max|}$) *The recursive definition of* $\overrightarrow{\mathcal{L}}$ *can be used to give a recursive definition for* $\overrightarrow{\mathcal{L}}_{|max|}$*:*

$$\overrightarrow{\mathcal{L}}_{|max|}(p) = (\langle \textbf{\textit{MAX}}\ a : a \in \Sigma_p : \overrightarrow{\mathcal{L}}_{|max|}(\delta(p,a)) \rangle + 1)\ \textbf{\textit{max}} \begin{cases} 0 & \textit{if } p \in F \\ -\infty & \textit{if } p \notin F \end{cases}$$

The above expression deserves some explanation: in the event that p has no out-transitions, the **max** quantification has an empty range and evaluates to the unit of **max**, namely $-\infty$. The righthand-side of the infix **max** considers the case that p is a final state, in which case its right language contains ε, of length 0. Since we usually have no useless states in this thesis, we cannot have a non-final state without out-transitions, and therefore we could simply use 0 as the second operand of the infix **max**.

Example 2.51 *In Example 2.31,* $\overrightarrow{\mathcal{L}}_{|max|}(3) = 0$, $\overrightarrow{\mathcal{L}}_{|max|}(6) = 3$ *and* $\overrightarrow{\mathcal{L}}_{|max|}(0) = 5$.

Property 2.52 *It follows from Property 2.50 that, for all states* p

$$\langle \forall\ a : a \in \Sigma_p : \overrightarrow{\mathcal{L}}_{|max|}(p) \geqslant \overrightarrow{\mathcal{L}}_{|max|}(\delta(p,a)) + 1 \rangle$$

Definition 2.53 (Height levels) *In an* ADFA*, for each* $k \in \mathbb{N}$ *we define a set of states*

$$HL_k = \{ p \mid p \in Q \wedge \overrightarrow{\mathcal{L}}_{|max|}(p) = k \}$$

State layer sets HL_k are a *partition* of Q. (That is, they are disjoint, and every state appears in *some* layer.)

Example 2.54 *In the trie of Example 2.31,*

$$\begin{array}{lcl} HL_0 & = & \{3,5,8,10,12,13\} \\ HL_1 & = & \{4,9,11\} \\ HL_2 & = & \{2,7\} \\ HL_3 & = & \{6\} \\ HL_4 & = & \{1\} \\ HL_5 & = & \{0\} \end{array}$$

[3] This restriction is placed because a cyclic DFA may have arbitrarily long paths (following a cycle) from a state to a final state.

Property 2.55 (Procedure merge) *Merging state* p *into* q *with an invocation* $\mathrm{merge}(p,q)$ *(which is only valid when* $\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)$*) does not change* $\overrightarrow{\mathcal{L}}(q)$*. Consequently,* $\overrightarrow{\mathcal{L}}_{|max|}(q)$ *remains unchanged and* q *remains in the same height level as before the invocation of* merge.

Property 2.56 (Height levels) *For* $k \geqslant 0$

$$HL_k = \emptyset \Rightarrow HL_{k+1} = \emptyset$$

This follows from the following contrapositive argument

$$
\begin{array}{ll}
 & HL_{k+1} \neq \emptyset \\
\equiv & \quad \textit{“Definition 2.53”} \\
 & \langle \exists\, p : p \in Q : \overrightarrow{\mathcal{L}}_{|max|}(p) = k+1 \rangle \\
\equiv & \quad \textit{“Property 2.50; } M \textit{ has no useless states”} \\
 & \langle \exists\, p : p \in Q : (\langle \textbf{\textit{MAX}}\ a : a \in \Sigma_p : \overrightarrow{\mathcal{L}}_{|max|}(\delta(p,a)) \rangle + 1)\ \textbf{\textit{max}}\ 0 = k+1 \rangle \\
\equiv & \quad \textit{“drop } \textbf{\textit{max}}\ 0 \textit{ because } k \geqslant 0 \textit{ from assumption and so } k+1 \geqslant 1\textit{”} \\
 & \langle \exists\, p : p \in Q : \langle \textbf{\textit{MAX}}\ a : a \in \Sigma_p : \overrightarrow{\mathcal{L}}_{|max|}(\delta(p,a)) \rangle + 1 = k+1 \rangle \\
\equiv & \quad \textit{“arithmetic”} \\
 & \langle \exists\, p : p \in Q : \langle \textbf{\textit{MAX}}\ a : a \in \Sigma_p : \overrightarrow{\mathcal{L}}_{|max|}(\delta(p,a)) \rangle = k \rangle \\
\Rightarrow & \quad \textit{“quantify states } \delta(p,a)\textit{”} \\
 & \langle \exists\, q : q \in Q : \overrightarrow{\mathcal{L}}_{|max|}(q) = k \rangle \\
\equiv & \quad \textit{“Definition 2.53”} \\
 & HL_k \neq \emptyset
\end{array}
$$

Definition 2.57 (State depth function) *Function* $\overleftarrow{\mathcal{L}}_{|min|} \in Q \longrightarrow \mathbb{N}$ *is defined as*

$$\overleftarrow{\mathcal{L}}_{|min|}(p) = \langle \textbf{\textit{MIN}}\ w : w \in \overleftarrow{\mathcal{L}}(p) : |w| \rangle$$

$\overleftarrow{\mathcal{L}}_{|min|}(p)$ is the length of the shortest path from s to p. In the literature, $\overleftarrow{\mathcal{L}}_{|min|}(p)$ is also known as the 'depth' of p. Note the asymmetry of functions $\overrightarrow{\mathcal{L}}_{|max|}$ and $\overleftarrow{\mathcal{L}}_{|min|}$. We will not need a recursive definition of $\overleftarrow{\mathcal{L}}_{|min|}$.

Definition 2.58 (Depth levels) *In an* ADFA*, for each* $k \in \mathbb{N}$ *we define a set of states at 'depth level* k*'*

$$DL_k = \{ p \mid p \in Q \wedge \overleftarrow{\mathcal{L}}_{|min|}(p) = k \}$$

Property 2.59 *The depth levels form a partition of* Q.

Notation 2.60 (Depth levels) *We use the following notational short-hand for* $k, l \in \mathbb{N}$

$$DL_{>k} = \langle \cup\, j : j > k : DL_j \rangle$$

and

$$DL_{\leqslant k} = \langle \cup\, j : 0 \leqslant j \leqslant k : DL_j \rangle$$

We can analogously define $DL_{[k,l]}$, $DL_{(k,l]}$ *etc.*

Definition 2.61 (Shortest word length of a DFA) *Function* minlen *is the length of the shortest word accepted by* M. *Formally,*

$$\mathsf{minlen} = \langle \boldsymbol{MIN}\ f : f \in F : \overleftarrow{\mathcal{L}}_{|min|}(f) \rangle$$

Clearly, minlen is the depth of a final state closest (in terms of path-length) to start state s.

Definition 2.62 (Lexicographically greatest word) *Define* lexmax *as the lexicographically greatest word in* $\mathcal{L}$. *Formally,*

$$\mathsf{lexmax} = \langle \boldsymbol{MAX}_{\sqsubseteq_\iota}\ w : w \in \mathcal{L} : w \rangle$$

Property 2.63 (Lexicographically greatest word) lexmax *can be found structurally in* M *by beginning at start state* s *and always following the out-transition on the highest ranked letter (under* $\leqslant$*) until no further out-transitions are possible.*

Property 2.64 lexmax *is unique since* $\sqsubseteq_\iota$ *is total.*

Property 2.65 *Note that if* $\mathcal{L} = \emptyset$ *then* lexmax $= \varepsilon$ *since* ε *is the unit of* $\boldsymbol{max}_{\sqsubseteq_\iota}$.

2.5 Minimality of automata

In this section, we present definitions and properties related to the minimality of automata, and follow this with two algorithms to determine the equivalence of states. Many of these properties and definitions do not appear explicitly in the literature.

Definition 2.66 (Minimality of a DFA) M *is minimal, written* Min(M) *or simply* Min, *iff it is the smallest (as measured by the number of states — see Definition 2.21)* DFA *accepting* $\mathcal{L}$.

Property 2.67 *A minimal* DFA *is unique up to isomorphism — see [HU79, §3.4].*

Definition 2.68 (MADFA) MADFA *is the set of all minimal* ADFA*s.*

Definition 2.69 (State equivalence) *Define* E *as an equivalence relation on states where*

$$\mathsf{E}(p, q) \equiv (\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q))$$

If two states p, q are equivalent under E, p can be merged into q using procedure merge — including some redirection of transitions — giving a smaller equivalent automaton.

Property 2.70 *Assuming no useless states, start state* s *is unique — that is, it is not equivalent to any other state.*

Property 2.71 (Recursive definition of E) *The recursive definition of* $\overrightarrow{\mathcal{L}}$ *(Property 2.39) gives rise to a recursive definition of* E *as follows*

$$
\begin{aligned}
& \mathsf{E}(p,q) \\
\equiv \quad & \text{\textit{“definition of} E\textit{”}} \\
& \overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q) \\
\equiv \quad & \text{\textit{“definition of language equality”}} \\
& \langle \forall\, v : v \in \Sigma^* : v \in \overrightarrow{\mathcal{L}}(p) \equiv v \in \overrightarrow{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{\textit{“split domain} } \Sigma^* \text{ \textit{into} } \{\varepsilon\} \cup \Sigma^+ \text{\textit{”}} \\
& \langle \forall\, v : v \in \{\varepsilon\} \cup \Sigma^+ : v \in \overrightarrow{\mathcal{L}}(p) \equiv v \in \overrightarrow{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{\textit{“split quantification”}} \\
& \langle \forall\, v : v \in \{\varepsilon\} : v \in \overrightarrow{\mathcal{L}}(p) \equiv v \in \overrightarrow{\mathcal{L}}(q) \rangle \\
& \wedge \langle \forall\, v : v \in \Sigma^+ : v \in \overrightarrow{\mathcal{L}}(p) \equiv v \in \overrightarrow{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{\textit{“one-point rule on the first universal quantification”}} \\
& (\varepsilon \in \overrightarrow{\mathcal{L}}(p) \equiv \varepsilon \in \overrightarrow{\mathcal{L}}(q)) \\
& \wedge \langle \forall\, v : v \in \Sigma^+ : v \in \overrightarrow{\mathcal{L}}(p) \equiv v \in \overrightarrow{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{\textit{“introduce dummies} } a \in \Sigma, w \in \Sigma^* \text{ \textit{such that} } v = aw \text{ \textit{in second quantification”}} \\
& (\varepsilon \in \overrightarrow{\mathcal{L}}(p) \equiv \varepsilon \in \overrightarrow{\mathcal{L}}(q)) \\
& \wedge \langle \forall\, a, w : a \in \Sigma, w \in \Sigma^* : aw \in \overrightarrow{\mathcal{L}}(p) \equiv aw \in \overrightarrow{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{“} \varepsilon \in \overrightarrow{\mathcal{L}}(r) \equiv r \in F \text{”} \\
& (p \in F \equiv q \in F) \\
& \wedge \langle \forall\, a, w : a \in \Sigma, w \in \Sigma^* : aw \in \overrightarrow{\mathcal{L}}(p) \equiv aw \in \overrightarrow{\mathcal{L}}(q) \rangle \\
\equiv \quad & \text{\textit{“in context,} } aw \in \overrightarrow{\mathcal{L}}(p) \equiv (a \in \Sigma_p \textbf{ cand } w \in \overrightarrow{\mathcal{L}}(\delta(p,a))) \text{\textit{”}} \\
& (p \in F \equiv q \in F) \\
& \wedge \langle \forall\, a, w : a \in \Sigma, w \in \Sigma^* : (a \in \Sigma_p \textbf{ cand } w \in \overrightarrow{\mathcal{L}}(\delta(p,a))) \equiv (a \in \Sigma_q \textbf{ cand } w \in \overrightarrow{\mathcal{L}}(\delta(q,a))) \rangle \\
\equiv \quad & \text{\textit{“split universal quantifier;} \textbf{cand} no longer needed with } a \in \Sigma_p \cap \Sigma_q \text{ \textit{in quantifier range”}} \\
& (p \in F \equiv q \in F) \wedge \langle \forall\, a : a \in \Sigma : a \in \Sigma_p \equiv a \in \Sigma_q \rangle \\
& \wedge \langle \forall\, a, w : a \in \Sigma_p \cap \Sigma_q, w \in \Sigma^* : w \in \overrightarrow{\mathcal{L}}(\delta(p,a)) \equiv w \in \overrightarrow{\mathcal{L}}(\delta(q,a)) \rangle \\
\equiv \quad & \text{\textit{“definition of alphabet equality”}} \\
& (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \\
& \wedge \langle \forall\, a, w : a \in \Sigma_p \cap \Sigma_q, w \in \Sigma^* : w \in \overrightarrow{\mathcal{L}}(\delta(p,a)) \equiv w \in \overrightarrow{\mathcal{L}}(\delta(q,a)) \rangle \\
\equiv \quad & \text{\textit{“definition of language equality”}} \\
& (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \\
& \wedge \langle \forall\, a : a \in \Sigma_p \cap \Sigma_q : \overrightarrow{\mathcal{L}}(\delta(p,a)) = \overrightarrow{\mathcal{L}}(\delta(q,a)) \rangle \\
\equiv \quad & \text{\textit{“definition of} E\textit{”}} \\
& (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \wedge \langle \forall\, a : a \in \Sigma_p \cap \Sigma_q : \mathsf{E}(\delta(p,a), \delta(q,a)) \rangle
\end{aligned}
$$

Corollary 2.72 *In an* ADFA, *for all states* p, q

$$
\begin{aligned}
& \overrightarrow{\mathcal{L}}_{|max|}(p) > \overrightarrow{\mathcal{L}}_{|max|}(q) \\
\Rightarrow \quad & \text{“} \overrightarrow{\mathcal{L}}(p) \text{ \textit{has a word of length} } \overrightarrow{\mathcal{L}}_{|max|}(p) \text{ \textit{that is not in} } \overrightarrow{\mathcal{L}}(q) \text{”} \\
& \langle \exists\, v : v \in \overrightarrow{\mathcal{L}}(p) \wedge |v| = \overrightarrow{\mathcal{L}}_{|max|}(p) : v \notin \overrightarrow{\mathcal{L}}(q) \rangle
\end{aligned}
$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

$\Rightarrow$ *"language equality"*

$\overrightarrow{\mathcal{L}}(p) \neq \overrightarrow{\mathcal{L}}(q)$

$\Rightarrow$ *"definition of* E*"*

$\neg\mathsf{E}(p, q)$

Furthermore

$\overrightarrow{\mathcal{L}}_{|max|}(p) \neq \overrightarrow{\mathcal{L}}_{|max|}(q)$

$\equiv$ *"definition of* $\neq$*"*

$\overrightarrow{\mathcal{L}}_{|max|}(p) > \overrightarrow{\mathcal{L}}_{|max|}(q) \vee \overrightarrow{\mathcal{L}}_{|max|}(p) < \overrightarrow{\mathcal{L}}_{|max|}(q)$

$\Rightarrow$ *"derivation above"*

$\neg\mathsf{E}(p, q) \vee \neg\mathsf{E}(q, p)$

$\equiv$ *"without loss of generality;* E *is an equivalence relation"*

$\neg\mathsf{E}(p, q)$

Corollary 2.73 *In an* ADFA*, for all states* p

$$\langle \forall\, u : u \in \Sigma^+ \wedge \delta^*(p, u) \neq \perp : \neg\mathsf{E}(p, \delta^*(p, u)) \rangle$$

This follows from Property 2.52 and Corollary 2.72.

Corollary 2.74 *In an* ADFA*, for all states* p, q

$\mathsf{Succ}^+(p, q)$

$\equiv$ *"Property 2.47"*

$\langle \exists\, u : u \in \Sigma^+ : \delta^*(p, u) = q \rangle$

$\Rightarrow$ *"Corollary 2.73"*

$\neg\mathsf{E}(p, q)$

Definition 2.75 (Pairwise inequivalent pair of sets) *Given two state sets* $X, Y \subseteq Q$*,* X *is said to be* pairwise inequivalent *(PI) against* Y*, written*

$$\mathsf{Pairwise_inequiv}(X, Y)$$

iff all pairs of states (taken respectively from X, Y*) are inequivalent. Formally,*

$$\mathsf{Pairwise_inequiv}(X, Y) \equiv \langle \forall\, p, q : p \neq q \wedge p \in X \wedge q \in Y : \neg\mathsf{E}(p, q) \rangle$$

Property 2.76 ($\mathsf{Pairwise_inequiv}$**)** *For three state sets* X, Y, Z

$$\mathsf{Pairwise_inequiv}(X \cup Y, Z) \equiv \mathsf{Pairwise_inequiv}(X, Z) \wedge \mathsf{Pairwise_inequiv}(Y, Z)$$

Definition 2.77 (Pairwise inequivalent states) *Define predicate*

$$\mathsf{Inequiv}(X) \equiv \langle \forall\, p, q : p \neq q \wedge p, q \in X : \neg\mathsf{E}(p, q) \rangle$$

When $\mathsf{Inequiv}(X)$ holds, we can take a notational shortcut and say that states X are 'minimized'.

Property 2.78 *Note that* $\mathsf{Inequiv}(X) \equiv \mathsf{Pairwise_inequiv}(X, X)$*. These two predicates could have been combined; they are separately defined for readability.*

Corollary 2.79 *Thanks to Corollaries 2.73 and 2.74 and the definition of* $\rightsquigarrow$*, we have (for any state* p *and string* w*)* $\mathsf{Inequiv}([p \overset{w}{\rightsquigarrow}])$*. (Here, we view* $[p \overset{w}{\rightsquigarrow}]$ *as a set.) Note that this holds even when* $w = \varepsilon$*, because* $[p \overset{\varepsilon}{\rightsquigarrow}] = p$*.*

Property 2.80 (Inequivalence of levels) *From Definition 2.53, Corollary 2.72, and Definition 2.75, we have (for* $i, j : i \neq j$*)*

$$\mathsf{Pairwise_inequiv}(HL_i, HL_j)$$

Property 2.81 *From Property 2.50, we have for all* $k > 0$

$$\mathsf{Succ}(HL_k) \subseteq \langle \cup\, j : 0 \leqslant j < k : HL_j \rangle$$

Note that, in general, we do not have

$$\mathsf{Succ}(HL_k) \subseteq HL_{k-1}$$

since states in HL_k *may have* some *successors in levels strictly less than* $k - 1$*.*

Property 2.82 (Minimality of a DFA) Min *is equivalent to:*

- *all states in* $Q - \{s\}$ *are useful, and*
- $\mathsf{Inequiv}(Q)$.

This is shown in [HU79, §3.4].

Property 2.83 *Given sets of states* X, Y*,*

$$\begin{array}{ll}
 & \mathsf{Inequiv}(X \cup Y) \\
\equiv & \quad \text{“definition of Inequiv”} \\
 & \langle \forall\, p, q : p \neq q \wedge p, q \in (X \cup Y) : \neg E(p, q) \rangle \\
\equiv & \quad \text{“elaborate range”} \\
 & \langle \forall\, p, q : p \neq q \wedge (p, q \in X \vee p, q \in Y \vee (p \in X \wedge q \in Y) \vee (p \in Y \wedge q \in X)) : \neg E(p, q) \rangle \\
\equiv & \quad \text{“without loss of generality, symmetry of E”} \\
 & \langle \forall\, p, q : p \neq q \wedge (p, q \in X \vee p, q \in Y \vee (p \in X \wedge q \in Y)) : \neg E(p, q) \rangle \\
\equiv & \quad \text{“split range”} \\
 & \langle \forall\, p, q : p \neq q \wedge p, q \in X : \neg E(p, q) \rangle \\
 & \wedge \langle \forall\, p, q : p \neq q \wedge p, q \in Y : \neg E(p, q) \rangle \\
 & \wedge \langle \forall\, p, q : p \neq q \wedge p \in X \wedge q \in Y : \neg E(p, q) \rangle \\
\equiv & \quad \text{“definition of Inequiv”} \\
 & \mathsf{Inequiv}(X) \wedge \mathsf{Inequiv}(Y) \wedge \langle \forall\, p, q : p \neq q \wedge p \in X \wedge q \in Y : \neg E(p, q) \rangle \\
\equiv & \quad \text{“definition of Pairwise_inequiv”} \\
 & \mathsf{Inequiv}(X) \wedge \mathsf{Inequiv}(Y) \wedge \mathsf{Pairwise_inequiv}(X, Y)
\end{array}$$

2.5.1 Computing E

Relation E can be computed using one of several known algorithms — see [Wat95, WD03, Wat01c]. While those algorithms have widely varying worst-case running times, the best known (in terms of asymptotic worst-case running time) algorithm is that of Hopcroft [Hop71, Gri73], with time $\mathcal{O}(|Q| \log |Q|)$.

Conjecture 2.84 *The general minimization algorithms (such as Hopcroft's, etc.) have linear worst-case running time on acyclic automata.*

2.5.2 Computing ¬E in ADFAs

Instead of computing (and using) E, we can compute its complement ¬E — known as the *distinguishability* relation — directly from Succ. Relation Succ is easily computed from transition function δ. Thanks to Corollary 2.74, we can then use Succ as a starting point to compute ¬E. We do not elaborate the algorithm here — see [Wat95, §7.3.2, 7.4.1–7.4.3] for more.

2.5.3 Computing $E(p, q)$ pointwise

In [Wat95, Wat01c, WD03], the recursive form of E (Property 2.71) is transformed into a program which computes it pointwise (i.e. for each pair of states). That program avoids endless recursion in cyclic DFAs by extra book-keeping. In an ADFA, such endless recursion is not an issue and the definition can be used directly in the following program (in which we have omitted the invariant):

Algorithm 2.1:

```
func eq(in p, q : Q) : 𝔹 →
    ⟦ var e : 𝔹
    | e := (p ∈ F ≡ q ∈ F) ∧ Σ_p = Σ_q;
      ⟦ var a : Σ
      | for a : a ∈ Σ_p ∩ Σ_q →
            as e → e := eq(δ(p, a), δ(q, a)) sa
        rof
      ⟧;
      { e ≡ E(p, q) }
      return e
    ⟧
cnuf
```

□

This algorithm has exponential worst-case running time — see [Wat95, §15.4] for a pathological example. *Memoization* (caching previously computed results) improves this significantly as is shown in [WD03].

2.5.4 A more efficient computation of E

The equivalence of two states can be determined more cheaply if we make a simple restriction on the states being tested (for equivalence).

Property 2.85 *Given two states* p, q

$$
\begin{array}{ll}
 & \mathsf{Inequiv}(\mathsf{Succ}(\{p,q\})) \\
\equiv & \quad \text{“definitions of Inequiv and E”} \\
 & \langle \forall\, m,n : m,n \in \mathsf{Succ}(\{p,q\}) : \mathsf{E}(m,n) \equiv (m = n) \rangle \\
\equiv & \quad \text{“Notation 2.46 rewritten in Eindhoven-style quantification”} \\
 & \langle \forall\, m,n : m,n \in \langle \cup\, a : a \in \Sigma_p : \{\delta(p,a)\} \rangle \cup \langle \cup\, b : b \in \Sigma_q : \{\delta(q,b)\} \rangle : \mathsf{E}(m,n) \equiv (m = n) \rangle \\
\equiv & \quad \text{“combine inner quantifications”} \\
 & \langle \forall\, m,n : m,n \in \langle \cup\, a,b : a \in \Sigma_p, b \in \Sigma_q : \{\delta(p,a), \delta(q,b)\} \rangle : \mathsf{E}(m,n) \equiv (m = n) \rangle \\
\Rightarrow & \quad \text{“restrict range and drop one dummy in inner quantification”} \\
 & \langle \forall\, m,n : m,n \in \langle \cup\, a : a \in \Sigma_p \cap \Sigma_q : \{\delta(p,a), \delta(q,a)\} \rangle : \mathsf{E}(m,n) \equiv (m = n) \rangle \\
\equiv & \quad \text{“one-point rule; change of dummy”} \\
 & \langle \forall\, a : a \in \Sigma_p \cap \Sigma_q : (\mathsf{E}(\delta(p,a), \delta(q,a)) \equiv (\delta(p,a) = \delta(q,a)) \rangle
\end{array}
$$

Property 2.86 *Given two states* p, q *such that* $\mathsf{Inequiv}(\mathsf{Succ}(\{p,q\}))$*, we have a more easily computed form of* $\mathsf{E}(p,q)$*:*

$$
\begin{array}{ll}
 & \mathsf{E}(p,q) \\
\equiv & \quad \text{“recursive definition of E”} \\
 & (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \wedge \langle \forall\, a : a \in \Sigma_p \cap \Sigma_q : \mathsf{E}(\delta(p,a), \delta(q,a)) \rangle \\
\equiv & \quad \text{“assumption Inequiv(Succ(\{p,q\})) and Property 2.85”} \\
 & (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \wedge \langle \forall\, a : a \in \Sigma_p \cap \Sigma_q : \delta(p,a) = \delta(q,a) \rangle
\end{array}
$$

This can be evaluated directly with the following function:

Algorithm 2.2:

```
func eq′(in p, q : Q) : 𝔹 →
    { pre Inequiv(Succ({p, q})) }
    |[ var e : 𝔹
    | e := (p ∈ F ≡ q ∈ F) ∧ Σp = Σq;
      |[ var a : Σ
      | for a : a ∈ Σp ∩ Σq →
              as e → e := (δ(p, a) = δ(q, a)) sa
        rof
      ]|;
      { post e ≡ E(p, q) }
      return e
    ]|
cnuf
```

□

In [WD03], Watson and Daciuk show how eq' can be implemented to have $\mathcal{O}(|\Sigma|)$ (constant) running-time, typically by using hashing and caching Σ_p for all states p. This is surprising, given that eq can be worst-case exponential in $|Q|$.

Chapter 3

A MADFA-construction algorithm skeleton

In this chapter, we present a general algorithm skeleton for constructing a MADFA from a set of words. One of the main contributions of this thesis is that all of the known algorithms for constructing MADFAs are presented using this common algorithm skeleton — it therefore forms the basis of a *taxonomy* of such algorithms, clearly illustrating where they differ and what they have in common. A less rigorous (and slightly differently structured) taxonomy of MADFA construction algorithms is given in [Wat01e].

The structure of the algorithm we have chosen will add the words one-by-one[1]. In some cases, the order in which the words are added is important — and so we assume some partial order $\leqslant$ on the words[2]. As we present the general algorithm, we will leave a number of things undetailed:

1. A structural invariant, $\mathsf{Struct}(D)$ (for the set of words D already processed), maintained on the ADFA; that is, $\mathsf{Struct}(D)$ holds both before and after a word w is added to the ADFA. Examples of such invariants are: *the* ADFA *has a trie structure*, *the* ADFA *is minimal*, etc.

2. The body of a procedure, add_word, used to add individual words.

3. The partial order $\leqslant$ on the words.

4. The body of a cleanup procedure, cleanup, applied to the ADFA after the words have been added, yielding the desired MADFA.

All of these are, in some sense, *meta-parameters* of the algorithm.

The algorithm skeleton adds a single word (from W) at a time to the words accepted by the automaton. With some of the add_word procedures, the intermediate automaton may not yet be minimal, requiring the corresponding cleanup procedure to further manipulate the ADFA, finally resulting in a MADFA. For this reason, we actually use the more general type ADFA for the automaton.

In the following algorithm, we maintain a partition of $W \subset \Sigma^*$ into D (for 'done') and T (for 'to-do') and assume word set W and ADFA $M = (Q, \delta, s, F)$ are global variables:

Algorithm 3.1:

---

[1] As Gerard Zwaan pointed out in personal communication [Zwa01], we could consider other structures — for example where several words are added at once. An alternative 'divide-and-conquer' approach would be to recursively halve set W, building a MADFA for each such smaller set, and combining the resulting MADFAs. Although such algorithms may be interesting (and remain as future work), they are not considered in this thesis.

