Jan Daciuk

Optimization of Automata

Gdańsk University of Technology
Publishing House

Jan Daciuk

Optimization of Automata

Gdańsk 2014

PRZEWODNICZĄCY KOMITETU REDAKCYJNEGO
WYDAWNICTWA POLITECHNIKI GDAŃSKIEJ
Janusz T. Cieśliński

REDAKTOR PUBLIKACJI NAUKOWYCH
Michał Szydłowski

RECENZENCI
Marek Kubale
Rogério Reis

PROJEKT OKŁADKI
Katarzyna Olszonowicz

Wydano za zgodą
Rektora Politechniki Gdańskiej

Oferta wydawnicza Politechniki Gdańskiej jest dostępna pod adresem
http://pg.edu.pl/wydawnictwo/katalog
zamówienia prosimy kierować na adres wydaw@pg.gda.pl

ISBN 978-83-7348-564-8

Contents

Acronyms **5**

Glossary **7**

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Incrementality . . . 9
1.3 Hashing . . . 10
1.4 Compression . . . 10
1.5 Organization of This Book . . . 10
1.6 Acknowledgements . . . 11

2 Definitions **13**
2.1 Finite-State Automata . . . 13
2.2 Tree Automata . . . 16

3 Sorted Incremental Construction **19**
3.1 Traditional, Non-Incremental Construction . . . 20
3.2 Incremental Construction of Minimal, Acyclic DFAs from sorted data . . . 28
3.3 Extension to Pseudo-Minimal Automata . . . 40
3.4 Extension to Cyclic Automata . . . 45

4 Incremental Construction from Unsorted Data **63**
4.1 Incremental MADFA Construction from Unsorted Data . . . 63
4.2 Extension to Pseudo-Minimal Automata . . . 79
4.3 Extension to Cyclic Automata . . . 89
4.4 Extension to Minimal Bottom-Up Tree Automata . . . 98
4.5 Extension to Pseudo-Minimal Bottom-Up Tree Automata . . . 116

5 Semi-Incremental Construction Algorithms **131**
5.1 Watson's Algorithm . . . 131
5.1.1 Extension to Pseudo-Minimal Automata . . . 137
5.1.2 Extension to Cyclic Automata . . . 144
5.1.3 Extension to Minimal Bottom-Up Tree Automata . . . 152
5.1.4 Extension to Pseudo-Minimal Bottom-Up Tree Automata . . . 164

5.2 Revuz's Algorithm . . . 166
5.3 Summary . . . 167

6 Hashing **171**
6.1 Perfect Hashing with Minimal DFAs --- Numbers in States . . . 172
6.2 Perfect Hashing with Minimal DFAs --- Numbers in Transitions . . . 176
6.3 Arbitrary Hashing with Pseudo-Minimal DFAs . . . 178
6.4 Variable Output Transducers . . . 180
6.5 Dynamic Perfect Hashing . . . 182
6.6 Perfect Hashing with Minimal DTAs . . . 185
6.7 Arbitrary Hashing with Pseudo-Minimal DTAs . . . 194
6.8 Summary . . . 194

7 Incremental Minimization Algorithms **197**
7.1 Original Algorithm by Bruce Watson . . . 197
7.2 Faster Minimization . . . 200
7.2.1 Presorting of States . . . 200
7.2.2 Saving Fewer State Pairs in S . . . 201
7.2.3 Full Memoization . . . 201
7.2.4 Improved Version . . . 203
7.3 Simplified Version . . . 208
7.4 New Version . . . 209

8 Memory-Efficient Representations **211**
8.1 Sharing Space . . . 212
8.2 Compressing Fields . . . 216
8.3 Using Frequency . . . 216
8.4 Summary . . . 217

9 Summary **219**

Bibliography **221**

Index **225**

Acronyms

CIAA Conference on Implementation and Applications of Automata. 8

DFA deterministic finite automaton. 9, 109, 148

DTA deterministic tree automaton. 12--14, 109, 111, 123, 125--129, 139

FSA finite-state automaton. 9

FSMNLP Finite State Methods in Natural Language Processing. 7

ISSCO Istituto Dalle Molle per gli studi semantici e cognitivi. 7

NLP natural language processing. 7, 213, 249

XML Extensible Markup Language. 211

Glossary

F final (accepting) states. 9

Q finite set of states. 9

$sgnt(q)$ signature of a state q, i.e. its finality and the suite of its outgoing transitions. 11, 14, 23

$|S|$ cardinality of a set S (number of items in S). 9

δ transition function. 9

δ^* extended transition function. 9

ε empty string. 9

$\mathcal{L}(M)$ language of an automaton M. 10

$\overleftarrow{\mathcal{L}}_m\ (q)$ left language of a state q in an automaton M. 10

$\overrightarrow{\mathcal{L}}_M\ (q)$ right language of state q in an automaton M. 10

$\simeq$ pseudo-equivalence relation on states. 11

q_0 start (initial) state. 9

Σ alphabet. 9

Σ_q output alphabet of a state, set of symbols serving as labels on outgoing transitions of state q. 10

$\perp$ sink state, target of undefined transitions, $\perp \notin Q$. 9

Σ^* string of symbols. 9, 67

Chapter 1

Introduction

1.1 Motivation

Finite-state machines are abstract mathematical devices that have found important practical applications in many areas, like natural language processing and computational linguistics, compiler construction, design of electronic circuits, computer graphics, communication protocols, requirements specification, etc. Variety of applications call for a variety of algorithms.

This book is conceived as an effort to gather all algorithms and methods developed or co-developed by the author of the book that concern three aspects of optimization of automata: incrementality, hashing, and compression. Some related algorithms and methods are given as well when they are needed to complete the picture. The author worked on the algorithms while preparing tools for natural language processing, and that domain remains their most frequent application area. This does not exclude other uses.

1.2 Incrementality

Minimal deterministic automata have the smallest number of states among all deterministic automata that recognize (or generate) the same language. As a result, they also occupy the least space in memory. In all applications where large automata are static, it is profitable to minimize them. Traditional automata construction algorithms obtain minimal automata in two steps. In the first step, a non-minimal automaton is constructed. In the second step, the automaton is minimized. While the minimal automaton is usually small, intermediate automata, e.g. tries, can be huge. Incremental construction algorithms construct minimal automata in one step, i.e. without the need for an intermediate automaton. After addition of each item, the automaton is kept very close to the minimal one. Those algorithms can be seen as algorithms modifying a minimal automaton so that it recognizes more items while maintaining minimality. Semi-incremental algorithms proceed in two steps, but the intermediate automaton is not much larger than the minimal one, and little work is needed to make it minimal. Small (in case of semi-incremental algorithms) or minimal (in case of fully incremental ones) size means that the algorithms require little memory. Similar algorithms exist for pseudo-minimal automata.

Incrementality in minimization algorithms has a special sense. All minimization algorithms except the incremental ones have unusable intermediate results. Most of them start by dividing states of an automaton into two classes: final and non-final states. Those classes are further divided until all states in all classes are equivalent to other states in their class. One class in the final partition is one state in the minimal automaton. Brzozowski's algorithm [5] is different in that it inverts the automaton, determinizes it, and then inverts and determinizes it again. This process also renders a usable automaton only at the end of the process. Incremental minimization finds pairs of equivalent states, deleting redundant states and redirecting transitions accordingly. After each such operation, an intermediate automaton may not be minimal, but it recognizes the same language, and it is smaller than the original one. When there are time limits for minimization, it can be interrupted after any minimization step.

1.3 Hashing

Although quite a lot of information can be encoded into a finite automaton, some information cannot be stored there without inflating the size of the automaton beyond reasonable bounds. In such cases, hashing techniques implement mapping from strings in the language of the automaton to any integer, or in case of pseudo-minimal automata, to just about any other data. Those integers can then serve as indexes to other data structures, such as e.g. vectors of pointers.

Hashing is implemented by placing some numbers either in states or in transitions, or in both. A mapping from strings to numbers can then be calculated in various ways. Usually numbers found along paths representing strings are added, but other solutions (one number per path, keep the last number found) are also possible.

Numbers needed for implementation of hashing can be added after construction, but they can also be added during it, which can lead to faster processing. Placing appropriate information for hashing implementation may be part of construction algorithms.

1.4 Compression

Minimization is not the only method to reduce the size of an automaton. Various compression techniques serve the same purpose, and they can be applied independently of minimization. They rely on various representation methods for automata. Compression not only reduces memory requirements for programs that use automata, but it can make them faster, as it reduces the time needed to load the automaton representation into memory. Some other representations of dictionaries would require additional time for initialization of their data structures. However, some representations may slow down word look-up. Certain representations are more suitable for some applications than for others. Automata can be optimized for specific needs.

1.5 Organization of This Book

The next chapter contains basic definitions that are used throughout this book. The following three chapters present construction algorithms for deterministic finite-state automata and de-

terministic tree automata. They construct either minimal (Chapter 3 and Chapter 4) or pseudo-minimal (Chapter 5) automata, and the algorithms are either incremental or semi-incremental. The choice of algorithms reflects the author's participation in development of either the basic versions of them, or their extensions.

Chapter 6 is dedicated to hashing implemented with automata --- both finite-state automata and tree automata. Chapter 7 describes incremental minimization developed by Bruce Watson and improved by the author of the present book. Chapter 8 discusses various compression methods for deterministic finite-state automata. The last chapter is a summary.

1.6 Acknowledgements

My research in finite automata began in 1996, when my friends and collaborators from Istituto Dalle Molle per gli studi semantici e cognitivi (ISSCO), Geneva, Switzerland (notably Susan Armstrong) took me to Archamps just behind the border with France. There was a natural language processing (NLP) seminar there with Lauri Karttunen who talked about finite-state transducers. He convinced me that finite automata and transducers were the best representation for dictionaries.

What Lauri did not say at the seminar was how to construct the finite-state machines. I had to invent my own algorithms. Back at ISSCO, Dominique Petitpierre encouraged me to work on a construction algorithm for unsorted data, Pierrette Boullion and Sabine Lehmann provided data for my first dictionaries.

The work done at ISSCO led to my Ph.D. thesis [12] prepared under supervision of Zygmunt Vetulani. It could not be completed without support of Stanisław Szejko --- my boss at that time. Max Silberztein was one of my reviewers, and he showed me his Intex --- a final (then) result of many years of research of French researchers.

Working on a paper for Finite State Methods in Natural Language Processing (FSMNLP) in Ankara in 1998, I contacted Bruce Watson. This led to many years of collaboration, and a few papers we wrote together. I learned much from him, and he influenced my research like no one else.

In Ankara, I met Gertjan van Noord --- my future boss in Alfa Informatica, Rijksuniversiteit Groningen, the Netherlands. Alfa Informatica was by far the best research environment I worked in. I could further develop my finite-state tools (and algorithms) there.

I met Agata Savary in Pretoria at Conference on Implementation and Applications of Automata (CIAA). She organized my visit to Blois, France, where I worked with her and with Denis Maurel on incremental and semi-incremental algorithms, and on hashing. She has excellent organizational skills, and she is exceptionally nice person too.

Mikel Forcada and Rafael Carrasco invited me to Alicante in hope to work together on finite automata. The result was an incremental construction algorithm for tree automata, which inspired further research, e.g. on perfect hashing with tree automata. I find the Spanish people very hospitable.

Dawid Weiss reimplemented my software in Java, and together with Marcin Miłkowski was for years my main source of feedback for my software. Later, he much improved my compression techniques. Marco Almeida discovered a bug in the improved incremental minimization algorithm that I developed with Bruce. He worked then with Nelma Moreira and Rogério Reis on their version of the algorithm. I also appreciate time spent with Nelma and

Rogério in Prague and in Blois. I worked with Kuba Piskorski and Strahil Ristiv on a chapter of a book. My work with Kuba concerned mostly compression. Strahil sent me updates on his state-of-the-art compression method in time for including a short info about it in this book.

My current boss Bogdan Wiszniewski made my life miserable recently starting every conversation with a question about my habilitation thesis. I had to start working on this book. I was also encouraged to do this work by our dean Krzysztof Goczyła and past vice dean Michał Mrozowski.

Chapter 2

Definitions

In the following definitions, we will use a notion of string. Let us call a set of symbols an **alphabet**. An alphabet is usually denoted as Σ. A **string** is a sequence of symbols of any finite length (including 0). The length of a string w is written as $|w|$. This is the same notation as for cardinality of a set, i.e. $|S|$ is the number of items in S. An empty string (a string of length 0) is denoted as ε. Note that $|\varepsilon| = 0$. An arbitrary sequence of items of the same type is denoted with a raised star, so to indicate that w is a string, we write $w \in \Sigma^*$. We index individual symbols in a string starting from 1, so $w = w_1 w_2 \dots w_{|w|}$.

2.1 Finite-State Automata

There are two kinds of a finite-state automaton (FSA): deterministic and nondeterministic one. We will use only deterministic automata throughout this book. A deterministic finite automaton (DFA) is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the **alphabet**, $\delta : Q \times \Sigma \to Q$ is the **transition function**, q_0 is the **start state** or the **initial state**, and $F \subseteq Q$ is the set of final states. If transition function is defined for every combination of state and symbol, then the automaton is called **complete**. Otherwise it is called **incomplete**. If for a certain $q \in Q$ and $a \in \Sigma$, $\delta(q, a)$ is not defined, we write $\delta(q, a) = \perp$.

The transition function δ can be extended to δ^* to act on strings rather than on single symbols. If $q \in Q$, $a \in \Sigma$, and $w \in \Sigma^*$, then

$$\begin{aligned} \delta^*(q, \epsilon) &\equiv q \\ \delta^*(q, aw) &\equiv \delta^*(\delta(q, a), w) \end{aligned} \tag{2.1}$$

A **path** in an automaton is defined as a sequence of transitions, such that a target of a transition is the source state of another transition. States are said to belong to the path if they are either source or target state of any transition in the path.

$$\pi = (q_1, \sigma_1, q_2)(q_2, \sigma_2, q_3) \dots (q_n, \sigma_n, q_{n+1}), (\forall 1 \leq i \leq n)\delta(q_i, \sigma_i) = q_{i+1} \tag{2.2}$$

The **right language** of a state q is defined as

$$\overrightarrow{\mathcal{L}}_M\ (q)_M \equiv \{w \in \Sigma^* | \delta^*(q, w) \in F\} \tag{2.3}$$

The **language of an automaton** M is defined as

$$\mathcal{L}(M)_M \equiv \{w \in \Sigma^* | \delta^*(q_0, w) \in F\} \tag{2.4}$$

Note that $\mathcal{L}(M) = \overrightarrow{\mathcal{L}}\ (q_0)$. If $\mathcal{L}(M) = L$ then we say that the automaton M recognizes the language L. The automaton is called **recognizer** as opposed to a **transducer**, which has output. The lower index M can be dropped if the context is obvious. We define the **output alphabet of a state** as a set of symbols on its outgoing transitions:

$$\Sigma_q = \{\sigma : \delta(q, \sigma) \neq \perp\} \tag{2.5}$$

The right language can also be defined recursively:

$$\overrightarrow{\mathcal{L}}\ (q) \equiv \{a\ \overrightarrow{\mathcal{L}}\ (\delta(q, a)) | a \in \Sigma_q\} \cup \begin{cases} \{\epsilon\} & \text{if } q \in F \\ \emptyset & \text{if } q \notin F \end{cases} \tag{2.6}$$

The **left language** of a state q is defined as:

$$\overleftarrow{\mathcal{L}}_m\ (q) \equiv \{w \in \Sigma^* | \delta^*(q_0, w) = q\} \tag{2.7}$$

Let $|S|$ be the cardinality of a set S. Among all automata recognizing the same language L, there is one (up to isomorphism) that has the smallest number of states. Such automaton is called **minimal automaton**.

$$(\forall M \text{ such that } \mathcal{L}(M) = \mathcal{L}(M_{min}))\ |Q| \geq |Q_{min}| \tag{2.8}$$

Two states are **equivalent** if and only if their right languages are equal.

$$(p \equiv q) \Leftrightarrow (\overrightarrow{\mathcal{L}}\ (p) = \overrightarrow{\mathcal{L}}\ (q)) \tag{2.9}$$

This equivalence relation divides all states of an automaton into classes of abstraction. In a minimal automaton, all classes have only one element. In other words, in a minimal automaton, if a pair of states is found equivalent, there is no pair of different states that are equivalent.

A state is q **reachable** if there is $w \in \Sigma^*$ such that $\delta^*(q_0, w) = q$. A state is **unreachable** if it is not reachable. A state q is **co-reachable** if there is $w \in \Sigma^*$ such that $\delta(q, w) \in F$. If in an automaton no pair of states is equivalent, there are no unreachable and no co-unreachable states, the automaton is minimal.

There is also a notion of pseudo-equivalence. A state is **divergent** if cardinality of its right language is greater than 1. A state p is **pseudo-equivalent** to another state q, denoted as $p \simeq q$ if they have the same right language and they are not divergent.

$$(p \simeq q) \Leftrightarrow \left(\left(\overrightarrow{\mathcal{L}}\ (p) = \overrightarrow{\mathcal{L}}\ (q)\right) \wedge \left(|\ \overrightarrow{\mathcal{L}}\ (p)| \leq 1\right)\right) \tag{2.10}$$

An automaton, in which if two states are pseudo-equivalent, they must be the same state, is a **pseudo-minimal automaton**. An alternative definition is possible. A state q is **confluence** if it has more than 1 incoming transition. A pseudo-minimal automaton is an automaton that has the smallest number of states among all automata that recognize the same language, and in which there is no path such that a confluence state follows a divergent one, or a state is both divergent and confluence.

We define a function *fin* that verifies finality of a state.

$$fin(q) = \begin{cases} 0 & \text{if } q \notin F \\ 1 & \text{otherwise} \end{cases} \tag{2.11}$$

A **signature** of a state is defined as its finality and a suite of its outgoing transitions (their labels and their targets).

$$sgnt(q) = (fin(q), \{(\sigma, \delta(q, \sigma)) : \sigma \in \Sigma_q\}) \tag{2.12}$$

If $(\exists w \in \Sigma^*, w \neq \varepsilon, \exists q \in Q)\delta^*(q, w) = q$, then the automaton is **cyclic**. Otherwise the automaton is **acyclic**.

A *height* $h(q)$ of a state q in an acyclic automaton is defined as:

$$h(q) = \max_{w \in \vec{\mathcal{L}}q} |w| \tag{2.13}$$

and it can be computed as:

$$h(q) = \begin{cases} 1 + \max_{\sigma \in \Sigma_q} h(\delta(q, \sigma)) & \Sigma_q \neq \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{2.14}$$

An automaton is usually drawn as a directed graph. The states are drawn as circles, final states as double circles, transitions as labeled arcs. The initial state has an incoming arc that does not come from any state. The graph is called a **transition diagram**.

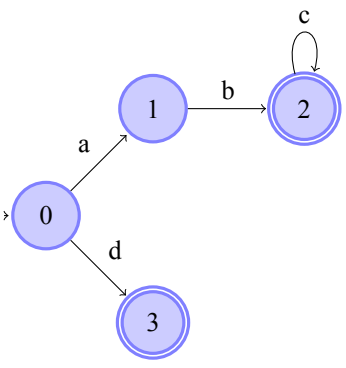


Figure 2.1: A transition graph of an automaton $A = (Q, \Sigma, \delta, q_0, F$, where $Q = \{0, 1, 2, 3\}$, $\Sigma = \{a, b, c, d\}$, $\delta(0, a) = 1$, $\delta(0, d) = 3$, $\delta(1, b) = 2$, $\delta(2, c) = 2$, $q_0 = 0$, $F = \{2, 3\}$.

An example of a transition graph of an automaton is shown in Figure 2.1. The language of the automaton is $\mathcal{L}(A) = abc^*|d$, i.e. it contains $d, a, ab, abc, abcc, abccc, \ldots$ and so on. The

right language of states is $\overrightarrow{\mathcal{L}}(0) = \mathcal{L}(M)$, $\overrightarrow{\mathcal{L}}(2) = c^*$, $\overrightarrow{\mathcal{L}}(1) = b\,\overrightarrow{\mathcal{L}}(2) = bc^*$, $\overrightarrow{\mathcal{L}}(3) = \varepsilon$. The automaton is minimal.

2.2 Tree Automata

Trees are defined as follows:

1. Each symbol $\sigma \in \Sigma$ is a tree.
2. For each $t = \sigma(t_1, \ldots, t_m)$, where $\sigma \in \Sigma$, and $t_1, \ldots, t_m$, $m \geq 0$ are trees, t is a tree.

Any subset of all trees T_Σ defined over an alphabet Σ is called a *tree language*. Trees are recognized (or generated) by tree automata.

There are two kinds of tree automata: *bottom-up* and *top-down*. Bottom-up automata are also called *frontier-to-root*, and top-down automata are also called *root-to-frontier*. Tree automata recognize trees. Bottom-up ones recognize trees starting from leaves, top-down start from the root. In this book, we use only bottom-up, deterministic tree automata since they are more powerful than their top-down counterparts. When referring to a bottom-up, deterministic tree automaton, we will often use a shorter name --- a deterministic tree automaton (DTA).

A *finite-state, bottom-up tree automaton* [11] is defined as $A = (Q, \Sigma, \Delta, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the *alphabet*, $\Delta = \left\{\tau_1, \ldots, \tau_{|\Delta|}\right\} \subset \bigcup_{i=0}^{m} \Sigma \times Q^{m+1}$ is a finite set of *transitions*, and $F \subseteq Q$ is a set of *final states*. Final states are also known as *accepting states*. In a *deterministic*, finite-state, bottom-up tree automaton, for each $(\sigma, q_1, \ldots, q_m) \in \Sigma \times Q^m$, $m \geq 0$, there is at most one $q \in Q$ such that $\tau = (\sigma, q_1, \ldots, q_m, q) \in \Delta$. In that case, we can define a function δ_m:

$$\delta_m(\sigma, q_1, ..., q_m) = \begin{cases} q & \text{if } q \in Q \text{ is such that } (\sigma, q_1, ..., q_m, q) \in \Delta \\ \perp & \text{if no such } q \in Q \text{ exists} \end{cases} \tag{2.15}$$

We will drop the index m in δ_m in unambiguous cases. States $q_1, \ldots, q_m$ are *source states*, and q is a *target state*. We can define an extended transition function on trees:

$$\delta_A(t) = \begin{cases} \delta_0(\sigma) & \text{if } t = \sigma \in \Sigma \\ \delta_m(\sigma, \delta_A(t_1), \ldots, \delta_A(t_m)) & \text{if } t = \sigma(t_1 \cdots t_m) \in T_\Sigma \setminus \Sigma \end{cases} \tag{2.16}$$

A language of a state q in an automaton A is a set of trees such that the extended transition function returns q for each of them:

$$L_A(q) = \{t \in T_\Sigma : \delta_A(t) = q\} \tag{2.17}$$

That language can be computed recursively:

$$L_A(q) = \bigcup_{\tau=(\sigma,r_1,\ldots,r_m,s)\in\Delta:s=q} \sigma(L_A(r_1), \ldots, L_A(r_m)) + \bigcup_{(\sigma,s)\in\Delta:s=q} \sigma \tag{2.18}$$

A language of the whole automaton A is the union of the languages of all its final states:

$$\mathcal{L}(A) = \bigcup_{q \in F} L_A(q). \tag{2.19}$$

Two automata A and B are equivalent if $\mathcal{L}(A) = \mathcal{L}(B)$. Two states p and q are equivalent (written $p \equiv q$) if and only if:

$$p \in F \leftrightarrow q \in F \tag{2.20}$$

and for all $m > 0$, all $k \leq m$, and all $\tau = (\sigma, r_1, \ldots, r_m) \in \Sigma \times Q^m$:

$$\delta_m(\sigma, r_1, \ldots, r_{k-1}, p, r_{k+1}, \ldots, r_m) \equiv \delta_m(\sigma, r_1, \ldots, r_{k-1}, q, r_{k+1}, \ldots, r_m) \tag{2.21}$$

The size of a DTA $A = (Q, \Sigma, \Delta, F)$ is the size of all its transitions:

$$|A| = |\Delta| \tag{2.22}$$

A **minimal deterministic tree automaton** A_{min} is the smallest automaton among all automata that recognize the same language:

$$A_{min} = A : \forall_{A' : \mathcal{L}(A') = \mathcal{L}(A)} \; |A| \geq |A_{min}| \tag{2.23}$$

The size of a DTA is the size of all its transitions, but the minimal DTA has also the minimal number of states. In a minimal automaton, each equivalence class for $\equiv$ has exactly one state.

Let $R_A(q)$ denote a set of trees in $T_{\Sigma \cup \{\#\}}$ that have exactly one leaf node labeled $\# \notin \Sigma$, and that the result of replacing that node with a tree in $L_A(q)$ is a tree in $\mathcal{L}(A)$. That node is a place holder used to generalize concatenation, so that $t\#s$ is a tree in $\mathcal{L}(A)$ obtained by replacing the node labeled # in t with a subtree s.

$$R_A(q) = \{t \in T_{\Sigma \cup \{\#\}} : \delta_A(t\#s) \in F \text{ for all } s \in L_A(q)\} \tag{2.24}$$

$R_A(q)$ plays the role analogous to the right language in FSAs. Two states p and q are equivalent $p \equiv q$ if they have identical R_A:

$$(p \equiv q) \Leftrightarrow (R_A(p) = R_A(q)) \tag{2.25}$$

A state q is called *divergent* if $|R_A(q)| > 1$.

Two states p and q are pseudo-equivalent, denoted as $p \simeq q$, if they are equivalent, and they are not divergent (their languages contain more than one tree):

$$(p \simeq q) \equiv ((p \equiv q) \wedge (|L_A(q)| > 1 \Rightarrow |R_A(q)| = 1)) \tag{2.26}$$

Pseudo-equivalence is an equivalence relation. In a *pseudo-minimal DTA*, each equivalence class has exactly one element. In a pseudo-equivalent DTA, each tree has a transition that is traversed when recognizing that tree and that is not traversed when recognizing any other tree. W call it a *proper* transition.

As in the case of ordinary finite-state automata, we define function *fin* that acts on a state:

$$\mathit{fin}(q) = \begin{cases} 0 & \text{if } q \notin F \\ 1 & \text{otherwise} \end{cases} \tag{2.27}$$

We also define a function *posn* that acts on a state q and a transition τ that returns all the positions of q as a source state of a transition:

$$\mathit{posn}(q, \tau = (\sigma, r_1, \ldots, r_m, n)) = \{i : r_i = q, 1 \leq i \leq m\} \tag{2.28}$$

The **signature** of a state q takes into account finality of the state as well as the label, the target, and positions of the source state q in all outgoing transitions of the state:

$$sgnt(q) = (fin(q), \bigcup_{\tau=(\sigma,r_1,\ldots,r_m,n):\mathit{posn}(q,\tau)\neq\emptyset} (\sigma, \mathit{posn}(q, \tau), n)) \tag{2.29}$$

A *fanout* of a state q is defined as:

$$\mathit{fanout}(q) = \{(\sigma, r_1, \ldots, r_m, n, i) : (\sigma, r_1, \ldots, r_m, n) \in \Delta, 1 \leq i \leq m, r_i = q\} \tag{2.30}$$

A *fanin* of a state q is defined as:

$$\mathit{fanin}(q) = \{(\sigma, r_1, \ldots, r_m, n) \in \Delta : n = q\} \tag{2.31}$$

The language of a transition $\tau = (\sigma, q_1, \ldots, q_m, q) \in \Delta$ is the subset of $L_A(q)$ recognized by following τ:

$$L_A(\tau) = \begin{cases} \sigma & \tau = (\sigma, q) \\ \bigcup_{(t_1,\ldots,t_m)\in L_A(q_1)\times\ldots\times L_A(q_m)} \sigma(t_1, \ldots, t_m) & \tau = (\sigma, q_1, \ldots, q_m, q) \end{cases} \tag{2.32}$$

A state $q \in Q$ is **reachable** if there is a tree t such that $\delta_A(t) = q$. A state is **unreachable** if it is not reachable. A state q is **co-reachable** if either

$$q \in F \tag{2.33}$$

$$\exists(\sigma, r_1, \ldots, r_m, n) \in \Delta : r_i = q \wedge 1 \leq m \wedge n \text{ is co-reachable} \tag{2.34}$$

Chapter 3

Incremental Construction from Sorted Data

This chapter is the first in a series of three chapters that present algorithms that construct either minimal or pseudo-minimal deterministic finite-state automata or minimal or pseudo-minimal deterministic bottom-up tree automata. The algorithms are selected because of their incrementality or semi-incrementality. There are other algorithms that can be seen as incremental, e.g. the Thompson construction [1]. They are not included here because they use different input and they are loosely related to the work of the author of this book.

Each chapter is divided into sections where the basic algorithm and its extensions are presented. The algorithm is described first in its basic version --- that is a version that constructs a minimal, acyclic, deterministic finite-state automaton. Then extensions of the algorithm are presented, in which pseudo-minimal acyclic automata, cyclic automata, and minimal, deterministic, acyclic bottom-up tree automata (if applicable) are constructed. Such presentation makes it easier for the author to underline similarities between various extensions of the same algorithm. See Section 5.3 for a summary of the algorithms and recommendations.

Incremental algorithms, presented in this chapter and in the next one, construct an automaton (a finite state automaton or a tree automaton) by adding items to the language of the automaton one by one in such a way that the size of the automaton during construction (measured in e.g. the number of states) remains very close to the size of the minimal or pseudo-minimal (depending on the extension of the algorithm) automaton recognizing a language consisting of all items added to the automaton so far. There are two families of such algorithms: incremental algorithm for sorted data (described in this chapter) and incremental algorithm for unsorted data (described in the next chapter, page 63).

The family of algorithms described in this chapter is the incremental construction for sorted data. This family uses the fact that data comes lexicographically sorted. The algorithms in this family are faster than competing algorithms. We start by presenting an algorithm that constructs a minimal acyclic deterministic finite-state automaton from a finite set of strings, and then present its extensions: construction of acyclic pseudo-minimal automata (page 40), and adding strings to cyclic automata (page 45).

3.1 Traditional, Non-Incremental Construction

Let us start with a description of a traditional method of constructing a minimal deterministic finite-state automaton. It can be divided into two steps. The first step consists of an algorithm that constructs an automaton that is not necessarily minimal. The second step is a minimization algorithm. Incremental construction obtains mininmal (or almost-minimal) automaton after each new string has been added. It can also be regarded as an algorithm for adding a string or a set of strings to an already existing automaton.

The simplest way to construct a deterministic automaton recognizing a set of strings is to construct an automaton in the form of a **trie**. A trie is a rooted tree with labels on edges consisting of single characters (usually letters). Each string is recognized along the path from the initial state to one of the final states. In a trie, one final state is associated with exactly one string. An algorithm for constructing a trie can be summarized as Algorithm 3.1.

Algorithm 3.1 Algorithm for constructing a trie --- function ConstructTrie.

```
1: function ConstructTrie
2:     Create M = ({q0}, Σ, ∅, q0, ∅)
3:     while input not empty do
4:         w ← newstring
5:         AddStringToTrie(M, w)
6:     end while
7:     return M
8: end function
```

Most of the work in the function ConstructTrie is done in procedure AddStringToTrie outlined as Algorithm 3.2.

Algorithm 3.2 Algorithm for adding a string to a trie --- function AddStringToTrie. M is a trie, and w is a string to be added. The procedure modifies M.

```
1: procedure AddStringToTrie(M, w)
2:     while There is a transition in M labeled with the next symbol of w do
3:         Follow the transition
4:     end while
5:     while There is a next symbol σ left in w do
6:         Create a new state p
7:         Create a transition from the current state to p labeled with σ
8:         Follow that transition
9:     end while
10:    Make the current state final
11: end procedure
```

Algorithm 3.2 can be rewritten more formally as Algorithm 3.3. Note that neither of the two `while` loops needs to be entered. If the first loop is not entered, the first symbol of the string to be added is different from the first symbol in any string currently stored in the trie. If the second loop is not entered, either the string is already in the trie, or it is a prefix of a

string already in the trie. In the first case, the trie remains intact (q is already in F). In the latter case, F is modified, i.e. one state becomes final.

Algorithm 3.3 Algorithm (more formal) for adding a string to a trie --- function AddStringToTrie. M is a trie, and w is a string to be added. The procedure modifies M.

```
 1: procedure AddStringToTrie(M, w)
 2:     q ← q_0
 3:     i ← 1
 4:     while δ(q, w_i) ≠ ⊥ do
 5:         q ← δ(q, w_i)
 6:         i ← i + 1
 7:     end while
 8:     while i ≤ |w| do
 9:         p ← new state
10:         δ(q, w_i) ← p
11:         i ← i + 1
12:         q ← p
13:     end while
14:     F ← F ∪ {q}
15: end procedure
```

An example of a trie is shown in Figure 3.1. It can be seen that every state but the initial one has one incoming transition. All leaf states are final states. Internal states can also be final. In such case, the automaton recognizes words that are prefixes of other words also contained in the language of the automaton.

Let us see how the trie construction algorithm works in practice. Our input data is a subset of the language of the trie in Figure 3.1 --- words: *bili*, *biliśmy*, *bijęę*, and *bij*, in that order. As specified in Algorithm 3.1, we start with an automaton having only a start state $q_0 = 0$. We add the word *bili*. Since there is only one state and no transitions in the automaton, the loop in lines 4--7 of Algorithm 3.3 does not run. The loop in lines 8--13 creates a chain of states 0, 1, 2, 5, and 12 (we use state number compatible with Figure 3.1), as well as a sequence of transitions between them labeled with subsequent letters of the word. In line 14 of Algorithm 3.3, state 12 is made final. The result can be seen in Figure 3.2.

The next word is *biliśmy*. The previous word ---*bili*--- is its prefix. Therefore, the loop in lines 4--7 traverses all transitions of the automaton, recognizing the prefix *bili*. Next, a chain of states 23, 45, and 48 is created, as well as transition between state 12 and state 23 labeled with *ś*, a transition between state 23 and state 45 labeled with *m*, and a transition between state 45 and state 48 labeled with *y*. In line 14 of Algorithm 3.3, state 48 is made final. The situation at the end is shown in Figure 3.3.

Now it is time to add *bijęę*. The first two letters are recognized by the trie in Figure 3.2. The transitions that bear them as labels are traversed in the `while` loop in lines 4--7 of Algorithm 3.3. The last two letters have to be added. The loop in lines 8--13 creates a chain of states 4 and 7, a transition from state 2 to state 4 labeled with *j*, and a transition from state 4 to state 10 labeled with *ę*. In line 14 of Algorithm 3.3, state 10 is made final. The result can be seen in Figure 3.4.

The last word to be added is *bij*. The first `while` loop uses all letters of the word. When

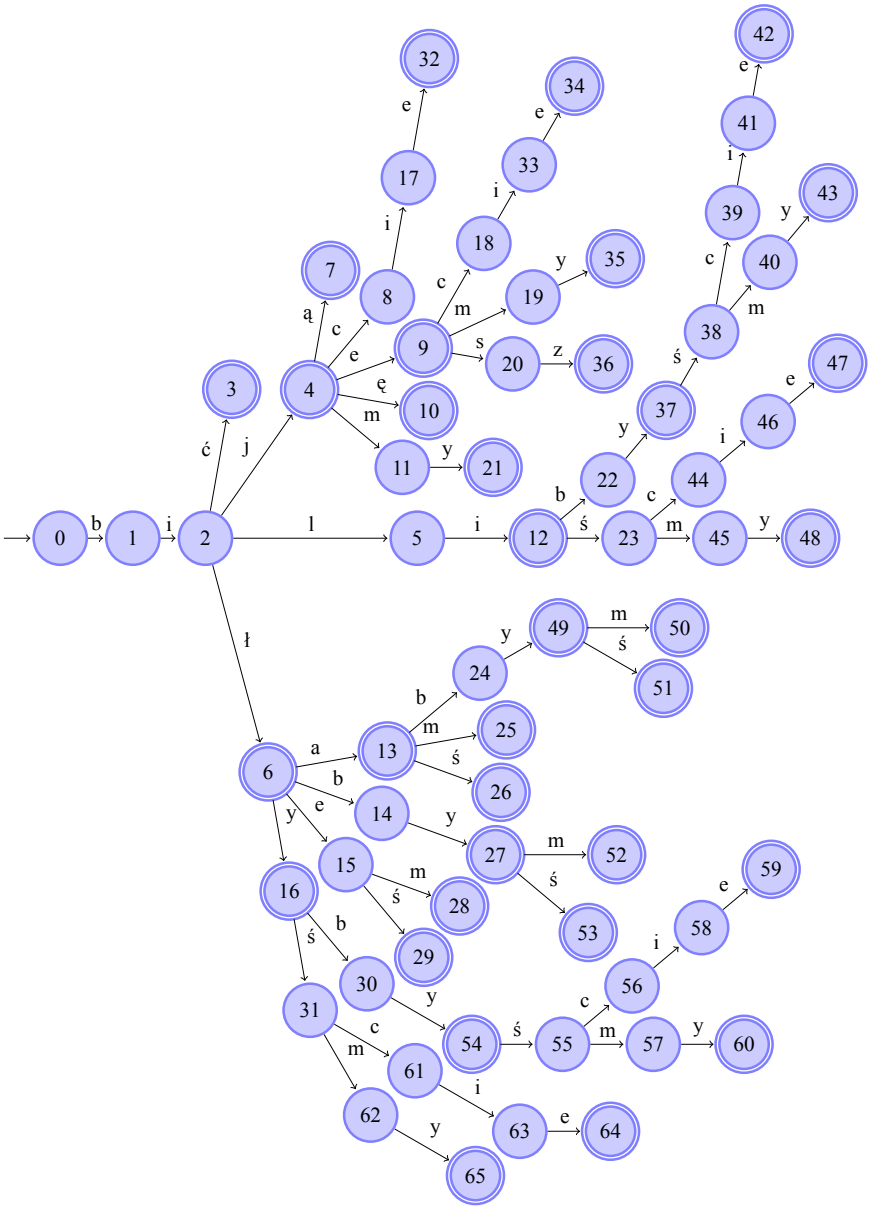


Figure 3.1: Example of a trie. The automaton recognizes flectional forms of a polish verb *bić*. Derived forms are not shown.

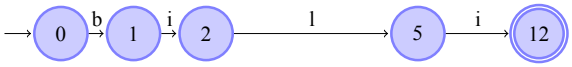


Figure 3.2: Example of a trie construction. Word *bili* has just been added to an empty automaton.

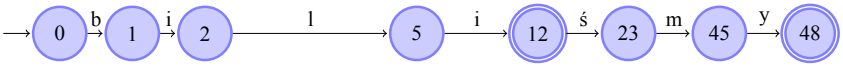


Figure 3.3: Example of a trie construction. Word *biliśmy* has just been added to an automaton already recognizing word *bili*.

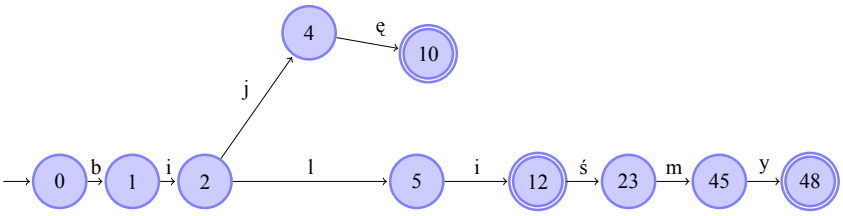


Figure 3.4: Example of a trie construction. Word *biją* has just been added to an automaton containing words *bili* and *biliśmy*.

we get to state 4, no letters of *bij* are left for building a new chain of states and transitions. The second loop is not run. However, the trie does not yet recognize the word *bij*. Instruction in line 14 does the trick --- state 4 is made final. The automaton is shown in Figure 3.5 (note that state 4 is now final).

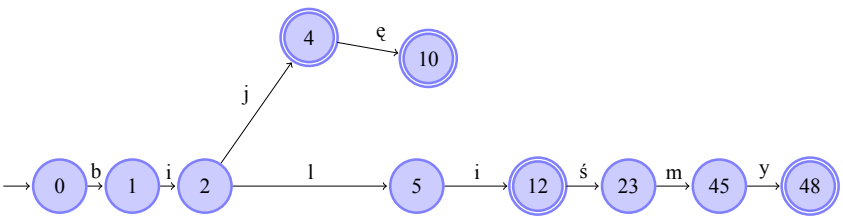


Figure 3.5: Example of a trie construction. Word *bij* has just been added to an automaton containing words *bili*, *biliśmy*, and *bij*.

Our goal is to obtain a minimal automaton. Therefore, the trie needs to be minimized. We could use general purpose minimization algorithms: Hopcroft algorithm [24] (the fastest), Hopcroft-Ullman algorithm [25], or Aho-Hopcroft-Ullman [1] (the simplest). The incremental minimization algorithm [41] is a bit slower for that purpose, and so is Brzozowski algorithm [5], which can be used on nondeterministic automata. However, our automaton is a trie, so we can use an algorithm that uses that fact.

Recall that two states are equivalent when they have the same language (see Equation (2.9)). We compare states of a trie pairwise and replace one with the other one when they are equiv-

alent. Replacement means that one of the states is deleted, and all transitions that lead to it are redirected to the other state. Instead of using (2.3), we use the recursive definition (2.6). Two states p and q are equivalent when they:

1. are both final or non-final
2. have the same number of outgoing transitions
3. have the same labels on outgoing transitions
4. targets of transitions with the same labels have the same right language

Formally (we compressed points 2 and 3 into a single condition):

$$p \equiv q \Leftrightarrow \begin{cases} (p \in F \wedge q \in F) \vee (p \notin F \wedge q \notin F) \\ \Sigma_p = \Sigma_q \\ (\forall \sigma \in \Sigma_p)\ \overrightarrow{\mathcal{L}}\ (\delta(p,\sigma)) = \overrightarrow{\mathcal{L}}\ (\delta(q,\sigma)) \end{cases} \tag{3.1}$$

Contrary to the incremental minimization algorithm [41], we do not initially divide states of an automaton into classes based on their finality, number of outgoing transitions, their labels, etc. However, we do divide them into two groups: those that have already been checked, and those that have not. Of course, at the beginning, the first group is empty. Each subsequent state from the unchecked group is checked for equivalence against all states in the checked group. It is possible to visit the states in the unchecked group in such an order that the targets of all their outgoing transitions have already been checked. Notice that that condition also holds for any state from the checked group. One such order is **postorder**. To perform operation P on state q and on all states reachable from state q, we use Algorithm 3.4.

Algorithm 3.4 Postorder method of performing operation P on state q of an acyclic automaton M, and on all states reachable from q.

```
1: procedure postorder(M, q)
2:     for σ ∈ Σ_q do
3:         postorder(M, δ(q, σ))
4:     end for
5:     P(q)
6: end procedure
```

One advantage of using postorder is that Equation (3.1) can be simplified to:

$$p \equiv q \Leftrightarrow \begin{cases} (p \in F \wedge q \in F) \vee (p \notin F \wedge q \notin F) \\ \Sigma_p = \Sigma_q \\ (\forall \sigma \in \Sigma_p)\delta(p,\sigma) = \delta(q,\sigma) \end{cases} \tag{3.2}$$

or even more compactly to:

$$p \equiv q \Leftrightarrow sgnt(p) = sgnt(q) \tag{3.3}$$

In our case, the operation P consists of comparison of a state q with states in the already checked (minimized) part, and possibly a replacement of an equivalent state. The division

into the checked and unchecked part is implemented so that we keep track only of the states in the checked part. They are stored in a sparse table called the **register**. Actually, we do not need to store whole states there; pointers to states or their indexes are enough. The sparse table uses a hash function that is computed on basis of signatures. In that way, finding an equivalent state is fast. This is done in constant time on average. Also storing a state in a register is done in constant time on average.

Algorithm 3.5 Minimization of a part of an acyclic M starting from state q. The register R is a global variable. M is modified by the function. Either q or its equivalent state is returned.To minimize the whole automaton, set R to $\emptyset$, and call Minimize with q_0 as the second argument.

```
 1: function minimize(M, q)
 2:     for σ ∈ Σ_q do
 3:         δ(q, σ) ← minimize(M, δ(q, σ))
 4:     end for
 5:     if (∃r ∈ R)r ≡ q then
 6:         delete q
 7:     else
 8:         R ← R ∪ {q}
 9:         r ← q
10:     end if
11:     return r
12: end function
```

The minimization algorithm is presented as Algorithm 3.5. In the `for` loop, the function is called recursively. The assignment statement either keeps the old state as the target of a transition or sets an equivalent state as that target depending on the outcome of an `if` statement in the recursive call. The `if` statement searches for an equivalent state. If one is found, then the current state is deleted, and the equivalent state is returned, and a transition that leads to the current state is redirected towards the equivalent state in the invocation of minimize one level up. If an equivalent state is not found, then the current state has unique right language among all states in the visited part of the automaton, hence the state is added to the register.

Let us run the algorithm on the trie automaton in Figure 3.1. minimize is called with 0 as the second argument, then 1, then 2, and then 3. Since there are no outgoing transitions in state 3, the `for` loop is not run. The register is empty, so the conditional statement adds state 3 to the register, and state 3 is returned. Control is transferred one level up to the call with state 2 as the second argument, then minimize is called with state 4, and with state 7. The register contains state 3, and state 7 is equivalent to state 3. State 7 is deleted, and the function returns state 3. The assignment statement in line 3 redirects the transition from state 4 leading previously to state 7 to the new target: state 3. The situation is shown in Figure 3.6.

Next, minimization is called with state 8, then state 17, and state 32. State 32 is equivalent to state 3. State 3 is deleted, and the transition to state 32 is redirected towards state 3. The register still contains only state 3. State 17 is not equivalent to state 3, so it is added to the register, and so is state 8. The transitions that lead to them are not redirected. This is shown in Figure 3.7.