[2] The order could also be 'unordered' for those algorithms in which words may be added in arbitrary orders.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
|[ var D, T : set of Σ*
| { W ⊂ Σ* }
  s := create();
  (Q, δ, s, F) := ({s}, ∅, s, ∅);
  D, T := ∅, W;
  { invariant: Struct(D)
    variant: |T| }
  do T ≠ ∅ → |[ var w : Σ*
              | let w : w is any minimal element of T under ⩽;
                { Struct(D) }
                add_word(w);
                { Struct(D ∪ {w}) }
                D, T := D ∪ {w}, T − {w}
                { Struct(D) }
              ]|
  od;
  { Struct(W) }
  cleanup()
  { Min ∧ 𝓛 = W }
]|
```

□

(Recall from Definition 2.18 that create implicitly updates M. Above, we explicitly initialize M for clarity.)

3.1 Specific instantiations

In each of the subsequent chapters, we will co-derive specific versions of: add_word, cleanup, Struct(D) and $\leqslant$. The general process is outlined in the following subsections.

3.1.1 Choosing a structural invariant

We begin with a choice of our structural invariant as one of:

1. $\mathsf{Struct}_\mathsf{T}(D) \equiv \mathsf{Is_trie} \wedge \mathcal{L} = D$, leading to nonincremental (i.e. those which first build an ADFA then minimize afterwards) trie-based algorithms in Chapter 4 on page 27.

2. $\mathsf{Struct}_\mathsf{N}(D) \equiv \mathcal{L} = D$, leading to more nonincremental algorithms in Chapter 5 on page 43.

3. $\mathsf{Struct}_\mathsf{I}(D) \equiv \mathsf{Min} \wedge \mathcal{L} = D$, leading to incrementally minimizing algorithms in Chapter 6 on page 49.

4. $\mathsf{Struct}_\mathsf{R}(D) \equiv \mathsf{Is_trie} \wedge \mathcal{L} = D^R$, leading to an algorithm related to Brzozowski's minimization algorithm. This is presented in Chapter 7 on page 61.

5. $\mathsf{Struct}_\mathsf{S}(D) \equiv \mathsf{Inequiv}(Q - [s \overset{\mathsf{lexmax}}{\rightsquigarrow}]) \wedge \mathsf{Confl_free}([s \overset{\mathsf{lexmax}}{\rightsquigarrow}]) \wedge \mathcal{L} = D$. This leads to the sorted algorithm by Daciuk, Mihov, and others, appearing in Chapter 9 and page 67.

6. $\mathsf{Struct}_\mathsf{D}(\mathrm{D}) \equiv \mathsf{Inequiv}(\mathrm{DL}_{>\mathsf{minlen}}) \wedge \mathsf{Confl_free}(\mathrm{DL}_{\leqslant \mathsf{minlen}}) \wedge \mathcal{L} = \mathrm{D}$. This leads to a new state-depth based algorithm, presented in Chapter 10 on page 75.
7. $\mathsf{Struct}_\mathsf{W}(\mathrm{D}) \equiv \mathsf{Inequiv}(\mathsf{Succ}^*(\mathrm{F})) \wedge \mathsf{Confl_free}(\mathrm{Q} - \mathsf{Succ}^*(\mathrm{F})) \wedge \mathcal{L} = \mathrm{D}$. This leads to a simplified version of the semi-incremental algorithm by Watson, given here in Chapter 11 page 83.

See Section 1.4 for the mapping from specific structural invariants to the literature.

3.1.2 Function add_word

We now have a specification for add_word (for a given Struct):

$\{\ \mathsf{Struct}(\mathrm{D})\ \}$
$\mathsf{add_word}(w)$
$\{\ \mathsf{Struct}(\mathrm{D} \cup \{w\})\ \}$

For each such body and Struct, we get a precondition. In some cases, the precondition can be established by adding the words in a particular order (a choice of $\leqslant$). For clarity, the versions of add_word will be given names of the form $\mathsf{add_word}_\mathsf{X}$ where X is the corresponding subscript of Struct.

3.1.3 Function cleanup

Given Struct, we also have a specification for cleanup:

$\{\ \mathsf{Struct}(W)\ \}$
cleanup
$\{\ \mathsf{Min} \wedge \mathcal{L} = W\ \}$

The versions of cleanup will be given names of the form $\mathsf{cleanup}_\mathsf{X}$ where X is the corresponding subscript of Struct.

3.2 Commentary

The common algorithm skeleton is a key aspect of the algorithm presentation in this thesis. All of the presently known algorithms have been successfully cast in this framework, and there is every reason to believe that newly discovered algorithms will also fit within this or a similar taxonomy.

Chapter 4

Trie intermediate ADFA

In this chapter, we maintain M as a trie during construction — using structural invariant

$$\mathsf{Struct_T(D)} \equiv \mathsf{Is_trie} \wedge \mathcal{L} = \mathsf{D}$$

Following the construction of the trie using $\mathsf{add_word_T}$, procedure $\mathsf{cleanup_T}$ merges equivalent states.

4.1 Procedure $\mathsf{add_word_T}$

For $\mathsf{add_word_T}$, we get specification (using shadow variable L to express the postcondition)

---

proc $\mathsf{add_word_T}(\textbf{in}\ w : \Sigma^*) \rightarrow$
  { **pre** $\mathsf{Is_trie} \wedge \mathcal{L} = \mathsf{L}$ }
  $S_{4.1}$
  { **post** $\mathsf{Is_trie} \wedge \mathcal{L} = \mathsf{L} \cup \{w\}$ }
corp

To make the final implementation $\mathsf{add_word_T}$ and $S_{4.1}$ reusable in other chapters, we weaken the precondition $\mathsf{Is_trie} \wedge \mathcal{L} = \mathsf{L}$ to

$$\mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}]) \wedge \mathcal{L} = \mathsf{L}$$

(This is a weakening because no state in a trie is a confluence, so no state in the $[s \overset{w}{\rightsquigarrow}]$ path is a confluence.) For the same reusability reasons, we will derive $S_{4.1}$ such that it only affects $[s \overset{w}{\rightsquigarrow}]$. This allows us to weaken the postcondition to express the following: if $\mathsf{Is_trie}$ holds before $\mathsf{add_word_T}$ is invoked then $\mathsf{Is_trie}$ will hold after. This postcondition gives us the inductive property that $\mathsf{Is_trie}$ holds after each word in W is added when $\mathsf{add_word_T}$ is used to construct M starting with the empty ADFA. (Note that the empty ADFA is also a trie.)

This gives us the following specification in which we use shadow variable M' to express the postcondition

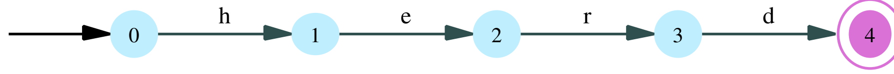
---

proc $\mathsf{add_word_T}(\textbf{in}\ w : \Sigma^*) \rightarrow$
  { **pre** $\mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}]) \wedge \mathcal{L} = \mathsf{L} \wedge M = M'$ }
  $S_{4.1}$
  { **post** $(\mathsf{Is_trie}(M') \Rightarrow \mathsf{Is_trie}) \wedge \mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}]) \wedge \mathcal{L} = \mathsf{L} \cup \{w\}$ }
corp

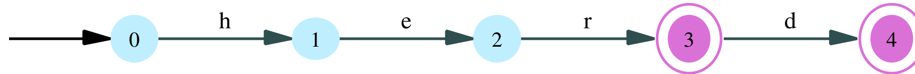
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

The simplest way to proceed in refining $S_{4.1}$ is to introduce a new state variable q, establish $q = \delta^*(s, w) \wedge q \neq \perp$ and then make q a final state (so that the ADFA accepts w), as in the following example.

Example 4.1 (Adding a prefix word) *Assume we initially have the following* ADFA *accepting* `herd`*:*



We wish to add the word `her`*, which is a prefix of* `herd`*. This results in state* 3 *becoming a final one, as in:*



We therefore have the following procedure

---

proc $\mathsf{add_word}_T(\textbf{in}\ w : \Sigma^*) \rightarrow$
 $\{\ \textbf{pre}\ \mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}]) \wedge \mathcal{L} = L \wedge M = M'\ \}$
 $[\![\ \textbf{var}\ q : \textsc{State}$
 $|\ S'_{4.1};$
  $\{\ q = \delta^*(s, w) \wedge q \neq \perp\ \}$
  $F := F \cup \{q\}$
 $]\!]$
 $\{\ \textbf{post}\ (\mathsf{Is_trie}(M') \Rightarrow \mathsf{Is_trie}) \wedge \mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}]) \wedge \mathcal{L} = L \cup \{w\}\ \}$
corp

We can continue our derivation with $S'_{4.1}$.

4.1.1 Adding only prefix words

In this section only, we assume that w is a prefix of a word already accepted by (Q, δ, s, F) — that is $\delta^*(s, w) \neq \perp$. Clearly, this is an unrealistic assumption — it is rarely applicable — but it forms a good starting point for a simple algorithm. We also introduce two additional variables $l, r : w = lr$ and maintain invariant $q = \delta^*(s, l)$, giving the following for $S'_{4.1}$

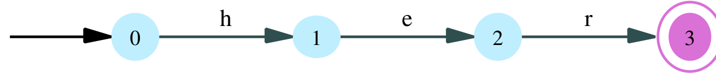
$\vdots$
$\{\ \delta^*(s, w) \neq \perp\ \}$
$[\![\ \textbf{var}\ l, r : \Sigma^*$
$|\ l, r, q := \varepsilon, w, s;$
 $\{\ \text{invariant: } w = lr \wedge q = \delta^*(s, l) \wedge q \neq \perp$
  $\text{variant: } |r|\ \}$
 $\textbf{do}\ r \neq \varepsilon \rightarrow$
  $l, r, q := l \cdot \mathsf{head}(r), \mathsf{tail}(r), \delta(q, \mathsf{head}(r))$
 $\textbf{od}$
$]\!]$
$\{\ q = \delta^*(s, w) \wedge q \neq \perp\ \}$
$\vdots$

Of course, precondition $\delta^*(s, w) \neq \perp$ cannot always be established just by adding the words in a certain order (i.e. by choosing $\leqslant$) and so we generalize this algorithm in the next section.

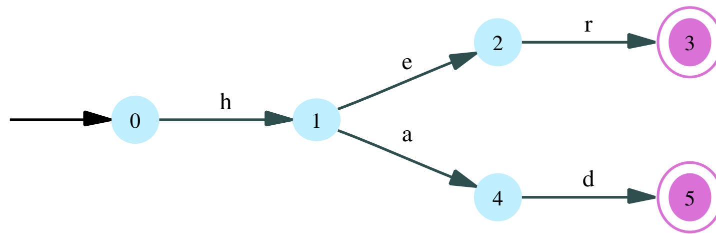
4.1.2 Adding a nonprefix word in a trie

In the case $\delta^*(s, w) = \bot$, we begin by finding the longest prefix l of w such that $\delta(s, l) \neq \bot$, then build additional states and transitions if required, as in the following example

Example 4.2 (Adding a word causing a create) *Initially, we have the following* ADFA *accepting* `her`*:*



We wish to add the word `had`*. The (longest common) prefix* `h` *(of* `had` *and* `her`*) lies on a path to state* 1*, at which point we are stuck and new states* 4 *and* 5 *must be created, eventually giving:*



To express that 'l is the longest prefix on a path reachable from s,' we use the following (using the invariant $q = \delta^*(s, l)$)

$$q \neq \bot \wedge (r = \varepsilon \textbf{ cor } \delta(q, \mathsf{head}(r)) = \bot)$$

Intuitively, this means that there is a full l-path from the start state s, and that either we have run out of symbols to consider (that is $r = \varepsilon$) or no further transitions are possible and we are stuck in state q.

Instead of our previous refinement of $S'_{4.1}$, we obtain

$\vdots$
$[\![$ **var** $l, r : \Sigma^*$
$|$ $S''_{4.1}$;
$\{\ q = \delta^*(s, l) \wedge q \neq \bot \wedge (r = \varepsilon \textbf{ cor } \delta(q, \mathsf{head}(r)) = \bot)\ \}$
$S'''_{4.1}$
$]\!]$
$\{\ q = \delta^*(s, w) \wedge q \neq \bot\ \}$
$\vdots$

Statement $S''_{4.1}$ simply follows the w-path through M until no further transition is possible, then statement $S'''_{4.1}$ extends M as necessary with new states and transitions. The final procedure is

---

proc $\mathsf{add_word}_T(\textbf{in } w : \Sigma^*) \rightarrow$
$\{\ \textbf{pre } \mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}]) \wedge \mathcal{L} = L \wedge M = M'\ \}$
$[\![$ **var** $q : \mathrm{STATE}$
$|$ $[\![$ **var** $l, r : \Sigma^*$
$|$ $l, r, q := \varepsilon, w, s$;
$\{$ invariant: $w = lr \wedge q = \delta^*(s, l) \wedge q \neq \bot$

```
            variant: |r| }
          do r ≠ ε cand δ(q, head(r)) ≠ ⊥ →
              l, r, q := l · head(r), tail(r), δ(q, head(r))
          od;
          { q = δ*(s, l) ∧ q ≠ ⊥ ∧ (r = ε cor δ(q, head(r)) = ⊥) }
          { invariant: w = lr ∧ q = δ*(s, l) ∧ q ≠ ⊥
            variant: |r| }
          do r ≠ ε → |[ var p : STATE
                       | p := create();
                         δ(q, head(r)), q := p, p;
                         l, r := l · head(r), tail(r)
                       ]|
          od
        ]|;
        { q = δ*(s, w) ∧ q ≠ ⊥ }
        F := F ∪ {q}
      ]|
      { post (Is_trie(M′) ⇒ Is_trie) ∧ Confl_free([s ⇝w]) ∧ 𝓛 = L ∪ {w} }
  corp
```

This algorithm corresponds closely to most trie-construction algorithms — including that sketched by Fredkin, the inventor of tries [Fre60]. A more complete example follows on page 38 in §4.3.

4.2 Procedure $\mathsf{cleanup}_T$

In this section, we consider minimization procedures with specification:

```
  proc cleanup_T() →
      { pre Is_trie ∧ 𝓛 = L }
      S_4.2
      { post Min ∧ 𝓛 = L }
  corp
```

To derive an implementation which will be usable in subsequent chapters as well, we relax the precondition to $\mathcal{L} = L$ — by dropping the first conjunct Is_trie, and allowing M to have confluence states.

We maintain a partition of our states Q into D, T, where D is a set of pairwise inequivalent states (that is $\mathsf{Inequiv}(D)$) which is not shrinking[1]. In each iteration, we choose a set of states N from T (which *is* shrinking), make $D \cup N$ pairwise inequivalent against D, then add N to D:

Algorithm 4.1:

```
  proc cleanup_T() →
      { pre 𝓛 = L }
      |[ var D, T : set of STATE
       | D, T := ∅, Q;
```

[1]Note that we do not state that D is *growing*, since it may in fact remain the same size for many iterations when redundant states are being merged.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
            { invariant: Inequiv(D) ∧ ℒ = L
              variant: |T| }
            do T ≠ ∅ → |[ var N : set of STATE
                        | let N : N ⊆ T ∧ N ≠ ∅;
                          T := T − N;
                          { N ≠ ∅ }
                          N : S'4.2;
                          { Inequiv(D ∪ N) }
                          D := D ∪ N
                          { Inequiv(D) }
                        ]|
            od
        ]|
        { post Min ∧ ℒ = L }
    corp
```

□

This gives a specification for statement $S'_{4.2}$: establish $\mathsf{Inequiv}(D \cup N)$ while changing only N (and implicitly M). Thanks to Property 2.83, we can rewrite $\mathsf{Inequiv}(D \cup N)$ as

$$\underbrace{\mathsf{Inequiv}(D)}_{\text{in invariant}} \wedge \underbrace{\mathsf{Inequiv}(N)}_{\textbf{let}\text{ in §4.2.1}} \wedge \underbrace{\mathsf{Pairwise_inequiv}(D, N)}_{\textbf{let}\text{ in §4.2.2}}$$

Conjunct $\mathsf{Inequiv}(D)$ is already in the repetition invariant, so we ignore it in refining $S'_{4.2}$ as our refined statement cannot change D. Of the remaining two conjuncts, we can move one of them into the **let** statement which selects N in the first place, thereby simplifying $S'_{4.2}$. In the following sections, we consider those two possibilities (i.e. which conjunct is moved into the **let** statement) by focusing on the body of the repetition in Algorithm 4.1.

4.2.1 Selecting $N : N \subseteq T \wedge N \neq \emptyset \wedge \mathsf{Inequiv}(N)$

If we place $\mathsf{Inequiv}(N)$ in the **let** statement condition (that is, we select state set N such that they are already known to be pairwise inequivalent), we obtain the following refinement in Algorithm 4.1:

```
    proc cleanupT,Inequiv(N)() →
        { pre ℒ = L }
        |[ var D, T : set of STATE
        | D, T := ∅, Q;
          { invariant: Inequiv(D) ∧ ℒ = L
            variant: |T| }
          do T ≠ ∅ → |[ var N : set of STATE
                      | let N : N ⊆ T ∧ N ≠ ∅ ∧ Inequiv(N);
                        T := T − N;
                        { N ≠ ∅ ∧ Inequiv(N) }
                        N : S4.2.1;
```

$$\{\ \overbrace{\underbrace{\mathsf{Inequiv}(D)}_{\text{invariant}} \wedge \underbrace{\mathsf{Inequiv}(N)}_{\textbf{let}\text{ statement}} \wedge \underbrace{\mathsf{Pairwise_inequiv}(D, N)}_{\text{establish in } S_{4.2.1}}}^{\mathsf{Inequiv}(D \cup N)}\ \}$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
                        D := D ∪ N
                        { Inequiv(D) }
                      ]|
          od
        ]|
        { post Min ∧ ℒ = L }
corp
```

The easiest implementation of the **let** statement is to choose $N : \mathsf{Inequiv}(N)$ as a single state $p : p \in T$, in which case $\mathsf{Inequiv}(\{p\})$ holds trivially. An alternative is to choose a path of states $[r \overset{x}{\rightsquigarrow}] \subseteq T$ (for some state r and string x). We consider each of these two possibilities in the next sections.

4.2.1.1 Selecting a single state

When selecting a single state, the resulting statement is:

```
⋮
{ invariant: Inequiv(D) ∧ ℒ = L
  variant: |T| }
do T ≠ ∅ → |[ var N : set of STATE
            | let N : N ⊆ T ∧ |N| = 1;
              T := T − N;
              { N ≠ ∅ ∧ Inequiv(N) }
              N : S_{4.2.1.1};
              { Inequiv(D) ∧ Inequiv(N) ∧ Pairwise_inequiv(D, N) }
              D := D ∪ N
              { Inequiv(D) }
            ]|
od
⋮
```

For clarity, we add a local state variable $p : N = \{p\}$ and use it almost everywhere[2] in place of N.

```
⋮
{ invariant: Inequiv(D) ∧ ℒ = L
  variant: |T| }
do T ≠ ∅ → |[ var N : set of STATE;
              var p : STATE
            | let N, p : N ⊆ T ∧ N = {p};
              T := T − {p};
              { {p} ≠ ∅ ∧ Inequiv({p}) }
              p : S'_{4.2.1.1};
              { Inequiv(D) ∧ Inequiv(N) ∧ Pairwise_inequiv(D, N) }
              D := D ∪ N
              { Inequiv(D) }
            ]|
od
```

[2] We still use N in the postcondition of $S'_{4.2.1.1}$ because at that point N may be $\emptyset$ if p is equivalent to a state in D.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

⋮

To establish $\mathsf{Pairwise_inequiv}(D, N)$, statement $S'_{4.2.1.1}$ looks for a $q \in D$ equivalent to p; there can be *at most* one such q since $\mathsf{Inequiv}(D)$:

```
    ⋮
    { invariant: Inequiv(D) ∧ ℒ = L
      variant: |T| }
    do T ≠ ∅ → |[ var N : set of STATE;
                  var p : STATE
                | let N, p : N ⊆ T ∧ N = {p};
                  T := T − {p};
                  { {p} ≠ ∅ ∧ Inequiv({p}) }
                  as ⟨∃ q : q ∈ D : E(p, q)⟩ →
                      let q : q ∈ D ∧ E(p, q);
                      merge(p, q);
                      N := ∅
                  sa;
                  { Inequiv(D) ∧ Inequiv(N) ∧ Pairwise_inequiv(D, N) }
                  D := D ∪ N
                  { Inequiv(D) }
                ]|
    od
    ⋮
```

The now-superfluous N can be removed, changing the update of D:

```
    ⋮
    { invariant: Inequiv(D) ∧ ℒ = L
      variant: |T| }
    do T ≠ ∅ → |[ var p : STATE
                | let p : p ∈ T;
                  T := T − {p};
                  { Inequiv({p}) }
                  if ⟨∃ q : q ∈ D : E(p, q)⟩ →
                     let q : q ∈ D ∧ E(p, q);
                     merge(p, q)
                  [] ¬⟨∃ q : q ∈ D : E(p, q)⟩ →
                     { Inequiv(D ∪ {p}) }
                     D := D ∪ {p}
                  fi
                  { Inequiv(D) }
                ]|
    od
    ⋮
```

The guard $\langle \exists\, q : q \in D : E(p, q)\rangle$ can be directly evaluated using function eq (see page 21). Our algorithm would be more efficient if we could use eq' (page 22). According to Property 2.86, we

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

can use eq' iff $\mathsf{Inequiv}(\mathsf{Succ}(\{p, q\}))$. If we add $\mathsf{Succ}(p) \subseteq D$ as a conjunct of the **let** statement selecting p, we inductively get $\mathsf{Succ}(D) \subseteq D$ as an additional invariant conjunct since D monotonically grows and is built-up state-by-state from such $p : \mathsf{Succ}(p) \subseteq D$. Thanks to this new conjunct and our other invariant conjunct, $\mathsf{Inequiv}(D)$, we have the required $\mathsf{Inequiv}(\mathsf{Succ}(\{p, q\}))$ for all $q \in D$. The resulting procedure is (where subscript ss stands for 'single state'):

```
proc cleanup_{T,Inequiv(N),ss}() →
    { pre 𝓛 = L }
    ⟦ var D, T : set of STATE
    | D, T := ∅, Q;
      { invariant: Inequiv(D) ∧ Succ(D) ⊆ D ∧ 𝓛 = L
        variant: |T| }
      do T ≠ ∅ → ⟦ var p : STATE
                 | let p : p ∈ T ∧ Succ(p) ⊆ D;
                   T := T − {p};
                   { Inequiv({p}) ∧ Succ(p) ⊆ D ∧ Inequiv(Succ({p})) }
                   if ⟨∃ q : q ∈ D : eq′(p, q)⟩ →
                      let q : q ∈ D ∧ eq′(p, q);
                      merge(p, q)
                   ▯ ¬⟨∃ q : q ∈ D : eq′(p, q)⟩ →
                      { Inequiv(D ∪ {p}) }
                      D := D ∪ {p}
                   fi
                   { Inequiv(D) }
                 ⟧
      od
    ⟧
    { post Min ∧ 𝓛 = L }
corp
```

We now consider whether such a state $p : \mathsf{Succ}(p) \subseteq D$ can always be selected in the **let**. From the invariant, we have $\mathsf{Inequiv}(D) \wedge \mathsf{Succ}(D) \subseteq D$ and by the repetition guard $T \neq \emptyset$. There are two (mutually exclusive) cases:

1. $HL_0 \not\subseteq D$ — there is a leaf state not yet in D, and $\mathsf{Succ}(HL_0) \subseteq D$ holds trivially since leaves have no successors, in which case we can select p from $HL_0 - D$ as such a leaf state; or

2. $HL_0 \subseteq D$ — let $k > 0$ be the smallest k such that $HL_k \not\subseteq D$. We can select p from HL_k since

$$
\begin{array}{ll}
 & \mathsf{Succ}(p) \\
\subseteq & \quad \text{“Property 2.81”} \\
 & \langle \cup\, j : 0 \leqslant j < k : HL_j \rangle \\
\subseteq & \quad \text{“assumption that } k \text{ is the smallest such that } HL_k \not\subseteq D\text{”} \\
 & D
\end{array}
$$

4.2.1.2 Selecting a path of states

From Corollary 2.79, we can select any path of states for N in the repetition of Algorithm 4.1:

```
⋮
{ invariant: Inequiv(D) ∧ ℒ = L
  variant: |T| }
do T ≠ ∅ → |[ var N : set of STATE
              | let N : N ⊆ T ∧ N ≠ ∅ ∧ N = [r ⇝x] for some r, x;
                T := T − N;
                { N ≠ ∅ ∧ Inequiv(N) }
                N : S_4.2.1.2
                { Inequiv(D) ∧ Inequiv(N) ∧ Pairwise_inequiv(D, N) }
                D := D ∪ N
                { Inequiv(D) }
              ]|
od
⋮
```

An implementation of $S_{4.2.1.2}$ will consider the individual states in $[r \overset{x}{\rightsquigarrow}]$, comparing them for equivalence (using eq) against states in D; those found to be equivalent will be merged, while the inequivalent ones are added to D. The details of such an implementation resemble those in the previous section and are not discussed further here. When $x = \varepsilon$, we have the degenerate path $[r \overset{x}{\rightsquigarrow}] = [r \overset{\varepsilon}{\rightsquigarrow}] = r$, yielding the single-state algorithm of §4.2.1.1.

Interestingly, we could have chosen any sequence of states $p_0, \ldots, p_j : \langle \forall\, i : 0 \leqslant i \leqslant j : p_{i+1} \in \mathsf{Succ}^+(p_i) \rangle$ which form a *reachability chain*. (Note that they need not be immediate successors.) Path $[r \overset{x}{\rightsquigarrow}]$ is just one such reachability chain.

4.2.2 Selecting $N : N \subseteq T \wedge N \neq \emptyset \wedge \mathsf{Pairwise_inequiv}(D, N)$

We return to the repetition of Algorithm 4.1 (page 30), instead placing $\mathsf{Pairwise_inequiv}(D, N)$ in the **let** statement — choose N such that each state therein is inequivalent to the states D already processed (though $\mathsf{Inequiv}(N)$ is still to be established in statement $S_{4.2.2}$). This gives the following:

```
proc cleanup_T,Pairwise_inequiv(D,N)() →
    { pre ℒ = L }
    |[ var D, T : set of STATE
    | D, T := ∅, Q;
      { invariant: Inequiv(D) ∧ ℒ = L
        variant: |T| }
      do T ≠ ∅ → |[ var N : set of STATE
                    | let N : N ⊆ T ∧ N ≠ ∅ ∧ Pairwise_inequiv(D, N);
                      T := T − N;
                      { N ≠ ∅ ∧ Pairwise_inequiv(D, N) }
                      N : S_4.2.2;
                                        Inequiv(D∪N)
                      { Inequiv(D) ∧ Inequiv(N) ∧ Pairwise_inequiv(D, N) }
                        invariant     S_4.2.2       let statement
                      D := D ∪ N
                      { Inequiv(D) }
                    ]|
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
            od
          ]|
          { post Min ∧ 𝓛 = L }
    corp
```

We focus on selecting N before returning to an implementation of $S_{4.2.2}$. From Property 2.80 (page 20), we know that (for $k \geqslant 0$) $\text{Pairwise_inequiv}(HL_{k+1}, HL_k)$. It follows that one way to select N is to proceed in ascending levels (i.e. starting with the leaves HL_0) within the ADFA. For this, we add variable k and select $N = HL_k$. Variable D accumulates the states from all visited levels, that is $D = \langle \cup\, j : 0 \leqslant j < k : HL_j \rangle$, while $T = \langle \cup\, j : j \geqslant k : HL_j \rangle$. Clearly, we then have $\text{Pairwise_inequiv}(D, N)$, which we express as $\text{Pairwise_inequiv}(D, HL_k)$.

```
    ⋮
    k := 0;
    { invariant: Inequiv(D) ∧ 𝓛 = L
                   ∧ D = ⟨∪ j : 0 ⩽ j < k : HL_j⟩
                   ∧ T = ⟨∪ j : j ⩾ k : HL_j⟩
                   ∧ Pairwise_inequiv(D, HL_k)
                     └──── Property 2.80 ────┘
      variant: |T| }
    do T ≠ ∅ → |[ var N : set of STATE
               | let N : N ⊆ T ∧ N = HL_k;
                 T := T − N;
                 { N ≠ ∅ ∧ Pairwise_inequiv(D, N) }
                           └── now in invariant ──┘
                 N : S'_4.2.2;
                 { Inequiv(D) ∧ Inequiv(N) ∧ Pairwise_inequiv(D, N) }
                                └ S'_4.2.2 ┘
                 D := D ∪ N;
                 { Inequiv(D) }
                 k := k + 1
               ]|
    od
    ⋮
```

From Property 2.81, we have

$$\text{Succ}(HL_k) \subseteq \langle \cup\, j : 0 \leqslant j < k : HL_j \rangle$$

That is, $\text{Succ}(HL_k) \subseteq D$. As in §4.2.1, we introduce invariant conjunct $\text{Succ}(D) \subseteq D$. (This is thanks to the monotonic growth of D and because it is built up from height levels where $\text{Succ}(HL_k) \subseteq D$.) With $\text{Inequiv}(D)$ from our invariant, we also have $\text{Inequiv}(\text{Succ}(HL_k))$. Since N is now redundant, we can eliminate it in favour of HL_k everywhere[3].

```
    ⋮
    k := 0;
    { invariant: Inequiv(D) ∧ 𝓛 = L
```

[3]Here, we assume we have some way of implementing HL_k easily and cheaply.

$$\begin{aligned}
&\wedge D = \langle \cup j : 0 \leqslant j < k : HL_j \rangle \\
&\wedge T = \langle \cup j : j \geqslant k : HL_j \rangle \\
&\wedge \mathsf{Pairwise_inequiv}(D, HL_k) \\
&\wedge \mathsf{Succ}(HL_k) \subseteq D \\
&\wedge \mathsf{Succ}(D) \subseteq D \\
&\wedge \mathsf{Inequiv}(\mathsf{Succ}(HL_k))
\end{aligned}$$

variant: $|T|$ }

do $T \neq \emptyset \rightarrow$
$T := T - HL_k;$
$\{\ HL_k \neq \emptyset\ \}$
$HL_k : S''_{4.2.2};$
$\{\ \underbrace{\mathsf{Inequiv}(D)}_{\text{invariant}} \wedge \underbrace{\mathsf{Inequiv}(HL_k)}_{S''_{4.2.2}} \wedge \underbrace{\mathsf{Pairwise_inequiv}(D, HL_k)}_{\text{invariant}}\ \}$
$D := D \cup HL_k;$
$\{\ \mathsf{Inequiv}(D)\ \}$
$k := k + 1$
od
⋮

With our invariant conjunct $T = \langle \cup j : j \geqslant k : HL_j \rangle$ and Property 2.56, we can change our repetition guard from $T \neq \emptyset$ to $HL_k \neq \emptyset$.

In $S''_{4.2.2}$, we will iterate over $p, q \in HL_k$, considering them for equivalence to each other. Since $\mathsf{Inequiv}(\mathsf{Succ}(HL_k))$ holds, we can directly use the simpler form for $E(p, q)$ (see Property 2.86) and therefore eq'. Recall from Property 2.55 that merging states does not change their height levels. $S''_{4.2.2}$ is now:

⋮
$\{\ HL_k \neq \emptyset\ \}$
$[\![$ **var** $p, q : \mathrm{STATE}$
$|$ **for** $p, q : p, q \in HL_k \rightarrow$
as $\mathsf{eq}'(p, q) \rightarrow \mathsf{merge}(p, q)$ **sa**
rof
$]\!];$
$\{\ \underbrace{\mathsf{Inequiv}(D)}_{\text{invariant}} \wedge \underbrace{\mathsf{Inequiv}(HL_k)}_{S''_{4.2.2}} \wedge \underbrace{\mathsf{Pairwise_inequiv}(D, HL_k)}_{\text{invariant}}\ \}$
⋮

The resulting algorithm, closely related to the one first presented by Revuz in [Rev92], is

Algorithm 4.2 (Revuz-like):

---

proc $\mathsf{cleanup}_{T,\mathsf{Pairwise_inequiv}(D,HL_k)}() \rightarrow$
$\{$ **pre** $\mathcal{L} = L\ \}$
$[\![$ **var** $D, T :$ **set of** $\mathrm{STATE}; k : \mathbb{N}$
$|\ D, T := \emptyset, Q;$
$k := 0;$
$\{$ invariant: $\mathsf{Inequiv}(D) \wedge \mathcal{L} = L$
$\wedge D = \langle \cup j : 0 \leqslant j < k : HL_j \rangle$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

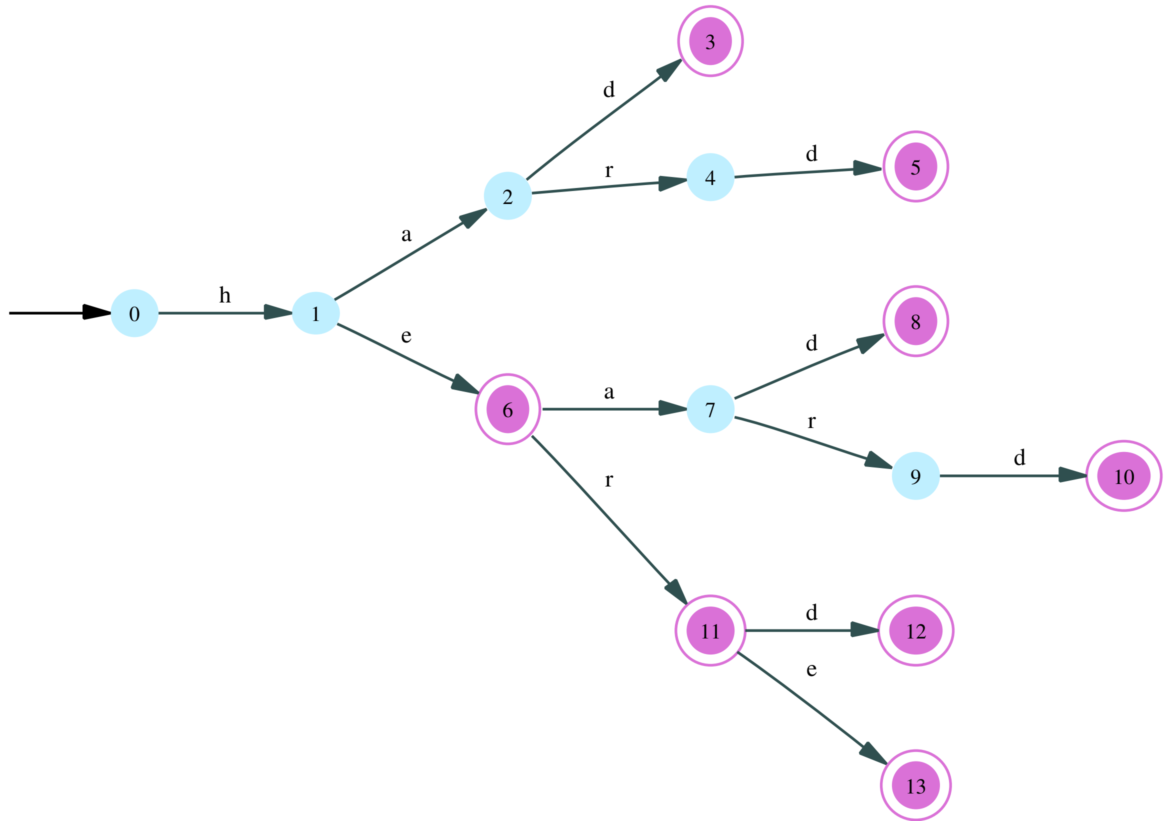

```
                         ∧ T = ⟨∪ j : j ⩾ k : HL_j⟩
                         ∧ Pairwise_inequiv(D, HL_k)
                         ∧ Succ(HL_k) ⊆ D
                         ∧ Inequiv(Succ(HL_k))
                         ∧ Succ(D) ⊆ D
          variant: |T| }
        do HL_k ≠ ∅ →
             T := T − HL_k;
             { HL_k ≠ ∅ }
             |[ var p, q : STATE
             | for p, q : p, q ∈ HL_k →
                    as eq′(p, q) → merge(p, q) sa
               rof
             ]|;
             { Inequiv(D) ∧ Inequiv(HL_k) ∧ Pairwise_inequiv(D, HL_k) }
             D := D ∪ HL_k;
             { Inequiv(D) }
             k := k + 1
        od
      ]|
      { post Min ∧ ℒ = L }
corp
```

□

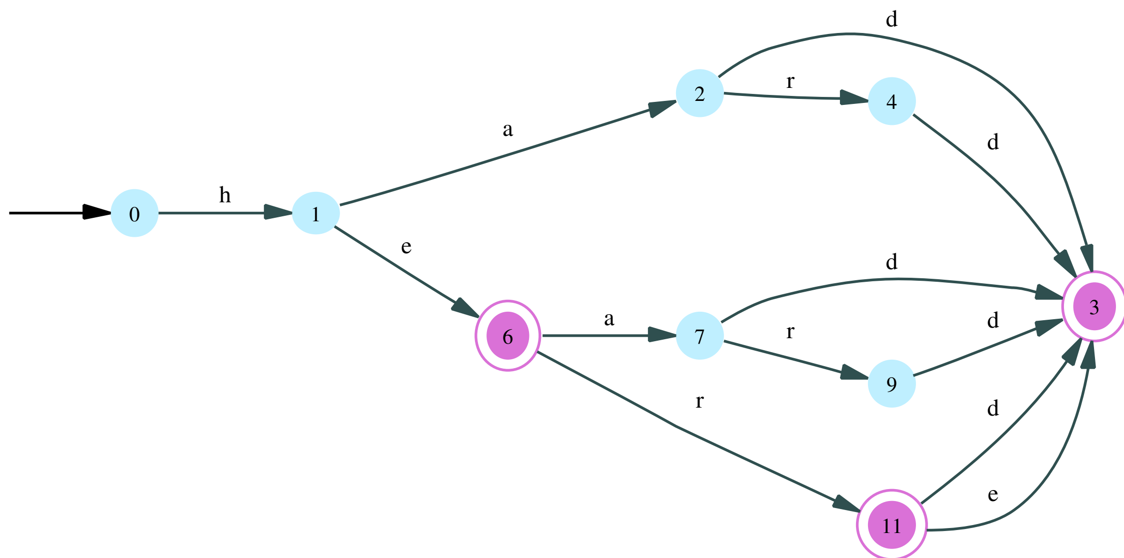
In [Rev92], Revuz shows that eq′ can be implemented using some clever coding tricks. Graña *et al* [GBA01] and Bubenzer [Bub11] make further improvements to this algorithm.

4.3 An example

In this section, we present an extended example. After adding `had`, `hard`, `head`, `heard`, `herd`, `here`, `her`, `he` using $\mathsf{add_word}_T{}'$, the resulting trie is

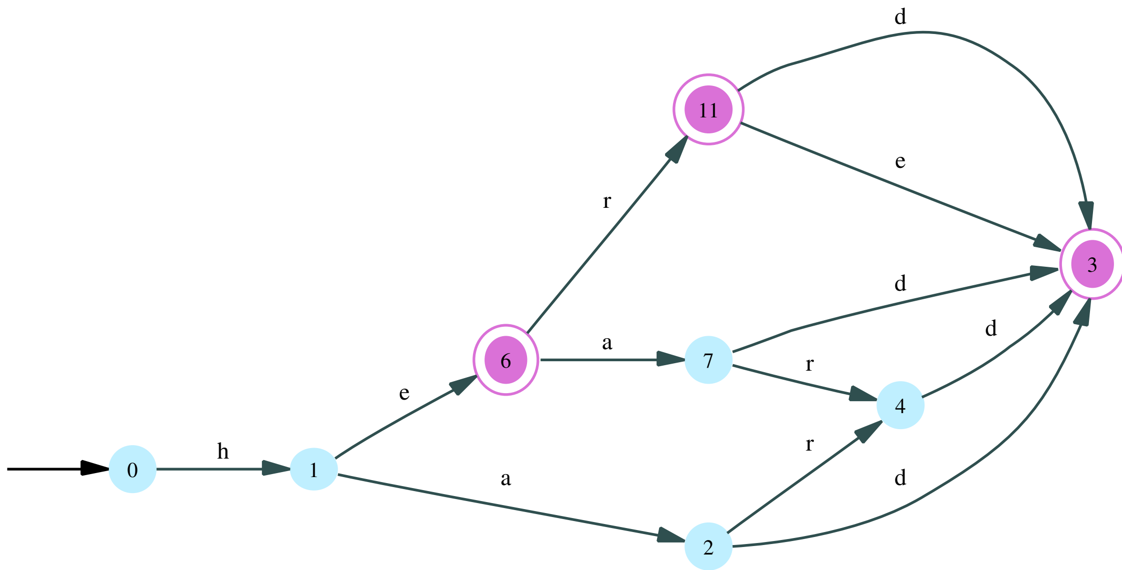


Now, consider Revuz's algorithm applied (as $\text{cleanup}_{T,\text{Pairwise_inequiv}(D,N)}$) to the trie given above. Initially, we consider $HL_0 = \{3, 5, 8, 10, 12, 13\}$, all of which can be merged, yielding

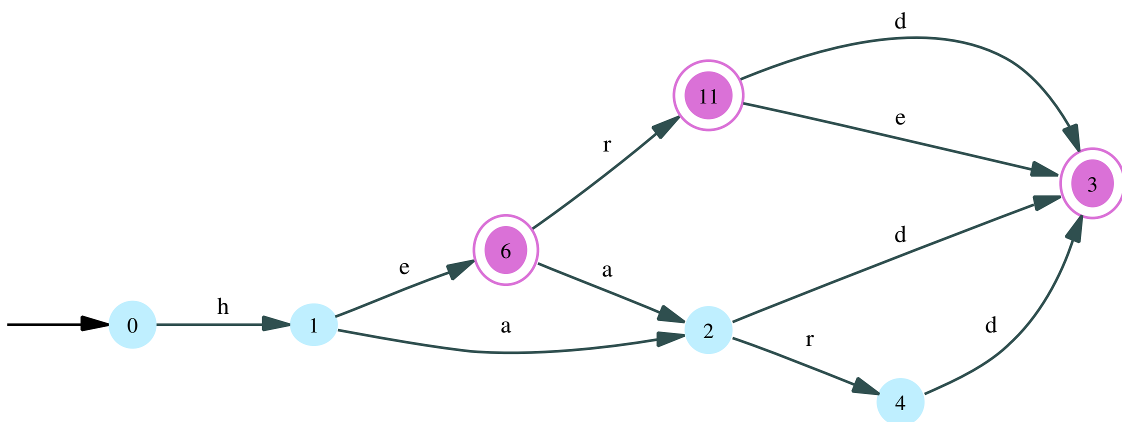


UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

We now examine the states in $\text{HL}_1 = \{4, 9, 11\}$ against themselves, and 9 is merged into 4 (note that merging could have occurred vice-versa, with 4 merged into 9); state 11 is not merged since it is final and the other two are not. The resulting automaton is



Considering $\text{HL}_2 = \{2, 7\}$, we find that the two states are equivalent and they are merged, giving



After considering $\text{HL}_3 = \{6\}, \text{HL}_4 = \{1\}, \text{HL}_5 = \{0\}$, the above ADFA is minimal.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

4.4 Time and space performance

For word w, invocation $\mathsf{add_word}_T(w)$ adds $\mathcal{O}(|w|)$ states in $\mathcal{O}(|w|)$ time if we assume that $\delta(p,a)$ and create take constant time. It follows that building M from word set W yields a trie of $\mathcal{O}(\sum_{w \in W} |w|)$ states in the same order of time. Revuz's version of $\mathsf{cleanup}_T$ can be shown to take time $\mathcal{O}(|M|)$ and can be implemented to take space corresponding to the longest word in W. It follows that the construction *and* minimization of M takes $\mathcal{O}(\sum_{w \in W} |w|)$ time and space — a surprising result given that minimization of an arbitrary DFA Z takes time $\mathcal{O}(|Z| \log |Z|)$.

4.4.1 Improvements

There are numerous improvements which can be made to the implementation of eq' and all versions of $\mathsf{cleanup}_T$, some of which are discussed in [Rev92, Dac98, GBA01, DMWW00, Bub11]:

- The states to be considered are sorted according to their level and whether or not they are final.
- The first iteration *always* merges all leaf states. This can be factored out to a separate statement before the iteration.
- The start state s is *never* equivalent to any other state, and need not be considered at all.
- In the implementation selecting N as a path (§4.2.1.2), there may be efficient orders of considering the states in N which are as yet unknown, for example, from the head to the tail of the path.

Significant further improvements were made by Bubenzer and reported in [Bub11].

4.5 Commentary

The algorithms presented in this chapter are the most rudimentary ones. As a result, they are rarely used in industrial-strength applications, though the implementations of $\mathsf{cleanup}_T$ are interesting in their own right for minimizing arbitrary ADFAs which have already been constructed.

Chapter 5

Arbitrary intermediate ADFA

In this chapter, we consider adding a word to an arbitrary ADFA — one in which confluences may be encountered (when adding word w) on the w-path — though we will make use of the fact that s (the start state) cannot be a confluence due to acyclicity[1]. The structural predicate is simply

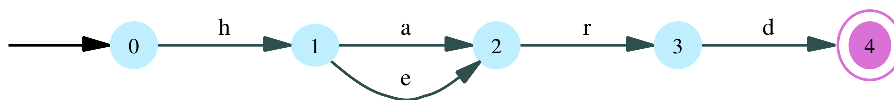
$$\mathsf{Struct}_{\mathsf{N}}(D) \equiv \mathcal{L} = D$$

Following construction, procedure $\mathsf{cleanup}_{\mathsf{N}}$ merges states in the same way as $\mathsf{cleanup}_{\mathsf{T}}$ does in Chapter 4.

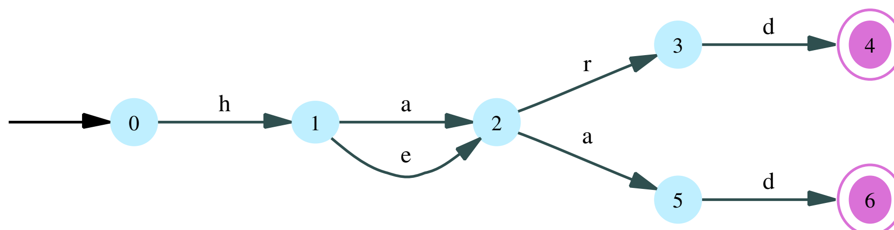
5.1 Procedure $\mathsf{add_word}_{\mathsf{N}}$

Without modification, the algorithms of Chapter 4 ($\mathsf{add_word}_{\mathsf{T}}$ and variants) may add words accidentally if a confluence state is encountered. Consider the following example.

Example 5.1 (Adding words accidentally) *Initially, we have the following* ADFA *accepting* `hard` *and* `herd` *(without useless states, at least two words are necessary to have a confluence state):*



While adding the new word `head`*, we arrive at confluence state* 2*. From state* 2*, there is no* a *out-transition and we naïvely extend the automaton, yielding:*

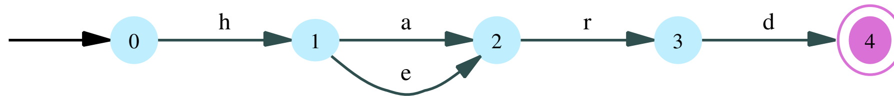


[1] Strictly, it *is* possible for s to be a confluence state if useless states are introduced, though we have excluded that case in this thesis.

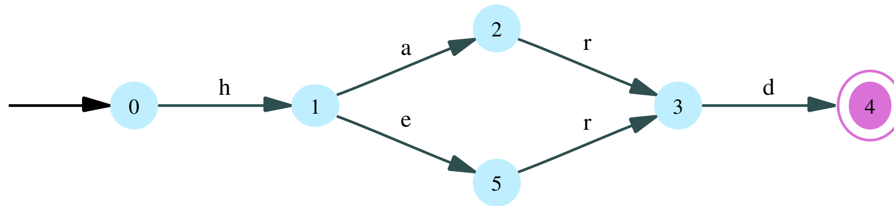
This ADFA *incorrectly also accepts* `haad`.

Clearly, a ‘cloning’ operation is required (see page 10 for the definition of cloning), as we see in the following corrected example.

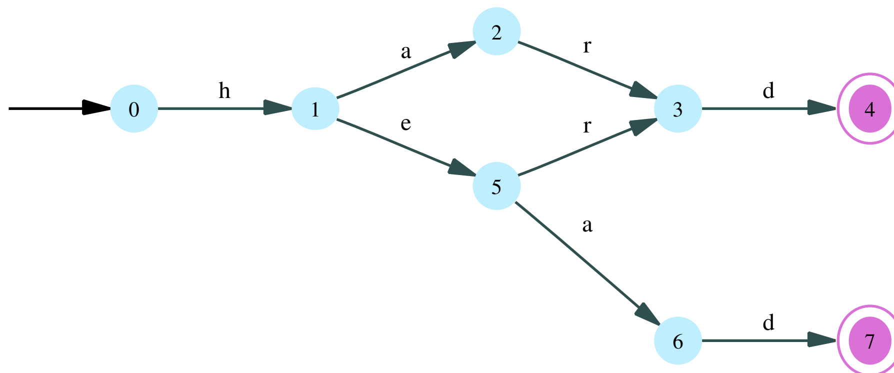
Example 5.2 (Adding a word causing a clone**)** *As in the previous example, we begin with the following* MADFA *accepting* `hard` *and* `herd`*:*



While adding the new word `head`*, we arrive at confluence state* 2 *which is cloned, yielding new state* 5*:*



Two additional states are then added — giving the final automaton:



When modifying function $\mathsf{add_word}_T$ (from Chapter 4 on page 27), the algorithm needs to clone confluence states, and confluence states can only be encountered in the first repetition since the second repetition is only creating new states — *none* of which can be a confluence state. We can modify the first repetition accordingly yielding the procedure body:

---

proc $\mathsf{add_word}_N(\mathbf{in}\ w : \Sigma^*) \rightarrow$
  $\{\ \mathbf{pre}\ \mathcal{L} = L\ \}$
  $[\![\ \mathbf{var}\ q : \textsc{State}$
  $|\ [\![\ \mathbf{var}\ l, r : \Sigma^*;\ \ p : \textsc{State}$
    $|\ l, r, q := \varepsilon, w, s;$
      $\{\ \text{invariant: } w = lr \wedge q = \delta^*(s, l) \wedge q \neq \bot \wedge \mathsf{Confl_free}([s \overset{l}{\rightsquigarrow}])$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
               variant: |r| }
             do r ≠ ε cand δ(q, head(r)) ≠ ⊥ →
                 p := δ(q, head(r));
                 as Is_confl(p) →
                     p := clone(p);
                     δ(q, head(r)) := p
                 sa;
                 q := p;
                 l, r := l · head(r), tail(r)
             od;
             { Confl_free([s ~w~>]) }
             { q = δ*(s, l) ∧ q ≠ ⊥ ∧ (r = ε cor δ(q, head(r)) = ⊥) }
             { invariant: w = lr ∧ q = δ*(s, l) ∧ q ≠ ⊥
               variant: |r| }
             do r ≠ ε → |[ var p : STATE
                         | p := create();
                           δ(q, head(r)), q := p, p;
                           l, r := l · head(r), tail(r)
                         ]|
             od
           ]|;
           { q = δ*(s, w) ∧ q ≠ ⊥ }
           F := F ∪ {q}
         ]|
         { post Confl_free([s ~w~>]) ∧ ℒ = L ∪ {w} }
corp
```

This, however, is also subject to improvement thanks to another observation:

> Once we encounter a confluence state on the w path and clone it, all subsequent states on the path (other than newly created ones) will also be confluences and will have to be cloned.

To see why this holds, consider confluence state p on the w-path: state clone(p) has out-transitions with the same labels and destinations as p, making each of p's successors also a confluence with in-transitions from *at least* p and clone(p).

For this reason, we can again split the first of the above repetitions into two sequentially composed repetitions, in our final algorithm:

```
proc add_word_N'(in w : Σ*) →
         { pre ℒ = L }
         |[ var q : STATE
         | |[ var l, r : Σ*; p : STATE
           | l, r, q := ε, w, s;
             { invariant: w = lr ∧ q = δ*(s, l) ∧ q ≠ ⊥ ∧ Confl_free([s ~l~>])
               variant: |r| }
             do r ≠ ε cand δ(q, head(r)) ≠ ⊥ cand ¬Is_confl(δ(q, head(r))) →
                 l, r, q := l · head(r), tail(r), δ(q, head(r))
             od;
```

```
            { invariant: w = lr ∧ q = δ*(s, l) ∧ q ≠ ⊥ ∧ Confl_free([s ⇝^l])
              variant: |r| }
            do r ≠ ε cand δ(q, head(r)) ≠ ⊥ →
                { Is_confl(δ(q, head(r))) }
                p := δ(q, head(r));
                { Is_confl(p) }
                p := clone(p);
                δ(q, head(r)), q := p, p;
                l, r := l · head(r), tail(r)
            od;
            { Confl_free([s ⇝^w]) }
            { q = δ*(s, l) ∧ q ≠ ⊥ ∧ (r = ε cor δ(q, head(r)) = ⊥) }
            { invariant: w = lr ∧ q = δ*(s, l) ∧ q ≠ ⊥
              variant: |r| }
            do r ≠ ε → p := create();
                       δ(q, head(r)), q := p, p;
                       l, r := l · head(r), tail(r)
            od
        ]|;
        { q = δ*(s, w) ∧ q ≠ ⊥ }
        F := F ∪ {q}
    ]|
    { post Confl_free([s ⇝^w]) ∧ ℒ = L ∪ {w} }
corp
```

Implementation 5.3 *This algorithm always clones confluences, which proves to be inefficient if they are subsequently found to be equivalent (and therefore merged). High-performance implementations of this algorithm perform a 'lazy cloning' (also known as 'virtual cloning') operation, substantially improving the performance [DMWW00].*

5.2 Procedure $\mathsf{cleanup}_N$

As in Chapter 4, for $\mathsf{cleanup}_N$ we can use any one of the general minimization algorithms from [Wat95] or a version of $\mathsf{cleanup}_T$ from §4.2.

5.3 Time and space performance

For word w, invocation $\mathsf{add_word}_N(w)$ adds $\mathcal{O}(|w|)$ states in $\mathcal{O}(|w|)$ time. It follows that building M from word set W yields an ADFA in $\mathcal{O}(\sum_{w \in W} |w|)$ time and space. The most efficient implementations of $\mathsf{cleanup}_T$ in §4.2 take $\mathcal{O}(|M|)$ time and space. It follows that the construction and minimization of M takes $\mathcal{O}(\sum_{w \in W} |w|)$ time and space.

5.4 Commentary

If the MADFA is built from scratch, $\mathsf{add_word}_N$ is uninteresting since the initial ADFA will be a trie in which no confluences occur. Procedure $\mathsf{add_word}_N$ is primarily interesting for adding words to

an ADFA in which some confluences already occur from previous minimization steps; $\mathsf{add_word}_N$ will also be used in Chapter 6 to derive an incremental algorithm. Interestingly, $\mathsf{add_word}_N$ works on cyclic DFA's.

Chapter 6

Minimal intermediate ADFA

This chapter presents an incremental algorithm. We maintain structural invariant

$$\mathsf{Struct_I}(D) \equiv \mathsf{Min} \wedge \mathcal{L} = D$$

Given that M is already minimal in the invariant (while adding words), $\mathsf{cleanup_I}$ is reduced to a **skip** statement.

6.1 Procedure $\mathsf{add_word_I}$

Our starting point for $\mathsf{add_word_I}$ is

---

proc $\mathsf{add_word_I}$(**in** $w : \Sigma^*$) $\rightarrow$
  { **pre** $\mathsf{Min} \wedge \mathcal{L} = L$ }
  $S_{6.1}$
  { **post** $\mathsf{Min} \wedge \mathcal{L} = L \cup \{w\}$ }
corp

In $S_{6.1}$, we will first use $\mathsf{add_word_N}'$ (from page 45) to add w, since M is initially minimal (that is, $\mathsf{Inequiv}(Q)$) and we may encounter confluences on the w-path. After w has been added, we consider those states whose right languages have changed and may now be equivalent to another — merging them and thereby restoring minimality.

Looking at Example 5.2 (page 44), it is easy to see that the only states with changed right languages are precisely those on the w-path, namely $[s \stackrel{w}{\rightsquigarrow}]$, and the remaining states will still be inequivalent. In other words, after w has been added, we have

$$\mathsf{Inequiv}(Q - [s \stackrel{w}{\rightsquigarrow}])$$

Actually, thanks to Property 2.70 we know that s is inequivalent to all other states, and we could have written $\mathsf{Inequiv}(Q - (s \stackrel{w}{\rightsquigarrow}])$, which leads to a slightly more efficient algorithm; for simplicity, we initially ignore that in this chapter. We also have some important properties of the states on path $[s \stackrel{w}{\rightsquigarrow}]$:

1. $\mathsf{Inequiv}([s \stackrel{w}{\rightsquigarrow}])$ — that is, no state on the path is equivalent to any other state on the path[1]. This is given in Corollary 2.79.

[1] States on $[s \stackrel{w}{\rightsquigarrow}]$ may, however, be equivalent to others in M.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

2. $\mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}])$ — that is, no state on the path is a confluence. This follows from the postcondition of $\mathsf{add_word}_N{}'$ (see page 45).

3. No state in $Q - [s \overset{w}{\rightsquigarrow}]$ has a transition to a state in $[s \overset{w}{\rightsquigarrow}]$. This follows from the second property above and Property 2.48.

Our new algorithm (in which we have strengthened post-condition of $\mathsf{add_word}_N{}'$ based on the discussion above) is

---

proc $\mathsf{add_word}_I(\textbf{in } w : \Sigma^*) \rightarrow$
$\{$ **pre** $\mathsf{Min} \wedge \mathcal{L} = L$ $\}$
$\mathsf{add_word}_N{}'(w);$
$$\{\ \underbrace{\mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}]) \wedge \mathcal{L} = L \cup \{w\}}_{\text{normal postcondition of } \mathsf{add_word}_N{}'}\ \}$$
$$\{\ \underbrace{\mathsf{Inequiv}(Q - [s \overset{w}{\rightsquigarrow}]) \wedge \mathsf{Inequiv}([s \overset{w}{\rightsquigarrow}])}_{\text{discussion above}}\ \}$$
$S'_{6.1}$
$$\{\ \overbrace{\mathsf{Inequiv}(Q)}^{\mathsf{Min}}\ \}$$
$\{$ **post** $\mathsf{Min} \wedge \mathcal{L} = L \cup \{w\}$ $\}$
corp

Given the precondition of $S'_{6.1}$

$$\mathsf{Inequiv}(Q - [s \overset{w}{\rightsquigarrow}]) \wedge \mathsf{Inequiv}([s \overset{w}{\rightsquigarrow}])$$

and Property 2.83, we need only establish

$$\mathsf{Pairwise_inequiv}(Q - [s \overset{w}{\rightsquigarrow}], [s \overset{w}{\rightsquigarrow}])$$

to equivalently have

$$\mathsf{Inequiv}((Q - [s \overset{w}{\rightsquigarrow}]) \cup [s \overset{w}{\rightsquigarrow}])$$

$\equiv \mathsf{Inequiv}(Q) \equiv \mathsf{Min}$. We can traverse the states $[s \overset{w}{\rightsquigarrow}]$ in several orders, among which from s to $\delta^*(s, w)$ ('top-down') or vice-versa ('bottom-up'); we will shortly see that it makes sense to consider them bottom-up. This is easily done with a recursive procedure (visit_min to be derived in §6.1.1 starting on page 51), traversing them in post-order and merging states found to be equivalent. An invocation $\mathsf{visit_min}(l, r)$ processes states $[\delta^*(s, l) \overset{r}{\rightsquigarrow}]$. (Note the inclusion of the first and last states.) Both l and r are passed recursively for proper bookkeeping and to express our invariant. The specification (where the pre- and postcondition correspond to the context of $S'_{6.1}$) is:

---

proc $\mathsf{visit_min}(\textbf{in } l, r : \Sigma^*) \rightarrow$
$\{$ **pre** $\mathsf{Confl_free}([s \overset{lr}{\rightsquigarrow}]) \wedge \mathsf{Inequiv}(Q - [s \overset{lr}{\rightsquigarrow}]) \wedge \delta^*(s, lr) \neq \bot \wedge \mathcal{L} = L$ $\}$
$S_{6.1.1}$
$\{$ **post** $\mathsf{Inequiv}(Q - [s \overset{l}{\rightsquigarrow})) \wedge \mathcal{L} = L$ $\}$
corp

Clearly, invocation $\mathsf{visit_min}(\varepsilon, w)$ satisfies the specification of $S'_{6.1}$ since, in the postcondition

$$Q - [s \overset{\varepsilon}{\rightsquigarrow}) = Q - \emptyset = Q$$

and we have our final version of $\mathsf{add_word}_I$.