Minimization proceeds to states 9, 18, 33, and 34. State 34 is equivalent to state 3, so it is deleted. Now non-final state 33 has one transition labeled with *e* leading to state 3, exactly

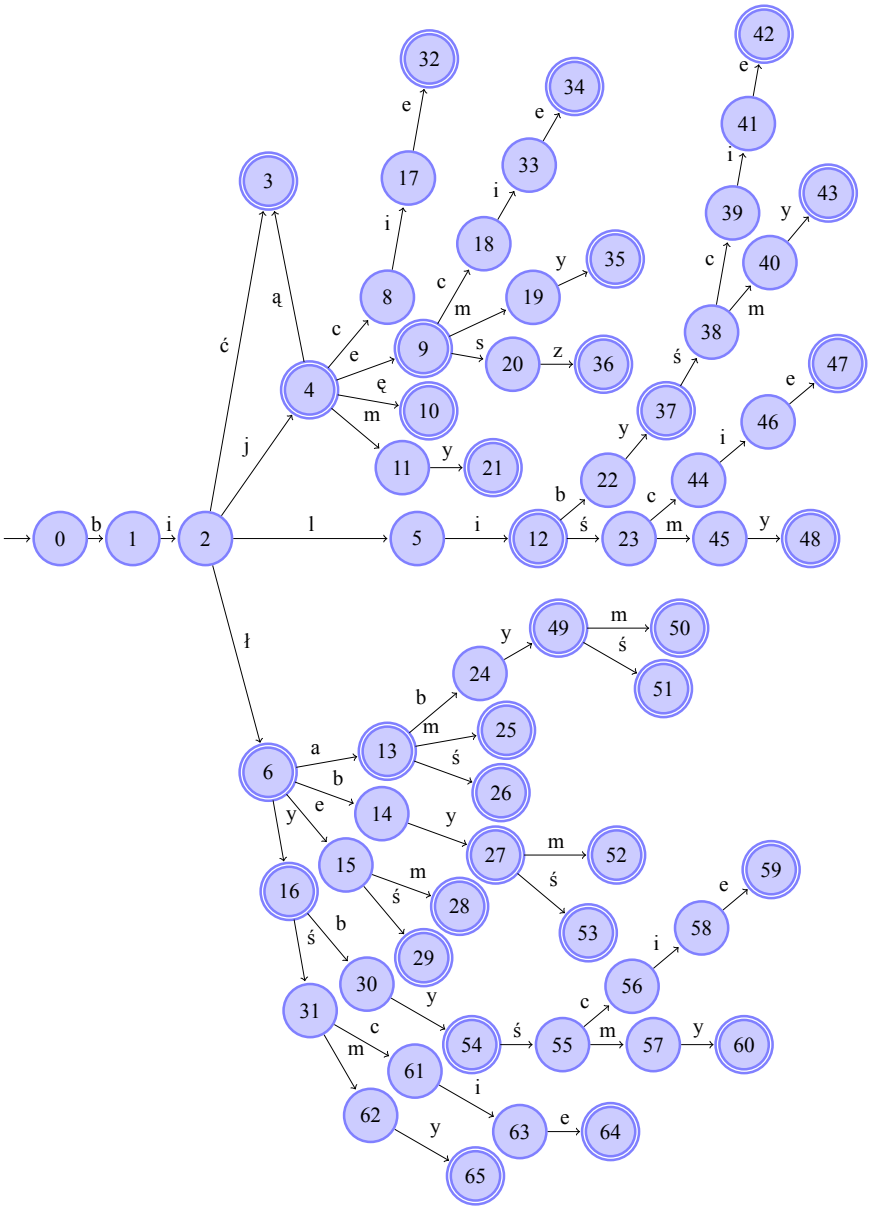


Figure 3.6: Minimization of the trie automaton in Figure 3.1. Minimization reached state 7. The register contains state 3.

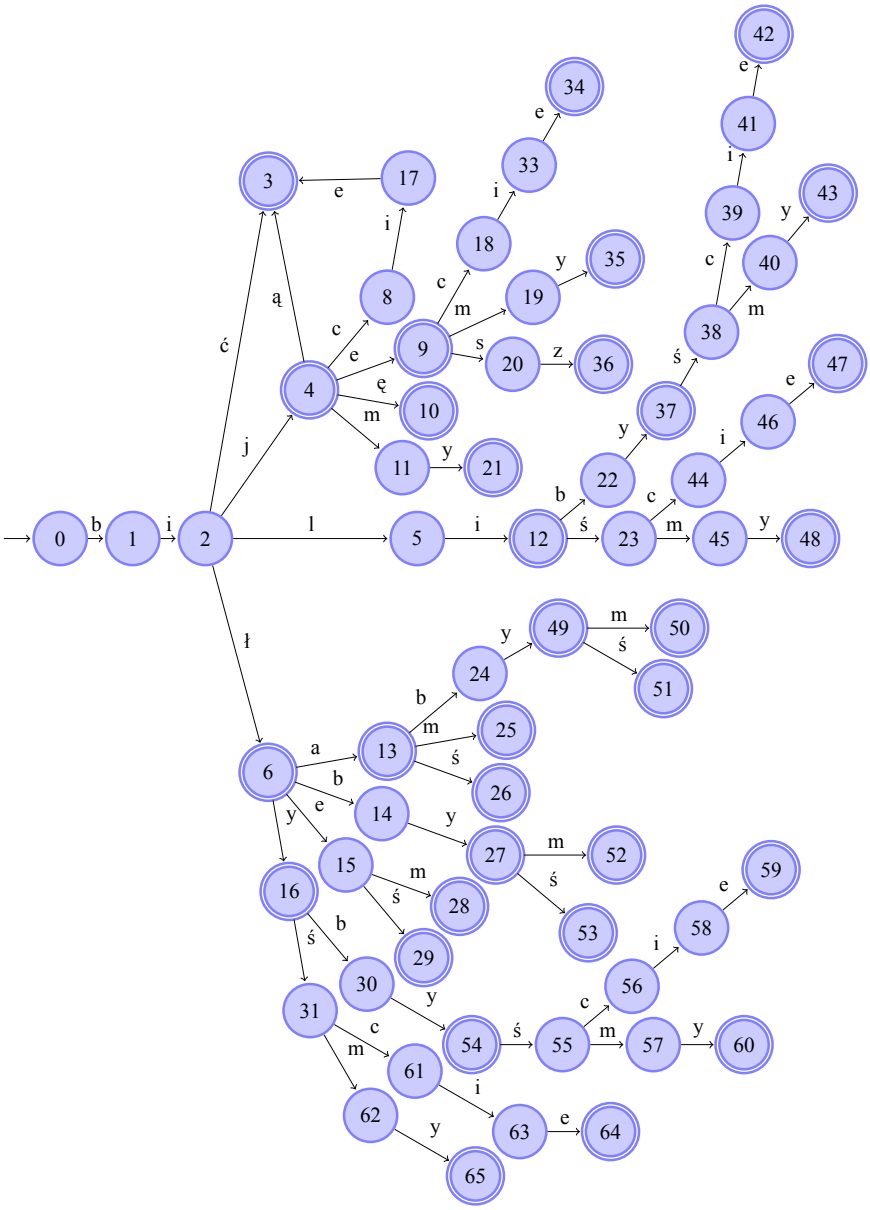


Figure 3.7: Minimization of the trie automaton in Figure 3.1. Minimization reached state 8. The register contains states 3, 17, and 8.

like state 17, so state 33 is replaced with state 17. Also state 18 is found equivalent to state 8, since both are non-final and both have a transition labeled with *i* leading to state 17. State 18 is deleted, and the transition to state 18 is redirected towards state 8, as shown in Figure 3.8.

Algorithm 3.5 minimizes the whole automaton in the form of a trie since it visits all states of the automaton, each time checking whether there is an equivalent state in the already analyzed part of the automaton. Since at the end all states have unique right languages, and there are no unreachable states, and replacement of redundant states with equivalent states does not change the language of the automaton, the automaton is minimal.

By comparing Figures 3.1 and 3.9, one can easily see that the minimal automaton is much smaller. Indeed, minimal automata for natural language dictionaries are much smaller than tries. Consider for example an automaton recognizing not only inflectional forms of the verb *bić*, but also those of the verb *pić*. The trie would have twice as many states but one, as only the start state would be shared between the two verbs. It would also have twice as many transitions as the trie in Figure 3.1. On the other hand, the minimal automaton would have exactly many states as the minimal automaton recognizing only the forms of the verb *bić* (as in Figure 3.9), and it would have only one more transition: from state 0 to state 1 labeled with *p*. The biggest problem with the traditional way of constructing a minimal acyclic automaton is the size of the intermediate trie. The solution is reduction of the size of the automaton during construction so that the size of the automaton during that process is close to the minimal one.

To keep the size of the automaton close to minimal during construction, one minimizes the automaton after each string has been added. However, doing that in a naive way would result in very long construction times as the same states would be reprocessed again and again, and much information acquired during previous runs of minimization would be lost. Therefore, minimization has to be *local*. Information between runs of the construction process and runs of the minimization process should be shared.

It is also possible to use specific information about input data. One such information is ordering. Ordered data is usually easier to process. The first incremental algorithm to be presented here uses data that lexicographically (alphabetically) sorted.

3.2 Incremental Construction of Minimal, Acyclic DFAs from sorted data

In the algorithm of incremental construction of minimal, acyclic automata from sorted strings, we have to synchronize two processes:

1. a trie-building process (Algorithm 3.1 and Algorithm 3.3),
2. a trie minimization process (Algorithm 3.5).

The two processes should work in turns, handing over control one to another in appropriate places. Local minimization should be performed only on those parts of the automaton that will not change when new strings are added.

Let us look at Figure 3.1 (page 22) again to see how the trie is constructed when strings come in sorted. Suppose that the word added to the trie was *bijmy*. It is recognized along the path of states 0, 1, 2, 4, 11, and 21. All words that come lexicographically earlier share some initial segment of that path (that segment can also be null in general case), and then their paths

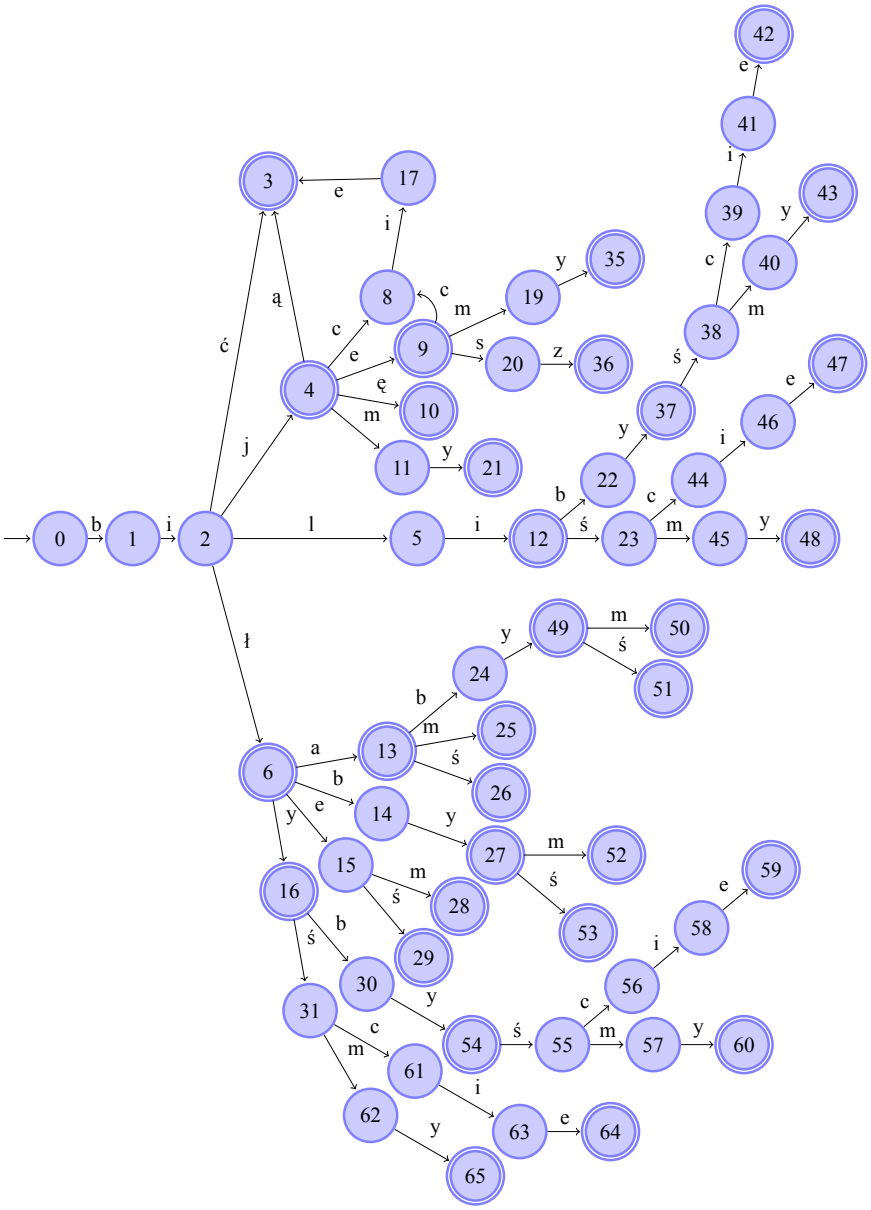


Figure 3.8: Minimization of the trie automaton in Figure 3.1. Minimization reached state 18. The register contains states 3, 17, and 8.

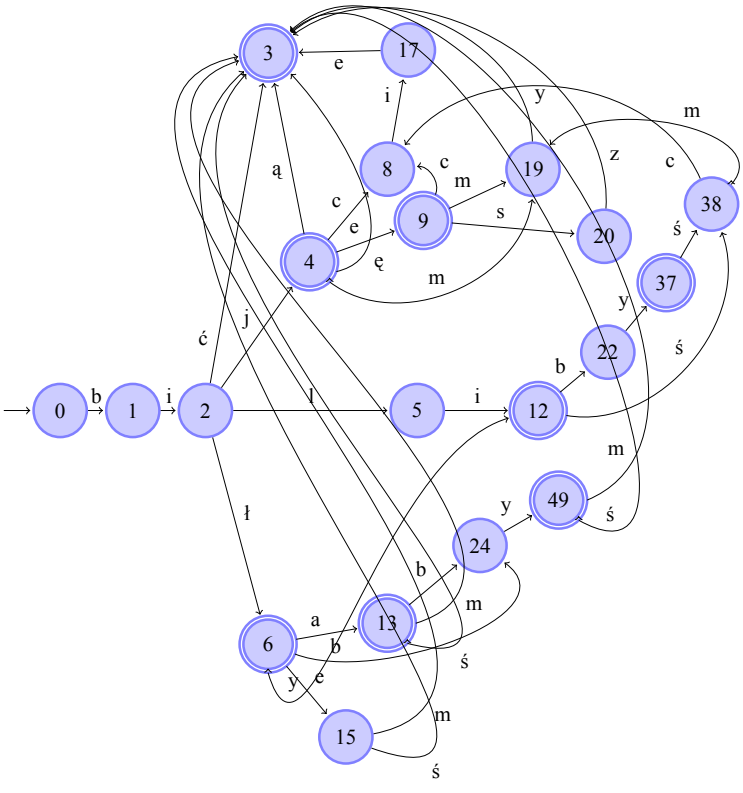


Figure 3.9: Minimal automaton recognizing the same language as the trie automaton in Figure 3.1. The minimal automaton has only 20 states, as opposed to 65 in the trie. In real natural language dictionaries, savings are much greater.

branch out above the path of *bijmy*. All words that are added afterward also share some initial segment (which can be null or cover the whole word), and then branch out below the path of *bijemy*. Let us see what happens when the next word is added. The next word is *bili*. Now the states that will not change also include states 4, 11, and 21, because they no longer lie on the path recognizing the word added as the last. Each time a new word is added, states that lie on the path of the previously added word but do not lie on the path of the last added word join the part that will not be subject of any future modifications. Sometimes that part will not be augmented. For example, when we add the next word: *bilibyście*, all states that lie on the path of the previously added word lie also on the path of the last added word. This happens because *bili* is a prefix of *bilibyście*.

Local minimization does change states. However, states undergo it *before* they enter the analyzed part. Afterward, they can only get more incoming transitions, as they may become targets of redirected transitions as equivalent states.

Algorithm 3.6 Incremental construction algorithm for sorted data.

```
 1: function SortedIncrementalConstruction
 2:     Create empty automaton M = ({q_0}, Σ, ∅, q_0, ∅)
 3:     R ← ∅                                          ▷ set empty register
 4:     w' ← ε                                         ▷ previous word
 5:     while input not empty do
 6:         w ← next word
 7:         AddStrSorted(M, w, w')
 8:         w' ← w
 9:     end while
10:     ReplOrReg(M, q_0, w')
11:     return M
12: end function
```

Function SortedIncrementalConstruction presented as Algorithm 3.6 creates an empty automaton. Then, for each input string, it calls AddStrSorted that does the main job of adding a string to the language of an automaton while maintaining its (almost) minimality. Finally, function ReplOrReg is called to perform minimization of the path recognizing the last word added. Recall that the part that is subject to local minimization is the path of the previously added word, that does not share states with the path of the last added word. When we add the last word from the input data, its whole path requires minimization. Function ReplOrReg is described later (page 32).

Function AddStrSorted given as Algorithm 3.7 has two `while` loops. In the first one, subsequent symbols of the input string are matched against labels of outgoing transitions of the current state q of the automaton starting from the initial state q_0. If an appropriate transition is found, it is traversed changing the current state. The loop will not run if either the automaton is empty (we are adding the first string), or when the first symbol of the current word is different from the first symbol of the previous word (and of any other word already added).

In the first `while` loop, we were traversing transitions of the previously added word. The conditional instruction checks whether there are any transitions left in the path of the previously added word. If there are any, that part of the path is minimized in a call to ReplOrReg.

Algorithm 3.7 Procedure AddStrSorted adds a string w to the language of an acyclic automaton M. Strings must come sorted.

```
1:  procedure AddStrSorted(M, w, w')
2:      q ← q_0
3:      i ← 1
4:      while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
5:          q ← δ(q, w_i)
6:          i ← i + 1
7:      end while
8:      if i ≤ |w'| then
9:          δ(q, w'_i) ← ReplOrReg(M, δ(q, w'_i), w'_{i+1...|w'|})
10:     end if
11:     while i ≤ |w| do
12:         δ(q, w_i) ← new state
13:         q ← δ(q, w_i)
14:         i ← i + 1
15:     end while
16:     F ← F ∪ {q}
17: end procedure
```

The function returns the first state of the path after minimization. If that state is replaced with an equivalent state, instruction in line 9 redirects a transition labeled with the next symbol of the previous word toward the equivalent state.

The last `while` loop deals with remaining symbols of the current input word. For each symbol, a new state is created, and a new transition with the symbol as the label is created between the current state q and the newly created state. The newly created state becomes the current state. Finally, state q ---the last state in the path--- is made final.

Algorithm 3.8 Function ReplOrReg performs local minimization. M is the automaton, q is the start of the path to be minimized, w is a sequence of labels that chooses a path starting from q. The function returns either state q or a state equivalent to it if found. The register R is a global variable.

```
1:  function ReplOrReg(M, q, w)
2:      if w ≠ ε then
3:          δ(q, w_1) ← ReplOrReg(M, δ(q, w_1), w_{2...|w|})
4:      end if
5:      if (∃r ∈ R)r ≡ q then
6:          delete q
7:          return r
8:      else
9:          R ← R ∪ {q}
10:         return q
11:     end if
12: end function
```

Function ReplOrReg is presented as Algorithm 3.8. It performs local minimization of a path of states in the automaton M starting from state q, and with labels on subsequent transitions being subsequent symbols in string w. The first conditional instruction checks whether the end of the path is reached. If it is not so, the function recursively calls itself with the next state on the path, i.e. $\delta(q, w_1)$, and with the remaining part of the string after removal of the first symbol. It keeps calling itself until the end of the path is reached. In that case, w is empty.

The second conditional instruction checks whether there exists a state r in the already minimized part of the automaton that is equivalent to state q. If it is so, state q is deleted, and state r is returned. When the call is returned, the assignment instruction in line 3 performs redirection of the transition that led to state q to state r. When no equivalent state in the register R exists, state q is added to the register R as q has a unique right language. Assignment instruction in line 3 does not perform any redirection as q is the original target of the transition that could be redirected there.

Let us see how the algorithm works on sample data. Our input data is the language of the trie automaton shown in Figure 3.1 (page 22) or of the minimal automaton shown in Figure 3.9 (page 30). Recall that the data is sorted lexicographically, so the words come in the following order: *bić, bij, biją, bijcie, bije, bijecie, bijemy, bijesz, biję, bijmy, bili, biliby, bilibyście, bilibyśmy, biliście, biliśmy, bił, biła, biłaby, biłabym, biłabyś, biłam, biłaś, biłby, biłbym, biłbyś, biłem, biłeś, biły, biłyby, biłybyście, biłybyśmy, biłyście, biłyśmy*. Function SortedIncrementalConstruction first creates an empty automaton with only an initial state. It also sets a previous word to an empty string ε. Subsequent `while` loop reads and processes words from input.

The first word is *bić*. It is read from the input and assigned to variable w. Function AddStrSorted is called with the automaton, *bić*, and ε as arguments. Current state q is set to the initial state q_0, and the current index of a symbol in the word to be added is set to 1. Since there are no transitions in the automaton, the first `while` loop does not run. Since before the loop w' was set to an empty string ε, and i to one, the conditional instruction is not executed either. The second `while` loop creates a chain of states and transitions so that the automaton recognizes *bić*. The automaton is shown in Figure 3.10.

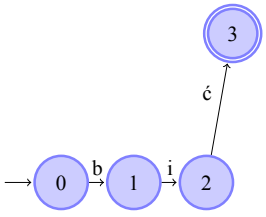


Figure 3.10: Incremental construction from sorted data. Input data is the language of an automaton shown in Figure 3.1. Procedure AddStrSorted(M,*bić*,ε) has just finished adding the word.

The next word to be added is *bij*. After AddStrSorted was called with *bić* as the second argument, w' (the previous word variable) is set to *bić*, and word *bij* is read from the input and set as value of variable w. Procedure AddStrSorted is called with *bij* and *bić* as the second and the third arguments. The first `while` loop is executed. As there is a transition from 0 to

1 labeled with *b*, it is traversed, and q becomes state 1. Next, transition labeled *i* is followed so that q becomes state 2. There is no transition labeled with *j* coming out from state 2. The `while` loop terminates. Variable i (an index of the current symbol in word w) is now 3. The condition $i \leq |w'|$ is met, and ReplOrReg($M, 3, \varepsilon$) is invoked. The second argument is 3 since $\delta(2, w'_3 = ć)$ is state 3. The third argument is ε as there are no symbols with indexes 4 and greater in w'.

Inside ReplOrReg, the recursive call in the first conditional instruction cannot be made as w is empty. Since the register R is empty, the condition of the second conditional instruction is false. State 3 is inserted into the register, and the function returns state 3.

Back in the procedure AddStrSorted, the assignment in line 9 does nothing (it does not change the target of the transition $\delta(2, ć)$) as state 3 was the target before the call to ReplOrReg. Variable i is still 3 and $|w|$ (the length of *bij*) is 3, so the second `while` loop is entered. A new state 4 is created, and it becomes the target of a transition leaving state 2 with a label *j*. State 4 is the new value of variable q. Variable i is incremented so that it becomes greater than the length of *bij*. The loop is terminated. State 4 becomes final. That moment is shown in Figure 3.11. The register R contains state 3.

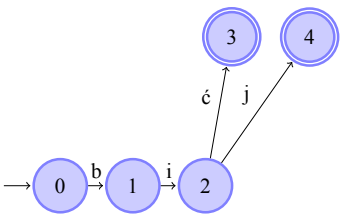


Figure 3.11: Incremental construction from sorted data. Input data is the language of an automaton shown in Figure 3.1. Procedure AddStrSorted(M,*bij*,*bić*) has just finished adding the word *bij*.

The current word *bij* becomes the previous word w'. The next word on the input is *bijq*. It is read into variable w. Procedure AddStrSorted is called with *bijq* and *bij* as the second and the third parameter. The current state q is set to state 0, the symbol index i is set to 1. The first `while` loop is entered. It is run 3 times so that q is set to 4, and i is set to 4. The condition $i \leq |w'|$ is not met. It means that there is no local minimization in this run of AddStrSorted. This happens because *bij* is a prefix of *bijq*. The second `while` loop is entered. A new state 7 (we keep the numbering of states as in Figure 3.1) is created, and it becomes the current state q. Also, variable i is incremented so that the loop condition is no longer met. State 7 becomes final as shown in Figure 3.12. The register R contains state 3.

The current word *bijq* becomes the previous word w', and w is set to the next word on the input --- *bijcie*. Procedure AddStrSorted(M, *bijcie*, *bijq*) is invoked. The first `while` loop runs until q becomes state 4, and i becomes 4. Since $|w'| = 4$, function ReplOrReg(M, 7, ε) is called. Inside ReplOrReg, recursive calls are skipped. State 3 from the register is found to be equivalent to state 7, so state 7 is deleted, and state 3 is returned by the function. In line 9 of AddStrSorted, the transition from state 4 to state 7 with label *q* is redirected towards state 3. The second `while` loop is entered. State 8 is created, and a transition from state 4 to state 8 labeled with *c*. Next, state 17 and a transition from state 8 to state 17 labeled with *i*. In the last run of the loop, state 32 and a transition from state 17 to state 32 labeled with *e* are

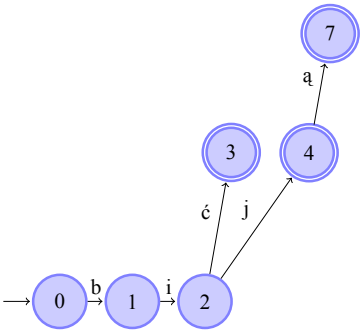


Figure 3.12: Incremental construction from sorted data. Input data is the language of an automaton shown in Figure 3.1. Procedure AddStrSorted(M,*bijq*,*bij*) has just finished adding the word *bijq*.

called to existence. Finally, the loop terminates, and state 32 is made final, which is shown in Figure 3.13. The register contains state 3. Note that all other states are on the path of the last added word.

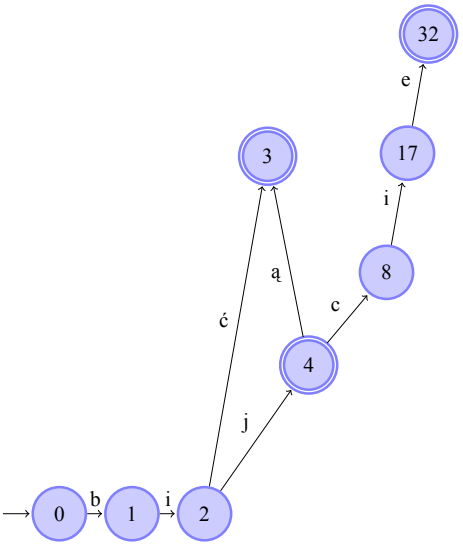


Figure 3.13: Incremental construction from sorted data. Input data is the language of an automaton shown in Figure 3.1. Procedure AddStrSorted(M,*bijcie*,*bijq*) has just finished adding the word *bijcie*.

In SortedIncrementalConstruction, after w' is set to *bijcie*, w is set to *bije* --- the next word on the input. An invocation of AddStrSorted(M, *bije*, *bijcie*) follows. The first `while` loop runs so that q is set to state 4, and i is set to 4. Function ReplOrReg(M, 8, *ie*) is called.

Inside *ReplOrReg*, the function calls itself as ReplOrReg(M, 17, *e*) in line 3 as $w = ie \neq \varepsilon$. In that recursive call w is still not empty, so another call to ReplOrReg(M, 32, ε) is made. There, finally, $w = \varepsilon$, so another recursive call is blocked. State 3 is found equivalent to state 32, so state 32 is deleted, and state 3 is returned. It becomes the target of the transition from state 17 labeled with *e* in line 3 of the call to function ReplOrReg(M, 17, *e*). As $\vec{\mathcal{L}}(17) = \{e\}$, and the register contains only state 3 with the right language equal to ε, the right language of state 17 is unique, so state 17 is inserted into the register, and it is returned by the function. One level up, in textscReplOrReg(M, 8, *ie*), no redirection is performed in line 3, as state 17 is the original target of the transition from state 8 labeled with *i*. State 8 is also found unique, so it is added to the register and returned by the function.

In function AddStrSorted, in line 9, no redirection is done as state 8 is unique. The second `while` loop is entered. State 9 and a transition from state 4 to state 9 are created labeled with *e* is created. Finally, state 9 is made final. The situation is presented in Figure 3.14. The register contains states 3, 8, and 17.

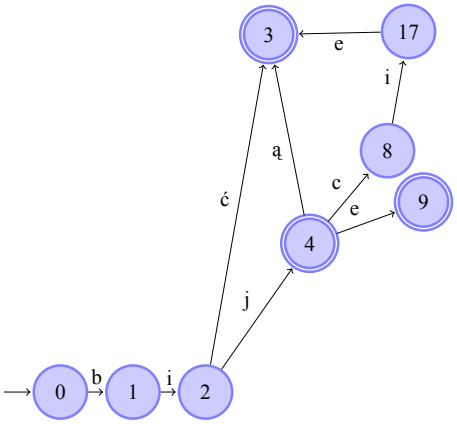


Figure 3.14: Incremental construction from sorted data. Input data is the language of an automaton shown in Figure 3.1. Procedure AddStrSorted(M,*bije*,*bijcie*) has just finished adding the word *bije*.

In SortedIncrementalConstruction, w' is set to *bije*, and w to the next word from the input --- *bijecie*. A call to AddStrSorted(M, *bijecie*, *bij*) is made. The first `while` loop leaves $q = 9$, and $i = 5$. The conditional instruction does not execute as *bije* is a prefix of *bijecie*, so function ReplOrReg is not called. The second `while` loop creates states 18, 33, and 34, as well as transitions from state 9 to state 18 labeled with *c*, from state 18 to state 33 labeled with *i*, and from state 33 to state 34 labeled with *e*. State 34 is made final as shown in Figure 3.15. The register contains states 3, 8, and 17.

The next word on the input is *bijemy*, so AddStrSorted(M, *bijemy*, *bijecie*) is called. After the first `while` loop, $q = 9$, and $i = 5$. A call to function ReplOrReg(M, 18, *ie*) is made, which results in a call to ReplOrReg(M, 33, *e*), which in turns calls ReplOrReg(M, 34, ε). Since $w = \varepsilon$, recursive calls are blocked. State 34 is found equivalent to state 3 from the register, so state 34 is deleted, and state 3 is returned. In line 3 of ReplOrReg(M, 33, *e*), a transition labeled with *e* is redirected towards state 3. Now a non-final state 33 has a single

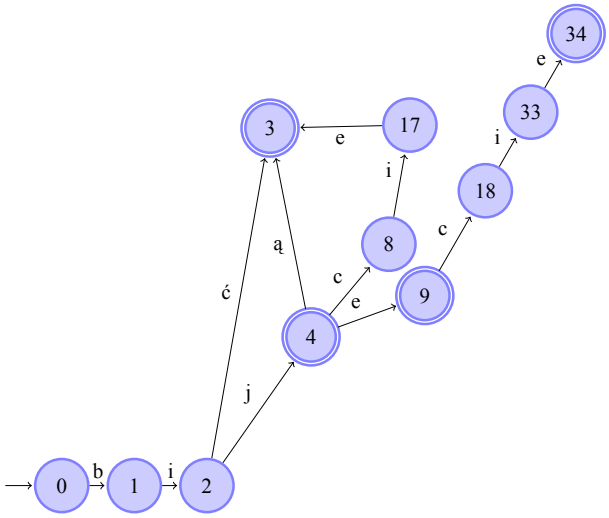


Figure 3.15: Incremental construction from sorted data. Input data is the language of an automaton shown in Figure 3.1. Procedure AddStrSorted(M,*bijecie*,*bije*) has just finished adding the word *bijecie*.

transition labeled with *e* leading to state 3. So is state 17 in the register. State 33 is deleted, and state 17 is returned so that the transition from state 18 labeled with *i* is redirected towards it in line 3 of ReplOrReg(M, 18, *ie*). Also state 18 is found equivalent to state 8, so state 18 is deleted, and state 8 is returned. Back in AddStrSorted(M, *bijemy*, *bijecie*), in line 9, a transition from state 9 to state 18 labeled with *c* is redirected towards state 8. The second `while` loop creates states 19 and 35, as well as transitions from state 9 to state 19 labeled with *m*, and from state 19 to state 35 labeled with *y*. State 35 is made final. The situation can be seen in Figure 3.16. The register contains states 3, 8, and 17.

We could continue describing how further words are added, but the volume of this book is limited, and we have shown all use cases of the procedure AddStrSorted. We leave adding the words as an exercise to the reader. However, after the last call to AddStrSorted, the automaton is not minimal, as shown in Figure 3.17. All states except for 0, 1, 2, 6, 16, 31, 62, and 65 are in the register.

So when the last call to AddStrSorted is finished, w' is set to *biłyśmy*, there are no more words on the input, the `while` loop terminates, and ReplOrReg(M, 0, *biłyśmy*) is called. Since the third argument is not an empty string, a series of recursive calls of ReplOrReg follows: ReplOrReg(M, 1, *iłyśmy*), ReplOrReg(M, 2, *łyśmy*), ReplOrReg(M, 6, *yśmy*), ReplOrReg(M, 16, *śmy*), ReplOrReg(M, 31, *my*), ReplOrReg(M, 62, *y*), and ReplOrReg(M, 65, ε). In that last call, further recursive calls are blocked since $w = \varepsilon$. State 65 is equivalent to state 3, so state 65 is deleted and ReplOrReg(M, 65, ε) returns state 3, which becomes the target of the transition $\delta(62, y)$. State 62 is equivalent to state 19, so state 62 is deleted, and ReplOrReg(M, 62, *y*) returns state 19, which becomes the target of the transition $\delta(31, m)$. State 31 is equivalent to state 38, so state 31 is deleted, and ReplOrReg(M, 31, *my*) returns state 38, which becomes the target of the transition $\delta(16, ś)$. State 16 is in turn equivalent to state 12,

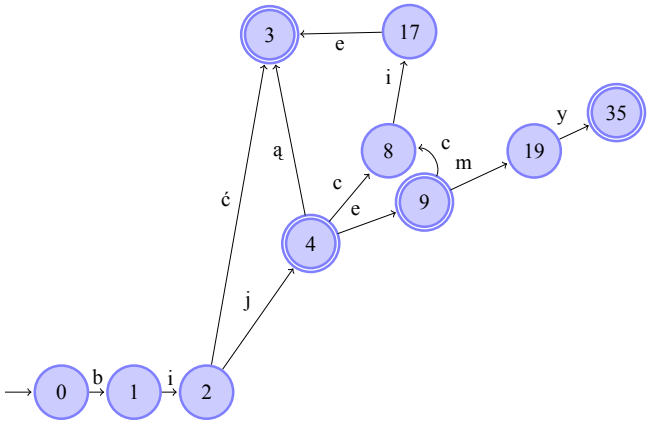


Figure 3.16: Incremental construction from sorted data. Input data is the language of an automaton shown in Figure 3.1. Procedure AddStrSorted(M,*bijemy*,*bijecie*) has just finished adding the word *bijemy*.

so state 16 is deleted, and ReplOrReg(M, 16, *śmy*) returns state 12, which becomes the target of $\delta(6, y)$. The remaining calls to ReplOrReg find that their second arguments have unique right languages, so they are added to the register, no further modifications to the structure of the automaton are made. The result can be seen in Figure 3.9 on page 30.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an automaton before function AddStrSorted is called. Let $M' = (Q', \Sigma, \delta', q_0', F')$ be the automaton after the function is called with word w and the previous word w'. The function creates confluence state only in the part of the automaton where no new outgoing transitions are to be created. In the first `while` loop of the function, the longest common prefix of w and w' is found and followed in the automaton. Confluence states can be created in function ReplOrReg by replacing states that recognize the remaining part of w' with equivalent states. Since the states are in the remaining part of w', and new words coming in lexicographical order can create outgoing transitions only from any prefix of the longest common prefix, the confluence states will never be modified. No future longest common prefix can go through them. Since the longest common prefix cannot go through confluence states, the first `while` loop goes through it, and the last `while` loop creates a chain of new states (the last one is made final) and transitions recognizing the suffix of w, we have $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$. No words are deleted as transitions are redirected only to equivalent states, and states and transitions are deleted only when they become unreachable, i.e. when they no longer contribute to the language of the automaton.

The function creates no useless states, i.e. no states that are either not reachable or not co-reachable. Function ReplOrReg redirects transitions so that states can temporarily become unreachable. When this happens, or more precisely *before* this happens, the states are deleted. Since transitions are redirected to equivalent states that have exactly the same suite of transitions, the deletion cannot create unreachable states. There are no states that are not co-reachable, as the new chains of states and transitions created in the last `while` loop in function AddStrSorted always end with a final state, and function ReplOrReg redirects transitions only to equivalent states.

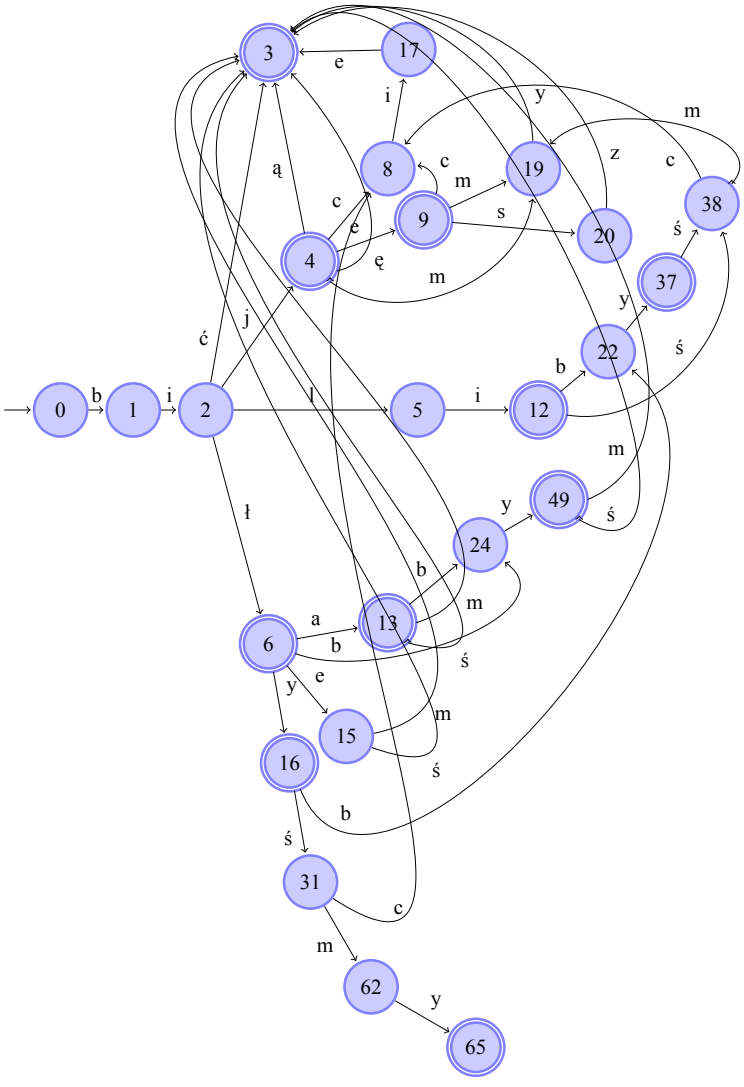


Figure 3.17: Incremental construction from sorted data. Input data is the language of an automaton shown in Figure 3.1. Procedure AddStrSorted(*M*,*biłyśmy*,*biłyście*) has just finished adding the word *biłyśmy*.

Apart from the states along the path recognizing the last word added, all states do not have an equivalent state outside the path. Indeed, the states in the path recognizing a word are divided into two parts: those belonging to the longest common prefix path, and the rest. The rest is checked using function ReplOrReg while visiting states in postorder, so that if an equivalent state for them exists, they are replaced with that state. The states of the longest common prefix path belong to the next word, so they are subject to the same division later on.

Function SortedIncrementalConstruction returns the minimal deterministic automaton recognizing all words read from the input. Indeed, since function AddStrSorted adds a word to the language of the automaton, the only states not yet minimized after that addition are those that lie along the path recognizing the last word added. Precisely that path is processed in the last call to ReplOrReg, thus the automaton is minimal.

The `while` loop in the function SortedIncrementalConstruction runs n times when there are n words or strings on the input. Reading a string w takes $|w|$ time. Copying a string can usually be done in constant time by copying a pointer. Procedure AddStrSorted runs n times. Two `while` loops inside that procedure run jointly $|w|$ times. Traversing a transition is done in constant time. Creating a new state is done in constant time. Making state final is done in constant time as it is implemented by setting a flag associated with the state. Function ReplOrReg is called at most once for each invocation of AddStrSorted. The former calls itself $|w|$ times. After a possible recursive call, a search for an equivalent state is performed with complexity $r(n)$. Deletion of a state takes constant time. Adding a state to the register takes $r(n)$ time. It follows that a top-level call to ReplOrReg including all recursive calls takes $\mathcal{O}(|w_{max}|r(n))$ time, where w_{max} is the longest word on the input. Similarly, a call to AddStrSorted is executed in $\mathcal{O}(|w_{max}|r(n))$ time. Finally, the time complexity of SortedIncrementalConstruction is $\mathcal{O}(n|w_{max}|r(n))$. In practice, operations on the register take constant time, so we assume $r(n) = 1$, and the complexity of SortedIncrementalConstruction is $\mathcal{O}(n|w_{max}|)$.

The incremental construction algorithm for sorted data was independently invented by Jan Daciuk [18, 21, 12], and Stoyan Mihov [30, 18], and then reinvented by Ciura and Deorowicz [10]. The algorithm has several extensions.

3.3 Extension to Pseudo-Minimal Automata

The incremental construction algorithm for sorted data can easily be adapted for construction of pseudo-minimal automata. The necessary changes must take into account cardinality of the right language of states. In other words, they must check whether two states to be merged are divergent. The main function PseudoSortedIncrementalConstruction is shown as Algorithm 3.9.

The main differences between PseudoSortedIncrementalConstruction and SortedIncrementalConstruction are the names of procedure and functions called, and an additional argument to PseudoReplOrReg in comparison to ReplOrReg, which has to be initialized.

The most important difference lies between functions ReplOrReg and PseudoReplOrReg. The latter keeps track of divergent states. Parameter d is set to true if the current state q is divergent. It is set to *false* in top-level calls to PseudoReplOrReg in both PseudoSortedIncrementalConstruction and PseudoAddStrSorted. That value is passed down in each recursive call until the end of a path is reached. Function FanOutC returns the number of outgoing

Algorithm 3.9 Algorithm for incremental construction of pseudo-minimal automata from sorted data.

```
1:  function PseudoSortedIncrementalConstruction
2:      Create empty automaton M = ({q_0}, Σ, ∅, q_0, ∅)
3:      R ← ∅                                          ▷ set empty register
4:      w' ← ε                                         ▷ previous word
5:      while input not empty do
6:          w ← next word
7:          PseudoAddStrSorted(M, w, w')
8:          w' ← w
9:      end while
10:     d ← false
11:     PseudoReplOrReg(M, q_0, w', d)
12:     return M;
13: end function
```

Algorithm 3.10 Procedure PseudoAddStrSorted adds a string to the language of an automaton. Strings must come sorted.

```
1:  procedure PseudoAddStrSorted(M, w, w')
2:      q ← q_0
3:      i ← 1
4:      while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
5:          q ← δ(q, w_i)
6:          i ← i + 1
7:      end while
8:      if i ≤ |w'| then
9:          d ← false
10:         δ(q, w'_i) ← PseudoReplOrReg(M, δ(q, w'_i), w'_{i+1...|w'|}, d)
11:     end if
12:     while i ≤ |w| do
13:         δ(q, w_i) ← new state
14:         q ← δ(q, w_i)
15:         i ← i + 1
16:     end while
17:     F ← F ∪ {q}
18: end procedure
```

transitions of a state. Parameter d is set to true if that value is greater than one, or if it is equal to one and the state is final. The parameter is an input/output parameter, so setting its value changes the actual parameter. That way, information of a divergent state propagates upwards in a call hierarchy. Also, the condition for redirecting a transition is modified. An equivalent state replaces state q only if d is false, i.e. when q is not divergent. Note that divergent states are never put into the register by the function.

Algorithm 3.11 Function PseudoReplOrReg performs local minimization. M is the automaton, q is the start of the path to be minimized, w is a sequence of labels that chooses a path starting from q. Parameter d is an input/output parameter, i.e. its value is transferred to the surrounding call. The function returns either state q or a state equivalent to it if found. The register R is a global variable.

```
 1: function PseudoReplOrReg(M, q, w, d)
 2:     if w ≠ ε then
 3:         δ(q, w_1) ← ReplOrReg(M, δ(q, w_1), w_{2...|w|}, d)
 4:     end if
 5:     if FanOutC(q) + fin(q) > 1 then
 6:         d ← true
 7:     end if
 8:     if d then
 9:         return q
10:     else
11:         if (∃r ∈ R)r ≡ q then
12:             delete q
13:             return r
14:         else
15:             R ← R ∪ {q}
16:             return q
17:         end if
18:     end if
19: end function
```

Let us see how the function PseudoSortedIncrementalConstruction constructs a pseudo-minimal automaton that recognizes the same language as the trie automaton in Figure 3.1 and the minimal automaton in Figure 3.9. The word *bić* is added exactly like in SortedIncrementalConstruction, as in PseudoAddStrSorted the first loop and a call to PseudoReplOrReg are not executed. The word *bij* is also added in the same manner as in SortedIncrementalConstruction. In PseudoAddStrSorted, there is a call to PseudoReplOrReg, but state 3 has no outgoing transitions, so d remains false, and pseudo-equivalence is equivalence.