---

proc add_word$_I$(**in** $w : \Sigma^*$) $\to$
  { **pre** Min $\land \mathcal{L} = L$ }
  add_word$_N{}'(w)$;
  { Confl_free$([s \overset{w}{\rightsquigarrow}]) \land \mathcal{L} = L \cup \{w\}$
   $\land$ Inequiv$(Q - [s \overset{w}{\rightsquigarrow}]) \land$ Inequiv$([s \overset{w}{\rightsquigarrow}])$ }
  visit_min(ε, w)
  { **post** Min $\land \mathcal{L} = L \cup \{w\}$ }
corp

6.1.1 Recursive helper procedure visit_min

We can now focus on deriving an implementation for $S_{6.1.1}$ in visit_min. We can rewrite the first conjunct of the visit_min postcondition:

Inequiv$(Q - [s \overset{l}{\rightsquigarrow}))$
$\equiv$ "definition of open ended range $[\ldots)$ and set calculus"
Inequiv$((Q - [s \overset{l}{\rightsquigarrow}]) \cup \{\delta^*(s, l)\})$
$\equiv$ "Property 2.83"
Inequiv$(Q - [s \overset{l}{\rightsquigarrow}]) \land$ Inequiv$(\{\delta^*(s, l)\}) \land$ Pairwise_inequiv$(Q - [s \overset{l}{\rightsquigarrow}], \{\delta^*(s, l)\})$
$\equiv$ "Inequiv always holds on a single state, $\delta^*(s, l)$ in this case"
Inequiv$(Q - [s \overset{l}{\rightsquigarrow}]) \land$ Pairwise_inequiv$(Q - [s \overset{l}{\rightsquigarrow}], \{\delta^*(s, l)\})$

(Note the similarity of this derivation to the reasoning surrounding the specification of visit_min on page 50.) We establish each of these last two conjuncts separately:

---

proc visit_min(**in** $l, r : \Sigma^*$) $\to$
  { **pre** Confl_free$([s \overset{lr}{\rightsquigarrow}]) \land$ Inequiv$(Q - [s \overset{lr}{\rightsquigarrow}]) \land \delta^*(s, lr) \neq \bot \land \mathcal{L} = L$ }
  $S'_{6.1.1}$;
  { Confl_free$([s \overset{l}{\rightsquigarrow}]) \land$ Inequiv$(Q - [s \overset{l}{\rightsquigarrow}]) \land \mathcal{L} = L$ }
  $S''_{6.1.1}$
  { Confl_free$([s \overset{l}{\rightsquigarrow})) \land \underbrace{\text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}]) \land \text{Pairwise_inequiv}(Q - [s \overset{l}{\rightsquigarrow}], \{\delta^*(s, l)\})}_{\equiv \text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow})) \text{ from derivation above}}$
   $\land \mathcal{L} = L$ }
  { **post** Inequiv$(Q - [s \overset{l}{\rightsquigarrow})) \land \mathcal{L} = L$ }
corp

Consider $S'_{6.1.1}$: its postcondition's first conjunct already holds when $r = \varepsilon$, and no deeper recursion is required in this case, making this our recursion termination condition. This leads to the following refinement:

---

proc visit_min(**in** $l, r : \Sigma^*$) $\to$
  { **pre** Confl_free$([s \overset{lr}{\rightsquigarrow}]) \land$ Inequiv$(Q - [s \overset{lr}{\rightsquigarrow}]) \land \delta^*(s, lr) \neq \bot \land \mathcal{L} = L$ }
  as $r \neq \varepsilon \to S'''_{6.1.1}$ **sa**;
  { Confl_free$([s \overset{l}{\rightsquigarrow}]) \land$ Inequiv$(Q - [s \overset{l}{\rightsquigarrow}]) \land \mathcal{L} = L$ }
  $S''_{6.1.1}$

$\{$ $\mathsf{Confl_free}([s \overset{l}{\leadsto})) \wedge \mathsf{Inequiv}(Q - [s \overset{l}{\leadsto}]) \wedge \mathsf{Pairwise_inequiv}(Q - [s \overset{l}{\leadsto}], \{\delta^*(s, l)\})$
$\wedge \mathcal{L} = L$ $\}$
$\{$ **post** $\mathsf{Inequiv}(Q - [s \overset{l}{\leadsto})) \wedge \mathcal{L} = L$ $\}$
corp

We can rewrite the postcondition conjunct of $S'''_{6.1.1}$:

$$\begin{array}{ll} & \mathsf{Inequiv}(Q - [s \overset{l}{\leadsto}]) \wedge \mathcal{L} = L \\ \equiv & \quad \text{“guard } r \neq \varepsilon \text{”} \\ & \mathsf{Inequiv}(Q - [s \overset{l \cdot \mathsf{head}(r)}{\leadsto})) \wedge \mathcal{L} = L \end{array}$$

This last line is established with a recursive invocation

$$\mathsf{visit_min}(l \cdot \mathsf{head}(r), \mathsf{tail}(r))$$

giving

---

proc $\mathsf{visit_min}($**in** $l, r : \Sigma^*) \rightarrow$
$\{$ **pre** $\mathsf{Confl_free}([s \overset{lr}{\leadsto}]) \wedge \mathsf{Inequiv}(Q - [s \overset{lr}{\leadsto}]) \wedge \delta^*(s, lr) \neq \bot \wedge \mathcal{L} = L$ $\}$
as $r \neq \varepsilon \rightarrow \mathsf{visit_min}(l \cdot \mathsf{head}(r), \mathsf{tail}(r))$
$\{$ $\mathsf{Inequiv}(Q - [s \overset{l \cdot \mathsf{head}(r)}{\leadsto}]) \wedge \mathcal{L} = L$ $\}$
sa;
$\{$ $\mathsf{Confl_free}([s \overset{l}{\leadsto}]) \wedge \mathsf{Inequiv}(Q - [s \overset{l}{\leadsto}]) \wedge \mathcal{L} = L$ $\}$
$S''_{6.1.1}$
$\{$ $\mathsf{Confl_free}([s \overset{l}{\leadsto})) \wedge \mathsf{Inequiv}(Q - [s \overset{l}{\leadsto}]) \wedge \mathsf{Pairwise_inequiv}(Q - [s \overset{l}{\leadsto}], \{\delta^*(s, l)\})$
$\wedge \mathcal{L} = L$ $\}$
$\{$ **post** $\mathsf{Inequiv}(Q - [s \overset{l}{\leadsto})) \wedge \mathcal{L} = L$ $\}$
corp

We can now turn to $S''_{6.1.1}$:

$\vdots$
$\{$ $\mathsf{Confl_free}([s \overset{l}{\leadsto}]) \wedge \mathsf{Inequiv}(Q - [s \overset{l}{\leadsto}]) \wedge \mathcal{L} = L$ $\}$
$S''_{6.1.1}$
$\{$ $\mathsf{Confl_free}([s \overset{l}{\leadsto})) \wedge \mathsf{Inequiv}(Q - [s \overset{l}{\leadsto}]) \wedge \mathsf{Pairwise_inequiv}(Q - [s \overset{l}{\leadsto}], \{\delta^*(s, l)\})$
$\wedge \mathcal{L} = L$ $\}$
$\vdots$

An implementation of $S''_{6.1.1}$ must check state $\delta^*(s, l)$ for equivalence against states in $Q - [s \overset{l}{\leadsto}]$, eliminating it if an equivalent one is found.

Indeed, it suffices to check $\delta^*(s, l)$ for equivalence against $Q - [s \overset{lr}{\leadsto}]$ (all states *except* $[s \overset{lr}{\leadsto}]$) because $\delta^*(s, l)$ lies on $[s \overset{lr}{\leadsto}]$, and $\mathsf{Inequiv}([s \overset{lr}{\leadsto}])$ thanks to Corollary 2.79. The resulting implementation of visit_min is now (where we introduce local variable $p = \delta^*(s, l)$ and use function eq to check state equivalence):

---

proc $\mathsf{visit_min}($**in** $l, r : \Sigma^*) \rightarrow$
$\{$ **pre** $\mathsf{Confl_free}([s \overset{lr}{\leadsto}]) \wedge \mathsf{Inequiv}(Q - [s \overset{lr}{\leadsto}]) \wedge \delta^*(s, lr) \neq \bot \wedge \mathcal{L} = L$ $\}$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

as $r \neq \varepsilon \rightarrow$ visit_min$(l \cdot \text{head}(r), \text{tail}(r))$
$\{\ \text{Inequiv}(Q - [s \overset{l \cdot \text{head}(r)}{\rightsquigarrow}]) \wedge \mathcal{L} = L\ \}$
sa;
$\{\ \text{Confl_free}([s \overset{l}{\rightsquigarrow}]) \wedge \text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}]) \wedge \mathcal{L} = L\ \}$
$[\![$ **var** $p, q : \text{STATE}$
$|\ p := \delta^*(s, l);$
as $\langle \exists\, q : q \in Q - [s \overset{lr}{\rightsquigarrow}] : \text{eq}(p, q) \rangle \rightarrow$
let $q : q \in Q - [s \overset{lr}{\rightsquigarrow}] \wedge \text{eq}(p, q);$
merge(p, q)
sa
$]\!]$
$\{\ \text{Confl_free}([s \overset{l}{\rightsquigarrow})) \wedge \text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}]) \wedge \text{Pairwise_inequiv}(Q - [s \overset{l}{\rightsquigarrow}], \{\delta^*(s, l)\})$
$\wedge\, \mathcal{L} = L\ \}$
$\{$ **post** $\text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow})) \wedge \mathcal{L} = L\ \}$
corp

For efficiency reasons, we would prefer eq$'$ over eq — see §2.5.1. We start with $\text{Confl_free}([s \overset{l}{\rightsquigarrow}]) \wedge \text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}])$, which holds before the second **as-sa** statement:

$\text{Confl_free}([s \overset{l}{\rightsquigarrow}]) \wedge \text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}])$

$\Rightarrow$ "Property 2.48 and first conjunct"

$\text{Succ}(Q - [s \overset{l}{\rightsquigarrow})) \subseteq Q - [s \overset{l}{\rightsquigarrow}] \wedge \text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}])$

$\Rightarrow$ "second conjunct and set containment in the first conjunct"

$\text{Inequiv}(\text{Succ}(Q - [s \overset{l}{\rightsquigarrow})))$

$\equiv$ "calculus of sets and state paths"

$\text{Inequiv}(\text{Succ}((Q - [s \overset{l}{\rightsquigarrow}]) \cup \{\delta^*(s, l)\}))$

This last line, with Property 2.86, implies that eq$'$ can be used to evaluate the guard in:

---

proc visit_min(**in** $l, r : \Sigma^*$) $\rightarrow$
$\{$ **pre** $\text{Confl_free}([s \overset{lr}{\rightsquigarrow}]) \wedge \text{Inequiv}(Q - [s \overset{lr}{\rightsquigarrow}]) \wedge \delta^*(s, lr) \neq \perp \wedge \mathcal{L} = L\ \}$
as $r \neq \varepsilon \rightarrow$ visit_min$(l \cdot \text{head}(r), \text{tail}(r))$
$\{\ \text{Inequiv}(Q - [s \overset{l \cdot \text{head}(r)}{\rightsquigarrow}]) \wedge \mathcal{L} = L\ \}$
sa;
$\{\ \text{Confl_free}([s \overset{l}{\rightsquigarrow}]) \wedge \text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}]) \wedge \mathcal{L} = L\ \}$
$\{\ \text{Inequiv}(\text{Succ}(Q - [s \overset{l}{\rightsquigarrow}) \cup \{\delta^*(s, l)\}))\ \}$
$[\![$ **var** $p : \text{STATE}$
$|\ p := \delta^*(s, l);$
as $\langle \exists\, q : q \in Q - [s \overset{lr}{\rightsquigarrow}] : \text{eq}'(p, q) \rangle \rightarrow$
let $q : q \in Q - [s \overset{lr}{\rightsquigarrow}] \wedge \text{eq}'(p, q);$
merge(p, q)
sa
$]\!]$
$\{\ \text{Confl_free}([s \overset{l}{\rightsquigarrow})) \wedge \text{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}]) \wedge \text{Pairwise_inequiv}(Q - [s \overset{l}{\rightsquigarrow}], \{\delta^*(s, l)\})$

$\wedge \mathcal{L} = L$ }
{ **post** $\mathsf{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}]) \wedge \mathcal{L} = L$ }
corp

To make this algorithm more practical (by avoiding recomputing $p = \delta^*(s, l)$), we could make p a parameter instead of recomputing it. This gives us a new version of our procedure:

---

proc $\mathsf{visit_min}'$(**in** $p : \text{STATE}$; **in** $l, r : \Sigma^*$) $\rightarrow$
{ **pre** $\mathsf{Confl_free}([s \overset{lr}{\rightsquigarrow}]) \wedge \mathsf{Inequiv}(Q - [s \overset{lr}{\rightsquigarrow}]) \wedge p = \delta^*(s, lr) \wedge p \neq \bot \wedge \mathcal{L} = L$ }
as $r \neq \varepsilon \rightarrow \mathsf{visit_min}'(\delta(p, \mathsf{head}(r)), l \cdot \mathsf{head}(r), \mathsf{tail}(r))$
{ $\mathsf{Inequiv}(Q - [s \overset{l \cdot \mathsf{head}(r)}{\rightsquigarrow}]) \wedge \mathcal{L} = L$ }
sa;
{ $\mathsf{Confl_free}([s \overset{l}{\rightsquigarrow}]) \wedge \mathsf{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}]) \wedge \mathcal{L} = L$ }
{ $\mathsf{Inequiv}(\mathsf{Succ}(Q - [s \overset{l}{\rightsquigarrow}) \cup \{p\}))$ }
as $\langle \exists\, q : q \in Q - [s \overset{lr}{\rightsquigarrow}] : \mathsf{eq}'(p, q) \rangle \rightarrow$
let $q : q \in Q - [s \overset{lr}{\rightsquigarrow}] \wedge \mathsf{eq}'(p, q)$;
$\mathsf{merge}(p, q)$
sa
{ $\mathsf{Confl_free}([s \overset{l}{\rightsquigarrow})) \wedge \mathsf{Inequiv}(Q - [s \overset{l}{\rightsquigarrow}]) \wedge \mathsf{Pairwise_inequiv}(Q - [s \overset{l}{\rightsquigarrow}], \{\delta^*(s, l)\})$
$\wedge \mathcal{L} = L$ }
{ **post** $\mathsf{Inequiv}(Q - [s \overset{l}{\rightsquigarrow})) \wedge \mathcal{L} = L$ }
corp

6.2 Procedure $\mathsf{cleanup}_I$

Since $\mathsf{add_word}_I$ maintains minimality, we trivially have the following procedure

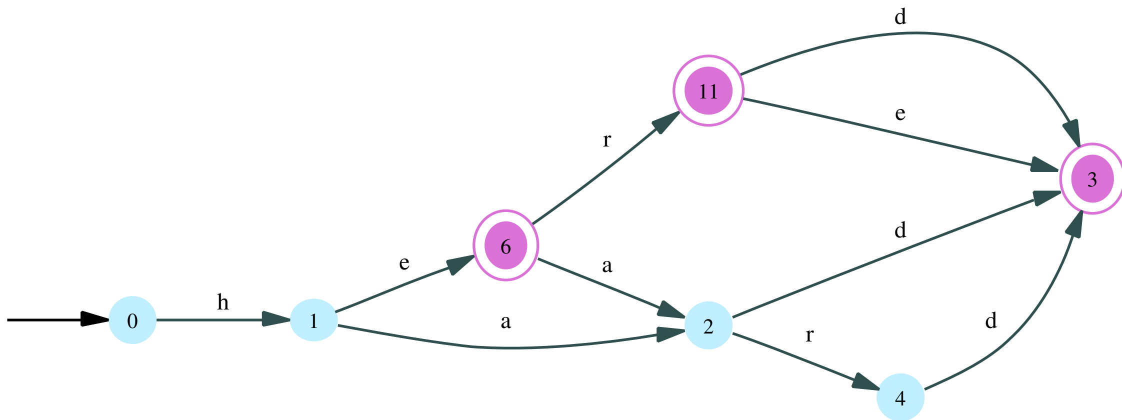
---

proc $\mathsf{cleanup}_I() \rightarrow$
{ **pre** $\mathsf{Min} \wedge \mathcal{L} = L$ }
skip
{ **post** $\mathsf{Min} \wedge \mathcal{L} = L$ }
corp

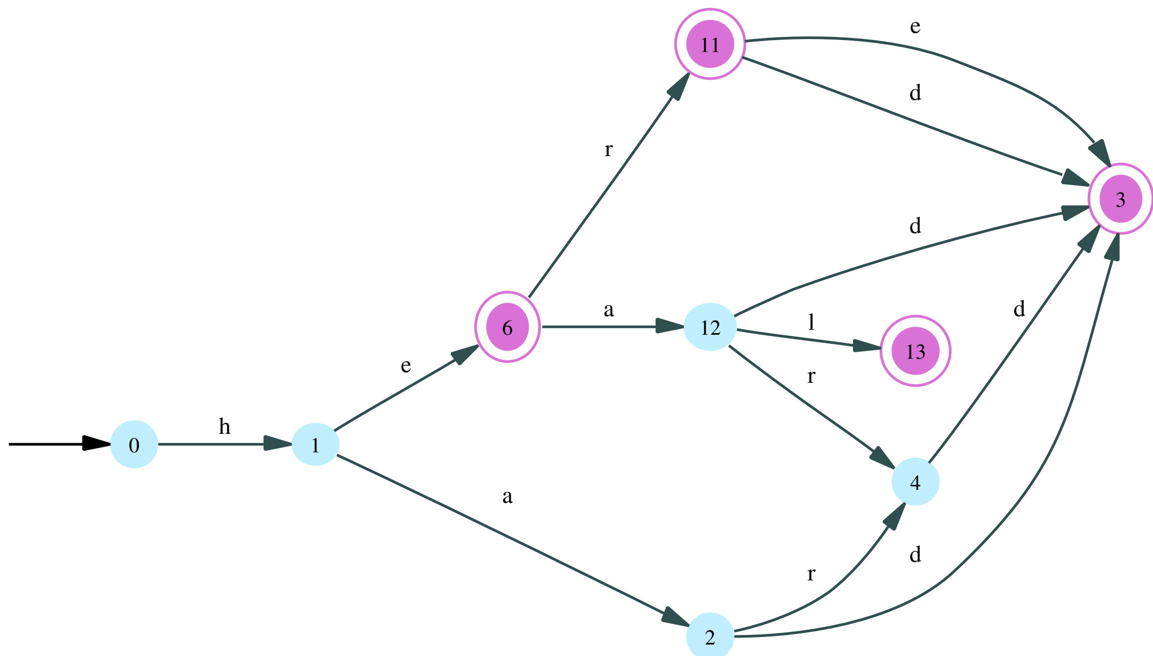
6.3 An example

Starting with the MADFA from §4.3 starting on page 38

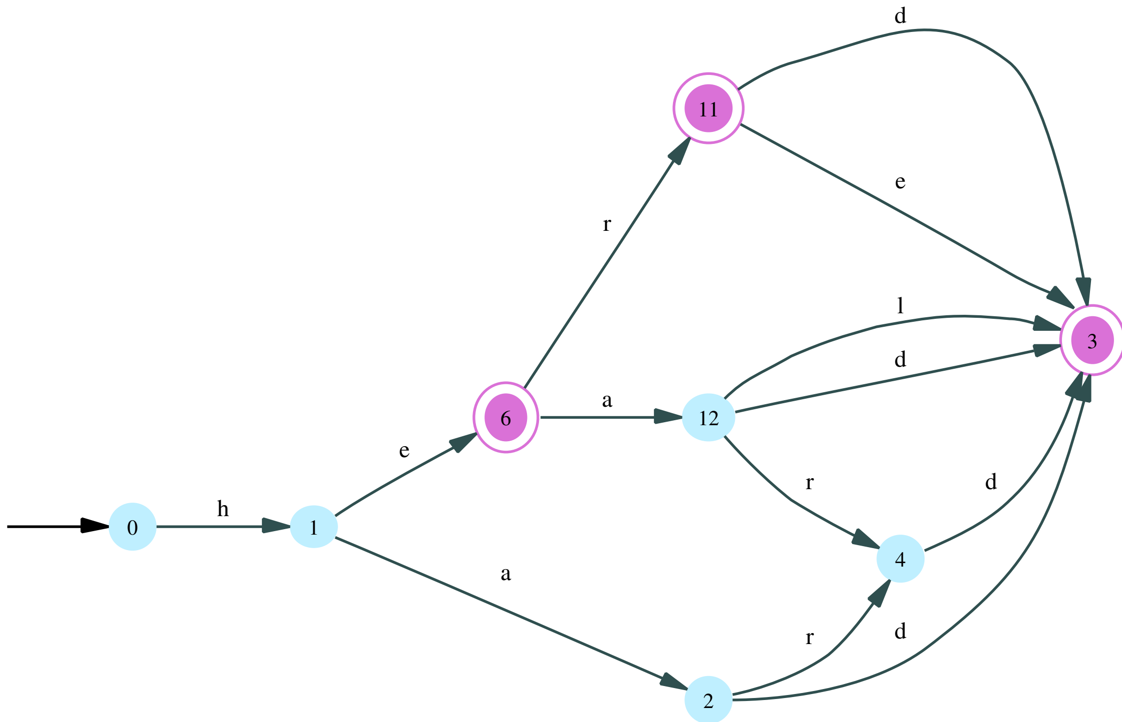
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



we add word `heal` using $\mathsf{add_word}_N{}'$, giving (where state 12 is a clone of 2 and 13 is new)

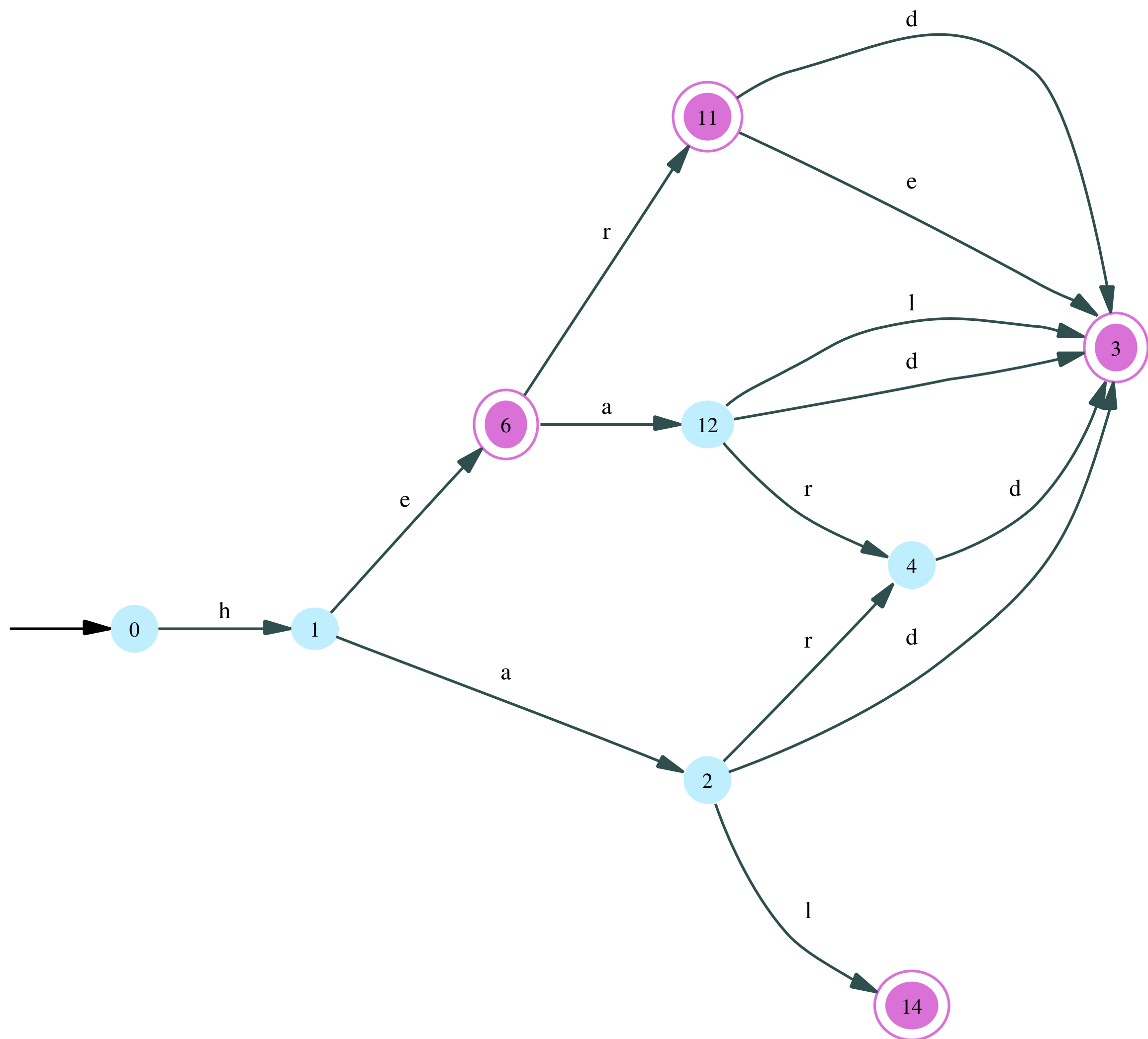


We then move on to visit_min(ε, `heal`), which considers $[s \overset{\mathtt{heal}}{\rightsquigarrow}] = [0, 1, 6, 12, 13]$ bottom up, we see that state 13 can be merged into state 3, giving



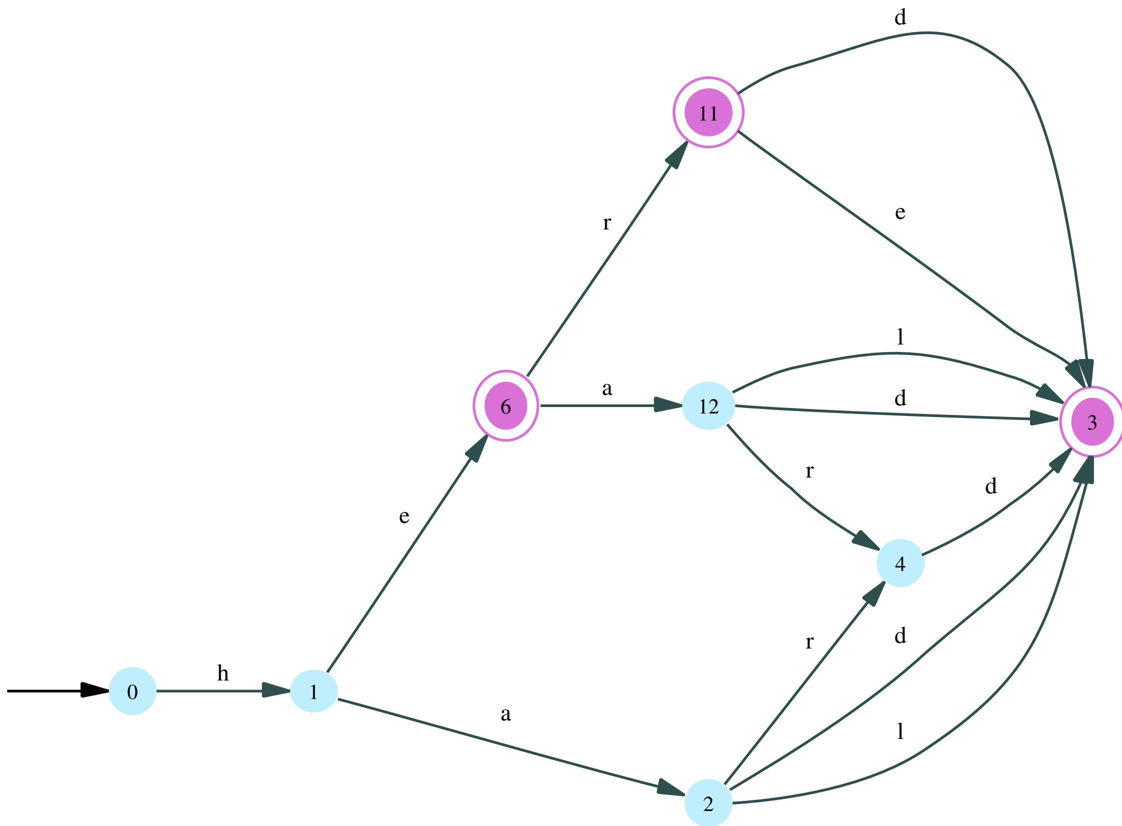
This ADFA is minimal since nothing further is minimized on $[s \overset{\mathtt{hea}}{\rightsquigarrow}] = [0, 1, 6, 12]$. Consider now adding word `hal` using $\mathsf{add_word_N}'$, giving

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



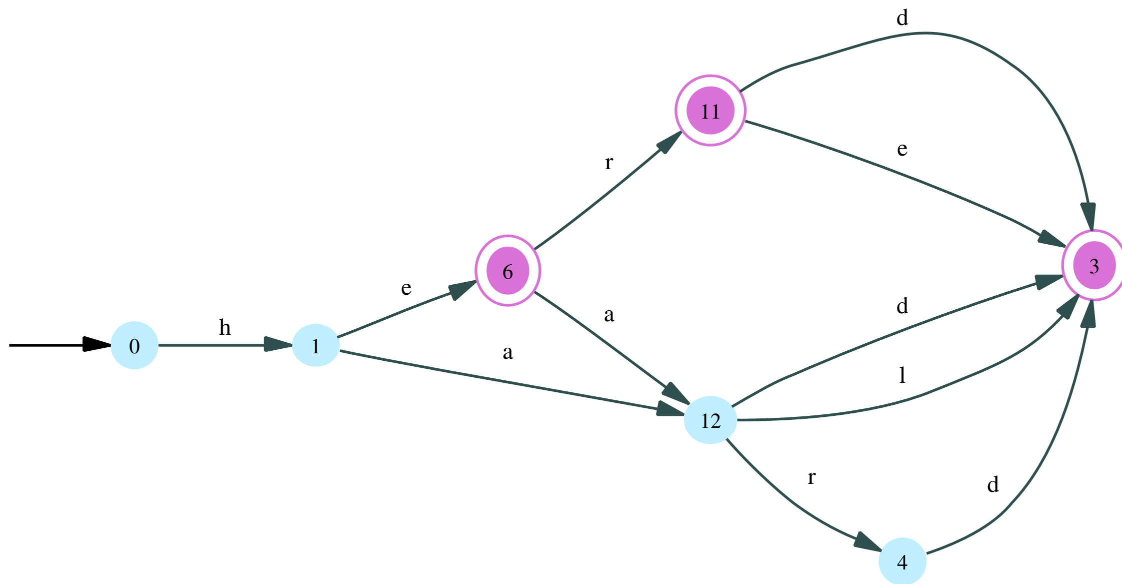
In the invocation of visit_min′ we consider $[s \overset{\texttt{hal}}{\rightsquigarrow}] = [0, 1, 2, 14]$, state 14 is merged with state 3, giving

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



States 2 and 12 are then found to be equivalent, giving

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



This last automaton is minimal.

6.4 Time and space performance

From Chapter 5, procedure $\mathsf{add_word}_N{}'$ takes $\mathcal{O}(|w|)$ time and space when adding word w. Using a clever coding of eq', an invocation $\mathsf{visit_min}'(s, \varepsilon, w)$ also takes $\mathcal{O}(|w|)$ time and space — see [WD03]. It follows that $\mathsf{add_word}_I$ can also be implemented in $\mathcal{O}(|w|)$ time and space.

There are three relatively easy improvements:

1. Given Property 2.70, we can use invocation $\mathsf{visit_min}'(\delta(s, \mathsf{head}(w)), \mathsf{head}(w), \mathsf{tail}(w))$ when $w \neq \varepsilon$.

2. While adding w with $\mathsf{add_word}_N{}'$, we already traverse path $[s \overset{w}{\rightsquigarrow}]$ and need not compute it anew within $\mathsf{visit_min}'$.

3. Another minor improvement can be made in the invocation of $\mathsf{add_word}_N{}'$: we need not create new states if they will subsequently be merged by $\mathsf{visit_min}'$. This approach, which is used in practice, requires some additional book-keeping, and has been presented in [Wat03b, Wat01d].

6.5 Commentary

Procedure $\mathsf{add_word}_I$ is essentially the same (modulo presentation and derivation style) as in [PAMS94, SFK95, Dac98, Mih99a, Mih99b, CD99] — though the greatest similarity is with those given in [DWW98, DMWW00]. Another version of this algorithm was presented in [Wat03b].

Chapter 7

Reversed trie intermediate ADFA

In this chapter, we maintain M as a trie corresponding to the reverse of the words added so far. Formally, the invariant is

$$\mathsf{Struct}_R(D) \equiv \mathsf{Is_trie} \wedge \mathcal{L} = D^R$$

The resulting ADFA accepts W^R. Minimality of M is achieved by reversing M (usually yielding a nondeterministic automaton) and determinizing it. The reversal and determinization steps are combined in this chapter into $\mathsf{cleanup}_R$. We will only present the procedures and examples — a full derivation of this algorithm can be found in [Wat01e, Wat02a] and most recently in [Wat02b], where an alternative derivation is given.

7.1 Procedure $\mathsf{add_word}_R$

As our specification, we get:

---

proc $\mathsf{add_word}_R(\textbf{in}\ w : \Sigma^*) \rightarrow$
  { **pre** $\mathsf{Is_trie} \wedge \mathcal{L} = L$ }
  $S_{7.1}$
  { **post** $\mathsf{Is_trie} \wedge \mathcal{L} = L \cup \{w^R\}$ }
corp

Given the specification of $\mathsf{add_word}_T$ (in Chapter 4), if we assume that argument w can be reversed as a primitive operation (it can be done in $\mathcal{O}(|w|)$ time and constant space), we implement $\mathsf{add_word}_R$ as

---

proc $\mathsf{add_word}_R(\textbf{in}\ w : \Sigma^*) \rightarrow$
  { **pre** $\mathsf{Is_trie} \wedge \mathcal{L} = L$ }
  $\mathsf{add_word}_T(w^R)$
  { **post** $\mathsf{Is_trie} \wedge \mathcal{L} = L \cup \{w^R\}$ }
corp

Naturally, it would also be easy to specialize $\mathsf{add_word}_T$ explicitly by expanding its body.

7.2 Procedure $\mathsf{cleanup}_R$

We now require a minimization procedure with specification:

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

---

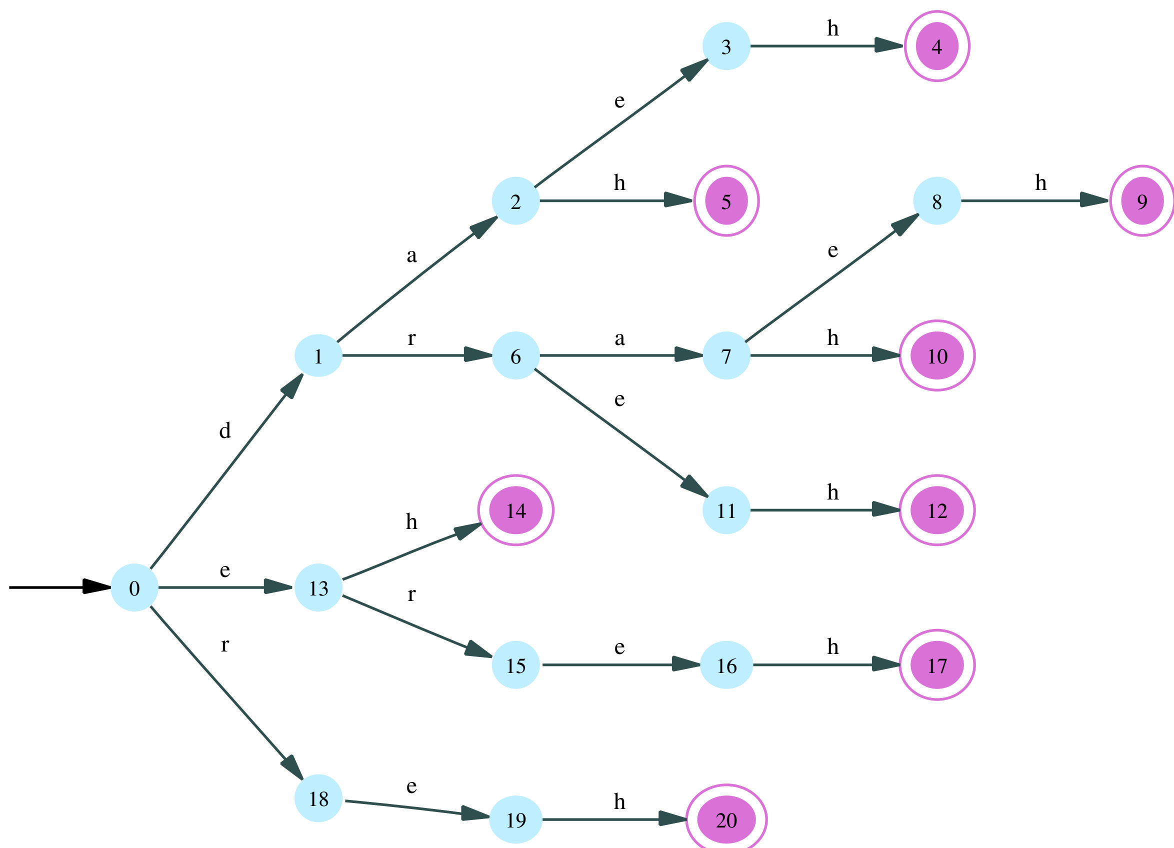
proc $\mathsf{cleanup_R}() \rightarrow$
  { **pre** $\mathsf{Is_trie} \wedge \mathcal{L} = L$ }
  $Q, \delta, s, F : S_{7.2}$
  { **post** $\mathsf{Min} \wedge \mathcal{L} = L^R$ }
corp

Without discussing the details (which are given originally in [Brz62a] and again in alternative forms in [Wat95, Wat00b, Wat02a], with surprisingly concise derivations), $\mathsf{cleanup_R}$ can be implemented by reversing M and determinizing the result using the automata determinization (also known as the 'subset construction') algorithm; see [HU79] for a detailed treatment of automata determinization.

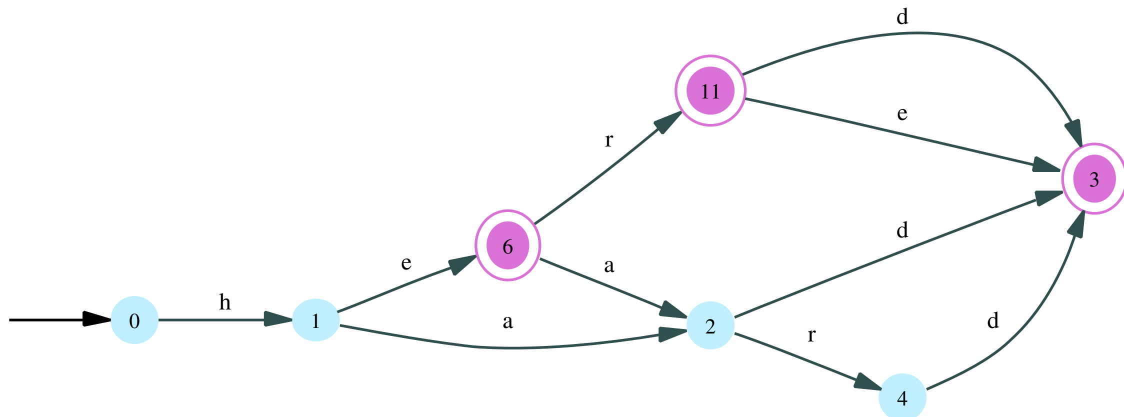
Implementation 7.1 *The efficiency of this algorithm hinges on a good encoding of the* ADFA *which is reversible — usually by storing the reversed transitions in addition to the forward transitions — and an efficient implementation of the determinization algorithm. Aspects of efficient determinization implementations are discussed in [JW96].*

7.3 An example

The reverse trie corresponding to `he`, `her`, `had`, `head`, `hard`, `herd`, `here`, `heard` is



The MADFA resulting from applying $\mathsf{cleanup_R}$ to the above reverse trie is



This ADFA is minimal.

7.4 Time and space performance

As a simple variant of procedure add_word$_T$ (in Chapter 4), procedure add_word$_R$ has the same running time and space, also yielding a trie of size $\mathcal{O}(\sum_{w \in W} |w|)$. I conjecture that procedure cleanup$_R$ takes time and space $\mathcal{O}(|M|)$. In particular, I conjecture the determinization step to be linear; the reversal step is easily done in linear time. Assuming this conjecture, the construction and minimization takes $\mathcal{O}(\sum_{w \in W} |w|)$ space and time.

7.5 Commentary

This algorithm is a specialization (to acyclic DFAs) of Brzozowski's DFA minimization algorithm [Brz62a, Brz62b]. More recently, it is described in [Wat00a, Wat02a]. Brzozowski's minimization algorithm has an interesting history, described in [Wat00b, Wat01a].

Chapter 8

Avoiding cloning while adding words

Performance profiling of an implementation of the algorithm presented in Chapter 6, shows that most of the execution time is spent on two operations:

1. Cloning confluence states.
2. Merging states found to be equivalent.

(Creating new states is a cheap operation in practice.) While the *merging* operation is generally unavoidable in constructing a MADFA, in the subsequent chapters, we focus on a performance improvement by eliminating cloning. Cloning is only applied to confluence states. In those chapters, the structural invariants and the word-orders will be chosen in such a way that the preconditions of the add_word variants are strengthenings of $\mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}])$ when adding word w. As a result, the body of each add_word variant will use $\mathsf{add_word}_{\mathsf{T}}$ (from Chapter 4), followed by further operations to restore the appropriate structural invariant and prepare for the next word to be added.

Chapter 9

Words in lexicographic order

In this chapter, we avoid the cloning operation by adding the words in lexicographic order so that part of the automaton will never be visited again while adding a word later. After those states are last visited, we consider them for equivalence with other states and merge them where possible — thereby enlarging the 'minimized' portion of the automaton. (Recall from Property 2.25 on page 11 that the merge operation usually creates confluence states.)

The only part of the automaton which will be 'unminimized' and is guaranteed not to have any confluences is the path of the lexicographically greatest word accepted by the automaton (the last word added in our ordering). The structural invariant is therefore

$$\mathsf{Struct_S}(D) \equiv \mathsf{Inequiv}(\underbrace{Q - [s \overset{\mathsf{lexmax}}{\rightsquigarrow}]}_{\text{won't visit again}}) \land \mathsf{Confl_free}(\underbrace{[s \overset{\mathsf{lexmax}}{\rightsquigarrow}]}_{\text{may partly visit}}) \land \mathcal{L} = D$$

The lexicographic order of word-adding yields the following precondition when adding word w

$$\mathsf{lexmax} \sqsubset_{\iota} w$$

Thanks to the definition of $\sqsubset_{\iota}$, we also have

$$\mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}])$$

in the pre- and postcondition. We do not add it there explicitly because w consists of two parts:

- A prefix shared with lexmax, namely $\mathsf{lexmax} \overset{\triangle}{_{\mathrm{p}}} w$. We already know that the path corresponding to this prefix is confluence-free because $\mathsf{Confl_free}([s \overset{\mathsf{lexmax}}{\rightsquigarrow}])$ holds from $\mathsf{Struct_S}$.
- The corresponding suffix w, namely $(\mathsf{lexmax} \overset{\triangle}{_{\mathrm{p}}} w)^{-1} w$. The path of this suffix is not yet in the automaton and corresponds to states still to be added.

We return to the structure of w in the next section.

9.1 Procedure $\mathsf{add_word_S}$

Our starting point is

---

proc $\mathsf{add_word_S}(\textbf{in}\ w : \Sigma^*) \rightarrow$
{ **pre** $\mathsf{lexmax} \sqsubset_{\iota} w$

$\wedge\ \mathsf{Inequiv}(Q - [s \overset{\mathsf{lexmax}}{\rightsquigarrow}])$
$\wedge\ \mathsf{Confl_free}([s \overset{\mathsf{lexmax}}{\rightsquigarrow}])$
$\wedge\ \mathcal{L} = \mathsf{L}\ \}$

$S_{9.1}$

$\{$ **post** $\mathsf{lexmax} = w$
$\wedge\ \mathsf{Inequiv}(Q - [s \overset{\mathsf{lexmax}}{\rightsquigarrow}])$
$\wedge\ \mathsf{Confl_free}([s \overset{\mathsf{lexmax}}{\rightsquigarrow}])$
$\wedge\ \mathcal{L} = \mathsf{L} \cup \{w\}\ \}$

corp

We will use $\mathsf{add_word}_T$ to add w, since we are guaranteed not to encounter any confluences on $[s \overset{w}{\rightsquigarrow}]$. Adding w will traverse the longest common prefix of lexmax and w, namely $\mathsf{lexmax} \overset{\triangle}{_p} w$, before adding new states for the remainder of w: $(\mathsf{lexmax} \overset{\triangle}{_p} w)^{-1} w$, as mentioned earlier. We introduce shadow variable z to capture the previous lexmax before the invocation of $\mathsf{add_word}_T$. After w has been added, we have

$$\mathsf{Confl_free}([s \overset{z}{\rightsquigarrow}] \cup [s \overset{w}{\rightsquigarrow}])$$

while the remainder of the automaton is minimized:

$$\mathsf{Inequiv}(Q - ([s \overset{z}{\rightsquigarrow}] \cup [s \overset{w}{\rightsquigarrow}]))$$

Our refined procedure is:

---

proc $\mathsf{add_word}_S$(**in** $w : \Sigma^*$) $\rightarrow$

$\{$ **pre** $\mathsf{lexmax} \sqsubset_l w \wedge z = \mathsf{lexmax}$
$\wedge\ \mathsf{Inequiv}(Q - [s \overset{\mathsf{lexmax}}{\rightsquigarrow}])$
$\wedge\ \mathsf{Confl_free}([s \overset{\mathsf{lexmax}}{\rightsquigarrow}])$
$\wedge\ \mathcal{L} = \mathsf{L}\ \}$

$\mathsf{add_word}_T(w)$;

$\{\ \mathsf{lexmax} = w$
$\wedge\ \mathsf{Inequiv}(Q - ([s \overset{z}{\rightsquigarrow}] \cup [s \overset{w}{\rightsquigarrow}]))$
$\wedge\ \mathsf{Confl_free}([s \overset{z}{\rightsquigarrow}] \cup [s \overset{w}{\rightsquigarrow}])$
$\wedge\ \mathcal{L} = \mathsf{L} \cup \{w\}\ \}$

$S'_{9.1}$

$\{$ **post** $\mathsf{lexmax} = w$
$\wedge\ \mathsf{Inequiv}(Q - [s \overset{\mathsf{lexmax}}{\rightsquigarrow}])$
$\wedge\ \mathsf{Confl_free}([s \overset{\mathsf{lexmax}}{\rightsquigarrow}])$
$\wedge\ \mathcal{L} = \mathsf{L} \cup \{w\}\ \}$

corp

Considering the specification (pre- and postcondition) of $S'_{9.1}$, we will have to minimize states $[s \overset{z}{\rightsquigarrow}] - [s \overset{w}{\rightsquigarrow}]$ — that is, all of the states on the z-path *after* the split from the w-path. In this case, 'minimizing' means comparing the states for equivalence against the already inequivalent states $Q - ([s \overset{z}{\rightsquigarrow}] \cup [s \overset{w}{\rightsquigarrow}])$.

Since the common path (which may be visited again) is $[s \overset{z \overset{\triangle}{_p} w}{\rightsquigarrow}]$, the states $[s \overset{z}{\rightsquigarrow}] - [s \overset{w}{\rightsquigarrow}]$ to be minimized can also be written

$$(\delta^*(s, z \overset{\triangle}{_p} w) \overset{(z \overset{\triangle}{_p} w)^{-1} z}{\rightsquigarrow}]$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

(Note the open/noninclusive beginning of this path.) If this path is empty (that is, $(z \overset{\triangle}{_P} w)^{-1}z = \varepsilon$, which occurs when z is a prefix of w), no states need to be minimized. Otherwise, the minimization step can be accomplished using procedure visit_min from §6.1.1, beginning on page 51. Recall that visit_min takes two arguments l, r and minimizes $[\delta^*(s, l) \overset{r}{\leadsto}]$. Note the closed/inclusive beginning of this path, so that the naïve invocation

$$\text{visit_min}(z \overset{\triangle}{_P} w, (z \overset{\triangle}{_P} w)^{-1}z)$$

would therefore accidentally also minimize state $\delta^*(s, z \overset{\triangle}{_P} w)$. What is needed instead is

$$\text{visit_min}(z \overset{\triangle}{_P} w \cdot \text{head}((z \overset{\triangle}{_P} w)^{-1}z), \text{tail}((z \overset{\triangle}{_P} w)^{-1}z))$$

The resulting algorithm (with correct invocation of visit_min and in which z is now a program variable[1] capturing the previous value of lexmax and u, v are two variables to aid in readability):

---

proc add_word$_S$(**in** $w : \Sigma^*$) $\rightarrow$
  { **pre** lexmax $\sqsubset_l w$
    $\wedge$ Inequiv$(Q - [s \overset{\text{lexmax}}{\leadsto}])$
    $\wedge$ Confl_free$([s \overset{\text{lexmax}}{\leadsto}])$
    $\wedge \mathcal{L} = L$ }
  ⟦ **var** $u, v, z : \Sigma^*$
  | $z :=$ lexmax;
   add_word$_T(w)$;
   { lexmax $= w$
    $\wedge$ Inequiv$(Q - ([s \overset{z}{\leadsto}] \cup [s \overset{w}{\leadsto}]))$
    $\wedge$ Confl_free$([s \overset{z}{\leadsto}] \cup [s \overset{w}{\leadsto}])$
    $\wedge \mathcal{L} = L \cup \{w\}$ }
   $u := z \overset{\triangle}{_P} w$;
   $v := u^{-1}z$;
   { $z = u \cdot v$ }
   as $v \neq \varepsilon \rightarrow$ visit_min$(u \cdot \text{head}(v), \text{tail}(v))$ **sa**
  ⟧
  { **post** lexmax $= w$
    $\wedge$ Inequiv$(Q - [s \overset{\text{lexmax}}{\leadsto}])$
    $\wedge$ Confl_free$([s \overset{\text{lexmax}}{\leadsto}])$
    $\wedge \mathcal{L} = L \cup \{w\}$ }
corp

Implementation 9.1 *The test in the* ***as-sa*** *statement can be simplified as follows:*

$v \neq \varepsilon$

$\equiv$ "*$v = u^{-1}z$ and $u = z \overset{\triangle}{_P} w$*"

$(z \overset{\triangle}{_P} w)^{-1}z \neq \varepsilon$

$\equiv$ "*definition of derivatives*"

$(z \overset{\triangle}{_P} w) \neq z$

[1] As opposed to a shadow variable used for expressing pre- and postconditions.

$\equiv$ *"definition of $\overset{\triangle}{\scriptstyle p}$"*

z is not a prefix of w

$\equiv$ *"definition of word-paths"*

$\delta^*(s, z) \notin [s \overset{w}{\rightsquigarrow}]$

This last form is easy to test: when it holds, $\mathsf{add_word}_T$ *will not have passed through state $\delta^*(s, z)$ while adding word w, and* $\mathsf{add_word}_T$ *can be modified to note whether or not state $\delta^*(s, z)$ was visited.*

9.1.1 A minor problem in using visit_min

In $\mathsf{add_word}_S$, we have taken a shortcut in our invocation

$$\mathsf{visit_min}(u \cdot \mathsf{head}(v), \mathsf{tail}(v))$$

Before the invocation, we have

$$\mathsf{Inequiv}(Q - ([s \overset{z}{\rightsquigarrow}] \cup [s \overset{w}{\rightsquigarrow}]))$$

By contrast, we can instantiate visit_min's precondition (taken from §6.1.1 on page 51) conjunct using our invocation above:

$\mathsf{Inequiv}(Q - [s \overset{lr}{\rightsquigarrow}])$

$\equiv$ "in our invocation of visit_min: $l = u \cdot \mathsf{head}(v)$ and $r = \mathsf{tail}(v)$"

$\mathsf{Inequiv}(Q - [s \overset{u \cdot \mathsf{head}(v) \cdot \mathsf{tail}(v)}{\rightsquigarrow}])$

$\equiv$ "definition of head and tail"

$\mathsf{Inequiv}(Q - [s \overset{u \cdot v}{\rightsquigarrow}])$

$\equiv$ "substituting variables' values $u = z \overset{\triangle}{\scriptstyle p} w$ and $v = u^{-1}z$"

$\mathsf{Inequiv}(Q - [s \overset{(z \overset{\triangle}{p} w) \cdot ((z \overset{\triangle}{p} w)^{-1} z)}{\rightsquigarrow}])$

$\equiv$ "string calculus, definition of common prefix and derivatives"

$\mathsf{Inequiv}(Q - [s \overset{z}{\rightsquigarrow}])$

Procedure visit_min therefore expects

$$\mathsf{Inequiv}(Q - [s \overset{z}{\rightsquigarrow}])$$

whereas we are only guaranteed the weaker

$$\mathsf{Inequiv}(Q - ([s \overset{z}{\rightsquigarrow}] \cup [s \overset{w}{\rightsquigarrow}]))$$

before it is invoked in $\mathsf{add_word}_S$. Indeed, at the point of invocation, we have $\mathsf{Confl_free}([s \overset{w}{\rightsquigarrow}])$ — those states are likely *not* inequivalent to states in $Q - [s \overset{z}{\rightsquigarrow}]$. In the body of visit_min, this discrepancy affects in the second **as-sa** (taken from page 53):

$\vdots$

as $\langle \exists\, q : q \in Q - [s \overset{lr}{\rightsquigarrow}] : \mathsf{eq}'(p, q) \rangle \rightarrow$

  let $q : q \in Q - [s \overset{lr}{\rightsquigarrow}] \wedge \mathsf{eq}'(p, q)$;

  $\mathsf{merge}(p, q)$

sa

$\vdots$

The range of the quantification (and the **let** statement) should be narrowed to exclude $[s \overset{w}{\leadsto}]$. This is most easily done by adding a parameter to visit_min and passing in w while specializing that procedure's body. We do not do that here.