When we add *bijęˊ*, the first difference occurs. The first `while` loop finds the longest common prefix path ending at state 4, i.e. variable q is set to state 4, and i is set to 4. Variable d is set to false and function PseudoReplOrReg(M, q=9, w=*sz*, d=*false*) is called. This is shown in Figure 3.18. The function calls itself as PseudoReplOrReg(M, q=20, w=*z*, d=*false*), and again as PseudoReplOrReg(M, q=9, $w = \varepsilon$, d=*false*). State 36 has no outgoing transitions, so d remains false. State 3 is equivalent to state 36, so state 36 is deleted, and state 3 is returned, so that in the upper level call, the transition going from state 20 to state 36 is redirected to state

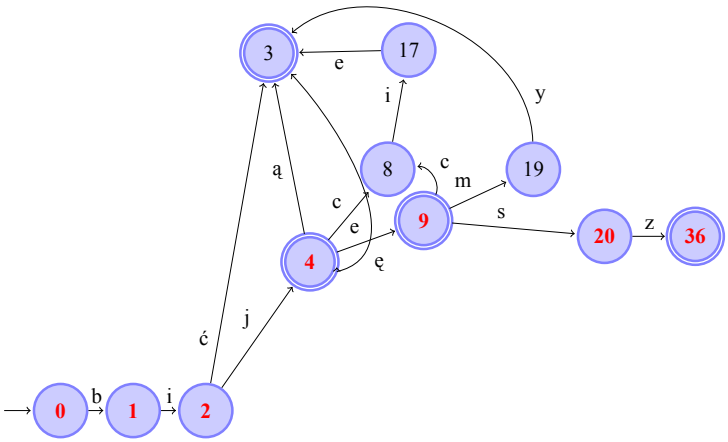


Figure 3.18: Construction of a pseudo-minimal automaton recognizing the same language as the trie automaton in Figure 3.1. Procedure PseudoAddStrSorted(M, *bilibyście*, *bijmy*) has just finished recognizing the longest common prefix *bi*.

3. State 20 has one outgoing transition, and it is not final, so d remains false. State 20 has no equivalent, so it is put into the register, and returned. State 9 has 3 outgoing transitions, and it is also final. Therefore, variable d is set to true. State 9 is returned by the function without registering it. Function ReplOrReg would also return the state, but it would also put it into the register.

Actually, the structure of the automaton remains the same as in the construction of minimal automata until we add *biłabym*, i.e. after we have added *bić, bij, bijcie, bije, bijecie, bijemy, bijesz, biję, bijmy, bilibyście*, *bilibyśmy*, and *biliście*. We add the word *biłabym* to the automaton shown in Figure 3.19.

Function PseudoSortedIncrementalConstruction calls procedure PseudoAddStrSorted(M, *biłabym*, *biliśmy*). After the first `while` loop in that procedure, q is state 2, and i is 3. Function PseudoReplOrReg(M, 5, *iśmy*) is called. A series of recursive calls follows: PseudoReplOrReg(M, 12, *śmy*), PseudoReplOrReg(M, 23, *my*), PseudoReplOrReg(M, 45, *y*), and PseudoReplOrReg(M, 48, ε). In the last call, recursion stops. State 48 has no outgoing transitions, so d remains false. State 48 is found equivalent to state 48, so state 48 is deleted, and PseudoReplOrReg(M, 48, ε) returns state 3 that replaces state 48 as the target of transition $\delta(45, y)$ in line 3 of PseudoReplOrReg(M, 45, *y*). That transition is still the single outgoing transition of state 45, so d remains false. State 45 is equivalent to state 19. Since d is false, state 45 is deleted, and PseudoReplOrReg(M, 45, *y*) returns state 19. In PseudoReplOrReg(M, 23, *my*), state 19 becomes the target of transition $\delta(23, m)$.

At this point in PseudoReplOrReg(M, 23, *my*), d is set to true as state 23 has two outgoing transitions: one labeled with *c* and leading to state 8, another one labeled with *m* and going to state 19. It is also non-final. It is exactly the same situation as with state 38. Function ReplOrReg would delete 23 and return state 38. However, d is true, and function PseudoReplOrReg returns state 23 without adding it to the register.

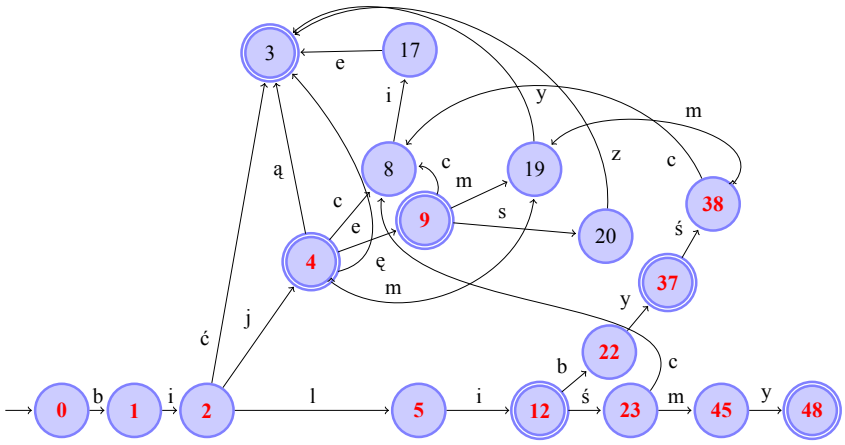


Figure 3.19: Construction of a pseudo-minimal automaton recognizing the same language as the trie automaton in Figure 3.1. Procedure PseudoAddStrSorted(M, *biliśmy*, *biliście*) has just finished adding the word *biliśmy*. Note that there are no states that are divergent and confluence at the same time, and no confluence state follows a divergent state in any path. The automaton is identical to the one obtained in the construction of minimal automata. States 4, 9, 22, 37, and 38 are not in the register as they are divergent.

In line 3 of PseudoReplOrReg(M, 12, *śmy*), transition $\delta(12, ś)$ is not altered; it still points to state 23. Variable d is true as set in the call that has just finished, but it is set to true again anyway, as final state 12 has 2 transitions. There is no state equivalent to state 12 in the register. Even if there were one, it could not be used because of the value of d. State 12 is returned. Transition $\delta(5, i)$ remains unchanged in PseudoReplOrReg(M, 5, *iśmy*), d remains true as passed from the recursive call just ended, so state 5 is returned by the function. Therefore, transition $\delta(2, l)$ is not altered in line 10 of PseudoAddStrSorted. The second `while` loop in PseudoAddStrSorted creates a chain of states and transitions that recognize the rest of the string *biłabym*. The situation is shown in Figure 3.20.

We leave adding the rest of the words as an exercise to the reader. The whole pseudo-minimal automaton recognizing the same language as the minimal automaton in Figure 3.1 and Figure 3.9 is presented in Figure 3.21. Note that the automaton has 28 states instead of 20 states of the minimal automaton. Also note that no confluence state is divergent, and no divergent state is reachable from any confluence state.

Computational complexity of this algorithm is exactly as of the algorithm constructing minimal automata. The additional checking whether a state is divergent can be done in constant time, so can be setting the value of parameter d and evaluating the additional condition for merger of pseudo-equivalent states. Therefore, execution time of this algorithm is linear in the size of its data, i.e. it is $\mathcal{O}(n|w_{max}|)$, where n is the number of words, and w_{max} is the longest string on the input.

This algorithm was developed by Jan Daciuk, Denis Maurel, and Agata Savary in [17]. Contrary to the algorithm developed by Dominique Revuz [31], this algorithm requires data

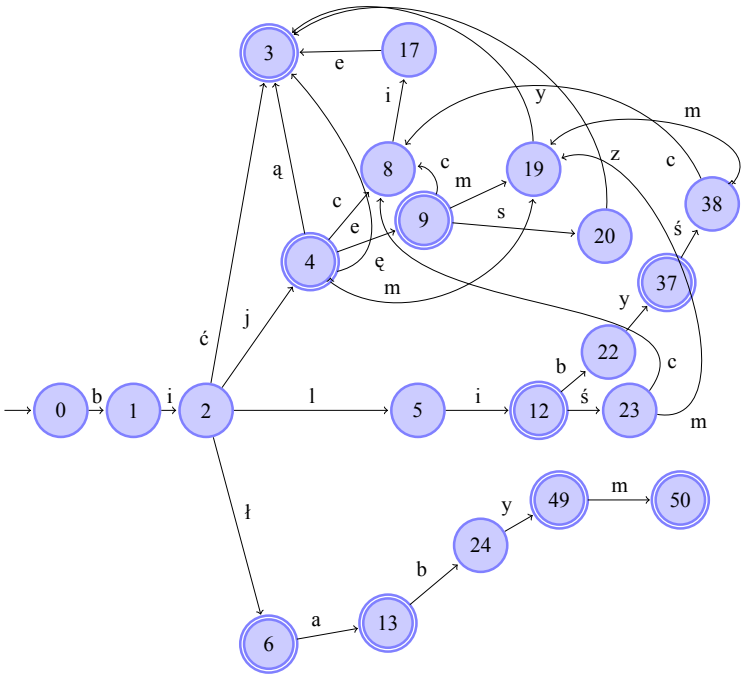


Figure 3.20: Construction of a pseudo-minimal automaton recognizing the same language as the trie automaton in Figure 3.1. Procedure PseudoAddStrSorted(M, *biliśmy*, *biliście*) has just finished adding the word *biliśmy*. Note that there are no states that are divergent and confluence at the same time, and no confluence state follows a divergent state in any path. The automaton is identical to the one obtained in the construction of minimal automata.

to be sorted lexicographically. Revuz'es algorithm requires data to be sorted on reversals of strings.

3.4 Extension to Cyclic Automata

The original algorithm for sorted data (see page 31) can be treated as an algorithm for adding strings to an automaton. After all strings are added, the path for the last word must undergo local minimization.

Suppose that we have created a minimal automaton, and then new data arrives, and new strings should be added to the language of the automaton. Can the algorithm be used to add them? Let us see an example.

Figure 3.22 shows a minimal automaton recognizing the same language as the automaton in Figure 3.9 except for words *biłaby* and *biłby*. Let us add the word *biłaby* using procedure AddStrSorted depicted as Algorithm 3.7 on page 32, with an empty string as the third parameter. The first `while` loop exits with q being state 49, and i equal to 7. The whole word

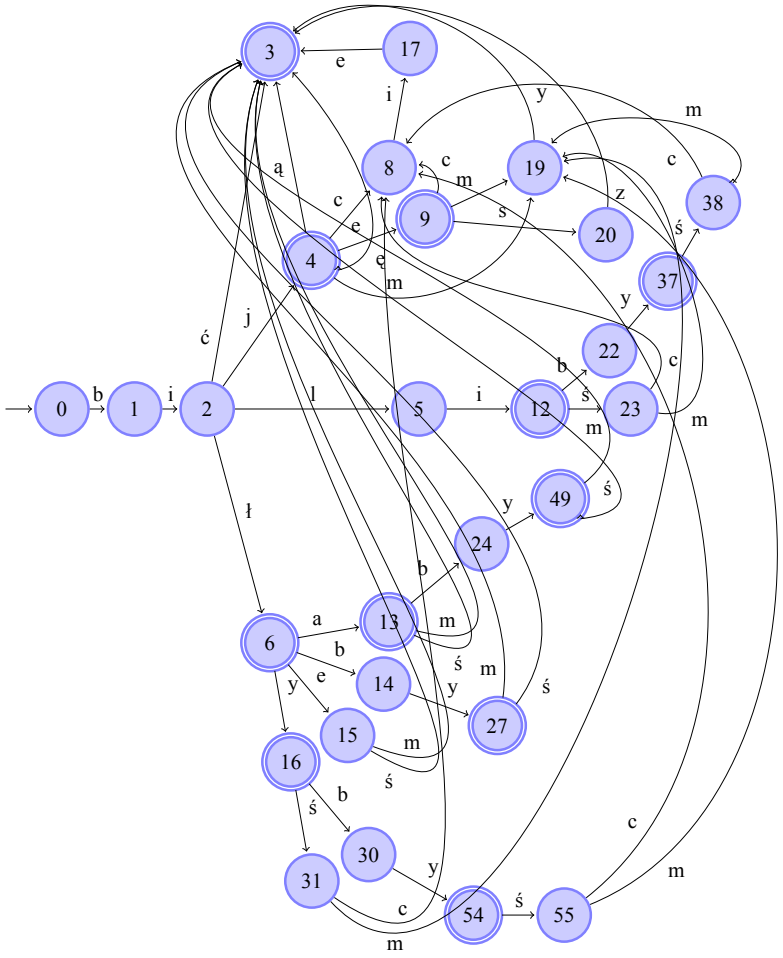


Figure 3.21: A pseudo-minimal automaton recognizing the same language as the trie automaton in Figure 3.1.

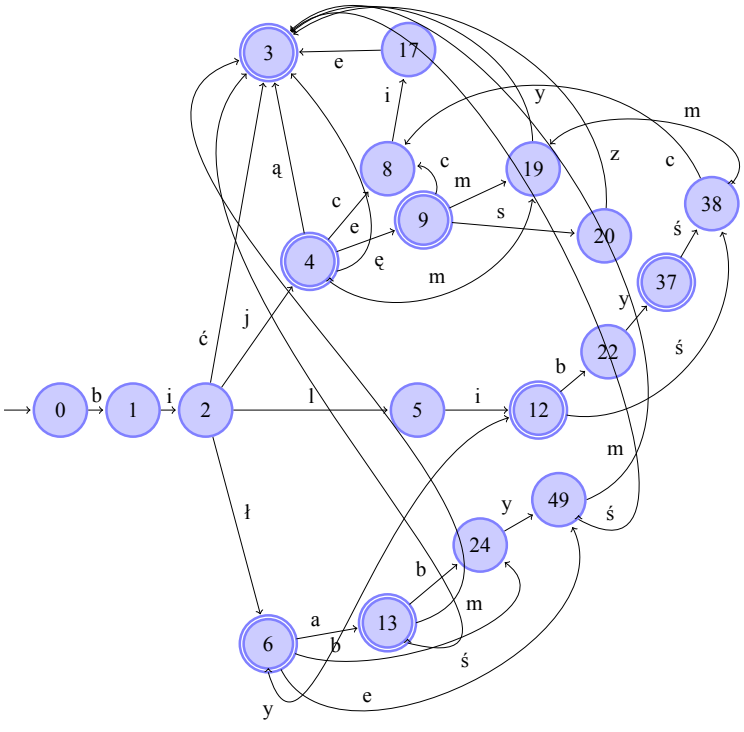


Figure 3.22: Minimal automaton recognizing all inflected forms of the Polish word *bić* except for the forms *biłaby* and *biłby*.

biłaby has been consumed. Since $w' = \varepsilon$, no minimization takes place. As the whole word was consumed, there is nothing more to add, and state 49 becomes final. Function SortedIncrementalConstruction calls function ReplOrReg, which finds that all states have their unique right language. The result is shown in Figure 3.23.

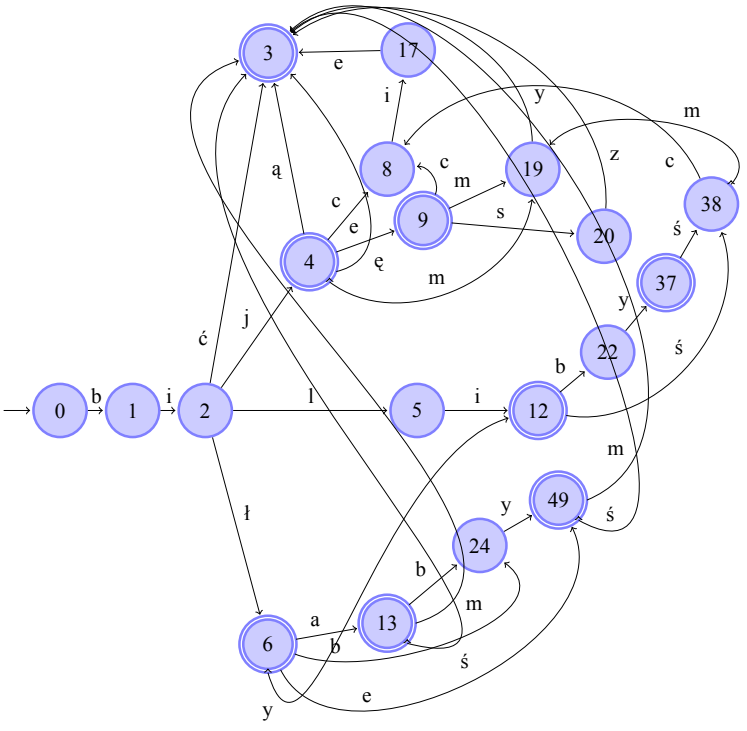


Figure 3.23: Automaton from Figure 3.22 after word *biłaby* has been added in a naive way.

It is easy to check that the resulting automaton, in Figure 3.23, recognizes word *biłby* as well. Note that Figures 3.23 and 3.9 are different, even though the automaton in Figure 3.23 is supposed to recognize the same language as the automaton in Figure 3.9. The latter has one more state. Since the former cannot be "more minimal" than the latter, and both are minimal, their languages differ. The language of the smaller automaton contains one more string: *biłe*, which does not belong to the paradigm of *bić*. Why is the string recognized? Making state 49 final served to recognize the words *biłaby* and *biłby*, as it is at the end of a path of states 0, 1, 2, 6, 13, 24, and 49, and labels on transitions along that path concatenate to form *biłaby*. However, there are two other paths of states 0, 1, 2, 6, and 49 as well as 0, 1, 2, 6, and 24 that

end in the same state. Labels on transitions along that second path form the string *biłe*. When state 49 was made final, it was not only the words *biłaby* and *biłby*, but also the string *biłe* that gained recognition. This happens because state 49 is a **confluence state** --- it has more than one incoming transition. It also means that there is more than one path going through that state. If the right language of such state is modified so that it has more items, there is more than one string added to the language of the automaton.

Recall that procedure AddStringToTrie, page 21 called by function ConstructTrie, page 20 can add words in arbitrary order. It builds a trie, and no other process modifies that trie. In a trie, there are no confluence states. Confluence states are results of minimization. Since there is no minimization in a trie (actually, a trie is the biggest deterministic acyclic automaton recognizing given language), there are no confluence states. Minimization replaces multiple copies of identical subtrees with a single representative. When only one of those multiple copies is to be modified, it needs to be extracted from the common representation. It is done by **cloning**.

Cloning means making an exact copy of a state with the same finality and the same suite of outgoing transitions. So for example, a clone of state 49 in Figure 3.22 is final, and it has two transitions, both leading to states 3: one labeled with *m*, the other one labeled with *ś*. In our example, the original state will change its number to 15, and we will name the clone state 49 to keep the original numbering of states from Figure 3.9. Now that we have separated the clone from equivalent states, we can modify its right language. The final result is shown in Figure 3.9, page 30 --- it is the minimal automaton for the whole lexeme of *bić*.

Algorithm 3.12 Function SortedCyclIncrConstruction adds sorted words to a language of a minimal automaton M using information from the register R and words to be added from the input. Both M and R are modified by the function, and M is returned.

```
 1: function SortedCyclIncrConstruction(M, R)
 2:     q' ← q0
 3:     q0 ← Clone(q0)
 4:     w' ← ε                                          ▷ previous word
 5:     while input not empty do
 6:         w ← next word
 7:         AddStrCyclSorted(M, w, w')
 8:         w' ← w
 9:     end while
10:     q0 ← ReplOrReg(M, q0, w')
11:     if q' ≠ q0 then
12:         Increment incoming transition counter for q'
13:         DeleteBranchUp(M, q')
14:     end if
15:     return M;
16: end function
```

Function SortedCyclIncrConstruction for adding strings to (possibly cyclic) automata is shown as Algorithm 3.12. Just like function SortedIncrementalConstruction, it returns a minimal automaton. Unlike function SortedIncrementalConstruction, it accepts two parameters: a minimal automaton M, and a register R. Therefore, it contains initialization of neither an

empty automaton, nor the register. A totally new item is cloning the initial state in line 3. Note that since a clone of a state has the same suite of outgoing transitions, all targets of outgoing transitions of a cloned state and of its clone are necessarily confluence states, as each of those targets has at least incoming transitions leading from both the cloned state and from the clone. Thus by cloning the initial state, we have made all states that are targets of transitions going out from q_0 confluence states. Confluence states form a barrier between the old, intact part of the automaton, and the new part modified by adding new strings. When the automaton is cyclic, the initial state may have incoming transitions. By cloning the initial state, we also prevent inadvertent addition of bogus strings to the language of the automaton. A call to ReplOrReg in line 10 can change the initial state. Either the new q_0 has a unique signature so that it remains the new initial state, or it is equivalent to q' (the old initial state), so that the current q_0 is deleted, and q' is restored as the new q_0. In the first case, states that are not reachable from q_0 are deleted by procedure DeleteBranchUp (see page 51). Instead of calling AddStrSorted, function SortedCyclIncrConstruction calls AddStrCyclSorted, which is shown as Algorithm 3.13.

Algorithm 3.13 Procedure AddStrCyclSorted adds a string to the language of an automaton. Strings must come sorted.

```
 1: procedure AddStrCyclSorted(M, w, w')
 2:     q ← q_0
 3:     i ← 1
 4:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ ∧ FanIn(q) = 1 do
 5:         q ← δ(q, w_i)
 6:         i ← i + 1
 7:     end while
 8:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
 9:         δ(q, w_i) ← Clone(δ(q, w_i))
10:         q ← δ(q, w_i)
11:         i ← i + 1
12:     end while
13:     if i ≤ |w'| then
14:         δ(q, w'_i) ← ReplOrReg(M, δ(q, w'_i), w'_{i+1...|w'|})
15:     end if
16:     while i ≤ |w| do
17:         δ(q, w_i) ← new state
18:         q ← δ(q, w_i)
19:         i ← i + 1
20:     end while
21:     F ← F ∪ {q}
22: end procedure
```

In procedure AddStrCyclSorted, the initial `while` loop has been replaced with two `while` loops. In the first one, subsequent symbols of the input string are matched against labels of outgoing transitions of the current state q of the automaton starting from the initial state q_0. If an appropriate transition is found, it is traversed changing the current state. The loop stops running if either subsequent symbol of the current word does not match any label on outgoing

transitions of the current state q, or when the target of such transition is a confluence state. For a state q, function FanIn returns the number of its incoming transitions. In an efficient implementation, that number is stored in a counter associated with a state.

The second `while` loop does the same job as the first one, but on confluence states. Obviously, it does not run if no confluence state is encountered. Inside the loop, the target state of a transition from the current state q labeled with the subsequent symbol w_i of the current word w on the input is cloned, and the transition is redirected towards the clone. In that way, a barrier between the old and the modified part of the automaton is pushed farther away.

Algorithm 3.14 Procedure DeleteBranchUp deletes a part of an automaton that is no longer reachable.

```
 1: procedure DeleteBranchUp(M, q)
 2:     Decrement incoming transition counter for q
 3:     if FanIn(q) ≤ 0 then
 4:         for σ ∈ Σ_q do
 5:             DeleteBranchUp(M, δ(q, σ))
 6:         end for
 7:         R ← R \ {q}
 8:         Delete q
 9:     end if
10: end procedure
```

Procedure DeleteBranchUp deletes a part of an automaton that is no longer reachable from the initial state. The initial state has been cloned, and the clone became the new initial state. If the new initial state after addition has been accomplished is not equivalent to the old one, and the old initial state had no incoming transitions, the old initial state is deleted. When it is deleted, states that are targets of its outgoing transitions may no longer have any incoming transitions, so they should be deleted too. The procedure starts by decreasing by one the counter of incoming transitions. If it reaches zero, DeleteBranchUp is invoked for the target of each transition leaving state q, and then state q is deleted. If the counter is negative, it means we are trying to delete a state that is already a parameter of an earlier call to DeleteBranchUp.

Now we have to prove that the algorithm works as expected. First, notice that cloning the initial state, and making the clone the new initial state does not change the language of the automaton, as the new initial state has exactly the same right language as the old one. This comes directly from the definition of cloning and from the recursive definition of the right language (2.6).

Second, calling AddStrCyclSorted(M, w) (either with the first string, or with any other string) adds w to the language of M. The first loop traverses transitions in M whose labels form a string that is the longest common prefix of w and the previously added string w' (or ε in case of the first string). Traversing transitions obviously does not change the language of the automaton. All states along the traversed path have a single incoming transition. In the second loop, if it is run, farther transitions in M are traversed. Their labels when concatenated form the next part of a prefix of a word being in the language of the original automaton before the call to SortedCyclIncrConstruction --- the part not covered by w'. During that traversal, target states of transitions are cloned, and transitions from the current state to the original target state are redirected towards the clones. Cloning a state does not change the language

of the automaton. As the clone has the same right language as the cloned state, redirection of transitions in the loop does not change the language of the automaton. Note that the clone has a single incoming transition.

Function ReplOrReg called in AddStrCyclSorted does not change the language of the automaton either. If an equivalent state is found, a transition is redirected towards it, and by the recursive definition of the right language (2.6) the language of the automaton does not change. If an equivalent state is not found, nothing changes.

Finally, the last `while` loop in AddStrCyclSorted creates a chain of states and transitions and makes the last state final so that the rest of string w is recognized. Note that independently of running or not running the second `while` loop, every state in the path recognizing the string w has a single incoming transition, therefore no additional string has been added to the language of the automaton.

After a call to AddStrCyclSorted(M, w), the only states that may not have unique right language are those that are visited when recognizing w and states that are equivalent to those states. The new initial state, and other states obtained by cloning in AddStrCyclSorted have initially the same right languages as the originals. Their right languages change as the result of adding new transitions in the third `while` loop in function AddStrCyclSorted, and the states may become equivalent to some other states in the other part of the automaton. When the next string is added, a part of the states that previously lied on the path of the current string no longer lie on the path of the newest string. They are all visited using function ReplOrReg, which checks whether equivalent states can be found elsewhere in the automaton. If equivalent states can be found, they replace states that originally lied in the path of the previous string, and the latter ones are deleted. If no equivalent states exist, then the states under examination have unique right language.

After a call to AddStrCyclSorted(M, w), the only states that are not in the register are those that lie on the path of the last string added to the automaton. Arguments for that are exactly the same as in the previous paragraph, as the mechanisms for updating the register are exactly the same.

After a call to ReplOrReg in line 10 of function SortedCyclIncrConstruction (page 49), all states have unique right language. Since the only states with non-unique right language were the states lying on the path recognizing the last string added to the language of the automaton (and states equivalent to them), function ReplOrReg is invoked with the initial state as a parameter, and the function ReplOrReg replaces all states with non-unique right languages with equivalent states while removing redundant states, no states with non-unique right language can be left afterwards. However, there can still be states with unique right language that do not contribute to the language of the automaton. They do not contribute because they are unreachable.

Procedure DeleteBranchUp removes unreachable states. Its correctness is explained on page 51 in the description of the function. The only unreachable states produced by the algorithm are the old initial state (if it has no incoming transitions) cloned at the beginning of SortedCyclIncrConstruction, and states reachable from it but not from the new initial state.

Let us see a small example that explains cloning of the initial state and removal of unreachable states. Figure 3.24 shows an automaton that recognizes words *cat* and *cow*. We want to add word *dog*.

The first thing to do is to clone the initial state. The situation after the cloning, just before the invocation of AddStrCyclSorted, is shown in Figure 3.25.

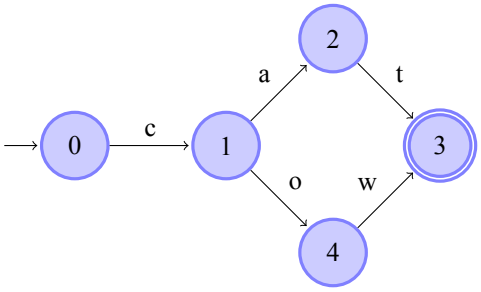


Figure 3.24: Automaton recognizing words *cat* and *cow*.

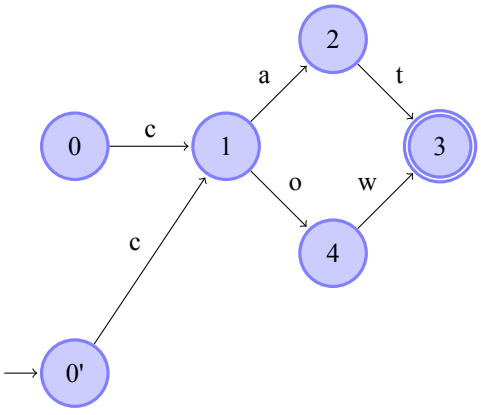


Figure 3.25: Automaton from Figure 3.24 after cloning of the initial state.

Next, AddStrCyclSorted is called to add the word *dog*. Since there is no transition from state 0 labeled with *d*, the two first `while` loops are skipped. And so is the invocation of ReplOrReg, because the previous word w' is empty. In the last `while` loop, three new states are created, as well as transitions leading to them. The last state is made final. The automaton is depicted in Figure 3.26.

Then ReplOrReg(M, 0', *dog*) is called. It deletes state 7, redirects a transition from state 6 to state 3, and puts states 6, 5, and 0' into the register as shown in Figure 3.27

Finally, DeleteBranchUp(M, 0) calls DeleteBranchUp(M, 1), which decrements incoming transition counter for state 1, then it removes state 0 from the register, and deletes the state. The final automaton is shown in Figure 3.28.

To show that DeleteBranchUp can delete more than one state, consider an automaton recognizing words *cat* and *dog* shown in Figure 3.29.

We want to add word *cow*. First, the initial state is cloned as shown in Figure 3.30.

The first `while` loop in AddStrCyclSorted is skipped. Actually, it is always skipped when adding the first string, as the targets of all transitions leaving the new initial state are confluence states. The second `while` loop clones state 1 as shown in Figure 3.31.

Since the previous word $w' = \varepsilon$, ReplOrReg is not invoked, and the last `while` loop creates two states with transitions leading to them, and then the last state is made final as

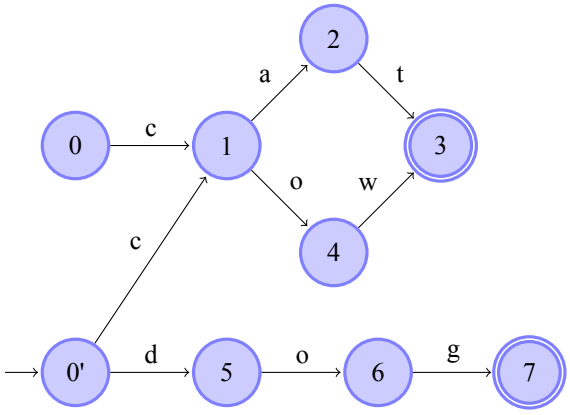


Figure 3.26: Automaton from Figure 3.24 after cloning of the initial state. States 0, 1, 2, 3, and 4 are in the register. States 0', 5, 6, and 7 are not.

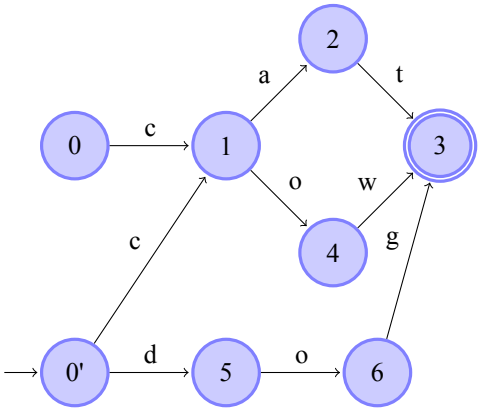


Figure 3.27: Automaton from Figure 3.26 after a call to ReplOrReg. All states are in the register. State 0 is unreachable.

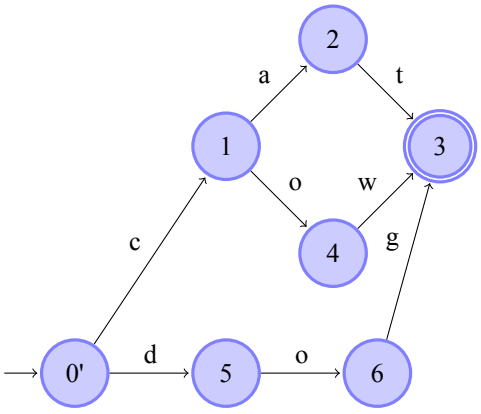


Figure 3.28: Automaton from Figure 3.27 after a call to DeleteBranchUp. All states are in the register. The automaton is minimal. It recognizes words *cat*, *cow*, and *dog*.

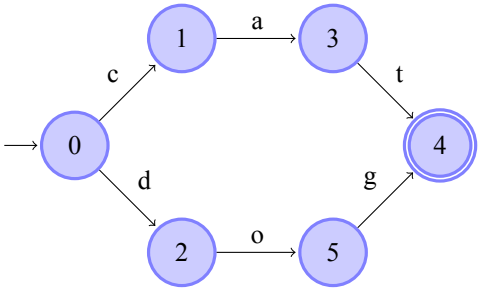


Figure 3.29: Minimal automaton recognizing words *cat* and *dog*.

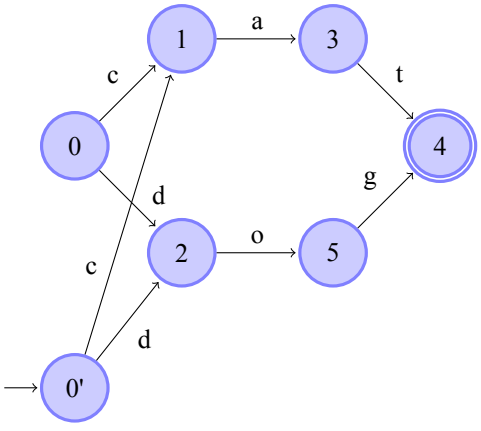


Figure 3.30: Minimal automaton recognizing words *cat* and *dog* from Figure 3.29 after cloning of the initial state.

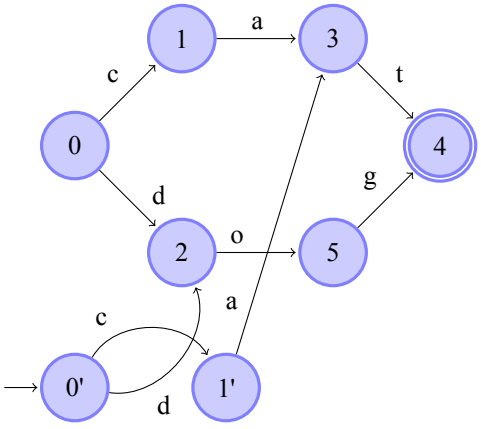


Figure 3.31: Minimal automaton recognizing words *cat* and *dog* during addition of word *cow* using SortedCyclIncrConstruction. AddStrCyclSorted has been called, and the second loop in it cloned state 1.

depicted in Figure 3.32.

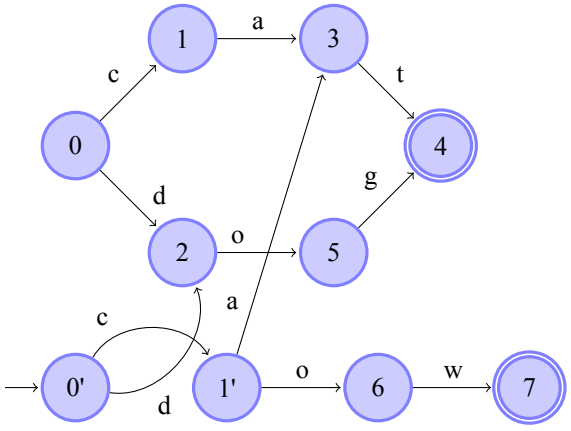


Figure 3.32: Minimal automaton recognizing words *cat* and *dog* during addition of word *cow* using SortedCyclIncrConstruction. AddStrCyclSorted has been called, and the last loop in it created states 6 and 7. All states except for 0', 1', 6, and 7 are in the register.

In line 10 of SortedCyclIncrConstruction, ReplOrReg(M, 0', *cow*); is invoked. It deletes states 7 redirecting its incoming transition to state 4, then it puts into the register states 6, 1', and 0'. Now all the states in the automaton have unique right language, and they are all in the register. However, notice that states 0 and 1 cannot be reached from the new initial state 0', so they do not contribute to the language of the automaton. The situation is shown in Figure 3.33.

Finally, DeleteBranchUp(M, 0) is called to delete unreachable states. It decrements the

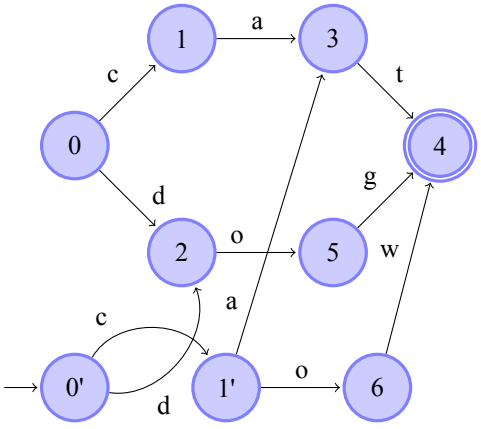


Figure 3.33: Minimal automaton recognizing words *cat* and *dog* during addition of word *cow* using SortedCyclIncrConstruction. ReplOrReg(M, 0', *cow*) has been called. All states are in the register.

incoming transition counter so that it becomes zero. Before the procedure was invoked, the counter was incremented to compensate for that decrease. When the counter reaches zero, it means that the state is to be deleted. Outgoing transitions are examined in search for other states to be deleted. DeleteBranchUp(M, 1) is called. It has one incoming transition, so the counter is decremented to zero. This triggers an invocation of DeleteBranchUp(M, 3). State 3 has two incoming transitions, so its counter is decreased to 1, and the state is not deleted. Back in the invocation DeleteBranchUp(M, 1), state 1 is removed from the register, and state 1 is deleted. Another transition going out from state 0 is examined, and DeleteBranchUp(M, 2) is called. State 2 has two incoming transitions, so its counter reaches one, and state 2 is kept. State 0 is removed from the register and deleted. The situation is shown in Figure 3.34.

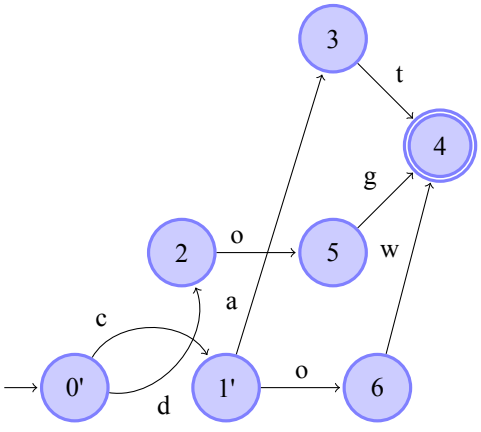


Figure 3.34: Minimal automaton recognizing words *cat* and *dog* during addition of word *cow* using SortedCyclIncrConstruction. DeleteBranchUp(M, 0) has been called.

Let us examine an example with a cyclic automaton. The initial automaton recognizes host names in URLS. The syntax of those names is simplified: they are non-empty sequences of lowercase Latin letters separated with dots. We want to add *cat*, *cow*, and *dog*. The initial automaton is shown in Figure 3.35.

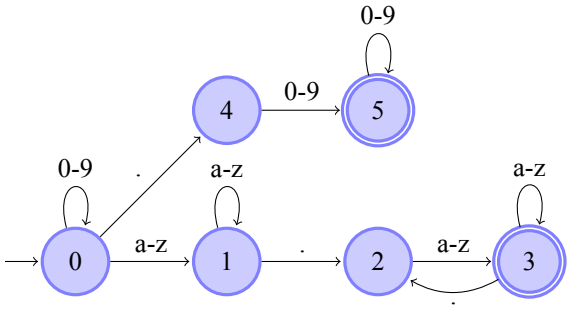


Figure 3.35: A minimal automaton recognizing simplified host names in URLs following the pattern $[a-z]^+(\backslash.[a-z]^+)^+$, and recognizing real numbers in format $[0-9]^*\backslash.[0-9]^+$. A transition labeled with *a-z* stands for many transitions labeled with consecutive letters *a, b, c*...*z*.

The initial state is cloned as in Figure 3.36.

The second `while` loop in AddStrCyclSorted first clones state 1 as shown in Figure 3.37.

State 1 has a loop labeled with letters so traversing transitions labeled *a* and *t* in the second loop of SortedCyclIncrConstruction clones that state again twice creating states 7 and 8. Since the previous word $w' = \varepsilon$, function ReplOrReg is not invoked, the last `while` loop does not run as all the letters of the word have already been used for traversal. State 8 is made final, as

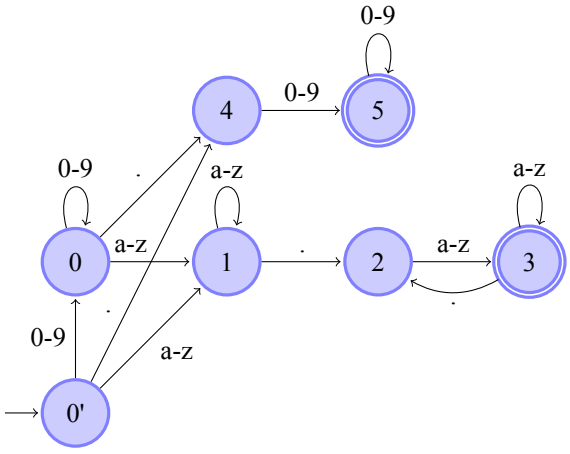


Figure 3.36: The minimal automaton recognizing simplified host names and real numbers from Figure 3.35 with the initial state 0 cloned as 0'. Note that the loop in state 0 becomes a transition to state 0 in state 0'.

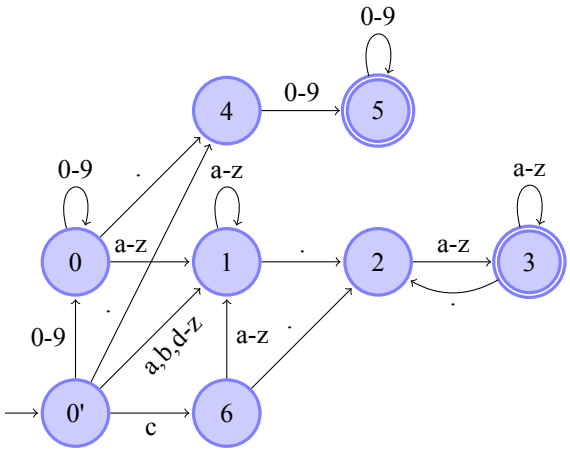


Figure 3.37: The minimal automaton recognizing simplified host names and real numbers from Figure 3.35 with the initial state 0 cloned as 0', and state 1 cloned as 6.

shown in Figure 3.38.

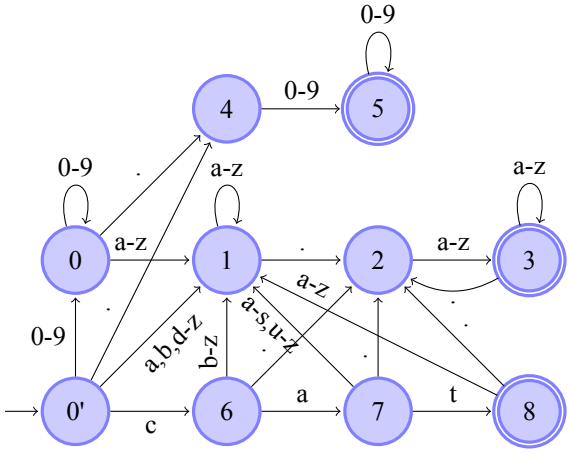


Figure 3.38: The minimal automaton recognizing simplified host names and real numbers from Figure 3.35 with the initial state 0 cloned as 0', and state 1 cloned as 6, 7 and 8. All states except for 0', 5, 7, and 8 are in the register.

The next word to add is *cow*. The first `while` loop in AddStrCyclSorted runs this time, so that a transition to state 6 is traversed. The transition from state 6 labeled with *o* leads to a confluence state 1, so state 1 is cloned as state 9. Similarly, state 1 is cloned again as state 10 when traversing a transition labeled with *w*. ReplOrReg is invoked with states 7 and 8. State 8 is final, and it has a transition leading to state 2. No other state in the register has that property, so state 8 is unique, and it is put into the register. Since state 7 is the sole state that

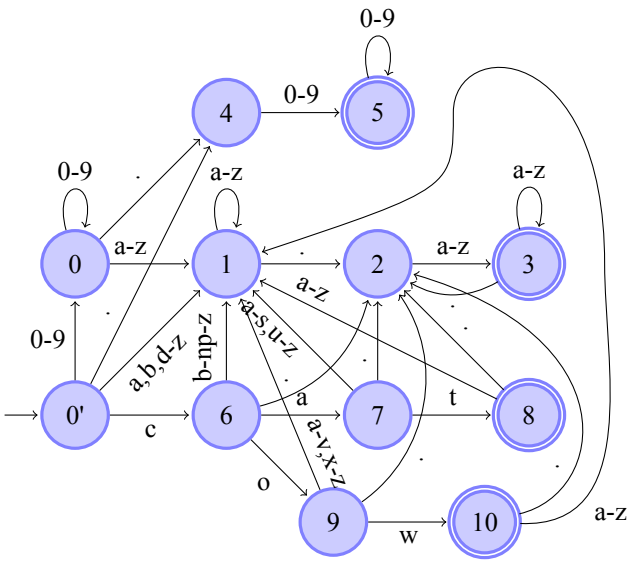


Figure 3.39: The minimal automaton recognizing simplified host names and real numbers from Figure 3.35 with *cat* and *cow* added.

has a transition leading to state 8, state 7 is also put into the register. The last `while` loop in AddStrCyclSorted does not run as there are no more letters to add, and state 10 becomes final as depicted in Figure 3.39.