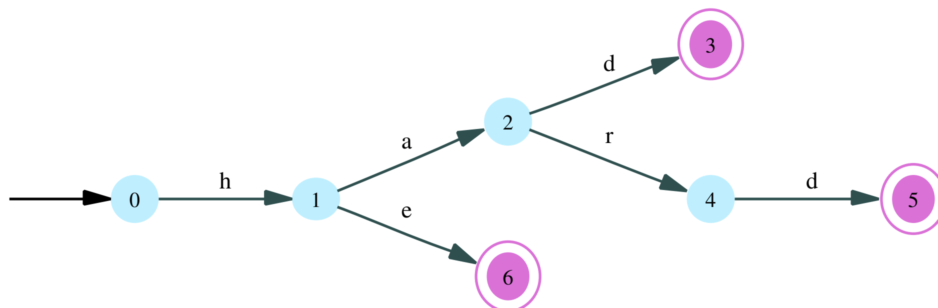
9.2 Procedure $\mathsf{cleanup}_S$

After the last word has been added, a final minimization step is required to deal with states $[s \overset{\mathsf{lexmax}}{\leadsto}]$. This is trivially done using visit_min′:

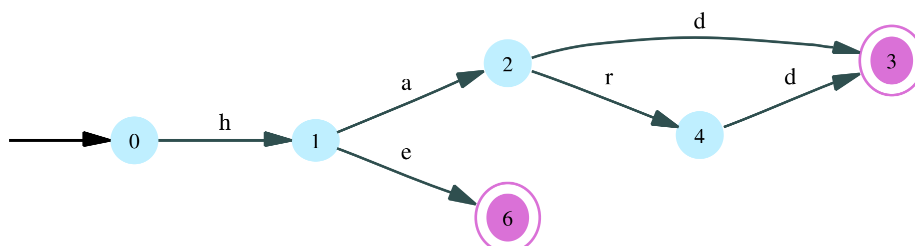
$$
\begin{array}{l}
\textbf{proc}\ \mathsf{cleanup}_S() \rightarrow \\
\quad \{\ \textbf{pre}\ \mathsf{Inequiv}(Q - [s \overset{\mathsf{lexmax}}{\leadsto}]) \\
\qquad \wedge\ \mathsf{Confl_free}([s \overset{\mathsf{lexmax}}{\leadsto}]) \\
\qquad \wedge\ \mathcal{L} = L\ \} \\
\quad \mathsf{visit_min}(\varepsilon, \mathsf{lexmax}) \\
\quad \{\ \textbf{post}\ \mathsf{Min} \wedge \mathcal{L} = L\ \} \\
\textbf{corp}
\end{array}
$$

9.3 An example

Consider adding the words (using $\mathsf{add_word}_S$) in the lexicographic order `had`, `hard`, `he`, `head`, `heard`, `her`, `herd`, `here`. After adding `he` using $\mathsf{add_word}_T$ (invoked from $\mathsf{add_word}_S$), but before minimizing, we have

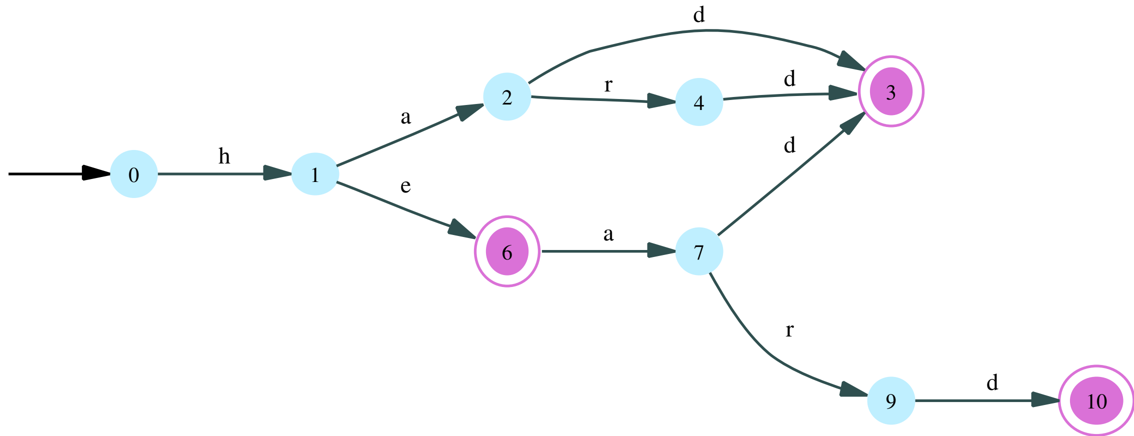


The minimization step considers the $(\underbrace{\mathtt{hard}}_{\mathsf{lexmax}} \overset{\triangle}{{}_{p}} \mathtt{he})^{-1}\mathtt{hard} = \mathtt{ard}$ path from state 1 (state path $(1, 2, 4, 5])$ and merges state 5 into 3, yielding

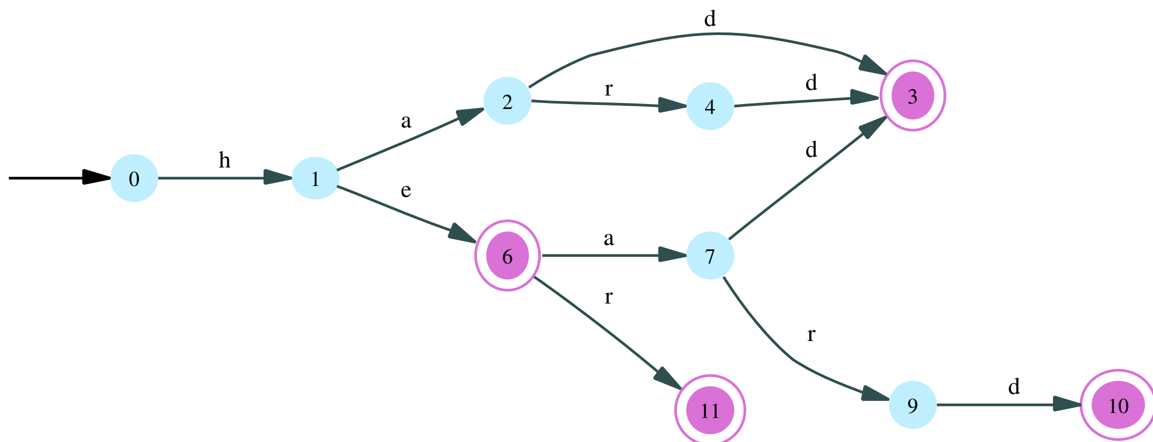


Adding `head` (an extension of `he`) and then `heard` and minimizing, we have

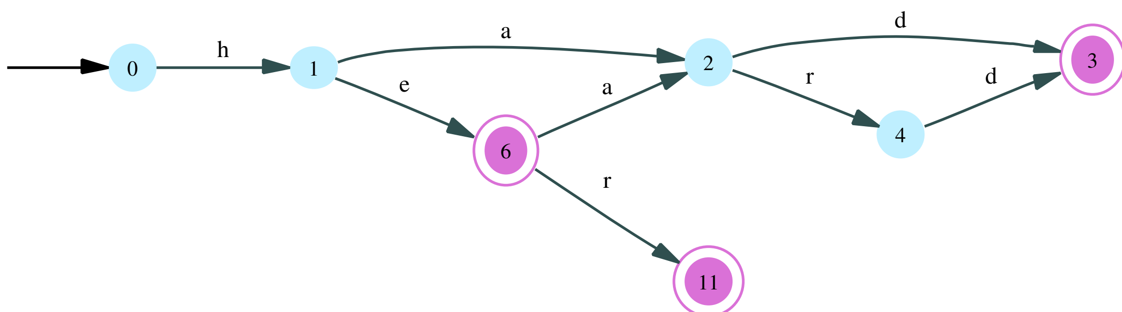
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



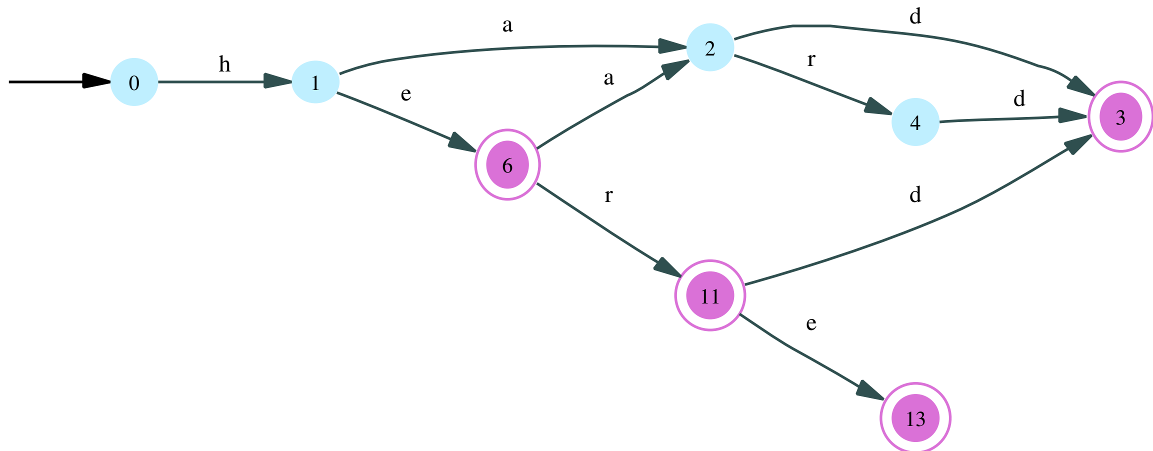
After adding `her`, but before minimizing, we have



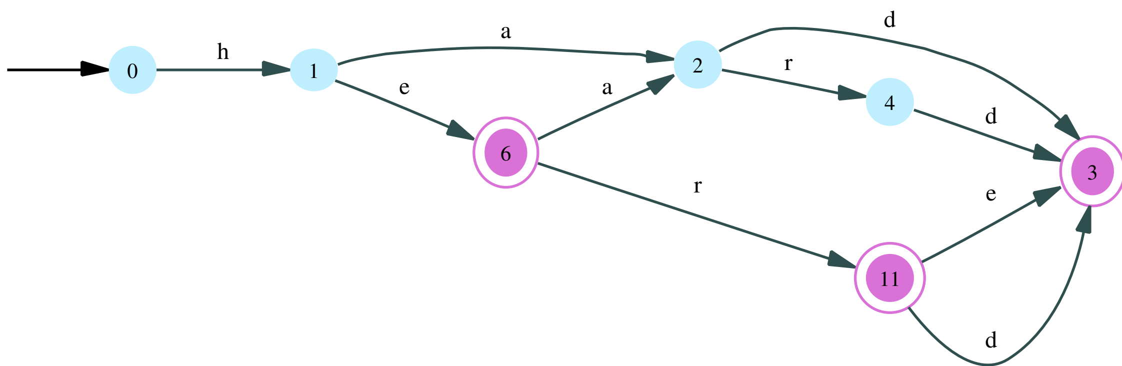
Consider $(\mathtt{heard} \overset{\triangle}{_{p}} \mathtt{her})^{-1}\mathtt{heard} = \mathtt{ard}$ from state 6 (state path $(6, 7, 9, 10])$ shows that state 10 can be merged into state 3; subsequently state 9 can be merged into state 4, and finally state 7 can be merged into 2, giving



Adding the last two words `herd` and `here` yields



The cleanup_S step considers $[0, 1, 6, 11, 13]$ and only serves to merge states 3 and 13, giving the MADFA



9.4 Time and space performance

Sorting the words requires up to $\mathcal{O}(|W| \log |W|)$ space and time; this is typically not taken into account in the MADFA construction time as W can be kept sorted in real-life applications. From Chapter 4, procedure add_word_T is time and space linear in the length of the word being added. As noted in Chapter 6, visit_min can be implemented to take linear time and space. It follows that add_word_S and cleanup_S are linear in the total lengths of the words.

9.4.1 Improvements

Numerous improvements are possible, most of which are noted in the literature relating to visit_min. They include the following:

1. Construct the new transitions and states (for $(z \overset{\triangle}{_p} w)^{-1} z$) lazily, since some of them may subsequently be merged in visit_min.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

2. [DMWW00] gives a way of finding state $\delta^*(s, z \overset{\triangle}{_p} w)$ without precomputing $z \overset{\triangle}{_p} w$.

9.5 Commentary

This algorithm was simultaneously derived by Daciuk and Mihov in their respective Ph.D. dissertations [Dac98] and [Mih99b]. Another presentation of the algorithm is given in [DMWW00].

Chapter 10

Words by decreasing length: minimizing depth layers

In this chapter, we derive a simple semi-incremental algorithm which, like the one in Chapter 9, depends upon an ordering of the words to avoid the relatively expensive cloning operation. The words are added in *any* order of decreasing length. No confluence states are encountered while adding a word w, and states below depth[1] $|w|$ are maintained pairwise inequivalent, while those at or above depth $|w|$ are not necessarily confluence states. As in Chapter 9, this will enable us to use $\mathsf{add_word}_\mathsf{T}$. Our first structural invariant is

$$\mathsf{Struct}_\mathsf{D}(D) \equiv \mathsf{Inequiv}(\underbrace{DL_{>\mathsf{minlen}}}_{\text{won't visit again}}) \wedge \mathsf{Confl_free}(\underbrace{DL_{\leqslant\mathsf{minlen}}}_{\text{may visit again}}) \wedge \mathcal{L} = D$$

10.1 Procedure $\mathsf{add_word}_\mathsf{D}$

Our starting point is

$$
\begin{array}{l}
\textbf{proc}\ \mathsf{add_word}_\mathsf{D}(\textbf{in}\ w : \Sigma^*) \rightarrow \\
\quad \{\ \textbf{pre}\ |w| \leqslant \mathsf{minlen} \\
\qquad \wedge\ \mathsf{Inequiv}(DL_{>\mathsf{minlen}}) \\
\qquad \wedge\ \mathsf{Confl_free}(DL_{\leqslant\mathsf{minlen}}) \\
\qquad \wedge\ \mathcal{L} = L\ \} \\
\quad S_{10.1} \\
\quad \{\ \textbf{post}\ |w| = \mathsf{minlen} \\
\qquad \wedge\ \mathsf{Inequiv}(DL_{>\mathsf{minlen}}) \\
\qquad \wedge\ \mathsf{Confl_free}(DL_{\leqslant\mathsf{minlen}}) \\
\qquad \wedge\ \mathcal{L} = L \cup \{w\}\ \} \\
\textbf{corp}
\end{array}
$$

Clearly, in $S_{10.1}$ we can use $\mathsf{add_word}_\mathsf{T}$. (Here we also introduce shadow variable k to capture minlen.)

$$
\begin{array}{l}
\textbf{proc}\ \mathsf{add_word}_\mathsf{D}(\textbf{in}\ w : \Sigma^*) \rightarrow \\
\quad \{\ \textbf{pre}\ |w| \leqslant \mathsf{minlen} \wedge k = \mathsf{minlen} \\
\qquad \wedge\ \mathsf{Inequiv}(DL_{>\mathsf{minlen}}) \\
\qquad \wedge\ \mathsf{Confl_free}(DL_{\leqslant\mathsf{minlen}})
\end{array}
$$

[1] *Not* height; recall from Definition 2.57 that depth is a state's *minimum* path-distance from s.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
                ∧ 𝓛 = L }
          { Confl_free([s ⇝w]) }
          add_word_T(w);
          { |w| = minlen
             ∧ Inequiv(DL_{>k})
             ∧ Confl_free(DL_{⩽k})
             ∧ 𝓛 = L ∪ {w} }
          S'_{10.1}
          { post |w| = minlen
                 ∧ Inequiv(DL_{>minlen})
                 ∧ Confl_free(DL_{⩽minlen})
                 ∧ 𝓛 = L ∪ {w} }
    corp
```

Given that the word-lengths are monotonically decreasing, while adding w we are assured that no states deeper than $|w|$ will be visited during future word-adding operations. After adding w, we can minimize all states in $DL_{>|w|}$; states $DL_{>k}$ have already been done, so we need only consider the difference $DL_{>|w|} - DL_{>k} = DL_{(|w|,k]}$. This gives us the following procedure, which implements $S'_{10.1}$ straightforwardly:

```
    proc depths_min(in i, j : ℕ) →
          { post Inequiv(DL_{>j})
                 ∧ 𝓛 = L }
          ⟦ var p, q : STATE
          | for p : p ∈ DL_{(i,j]} →
                as ⟨∃ q : q ∈ DL_{>j} : eq(p, q)⟩ →
                   let q : q ∈ DL_{>j} ∧ eq(p, q);
                   merge(p, q)
                sa
            rof
          ⟧
          { post Inequiv(DL_{>min(i,j)})
                 ∧ 𝓛 = L }
    corp
```

Our final procedure is:

```
    proc add_word_D(in w : Σ*) →
          { pre |w| ⩽ minlen ∧ k = minlen
                ∧ Inequiv(DL_{>minlen})
                ∧ Confl_free(DL_{⩽minlen})
                ∧ 𝓛 = L }
          { Confl_free([s ⇝w]) }
          add_word_T(w);
          { |w| = minlen
             ∧ Inequiv(DL_{>k})
             ∧ Confl_free(DL_{⩽k})
             ∧ 𝓛 = L ∪ {w} }
          depths_min(|w|, k)
          { post |w| = minlen
```

$\wedge\ \mathsf{Inequiv}(\mathrm{DL}_{>\mathsf{minlen}})$
$\wedge\ \mathsf{Confl_free}(\mathrm{DL}_{\leqslant\mathsf{minlen}})$
$\wedge\ \mathcal{L} = \mathrm{L} \cup \{w\}\ \}$
corp

10.2 Procedure $\mathsf{cleanup}_\mathrm{D}$

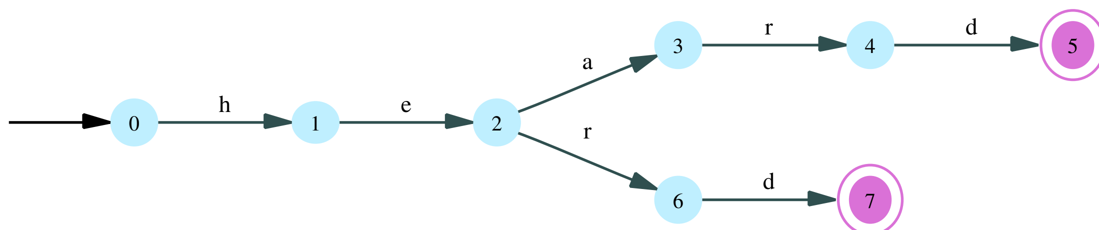
Once the last word has been added, a final minimization step deals with the states $\mathrm{DL}_{(0,\mathsf{minlen}]}$ in:

proc $\mathsf{cleanup}_\mathrm{D}() \rightarrow$
$\{$ **pre** $\mathsf{Struct}_\mathrm{D}(\mathrm{L})\ \}$
$\mathsf{depths_min}(0, \mathsf{minlen})$
$\{$ **post** $\mathsf{Min} \wedge \mathcal{L} = \mathrm{L}\ \}$
corp

Note that we do not explicitly consider $\mathrm{DL}_0 = \{s\}$ for minimization; that would have been unnecessary according to Property 2.70.

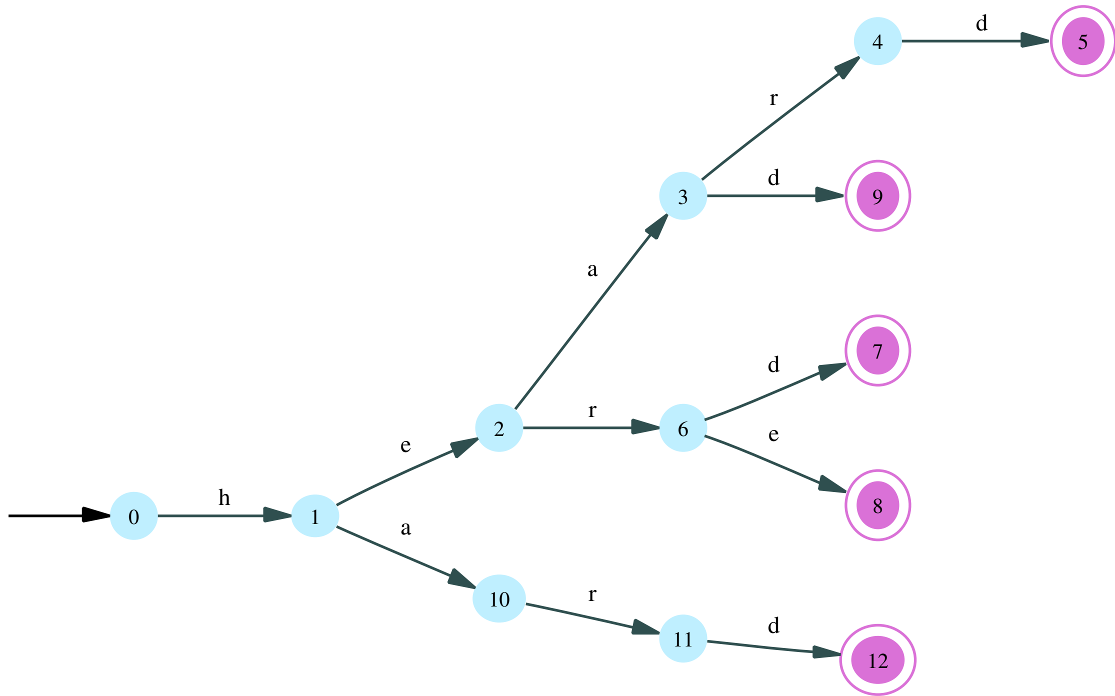
10.3 An example

Consider adding the words (using $\mathsf{add_word}_\mathrm{D}$) in the order `heard`, `herd`, `here`, `head`, `hard`, `her`, `had`, `he`. After adding `herd`, we have

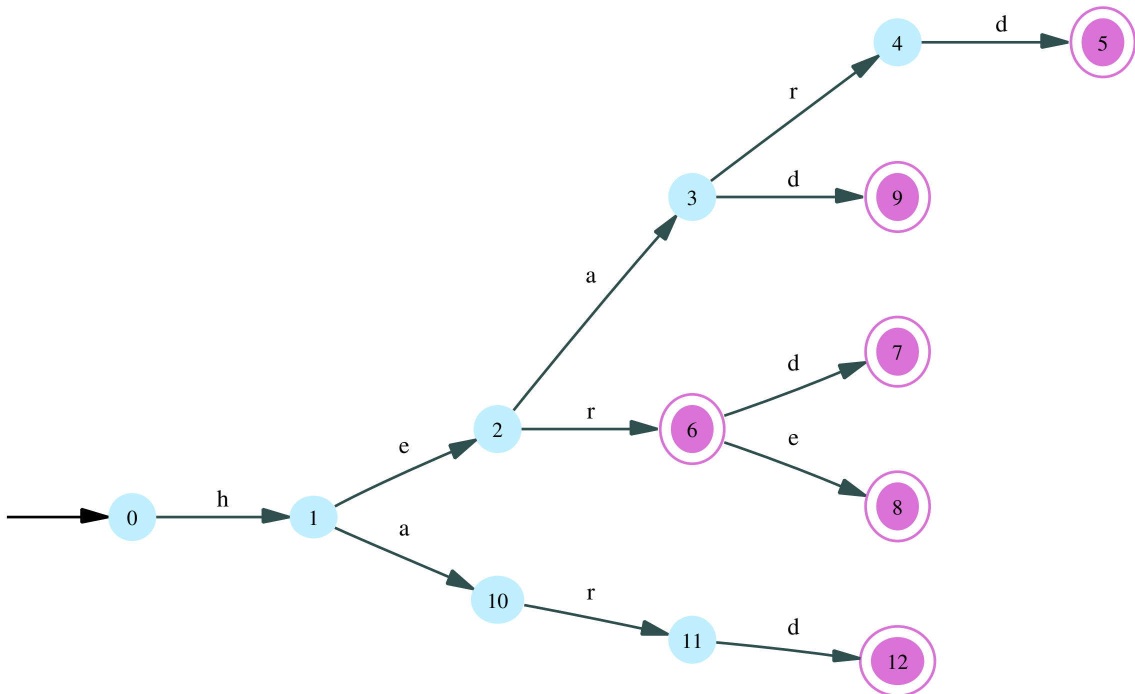


We can now minimize $\mathrm{DL}_{(4,5]} = \{5\}$ against $\mathrm{DL}_{>5} = \emptyset$, and the automaton remains unchanged. After adding `here`, `head`, `hard`, we have

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

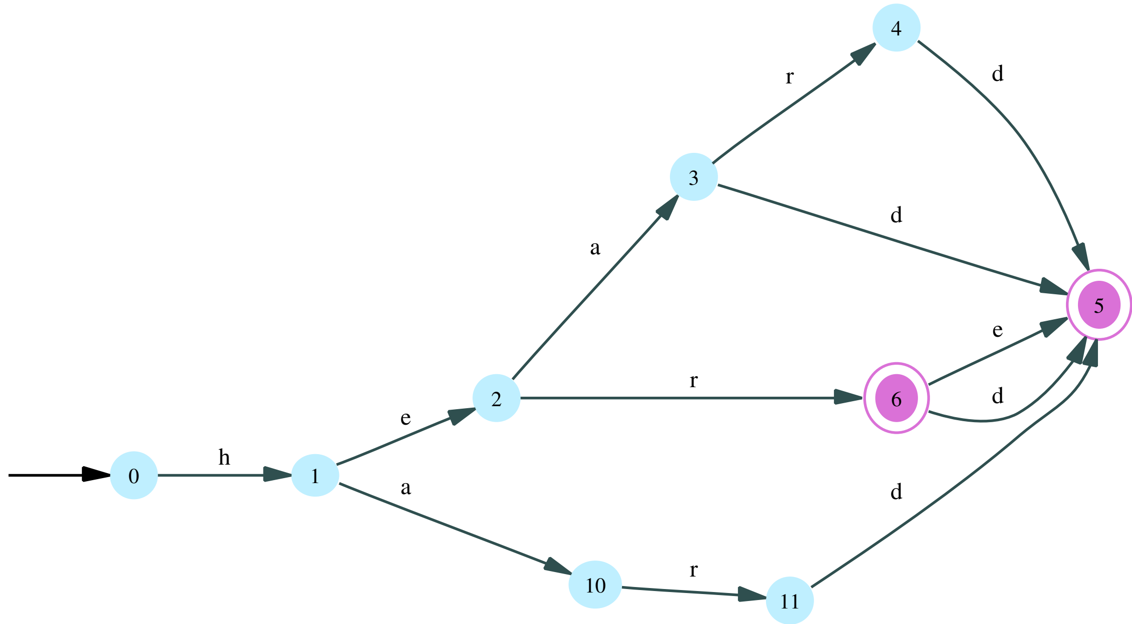


Once `her` is added (state 6 is made final), but before minimization, we have

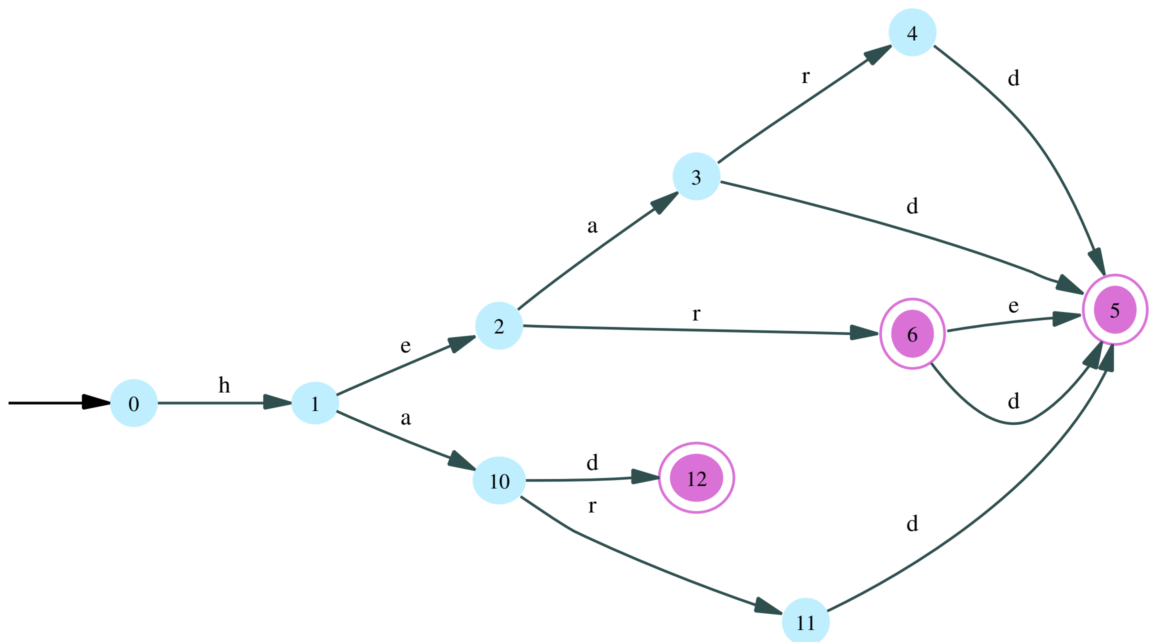


We can minimize states $DL_{(3,4]} = \{4, 7, 8, 9, 12\}$ against $DL_{>4} = \{5\}$ and we see that all except 4 are final and without out-transitions and can be merged into state 5, giving

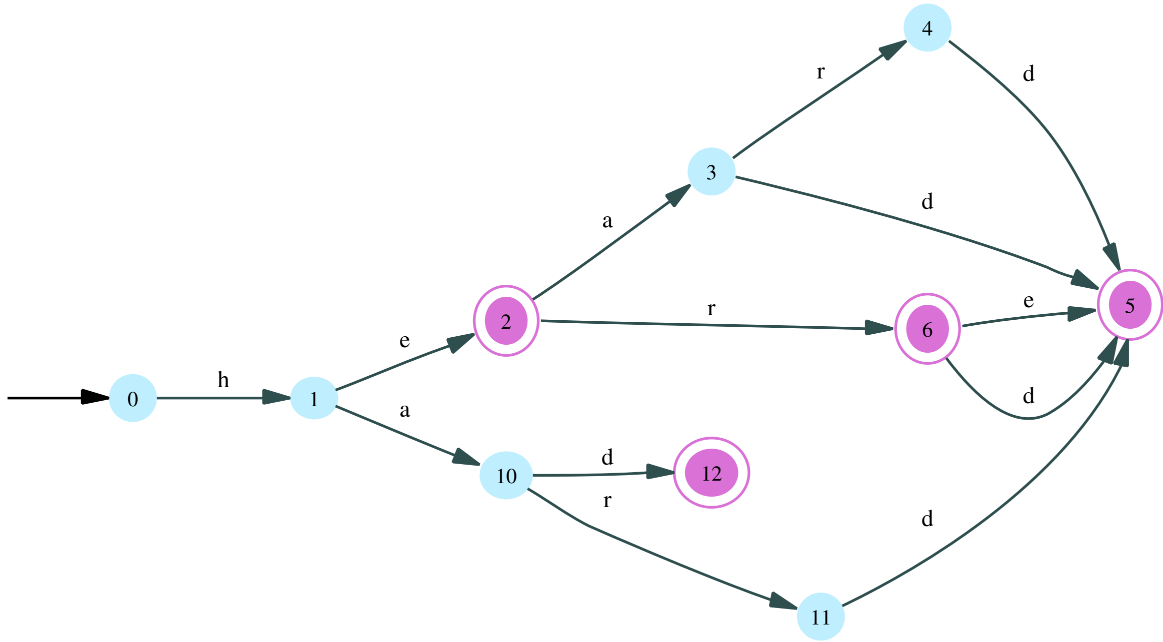
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



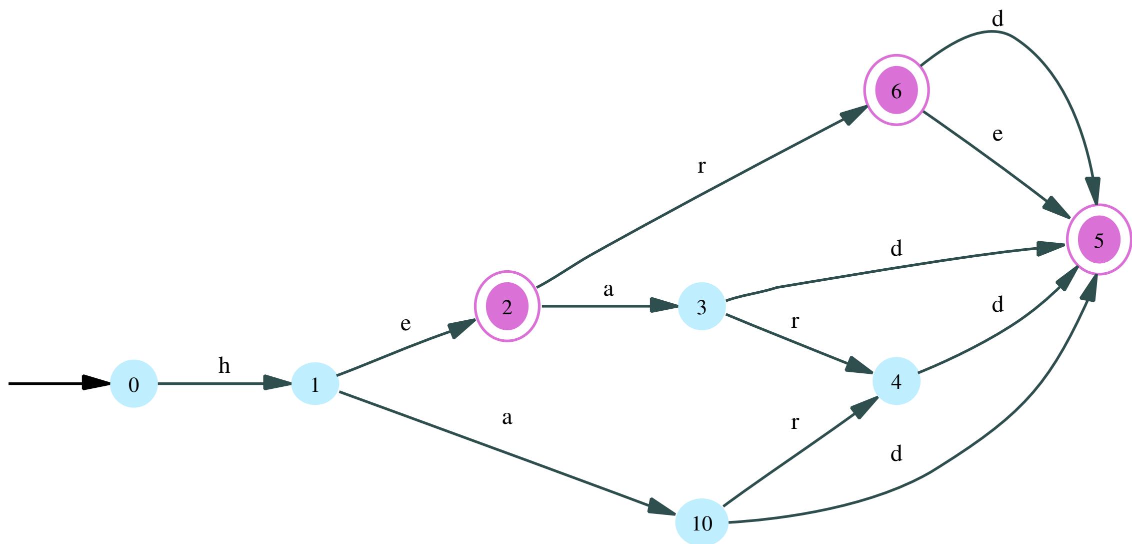
Adding `had` gives



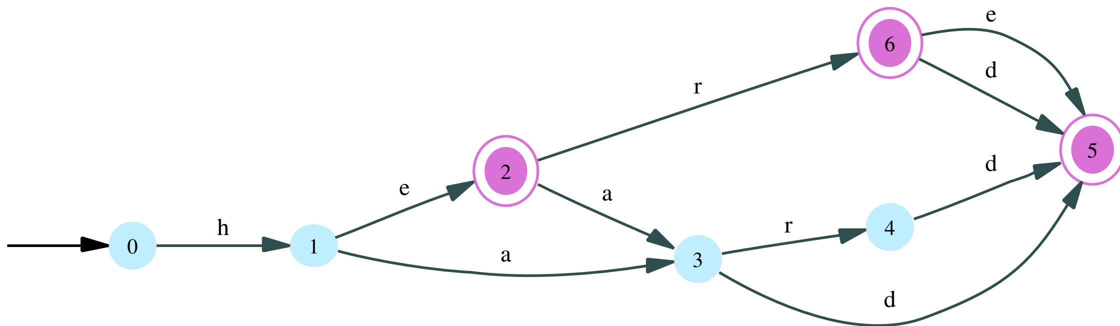
and `depths_min` has nothing to minimize. Finally, adding `he` gives



This means that we can minimize states $DL_{(2,3]} = \{3, 6, 11, 12\}$ against $DL_{>3} = \{4, 5\}$. We see that state 11 can be merged into 4 and 12 into 5, giving



Our cleanup phase ($\mathsf{cleanup}_D$ invoking $\mathsf{depths_min}(0, 2)$) minimizes $DL_{(0,2]} = \{1, 2, 10\}$ against $DL_{>2} = \{3, 4, 5, 6\}$. We can then merge state 10 into 3 giving our minimal automaton



In this particular example, each step (except for the invocation of cleanup_D) considered states at a single depth; this is not generally the case and holds in this example only because our set W consists of words of lengths $5, 4, 3, 2$ — each possible length in $[5, 2]$.

10.4 Time and space performance

Due to the simple implementation of depths_min, this algorithm is not particularly efficient. Word set W can be sorted into *some* order of decreasing length in time and space $\mathcal{O}(|W|)$. Procedures add_word_D and cleanup_D potentially require exponential time and space (due to eq) while adding w.

Conjecture 10.1 *There is a strengthening of the invariant which would allow the use of* eq′*, which is much more efficient.*

More straightforward improvements are immediately possible, including maintaining minlen in a global variable instead of recomputing it.

10.5 Commentary

The algorithm presented here is a new one, not previously appearing in the literature.

Chapter 11

Words by decreasing length: minimizing semi-incrementally

In this chapter, we derive another algorithm which (as in Chapter 10) relies on the words being added in *any* order of decreasing length. Similarly, to avoid the cloning operation (and therefore allow us to use add_word_T), we require the to-be-traversed paths to remain confluence free. Since the words will be added in order of decreasing length, we are guaranteed never to encounter a final state while adding some word w. Intuitively, we could minimize that part of the automaton reachable from final states — a tighter invariant than in Chapter 10. Formally, we maintain invariant

$$\text{Struct}_W(D) \equiv \text{Inequiv}(\underbrace{\text{Succ}^*(F)}_{\text{won't visit again}}) \wedge \text{Confl_free}(\underbrace{Q - \text{Succ}^*(F)}_{\text{may partly visit}}) \wedge \mathcal{L} = D$$

11.1 Procedure add_word_W

Our starting point is

$\textbf{proc}\ \text{add_word}_W(\textbf{in}\ w : \Sigma^*) \rightarrow$
$\{\ \textbf{pre}\ |w| \leqslant \text{minlen}$
$\wedge\ \text{Inequiv}(\text{Succ}^*(F)) \wedge \text{Confl_free}(Q - \text{Succ}^*(F))$
$\wedge\ \mathcal{L} = L\ \}$
$S_{11.1}$
$\{\ \textbf{post}\ |w| = \text{minlen}$
$\wedge\ \text{Inequiv}(\text{Succ}^*(F)) \wedge \text{Confl_free}(Q - \text{Succ}^*(F))$
$\wedge\ \mathcal{L} = L \cup \{w\}\ \}$
corp

After adding word w using add_word_T, we will have made state $\delta^*(s, w)$ final. Since we are adding words in order of decreasing length, and will never pass through state $\delta^*(s, w)$ or any other final state while adding another word, we can minimize states $\text{Succ}^*(\delta^*(s, w))$, keeping in mind that states $\text{Succ}^*(F - \delta^*(s, w))$ will already be minimized according to Struct_W. Our revised version (where we introduce shadow variable F' to capture F for expressing our postconditions) is

---

$\textbf{proc}\ \text{add_word}_W(\textbf{in}\ w : \Sigma^*) \rightarrow$
$\{\ \textbf{pre}\ |w| \leqslant \text{minlen}$
$\wedge\ \text{Inequiv}(\text{Succ}^*(F)) \wedge \text{Confl_free}(Q - \text{Succ}^*(F))$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

$\quad\quad \wedge\ \mathcal{L} = L$
$\quad\quad \wedge\ F' = F\ \}$
$\quad \mathsf{add_word}_T(w);$
$\quad \{\ F = F' \cup \{\delta^*(s,w)\} \wedge \delta^*(s,w) \notin F'\ \}$
$\quad \{\ \mathsf{Inequiv}(\mathsf{Succ}^*(F')) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(F'))$
$\quad\quad \wedge\ \mathcal{L} = L \cup \{w\}\ \}$
$\quad S'_{11.1}$
$\quad \{$ **post** $|w| = \mathsf{minlen}$
$\quad\quad\quad \wedge\ \mathsf{Inequiv}(\mathsf{Succ}^*(F)) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(F))$
$\quad\quad\quad \wedge\ \mathcal{L} = L \cup \{w\}\ \}$
corp

To implement $S'_{11.1}$, we rewrite two of our postcondition conjuncts:

$$\begin{array}{ll} & \mathsf{Inequiv}(\mathsf{Succ}^*(F)) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(F)) \\ \equiv & \quad \text{“value of } F \text{ after } \mathsf{add_word}_T \text{ invocation: } F = F' \cup \{\delta^*(s,w)\}\text{”} \\ & \mathsf{Inequiv}(\mathsf{Succ}^*(F' \cup \{\delta^*(s,w)\})) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(F' \cup \{\delta^*(s,w)\})) \end{array}$$

We can establish this using a helper procedure specified as

---

proc $\mathsf{semi_min}($**in** $p : \textsc{State};$ **in** $U :$ **set of** $\textsc{State}) \rightarrow$
$\quad \{$ **pre** $\mathsf{Inequiv}(\mathsf{Succ}^*(U)) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(U))$
$\quad\quad\quad \wedge\ p \notin \mathsf{Succ}^*(U) \wedge \mathcal{L} = L\ \}$
$\quad S_{11.1.1}$
$\quad \{$ **post** $\mathsf{Inequiv}(\mathsf{Succ}^*(U \cup \{p\})) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(U \cup \{p\}))$
$\quad\quad\quad \wedge\ \mathcal{L} = L\ \}$
corp

(Note the similarities between this procedure's specification and the specification of statement $S'_{11.1}$ above, as well as the similarity to procedure visit_min given in §6.1.1.)

In §11.1.1, we will derive an implementation of semi_min. Using semi_min, we can complete our implementation of $\mathsf{add_word}_W$, where we have changed F' into a program variable for use in our invocation of semi_min:

---

proc $\mathsf{add_word}_W($**in** $w : \Sigma^*) \rightarrow$
$\quad \{$ **pre** $|w| \leqslant \mathsf{minlen}$
$\quad\quad\quad \wedge\ \mathsf{Inequiv}(\mathsf{Succ}^*(F)) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(F))$
$\quad\quad\quad \wedge\ \mathcal{L} = L\ \}$
$\quad [\![$ **var** $F' :$ **set of** $\textsc{State}$
$\quad |\ F' := F;$
$\quad\quad \mathsf{add_word}_T(w);$
$\quad\quad \{\ F = F' \cup \{\delta^*(s,w)\} \wedge \delta^*(s,w) \notin F'\ \}$
$\quad\quad \{\ \mathsf{Inequiv}(\mathsf{Succ}^*(F')) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(F'))$
$\quad\quad\quad \wedge\ \mathcal{L} = L \cup \{w\}\ \}$
$\quad\quad \mathsf{semi_min}(\delta^*(s,w), F')$
$\quad]\!]$
$\quad \{$ **post** $|w| = \mathsf{minlen}$
$\quad\quad\quad \wedge\ \mathsf{Inequiv}(\mathsf{Succ}^*(F)) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(F))$
$\quad\quad\quad \wedge\ \mathcal{L} = L \cup \{w\}\ \}$
corp

11.1.1 Procedure semi_min

Recall semi_min's postcondition

$$\mathsf{Inequiv}(\mathsf{Succ}^*(\mathcal{U} \cup \{p\})) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(\mathcal{U} \cup \{p\}))$$

We rewrite this into a form which will be more easily established by two sequential statements, beginning with the first conjunct:

$$\begin{array}{ll}
& \mathsf{Inequiv}(\mathsf{Succ}^*(\mathcal{U} \cup \{p\})) \\
\equiv & \quad \text{“}\mathsf{Succ}^* \text{ distributes over } \cup\text{; notational shortcut: } \mathsf{Succ}^*(p) \text{ for } \mathsf{Succ}^*(\{p\})\text{”} \\
& \mathsf{Inequiv}(\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^*(p)) \\
\equiv & \quad \text{“Property 2.47”} \\
& \mathsf{Inequiv}(\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p) \cup \{p\}) \\
\equiv & \quad \text{“associativity of } \cup\text{”} \\
& \mathsf{Inequiv}((\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p)) \cup \{p\}) \\
\equiv & \quad \text{“Property 2.83; re-order conjuncts w.r.t. the form of that property”} \\
& \mathsf{Inequiv}(\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p)) \wedge \mathsf{Pairwise_inequiv}(\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p), \{p\}) \wedge \mathsf{Inequiv}(\{p\}) \\
\equiv & \quad \text{“Inequiv always holds on a single state, } p \text{ in this case”} \\
& \underbrace{\mathsf{Inequiv}(\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p))}_{\text{establish in } S'_{11.1.1} \text{ below}} \wedge \underbrace{\mathsf{Pairwise_inequiv}(\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p), \{p\})}_{\text{establish in } S''_{11.1.1} \text{ below}}
\end{array}$$

Intuitively

- $S'_{11.1.1}$ 'minimizes' $\mathsf{Succ}^+(p)$ with respect to our already-minimized states $\mathsf{Succ}^*(\mathcal{U})$. Notably, state p is not to be minimized by $S'_{11.1.1}$. Recall that $\mathsf{Confl_free}(Q - \mathsf{Succ}^*(\mathcal{U}))$ is a precondition conjunct of semi_min. After $S'_{11.1.1}$, additionally states $\mathsf{Succ}^+(p)$ may not be confluences. To capture this we add

  $$\mathsf{Confl_free}(Q - (\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p)))$$

  as a $S'_{11.1.1}$ postcondition conjunct.

- $S''_{11.1.1}$ is to minimize p against states $\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p)$.

This gives us two statements establishing the conjuncts in the derivation above:

$$\begin{array}{l}
\textbf{proc } \mathsf{semi_min}(\textbf{in } p : \textsc{State};\ \textbf{in } \mathcal{U} : \textbf{set of } \textsc{State}) \rightarrow \\
\quad \{\ \textbf{pre } \mathsf{Inequiv}(\mathsf{Succ}^*(\mathcal{U})) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(\mathcal{U})) \\
\quad\quad\quad \wedge\ p \notin \mathsf{Succ}^*(\mathcal{U}) \wedge \mathcal{L} = L\ \} \\
\quad S'_{11.1.1} \\
\quad \{\ \overbrace{\mathsf{Inequiv}(\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p)) \wedge \mathsf{Confl_free}(Q - (\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p)))}^{\text{see derivation above}} \\
\quad\quad \wedge\ \mathcal{L} = L\ \} \\
\quad S''_{11.1.1} \\
\quad \{\ \overbrace{\mathsf{Pairwise_inequiv}(\mathsf{Succ}^*(\mathcal{U}) \cup \mathsf{Succ}^+(p), \{p\})}^{\text{see derivation above}}\ \} \\
\quad \{\ \textbf{post } \mathsf{Inequiv}(\mathsf{Succ}^*(\mathcal{U} \cup \{p\})) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(\mathcal{U} \cup \{p\})) \\
\quad\quad\quad \wedge\ \mathcal{L} = L\ \} \\
\textbf{corp}
\end{array}$$

In the next two sections, we refine $S'_{11.1.1}$ and $S''_{11.1.1}$ respectively.

11.1.1.1 Refining $S'_{11.1.1}$

We can rewrite the first two postcondition conjuncts for $S'_{11.1.1}$

$$\mathsf{Inequiv}(\mathsf{Succ}^*(U) \cup \mathsf{Succ}^+(p)) \wedge \mathsf{Confl_free}(Q - (\mathsf{Succ}^*(U) \cup \mathsf{Succ}^+(p)))$$

$\equiv$ "Property 2.47 twice, once in each conjunct"

$$\mathsf{Inequiv}(\mathsf{Succ}^*(U) \cup \mathsf{Succ}^*(\mathsf{Succ}(p))) \wedge \mathsf{Confl_free}(Q - (\mathsf{Succ}^*(U) \cup \mathsf{Succ}^*(\mathsf{Succ}(p))))$$

$\equiv$ "Succ^* distributes over $\cup$"

$$\mathsf{Inequiv}(\mathsf{Succ}^*(U \cup \mathsf{Succ}(p))) \wedge \mathsf{Confl_free}(Q - (\mathsf{Succ}^*(U \cup \mathsf{Succ}(p))))$$

Note the similarity of the last derivation line to the postcondition conjuncts of semi_min, namely

$$\mathsf{Inequiv}(\mathsf{Succ}^*(U \cup \{p\})) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(U \cup \{p\}))$$

We can most easily establish our predicate by considering the states $\mathsf{Succ}(p)$ one-by-one in recursive invocations of semi_min. We accumulate those states of $\mathsf{Succ}(p)$ in local variable V for later recursive invocations (since they are by then part of the minimized states) in our $S'_{11.1.1}$ refinement (where local variable q is used for iterating over $\mathsf{Succ}(p)$)

```
⋮
{ pre Inequiv(Succ*(U)) ∧ Confl_free(Q − Succ*(U))
      ∧ p ∉ Succ*(U) ∧ ℒ = L }
|[ q : STATE;  V : set of STATE
| V := ∅;
  { invariant: V ⊆ Succ(p) ∧ Inequiv(Succ*(U ∪ V)) ∧ Confl_free(Q − Succ*(U ∪ V))
    variant: |Succ(p) − V| }
  for q : q ∈ Succ(p) →
      as q ∉ U →
         semi_min(q, U ∪ V)
         { Inequiv(Succ*(U ∪ V ∪ {q})) ∧ Confl_free(Q − Succ*(U ∪ V ∪ {q})) }
      sa;
      V := V ∪ {q}
  rof
  { V = Succ(p) }
]|;
{ Inequiv(Succ*(U) ∪ Succ⁺(p)) ∧ Confl_free(Q − (Succ*(U) ∪ Succ⁺(p)))
  ∧ ℒ = L }
⋮
```

11.1.1.2 Refining $S''_{11.1.1}$

We can rewrite the postcondition for $S''_{11.1.1}$ (see page 85 for how we obtained it in the first place)

$$\mathsf{Pairwise_inequiv}(\mathsf{Succ}^*(U) \cup \mathsf{Succ}^+(p), \{p\})$$

$\equiv$ "Property 2.76"

$$\mathsf{Pairwise_inequiv}(\mathsf{Succ}^*(U), \{p\}) \wedge \mathsf{Pairwise_inequiv}(\mathsf{Succ}^+(p), \{p\})$$

$\equiv$ "second conjunct, Corollary 2.74: p is inequivalent to all states $\mathsf{Succ}^+(p)$"

$$\mathsf{Pairwise_inequiv}(\mathsf{Succ}^*(U), \{p\})$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

In our refinement of $S''_{11.1.1}$, we consider p against $\mathsf{Succ}^*(U)$, looking for an equivalent state

---

$\vdots$

$\{ \ \mathsf{Inequiv}(\mathsf{Succ}^*(U) \cup \mathsf{Succ}^+(p)) \wedge \mathsf{Confl_free}(Q - (\mathsf{Succ}^*(U) \cup \mathsf{Succ}^+(p)))$
$\quad \wedge \mathcal{L} = L \ \}$
$[\![$ **var** $q : \mathrm{STATE}$
$|$ **as** $\langle \exists\, q : q \in \mathsf{Succ}^*(U) : \mathsf{eq}(p, q) \rangle \rightarrow$
  let $q : q \in \mathsf{Succ}^*(U) \wedge \mathsf{eq}(p, q)$;
  $\mathsf{merge}(p, q)$; $p := q$
sa
$]\!]$
$\{ \ \mathsf{Pairwise_inequiv}(\mathsf{Succ}^*(U) \cup \mathsf{Succ}^+(p), \{p\}) \ \}$
$\{$ **post** $\mathsf{Inequiv}(\mathsf{Succ}^*(U \cup \{p\})) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(U \cup \{p\}))$
$\qquad \wedge \mathcal{L} = L \ \}$

$\vdots$

The seemingly redundant assignment of q to p in the above **as-sa** statement has a purpose: without it, after the merge of p *into* q, variable/parameter p does not refer to a valid state[1], making its use in our postcondition dubious. State q is of course equivalent, and the assignment allows us to leave the postcondition untouched; in an implementation it would be omitted.

Recall that eq is less efficient than eq′ (see §2.5.3 and 2.5.4). In the precondition of $S''_{11.1.1}$ above, we have conjunct $\mathsf{Inequiv}(\mathsf{Succ}^*(U) \cup \mathsf{Succ}^+(p))$. Thanks to Property 2.86, we also have $\mathsf{Inequiv}(\mathsf{Succ}(p))$ — allowing us to use eq′.

11.1.1.3 A final version of semi_min

The procedure is:

---

proc $\mathsf{semi_min}($**in** $p : \mathrm{STATE}$; **in** $U :$ **set of** $\mathrm{STATE}) \rightarrow$
$\{$ **pre** $\mathsf{Inequiv}(\mathsf{Succ}^*(U)) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(U))$
$\qquad \wedge\, p \notin \mathsf{Succ}^*(U) \wedge \mathcal{L} = L \ \}$
$[\![\ q : \mathrm{STATE}$; $V :$ **set of** STATE
$| \ V := \emptyset$;
$\{$ invariant: $V \subseteq \mathsf{Succ}(p) \wedge \mathsf{Inequiv}(\mathsf{Succ}^*(U \cup V)) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(U \cup V))$
variant: $|\mathsf{Succ}(p) - V| \ \}$
for $q : q \in \mathsf{Succ}(p) \rightarrow$
  as $q \notin U \rightarrow$
    $\mathsf{semi_min}(q, U \cup V)$
    $\{ \ \mathsf{Inequiv}(\mathsf{Succ}^*(U \cup V \cup \{q\})) \wedge \mathsf{Confl_free}(Q - \mathsf{Succ}^*(U \cup V \cup \{q\})) \ \}$
  sa;
  $V := V \cup \{q\}$
rof
$\{ \ V = \mathsf{Succ}(p) \ \}$
$]\!]$;
$\{ \ \mathsf{Inequiv}(\mathsf{Succ}^*(U) \cup \mathsf{Succ}^+(p)) \wedge \mathsf{Confl_free}(Q - (\mathsf{Succ}^*(U) \cup \mathsf{Succ}^+(p)))$
$\quad \wedge \mathcal{L} = L \ \}$
$[\![$ **var** $q : \mathrm{STATE}$

---

[1]Recall from Definition 2.18 that invocation $\mathsf{merge}(p, q)$ deletes p.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
        | as ⟨∃ q : q ∈ Succ*(U) : eq′(p, q)⟩ →
              let q : q ∈ Succ*(U) ∧ eq′(p, q);
              merge(p, q);  p := q
          sa
        ]|
        { Pairwise_inequiv(Succ*(U) ∪ Succ⁺(p), {p}) }
        { post Inequiv(Succ*(U ∪ {p})) ∧ Confl_free(Q − Succ*(U ∪ {p}))
                 ∧ ℒ = L }
corp
```

11.2 Procedure $\mathsf{cleanup}_W$

Once the last word has been added, our cleanup step deals with states $\mathsf{Succ}^*(s) - \mathsf{Succ}^*(F)$. We can rewrite the postcondition for $\mathsf{cleanup}_W$

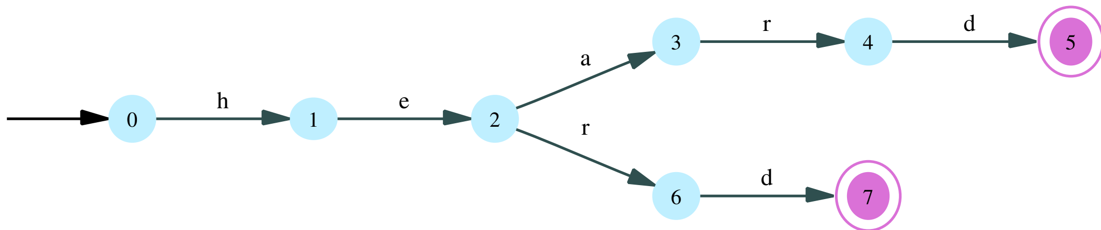
$$
\begin{array}{ll}
 & \mathsf{Min} \\
\equiv & \quad \text{“definition of Min”} \\
 & \mathsf{Inequiv}(Q) \\
\equiv & \quad \text{“no useless states, so all are reachable from } s\text{”} \\
 & \mathsf{Inequiv}(\mathsf{Succ}(s)) \\
\equiv & \quad \text{“set calculus, keeping in mind the postcondition of semi_min”} \\
 & \mathsf{Inequiv}(\mathsf{Succ}^*(F - \{s\} \cup \{s\}))
\end{array}
$$

This is easily established with invocation $\mathsf{semi_min}(s, F - \{s\})$. Our simple cleanup procedure is:

```
proc cleanup_W() →
        { pre Struct_W(L) }
        semi_min(s, F − {s})
        { post Min ∧ ℒ = L }
corp
```

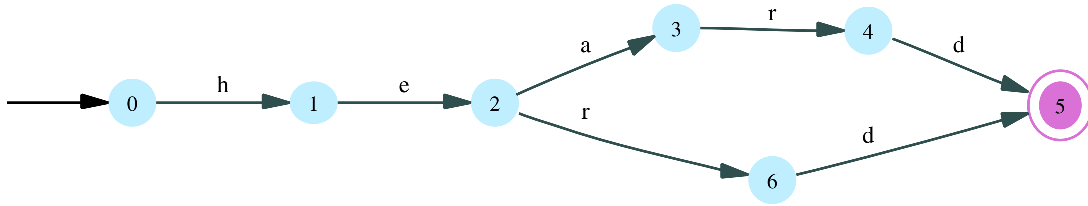
11.3 An example

Consider adding the words (using $\mathsf{add_word}_W$) in the order `heard`, `herd`, `here`, `head`, `hard`, `her`, `had`, `he`. After adding `herd` with $\mathsf{add_word}_T$, we have

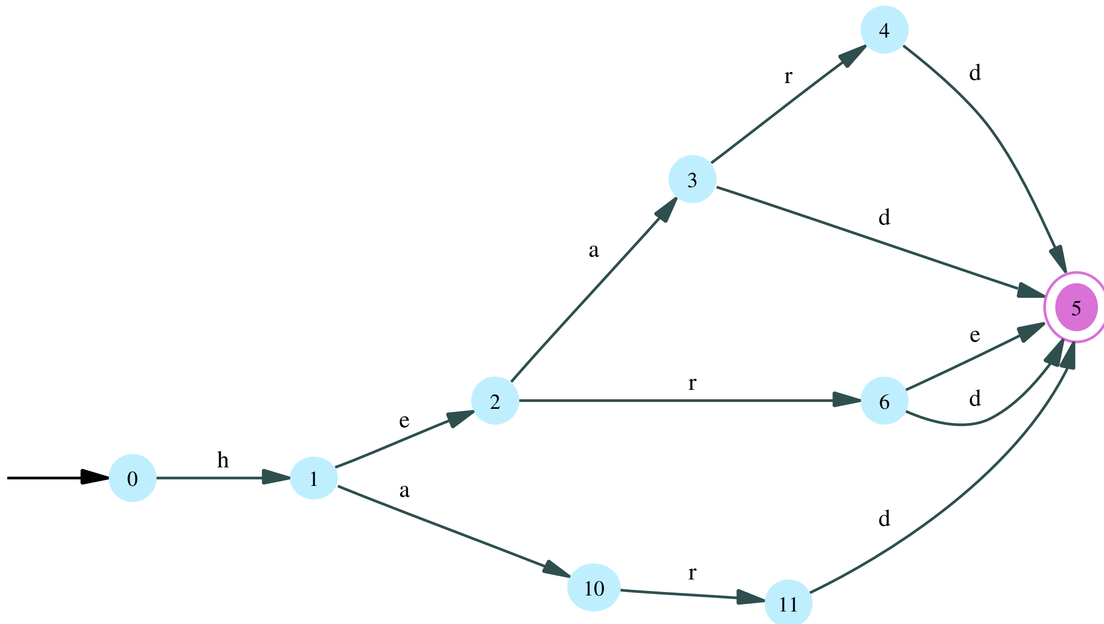


The minimization step considers states $\mathsf{Succ}^*(7) = \{7\}$ against the already-unique set $\mathsf{Succ}^*(5) = \{5\}$ in invocation $\mathsf{semi_min}(7, \{5\})$, giving

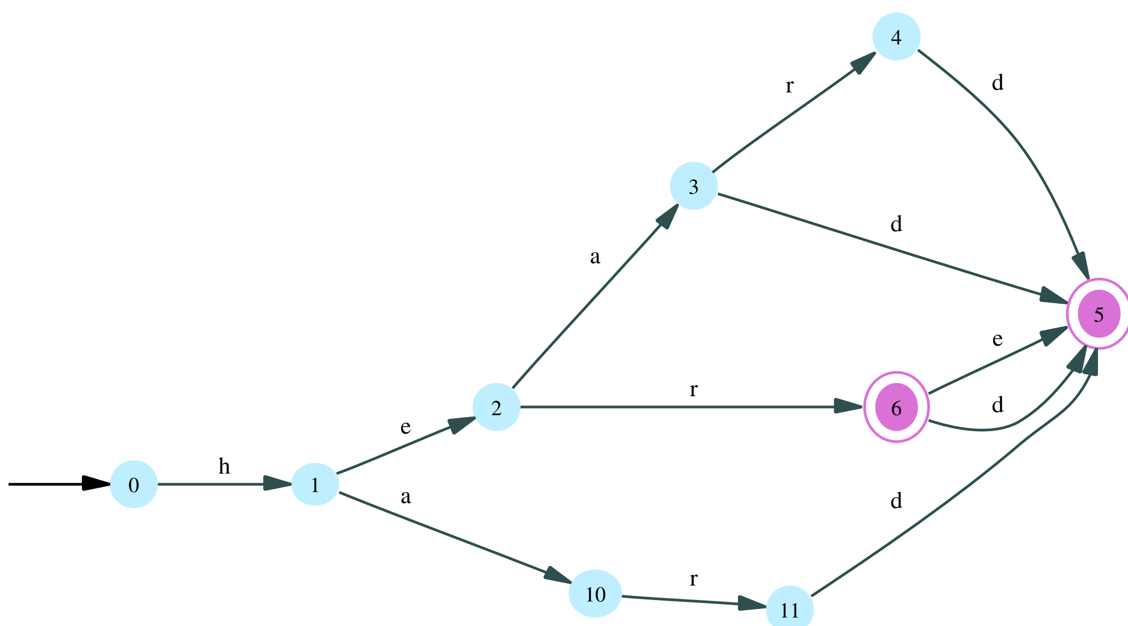
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