The next word is *dog*. The first `while` loop in AddStrCyclSorted does not run as the transition labeled with *d* going out from state 0' goes to a confluence state 1. In the next `while` loop, state 1 is cloned three times as state 11, 12, and 13. ReplOrReg(M, 6, *ow*) is called, which calls ReplOrReg(M, 9, *w*), which in turn calls ReplOrReg(M, 10, ε). State 10 is equivalent to state 8, so state 10 is deleted, and the transition from state 9 labeled with *w* is redirected towards state 8. There are no more letters of *dog* to be used, so the third loop does not run. State 13 becomes final. This is shown in Figure 3.40.

Finally AddStrCyclSorted finishes, and ReplOrReg(M, 11, *og*) is called. It invokes ReplOrReg(M, 12, *g*), which in turn invokes ReplOrReg(M, 13, ε). State 13 is equivalent to state 8, so state 13 is deleted, and the transition labeled with *g* is redirected towards state 8. States 12 and 11 are unique, so they are added to the register. The resulting automaton is shown in Figure 3.41.

At the end of processing in SortedCyclIncrConstruction, the incoming transition counter for state 0 is incremented and DeleteBranchUp is invoked with that state as a parameter. Since that state has many incoming transitions, there are no recursive calls and the state is not deleted. Therefore, the automaton in Figure 3.41 is the minimal automaton.

This algorithm was developed by the author, and it was published in [15, 16].

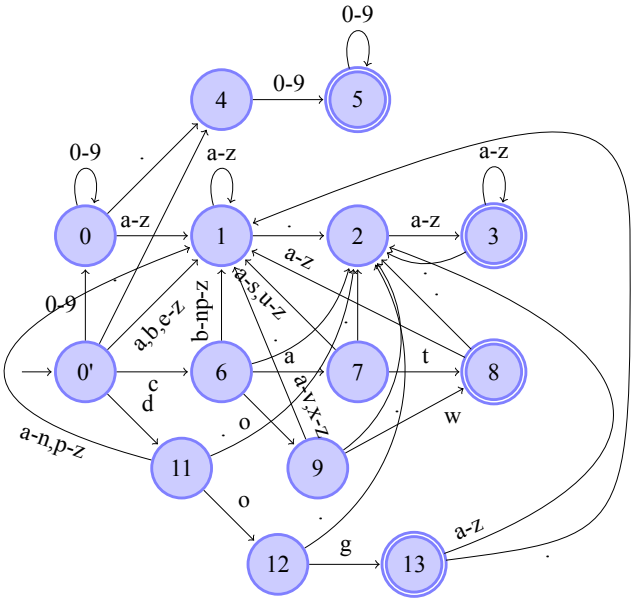


Figure 3.40: The minimal automaton recognizing simplified host names and real numbers from Figure 3.35 with *cat*, *cow*, and *dog* added.

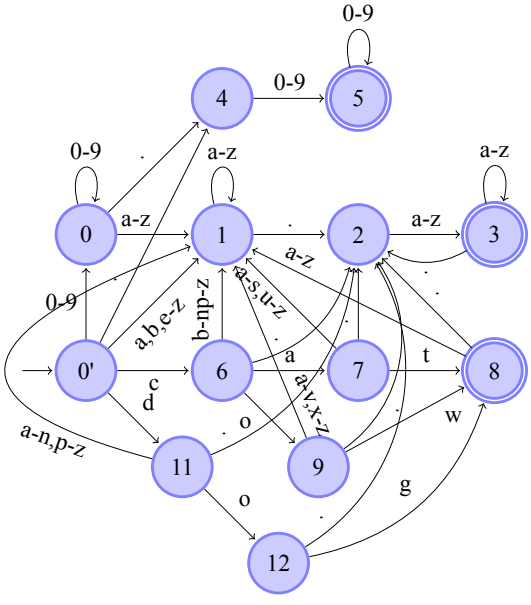


Figure 3.41: The minimal automaton recognizing simplified host names and real numbers from Figure 3.35 with *cat*, *cow*, and *dog* added after an invocation of ReplOrReg and LocalMinimization.

Chapter 4

Incremental Construction Algorithm for Unsorted Data

The algorithm for sorted data cannot handle duplicates or data coming in arbitrary order. It uses information about the ordering of data to determine which states should no longer be processed. Only such states are put into the register, so that they can replace other equivalent states. When they replace other states, they become confluence states..

When strings come in arbitrary order, we can no longer determine in advance which states will change their signatures in the future. As we want to keep the automaton close to the minimal one, we have to perform minimization after each addition till the end, that is until we get a minimal automaton.

Recall from an example on page 45, that we cannot just follow transitions starting from the initial state and labeled with subsequent symbols in the input string, and then create a chain of states and transitions recognizing the rest of the string. If we were to do so, additional strings would be added to the language of the automaton. That would happen because of the potential presence of *confluence states*. Let $w = uvz$, $u, v, w, z \in \Sigma^*$ be a string to be added to the automaton M, and that uv is the longest string such that $\delta^*(q_0, uv) \neq \perp$. If for any u, $q = \delta^*(q_0, u)$ is a confluence state, i.e. $|\overleftarrow{\mathcal{L}}(q)| > 1$, then not only w, but also all strings $\{u'vz : u' \in \overleftarrow{\mathcal{L}}(q), u' \neq u\}$ will be added to the language of the automaton.

Confluence states are result of minimization. They appear when two or more isomorphic subtrees of a trie are folded together so that only one copy is maintained. A solution has also been given in the example: it was cloning the confluence state. By cloning, a single subtree is taken out from a set of folded subtrees so that the other subtrees are not modified when the right language of the clone changes.

4.1 Incremental Construction of Acyclic DFAs from Unsorted Data

By following transitions starting from the initial state and labeled with subsequent symbols in the input string, and redirecting transitions with confluence states as their targets to clones of

those states, and then by creating a chain of states and transitions that recognize the missing part of the input string, we are able to add the string to the language of the automaton in such a way that no additional strings are added there. However, the resulting automaton is not minimal.

Since by adding a string, only a part of the automaton is modified, it is more efficient to locally minimize only the part that has changed, instead of globally minimizing the whole automaton from the scratch. The procedure outlined in the previous paragraph guarantees that only the states on the path of the newly added string change their right language. Therefore, only those states should be subject to local minimization. As only the signature of a state is computed instead of the whole right language for equivalence checking, not all states in the path of the newly added string need to be examined. Certainly, the states that are created as a result of adding new string, and a state that directly precedes them, need to undergo local minimization. This includes the states that form a path that recognizes the suffix of the newly added string, and the clones of confluence states.

Algorithm 4.1 Incremental construction algorithm for unsorted data.

```
1: function UnsortedIncrementalConstruction
2:     Create empty automaton M = ({q0}, Σ, ∅, q0, ∅)
3:     R ← {q0}
4:     while input not empty do
5:         w ← next word
6:         AddStrUnsorted(M, w)
7:     end while
8:     return M
9: end function
```

Function UnsortedIncrementalConstruction is given as Algorithm 4.1. In comparison with function SortedIncrementalConstruction given as Algorithm 3.6 on page 31, one can see the following four differences:

1. The previous string is not remembered nor passed to function AddStrUnsorted.
2. There is no call to function ReplOrReg.
3. Function AddStrSorted has been replaced with function AddStrUnsorted.
4. The register is initialized with the initial state q_0.

The first difference comes from the fact that processing the current string does not directly depend on the previous string. The second one is the result of performing local minimization with no states left for later handling, i.e. with the minimal automaton as the outcome. The third difference is obvious, as the functions implement the core of their respective algorithms. The fourth difference stems from the fact that function AddStrUnsorted minimizes an automaton completely, and that the register should reflect that.

Procedure AddStrUnsorted is given as Algorithm 4.2. It begins by setting initial values of variables. Variable P holds a path of non-confluence states of the automaton that are visited during recognition of the initial segment of the string w. In the first `while` loop, that path is filled in with states. The last state in that path, stored in variable p, is the only state present

Algorithm 4.2 Procedure AddStrUnsorted adds a string w to the language of an acyclic automaton M. No assumption about previously added strings is made.

```
 1: procedure AddStrUnsorted(M, w)
 2:     q ← q_0
 3:     i ← 1
 4:     P ← ε                          ▷ path of non-confluence states in the common prefix
 5:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ ∧ FanIn(δ(q, w_i)) = 1 do
 6:         push q onto P
 7:         q ← δ(q, w_i)
 8:         i ← i + 1
 9:     end while
10:     p ← q
11:     j ← i
12:     R ← R \ {p}
13:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
14:         δ(q, w_i) ← Clone(δ(q, w_i))
15:         q ← δ(q, w_i)
16:         i ← i + 1
17:     end while
18:     while i ≤ |w| do
19:         δ(q, w_i) ← new state
20:         q ← δ(q, w_i)
21:         i ← i + 1
22:     end while
23:     F ← F ∪ {q}
24:     LocalMinimization(M, P, w, ReplOrReg(M, p, w_{j...|w|}), p, j)
25: end procedure
```

in the automaton before addition of w that obligatory changes not only its right language, but also its signature as the result of adding w to the language of M, unless w is already in the language of the automaton. Along with the last state stored in p, the current symbol index i in the string w is stored in j. The state p is removed from the register as it will receive a new outgoing transition (or it will change its finality).

The second `while` loop creates clones of confluence states along the path of w and redirects transitions from the previous state on the path towards the clones. Recall that cloning a state makes targets of its outgoing transitions confluence states too. Clones of confluence states are new states; they are not in the register.

The third `while` loop creates a chain of states and transitions between them so that the rest of the string w is recognized. This loop does not differ from analogous loops in other analogous string adding functions being presented so far. The last state in the chain is made final. Notice that the remaining part of w can be empty, i.e. $\delta(q_0, w) \in Q$ already before the addition. In such case, no states and no transitions are created, but either p or the last clone of a confluence state (if present) is made final. This concludes the ``forward" part of the function.

In the ``backward" part of the function, two other functions are called: ReplOrReg (see

Algorithm 3.8 on page 32) is called to evaluate a parameter of a call to LocalMinimization. Contrary to the situation in all versions of the algorithm for sorted data, function ReplOrReg is invoked on a part of the path of the current string, not the previous one. The part of the path begins with the state stored in p. That state is the first state in the path recognizing w that is not in the register. The top level call to the function returns either the same state, or another state equivalent to it.

Algorithm 4.3 Procedure LocalMinimization. M is the automaton, P is the path recognizing the word w, s is the state returned by the top-level call to function ReplOrReg, q is the original state either the same as s or equivalent to it before it has replaced with s, i is the position in the path P and in the word w (an index on those structures).

```
 1: procedure LocalMinimization(M, P, w, s, q, i)
 2:     while P not empty ∧ s ≠ q do
 3:         pop p from P
 4:         R ← R \ {p}
 5:         δ(p, w_i) ← s
 6:         i ← i − 1
 7:         q ← p
 8:         s ← p
 9:         if ∃r ∈ R : r ≡ p then
10:             delete s
11:             s ← r
12:         else
13:             R ← R ∪ {p}
14:         end if
15:     end while
16: end procedure
```

At the end of AddStrUnsorted, procedure LocalMinimization is invoked. Its first parameter is the automaton M, the second -- the path P (here as a sequence of states) recognizing the initial part of the string w (the third parameter) being added. The fourth parameter s is the state returned by the function ReplOrReg, the fifth q -- the state hold in p in AddStrUnsorted. The last parameter i is the number of the symbol in w that labels a transition from p to the next state on the path. The procedure minimizes the initial segment of the path recognizing w. That path is stored in the parameter P that is a stack of states. Contrary to the function ReplOrReg, which operates on new states that are not yet in the register, LocalMinimization handles states in the path of w already present in the automaton before w was added. All those states change their right language, but not all of them change their signature. At the beginning, it is the state preceding the state given as parameters s and q that changes (this happens when q is different from s) its signature. If its right language remains unique in M, then the signature of the state preceding it in P does not change, nor do signatures of any states in P closer to q_0. If the right language is not unique, the state is replaced with an equivalent one, and the preceding state changes its signature. So, before the transition from the preceding state to the current state is modified, the preceding state p is removed from the register. After the modification, a search for a state equivalent to state p is performed. If an equivalent state is not found, state p is put into the register again, the `while` loop and the whole procedure

is completed. If not, q becomes p, s becomes a state equivalent to p, and the `while` loop is executed again, with p becoming the previous state in the path P. The loop ends when either no state equivalent to a previous state is found, or when the stack P is empty, i.e. when the last state handled in the loop is the initial state.

Let us check how the algorithm works in practice. Function UnsortedIncrementalConstruction creates an empty automaton with an initial state $q_0 = 0$ and it sets a register with $\{0\}$. Then it reads subsequent strings from the input and adds them to the language of the automaton using procedure AddStrUnsorted. Each time a string is added, the automaton is completely minimized, and the register is updated accordingly. Let us add word *bić* to an empty automaton. Procedure AddStrUnsorted initializes variables, and skips the first `while` loop as there are no transitions to follow. Variable p becomes 0, j becomes 1, and state 0 is removed from the register. Since there are no transitions in the automaton, the second `while` loop does not run either. The third `while` loop creates states 1, 2, and 3, as well as transitions from state 0 to state 1 labeled with *b*, from state 1 to state 2 labeled with *i*, and from state 2 to state 3 labeled with *ć*. Right after that, state 3 becomes final. Function ReplOrReg(M, 0, *bić*) is invoked. It calls ReplOrReg(M, 1, *ić*), which calls ReplOrReg(M, 2, *ć*), which calls ReplOrReg(M, 3, ε). In the last call, state 3 is found to be unique, and it is added to the register, as well as returned to the call one level up. There, state 2 is found to be unique, and so are state 1 and 0 higher up. The top level call returns state 0. All states of the automaton are in the register. Procedure LocalMinimization(M, ε, *bić*, 0, 0, 1) is invoked. Since the stack P is empty, the `while` loop is not entered, and the procedure immediately returns. The automaton is depicted in Figure 4.1

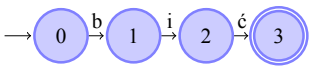


Figure 4.1: Minimal automaton recognizing word *bić*.

As words are added in arbitrary order, and procedure AddStrUnsorted minimizes the automaton completely, we can skip addition of several strings.

Consider an automaton recognizing all forms of a verb *bić* except for *biłaby* and *biłby*. The word *biłaby* is to be added to the language of the automaton. This is the same situation as described at the beginning of Section 3.4 on page 45 in the example illustrating the behavior of confluence states. The automaton shown in Figure 4.2 is the same as that in Figure 3.22 on page 47.

Procedure AddStrUnsorted starts with initializing variables, and then the first `while` loop is entered. Transitions with subsequent letters of *biłaby* are followed from state 0 until state 24 is reached. State 24 is a confluence state. Inside the loop, states 0, 1, 2, 6, and 13 are pushed onto the stack P. Variable p becomes 13, j becomes 5, and state 13 is removed from the register. Now the second `while` loop is entered. State 24 is cloned as state 50. Variable q is now 50, and i is now 6. The situation is shown in Figure 4.3.

Transition from state 50 labeled with y leads to state 49 that is also a confluence state. The `while` loop is run again. State 49 is cloned as state 51, which becomes a new value of q. Variable i is incremented to 7. Since that value is bigger than the length of the current string w, this loop exits, and the third loop is skipped. State 51 becomes final. This completes the ``forward" step as shown in Figure 4.4. The last statement in procedure AddStrUnsorted is a

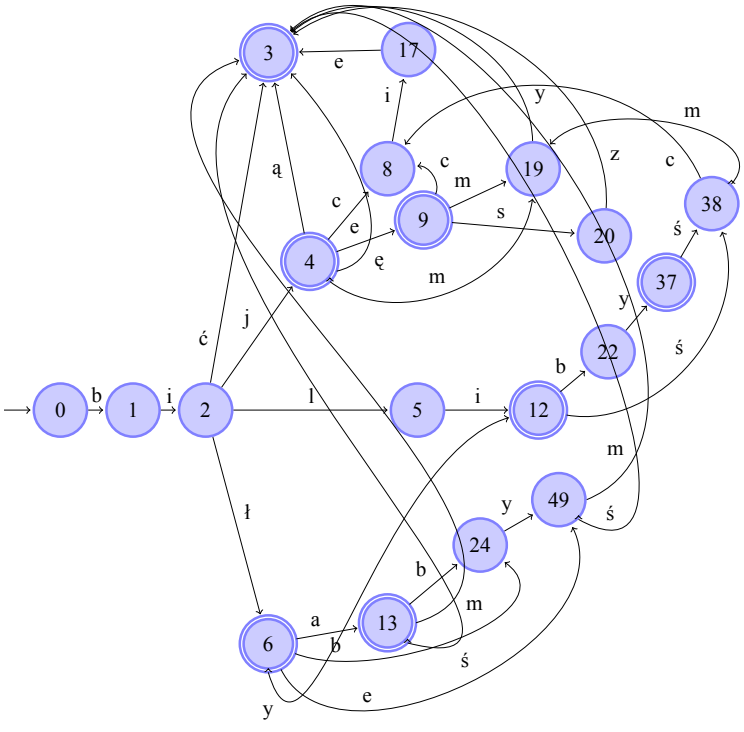


Figure 4.2: Minimal automaton recognizing all inflected forms of the Polish word *bić* except for the forms *biłaby* and *biłby*. A copy of Figure 3.22.

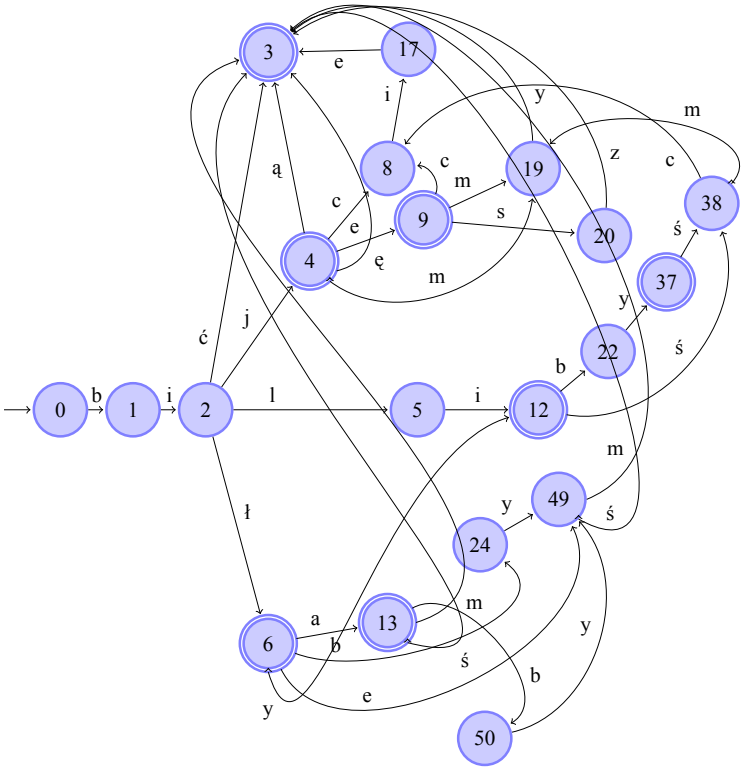


Figure 4.3: Automaton from Figure 4.2. Word *biłaby* is being added. State 24 has just been cloned as state 50.

call to LocalMinimization, which includes a call to ReplOrReg. Recall that p is 13, and j is 5. ReplOrReg(M, 13, *by*) calls ReplOrReg(M, 50, *y*), which in turn calls ReplOrReg(M, 51, ε).

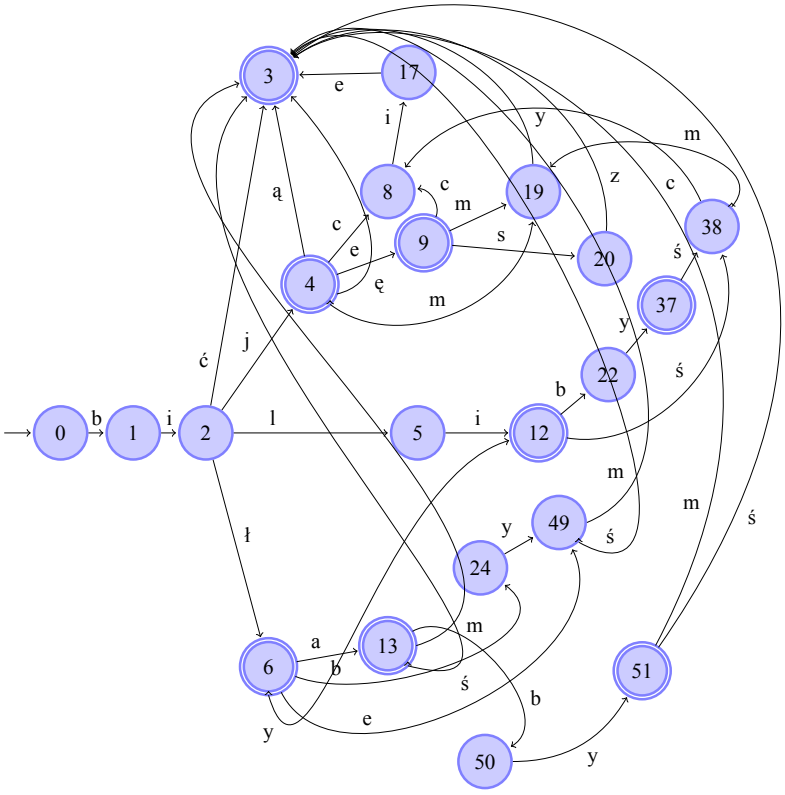


Figure 4.4: Automaton from Figure 4.2. Word *biłaby* is being added. State 49 has just been cloned as state 51.

The right language of state 51 is $\{\varepsilon, \text{m}, \text{ś}\}$. That right language is unique in M. The state is added to the register and returned by the function. As $\overrightarrow{\mathcal{L}}(51)$ is unique, so must be $\overrightarrow{\mathcal{L}}(50)$, so state 50 is added to the register and returned by the function. And so is state 13, which is also added to the register and returned. Now LocalMinimization(M, (0, 1, 2, 6, 13), *biłaby*, 13, 13, 5) is invoked. The stack P is not empty, but both s and q are state 13, because the signature of state 13 has not been modified. The `while` loop is not entered, and the procedure terminates. All states are in the register, and the automaton is minimal.

Now let us add word *biłby*. AddStrUnsorted(M, *biłby*) is called. Variable q is set to state

0, i to 1, P to an empty stack. The first `while` loop is entered. States 0, 1, 2, 6, and 24 are visited and states 0, 1, 2, and 6 are pushed onto stack P. Variable q becomes 24, and i becomes 5. The loop terminates as state 49 is a confluence state. Variable p becomes 24, j becomes 5, and state 24 is removed from the register. The second `while` loop is entered. State 49 is cloned as state 52. Variable q becomes 15, i becomes 6. Since $6 < |bilby|$, the loop terminates. State 52 becomes final, as portrayed in Figure 4.5.

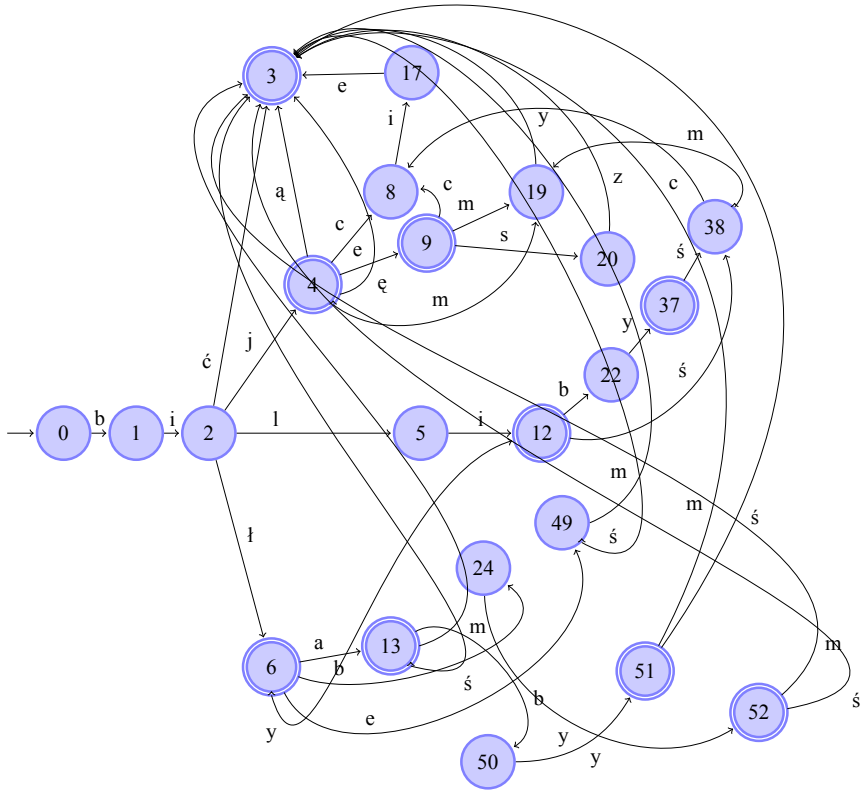


Figure 4.5: Automaton from Figure 4.4. Word *bilby* is being added. State 49 has just been cloned as state 52.

Procedure LocalMinimization is invoked, which calls function ReplOrReg(M, 24, y), which then calls ReplOrReg(M, 52, ε). State 52 is found equivalent to state 51, so state 52 is removed, and a transition labeled y from state 24 to state 52 is redirected towards state 51, which is returned by the function. The situation is shown in Figure 4.6.

In ReplOrReg(M, 24, y), state 24 is found equivalent to state 50, so state 24 is removed.

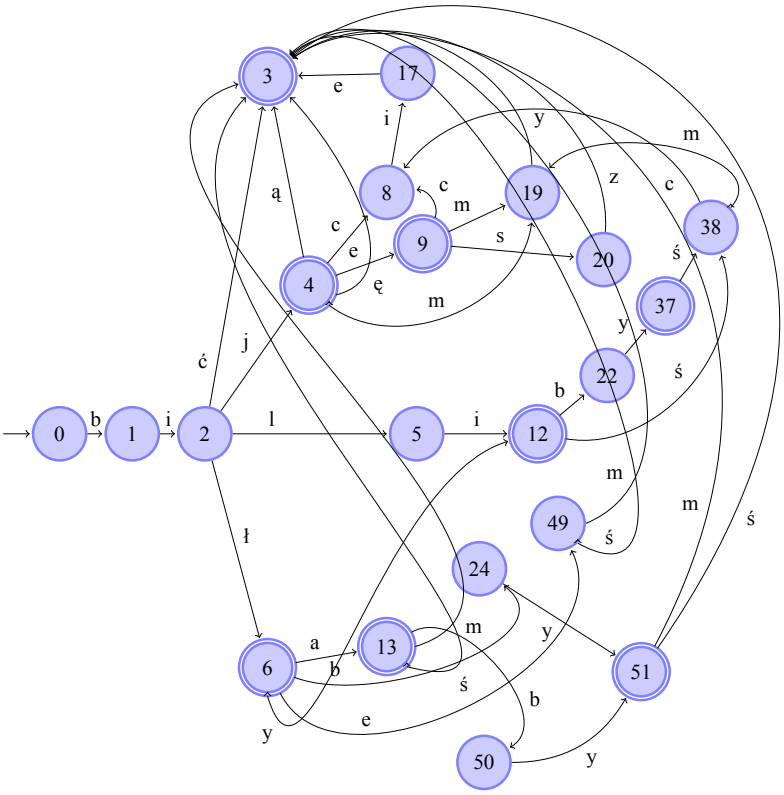


Figure 4.6: Automaton from Figure 4.4. Word *biłby* is being added. State 15 has just been removed and its incoming transition has been redirected towards state 51, and it has been made final.

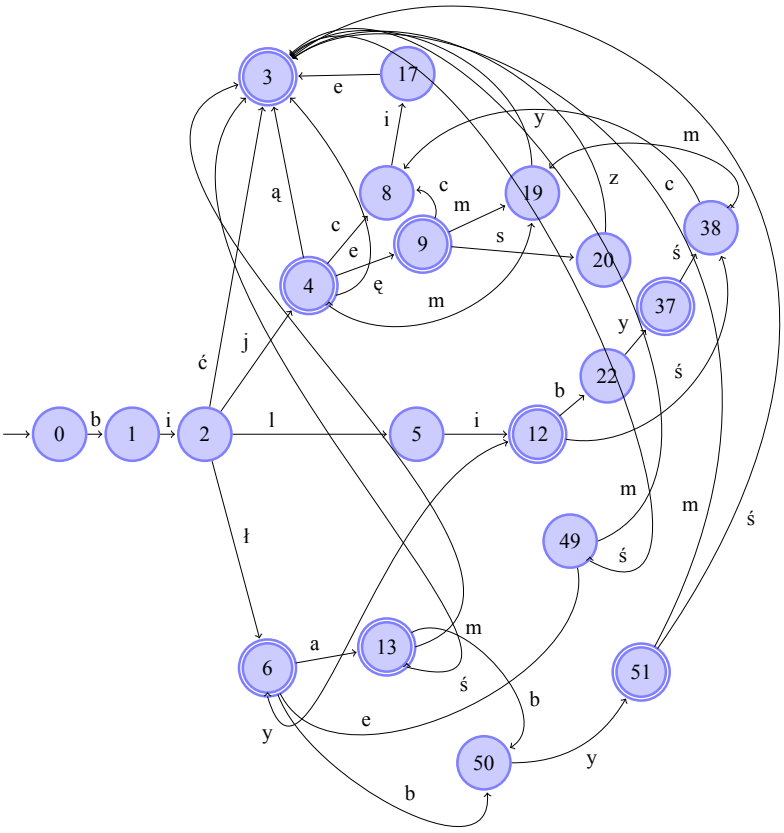


Figure 4.7: Automaton from Figure 4.4. Word *biłby* is being added. State 24 has just been removed and its incoming transition has been redirected towards state 50.

State 50 is returned from ReplOrReg(M, 24, y) to become the third parameter in the invocation of LocalMinimization(M, (0, 1, 2, 6), *biłby*, 50, 24, 5). In the `while` loop, the stack is not empty and states $s = 50$ and $q = 24$ are different. State 6 is popped from the stack, stored in p, and removed from the register. Then, a transition from state 6 to state 24 labeled with b is redirected to state 50 (see Figure 4.7). Variable i is decremented to 4, state q becomes 6, and state s becomes 6. A search for a state equivalent to state 6 is conducted in the register. No such state can be found, so state 6 is added back to the register (this time with a different right language). Now $s = q = 6$, so the `while` loop cannot be run, and the procedure finishes. The minimal automaton is shown in Figure 4.7. It is equivalent to the automaton in Figure 3.9.

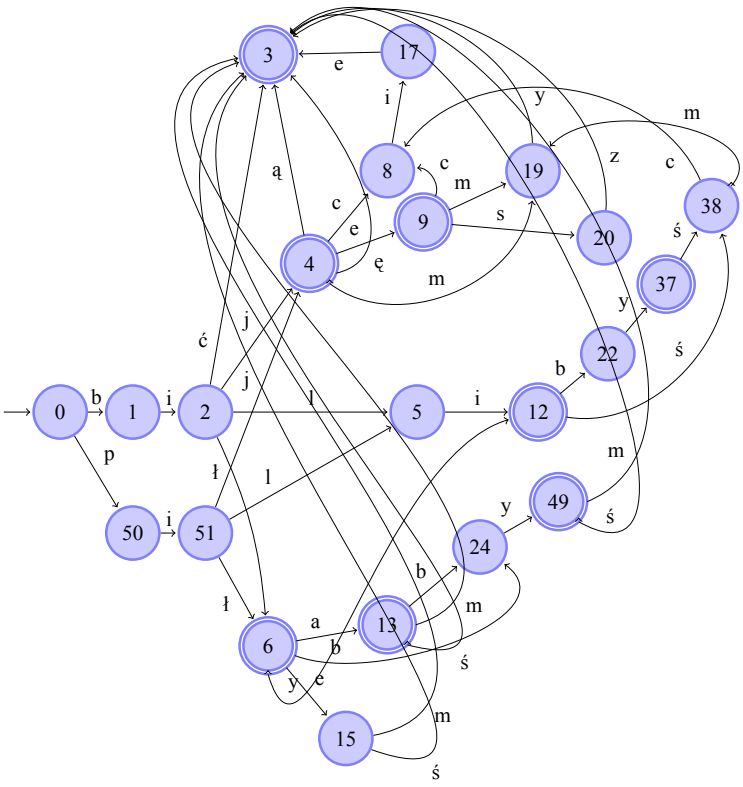


Figure 4.8: Minimal automaton recognizing all inflected forms of lexeme *bić*, and all inflected forms of lexeme *pić* except for the form *pić* (the infinitive, which is the canonical form).

Let us consider the last example. Figure 4.8 recognizes all inflected forms of lexemes *bić*

and *pić* except for one: the infinitive *pić*. We add that missing form. All states are in the register R, and we call AddStrUnsorted(M, *pić*). After the first `while` loop, $P = (0, 50)$, $q = 51$, $i = 3$. The loop terminates, as there is no transition leaving state 51 labeled with *ć*. Variable p becomes 51, j becomes 3, and state 51 is removed from the register. The second `while` loop does not run as there is no transition leaving state 51 labeled with *ć*. In the third loop, state 52 is created along with a transition leading to it from state 51 and labeled with *ć*. State q becomes 52, and i becomes 4, which terminates the loop. State 52 becomes final --- see Figure 4.9. State 52 is not in the register. Neither is state 51, which has just been removed from there. All other states are in the register.

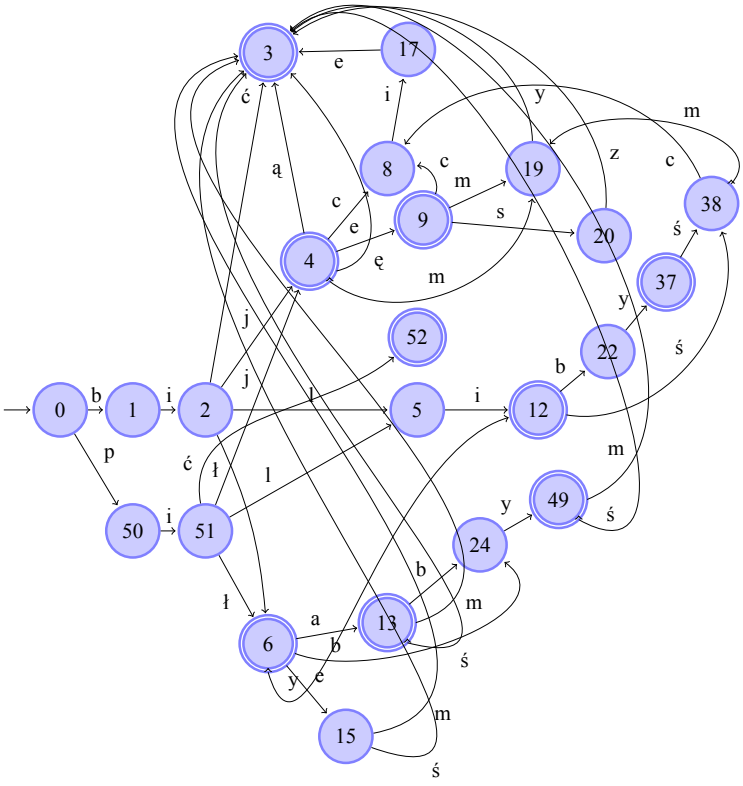


Figure 4.9: Minimal automaton recognizing all inflected forms of lexeme *bić*, and all inflected forms of lexeme *pić*. The automaton is minimal except for the path recognizing the word form *pić*, which has just been added with a call to AddStrUnsorted. State 52 has just been made final before a call to LocalMinimization.

Procedure LocalMinimization(M, $P = (0, 50)$, w =*pić*, ReplOrReg(M, $p = 51$, *ć*),

$p = 51$, $j = 3$) is called. In that invocation, ReplOrReg(M, 51, *ć*), 51, 3) is called first. It calls itself as ReplOrReg(M, 52, ε). State 3 is equivalent to state 52, so state 52 is deleted, and state 3 is returned. In the top level call to ReplOrReg, a transition from state 51 to state 52 labeled with *ć* is redirected towards state 3. State 51 is now equivalent to state 2, so state 51 is deleted, and state 2 is returned, so that the call to LocalMinimization is LocalMinimization(M, (0, 50), *pić*, 2, 51, 3). As $s = 2 \neq q = 51$, and the stack is not empty, the `while` loop is entered. State 50 is popped from the stack, stored in variable p, and removed from the register. A transition from state 50 to state 51 labeled with *i* is redirected towards state 2. Variable i is decremented to reach 1, q and s become state 50. State 50 is equivalent to state 1. State 50 is deleted, and s becomes state 1. Now the stack contains state 0, $s = 1$, and $q = 50$. The `while` loop runs once more. State 0 is popped from the stack, stored in p, and removed from the register. A transition from state 0 to state 50 labeled with *p* is redirected towards state 1. Variable i is decremented so that it is equal to 0, q and s become state 0. There is no state equivalent to state 0, so state 0 is put back into the register. As the stack is now empty, and variables s and q are both equal to state 0, the loop and the procedure terminate. The resulting automaton is shown in Figure 4.10.

Let us prove that the algorithm is correct. It is sufficient to show that procedure AddStrUnsorted is correct. Let $M = (Q, \Sigma, \delta, q_0, F))$ be the automaton before an AddStrUnsorted invocation, $M' = (Q', \Sigma, \delta', q_0', F')$ --- the automaton after the invocation, R --- the register before the invocation, R' --- the register afterwards, and w the word to be added. We show that:

1. $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$
2. M' is minimal
3. $R' = Q'$

To prove the first point, let us notice that the first `while` loop in procedure AddStrUnsorted does not change the automaton at all. It only changes the current state and the current symbol number. It also puts states on the stack P. The second `while` loop does change the automaton. New states are created by cloning, and transitions are redirected to the clones. However, no preexisting states change their right languages since their outgoing transitions can only be redirected to states equivalent to their previous targets. That loop also ensures that all states lying on the path of w in the automaton have only one incoming state (except for the initial state), and that that state is also on the same path. After those two loops, $\delta^*(q_0, w_{1\ldots j-1}) = p$, and if $j \leq |w|$, then $\delta(p, w_j) = \perp$. The last `while` loop and the statement that follows it immediately create a chain of states and transitions such that $\delta^*(p, w_{j\ldots|w|}) \in F$. As only one final state is added, and all the states on the path recognizing w have only one incoming transition, only w is added to the language of the automaton. An invocation of procedure LocalMinimization in the last line of procedure AddStrUnsorted contains a call to function ReplOrReg. That function does not change the right language of any state, as it replaces targets of outgoing transitions with equivalent states. It does delete states, but it ensures that the preceding states on the path recognizing w are all reachable. Therefore, the language of the automaton does not change. Finally, procedure LocalMinimization is similar, in its behavior, to function ReplOrReg. It does not change the right language of any state. It does delete states, but since it ensures that the states on the path recognizing w are all reachable,

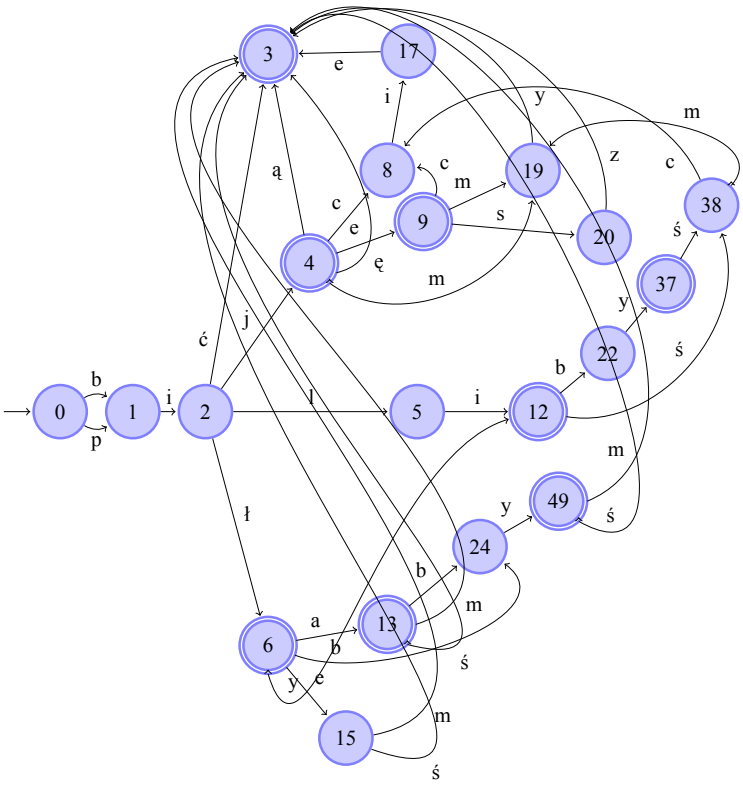


Figure 4.10: Minimal automaton recognizing all forms of lexemes *bić* and *pić*.

the right language of q_0 does not change, so the language of the automaton does not change. Therefore $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$.

To prove the second point, let us notice that the automaton is minimal before procedure AddStrUnsorted is called. The first loop in the procedure does not change the automaton, and the second loop only possibly creates copies of states on the path recognizing w. The third loop creates new states, which might have equivalent ones somewhere else in the automaton. Those new states lie on the path recognizing w. So, before the invocation of procedure LocalMinimization, only the states that lie on the path recognizing w (and the states equivalent to them) might have a right language that is not unique. Function ReplOrReg examines the states that form the (possibly empty) suffix of the word w. Those states are processed using postorder, so that the simplified form of Equation (3.2) (page 24) can be used. If equivalent states are found, they replace the states created in the third loop of procedure AddStrUnsorted.

When the top-level invocation of function ReplOrReg finishes, the states recognizing the suffix of w have all unique right languages. Procedure LocalMinimization proceeds in a similar way. States in the path of w are replaced with their equivalents if such states exist using postorder. However, the procedure can stop before reaching the initial state. It stops when a subsequent state (let's call it q) does not have an equivalent. A question can be asked whether states that have an equivalent state somewhere else may exist on the path from q_0 to q. Since the states are processed in postorder, and the signature of the states preceding q on the path of w does not change, such states cannot exist. Thus M' is minimal.

Changes in the register follow the changes in the inventory of states of the automaton. The second `while` loop in procedure AddStrUnsorted creates clones of existing states without adding them to the register. Before the loop is executed or skipped, the last visited state is removed from the register and stored in variable p. When the loop is executed, a transition from state p is redirected towards the clone of its original target state. When the loop is skipped, i.e. when the next state on the path is not confluence, either the third loop creates a new transition going out from state p, or state p becomes final. In all those cases state p changes its signature. So the state is removed from the register before it changes its signature. In function ReplOrReg, all the states created in the third loop of procedure AddStrUnsorted are either deleted or put into the register. The same happens with state p[1]. In procedure LocalMinimization, states are removed from the register before they change their signature. If state is removed from the register, it changes the target of one of its outgoing transitions, and then it is either deleted, or it is put back into the register. If the state is deleted, then the preceding state (if any) is removed from the register, and it undergoes the same procedure. Thus $R' = Q'$.

As with other algorithms described in this book, we will assume that operations on the register take constant amount of time. If it is not so, then the complexity function must include a factor responsible for modeling the complexity of register operations. Under our assumption, creating an empty automaton and putting one state into the register in function UnsortedIncrementalConstruction takes constant time. The `while` loop is executed n times, where n is the number of words on the input. The loop contains two instructions: reading a word from the input, and an invocation of procedure AddStrUnsorted. Reading a word takes $\mathcal{O}(|w_{max}|)$ time, where w_{max} is the longest word on the input.

Procedure AddStrUnsorted has three `while` loops, a few assignments, one operation on the register, and an invocation of procedure LocalMinimization. The assignments, the operations on the register, and making a state final[2] take constant amount of time. The first `while` loop contains checking a few conditions, putting a state onto a stack, and two assignments. They can all be done in constant time, The loop is executed at most $|w_{max}|$ times. The second `while` loop is similar, but instead of putting a state onto a stack, we have cloning, which also takes constant (proportional to $|\Sigma|$) amount of time. The third `while` loop contains checking a simple condition, creation of a new state, and assignments. It runs in constant time at most $|w_{max}|$ times.

A call to procedure LocalMinimization contains an invocation of ReplOrReg. The latter calls itself at most $w_{max}|$ times as it operates along the path of w. Except for nested calls, each

[1] State p can be deleted in function ReplOrReg. The transition from the preceding state in the path recognizing w will point to a deleted state. As long as only the address of the deleted state is stored, it is correct. The preceding state will first be removed from the register, and then the address of the target state will be changed.

[2] Implemented as setting a flag on the state.

invocation of ReplOrReg is executed in constant time. Thus the top-level call to ReplOrReg has total time complexity of $\mathcal{O}(|w_{max}|)$. The loop in procedure LocalMinimization executes at most $|w_{max}|$ times. It contains only constant-time operations: assignments, register operations, deletion of states. Therefore, the complexity of UnsortedIncrementalConstruction is $\mathcal{O}(n|w_{max}|)$.

As to memory complexity, we need space for the states (the number of states $|Q|$ is never greater than the number of states in the minimal automaton recognizing the current language of the automaton plus the length of the last word added to the automaton), the number of edges (at most $|\Sigma||Q|$), and the register (proportional to $|Q|$).

This algorithm was first described by J. Aoe, K. Morimoto, and M. Hase in [4], then developed independently by Jan Daciuk [18, 21, 12], Bruce Watson and Richard Watson [18, 21], and Dominique Revuz [32].

4.2 Extension to Pseudo-Minimal Automata

The incremental algorithm for unsorted data can easily be extended so that it constructs pseudo-minimal automata. The main function, given as Algorithm 4.4, calls a slightly different function for adding new words.

Algorithm 4.4 Incremental construction algorithm of pseudo-minimal automata from unsorted data.

```
1: function PseudoUnsortedIncrementalConstruction
2:     Create empty automaton M = ({q0}, Σ, ∅, q0, ∅)
3:     R ← {q0}
4:     while input not empty do
5:         w ← next word
6:         PseudoAddStrUnsorted(M, w)
7:     end while
8:     return M
9: end function
```

That function is PseudoUnsortedIncrementalConstruction, presented here as Algorithm 4.2. It differs from UnsortedIncrementalConstruction only in the functions it calls. Instead of ReplOrReg, PseudoReplOrReg (Algorithm 3.11, page 42) is called. Instead of LocalMinimization, PseudoLocalMinimization is called. The latter is given as Algorithm 4.6. To keep track of divergent states, variable d is introduced. It is initialized to false, and it becomes true when function PseudoReplOrReg returns a divergent state.

Procedure PseudoLocalMinimization is shown as Algorithm 4.6. It does the same job as procedure LocalMinimization, but it keeps track whether states are divergent, and in that case, replacement with an equivalent state does not take place, and the procedure terminates.

Let us see how the algorithm works. Figure 4.11 shows a pseudo-minimal automaton recognizing all inflected forms of lexeme *bić* except for forms *biłaby, biłabyś, biłby*, and *biłbyś*. We add *biłby*. The register contains states 3, 8, 9, 17, 19, 20, 24, and 49. It does not contain divergent states.

Variable q is set to state 0, variable i is set to 1, the stack P is empty. The first loop traverses non-confluence states 0, 1, 2, and 6. States 0, 1, and 2 are put onto stack P. Variables q and

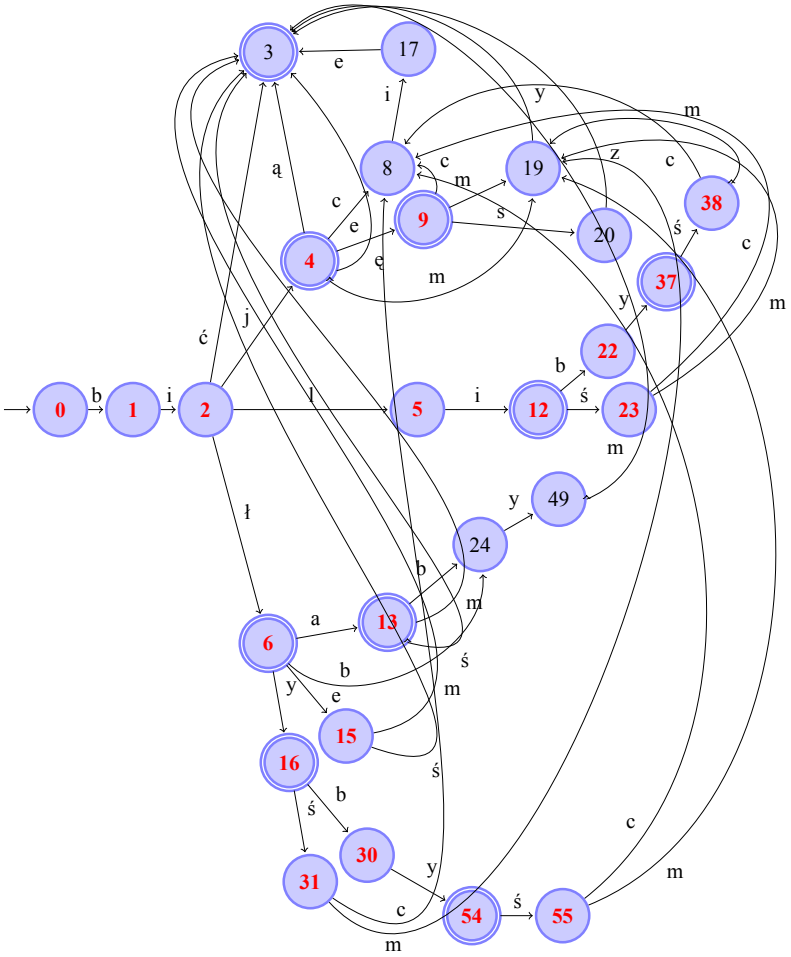


Figure 4.11: Pseudo-minimal automaton recognizing all inflected forms of lexeme *bić* except for *biłaby, biłabyś, biłby*, and *biłbyś*. (Divergent) states not contained in the register are shown in different color. Actually, there may well be some divergent states that are in the register, e.g. states 0 and 1 could be there, but this is what we assume.

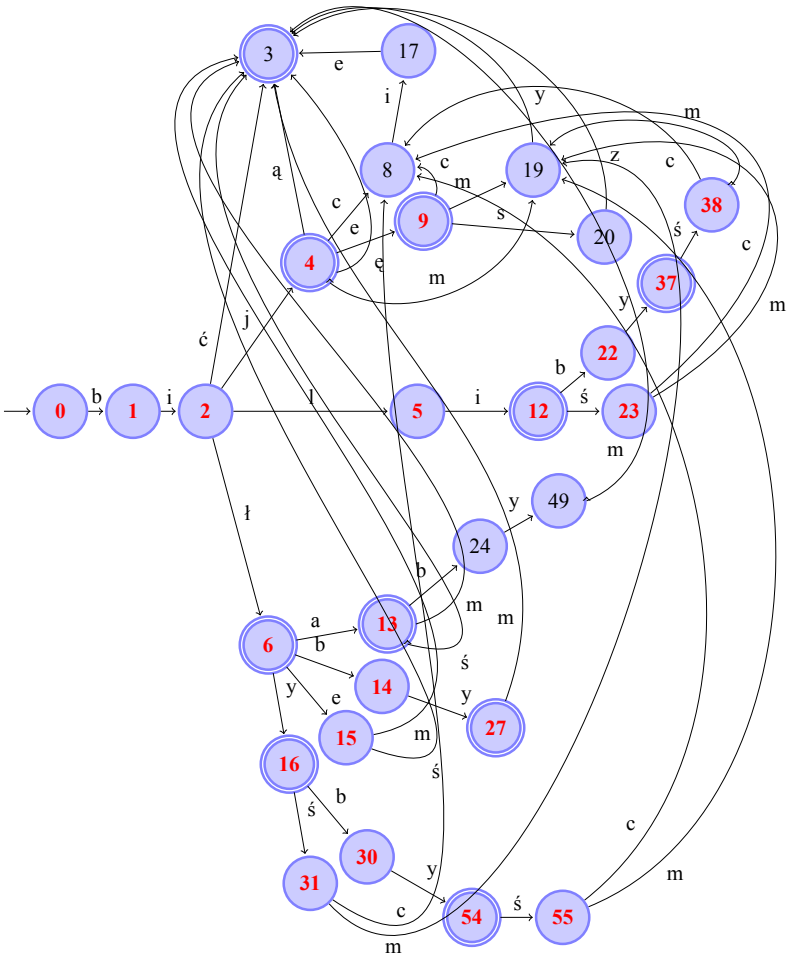


Figure 4.12: Automaton in Figure 4.11 during addition of word *biłby* using procedure PseudoAddStrUnsorted. State 24 has been cloned as state 14. State 49 has been cloned as state 27. Procedure PseudoLocalMinimization and function PseudoReplOrReg do not change the automaton.

Algorithm 4.5 Procedure PseudoAddStrUnsorted adds a string w to the language of an acyclic, pseudo-minimal automaton M. No assumption about previously added strings is made.

```
 1: procedure PseudoAddStrUnsorted(M, w)
 2:     q ← q_0
 3:     i ← 1
 4:     P ← ε                          ▷ path of non-confluence states in the common prefix
 5:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ ∧ FanIn(δ(q, w_i)) = 1 do
 6:         push q onto P
 7:         q ← δ(q, w_i)
 8:         i ← i + 1
 9:     end while
10:     p ← q
11:     j ← i
12:     R ← R \ {p}
13:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
14:         δ(q, w_i) ← Clone(δ(q, w_i))
15:         q ← δ(q, w_i)
16:         i ← i + 1
17:     end while
18:     while i ≤ |w| do
19:         δ(q, w_i) ← new state
20:         q ← δ(q, w_i)
21:         i ← i + 1
22:     end while
23:     F ← F ∪ {q}
24:     d ← false
25:     PseudoLocalMinimization(M, P, w, PseudoReplOrReg(M, p, w_{j...|w|}, d), p, j, d)
26: end procedure
```

p become state 6, variables i and j become 4. State 6 is not in the register, so it cannot be removed from there. In the second `while` loop, state 24 is cloned as state 14 with a transition labeled with *y* to state 49, and the transition from state 6 to state 24 labeled with *b* is redirected towards state 14. Variable q becomes state 14, and i becomes 5. In the next run of the loop, state 49 is cloned as state 27 (with a transition labeled with *m* to state 3), and the transition from state 14 to state 49 labeled with *y* is redirected towards state 27. Variable q becomes state 27, and i becomes 6. Since $|w| = 5$, the loop terminates, and state 27 is made final. Variable d is set to false. The situation is shown in Figure 4.12.

Function PseudoReplOrReg(M, p=6, w=*by*, d=*false*) is called. As w is not empty, it calls PseudoReplOrReg(M, p=14, w=*y*, d=*false*), which in turn calls PseudoReplOrReg(M, p=27, $w = \varepsilon$, d=*false*). As $w = \varepsilon$, no further recursive call is made. State 27 is final and it has one outgoing transition, so the condition for setting variable d is true, and d is set to true. As d is true, state 27 is not added to the register, and state 27 is returned. In the call one level up, parameter d is updated to *true*, so the condition on setting d to true does not matter --- d remains *true*. State 14 is returned. In the top level call to ReplOrReg, d is updated to *true*,

Algorithm 4.6 Procedure PseudoLocalMinimization. M is the automaton, P is the path recognizing the word w, s is the state returned by the top-level call to function ReplOrReg, q is the original state either the same as s or equivalent to it before it has replaced with s, i is the position in the path P and in the word w (an index on those structures), d is true when $|\overrightarrow{\mathcal{L}}(s)| > 1$.

```
1:  procedure PseudoLocalMinimization(M, P, w, s, q, i, d)
2:      while P not empty ∧ s ≠ q ∧ ¬d do
3:          pop p from P
4:          R ← R \ {p}
5:          δ(p, w_i) ← s
6:          i ← i − 1
7:          q ← p
8:          s ← p
9:          d ← FanIn(p) > 1
10:         if ¬d then
11:             if ∃r ∈ R : r ≡ p then
12:                 delete s
13:                 s ← r
14:             else
15:                 R ← R ∪ {p}
16:             end if
17:         end if
18:     end while
19: end procedure
```

and state 6 is returned by the function. So procedure PseudoLocalMinimization is called as PseudoLocalMinimization(M, $P = \{0, 1, 2\}$, w=*bił*, s=6, p=6, j=4, d=*false*). Since $s = p$, the `while` loop does not run and the procedure terminates immediately. Figure 4.12 is still up to date.

Now we add the form *biłaby*. In procedure PseudoAddStrUnsorted, q is set to state 0, i is set to 1, and the stack is emptied. In the first `while` loop, the whole word is consumed. States 0, 1, 2, 6, 13, and 24 are put onto the stack. Variables p and q are set to state 49, i and j are set to 7. State 49 is removed from the register. Since $i > |w|$, the second and the third `while` loops do not run. State 49 is made final, and d is set to *false*. Procedure PseudoLocalMinimization(M, $P = \{0, 1, 2, 6, 13, 24\}$, w=*biłaby*, PseudoReplOrReg(M,p=49,ε,d=*false*), p=49, j=7, d=*false*) is called. It calls function PseudoReplOrReg(M,p=49,ε,d=*false*) first. Since its third parameter w is a null string, no recursive call is made. State 49 has one outgoing transition, and it has just been made final, so d is set to *true*. Its right language is $\{\varepsilon, m\}$. State 49 is equivalent to state 27. As d is set to *true*, state 49 is returned by the function. It is not added to the register. Now procedure PseudoLocalMinimization can be called. Since $s = q = 49$, the `while` loop does not run, and the procedure terminates immediately. The resulting automaton is portrayed in Figure 4.13.

We now add *biłabyś*. The first loop in procedure PseudoAddStrUnsorted end with variables p and q set to state 49, i and j set to 7, states 0, 1, 2, 6, 13, and 24 put onto stack. An

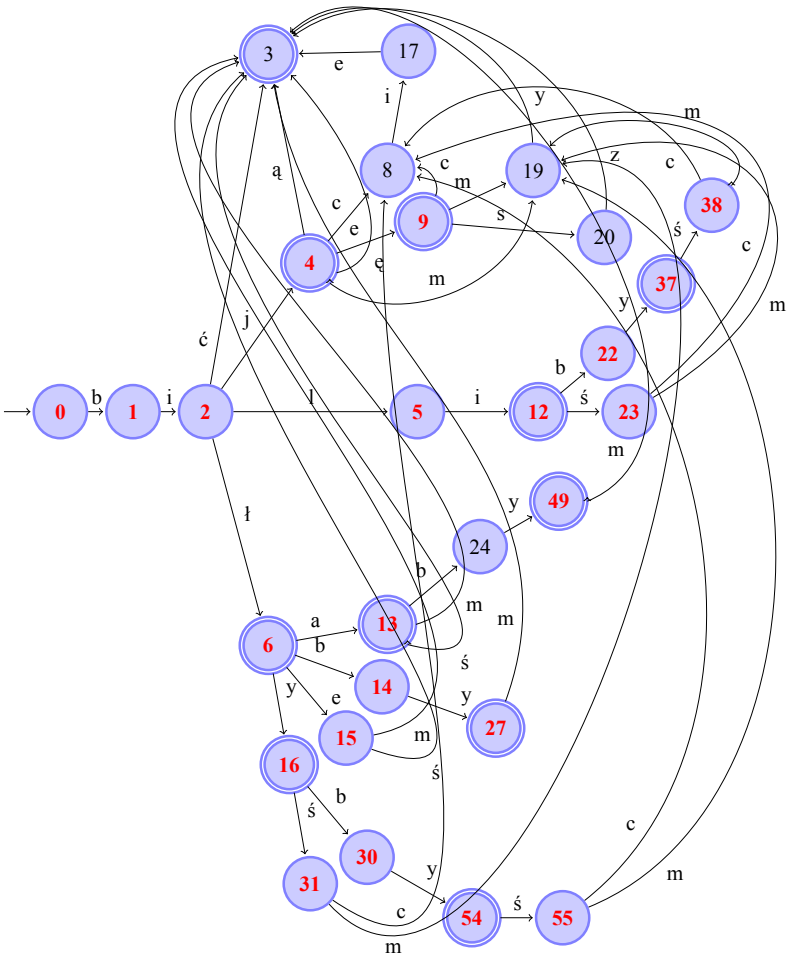


Figure 4.13: Automaton in Figure 4.12 during addition of word *bilaby* using procedure PseudoAddStrUnsorted. State 49 has been made final. Neither PseudoReplOrReg nor PseudoLocalMinimization change the structure of the automaton.

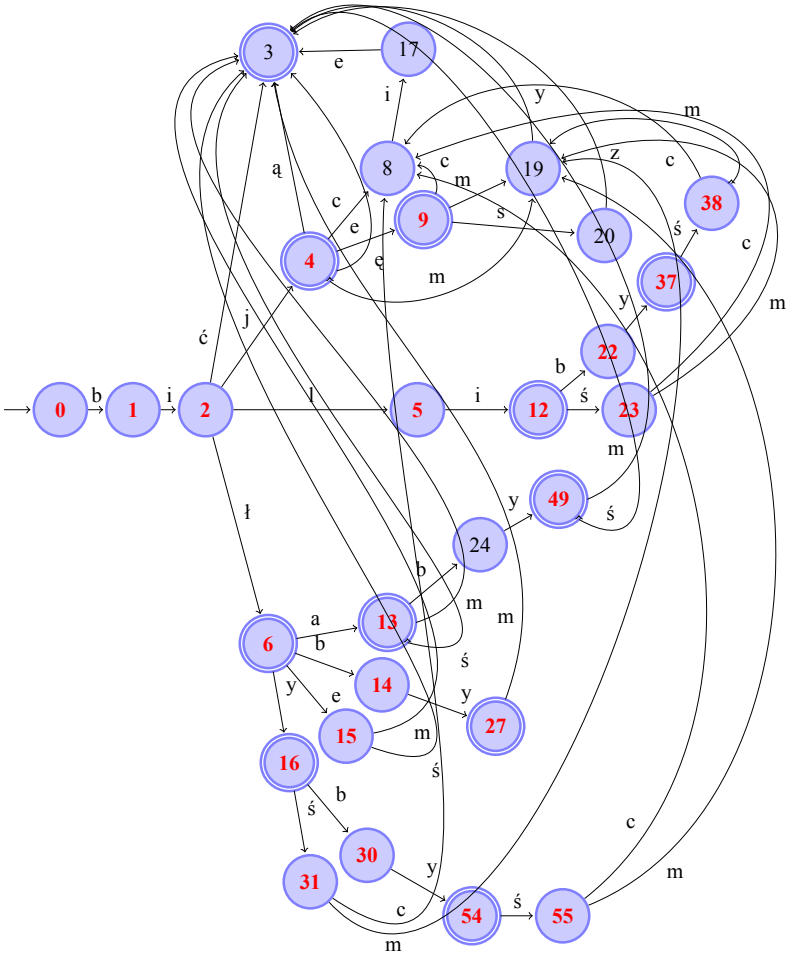


Figure 4.14: Automaton in Figure 4.13 after addition of word *biłabyś* using procedure PseudoAddStrUnsorted. Final state 56 was created, and a transition from state 49 to state 56 labeled with *ś* was created as well. A call to PseudoReplOrReg deleted state 56, and redirected its incoming transition to state 3. Procedure PseudoLocalMinimization again had nothing to do.

attempt to remove state 49 has been made, but state 49 is divergent, so it not in the register. The second `while` loop does not run as there are no confluence states along the path recognizing any prefix of *biłabyś*. The third `while` loop creates a new state 56, and a transition labeled *ś* from state 49 to state 56. State 56 is made final, and variable d is set to *false*. Function PseudoReplOrReg(M, 49, *ś*, *d=false*) is called to evaluate the fourth parameter of an invocation of procedure PseudoLocalMinimization. It calls itself as PseudoReplOrReg(M, 56, ε, *d=false*). As the third parameter is an empty string, no further recursive calls are made. Variable d remains *false* as state 56 has no outgoing transitions. Since state 3 is equivalent to state 56, and d is false, state 56 is deleted, and the transition from state 49 labeled with *ś* to state 56 is redirected to state 3 when the function returns to one level up. This time d is set to *true* as state 49 has now two outgoing transitions, and it is final. The state is not added to the register, and it returned by the function. Procedure PseudoLocalMinimization(M, $P = \{0, 1, 2, 6, 13, 24\}$, w=*biłabyś*, s=49, q=49, i=7, *d=true*) can now be executed. Since $s = q$, the `while` loop is not executed and the procedure terminates immediately. The resulting automaton is shown in Figure 4.14. Note that state 24 is divergent, and it remains in the register. It was put there when it was not a divergent state. Procedure PseudoLocalMinimization finished before it reached that state since state 24 is now divergent. Its presence in the register does not harm the algorithm.

The next word to be added is *biłbyś*. In the first `while` loop of procedure PseudoAddStrUnsorted, states 0, 1, 2, 6, and 14 are put onto the stack P, variables p and q are set to state 27, i and j are set to 6, and an attempt is made to remove (divergent) state 27 from the register. The second `while` loop does not run. The third `while` loop creates a new state 57, and a transition from state 27 to state 57 labeled with *ś*. Variable q becomes state 57, i is set to 7, state 57 becomes final, and d is set to *false*. Function PseudoReplOrReg(M, 27, *ś*, *d=false*) is called, while calls itself again as PseudoReplOrReg(M, 57, ε, *d=false*). In the nested call, recursion ends. As state 57 has no outgoing transitions, d remains false. State 3 is found to be equivalent to state 57, state 57 is deleted, and state 3 is returned. In the upper level invocation of PseudoReplOrReg, the incoming transition of state 57 is redirected to state 3. As state q=27 is final, and it has two outgoing transitions, parameter d is set to true. State 27 has an equivalent state 49, but as state 27 is divergent (and so is state 49), state 27 is not deleted. The function returns state 27. Procedure PseudoLocalMinimization(M, $P = \{0, 1, 2, 6, 16\}$,w=*biłbyś*,s=27,q=27,i=6,*d=true*) is called. Since $s = q = 27$, the `while` loop does not run, and the procedure terminates. The resulting automaton is shown in Figure 4.15.

To prove that the algorithm is correct, we need to prove that procedure PseudoAddStrUnsorted is correct. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a pseudo-minimal automaton before an invocation of PseudoAddStrUnsorted, R -- the register before the invocation, w -- the string being added, M' -- the automaton after the invocation, and R' -- the register after the invocation. We assume that the register contains all non-divergent states. It can also contain divergent states. We must show that:

1. $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$
2. ($\forall q \in Q'$ such that q is not divergent) $q \in R'$
3. M' is pseudo-minimal.

To prove the first statement, we note that the only possibility of adding anything to the

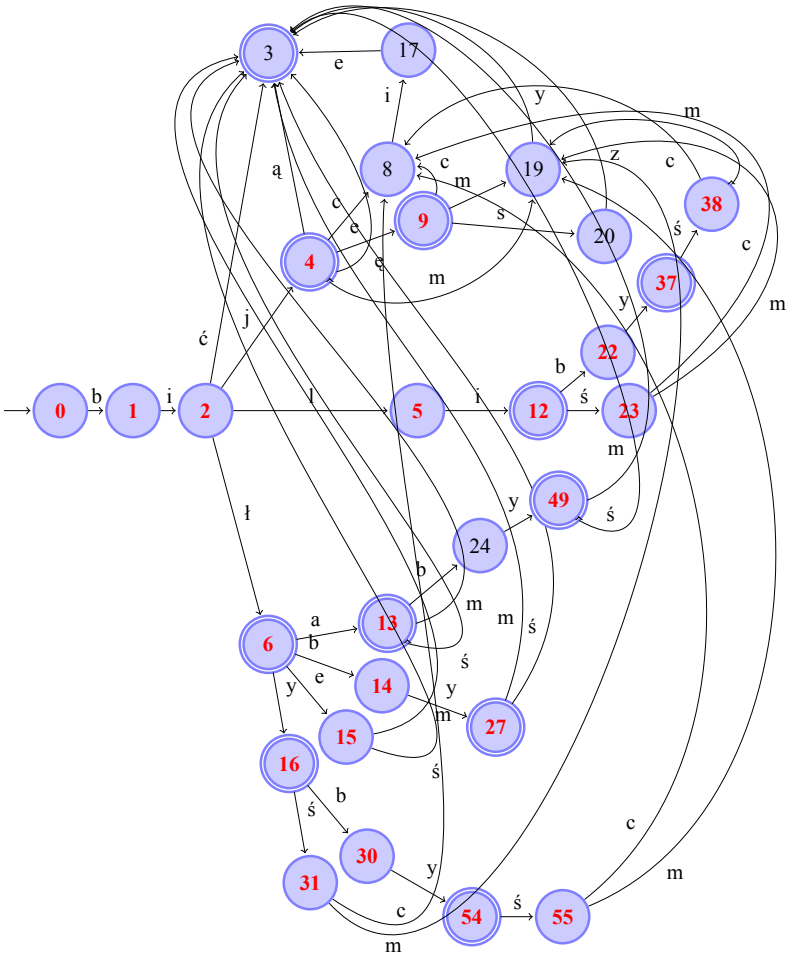


Figure 4.15: Pseudo-minimal automaton recognizing the same language as the trie automaton in Figure 3.1. A copy of Figure 3.21.

language of the automaton exists in the third loop of procedure PseudoAddStrUnsorted, and making the last state in the path recognizing word w under investigation final. That possibility consists of creating a new outgoing transition for the last state in the longest common prefix path of w and some word already in the language of the automaton (if the third loop is executed, i.e. if w is not a prefix of any word already recognized by the automaton), and by making the last state in the path of w final. Since cloning confluence states in the second `while` loop in the procedure makes sure no confluence state is in the path of w when the last state is made final, and that state is $\delta^*(q_0, w)$, then the only word that is added to the language of the automaton is w.

The second point is important, because it allows us to replace states with their pseudo-equivalent counterparts. The register can also contain divergent states. Some non-divergent states may become divergent during processing, and the local minimization may stop before examining them, as it only uses their signatures, and not their right languages. We have seen such a situation in the example above.

The states that are not in the register are either newly created states (brand new or cloned), or states removed from the register. All new states are created in the third `while` loop of procedure PseudoAddStrUnsorted while adding a new word, and they are all processed by function PseudoReplOrReg. They are put into the register unless there exists an equivalent state already in the register. The same happens with cloned states. The state removed from the register in line 12 of procedure PseudoAddStrUnsorted is handled by function PseudoReplOrReg in exactly the same way. Other states are removed from the register in procedure PseudoLocalMinimization. When a state is removed from the register in that procedure, then either:

1. it is divergent, so it should not be in the register,
2. it is non-divergent, and it has an equivalent in the register, so it is deleted,
3. it is non-divergent, and it has no equivalent in the register, so it is put into it.

There may be doubts whether an equivalent state can always be found, as inequality of signatures no longer implies inequivalence. However, that implication still holds for non-divergent states, and only such states are put into the register, and only such states are search for there.

Point three means that all states are reachable and co-reachable, and that all non-divergent states have different right languages. States are created in procedure PseudoAddStrUnsorted, and that creation is done as a part of creation of a new transition reaching the new state. A state may become unreachable by redirecting its incoming transition towards another state. This can be done either in function PseudoReplOrReg, or in procedure PseudoLocalMinimization, or in the second `while` loop in procedure PseudoAddStrUnsorted. In the first two cases, the state has a single incoming transition, and it is deleted. In the third case, the confluence state is cloned, one of its incoming transitions is redirected, but its other incoming transitions stay intact, so the state remains reachable.

Every newly created state is checked against all states in the register in function PseudoReplOrReg. If the states are equivalent, the new state is replaced by an old one. Existing states can either change their signature by receiving a new outgoing transition in the first run of the third `while` loop of procedure PseudoAddStrUnsorted, or change their signature because states directly reachable from them change their signature. In the first case, they are handled exactly like the new states, i.e. by function PseudoReplOrReg. In the second case,

they are handled by procedure PseudoLocalMinimization. The procedure looks for equivalent states in the register only for non-divergent states. A visit to a divergent state terminates the procedure. It is correct since divergent states cannot be preceded by non-divergent states. Equivalent states are replaced with their counterparts found in the register.

The `while` loop in function PseudoUnsortedIncrementalConstruction runs n times, where n is the number of words on the input. Inside the loop, there is reading a word from the input --- running in time $\mathcal{O}(|w_{max}|)$, where w_{max} is the longest word on the input, and a call to procedure PseudoAddStrUnsorted. Inside the procedure, there is a removal from the register, making a state final, a call to procedure PseudoLocalMinimization, a few simple assignments, and three `while` loops. We assume that all register operations run in $\mathcal{O}(1)$, i.e. in constant time. Making a state final means changing a flag associated with the state --- a constant time operation. The first `while` loop contains putting a state onto stack, traversing a transition, and incrementing a counter --- all constant time operations. The second `while` loop also contains cloning, which can be done in constant time (strictly speaking: $\mathcal{O}(|\Sigma|)$). The third `while` loop contains creation of a new state --- also $\mathcal{O}(1)$ operation. Function PseudoReplOrReg is called to evaluate an actual parameter for an invocation of procedure PseudoLocalMinimization. The function calls itself at most $|w_{max}|$ times. In each call, a transition can be redirected, the number of outgoing transitions and finality of a state is checked a state can be deleted, and a state can be put into the register. They are all constant time operations. Also, to invoke the function recursively, a transition is traversed, and a string is shortened at the beginning. These are also constant time operations. Therefore, all recursive calls to PseudoReplOrReg take $\mathcal{O}(|w_{max}|)$ time. The `while` loop in procedure PseudoLocalMinimization runs at most $|w_{max}|$ times. All operations inside the loop are constant time. Thus the whole algorithm runs in time $\mathcal{O}(n|w_{max}|)$. If the assumption about constant time register operations does not hold, that factor should be included in the estimation.

This algorithm was proposed by Jan Daciuk, Denis Maurel, and Agata Savary in [17].

4.3 Extension to Cyclic Automata

The algorithm for unsorted data can easily be extended so that it adds strings or words to a minimal cyclic automaton. The extension is done in a similar way to the one that was used to extend the algorithm for sorted data. Actually, the extension of the unsorted data algorithm was done first. This algorithm (Algorithm 4.7) is more flexible than the one for sorted data, but it is slower.

Algorithm 4.7 Function UnsortedCyclIncrConstruction adds unsorted words to a language of a minimal automaton M using information from the register R and words to be added from the input. Both M and R are modified by the function, and M is returned.

```
1: function UnsortedCyclIncrConstruction(M, R)
2:     while input not empty do
3:         w ← next word
4:         AddStrCyclUnsorted(M, w)
5:     end while
6:     return M;
7: end function
```

When comparing it to function SortedCyclIncrConstruction, one can see the absence of cloning of the initial state, and then the absence of deletion of the old initial state when it becomes unreachable. However, the idea remains the same. The missing parts have been moved to procedure AddStrCyclUnsorted, because the automaton is minimized till the end every time a new string is added to its language. That procedure looks like a simplified version of procedure AddStrUnsorted. The first `while` loop is gone, and invocation of procedure LocalMinimization is gone, but a call to ReplOrReg remains, the stack is gone. The first `while` loop is gone, because when the initial state is cloned, its outgoing transitions are cloned, and the targets of those transitions become automatically confluence states. Since the stack recorded non-confluence part of the common prefix, it is no longer needed. As the clone of the initial state is not yet in the register, it does not need to be removed from there. Procedure LocalMinimization was used to process the same part, therefore is not needed either. Procedure DeleteBranchUp is given as Algorithm 3.14 on page 51.

Algorithm 4.8 Procedure AddStrCyclUnsorted adds a string w to the language of a possibly cyclic automaton M. No assumption about previously added strings is made.

```
 1: procedure AddStrCyclUnsorted(M, w)
 2:     q' ← q_0
 3:     q_0 ← Clone(q_0)
 4:     q ← q_0
 5:     i ← 1
 6:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
 7:         δ(q, w_i) ← Clone(δ(q, w_i))
 8:         q ← δ(q, w_i)
 9:         i ← i + 1
10:     end while
11:     while i ≤ |w| do
12:         δ(q, w_i) ← new state
13:         q ← δ(q, w_i)
14:         i ← i + 1
15:     end while
16:     F ← F ∪ {q}
17:     q_0 ← ReplOrReg(M, q_0, w)
18:     if q' ≠ q_0 then
19:         Increment incoming transition counter for q'
20:         DeleteBranchUp(M, q')
21:     end if
22: end procedure
```

Algorithm 4.8 is given here as in [9]. However, it can be further optimized. Such optimization speeds up the algorithm, but it also makes it more complex. It is based on an observation that when the initial state has no incoming transitions, it does not have to be cloned. Procedure AddStrUnsorted (Algorithm 4.2, page 65) is sufficient to add a string in such case. The optimized version of procedure AddStrCyclUnsorted is given as Algorithm 4.9. We will use the optimized version from this point on.

Let us see how the algorithm works in practice. Figure 4.16 shows a copy of Figure 3.35

Algorithm 4.9 Optimized version of procedure AddStrCyclUnsorted adds a string w to the language of a possibly cyclic automaton M. No assumption about previously added strings is made. The initial state is cloned only if it has incoming transitions.

```
 1: procedure AddStrCyclUnsorted(M, w)
 2:     if FanIn(q_0) = 0 then
 3:         AddStrUnsorted(M, w)
 4:     else
 5:         q' ← q_0
 6:         q_0 ← Clone(q_0)
 7:         q ← q_0
 8:         i ← 1
 9:         while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
10:             δ(q, w_i) ← Clone(δ(q, w_i))
11:             q ← δ(q, w_i)
12:             i ← i + 1
13:         end while
14:         while i ≤ |w| do
15:             δ(q, w_i) ← new state
16:             q ← δ(q, w_i)
17:             i ← i + 1
18:         end while
19:         F ← F ∪ {q}
20:         q_0 ← ReplOrReg(M, q_0, w)
21:         if q' ≠ q_0 then
22:             Increment incoming transition counter for q'
23:             DeleteBranchUp(M, q')
24:         end if
25:     end if
26: end procedure
```

on page 58. It is an automaton that recognizes simplified host names in URLs. To show the differences between this algorithm and the algorithm for sorted data, we add words *cat, cow*, and *dog*.

We start with *cat*. Function UnsortedCyclIncrConstruction reads *cat* from the input, and it calls procedure AddStrCyclUnsorted. The initial state has incoming transitions. The old initial state 0 is stored in variable q', and state 0 is cloned as state 6 --- the new initial state. Variable q is set to state 6, i is set to 1. State 1 is cloned as state 7, and the transition from state 6 to state 7 labeled with *c* is redirected to state 7. Variable q becomes state 7, and i is incremented to 2. The situation is shown in Figure 4.17.

In the next run of the first `while` loop, state 1 is cloned again as state 8. Variable q becomes state 8, and i is incremented to 3. The transition from state 7 to state 1 labeled with *a* (remember that a transition labeled *a-z* in the figures is actually a set of 26 transitions) is redirected towards state 8. In the third run, state 1 is cloned yet again as state 9, q becomes state 9, and i becomes 4. A transition labeled with *t* from state 8 to state 1 is redirected towards state 9. Since $4 = i > |w| = 3$, the second `while` loop is not entered. State 9 becomes final

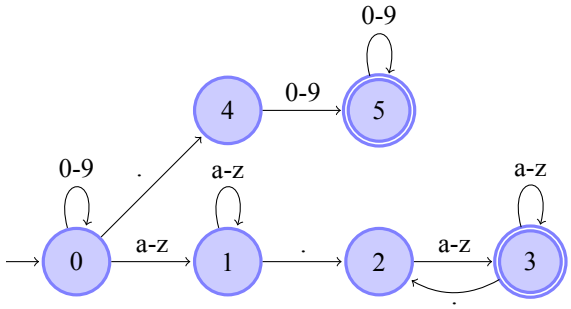


Figure 4.16: A minimal automaton recognizing simplified host names in URLs following the pattern $[a-z]^+(\.[a-z]^+)^+$, and recognizing real numbers in format $[0-9]^*\.[0-9]^+$. A transition labeled with *a-z* stands for many transitions labeled with consecutive letters *a, b, c*...*z*. A copy of Figure 3.35.

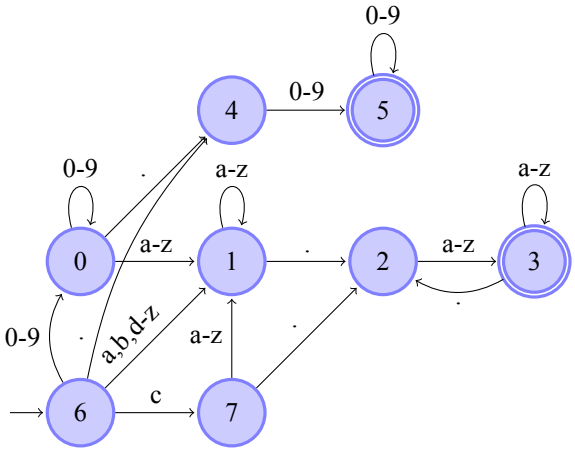


Figure 4.17: The automaton from Figure 4.16 during addition of word *cat*. State 1 has just been cloned as state 7 while consuming letter *c*.

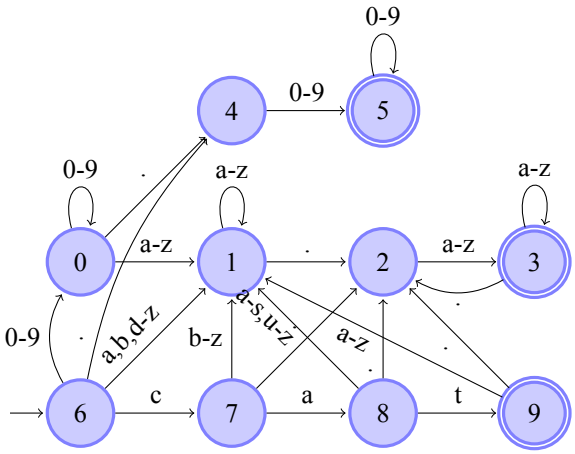


Figure 4.18: The automaton from Figure 4.16 during addition of word *cat*. State 1 has just been cloned as state 9 while consuming letter *t*.

as shown in Figure 4.18.

Function ReplOrReg(M, 6, *cat*) is called. It calls itself until state 9 is reached, and parameter w is an empty string. There is no state in the register that would be equivalent to state 9, so state 9 is added to the register and returned by the function to an upper-level invocation. State 8 is also found to be unique, so it is added to the register and returned by the function. The same happens with states 7 and 6. As q_0 is now state 6, and it is different from the previous initial state q', which is state 0, the incoming transition counter for state 0 is incremented and procedure DeleteBranchUp(M, 0) is called. The counter is decremented. As it is not equal to zero DeleteBranchUp is finished with no state deleted. Figure 4.18 is still valid.

It is now time to add *cow*. State 6 has no incoming transitions, so AddStrUnsorted is called. Variable q becomes state 6, i becomes 1, and the stack P is emptied. State 7 is not confluence, so the first `while` loop is entered. State 6 is put onto stack. Variable q becomes state 7, i becomes 2. As the transition from state 7 labeled with *o* points to confluence state 1, the loop ends. Variable p becomes state 7, j becomes 2, and state 7 is removed from the register. In the second `while` loop, state 1 is cloned as state 10. Transition from state 7 labeled with *o* to state 1 is redirected to state 10, and i is incremented to to 3. The loop runs once again. State 1 is cloned again as state 11, and the transition from state 10 to state 1 labeled with w is redirected to state 11. Variable q becomes state 11, i reaches 4, and the loop terminates. The third `while` loop is not entered as i points past the end of the word. State 11 becomes final. The resulting automaton is shown in Figure 4.19. For comparison, Figure 4.20 shows what the unoptimized version would build at this point (before minimization).

Function ReplOrReg is called to evaluate a parameter for LocalMinimization. It is called as ReplOrReg(M, 7, *ow*), which calls ReplOrReg(M, 10, w), which calls ReplOrReg(M, 11, ε). State 11 is found to be equivalent to state 9, so state 11 is deleted, and state 9 is returned, becoming the target of the transition from state 10 labeled with w one level up in the call hierarchy. State 10 is unique, so it is added to the register and returned by the function. State 7 is also unique, so it is added to the register and returned by the top-level invocation of

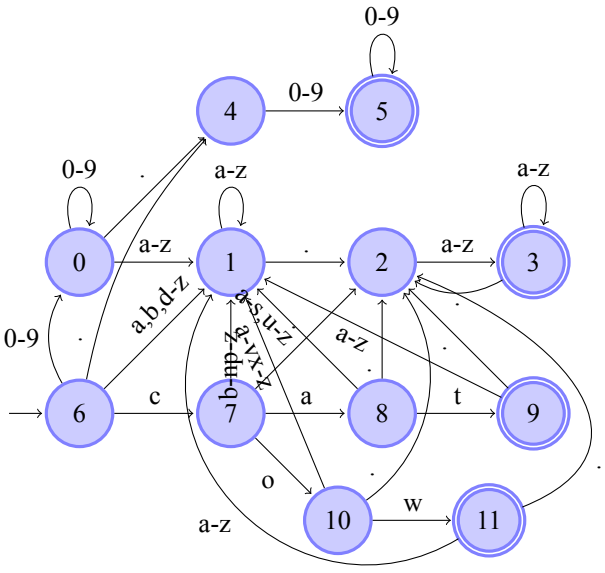


Figure 4.19: The automaton from Figure 4.16 during addition of word *cow*. State 1 has just been cloned as state 11 while consuming letter *w*.

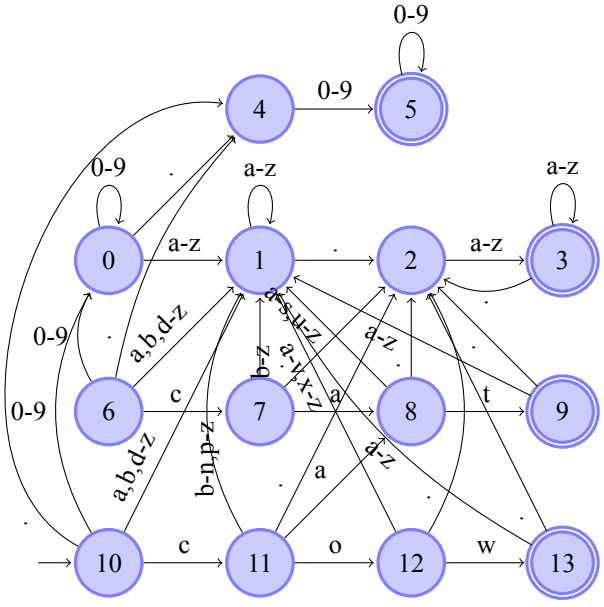


Figure 4.20: The automaton from Figure 4.16 during addition of word *ow* just before ReplOrReg is called. Unoptimized version of the algorithm. Compare Figure 4.19.

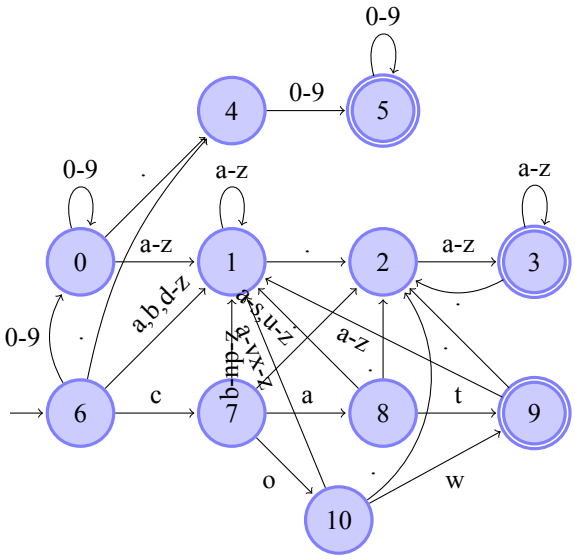


Figure 4.21: The automaton from Figure 4.16 after word *cow* has been added. State 11 has been merged with state 9.

function ReplOrReg. Procedure LocalMinimization(M, $P = \{6\}$, w=*cow*, $s = 7$, $q = 7$, $i = 2$) is called. The procedure is given as Algorithm 4.3 on page 66. Since q and s are the same state, the loop in the procedure is not entered, and the procedure ends. This concludes procedure AddStrUnsorted, and then AddStrCyclUnsorted, so the control returns to function UnsortedCyclIncrConstruction. The resulting automaton is shown in Figure 4.21.