After adding `here`, `head`, `hard` and minimizing, we have

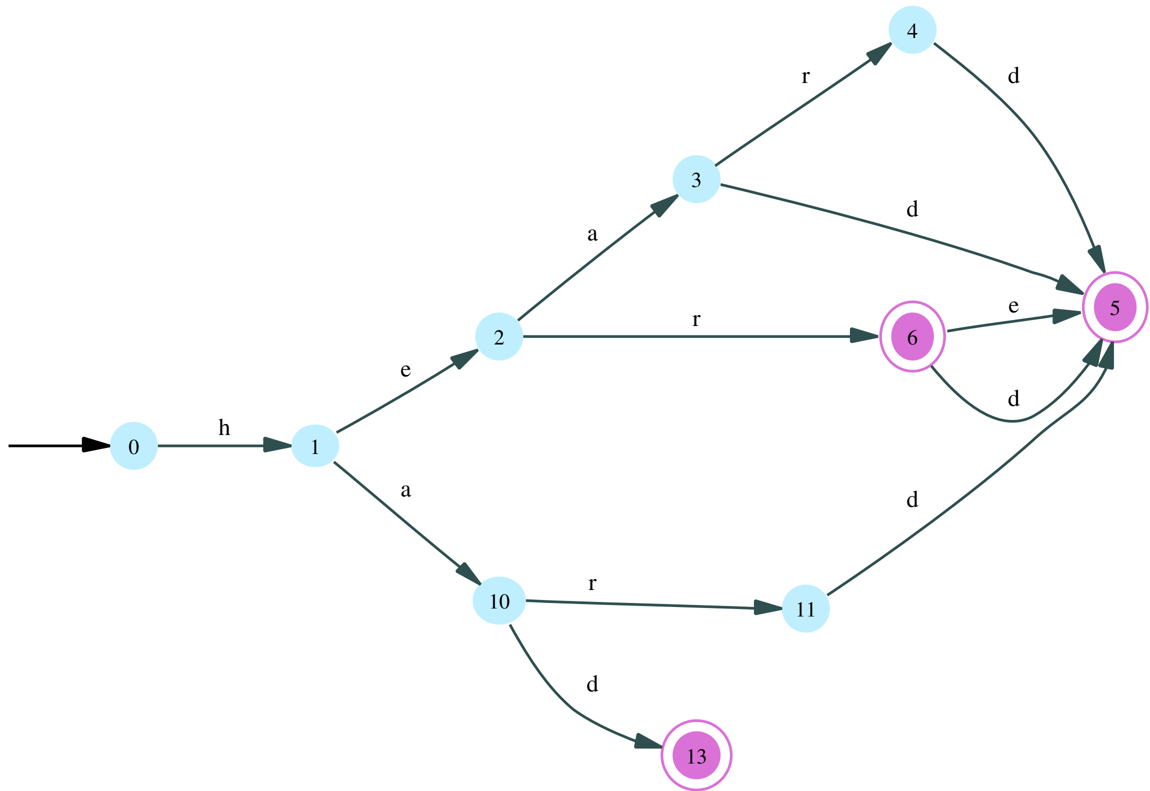


While adding `her` with add_word$_T$, we make state 6 final, yielding an automaton that remains the same after the semi_min$(6, \{5\})$ minimization step (states 6 and 5 are inequivalent)

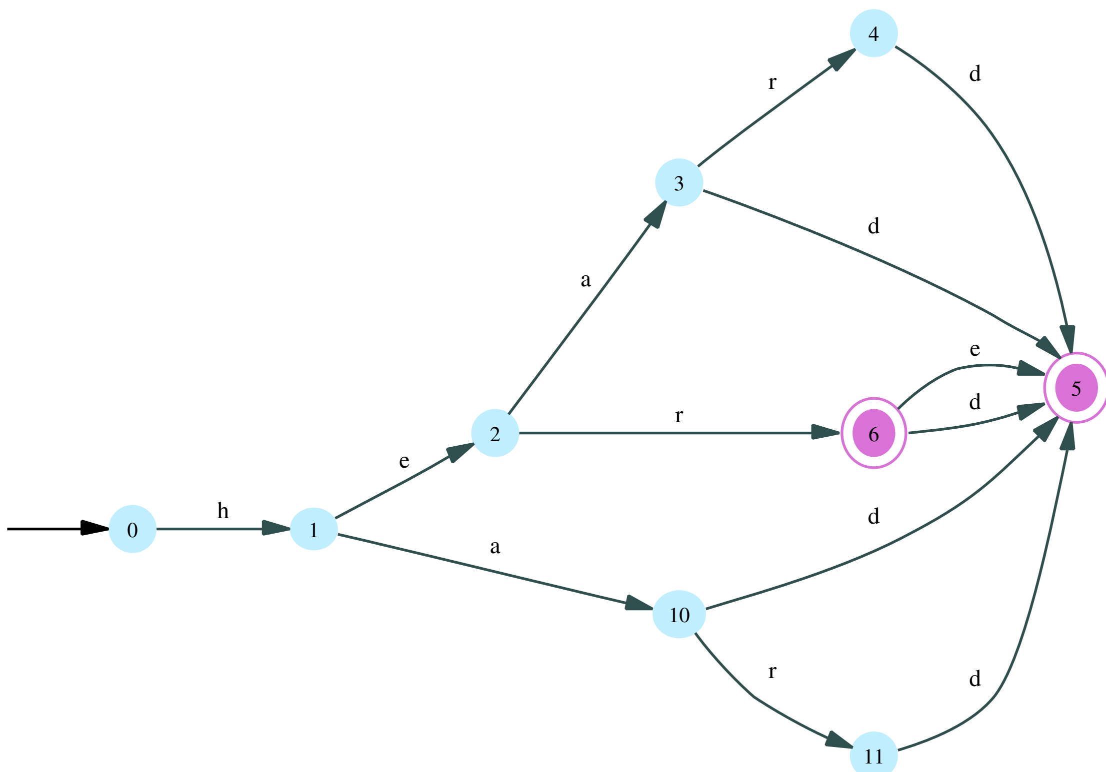


UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

After adding `had` we get a new state and

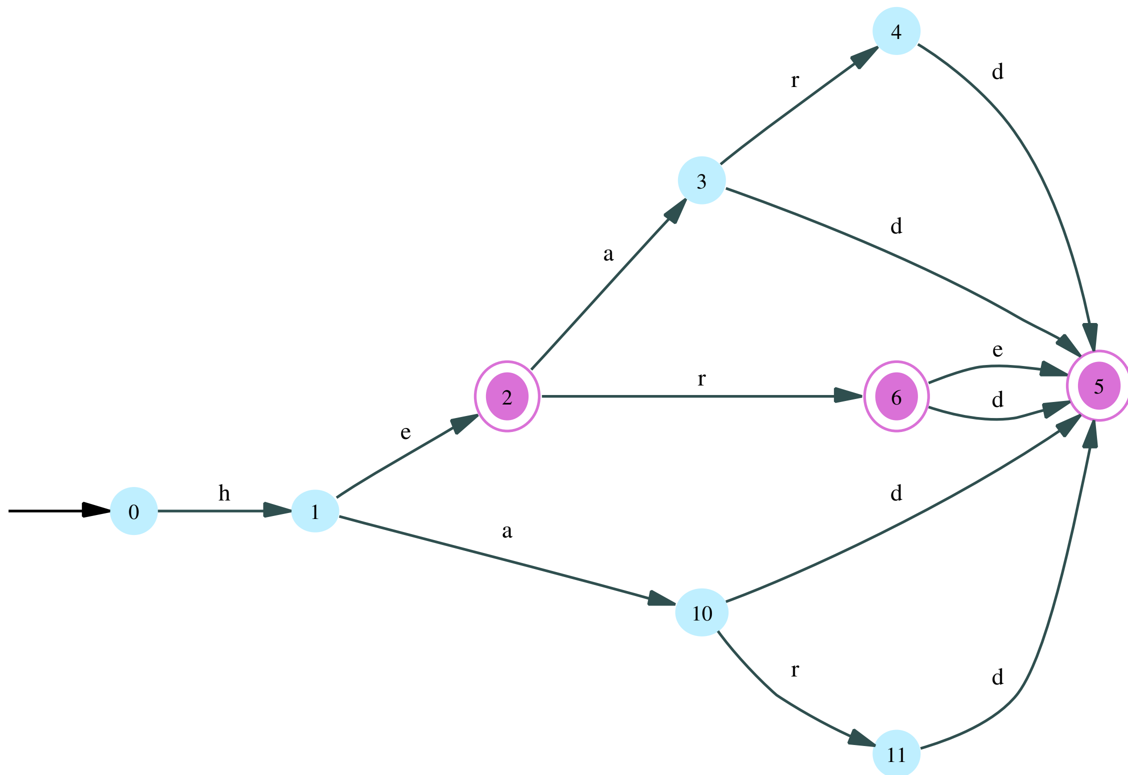


Our minimization step considers $\mathsf{Succ}^*(13) = \{13\}$ against $\mathsf{Succ}^*(\{5,6\}) = \{5,6\}$ in $\mathsf{semi_min}(13, \{5,6\})$, allowing us to merge 13 into 5



UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Finally, after adding `he`, we make state 2 final



The minimization invocation $\mathsf{semi_min}(2, \{5, 6\})$ considers $\mathsf{Succ}^*(2) - \mathsf{Succ}^*(\{5, 6\}) = \{2, 3, 4\}$ against $\mathsf{Succ}^*(\{5, 6\}) = \{5, 6\}$, leaving our automaton unchanged.

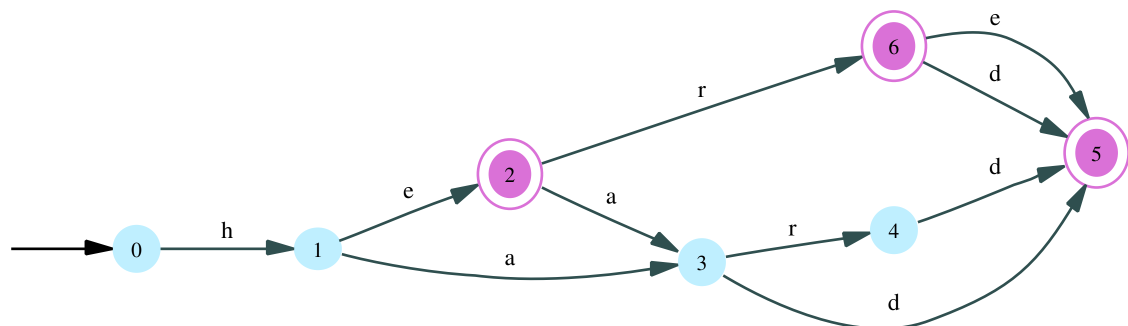
The $\mathsf{cleanup}_W$ invokes $\mathsf{semi_min}(0, \{2, 5, 6\})$ and considers

$$\mathsf{Succ}^*(0) - \mathsf{Succ}^*(\{2, 5, 6\}) = \{0, 1, 10, 11\}$$

(bottom-up) against

$$\mathsf{Succ}^*(\{2, 5, 6\}) = \{2, 3, 4, 5, 6\}$$

and state 11 is merged into 4 and 10 into 3 giving our minimal result



UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

11.4 Time and space performance

Word set W can be sorted into *some* order of decreasing length in time and space $\mathcal{O}(|W|)$. As is shown in [Wat98b, Wat03a], procedure $\mathsf{add_word}_W$ requires time and space $|w|$ while adding w, and $\mathsf{cleanup}_W$ takes time and space $|M|$. In [Wat03a], further performance improvements are discussed.

A simple implementation of this algorithm has proven to be efficient in practice [Wat03a]. In that paper, several other optimizations are discussed.

11.5 Commentary

As mentioned in Chapter 1, this algorithm was a new algorithm presented with a different derivation in [Wat98b] and in [Wat03a].

Bibliography

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 2nd edition, 2007.

[Brz62a] Janusz A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. volume 12 of *MRI Symposia Series*, pages 529–561, Polytechnic Institute of Brooklyn, 1962. Polytechnic Press.

[Brz62b] Janusz A. Brzozowski. *Regular Expression Techniques for Sequential Circuits*. PhD thesis, Princeton University, Princeton, New Jersey, June 1962.

[Bub11] Johannes Bubenzer. Construction of minimal ADFAs. Diplomarbeit, Universität Potsdam, Germany, 2011.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[CD99] Marcin Ciura and Sebastian Deorowicz. Experimental study of finite automata storing static lexicons. Technical report, Silesian Technical University, Poland, November 1999.

[CF02] Rafael C. Carrasco and Mikel L. Forcada. Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(2):207–216, June 2002.

[Cle08] Loek Cleophas. *Taxonomies and Toolkits of Tree Automata Algorithms*. PhD thesis, Eindhoven University of Technology, the Netherlands, April 2008.

[CR94] Maxime A. Crochemore and Wojciech Rytter. *Text Algorithms*. Oxford University Press, 1994.

[CR03] Maxime A. Crochemore and Wojciech Rytter. *Jewels of Stringology*. World Scientific Publishing Company, 2003.

[Dac98] Jan Daciuk. *Incremental Construction of Finite-State Automata and Transducers, and Their Use in Natural Language Processing*. PhD thesis, Technical University of Gdańsk, Poland, 1998.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[DMWW00] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard E. Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April 2000.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[DWW98] Jan Daciuk, Bruce W. Watson, and Richard E. Watson. Incremental construction of minimal acyclic finite state automata and transducers. In Lauri Karttunen and Kemal Oflazer, editors, *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, pages 48–56, Ankara, Turkey, June 1998.

[Fre60] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[GBA01] Jorge Graña, Fco. Mario Barcala, and Miguel A. Alonso. Compilation methods of minimal acyclic finite-state automata for large dictionaries. In Watson and Wood [WW01b], pages 116–129.

[GBY91] Gaston H. Gonnet and Ricardo Baeza-Yates. *Handbook of Algorithms and Data Structures (In Pascal and C)*. Addison-Wesley, second edition, 1991.

[Gri73] David Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[Gri80] David Gries. *The Science of Computer Programming*. Springer-Verlag, second edition, 1980.

[Gus97] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[Hop71] John E. Hopcroft. *An* $n \log n$ *algorithm for minimizing the states in a finite automaton*, pages 189–196. Academic Press, 1971.

[HP98] Gerard J. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 3 (1998)(1), 1998.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[JM00] Daniel Jurafsky and James H. Martin. *Speech and Language Processing*. Prentice Hall, 2000.

[JW96] J. Howard Johnson and Derick Wood. Instruction computation in subset construction. In Raymond et al. [RWY96], pages 64–71.

[Mih99a] Stoyan Mihov. Direct building of minimal automaton for given list. Technical report, Bulgarian Academy of Science, 1999.

[Mih99b] Stoyan Mihov. *Direct Building of Minimal Automaton for Given List*. PhD thesis, Bulgarian Academy of Science, 1999.

[PAMS94] K.-H. Park, Jun-Ichi Aoe, K. Morimoto, and M. Shishibori. An algorithm for dynamic processing of DAWGs. *International Journal of Computational Mathematics*, 54:155–173, 1994.

[Pev00] Pavel Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. MIT Press, 2000.

[Rev91] Dominique Revuz. *Dictionnaires et lexiques: méthodes et algorithmes.* PhD thesis, Institut Blaise Pascal, LITP 91.44, Paris, France, 1991.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[Rev92] Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92:181–189, 1992.

[Rev00] Dominique Revuz. Dynamic acyclic minimal automaton. In Yu and Păun [YP00], pages 226–232.

[RWY96] Darrell Raymond, Derick Wood, and Sheng Yu, editors. *Proceedings of the First Workshop on Implementing Automata*, volume 1260 of *Lecture Notes in Computer Science*, London, Canada, August 1996. Springer-Verlag.

[SFK95] K.N. Sgarbas, N.D. Fakotakis, and G.K. Kokkinakis. Two algorithms for incremental construction of directed acyclic word graphs. *International Journal of Artificial Intelligence Tools*, 4:369–381, 1995.

[SKW08] Tinus Strauss, Derrick G. Kourie, and Bruce W. Watson. A concurrent specification of Brzozowski's DFA construction algorithm. *International Journal of Foundations of Computer Science*, 19(1):125–135, February 2008.

[Smy03] William F. Smyth. *Computing Patterns in Strings*. Addison-Wesley, 2003.

[Wat93a] Bruce W. Watson. A taxonomy of deterministic finite automata minimization algorithms. Technical Report 44, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 1993.

[Wat93b] Bruce W. Watson. A taxonomy of finite automata construction algorithms. Technical Report 43, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 1993.

[Wat94a] Bruce W. Watson. The design of the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions. Technical Report 22, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 1994.

[Wat94b] Bruce W. Watson. An introduction to the FIRE Engine: A C++ toolkit for FInite automata and Regular Expressions. Technical Report 21, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, 1994.

[Wat95] Bruce W. Watson. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Faculty of Computing Science, Eindhoven University of Technology, the Netherlands, September 1995.

[Wat96a] Bruce W. Watson. The FIRE Lite: FAs and REs in C++. In Raymond et al. [RWY96], pages 167–188.

[Wat96b] Bruce W. Watson. Implementing and using finite automata toolkits. In András Kornai, editor, *Proceedings of the Twelfth European Conference on Artificial Intelligence*, pages 97–100, Budapest, Hungary, August 1996.

[Wat96c] Bruce W. Watson. Implementing and using finite automata toolkits. *Journal of Natural Language Engineering*, 2(4):295–302, December 1996.

[Wat97] Bruce W. Watson. Practical optimizations for automata. In Derick Wood and Sheng Yu, editors, *Proceedings of the Second Workshop on Implementing Automata*, volume 1436 of *Lecture Notes in Computer Science*, pages 232–240, London, Canada, September 1997. Springer-Verlag.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[Wat98a] Bruce W. Watson. An early-retirement plan for the states. In Jan Holub, editor, *Proceedings of the Third Prague Stringologic Workshop*, pages 119–124, Prague, Czech Republic, September 1998. Czech Technical University.

[Wat98b] Bruce W. Watson. A fast new semi-incremental algorithm for the construction of minimal acyclic DFAs. In Derick Wood and Denis Maurel, editors, *Proceedings of the Third Workshop on Implementing Automata*, volume 1660 of *Lecture Notes in Computer Science*, pages 91–98, Rouen, France, September 1998. Springer-Verlag.

[Wat99a] Bruce W. Watson. Implementing and using finite automata toolkits. In András Kornai, editor, *Extended Finite State Models of Language*. Cambridge University Press, 1999.

[Wat99b] Bruce W. Watson. The OpenFIRE initiative. In Jun-Ichi Aoe, editor, *Proceedings of the International Conference on Computer Processing of Oriental Languages*, volume 2, pages 421–424, Tokushima, Japan, March 1999.

[Wat99c] Bruce W. Watson. A taxonomy of algorithms for constructing minimal acyclic deterministic automata. In Helmut Jürgensen, editor, *Proceedings of the Fourth Workshop on Implementing Automata*, Potsdam, Germany, July 1999. Springer-Verlag.

[Wat00a] Bruce W. Watson. Directly constructing minimal DFAs: Combining two algorithms by Brzozowski. In Yu and Păun [YP00], pages 242–249.

[Wat00b] Bruce W. Watson. A history of Brzozowski's DFA minimization algorithm. In Yu and Păun [YP00].

[Wat01a] Bruce W. Watson. A history of Brzozowski's DFA minimization algorithm. Technical report, Department of Computer Science, University of Pretoria, South Africa, 2001.

[Wat01b] Bruce W. Watson. An incremental DFA minimization algorithm. Technical report, Department of Computer Science, University of Pretoria, South Africa, 2001.

[Wat01c] Bruce W. Watson. An incremental DFA minimization algorithm. In Lauri Karttunen, Kimmo Koskenniemi, and Gertjan van Noord, editors, *Proceedings of the Second International Workshop on Finite State Methods in Natural Language Processing*, Helsinki, Finland, August 2001.

[Wat01d] Bruce W. Watson. A new recursive algorithm for building minimal acyclic deterministic finite automata. Technical report, Department of Computer Science, University of Pretoria, South Africa, 2001.

[Wat01e] Bruce W. Watson. A taxonomy of algorithms for constructing minimal acyclic deterministic finite automata. *South African Computer Journal*, (27):12–17, 2001.

[Wat02a] Bruce W. Watson. Directly constructing minimal DFAs: Combining two algorithms by Brzozowski. *South African Computer Journal*, 29:17–23, December 2002.

[Wat02b] Bruce W. Watson. A fast and simple algorithm for constructing minimal acyclic deterministic finite automata. *Journal of Universal Computer Science*, 8(2):363–367, 2002.

[Wat03a] Bruce W. Watson. A new algorithm for the construction of minimal acyclic DFAs. *Science of Computer Programming*, 48:81–97, 2003.

[Wat03b] Bruce W. Watson. A new recursive incremental algorithm for building minimal acyclic deterministic finite automata. In Carlos Martin-Vide and Victor Mitrana, editors, *Grammars and Automata for String Processing: From Mathematics and Computer Science to Biology, and Back*, pages 189–200. Taylor and Francis, 2003.

[Wat03c] Bruce W. Watson. A new regular grammar pattern matching algorithm. *Theoretical Computer Science*, 299(1–3):509–521, 2003.

[Wat04] Bruce W. Watson. Reducing memory requirements during finite automata construction. *Software — Practice & Experience*, 34(3):239–248, 2004.

[Wat10] Bruce W. Watson. *Finite automata algorithms*. World Scientific Press, Singapore, 2010.

[WD03] Bruce W. Watson and Jan Daciuk. An efficient incremental DFA minimization algorithm. *Journal of Natural Language Engineering*, 9(1):49–64, 2003.

[WW01a] Bruce W. Watson and Derick Wood, editors. *Preproceedings of the Sixth Conference on Implementations and Applications of Automata*, Pretoria, South Africa, July 2001. University of Pretoria Press.

[WW01b] Bruce W. Watson and Derick Wood, editors. *Proceedings of the Sixth Conference on Implementations and Applications of Automata*, volume 2494, Pretoria, South Africa, July 2001. Springer-Verlag.

[WW03] Bruce W. Watson and Richard E. Watson. A Boyer-Moore-style algorithm for regular expression pattern matching. *Science of Computer Programming*, 48:99–117, 2003.

[WW04] Bruce W. Watson and Derick Wood, editors. *Special Issue: Implementations and Applications of Automata*, volume 313. Journal of Theoretical Computer Science, 2004.

[YP00] Sheng Yu and Andre Păun, editors. *Proceedings of the Fifth Conference on Implementations and Applications of Automata*, volume 2088, London, Canada, July 2000. Springer-Verlag.

[Zwa01] Gerard Zwaan. Personal communication. 2001.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Index

w-path, 13–15, 20, 24, 27–30, 32, 35, 44–46, 49–57, 59, 65, 67–71, 76

ADFA, *see* finite automata, deterministic, acyclic
alphabet, 1, 8–16, 18, 19, 21–24, 27–29, 44, 45, 49–54, 61, 67–69, 75, 76, 83, 84, 100

'big-oh', 21, 22, 41, 46, 59, 61, 63, 73, 81, 92
Booleans, 21, 22
$\perp$, *see* undefined

Confl_free, *see* confluence-free
confluence, 11, 45, 46, 99
confluence-free, 11, 14, 24, 25, 27–30, 44–46, 50–54, 65, 67–71, 75–77, 83–88, 99

δ, *see* transition function
depth level, 16, 25, 75–78, 80
depth-sandwich minimization procedure, 76, 77, 79–81
DFA, *see* finite automata, deterministic

empty string, 8, 9, 12, 13, 15, 17, 18, 20, 28–30, 35, 44–46, 50–55, 59, 69, 71
eq, *see* equivalent state function
equivalence, of states, 17–22, 33, 37
equivalent state function, 21, 22, 33–35, 37, 38, 41, 52–54, 59, 70, 76, 81, 87, 88, 99

false, 7
finite automata
- deterministic, 10, 11, 13, 15, 17, 20, 21, 41, 47, 63, 99
  - acyclic, i, 1, 2, 11, 15–19, 21, 23, 24, 27, 28, 36, 40, 41, 43, 46, 47, 49, 56, 61–63, 99
  - minimal acyclic, i, 1, 17, 23, 44, 46, 54, 62, 65, 73, 99

function, add word, 23–25, 27–29, 38, 41, 43–47, 49–51, 54–56, 59, 61, 63, 65, 67–71, 73, 75–77, 81, 83, 84, 88, 89, 92
function, cleanup, 23–25, 27, 30, 31, 34, 35, 37, 39, 41, 43, 46, 49, 54, 61–63, 71, 73, 77, 80, 81, 88, 91, 92
function, state cloning, 10, 44–46
function, state creation, 10, 24, 29, 30, 41, 45, 46
function, state merging, 10, 11, 16, 17, 33, 34, 37, 38, 53, 54, 67, 70, 76, 87, 88

head of a string, 8, 28–30, 45, 46, 52–54, 59, 69, 70
height level, 15, 16, 20, 34, 36–40

Inequiv, *see* pairwise inequivalent states
Is_confl, *see* confluence
Is_trie, *see* trie

language, of an automaton, 12, 13, 15–19, 24, 25, 27–38, 43–46, 49–54, 61, 62, 67–69, 71, 75–77, 83–88
left language, 12, 13, 16, 17
lexicographic ordering, 9, 17, 67–69
lexicographically greatest word, 17, 24, 67–69, 71
longest common prefix, 9, 67–74
longest right word length function, 15, 16, 18, 19

MADFA, *see* finite automata, deterministic, minimal acyclic
max, 15–19
min, 16, 17, 76
minimality, 17, 20, 24, 25, 30–32, 34, 36, 38, 49–51, 54, 62, 71, 77, 88

naturals, 15, 16, 37, 76

$\leqslant$, *see* partial order on words
pairwise inequivalent sets, 19, 20, 31–33, 35–39, 50–54, 85–88, 100
pairwise inequivalent states, 19, 20, 22, 24, 25, 30–38, 49–54, 67–71, 75–77, 83–88, 99
partial order on words, 23–25, 28, 99

path minimization procedure, 50–55, 57, 59, 69–71, 73, 84, 100
Pairwise_inequiv, *see* pairwise inequivalent sets
powerset, 7, 10

quantification
- existential, 14, 16, 18, 19, 33, 34, 53, 54, 70, 76, 87, 88
- maximum, 15, 16
- minimum, 16, 17
- union, 13, 14, 16, 20, 22, 34, 36–38
- universal, 11, 15, 18–20, 22, 35

reachability of states, 14, 15, 19–22, 25, 34–38, 53, 54, 83–88, 90, 91
right language, 12, 13, 15–19

semi-incremental minimization procedure, 84–91
set cardinality, 11, 15, 16, 18, 21, 22, 24, 28, 30–38, 41, 45, 46, 59, 61, 63, 73, 75, 76, 81, 83, 84, 86, 87, 92
shortest left word length function, 16, 17
shortest word length, 17, 25, 75–77, 81, 83, 84
Σ, *see* alphabet
state universe, 10, 28–38, 44, 45, 53, 54, 76, 84–87
Struct, *see* structural invariant
structural invariant, 23–25, 100
structural invariant, Daciuk-Mihov, 24, 67
structural invariant, depth-based, 25, 75, 77
structural invariant, double-reversal, 24, 61
structural invariant, incremental, 24, 49
structural invariant, nonincremental, 24, 43
structural invariant, nonincremental, trie structures, 24, 27
structural invariant, Watson, 25, 83, 88

tail of a string, 8, 9, 28, 30, 45, 46, 52–54, 59, 69, 70
transition function, 10–16, 18, 19, 21–24, 28–30, 41, 44–46, 50–54, 59, 62, 68–70, 74, 83, 84, 99
trie, 11, 24, 27, 28, 30, 61, 62, 99
true, 7

undefined, 10–12, 28–30, 45, 46, 99

visit_min, *see* path minimization procedure

Colophon

This thesis was produced on a variety of Apple Macintosh Powerbook and Macbook laptops, running MacOS X (the latest being *Snow Leopard*). All of the text, algorithmic and mathematics typesetting was done in LaTeX version 2ϵ in the Charter text font with the Euler math font. The automata diagrams were generated using the *graphviz* graph-drawing tool.