The last word read is *dog*. In function AddStrCyclUnsorted, the initial state 6 has no incoming transitions, so procedure AddStrUnsorted is called again to handle addition of this word. Variable q is set to state 6, i is set to 1, and the stack is emptied. A transition labeled with d points to state 1, which is a confluence state, so the first `while` loop is not entered. Variable p is set to state 6, j is set to 1, and state 6 is removed from the register. In the second `while` loop, state 1 is cloned as state 11, and a transition labeled with d from state 6 to state 1 is redirected to state 11. Variable q is set to state 11, and i is incremented to 2. State 1 is cloned again, this time as state 12, and the transition from state 11 to state 1 labeled with o is redirected to state 12. Variable q is set to state 12, and i is set to 3. This time state 1 is cloned as state 13, and a transition from state 12 to state 1 labeled with g is redirected to state 13. Variable q becomes state 13, and i becomes 4. Since index i points now past the end of the word, the loop terminates, and the third `while` loop is not entered. State 13 is made final. Function ReplOrReg(M, 6, *dog*) is called to evaluate the fourth actual parameter of procedure LocalMinimization. The function calls itself as ReplOrReg(M, 11, *og*), then ReplOrReg(M, 12, g), and finally as ReplOrReg(M, 13, ε). State 13 is found equivalent to state 9, so state 13 is deleted, and state 9 is returned. In the upper-level call, a transition from state 12 to state 13 labeled with g is redirected towards state 9. State 12 is found to be unique, so it is returned by the function. One level up, state 11 is found to be unique, and it is returned by the function. Procedure LocalMinimization(M, $P = \{\}$, w=*dog*, $s = 11$, $q = 11$, $i = 1$) is called, and

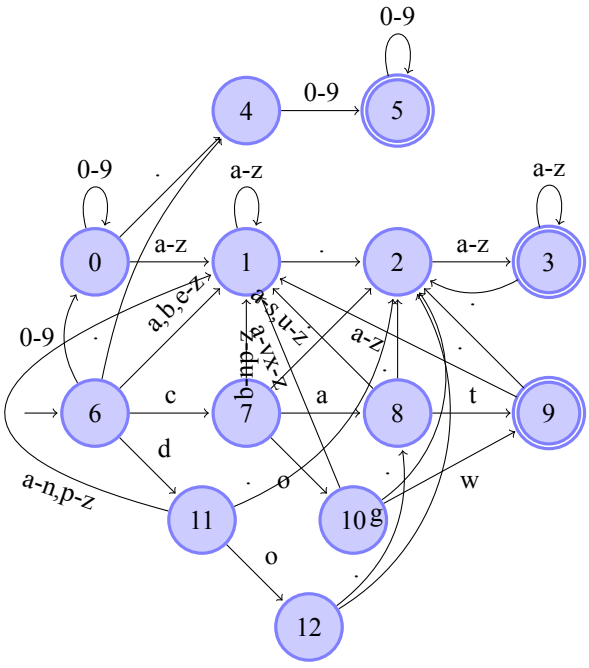


Figure 4.22: The automaton from Figure 4.16 after word *dog* has been added. State 11 has been merged with state 9. Compare with Figure 3.41.

it finishes immediately, as P is an empty stack, and $s = q = 6$. The resulting automaton is depicted in Figure 4.22.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an automaton before an invocation of function AddStrCyclUnsorted, R --- the register before the call, $M' = (Q', \Sigma, \delta', q'_0, F')$ --- the automaton after the call, R' --- the register after the call, w --- the word to be added to the language of the automaton during the call (a parameter in the call). We assume that M is minimal, and $Q = R$. There are two cases. When q_0 has no incoming transitions, procedure AddStrUnsorted does all the job, and the proof of correctness for that function also holds here. To prove that the algorithm is correct when q_0 does have incoming transitions, one needs to show that:

1. $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$
2. $R' = Q'$
3. M' is minimal

To prove the first point, let us notice that cloning the initial state and making the clone become the new initial state does not change the language of the automaton. What it does do is that it makes sure that the cardinality of the left language is exactly 1. This means that adding a new outgoing transition to the initial state or to any state reachable from the initial state such that the path from q_0 to that state (including the state itself) does not include any confluence states does not change the right language of any state outside that path. The first `while` loop

in procedure AddStrCyclUnsorted ensures that there are no confluence states on that path. The loop does not change the language of the automaton as the clones have a right language identical to that of the cloned states. Let $w = uv$ such that $\delta^*(q_0, u)$ be the value of variable q after the first `while` loop. Let us name that state q'. The second `while` loop creates a chain of states and single transitions between them ending in a state q'' such that $\delta^*(q', w) = q''$. State q'' is made final. As $\overleftarrow{\mathcal{L}}(q'') = \{w\}$, $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$ before a call to function ReplOrReg. Function ReplOrReg can delete states, but as proved several times elsewhere in this book, it does not change the language of the automaton. Neither creates it any unreachable states. This concludes the proof of the first point.

As to the second point, just before the top-level invocation of function ReplOrReg, all states lying on the path recognizing w from q_0 to $\delta^*(q_0, w)$ are the only states not in the register. Function ReplOrReg either deletes them or puts them into the register, one by one, starting from $\delta^*(q_0, w)$. The previous initial state may becomes unreachable. In that case, it is removed from the register before its deletion. The same happens with other states that become unreachable in procedure DeleteBranchUp. Therefore, at the end of the process, $R' = Q'$.

Procedure AddStrCyclUnsorted does not create any unreachable states. In case of the original initial state, it is explicitly checked at the end of the procedure whether it became unreachable. If that is the case, the state is deleted, and so are other state that are deprived of incoming transitions by deleting the original initial state or states reachable from it. As at the end of processing $R' = Q'$, states are put into the register when they have a unique right language, their signature is never modified without removing them from the register, and the processing of paths is done in postorder, M' is minimal.

Function UnsortedCyclIncrConstruction has a `while` loop that runs n times, where n is the number of words on the input. Each time, it calls procedure AddStrCyclUnsorted. The beginning of the procedure contains a conditional branch with the condition being a check on the number of incoming transitions of a state. Since this is implemented with a counter associated with each state, it is done in constant time. If the condition is *true*, procedure AddStrUnsorted is called. As proved earlier (page 78), that procedure runs in $\mathcal{O}(|w_{max}|)$ time. If the condition is *false*, there follow a few simple assignments with a call to function Clone that runs in constant time. Then there is a `while` loop that runs at most $|w_{max}|$ times, and which contains simple assignments, a Clone invocation, and single transition traversals --- all running in constant time. The condition for the loop is also checked in constant time (instead of checking literally $i < |w|$, we rather check an end-of-string marker in w). The second `while` loop is similar, but instead of cloning, it contains a creation of a new state, also a constant-time operation. Making a state final is done with a flag on the state --- a constant-time operation. All recursive invocations of function ReplOrReg in one run of procedure AddStrCyclUnsorted take $\mathcal{O}(|w_{max}|)$ time as proved earlier. Since unreachable states could only be created by cloning existing states and redirecting incoming transitions off the cloned states, function DeleteBranchUp can be called up to $|w_{max}||\Sigma|$ times. Treating $|\Sigma|$ as constant, as it is usual in automata theory, we get $\mathcal{O}(n|w_{max}|)$ for the whole algorithm.

This algorithm was prosed by Rafael Carrasco and Mikel Forcada in [9]. The optimized version was proposed by the author of this book.

4.4 Extension to Minimal Bottom-Up Tree Automata

An extension of the incremental construction algorithm to deterministic, bottom-up tree automata is complicated by the fact, that the items that are added to automata are trees, and not strings, and that the automata are tree automata. Trees are not linear, so it is difficult to arrange their processing in the same, orderly way as it was with strings. `While` loops following each other in the version for strings are replaced with recursion. The main function for the algorithm, function UnsortedDTAconstruction is given as Algorithm 4.10. There is nothing peculiar here; trees are read one by one, and function AddTree is called for each of them. Reading a tree is, however, more complicated. We will not go into details of that process.

Algorithm 4.10 Function UnsortedDTAconstruction adds trees to the language of a DTA A. Trees are read from the standard input. A register R initially contains all states of A. Both A and R are modified by the function, so that at the end, $|Q| = |R|$. The function returns the minimal automaton that recognizes all trees recognized by the original automaton and all trees read from the input.

```
1: function UnsortedDTAconstruction(A, R)
2:     Create an empty automaton A
3:     R ← ∅
4:     while input not empty do
5:         t ← next tree
6:         (A, R) ← AddTree(A,R,t)
7:     end while
8: end function
```

Function AddTree, given as Algorithm 4.11, has two parts. They are more clearly articulated here than in previous algorithms. In the forward step, a tree is added to the language of the automaton. This is achieved by a call to function Split, and making $\delta_A(t)$ final. In the backward step, the automaton is minimized by invocation of function LocMin. Function AddTree uses two additional variables: Θ and Ω. The first one holds a set of states that are either new or modified (their signature changes) in the current call of AddTree. The second one is a stack of states visited during addition of the current tree t. It plays almost the same role as variable P in Algorithm 4.2 (page 65).

Function Split adds a tree t to the language of the automaton, except for making $\delta_A(t)$ final. Making $\delta_A(t)$ final is outside of function Split, as the function is used recursively to follow or create paths for the subtrees of t. The tree t is decomposed into subtrees $t_1, \ldots, t_m$. For each of them $\delta_A(t_k)$ is returned by a nested call to Split, and stored in the variable r_k. Recursion ends as the tree is finite, and leaves of the tree have $m = 0$. Recursion replaces three `while` loops present in procedure AddStrUnsorted (Algorithm 4.2, page 65). It only replaces the loops as control structures. The contents of the loops is placed after the recursive calls. The loops are organized sequentially. In function Split, the choice which loop contents to execute is made in each call separately. There are three cases. They correspond to the three loops. The first case is when there is a transition from $(r_1, \ldots, r_k)$ labeled with σ to a state q, and q has a single incoming transition. This case corresponds to the first `while` loop contents. The only action taken then is storing the single incoming transition in variable B_q. The second case is when state q has more than one incoming transition. This corresponds

Algorithm 4.11 Function AddTree adds a tree t to the language of a minimal DTA A. All states of the DTA are in the register R before the call. The procedure modifies A, so that it recognizes the original language and the tree t, and it also modifies R so that it contains all states of the new automaton. Variable Θ holds a set of states that are new or that changes their signature. A stack Ω holds the same states as θ in chronological order of those visits. It may later be changed by LocMin.

```
 1: function AddTree(A, R, t)
 2:     Θ ← ∅
 3:     Ω ← ∅
 4:     q ← Split(A, R, Θ, Ω, t)
 5:     R ← R \ {q}
 6:     F ← F ∪ {q}
 7:     Θ ← Θ ∪ {q}
 8:     push q onto Ω
 9:     A ← LocMin(A, R, Θ, Ω)
10:     return (A, R)
11: end function
```

to the second `while` loop, and state q needs to be cloned. The transition going to q must be redirected to state n, i.e. to the clone of state q. When no transition from $(r_1, \ldots, r_m)$ labeled with σ exists, it needs to be built, just like it is done in the third loop in procedure AddStrUnsorted. In the last two cases, the source states $r_1, \ldots, r_m$ of the transition being traversed (either new or modified) are added to the set Θ and are removed from the register R. In all cases the traversed transition is stored in B_q, so that it is possible to go back from the state. Also, the source states are put onto stack. Note that all states target visited by function Split have single incoming transitions. When the target state has more incoming transitions, it is cloned and the clone receives a single incoming transition.

Cloning a state in a DTA is more complicated than in a DFA. This stems from the fact that the same state can be present several times in the same transition.

Let us take an example. Figure 4.23 shows a minimal DTA recognizing trees *a(a,a), a(a,b), a(b,a)*, and *a(b,b)*. The language of state 2 is $L_A(1) = \{a, b\}$.

Suppose we want to add *b(b,b)* to the language of the DTA. Since *b* is a subtree of both the tree we want to add, and of three of four trees already recognized by the automaton, we need to clone state 1. A naive, intuitive way of cloning would be to create another state 3 with one outgoing transition labeled *a*, and with two source states being both the clone of state 1. Then we create another transition labeled *b* with both source states being state 3 and leading to a new state 4, which we made final. The situation is shown in Figure 4.24.

Note that the DTA in Figure 4.24 does not recognize trees *a(a,b)* and *a(b,a)* anymore. Let us investigate why this is so. In the DTA in Figure 4.23, $L_A(1) = \{a, b\}$. When we create an exact copy of state 1 and store it as state 3, and make an exact copy of transition t1 with state 3 replacing state 1 in both positions, and store it as transition t2, and then we redirect a transition with label *b* and no source states so that instead of going to state 1, it goes to state 3, then $L_A(1) = \{a\}$, and $L_A(3) = \{b\}$. So in both $(a, 1, 1, 2)$ and $(a, 3, 3, 2)$ both subtrees must be exactly the same. However, the automaton in Figure 4.23 recognizes the same language as the DTA in Figure 4.25. In the latter figure, states 1 and 3 are equivalent.

Algorithm 4.12 Function Split adds a tree t to the language of an automaton A without making $\delta_A(t)$ final. The register R contains all states in the minimized part of the automaton. The set Θ contains all states that need to be submitted to minimization. The function returns $\delta_A(t)$. The automaton A, the register R, the set Θ, and the stack Ω are modified inside the function.

```
1:  function Split(A, R, Θ, Ω, t = σ(t_1, ..., t_m))
2:      for k ∈ 1, ..., m do
3:          r_k ← Split(A, R, Θ, Ω, t_k)
4:      end for
5:      q ← δ_m(σ, r_1, ..., r_m)
6:      if q = ⊥ ∨ C_q ≠ 1 then
7:          if q = ⊥ then
8:              n ← new state
9:              Δ ← Δ ∪ {(σ, r_1, ..., r_n, n)}
10:             C_n ← 1
11:         else
12:             n ← DTAClone(A, q)
13:             Δ ← Δ \ {(σ, r_1, ..., r_m, q)} ∪ {(σ, r_1, ..., r_m, n)}
14:             C_q ← C_q − 1
15:             C_n ← 1
16:         end if
17:         q ← n
18:         for i ∈ 1, ..., m do
19:             if r_i ∉ Θ then
20:                 Θ ← Θ ∪ {r_i}
21:                 R ← R \ {r_i}
22:             end if
23:         end for
24:     end if
25:     push r_1, ..., r_m onto Ω
26:     B_q ← (σ, r_1, ..., r_m, q)
27:     return q
28: end function
```

Algorithm 4.13 Function DTAClone clones a state q in a DTA A.

```
1:  function DTAClone(A, q)
2:      n ← new state
3:      for (σ, q_1, ..., q_m, p) ∈ Δ such that m > 0, k ≤ m, q = q_k and
        δ(σ, q_1, ..., q_{k−1}, n, q_{k+1}, ..., q_m, p) = ⊥ do
4:          Δ ← Δ ∪ {(σ, q_1, ..., q_{k−1}, n, q_{k+1}, ..., q_m, p)}
5:          C_p ← C_p + 1
6:      end for
7:      if q ∈ F then
8:          F ← F ∪ {n}
9:      end if
10: end function
```

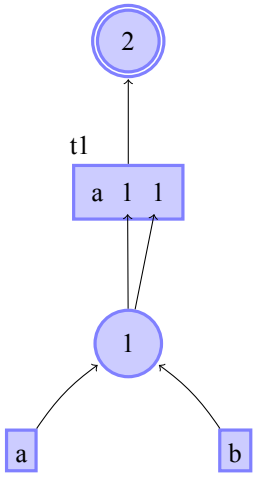


Figure 4.23: A deterministic bottom-up tree automaton recognizing a tree language consisting of trees *a(a,a), a(a,b), a(b,a)*, and *a(b,b)*.

A conclusion that comes from the example is that in cloning, a new state (a clone) is created, and then for each transition with the cloned state anywhere among source states, transitions with all combinations of the cloned state and the clone in positions of the cloned state must be added. Since function has a `while` loop that goes over all transitions in Δ, and new transitions augment Δ, all appropriate transitions are created. The minimal automaton recognizing trees *a(a,a), a(a,b), a(b,a), a(b,b)*, and *b(b,b)* is shown in Figure 4.26.

Local minimization, invoked in line 9 of function AddTree on page 99, is performed by function LocMin, presented here as Algorithm 4.14.

In function LocMin, we go back from the root to the leaves, looking for equivalent states. The stack Ω sets an order in which we visit states. We store transitions instead of storing only their target states to have their source states ready.

Let us take an example. We start with an empty automaton, and we add *a(b(a,b),b(a,b))*, i.e. the tree is read from the input in function UnsortedDTAconstruction. Function AddTree(A, $R = \emptyset$, *a(b(a,b),b(a,b))*) is called. It initialized the set Θ and the stack Ω, and then it calls function Split(A, $R = \emptyset$, $\Theta = \emptyset$, $\Omega = ()$, *a(b(a,b),b(a,b))*). That function divides its last parameter (i.e. the input tree) into subtrees, and then it calls itself with the last parameter being the first subtree *b(a,b)*. That invocation of the function does the same with the subtree, i.e. it calls itself as Split(A, $R = \emptyset$, $\Theta = \emptyset$, $\Omega = ()$, *a*). Since *a* is a leaf, recursion stops here. As the DTA is empty, $\delta_0(a)$ returns a sink state $\perp$. A new state 1 is created and stored in n. A new transition $(a, 1)$ is added to the empty set of transitions Δ. The counter C_1 of incoming transition of state 1 is set to 1. Variable q is set to state 1. Since the transition has no source states, Θ and R remain unchanged, i.e. empty. Variable B_1 is set to $(a, 1)$, and state 1 is returned by the function to be assigned to r_1 one level up in the call hierarchy. Function Split is invoked again as Split(A, $R = \emptyset$, $\Theta = \emptyset$, $\Omega = ()$, *b*). As $\delta_0(b) = \perp$, q is set to $\perp$. A new state 2 is created, and n is set to 2. A new transition $(b, 2)$ is created, and added to Δ, so Δ becomes $\{(a, 1), (b, 2)\}$. The counter C_2 is set to 1, q is set to 2, sets Θ and R remain unchanged. B_2 is set to $(b, 2)$. State 2 is returned to become r_2 in the upper level invocation

Algorithm 4.14 Function LocMin.

```
1:  function LocMin(A, R, Θ, Ω)
2:      while Ω ≠ ∅ do
3:          pop n from Ω
4:          if p ∈ Θ then
5:              q ← FindEquiv(A, R, n)
6:              if q ≠ n then
7:                  τ = (σ, r_1, ..., r_m, n) ← B_n
8:                  F ← F \ {q}
9:                  Θ ← Θ ∪ {r_1, ..., r_m}
10:                 R ← R \ {r_1, ..., r_m}
11:                 Δ ← Δ \ {τ} ∪ (σ, r_1, ..., r_n, q)}
12:                 C_q ← C_q + 1
13:                 for κ = (a, s_1, ..., s_m, j) ∈ Δ such that ∃i : s_i = q do
14:                     C_j ← C_j − 1
15:                     Δ ← Δ \ {κ}
16:                 end for
17:                 delete q
18:             else
19:                 R ← R ∪ {q}
20:             end if
21:             Θ ← Θ \ {q}
22:         end if
23:     end while
24:     return A
25: end function
```

Algorithm 4.15 Function FindEquiv searches for a state in the register R that is the target of the transition τ in automaton A.

```
1:  function FindEquiv(A, R, p)
2:      if ∃q ∈ R such that
```

- $p \in F \equiv q \in F$
- for all $(s, i_1, \ldots, i_m, j) \in \Delta$ and for all $k \leq m$ such that $i_k = p$ or $i_k = q$ $\delta_m(s, i_1, i_{k-1}, p, i_{k+1}, \ldots, i_m) = \delta_m(s, i_1, i_{k-1}, q, i_{k+1}, \ldots, i_m)$

```
    then
3:          return q
4:      else
5:          return p
6:      end if
7:  end function
```

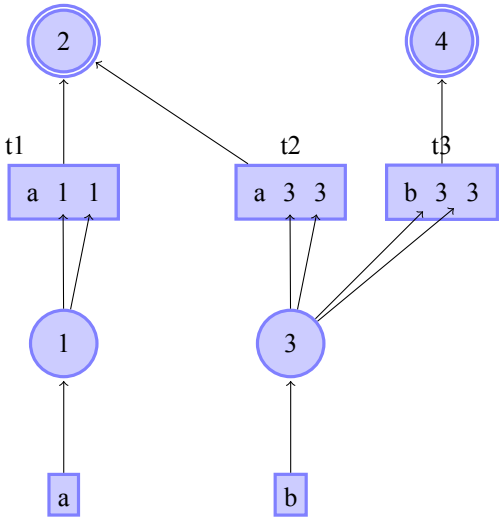


Figure 4.24: The DTA from Figure 4.23 with a tree *b(b,b)* added in a naive way. The DTA does not recognize trees *a(a,b)* and *a(b,a)* anymore.

of Split.

Since $\delta_2(b, 1, 2) = \perp$, q is set to $\perp$. A new state 3 is created, and a new transition $(b, 1, 2, 3)$ is created (added to Δ). The incoming transition counter for state 3 is set to 1. Variable q is set to state 3. States 1 and 2 are added to Θ. Since R is empty, they cannot be removed from there. States 1 and 2 are put onto stack Ω. Back transition B_3 is set to $(b, 1, 2, 3)$. The function returns state 3, which is assigned to variable r_1 in the top-level invocation of Split.

The top-level Split invocation now calls Split(A, $R = \emptyset$, $\Theta = \{1, 2\}$, $\Omega = (1, 2)$, $t = b(a, b)$). Another nested call follows: Split(A, $R = \emptyset$, $\Theta = \{1, 2\}$, $\Omega = (1, 2)$, $t = a$). As a is a leaf, recursion stops. Transition function $\delta_0(a)$ returns state 1. The incoming transition counter for state 1 is equal to 1, so the body of the conditional instruction is skipped. The back transition B_1 for state 1 remains $(a, 1)$. State 1 is returned by the function, and it is assigned to r_1 in the upper-level call to Split. Then Split(A, $R = \emptyset$, $\Theta = \{1, 2\}$, $\Omega = (1, 2)$, $t = b$) is called. It returns state 2 without modifying anything else. In the upper-level call, state 2 is assigned to r_2. As $\delta_2(b, 1, 2)$ returns state 3, and $C_3 = 1$, states 1 and 2 are put (again) onto Ω, nothing else changes, and the function returns state 3 to be assigned to r_2 in the top-level call. Variable q gets a value of $\delta_2(a, 3, 3)$, i.e. $\perp$ as there is no such transition in the DTA. Variable n becomes a new state 4, a new transition $(a, 3, 3, 4)$ is created (added to Δ), and C_4 becomes 1. As state 3 is absent in R, the register remains unchanged. State 3 is added twice to Θ. State 3 is put onto stack Ω. The back transition B_4 is set to $(a, 3, 3, 4)$, and state 4 is returned by the function. We are back in AddTree. State 4 is made final. It is also added to Θ, and put onto stack Ω, which is now $(1, 2, 1, 2, 3, 3, 4)$.

Function LocMin(A, $R = \emptyset$, $\Theta = \{1, 2, 3, 4\}$, $\Omega = (1, 2, 1, 2, 3, 3, 4)$) is called. State 4 is popped and removed from the stack Ω. Function FindEquiv(A, $R = \emptyset$, 4) is invoked. Since $F = \{4\}$, there is no other final, so there is no other equivalent state, and function FindEquiv

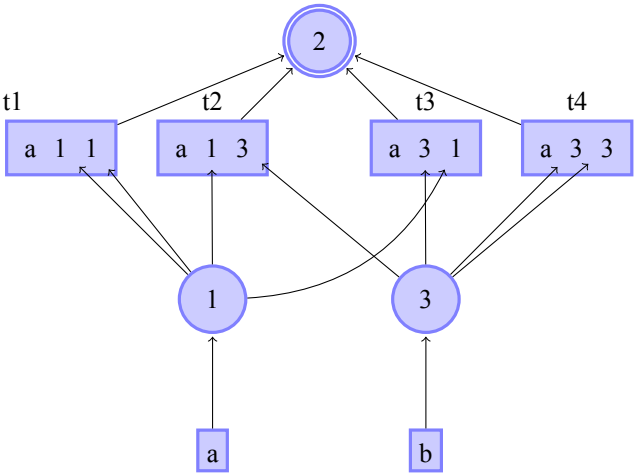


Figure 4.25: The non-minimal recognizing the same language as DTA in Figure 4.23. States 1 and 3 are equivalent.

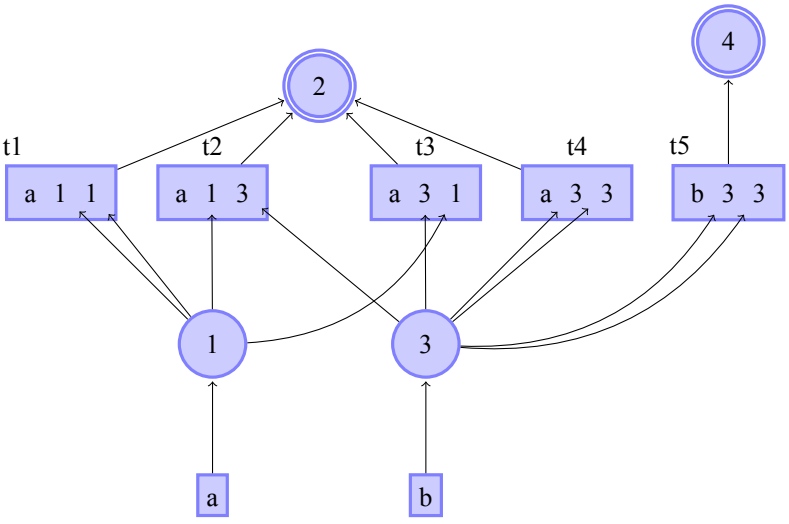


Figure 4.26: A minimal DTA recognizing trees *a(a,a), a(a,b), a(b,a), a(b,b)*, and *b(b,b)*.

returns state 4. In function LocMin, state 4 is added to the register R, and removed from Θ, which is now $\{1, 2, 3\}$. The next state on top of the stack is state 3. It is popped and removed from there. Function FindEquiv cannot find an equivalent state in the register. It returns state 3, which is added to the register, and removed from Θ, which is now $\{1, 2\}$. State 3 is found on top of Ω again, but since it is no longer in Θ, it is silently dropped from the top of the stack. State 2 is popped and removed from the stack Ω. Again, no equivalent state can be found for it in function FindEquiv, so the state is added to the register and removed from Θ. The same

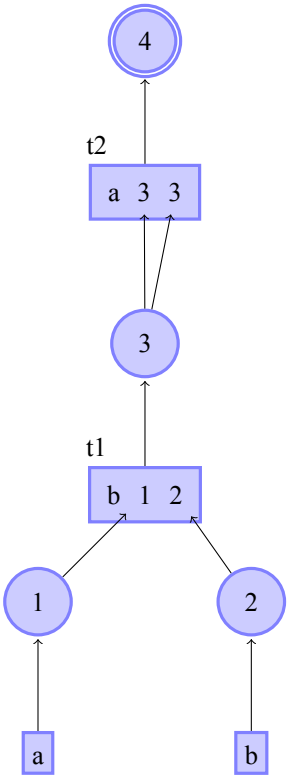


Figure 4.27: Unsorted construction for bottom-up DTAs: a tree *a(b(a,b),b(a,b))* has just been added.

happens to state 1. States 2 and 1 are twice on the stack, but as they are no longer in Θ, they are removed from there with no further action. So Figure 4.27 is also valid for the minimal DTA recognizing the tree *a(b(a,b),b(a,b))*.

Let the next tree on the input be *b(b(a,b),a(b,a),b)*. Function UnsortedDTAconstruction calls AddTree(A, $R = \{1,2,3,4\}$, *b(b(a,b),a(b,a),b)*). It empties Θ and Ω, which are already empty, and calls Split(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b(b(a,b),a(b,a),b)*). Function Split divides t into subtrees, and calls itself as Split(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b(a,b)*), which calls Split(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*). Recursion stops here, and $\delta_0(a)$ returns state 1, which is assigned to variable q. As $C_1 = 1$, B_1 gets transition $(a,1)$ again, and the function returns state 1, which is assigned to variable r_1 one level up in the call hierarchy. A call to Split(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b*) sets B_2 as $(b,2)$, and returns state 2, which is assigned to r_2 one level up. The transition function $\delta_2(b,1,2)$ returns state 3, with $C_3 = 1$. States 1 and 2 are put onto Ω, and $B_3 = (b,1,2,3)$. State 3 is returned and assigned to r_1 in the top-level invocation of Split. That level then calls Split(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = (1,2)$, *t=a(b,a)*), which calls Split(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = (1,2)$, *t=b*). The last call returns state 2 without changing anything. Variable r_2 is set to state 2. Another call: Split(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = (1,2)$, *t=a*) returns state

1 without modifying anything. Variable r_2 is set to state 1. As $\delta_2(a, 2, 1)$ returns $\perp$, a new state 5, and a new transition $(a, 2, 1, 5)$ are created. C_5 is set to 1. States 1 and 2 are added to Θ so that $\Theta = \{1, 2\}$ and removed from the register R, which is now $\{3, 4\}$. States 1 and 2 are pushed onto the stack $\Omega = (1, 2, 2, 1)$, and B_5 is set to $(a, 2, 1, 5)$. State 5 is returned, and it is assigned to r_2 in the top-level invocation of Split. At the top level, a call to Split(A, $R = \{3, 4\}$, $\Theta = \{1, 2\}$, $\Omega = (1, 2, 2, 1)$, *t=b*) returns state 2 without modifying anything. Variable r_3 is set to state 2 at the top level. As $\delta_3(b(3, 5, 2)$ returns $\perp$, a new state 6 and a new transition $(a, 3, 5, 2, 6)$ are created. The counter C_6 is set to 1. States 3 and 5 are added to Θ, as state 2 is already there. Now $\Theta = \{1, 2, 3, 5\}$. State 3 is removed from the register (states 2 and 5 are not there anyway), so that R contains only state 4. States 3, 5, and 2 are pushed onto the stack Ω, and B_6 is set to $(a, 3, 5, 2, 6)$. State 6 is returned to function AddTree. State 6 is made final, and it is added to Θ, so that Θ is now $\{1, 2, 3, 5, 6\}$. It is also put onto the stack so that $\Omega = (1, 2, 2, 1, 3, 5, 2, 6)$ The situation is shown in Figure 4.28.

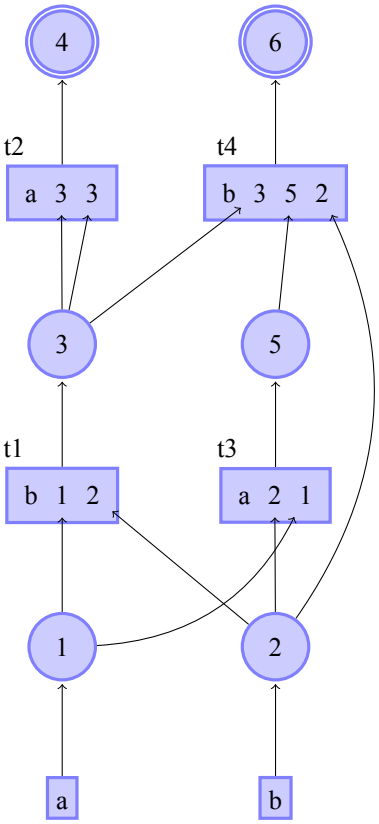


Figure 4.28: Unsorted construction for bottom-up DTAs: a tree *b(b(a,b),a(b,a),b)* has just been added.

Function LocMin(A, $R = \{4\}$, $\Theta = \{1, 2, 3, 5, 6\}$, $\Omega = (1, 2, 2, 1, 3, 5, 2, 6)$) is called. State 6 is popped and removed from Ω. Function FindEquiv returns state 4. The single transition coming to state 6 is retrieved from B_6 and stored in $\tau = (b, 3, 5, 2, 6)$. That transition

is removed. An identical transition with a target state set as state 4 is added. State 4 remains final. States 2, 3, and 5 are already in Θ. C_4 is set to 2, and state 6 is deleted. State 2 is popped and removed from the stack Ω. Function FindEquiv cannot find an equivalent state for it, so the state is returned by FindEquiv, added to the register R in LocMin, and removed from Θ. The next state on top of the stack is state 5. It is popped and removed from there. It has no equivalent state, so it is added to the register, and removed from Θ. The next state is state 3, which is handled in exactly the same way. The same happens to states 1 and 2. Two additional instances of states 2 and 1 are removed silently from the top of the stack Ω as they are no longer present in Θ. The minimized automaton is shown in Figure 4.29.

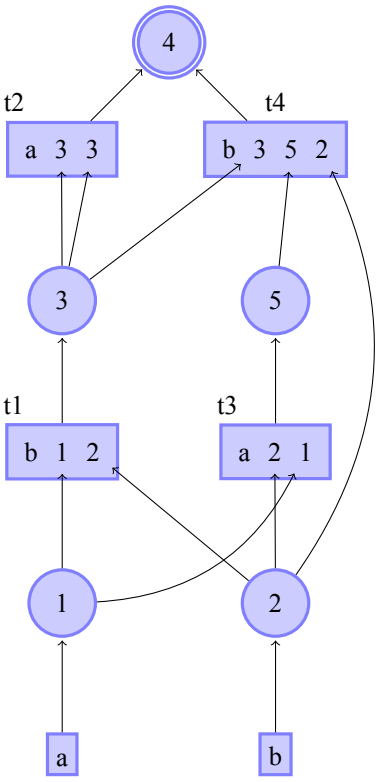


Figure 4.29: Unsorted construction for bottom-up DTAs: a tree *b(b(a,b),a(b,a),b)* has been added, and the DTA has been minimized.

The next tree on the input is *a(a(a,a),a(a,a))*. Function UnsortedDTAconstruction calls AddTree(A, $R = \{1,2,3,4,5\}$, *t=a(a(a,a),a(a,a))*). Function AddTree calls function Split(A, $R = \{1,2,3,4,5\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a(a,a),a(a,a))*), which divides t into subtrees, and calls itself as Split(A, $R = \{1,2,3,4,5\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a,a)*), which calls Split(A, $R = \{1,2,3,4,5\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*). Recursion stops here. As $C_1 = 1$, B_1 is set again to the same value it had before, and state 1 is returned to be assigned to r_1 one level up. An identical call follows, and state 1 is assigned to r_2 one level up. As $\delta_2(a,1,1)$ returns $\perp$, a new state 6, and a new transition $(a,1,1,6)$ are created. C_6 is set to 1. State 1 is added to

Θ, removed from the register R, and it is also pushed twice onto the stack Ω. B_6 is set to $(a, 1, 1, 6)$. State 6 is returned and is assigned to r_1 in the top-level invocation of function Split. A call to Split(A, $R = \{1, 2, 3, 4, 5\}$, $\Theta = \emptyset$, $\Omega = (1, 1)$, *t=a(a,a)*) follows. Leaves are handled in exactly the same way as before, but this time $\delta_2(a, 1, 1) = 6$. No new state or transition is created, and Θ and R remain unchanged. State 1 is pushed again twice onto the stack Ω. The back transition B_6 remains the same. State 6 is assigned to r_2 in the top-level invocation of function Split. As $\delta_2(a, 6, 6) = \perp$, a new state 7, and a new transition $(a, 6, 6, 7)$ are created. C_7 is set to 1. State 6 is added to Θ and is pushed twice onto the stack Ω. It cannot be removed from the register as it is not yet there. B_7 is set to $(a, 6, 6, 7)$. State 7 is returned, and it is made final in function AddTree. It is also added to Θ, and it is pushed onto Ω. The situation is shown in Figure 4.30.

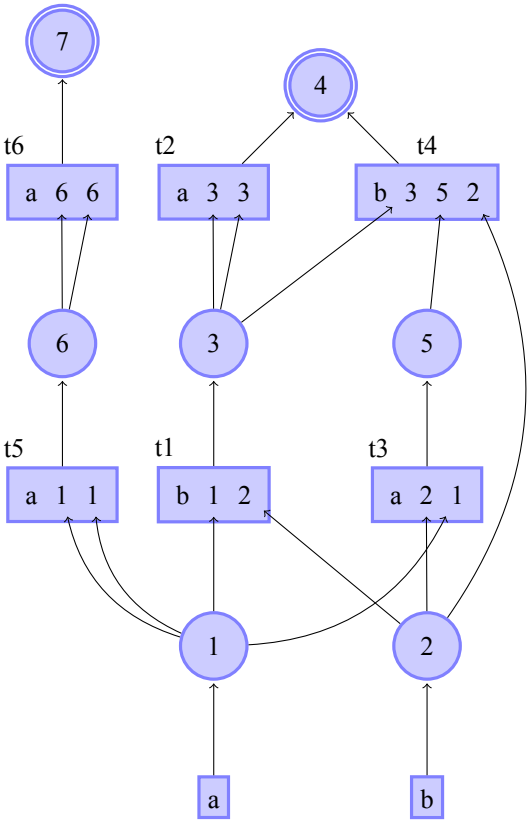


Figure 4.30: Unsorted construction for bottom-up DTAs: a tree *a(a(a,a),a(a,a))* has just been added.

Function LocMin(A, $R = \{2, 3, 4, 5\}$, $\Theta = \{1, 6, 7\}$, $\Omega = (1, 1, 1, 1, 6, 6, 7)$) is called. State 7 is popped and removed from Ω. Function FindEquiv finds an equivalent state for it: state 4. The transition $\tau = (a, 6, 6, 7)$ is replaced with $(a, 6, 6, 4)$, state 7 is deleted, and C_4 is incremented to 3. State 6 is not added to Θ as it is already there. State 6 is popped and removed from Ω. No equivalent state can be found for it, so it is added to R, so that

$R = \{2, 3, 4, 5, 6\}$. State 6 is popped from Ω again, but this time it is not in Θ, so nothing happens. State 1 is popped and removed from Ω. No state is found equivalent to it, so it is added to R and removed from Θ. All other instances of state 1 on Ω are simply removed as the state is no longer in Θ.

The next tree on the input is *a(a(a,a),b(a,b))*. Function Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a(a,a),b(a,b))*) divides t into subtrees. It then calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a,a)*), which calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*). We are at a leaf. The function returns state 1, which is assigned to r_1 one level up. The same call follows, and state 1 is assigned to r_2 this time. As $\delta_2(a, 1, 1) = 6$, and $C_6 = 1$, state 1 is pushed twice onto Ω, and state 6 is returned and assigned to r_1 in the top-level call to Split. Then Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = (1, 1)$, *t=b(a,b)*) is invoked. It calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = (1, 1)$, *t=a*), which returns state 1, and Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = (1, 1)$, *t=b*), which returns state 2. With $r_1 = 1$ and $r_2 = 2$, $\delta_2(b, 1, 2) = 3$, and $C_3 = 1$. States 1 and 2 are pushed onto Ω, B_3 remains the same, and state 3 is returned and assigned to r_2 in the top-level invocation of Split. As $\delta_2(a, 6, 3) = \perp$, a new state 7 and a new transition $(a, 6, 3, 7)$ are created, with C_7 set to 1. States 6 and 3 are added to Θ, so that $\Theta = \{3, 6\}$, and removed from the register, so that $R = \{1, 2, 4, 5\}$. States 6 and 3 are pushed onto Ω, so that $\Omega = (1, 1, 1, 2, 6, 3)$. B_7 becomes $(a, 6, 3, 7)$, and state 7 is returned by the function. It is made final, is added to Θ, and is pushed onto Ω in function AddTree. The resulting tree is shown in Figure 4.31.

Function LocMin(A, $R = \{1, 2, 4, 5\}$, $\Theta = \{3, 6, 7\}$, $\Omega = (1, 1, 1, 2, 6, 3, 7)$) is called. State 7 is popped and removed from the top of Ω. Function FindEquiv returns state 4. Transition $\tau = (a, 6, 3, 7)$ is replaced with $(a, 6, 3, 4)$. States 6 and 3 are already in Θ, C_4 is incremented to 4, and state 7 is deleted and removed from Θ. State 3 is popped and removed from Ω. It is unique, so it is added to R and is removed from Θ. The next state on top of Ω is state 6, and it is treated in the same way. So are state 2 and state 1. Two additional copies of state 1 are removed from the stack without any further processing as state 1 is no longer in Θ.

The next tree on the input is *a(b(a,b),a(a,a))*. Function AddTree(A, $R = \{1, 2, 3, 4, 5, 6\}$, *t=a(b(a,b),a(a,a))*) calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(b(a,b),a(a,a))*), which splits T into subtrees and calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b(a,b)*), which calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*). We are at a leaf. The function returns state 1, which is assigned to r_1 one level up. A call to Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b*) follows. The function returns state 2, which is assigned to r_2. As $\delta_2(b, 1, 2) = 3$, and $C_3 = 1$, states 1 and 2 are pushed onto Ω, B_3 remains unchanged, and state 3 is returned and assigned to r_1 in the top-level invocation of function Split. A call to Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = (1, 2)$, *t=a(a,a)*) follows. It calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = (1, 2)$, *t=a*). The function returns state 1, which is assigned to r_1 one level up. An identical call follows with the same result, and state 1 is assigned to r_2. As $\delta_2(a, 1, 1) = 6$, and $C_6 = 1$, state 1 is pushed twice onto Ω, and state 6 is returned and assigned to r_2 in the top-level invocation of Split. Since $\delta_2(a, 3, 6) = \perp$, a new state 7 and a new transition $(a, 3, 6, 7)$ are created, with C_7 set to 1. States 3 and 6 are added to Θ, so that $\Theta = \{3, 6\}$, removed from the register, so that $R = \{1, 2, 4, 5\}$, and pushed onto Ω, so that $\Omega = (1, 2, 1, 1, 3, 6)$. State 7 is returned, and it is made final, is added to Θ, and is pushed onto Ω.

In function LocMin(A, $R = \{1, 2, 4, 5\}$, $\Theta = \{3, 6, 7\}$, $\Omega = (1, 2, 1, 1, 3, 6, 7)$) invocation, state 7 is popped and removed from the top of Ω. State 7 is equivalent to state 4, so

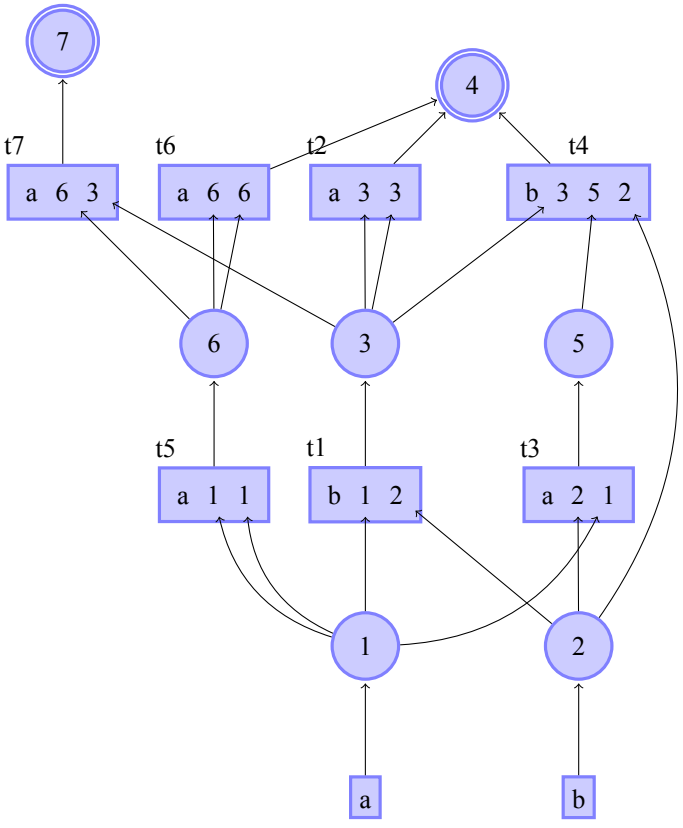


Figure 4.31: Unsorted construction for bottom-up DTAs: a tree *a(a(a,a),b(a,b))* has just been added.

transition $\tau = (a, 3, 6, 7)$ is replaced with $(a, 3, 6, 4)$, C_4 is incremented to 5, and state 7 is deleted and removed from Θ. State 6 is popped and removed from the top of Ω. It is found unique, so it is added to R and removed from Θ. The same happens to state 3. The remaining contents of Ω is not present in Θ, so it is removed without any further action.

The next string on the input is *b(a(a,a),a(b,a),b)*. Function AddTree(A, $R = \{1, 2, 3, 4, 5, 6\}$, *t=b(a(a,a),a(b,a),b)*) calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b(a(a,a),a(b,a),b)*), which divided t into subtrees and calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a,a)*), which calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*). It returns state 1 without modifying anything else. The state is assigned to r_1 one level up. Another call to Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*) does exactly the same thing, with state 1 being assigned to r_2 one level up. State 1 is push twice onto Ω, and state 6 is returned to the top-level invocation of Split, where it is assigned to r_1. A call to Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = (1, 1)$, *t=a(b,a)*) follows, which calls Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = (1, 1)$, *t=b*) that assigns state 2 to r_1, and Split(A, $R = \{1, 2, 3, 4, 5, 6\}$, $\Theta = \emptyset$, $\Omega = (1, 1)$, *t=a*) that assigns state 1 to r_2. After having pushed state 2 and state 1 onto Ω, the function returns state 5, which is assigned to r_2 in the top-level

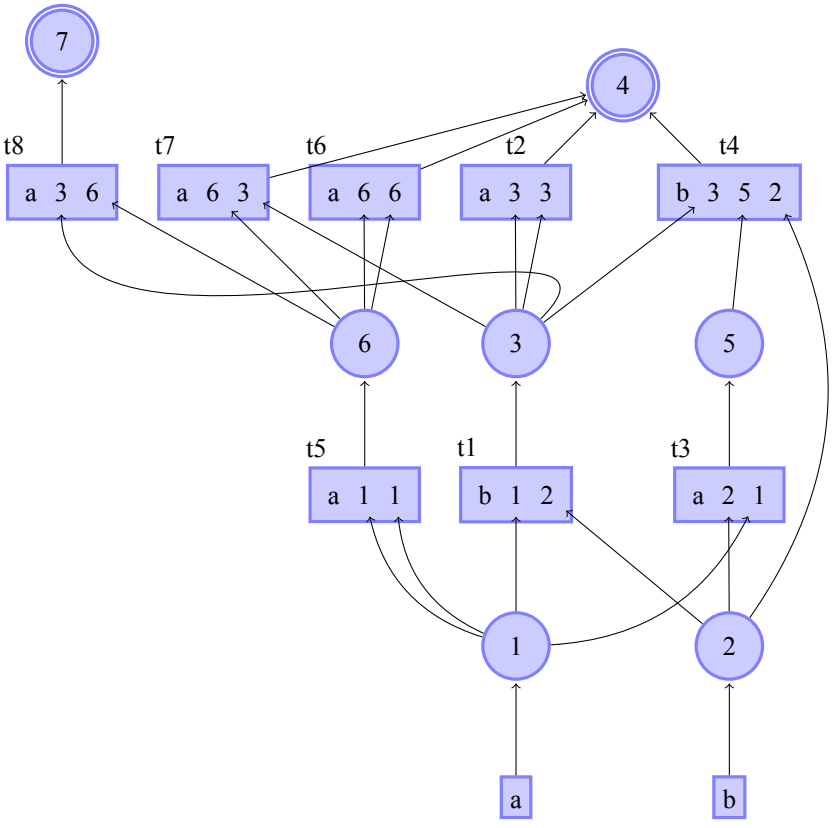


Figure 4.32: Unsorted construction for bottom-up DTAs: a tree *a(b(a,b),a(a,a))* has just been added.

invocation of Split. Another call to Split(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,1,2,1)$, *t=b*) returns state 2, which is assigned to r_3 in the top-level call. As $\delta_3(a,6,5,2) = \bot$, a new state 7 and a new transition $(b,6,5,2,7)$ are created, with C_7 set to 1. States 2, 5, and 6 are added to Θ, and they are removed from R, so that $R = \{1,3,4\}$. States 6, 5, and 3 are put onto Ω, so that $\Omega = (1,1,2,1,6,5,2)$. B_7 is set to $(b,6,5,2,7)$, and state 7 is returned to AddTree, where it is made final, it is added to Θ, and it is pushed onto Ω.

In function LocMin(A, $R = \{1,3,4\}$, $\Theta = \{2,5,6,7\}$, $\Omega = (1,1,2,1,6,5,2,7)$), state 7 is popped and removed from Ω. It is found equivalent to state 4, so transition $\tau = (b,6,5,2,7)$ is replaced with $(b,6,5,2,4)$. States 2, 5, and 6 are already in Θ. C_4 is incremented to 6, and state 7 is deleted and removed from Θ. The resulting DTA is shown in Figure 4.33.

State 2 is popped and removed from Ω. Function FindEquiv cannot find an equivalent state for it, so the state is added to R and removed from Θ. State 5 is popped and removed from Ω. It is found unique, so the state is added to R and removed from Θ. State 6 is popped and removed from Ω. Function FindEquiv finds an equivalent state: state 3. The single transition $\tau = (a,1,1,6)$ going to state 6 is redirected to state 3. State 1 is added to Θ and removed from the register. All outgoing transitions of state 6 are deleted, and the incoming

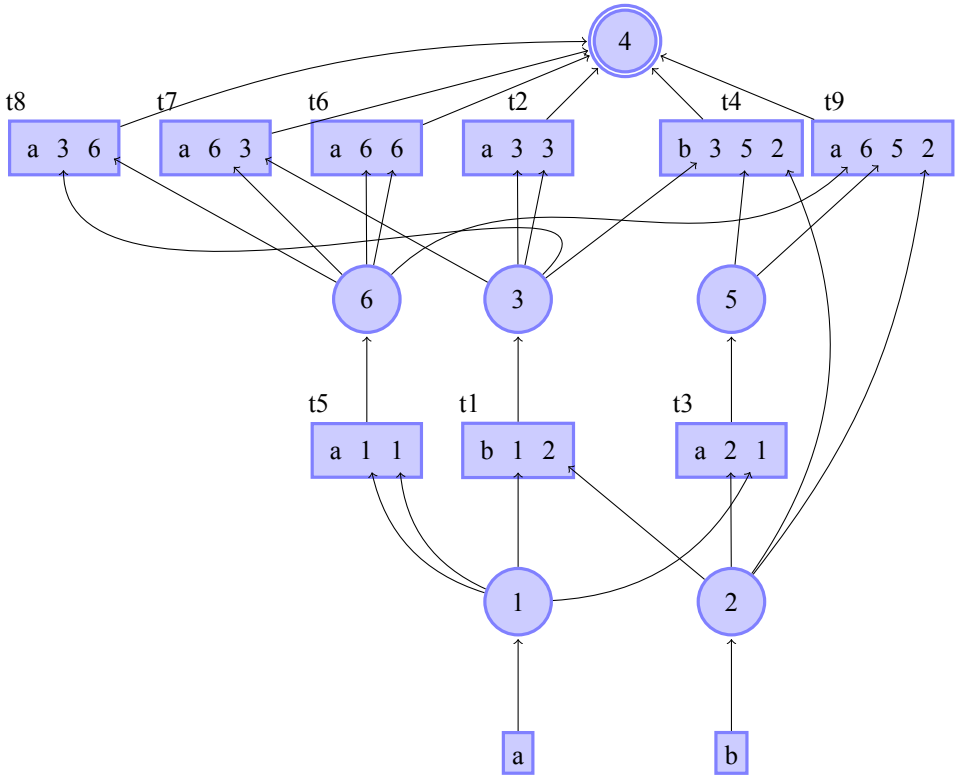


Figure 4.33: Unsorted construction for bottom-up DTAs: a tree *b(a(a,a),a(b,a),b)* has just been added. Function LocMin has just replaced state 7 with state 4, deleting state 4.

transition counter C_4 for their target state 4 is decremented accordingly so that it reaches 2. State 6 is removed from Θ and deleted. State 1 is popped and removed from the top of Ω. The state is found unique, so it is added to R and removed from Θ. The same happens to state 2. Remaining instances of state 1 are removed from Ω without further action as state 1 is no longer in Θ. The resulting DTA is depicted in Figure 4.34.

The last tree on the input is *a(a,a)*. Function AddTree(A, $R = \{1,2,3,4,5\}$, *t=a(a,a)*) calls Split(A, $R = \{1,2,3,4,5\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a,a)*), which calls Split(A, $R = \{1,2,3,4,5\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*) twice, each call returning state 1 without performing any other action. State 1 is assigned to r_1 and r_2. As $\delta_2(a,1,1) = 3$ and $C_3 = 2$, DTA-Clone(A, $q = 3$) is called. It creates a new state 6, and new transitions $(a,3,6,4)$, $(a,6,3,4)$, $(a,6,6,4)$, and $(b,6,5,2)$, incrementing C_4 to 6. A transition $(a,1,1,3)$ is redirected to state 6. Also C_3 is decremented to 1, as the redirected transition goes to state 6, which gets $C_6 = 1$. State 1 is added to Θ, and removed from the register. It is also pushed twice onto Ω. B_6 is set to $(a,1,1,6)$. State 6 is returned to function AddTree, which makes it final, adds it to Θ, and pushes it onto Ω. The resulting automaton is shown in Figure 4.35.

In function LocMin(A, $R = \{2,3,4,5\}$, $\Theta = \{1,6\}$, $\Omega = (1,1,6)$), state 6 is popped and removed from Ω. It is found unique, so it is added to R and is removed from Θ. The

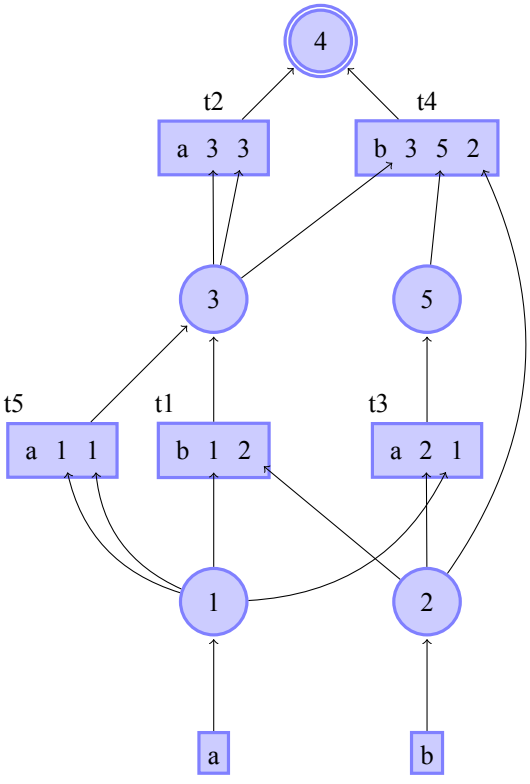


Figure 4.34: Unsorted construction for bottom-up DTAs: a tree *b(a(a,a),a(b,a),b)* has been added. Function LocMin has minimized the automaton.

same happens with state 1. The remaining copy of state 1 is removed from Ω without further action as $\Theta = \emptyset$. The DTA in Figure 4.35 is the minimal automaton.

Let DTA $A = (Q, \Sigma, \Delta, F)$ be the minimal automaton before function AddTree(A, R, t) is called, let R be the contents of the register before the call ($R = Q$), and t be the tree to be added to the language of the automaton. Let $A' = (Q', \Sigma, \Delta', F')$ be the automaton just before the call to LocMin, let R be the contents of the register before the call, and Θ' and Ω' be the contents of Θ and Ω at the same time. Let $A'' = (Q'', \Sigma, \Delta'', F'')$ be the automaton after the invocation of AddTree is completed, with the register contents R''. Transition functions δ_m, δ'_m, and δ''_m are the transition functions in the respective stages of the process of adding a tree to the language of the automaton. We will show that:

1. $\mathcal{L}(A') = \mathcal{L}(A) \cup \{t\}$
2. Every state in A' is reachable and co-reachable.
3. $\mathcal{L}(A'') = \mathcal{L}(A')$
4. A'' is minimal

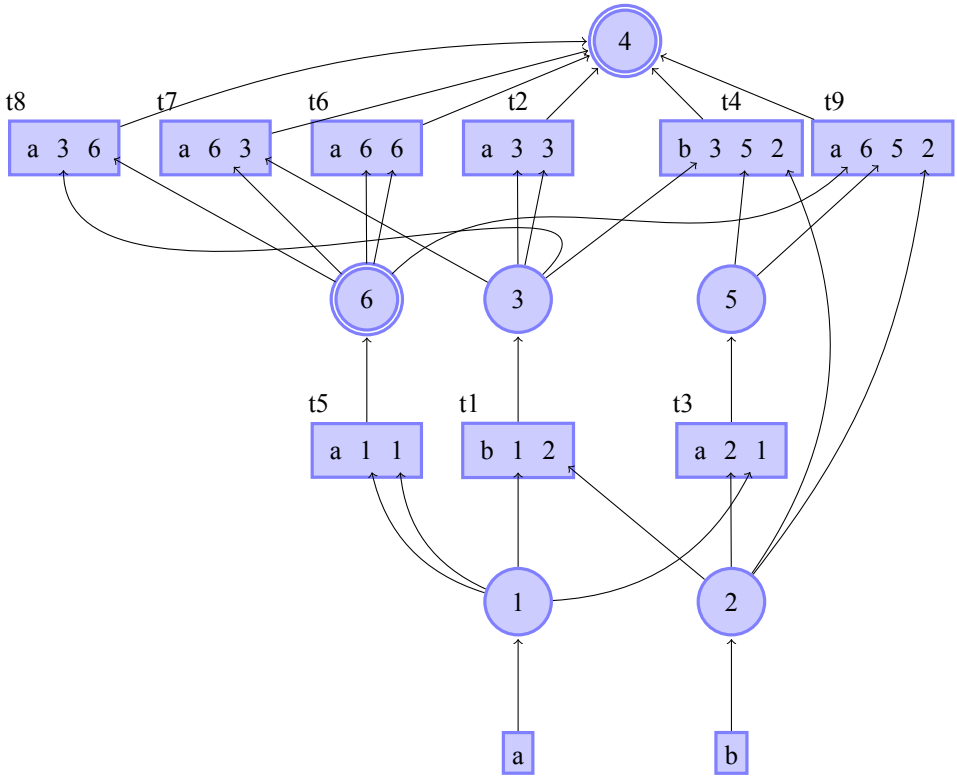


Figure 4.35: Unsorted construction for bottom-up DTAs: a tree *a(a,a)* has just been added.

To prove point 1, let us notice that adding a tree is handled almost entirely by function Split. It divides a tree $t = \sigma(t_1, \ldots, t_m)$ into subtrees $t_1, \ldots, t_m$, and makes sure that all changes necessary to make $r_1 = \delta_A(t_1), \ldots, r_m = \delta_A(t_m)$ so that each r_i has $C_{r_i} = 1$. If $\delta_m(\sigma, r_1, \ldots, r_m) = \bot$, it means that a new state and a new transition must be added to the DTA, and the function creates a new state n and a new transition $(\sigma, r_1, \ldots, r_m, n)$ so that $\delta'_m(\sigma, r_1, \ldots, r_m) = n$. Note that $C_n = 1$, and $L_{A'}(n) = t$. Otherwise the tree t is already a subtree (not necessarily proper if $q = \delta_m(\sigma, r_1, \ldots, r_m) \in F$) of a tree already present in the language of the DTA. If $C_q = 1$, then nothing needs to be done for the tree as $L_A(q) = t$. Otherwise q is cloned, so that C_q is decremented by one, and the incoming transition counter C'_n for the clone is 1. Cloning, as discussed above, ensures that the clone n is used in exactly the same contexts as q, therefore cloning does not change the language of the automaton. The top-level invocation of function Split returns state q such that $L_{A'}(q) = \{t\}$. In function AddTree, q is made final, so by definition (2.19), $t \in \mathcal{L}(A')$. No tree is deleted from $\mathcal{L}(A)$, as no states are deleted in function Split, and transitions are redirected only to clones of original targets that function in exactly the same way as the original targets. The only new states that are added are states that are created when for a particular subtree s of the tree t, $\delta_A(s) = \bot$, and after that creation of a state n, $L_{A'}(n) = s$. As $C_n = 1$, and only $\delta_{A'}(t)$ is made final, nothing but t is added to $\mathcal{L}(A)$.

To prove point 2, notice that every newly created state q is made reachable as there is a tree t such that $L_A(q) = \{t\}$. A state may become unreachable by deleting a state or a transition, or by redirecting a transition. Function Split deletes no states and no transitions. When it redirects a transition, its original target state has more than one incoming transition, so it stays reachable. A state may stop being co-reachable by deleting a state or a transition, or by redirecting a transition. Function Split deletes no states and no transitions. When it redirects a transition, it redirects it towards a clone of the original target state which behaves in exactly the same way as the original target state, so the new target state must be co-reachable.

Function LocMin would be responsible for any differences between $\mathcal{L}(A')$ and $\mathcal{L}(A'')$. There are no differences there, i.e. that local minimization does not change the language of the automaton. Notice that function LocMin replaces a state with an equivalent one, and redirects to that state transitions that originally went to the state being replaced. Such operation does not change the language of the automaton.

To prove the last point, notice that the states not visited during addition of a new tree t (usually most states of the automaton) are pairwise inequivalent. They do not need to be reprocessed. All other states are put onto stack Ω in the order of those visits, and are popped from there in the reverse order. Notice also that throughout function AddTree, the sum of the contents of the register and the current contents of the set Θ form Q, and the first sets are disjoint. In function AddTree, $\delta_A(t)$ is added to Θ, and removed from R. In function Split, each state added to Θ is removed from R (the last `for` loop). In function LocMin, only states in Θ are processed, they are either added to the register or deleted, and they are always removed from Θ. Only states put onto Ω may need to be processed, and they are in the proper order in Ω. The remaining question is whether Θ is large enough to contain all states that require processing. It is clear that all new states (including clones) should be processed, i.e. it should be checked whether there is an equivalent state in the minimized part of the DTA. All such states are added to Θ in function Split (because they are source states of new transitions) or in function AddTree (the root). The remaining states, when they are processed in postorder (here it means that targets of outgoing transitions of any state are processed before that state), may become equivalent to some other states only when they change their signatures. This happens on three occasions: when new transitions are added to a state in function Split, when a transition of a state is redirected towards a clone of the original target of that transition in function Split, and when the original target is replaced with an equivalent state in function LocMin. In all these three cases source states of new or modified transitions are removed from the register and added to Θ if they are not already there. There may be more than one instance of a state in Ω. When the state is processed, it is removed from Θ, so that it is no longer processed, unless it changes its signature again. Therefore, the DTA A'' is minimal.

We do not provide any analysis of computational complexity of the algorithm. Due to the complexity of cloning and merging states, such analysis would have to take into account properties of input trees. In an efficient program, the register must be implemented as a hash table. The hash function must act on the signature of a state (see Equation (2.29)).

This algorithm is a result of cooperation of the author with Rafael Carrasco and Mikel Forcada. It was published in [8].

4.5 Extension to Pseudo-Minimal Bottom-Up Tree Automata

Like other extensions to pseudo-minimal automata described earlier in this book, this one relies mainly on replacing equivalence with pseudo-equivalence. State q should be replaced with an equivalent one, only if $|R_A(q)| \leq 1$. Let us investigate, when this condition is met.

$$|R_A(q)| \leq 1 \equiv \begin{cases} \text{true} & \text{fanout}(q) = \emptyset \\ \text{false} & |\text{fanout}(q)| > 1 \\ \text{false} & |\text{fanout}(q)| = 1 \wedge q \in F \\ |R_A(n)| \leq 1 \wedge \forall_{k \neq j} |L_A(r_k)| \leq 1 & \text{otherwise with fanout}(q) = \\ & \{(\sigma, r_1, \ldots, r_m, n, j)\} \end{cases} \tag{4.1}$$

The first point in the condition is obvious. The second and the third one are facets of the same situation, as a final state implies the acceptance of another (additional) tree. The last point is less obvious. It describes a situation where from the destination state of the single outgoing transition τ of q forwards, there is a single tree top, but other source states in τ may contribute more than one tree each. The first three points can obviously be computed locally in q. In the fourth point, $|L_A(r_k)|$ can easily be computed on the fly during addition of a tree (a state being cloned has its L_A decreased by one, and all states visited during that phase have L_A equal to 1), and during local minimization by adding L_A's of merged states.

$$|L_A(q)| = \sum_{\tau=(\sigma,r_1,\ldots,r_m,n) \in \Delta : n=q} \sum_{i=1,\ldots,m} |L_A(r_i)| + |\{(\sigma, n) \in \Delta : n = q\}| \tag{4.2}$$

If $\exists_{k \neq j} |L_A(r_k)| > 1$, then $|R_A(n)| \leq 1$, because n is processed before q, and the condition must also be met at n.

Let us take an example. Figure 4.36 shows a DTA recognizing trees *a(a(a,a)), a(a(a,b)), a(a(b,a)), a(a(b,b))*. We are in the middle of pseudo-minimization. States 1 and 2 are equivalent. Are they pseudo-equivalent? Yes, they are pseudo-equivalent, because $|\text{fanout}(2)| = 4 > 1$, and $R_A(2) =$*a(a(a,#)),a(a(#,a)),a(a(#,b)),a(a(b,#))*.

Fortunately, R_A can be computed in a much simple way. R_A of a new state (either completely new or cloned) is always one. R_A of other states visited during addition of a tree is increased by one. R_A of other states (not visited) is not affected. Minimization replaces a state with another one with identical R_A. Unfortunately, states can be visited more than once during addition of the same tree. Moreover, a newly created state can be visited again as an unmodified, existing state. It is possible to use two sets of states for following those changes and updating R_A counters, but a simpler solution is to use markers associated with every state. The markers can take three values: *unchanged* -- not (yet) updated during addition of the current tree, *modified* -- with the counter incremented, and *new* -- with the counter set to one. The markers act as a global vector M. Such computation of $|R_A|$ is valid only for states with $|L_A|$ equal to one[3], but only such states are processed in the forward step of the algorithm.

The main function PseudoUnsortedDTAconstruction is given as Algorithm 4.16. The only difference to function UnsortedDTAconstruction (page 98) is the use of of the function PseudoAddTree instead of AddTree.

[3]For states q with $|L_A(q)| > 1$, the computed value of $|R_A(q)|$ should be divided by $|L_A(q)|$.

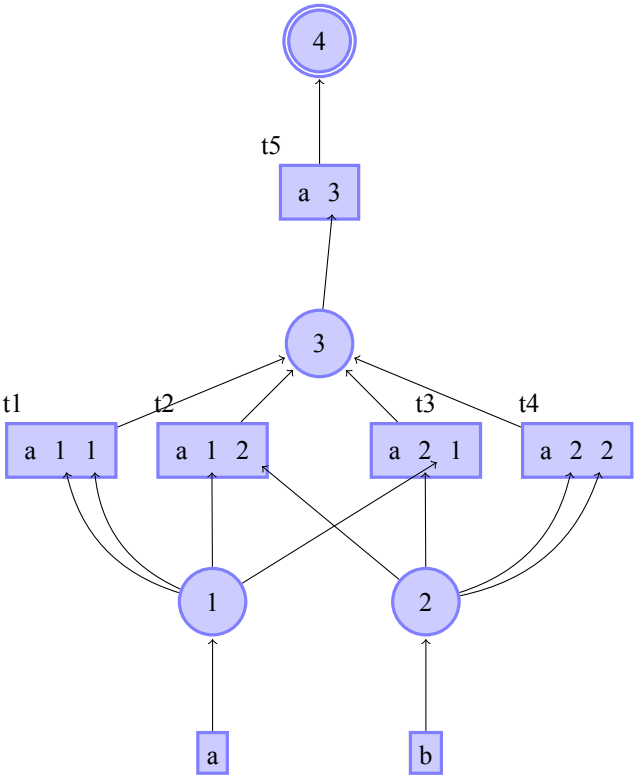


Figure 4.36: A DTA recognizing trees *a(a(a,a)), a(a(a,b)), a(a(b,a)), a(a(b,b))*. State 1 is equivalent to state 2. Are they pseudo-equivalent?

Function PseudoAddTree also looks familiar --- Algorithm 4.17. It is identical to function AddTree (page 99), except that it calls function PseudoSplit instead of Split and LocPseudoMin instead of LocMin.

In function PseudoSplit, we maintain the counters T for R_A. We not only need to maintain the counters T themselves, but we must also set markers M on states so that the counters T can be changed only once during addition of a single tree. Note that when the state appears not to be new (i.e. the transition function returns a state), we still have to check the marker M as the state might have been created to recognize an identical subtree somewhere else in the current tree.

Function PseudoLocMin is different from function LocMin in only two places. Firstly, function FindEquiv is called only when $|R_A(n)| \leq 1$. This is the condition for pseudo-equivalence. Secondly, we clear the markers for all states in Ω (they are not used in PseudoLocMin anyway) before the next invocation of function AddTree.

Let us take the same example of input data as in the previous section. Function PseudoUnsortedDTAconstruction is called. It creates an empty automaton, resets the register, and it reads the first tree from the input. The tree is *a(b(a,b),b(a,b))*. Function PseudoAddTree(A, $R = \emptyset$, *t=a(b(a,b),b(a,b))*) is called. It resets Θ and Ω, and it calls function PseudoSplit(A,

Algorithm 4.16 Function PseudoUnsortedDTAconstruction adds trees to the language of a DTA A. Trees are read from the standard input. A register R initially contains all states of A. Both A and R are modified by the function, so that at the end, $|Q| = |R|$. The function returns a pseudo-minimal automaton that recognizes all trees recognized by the original automaton and all trees read from the input.

```
1: function PseudoUnsortedDTAconstruction(A, R)
2:     Create an empty automaton A
3:     R ← ∅
4:     while input not empty do
5:         t ← next tree
6:         (A, R) ← PseudoAddTree(A,R,t)
7:     end while
8:     return A
9: end function
```

Algorithm 4.17 Function PseudoAddTree adds a tree t to the language of a pseudo-minimal DTA A. All states of the DTA are in the register R before the call. The procedure modifies A, so that it recognizes the original language and the tree t, and it also modifies R so that it contains all states of the new automaton. Variable Θ holds a set of states that are new or that changes their signature. A stack Ω holds the same states as Θ in chronological order of those visits. It may later be changed by LocPseudoMin.

```
1: function PseudoAddTree(A, R, t)
2:     Θ ← ∅
3:     Ω ← ∅
4:     q ← PseudoSplit(A, R, Θ, Ω, t)
5:     R ← R \ {q}
6:     F ← F ∪ {q}
7:     Θ ← Θ ∪ {q}
8:     push q onto Ω
9:     A ← LocPseudoMin(A, R, Θ, Ω)
10:    return (A, R)
11: end function
```

Algorithm 4.18 Function PseudoSplit adds a tree t to the language of a pseudo-minimal automaton A without making $\delta_A(t)$ final. The register R contains all states in the minimized part of the automaton. The set Θ contains all states that need to be submitted to minimization. The function returns $\delta_A(t)$. The automaton A, the register R, the set Θ, and the stack Ω are modified inside the function. T and L are global vectors of counters for R_A and L_A, respectively. M is a global vector of markers --- see the text.

```
 1: function PseudoSplit(A, R, Θ, Ω, Ψ, t = σ(t_1, ..., t_m))
 2:     for k ∈ 1, ..., m do
 3:         r_k ← Split(A, R, Θ, Ω, t_k)
 4:     end for
 5:     q ← δ_m(σ, r_1, ..., r_m)
 6:     if q = ⊥ ∨ C_q ≠ 1 then
 7:         if q = ⊥ then
 8:             n ← new state
 9:             Δ ← Δ ∪ {(σ, r_1, ..., r_n, n)}
10:             C_n ← 1
11:         else
12:             n ← DTAClone(A, q)
13:             Δ ← Δ \ (σ, r_1, ..., r_m, q) ∪ (σ, r_1, ..., r_m, n)
14:             C_q ← C_q − 1
15:             C_n ← 1
16:         end if
17:         T_n ← 1
18:         M_n ← new
19:         q ← n
20:         for i ∈ 1, ..., m do
21:             if r_i ∉ Θ then
22:                 Θ ← Θ ∪ {r_i}
23:                 R ← R \ {r_i}
24:             end if
25:         end for
26:     else if M_q = unchanged then
27:         M_q ← modified
28:         T_q ← T_q + 1
29:     end if
30:     push r_1, ..., r_m onto Ω
31:     B_q ← (σ, r_1, ..., r_m, q)
32:     return q
33: end function
```

Algorithm 4.19 Function PseudoLocMin.

```
1:  function PseudoLocMin(A, R, Θ, Ω)
2:      while Ω ≠ ∅ do
3:          pop n from Ω
4:          if n ∈ Θ then
5:              if T_n ≤ 1 then
6:                  q ← FindEquiv(A, R, n)
7:              else
8:                  q ← n
9:              end if
10:             if q ≠ n then
11:                 τ = (σ, r_1, ..., r_m, n) ← B_n
12:                 F ← F \ {q}
13:                 Θ ← Θ ∪ {r_1, ..., r_m}
14:                 R ← R \ {r_1, ..., r_m}
15:                 Δ ← Δ \ {τ} ∪ (σ, r_1, ..., r_n, q)}
16:                 C_q ← C_q + 1
17:                 for κ = (a, s_1, ..., s_m, j) ∈ Δ such that ∃i : s_i = q do
18:                     C_j ← C_j − 1
19:                     Δ ← Δ \ {κ}
20:                 end for
21:                 delete q
22:             else
23:                 R ← R ∪ {q}
24:             end if
25:             Θ ← Θ \ {q}
26:         end if
27:         M_n ← unchanged
28:     end while
29:     return A
30: end function
```

$R = \emptyset, \Theta = \emptyset, \Omega = ()$, *t=a(b(a,b),b(a,b))*). Function PseudoSplit dived the tree into subtrees, and calls itself as PseudoSplit(A, $R = \emptyset, \Theta = \emptyset, \Omega = ()$, *t=b(a,b)*), which calls itself again as PseudoSplit(A, $R = \emptyset, \Theta = \emptyset, \Omega = ()$, *t=a*). A new state 1 is created, a new transition $(a, 1)$ is added to Δ, C_1 is set to 1, T_1 is set to 1, M_1 is set to *new*, B_1 is set to $(a, 1)$, and state 1 is returned one level up, where it is assigned to variable r_1. Function PseudoSplit(A, $R = \emptyset$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*) is called again, this time with *b* as the last parameter. It creates a new state 2 and a new transition $(b, 2)$, C_2 is set to 1, T_2 is set to 1, M_2 is set to *new*, states 1 and 2 are added to Θ and pushed onto Ω, B_2 is set to $(b, 2)$, and state 2 is returned one level up, where it is assigned to variable r_2. At that level, new state 3 and a new transition $(b, 1, 2, 3)$ are created. C_3 is set to 1, T_3 is set to 1, M_3 is set to *new*, B_3 is set to $(b, 1, 2, 3)$, and state 3 is returned to the top-level invocation of PseudoSplit to be assigned to variable r_1. A call to PseudoSplit(A, $R = \emptyset$, $\Theta = \{1, 2\}$, $\Omega = (1, 2)$, *t=b(a,b)*) follows, which calls PseudoSplit(A, $R = \emptyset$, $\Theta = \{1, 2\}$, $\Omega = (1, 2)$, *t=a*). This time, the transition exists, and q is set to 1. It is checked that $M_1 =$ *new*, so T_1 is unchanged. B_1 is set to the same transition as before, and state 1 is returned to be assigned to variable r_1 one level up. PseudoSplit(A, $R = \emptyset$, $\Theta = \{1, 2\}$, $\Omega = (1, 2)$, *t=b*) is called, and it also returns state 2 without modifying anything else. Variable r_2 becomes state 2. The transition function returns state 3. As $M_3 =$ *new*, T_3 remains 1, states 1 and 2 are pushed onto Ω, and state 3 is returned to be assigned to r_2 at the top level. A new state 4 and a new transition $(a, 3, 3, 4)$ are created. c_4 is set to 1, T_4 is set to 1, M_4 is set to *new*, state 3 is added to Θ, and pushed twice onto Ω. B_4 is set to $(a, 3, 3, 4)$, and state 4 is returned to function PseudoAddTree. State 4 is made final, it is added to Θ and pushed onto Ω. The situation is shown in Figure 4.37.

A call to PseudoLocMin(A, $R = \emptyset$, $\Theta = \{1, 2, 3, 4\}$, $\Omega = (1, 2, 1, 2, 3, 3, 4)$) follows. State 4 is popped and removed from Ω. As $n \in \Theta$ and $T_4 \leq 1$, function FindEquiv is called, which returns the same state. State 4 is added to the register, and removed from Θ, and M_4 is set to *unchanged*. State 3 is popped and removed from Ω. The condition on $|R_A|$ holds, but no equivalent state can be found, so state 3 is added to the register, and removed from Θ, and M_3 is set to *unchanged*. State 3 is again popped and removed from Ω. This time, it is not in Θ, so only M_3 is again set to *unchanged*. State 2 is popped and removed from Ω. The condition on $|R_A|$ holds, but no equivalent state can be found. State 2 is added to R, and removed from Θ. M_2 is set to *unchanged*. State 1 is popped and removed from Ω. The condition on $|R_A|$ holds, but no equivalent state can be found. State 1 is added to R, and removed from Θ. M_1 is set to *unchanged*. State 2 is popped and removed from Ω again, but this time it is not in Θ, so only M_2 is set again to *unchanged*. The same happens to the remaining instance of state 1. The automaton in Figure 4.37 is minimal.

The next tree on the input is *b(b(a,b),a(b,a),b)*. Function AddTree(A, $R = \{1, 2, 3, 4\}$, *t=b(b(a,b),a(b,a),b)*) is called. It resets Θ and Ω, and calls function PseudoSplit(A, $R = \{1, 2, 3, 4\}$, $\Theta = \emptyset, \Omega = ()$, *t=b(b(a,b),a(b,a),b)*), which calls PseudoSplit(A, $R = \{1, 2, 3, 4\}$, $\Theta = \emptyset, \Omega = ()$, *t=b(a,b)*), which calls PseudoSplit(A, $R = \{1, 2, 3, 4\}$, $\Theta = \emptyset, \Omega = ()$, *t=a*). Recursion stops here. As $\delta_0(a) = 1$ and $C_1 = 1$ and $M_1 =$ *unchanged*, M_1 is set to *modified*, and T_1 is incremented to 2. B_1 is set to $(a, 1)$ again, and state 1 is returned one level up to be assigned to r_1. PseudoSplit(A, $R = \{1, 2, 3, 4\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b*) is called. As $\delta_0(b) = 2$ and $C_2 = 1$ and $M_2 =$ *unchanged*, M_2 is set to *modified*, and T_2 is incremented to 2. B_2 is set to $(b, 2)$ again, and state 2 is returned one level up to be assigned to r_2. The transition function returns state 3, $C_3 = 1$, $M_3 =$ *unchanged*, so M_3 is set to *modified*, and T_3 is incremented to 2. States 1 and 2 are put onto Ω, B_3 remains $(b, 1, 2, 3)$, and state 3 is returned to the top-

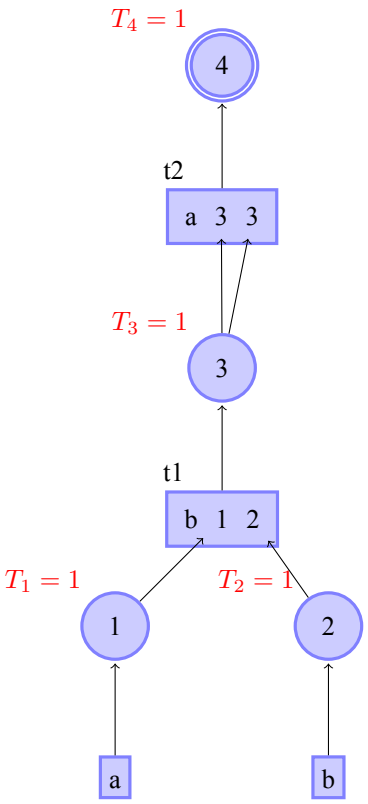


Figure 4.37: Unsorted construction of bottom-up pseudo-minimal DTAs: a tree *a(b(a,b),b(a,b))* has just been added.

level of PseudoSplit to be assigned to variable r_1. A call to PseudoSplit(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = (1,2)$, *t=a(b,a)*) follows, and then to PseudoSplit(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = (1,2)$, *t=b*). Again, $\delta_0(b) = 1$ and $C_2 = 1$, but $M_2 =$ *modified*, so neither M_2 nor T_2 are changed. Nor is B_2 (it is assigned the same value), and state 2 is returned one level up to be assigned to r_1. Next PseudoSplit(A, $R = \{1,2,3,4\}$, $\Theta = \emptyset$, $\Omega = (1,2)$, *t=a*) is called that returns state 1 without any further actions. That state is assigned to variable r_2 one level up. At that level, $\delta_2(a,2,1) = \perp$, so new state 5 and a new transition $(a,2,1,5)$ are created. C_5 is set to 1, so is T_5, and M_5 is set to *new*. States 2 and 1 are added to Θ, removed from the register, and pushed onto Ω. B_5 is set to $(a,2,1,5)$, and state 5 is returned to the top-level invocation of PseudoSplit to be assigned to variable r_2. Then PseudoSplit(A, $R = \{3,4\}$, $\Theta = \{1,2\}$, $\Omega = (1,2,2,1)$, *t=b*) is called. It returns state 2 without any further action. At the top-level invocation of function PseudoSplit, $\delta_3(b,3,5,2) = \perp$, so a new state 6 and a new transition $(b,3,5,2,6)$) are created, with C_6 set to 1. T_6 is set to 1, M_6 is set to *new*, state 3 is removed from the register, and states 3 and 5 and added to Θ. States 3, 5 and 2 are pushed onto Ω, and B_6 is set to $(b,3,5,2,6)$. State 6 is returned to function PseudoAddTree, where it is made final, it is added to Θ and pushed onto Ω.

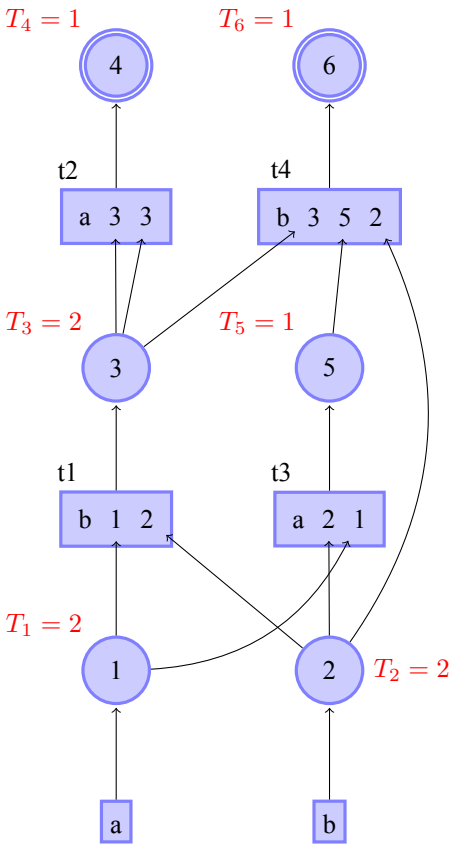


Figure 4.38: Unsorted construction of pseudo-minimal bottom-up DTAs: a tree *b(b(a,b),a(b,a),b)* has just been added.

Function PseudoLocMin(A, $R = \{4\}$, $\Theta = \{1, 2, 3, 5, 6\}$, $\Omega = (1, 2, 2, 1, 3, 5, 2, 6)$) is called. State 6 is popped and removed from Ω. It is in Θ, and $T_6 = 1$, so function FindEquiv is called. It finds that state 4 is equivalent to state 6. States 3, 5, and 2 are already in Θ, and they are not in the register. The transition leading to state 6 is redirected to state 4. State 6 is deleted, and C_4 is incremented to 2. State 6 is removed from Θ, and M_6 is set to *unchanged*. State 2 is popped and removed from Ω. It is in Θ, but $T_2 = 2$, so state 2 is added to the register and removed form Θ, and M_2 is set to *unchanged*. State 5 is popped and removed from Ω. It is in Θ, and $T_5 = 1$, so function FindEquiv is called, but it finds no equivalent state, so state 5 is added to the register, removed from Θ, and M_5 is set to *unchanged*. State 3 is popped and removed from Ω. It is in Θ, but $T_3 = 2$, so state 3 is added to the register, and it is removed from Θ, with M_6 set to *unchanged*. State 1 is popped and removed from the top of Ω. It is in Θ, but $T_1 = 2$, so it is added to the register and removed from Θ, and M_1 is set to *unchanged*. The remaining instances of states 2 and 1 on Ω are removed from there without further actions as the states are no longer in Ω, and their markers are both set to *unchanged*.

The next tree on the input is *a(a(a,a),a(a,a))*. Function PseudoAddTree(A, $R = \{1,2,3,4,5\}$, *t=a(a(a,a),a(a,a))*) resets Θ and Ω, and calls function PseudoSplit(A, $R = \{1,2,3,4,5\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a(a,a),a(a,a))*), which calls PseudoSplit(A, $R = \{1,2,3,4,5\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a,a)*), which calls PseudoSplit(A, $R = \{1,2,3,4,5\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*). As $\delta_0(a) = 1$, $C_1 = 1$, and $M_1 =$ *unchanged*, M_1 is set to *modified*, and T_1 is incremented to 3. State 1 is returned one level up and assigned to r_1. PseudoSplit(A, $R = \{1,2,3,4,5\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*) is called again. This time, $M_1 =$ *modified*, so state 1 is returned without any other action, and state 1 is assigned to r_2 one level up. At that level, $\delta_2(a,1,1) = \perp$, so a new state 6 and a new transition $(a,1,1,6)$ are created, with C_6 set to 1. T_6 is set to 1, M_6 is set to *new*, state 1 is removed from the register and added to Θ. It is pushed twice onto Ω. B_6 is set to $(a,1,1,6)$, and state 6 is returned to the top-level invocation of PseudoSplit, where it is assigned to variable r_1. PseudoSplit(A, $R = \{2,3,4,5\}$, $\Theta = \{1\}$, $\Omega = (1,1)$, *t=a(a,a)*) is called again, and then PseudoSplit(A, $R = \{2,3,4,5\}$, $\Theta = \{1\}$, $\Omega = (1,1)$, *t=a*). The last call returns state 1 without any modification of variables, and that value is assigned to variable r_1 one level up in the call hierarchy. PseudoSplit(A, $R = \{2,3,4,5\}$, $\Theta = \{1\}$, $\Omega = (1,1)$, *t=a*) is called again with identical result, and this time state 1 is assigned to variable r_2. As $\delta_2(a,1,1) = 6$ and $C_6 = 1$ and $M_6 =$ *new*, state 1 is pushed twice onto Ω, and state 6 is returned to be assigned to variable r_2 at the top-level invocation of PseudoSplit. As $\delta_2(a,6,6) = \perp$, a new state 7 and a new transition $(a,6,6,7)$ are created, with C_7 set to 1. T_7 is set to 1, M_7 is set to *new*, state 6 is added to Θ and pushed twice onto Ω, B_7 is set to $(a,6,6,7)$, and state 7 is returned to function PseudoAddTree, where it is made final, it is added to Θ and pushed onto Ω. The situation is shown in Figure 4.39.

Function PseudoAddTree calls function PseudoSplit(A, $R = \{2,3,4,5\}$, $\Theta = \{1,6,7\}$, $\Omega = (1,1,1,1,6,6,7)$). State 7 is popped and removed from the register. It is in Θ, and $T_7 = 1$, so function FindEquiv finds a state that is equivalent to state 7, i.e. state 4. The transition leading to state 7 is redirected towards state 4, state 7 is deleted, and C_4 is incremented to 3. State 7 is removed from Θ, and M_7 is set to *unchanged*. State 6 is popped and removed from the top of Ω. It is in Θ, and $T_6 = 1$, but FindEquiv cannot find an equivalent state, so state 6 is added to the register, removed from Θ, and M_6 is set to *unchanged*. Another copy of state 6 is popped and removed from Ω. This time it is not in Θ, so nothing happens -- M_6 is already set to *unchanged*. State 1 is popped and removed from Ω. It is in Θ, but $T_1 = 3$, so it is added to the register, removed from Θ, and M_1 is set to *unchanged*. Remaining copies of state 1 are removed from Ω without any change to variables.

The next tree on the input is *a(a(a,a),b(a,b))*. Function PseudoAddTree(A, $R = \{1,2,3,4,5,6\}$, *t=a(a(a,a),b(a,b))*) resets Θ and Ω, and calls function PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a(a,a),b(a,b))*), which calls PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a,a)*), which calls PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*). It increments T_1 to 4, sets M_1 to *modified*, and returns state 1 without modifying any variables. One level up, state 1 is assigned to variable r_1. Another call to PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*) follows, which this time only returns state 1, which is assigned to variable r_2. As $\delta_2(a,1,1) = 6$ and $C_6 = 1$, T_6 is incremented to 2, and M_6 is set to *modified*. State 1 is pushed twice onto Ω. State 6 is returned to be assigned to variable r_1 at the top-level invocation of function PseudoSplit. Next, PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,1)$, *t=b(a,b)*) is called, which calls PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,1)$, *t=a*). The last call returns state 1 without modifying any variables, and state 1 is assigned to variable r_1. PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,1)$,

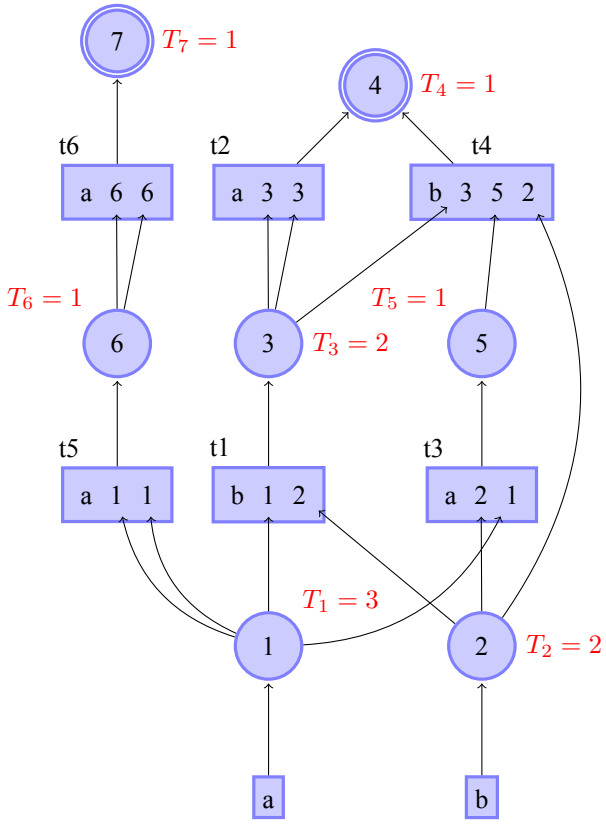


Figure 4.39: Unsorted construction of pseudo-minimal bottom-up DTAs: a tree *a(a(a,a),a(a,a))* has just been added.

t=b) is called. It sets M_2 to *modified*, increments T_2 to 3, and returns state 2, which is assigned to variable r_2 one level up. As $\delta_2(b, 1, 2) = 3$ and $C_3 = 1$, M_3 is set to *modified*, and T_3 is incremented to 3. States 1 and 2 are pushed onto Ω, and state 3 is returned to be assigned to variable r_2 at the top-level invocation of function PseudoSplit. Since $\delta_2(a, 6, 3) = \perp$, a new state 7 and a new transition $(a, 6, 3, 7)$ are created, with C_7 set to 1. T_7 is set to 1, and M_7 is set to *new*. States 6 and 3 are removed from the register, added to Θ, and pushed onto Ω, B_7 is set to $(a, 6, 3, 7)$, and state 7 is returned to function PseudoAddTree, where it is made final, it is added to Θ and pushed onto Ω. The situation is shown in Figure 4.40.

Function PseudoAddTree calls function PseudoLocMin(A, $R = \{1, 2, 4, 5\}$, $\Theta = \{3, 6, 7\}$, $\Omega = (1, 1, 1, 2, 6, 3, 7)$). State 7 is popped and removed from Ω. It is in Θ, and $T_7 = 1$, so function FindEquiv finds that state 4 is equivalent to state 7. The transition leading to state 7 is redirected towards state 4, so that C_4 is incremented to 4, state 7 is deleted, and states 3 and 6 do not need to be added to Θ since they are already there. State 7 is removed from Θ, and M_7 is set to *unchanged*. State 3 is popped and removed from Ω. It is in Θ, but $T_3 = 3$, so the state is added to the register and removed from Θ. State 6 is popped and removed from Ω. It is in Θ, but $T_6 = 2$, so state 6 is added to the register and removed from Θ. Since Θ is

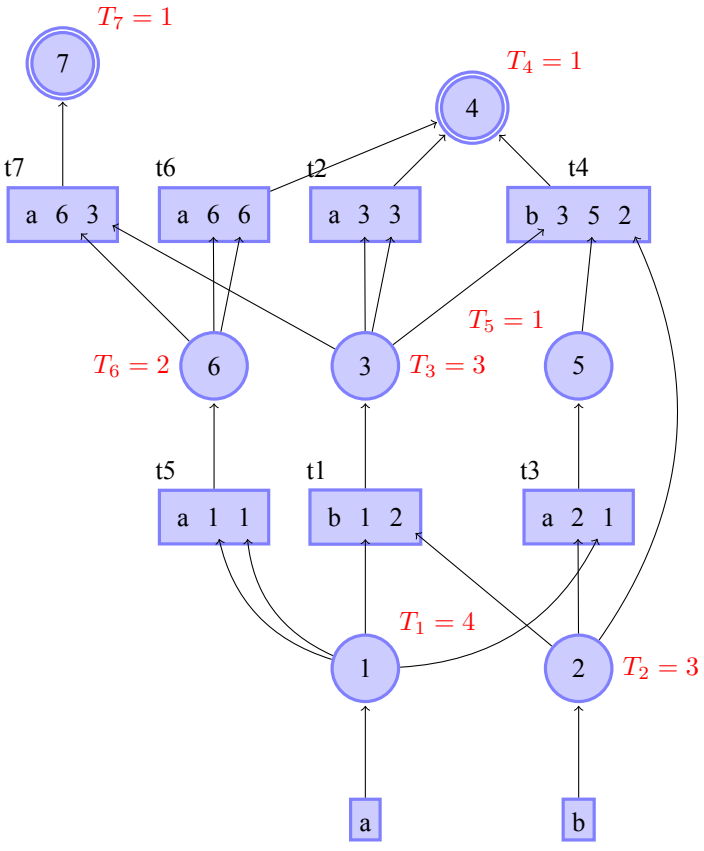


Figure 4.40: Unsorted construction of pseudo-minimal bottom-up DTAs: a tree *a(a(a,a),b(a,b))* has just been added.

now empty, the remaining states q are removed from Ω only to change M_q to *unchanged* if necessary.

The next tree on the input is *a(b(a,b),a(a,a))*. Function PseudoAddTree resets Θ and Ω, and calls function PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(b(a,b),a(a,a))*), which calls PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b(a,b)*), which calls PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*). The last call sets M_1 to *modified*, increments T_1 to 5, modifies no more variables, and returns state 1 to be assigned to variable r_1 one level up in the call hierarchy. Next, PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b*) is called. It sets M_2 to *modified*, increments T_2 to 4, changes no more variables, and returns state 2, which is assigned to variable r_2 one level up. As $\delta_2(b,1,2) = 3$ and $C_3 = 1$, M_3 is set to *modified*, and T_3 is incremented to 4. States 1 and 2 are pushed onto Ω. State 3 is returned, and it is assigned to variable r_1 in the top-level invocation of PseudoSplit. Now, PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,2)$, *t=a(a,a)*) is called, which calls PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,2)$, *t=a*). The last call returns state 1 without any change to variables. State 1 is assigned to variable r_1 one level up. An

identical call follows with identical result, but this time state 1 is assigned to variable r_2. As $\delta_2(1,1,1) = 6$ and $C_6 = 1$, M_6 is set to *modified*, and T_6 is incremented to 3. State 1 is pushed twice onto Ω, and state 6 is returned to be assigned to variable r_2 in the top-level invocation of function PseudoSplit. As $\delta_2(a,3,6) = \perp$, a new state 7 and a new transition $(a,3,6,7)$ are created, with $C_7 = 1$. T_7 is set to 1, M_7 is set to *new*, states 3 and 6 are removed from the register and are added to Θ. They are also pushed onto Ω. B_7 is set to $(a,3,6,7)$. State 7 is returned to PseudoAddTree, where it is made final, it is added to Θ, and pushed onto Ω. The situation is shown in Figure 4.41.

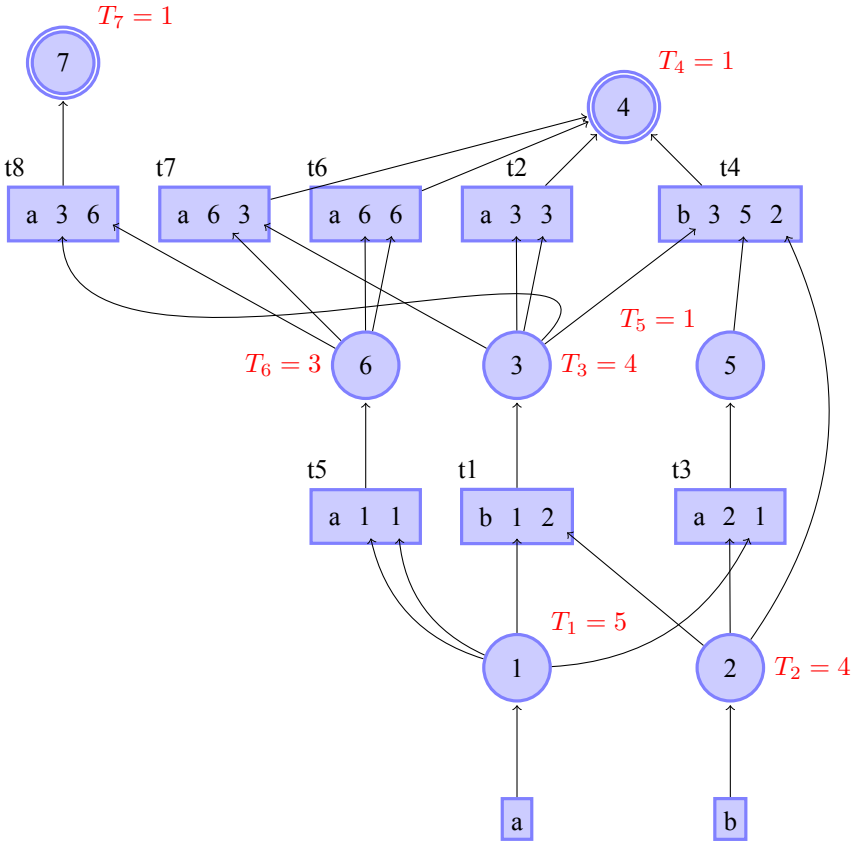


Figure 4.41: Unsorted construction of pseudo-minimal bottom-up DTAs: a tree *a(b(a,b),a(a,a))* has just been added.

Function PseudoAddTree calls PseudoLocMin(A, $R = \{1,2,4,5\}$, $\Theta = \{3,6,7\}$, $\Omega = (1,2,1,1,3,6,7)$). State 7 is popped and removed from Ω. It is in Θ, and $T_7 = 1$, so function FindEquiv finds an equivalent state -- state 4. The transition leading to state 7 is redirected to state 4, with C_4 incremented to 5. State 7 is deleted and removed from Θ, and M_7 is set to *unchanged*. State 6 is popped and removed from Ω. It is in Θ, but $T_6 = 3$, so the state is added to the register and removed from Θ. M_6 is set to *unchanged*. State 3 is popped and removed from Ω. It is in Θ, but $T_3 = 4$, so it is added to the register and removed from Θ.

Also, M_3 is set to *unchanged*. The remaining contents of Ω is only used to set appropriate M_q values to *unchanged* as those states are not in Θ.

The next string on the input is *b(a(a,a),a(b,a),b)*. Function PseudoAddTree resets Θ and Ω, and calls function PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=b(a(a,a),a(b,a),b)*), which calls PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a(a,a)*), which calls PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*). The last call sets M_1 to *modified*, increments T_1 to 6, and returns state 1, which is assigned to variable r_1 one level up. An identical call PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = ()$, *t=a*) only returns state 1, which is assigned to variable r_2 one level up. At that level, M_6 is set to *modified*, and T_6 is incremented to 4. State 1 is pushed twice onto Ω, and state 6 is returned to be assigned to variable r_1 at the top-level invocation of function PseudoSplit. A call to PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,1)$, *t=a(b,a)*) follows, which calls PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,1)$, *t=b*). The last call sets M_2 to *modified*, increments T_2 to 5, and returns state 2 to be assigned to variable r_1 one level up. A call to PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,1)$, *t=a*) returns state 1 to be assigned to variable r_2 one level up. At that level, M_5 is set to *modified*, T_5 is incremented to 2, and states 2 and 1 are pushed onto Ω. State 5 is returned and assigned to variable r_2 at the top-level invocation of function PseudoSplit. Next, PseudoSplit(A, $R = \{1,2,3,4,5,6\}$, $\Theta = \emptyset$, $\Omega = (1,1,2,1)$, *t=b*) is called, which returns state 2 to be assigned to variable r_3 at the top-level invocation of function PseudoSplit. At that level, a new state 7 and a new transition $(b,6,5,2)$ are created, with $C_7 = 1$, $T_7 = 1$, and $M_7 =$ *new*. States 6, 5, and 2 are added to Θ, removed from the register, and pushed onto Ω. B_7 is set to $(b,6,5,2)$. State 7 is returned to function PseudoAddTree, where it is made final, it is added to Θ, and pushed onto Ω.

Function PseudoAddTree calls PseudoLocMin(A, $R = \{1,3,4\}$, $\Theta = \{2,5,6,7\}$, $\Omega = (1,1,2,1,6,5,2,7)$). State 7 is popped and removed from Ω. The state is in Θ and $T_7 = 1$, so FindEquiv finds that state 4 is equivalent to state 7. The transition that leads to state 7 is redirected to state 4 with C_4 incremented to 6. State 7 is deleted and removed from Θ with M_7 set to unchanged. The situation is shown in Figure 4.42.

State 2 is popped and removed from Ω. It is in Θ, but $T_2 = 4$, so function FindEquiv is not called. Were it to be called, it would find no equivalent state. State 2 is added to the register and removed from Θ, with M_2 set to *unchanged*. State 5 is popped and removed from Ω. It is in Θ. Although it is unique, function FindEquiv has no chance to find that as $T_5 = 2$. The state is added to the register and removed from Θ, and M_5 is set to *unchanged*. State 6 is popped and removed from Ω. It is in Θ, and it is equivalent to state 3, but since $T_6 = 4$, function FindEquiv is not called, and state 6 is added to the register even though R already contains an equivalent state. It is also removed from Θ, and M_6 is set to *unchanged*. Remaining states on Ω are not in Θ, so there are no further changes to the automaton, and the DTA in Figure 4.42 is pseudo-minimal.

This construction closely follows that from the previous section. We have replaced equivalence with pseudo-equivalence based on the definition of a pseudo-minimal automaton. It remains to prove that the values of T_q, i.e. the values of $|R_A(q)|$ are calculated correctly. It is easy to see that for a given state q the value of $|R_A(q)|$ is the number of trees that need a visit to state q during their recognition by the DTA divided by $|L_A(q)|$, as all trees in $L_A(q)$ have common „tree tops". Since the construction ensures that all states q that are potential parameters of FindEquiv have $|L_A(q)| = 1$ before the call, it remains to just count the trees. New states and states that are clones of existing states get the value 1, and all other states

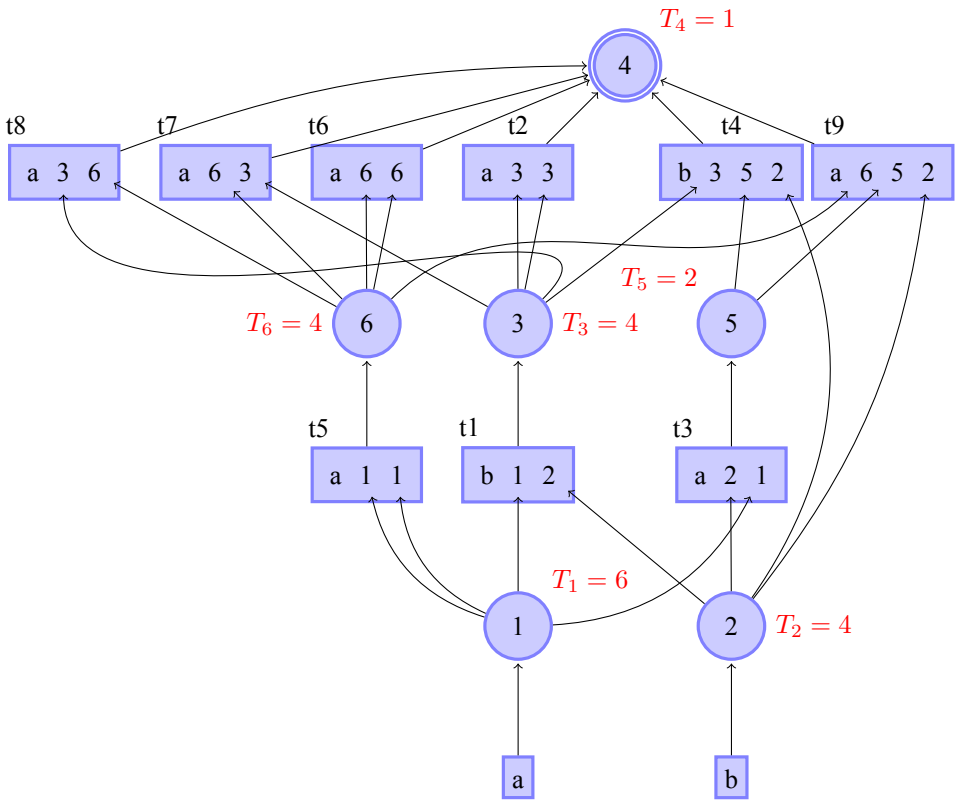


Figure 4.42: Unsorted construction of pseudo-minimal bottom-up DTAs: a tree *b(a(a,a),a(b,a),b)* has just been added. Function LocMin has just replaced state 7 with state 4, deleting state 4.

visited during a call to PseudoAddTree (except for states that are cloned, but their situation does not change anyway) get their values incremented by one. Markers M are used to ensure that the values in T change at most once during addition of a single tree, and values *new* of markers M are set before *modified*, which guaranties that new states always get the value 1. Function PseudoLocMin clears all affected markers so that they can be reused for a new tree.

As with the algorithm in the previous section, we do not give complexity estimation of the algorithm here, for the same reason. Remarks about implementation from the previous section are also valid here.

The algorithm was developed in cooperation with Rafael Carrasco. It was published in [7].

Chapter 5

Semi-Incremental Construction Algorithms

5.1 Watson's Algorithm

Watson's algorithm is an example of *semi-incremental* construction algorithm. This means that although *some* effort is taken to reduce the size of the automaton during construction, it can still be significantly larger than the minimal one. It is a trade-off between memory requirements and speed as semi-incremental algorithms are simpler than incremental ones, and at least two sources[1] show that they are also faster. The speed comes partially from a special property of input data: data must be sorted. In case of Watson's algorithm it must be sorted in any order of decreasing length. The construction is given as Algorithm 5.1. In line 5, three functions are called.

Algorithm 5.1 Watson's construction algorithm.

```
1: function WatsonConstruction
2:     Create an empty automaton M, register R, and a stack X
3:     while there are words on the input do
4:         w ← next word on the input
5:         minim(M,R,BuildStack(M,X,AddWord(M,w)))
6:     end while
7:     minim(M,R,BuildStack(M,X,q0))
8:     return M
9: end function
```

Function AddWord simply adds a word to the language of the automaton. It assumes there are no confluence states on the way, so it works as if we were adding a word to a trie. The function first uses existing states and transitions to recognize a (possibly empty) prefix of the word, and then creates states and transitions to recognize the missing ending. The function, given as Algorithm 5.2, returns the final state at the end of the path recognizing the word.

[1] Bruce Watson's paper introducing the algorithm and one less formal source

Algorithm 5.2 Function AddWord adds a word w to the language of an automaton M. It assumes that there are no confluence on the path recognizing the word. The function returns the final state created to recognize the word.

```
1:  function AddWord(M, w)
2:      q ← q_0
3:      i ← 1
4:      while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
5:          q ← δ(q, w_i)
6:          i ← i + 1
7:      end while
8:      while i ≤ |w| do
9:          δ(q, w_i) ← new state
10:         q ← δ(q, w_i)
11:         i ← i + 1
12:     end while
13:     F ← F ∪ {q}
14:     return q
15: end function
```

Function BuildStack pushes its second argument q on stack X, and so it does with all states reachable from q up to but excluding nearest final states. The states are pushed onto the stack in such order that if there is a path in the automaton leading from state q to state p, then state p is pushed earlier than state q. The stack is returned by the function, which is given as Algorithm 5.3.

Algorithm 5.3 Function BuildStack builds a stack X of states starting from state q. The stack is returned by the function.

```
1: function BuildStack(M, X, q)
2:     push q onto X
3:     for σ ∈ Σ_q do
4:         if δ(q, σ) ∉ F then
5:             X ← BuildStack(M, X, δ(q, σ))
6:         end if
7:     end for
8: end function
```

Procedure Minim performs local minimization of states stored on the stack X starting from the top of the stack. States are popped from the stack one by one. If there is an equivalent state r in the register R, the current state q is deleted, and the single transition coming to state q is redirected towards state r. If not, state q is added to the register.

Let us take an example. We construct a DFA from the following words: *bij, bijmy, bijcie, bijemy, bijecie*, coming in sorted on decreasing length: *bijecie, bijemy, bijcie, bijmy, bij*. Function WatsonConstruction creates an empty automaton $(\{0\}, \Sigma, \emptyset, 0, \emptyset)$, an empty register R, and an empty stack X. The first word ---*bijecie*--- is read from the input, and AddWord(M, w=*bijecie*) is called. It sets q to state 0, i to 1, and it skips the first loop as there are no tran-

Algorithm 5.4 Procedure Minim examines the states on the stack X to see if they are equivalent to states in the register R, and replaces them if necessary.

```
1:  procedure Minim(M, R, X)
2:      while |X| > 0 do
3:          pop q from X
4:          if ∃_{r∈R:r≡q} then
5:              redirect the single incoming transition of q towards r
6:              delete q
7:          else
8:              F ← F ∪ {q}
9:          end if
10:     end while
11: end procedure
```

sitions to follow. In the second loop, a chain of states 0, 1,..., 7 is created, as well as a chain of transitions joining those states and labeled with consecutive letters of the word as shown in Figure 5.1, with state 7 made final right after the loop.

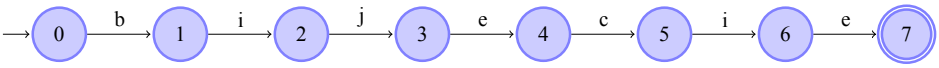


Figure 5.1: Watson construction: word *bijecie* has just been added to an empty automaton.

Function AddWord returns state 7. It is put by function BuildStack onto stack X. Since state 7 has no outgoing transitions, no other states are put onto stack, which is returned by function BuildStack to procedure Minim. That single state is compared with the empty contents of the register R. Since no equivalent state can be found in an empty set, state 7 is added to R.

Word *bijemy* is read from the input. Function AddWord(M, w=*bijemy*) is called. The first loop traverses states and transitions up to and including state 4. The second loop creates a chain of states and transitions recognizing the rest of the word, with the last word made final, as depicted in Figure 5.2.

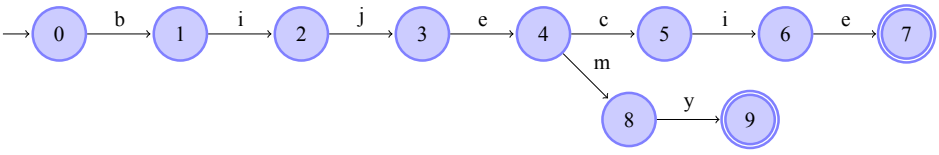


Figure 5.2: Watson construction: word *bijemy* has just been added to an automaton recognizing word *bijecie*.

State 9 is returned by function AddWord to function BuildStack, which puts the state onto stack X, and then the stack is returned to procedure Minim. In that function, state 9 is popped from the stack, and compared against the only state in the register, i.e. against state 7. As the

states are equivalent, the transition from state 8 to state 9 is redirected to state 7, and state 9 is deleted, as seen in Figure 5.3.

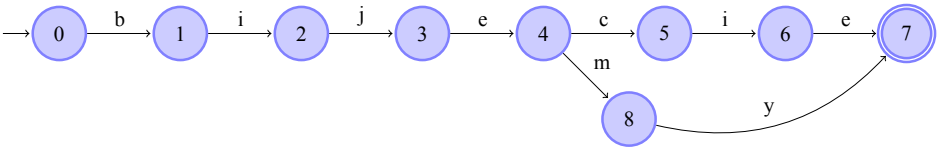


Figure 5.3: Watson construction: word *bijemy* has just been added to an automaton recognizing *bijemy*. Procedure Minim has just deleted state 9, and it redirected a transition that led to that state to state 7.

The next word is read from the input, and function AddWord(M, m=*bijcie*) is called. States and transitions up to and including state 3 are traversed in the first loop. The second loop creates states 9, 10, and 11, as well as transitions between them and between state 3 and state 9 with labels that spell out the suffix of the word. State 11 is made final, as seen in Figure 5.4.

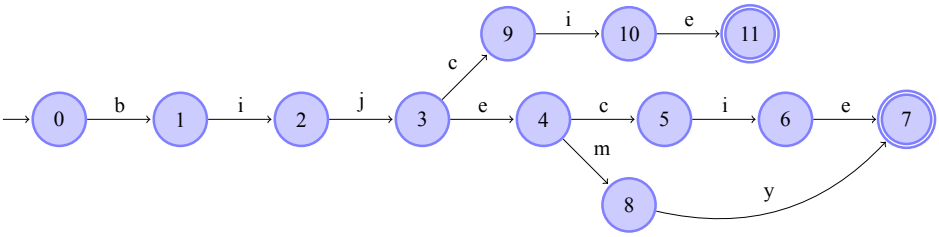


Figure 5.4: Watson construction: word *bijcie* has just been added to an automaton recognizing words *bijecie* and *bijemy*.

Function AddWord returns state 11, which is put onto stack X by function BuildStack. The stack is returned to procedure Minim that pops state 11 from it, finds that it is equivalent to state 7, redirects incoming transition of state 11 towards state 7, and deletes state 11. This situation is depicted in Figure 5.5.

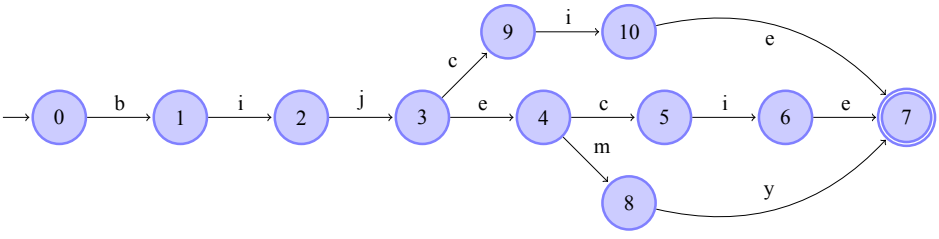


Figure 5.5: Watson construction: word *bijcie* has just been added to an automaton recognizing words *bijecie* and *bijemy*. Procedure Minim has just deleted state 11, and it redirected a transition that led to that state to state 7.

The next word on the input is read, and function AddWord(M, w=*bijmy*) is called. In the first loop, states and transitions up to and including state 3 are traversed, so that the prefix *bij* is recognized. In the second loop, states 11 and 12 are created, as well as transitions between state 3 and state 11, and between state 11 and state 12. State 12 is made final just after the loop as seen in Figure 5.6.

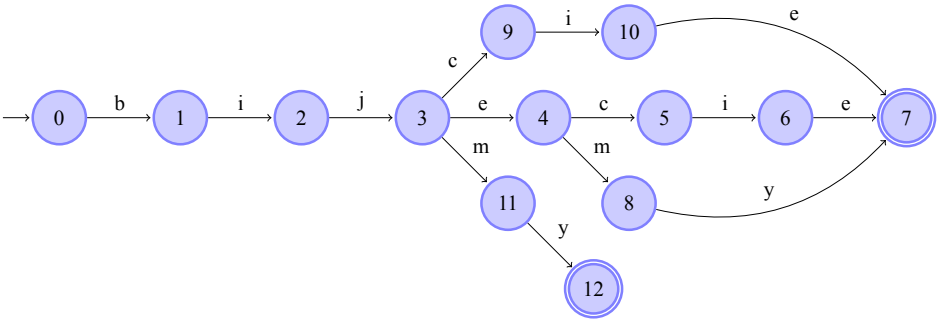


Figure 5.6: Watson construction: word *bijmy* has just been added to an automaton recognizing words *bijecie*, *bijemy*, and *bijcie*.

State 12 is returned by function AddWord to function BuildStack that puts in onto stack X. The stack is returned to procedure Minim that pops state 12 from the stack, finds that it is equivalent to state 7, redirects the sole incoming transition of state 12 to state 7, and deletes state 12, as depicted in Figure 5.7.

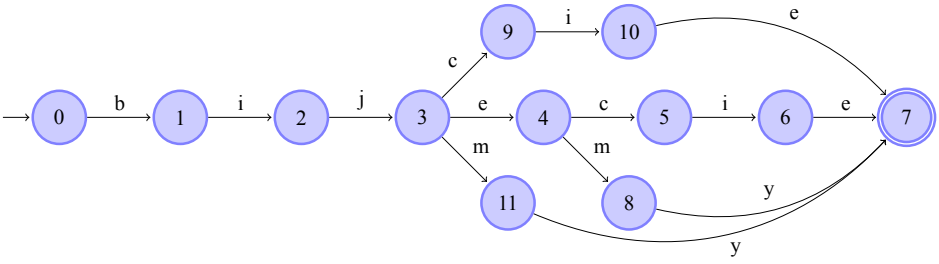


Figure 5.7: Watson construction: word *bijmy* has just been added to an automaton recognizing words *bijecie*, *bijemy*, and *bijcie*. Procedure Minim has just redirected the incoming transition of state 12 towards state 7, and deleted state 12.

The last word is read from the input, and function AddWord(M, w=*bij*) is called. The first loop traverses transitions and states up to and including state 3. The second loop does not run as the whole word has already been used for the traversal. State 3 is made final. It is returned to function BuildStack. The function puts state 3 onto stack, and examines its outgoing transitions. It calls BuildStack(M, $X = \{3\}$,9), which puts state 9 onto stack and calls BuildStack(M, $X = \{3, 9\}$, 10), which in turn puts state 10 onto stack and determines that state 7 is final, and no other states are reachable from state 10, so the control returns to the upper-level invocation, and then to the top level invocation of BuildStack. From

that level, BuildStack(M, $X = \{3, 9, 10\}$, 4) is called. It puts state 4 onto stack, and calls BuildStack(M, $X = \{3, 9, 10, 4\}$, 5). At that invocation, state 5 is put onto stack, and BuildStack(M, $X = \{3, 9, 10, 4, 5\}$, 6) is called. It puts state 6 onto stack, and determines that state 7 is final, so BuildStack cannot reprocess it. Control returns two levels up, and BuildStack(M, $X = \{3, 9, 10, 4, 5, 6\}$, 8) is called, where state 8 is put onto stack. Control returns to the top-level call of BuildStack, where BuildStack(M, $X = \{3, 9, 10, 4, 5, 6, 8\}$, 11) is called. It puts state 11 onto stack, and returns to the top level call of BuildStack, which returns the stack to procedure Minim(M, $R = \{7\}$, $X = \{3, 9, 10, 4, 5, 6, 8, 11\}$).

In procedure Minim, state 11 is popped from the stack. As the register R contains only state 7, and state 11 is different, state 11 is added to the register. State 8 is popped from the stack, and it is found equivalent to state 11, so the incoming transition of state 8 is redirected to state 11, and state 8 is deleted. State 6 is popped from the stack, and it is found unique, so it is added to the register. The same happens to states 5 and 4. State 10 is popped from the stack. It is found equivalent to state 6, so the sole incoming transition of state 10 is redirected to state 6, and state 10 is deleted. State 9 is popped from the stack. It is found equivalent to state 5, so the sole incoming transition of state 9 is redirected to state 5, and state 9 is deleted. State 3 is popped from the stack. It is found unique, so it is added to the register. The situation is shown in Figure 5.8.

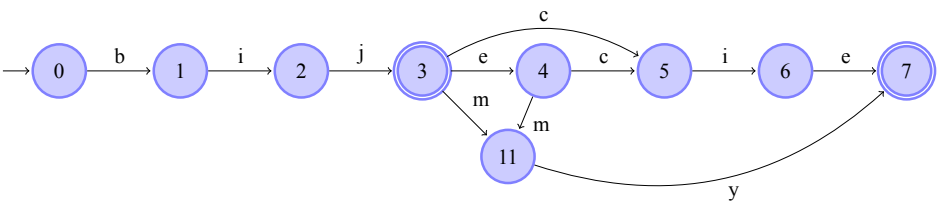


Figure 5.8: Watson construction: word *bij* has just been added to an automaton recognizing words *bijecie*, *bijemy*, *bijcie*, and *bijmy*. Procedure Minim has just deleted states 8, 9, and 10.

Finally, function BuildStack(M, $X = \emptyset$, 0) is called. State 0 is put onto stack, then BuildStack(M, $X = \{0\}$, 1) puts state 1 onto stack and calls BuildStack(M, $X = \{0, 1\}$, 2). The last call puts state 2 onto stack. It cannot call BuildStack(M, $X = \{0, 1, 2\}$, 3), as state 3 is final. The stack is returned to procedure Minim(M, $R = \{3, 4, 5, 6, 7, 11\}$, $X = \{0, 1, 2\}$). States are popped from the stack one by one, they are all found unique, and they are all added to the register. So the automaton in Figure 5.8 is minimal.

In this algorithm, just like in the algorithm for sorted data described in Chapter 3 on page 19, we have two processes running in parallel. The first one builds a trie, the second one minimizes it. Those two processes are also synchronized, but in a much simpler way. Let w be a word to be added to the language of an automaton. Let $M = (Q, \Sigma, \delta, q_0, F)$ be the automaton before a call to AddWord, and R --- the register before that call. Let $M' = (Q', \Sigma, \delta', q_0', F')$ be the automaton right after the call. Let $M'' = (Q'', \Sigma, \delta'', q_0'', F'')$ be the automaton right after the call to Minim in the main loop of the algorithm.

1. $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$
2. $\mathcal{L}(M'') = \mathcal{L}(M')$

3. M'' is minimal

To see that point 1 is true, let us notice that function AddWord creates new transitions and new states only for the part of w that is not yet stored in the automaton. If w is a prefix of a word already in $\mathcal{L}(M)$, then only $\delta^*(q_0, w)$ is made final. Otherwise let $w = uv$ such that $\delta^*(q_0, u) \in Q$ and $\delta^*(\delta(q_0, u), v_1) = \perp$. Then state $\delta^*(q_0, u)$ gets a new outgoing transition that leads to a chain of states and transitions recognizing v with state $\delta(q_0, w)$ being a final state. This process guaranties that $w \in \mathcal{L}(M')$. Also, all states are reachable and co-reachable. To prove that $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$, one must show that for any prefix w' of w, $\delta^*(q_0, w')$ is not a confluence state. A state can become confluence only as a result of minimization in procedure Minim. States can undergo that local minimization only when length of the path from q_0 that leads to them is greater than or equal to the length of the current word or at the final minimization, when no additional word can be added. Thus confluence states cannot be visited by function AddWord except for the last state, i.e. $\delta(q_0, w)$, where it does not matter.

To prove the second point, we note that the procedure Minim replaces a state with an equivalent state, which does not change the language of the automaton. Deletion of replaced states does not create unreachable states as the deleted states have the same suite of transitions as their equivalents that replace them.

To prove the third point, we notice that the states are processed in procedure Minim in such order that any state reachable from a given state is treated before the given state. This comes from the ordering of input data, and from the way function BuildStack works, i.e. it puts its argument onto the stack, and then proceeds to call itself with arguments being the targets of outgoing transitions of the current state. Since all the states are processed in this manner, there can be no pair of states that are pairwise equivalent. As there are no unreachable or not co-reachable states, the automaton has to be minimal.

The main loop in function WatsonConstruction runs n times, where n is the number of words on the input. Function AddWord has two loops that run up to $|w_{max}|$ times in total, where w_{max} is the longest word on the input. Function BuildStack is invoked at most $n|w_{max}|$ times. The inner loop is used for recursive calls, so it is already taken into account when counting invocations of BuildStack. The conditional instruction inside procedure Minim runs at most $n|w_{max}|$ times (as many times as function BuildStack has put some state onto the stack). We assume ---as before--- that register operations take constant time. Then the complexity of the whole algorithm is $\mathcal{O}(n|w_{max}|)$.

This algorithm was developed by Bruce Watson and it was published in [38].

5.1.1 Extension to Pseudo-Minimal Automata

Watson's algorithm can easily be extended to produce pseudo-minimal automata. The difficulty here is to maintain information about divergence of states. States are no longer processed in such way that we could compute that feature from the end of paths recognizing words. The solution to that problem is to keep track of cardinality of the right language in each state during the forward step of the algorithm. We introduce counters D_q for cardinality of the right language of states q. The main algorithm is practically identical to that of the original Watson's algorithm (page 131), and it is given here as Algorithm 5.5.

Function PseudoAddWord (Algorithm 5.6 is also almost identical to function AddWord

Algorithm 5.5 Extension of Watson's construction algorithm to pseudo-minimal automata.

```
1: function PseudoWatsonConstruction
2:     Create an empty automaton M, register R, and a stack X
3:     while there are words on the input do
4:         w ← next word on the input
5:         PseudoMinim(M,R,BuildStack(M,X,PseudoAddWord(M,w)))
6:     end while
7:     PseudoMinim(M,R,BuildStack(M,X,q0))
8:     return M
9: end function
```

(page 132). We maintain counters D_q for $|\overrightarrow{\mathcal{L}}(q)|$ using a method borrowed from Section 4.5.

Algorithm 5.6 Function PseudoAddWord adds a word w to the language of an automaton M. It assumes that there are no confluence on the path recognizing the word. The function returns the final state created to recognize the word. It also maintains counters D_q for $|\overrightarrow{\mathcal{L}}(q)|$ of states q.

```
1: function PseudoAddWord(M, w)
2:     q ← q0
3:     D_q ← D_q + 1
4:     i ← 1
5:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
6:         q ← δ(q, w_i)
7:         D_q ← D_q + 1
8:         i ← i + 1
9:     end while
10:    while i ≤ |w| do
11:        δ(q, w_i) ← new state
12:        q ← δ(q, w_i)
13:        D_q ← 1
14:        i ← i + 1
15:    end while
16:    F ← F ∪ {q}
17:    return q
18: end function
```

Function BuildStack can be used without modifications. Function PseudoMinim (Algorithm 5.7) uses the counters that were introduced earlier to transform equivalence into pseudo-equivalence.

Let us see how the algorithm works with an example. On the input we have words: *biliście, biliśmy, bijecie, bijemy, bijcie, bijmy*, and *bij*. Function PseudoWatsonConstruction creates an empty automaton with the initial state 0 and $D_0 = 0$, and reads *biliście* from the input. Function PseudoAddWord(M, w=*biliście*) is called. It sets q to state 0, Increments D_0 to 1, and sets i to 1. The first `while` loop does not run as there are no transitions to traverse. The second `while` loop creates states 1, 2,…,8, setting D_q for each of those states to 1, and it

Algorithm 5.7 Procedure PseudoMinim examines the states on the stack X to see if they are equivalent to states in the register R, and replaces them if necessary.

```
1:  procedure PseudoMinim(M, R, X)
2:      while |X| > 0 do
3:          pop q from X
4:          if D_q = 1 ∧ ∃_{r∈R:r≡q} then
5:              redirect the single incoming transition of q towards r
6:              delete q
7:          else
8:              R ← R ∪ {q}
9:          end if
10:     end while
11: end procedure
```

creates transitions between the states, then state 8 is made final, as can be seen in Figure 5.9. State 8 is returned to function BuildStack, which puts it onto stack. The stack is returned to procedure PseudoMinim, which pops state 8 from it, and as the register is empty, it adds state 8 to R. Figure 5.9 is still valid.

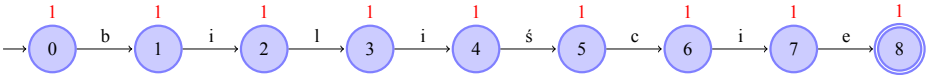


Figure 5.9: Watson construction of pseudo-minimal automata: word *biliście* has just been added to an empty automaton.

Function PseudoWatsonConstruction reads word biliśmy, and calls PseudoAddWord(M, w=biliśmy). Going through the first loop, the function traverses states and transitions up to and including state 5, incrementing D_q for each of them by one to 2. In the second loop, states 9 and 10 are created, with $D_9 = D_{10} = 1$, and transitions to both of those states. State 10 is made final just after the loop, and it is returned to function BuildStack, which puts it onto stack. The stack is returned to procedure PseudoMinim that pops state 10 from the stack. Since $D_{10} = 1$ and state 8 is equivalent to state 10, state 10 is deleted, and the incoming transition of state 10 is redirected to state 8. This is shown in Figure 5.10.

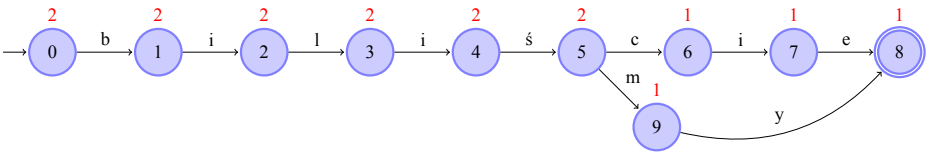


Figure 5.10: Watson construction of pseudo-minimal automata: word *biliśmy* has just been added to an automaton recognizing word *biliście*.

The next word on the input is *bijecie*. Function PseudoAddWord traverses states and transitions from the initial state up to and including state 2, incrementing their D_q to 3. The

second loop creates states 10, 11, 12, 13, and 14, with q_q of each state set to 1. It also creates transitions leading to them. State 14 is made final, and it is returned to function BuildStack, which puts in onto stack. The stack is returned to procedure PseudoMinim, which pops state 14. Since $D_{13} = 1$ and state 8 is equivalent to state 14, the incoming transition of state 14 is redirected to state 8, and state 14 is deleted. The automaton is shown in Figure 5.11.

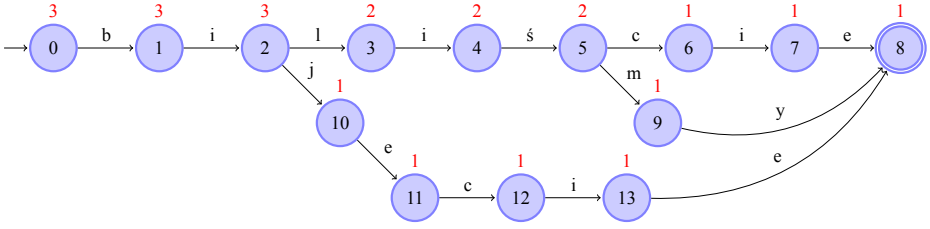


Figure 5.11: Watson construction of pseudo-minimal automata: word *bijecie* has just been added to an automaton recognizing words *biliście* and *biliśmy*.

The next word on the input is *bijemy*. In the first `while` loop in function PseudoAddWord, states and transitions up to and including state 11 are traversed, incrementing their D_q. In the second `while` loop, states 14 and 15 as well as transitions leading to them are created. States receive $D_q = 1$. After the loop, state 15 is made final, and it is returned to function BuildStack, where it is put onto stack. The stack is returned to procedure PseudoMinim. State 15 is popped. As $D_{15} = 1$ and state 15 is equivalent to state 8, the incoming transition of state 15 is redirected towards state 8, and state 15 is deleted, as shown in Figure 5.12.

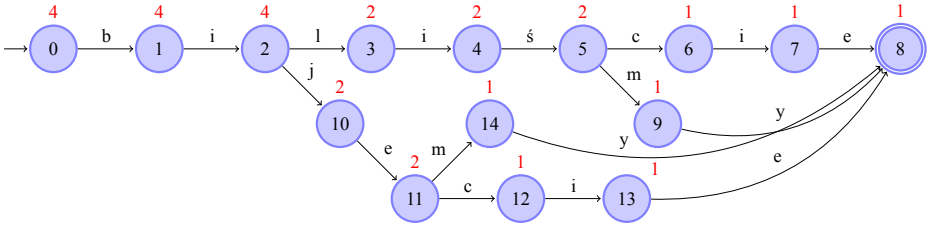


Figure 5.12: Watson construction of pseudo-minimal automata: word *bijemy* has just been added to an automaton recognizing words *biliście biliśmy*, and *bijecie*.

The next word on the input is *bijcie*. In the first `while` loop of function PseudoAddWord, states and transitions up to and including state 10 are traversed, and D_q for states are incremented. In the second `while` loop, states 15, 16, and 17 are created with $D_q = 1$, as well as transitions leading to them. After the loop, state 17 is made final, and it returned to function BuildStack, where it is put onto stack. The stack is returned to procedure PseudoMinim, where state 17 is popped. As $D_{17} = 1$ and state 8 is found equivalent to state 17, the incoming transition of state 17 is redirected towards state 8, and state 17 is deleted. Figure 5.13 shows the resulting automaton.

The next word is *bijmy*. In the first `while` loop of function PseudoAddWord, states and transitions up to and including state 10 are traversed, with their D_q incremented. In the second

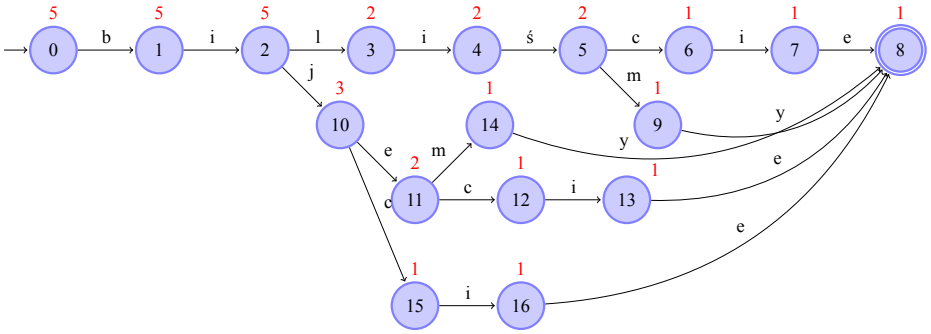


Figure 5.13: Watson construction of pseudo-minimal automata: word *bijcie* has just been added to an automaton recognizing words *biliście biliśmy*, *bijecie*, and *bijemy*.

`while` loop, states 17 and 18 with $D_q = 1$, as well as transitions that lead to them, are created. After the loop, state 18 is made final, and it is returned to function BuildStack, that puts it onto stack. The stack is returned to procedure PseudoMinim. State 18 is popped from the stack. As $D_{18} = 1$ and state 18 is equivalent to state 8, the incoming transition of state 18 is redirected towards state 8, and state 18 is deleted. The resulting automaton is shown in Figure 5.14.

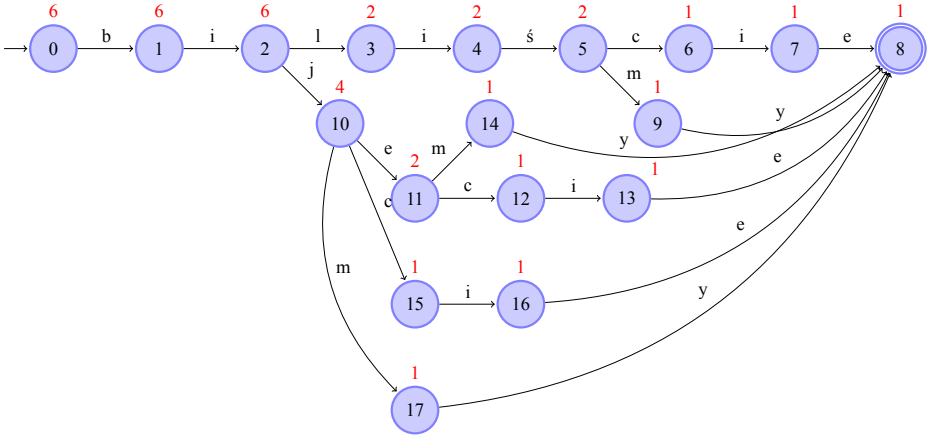


Figure 5.14: Watson construction of pseudo-minimal automata: word *bijmy* has just been added to an automaton recognizing words *biliście biliśmy*, *bijecie*, *bijemy*, and *bijcie*.

The last word is *bij*. In the first `while` loop in function PseudoAddWord, states and transitions up to and including state 10 are traversed with their D_q incremented. The second `while` loop does not run as all letters of the word have been already consumed. State 10 is made final, and it is returned to function BuildStack. State 10 is put onto stack, and BuildStack is called again with state 11. State 11 is put onto stack, and BuildStack(M, X, 12) is called. State 12 is put onto stack, and BuildStack(M, X, 13) is invoked. State 13 is put onto stack. As state 8 if final, recursion stops here, and control returns two levels up to call BuildStack(M,

X, 14). State 14 is put onto stack, and we are back at the top-level invocation of function BuildStack, which calls itself with state 15 now. State 15 is put onto stack, and BuildStack(M, X, 16) is called. State 16 is put onto stack, and BuildStack(M, X, 17) puts state 17 onto stack. The stack $X = (10, 11, 12, 13, 14, 15, 16, 17)$ is returned to procedure PseudoMinim.

In that procedure, state 17 is popped. Although $D_{17} = 1$, the register contains only state 8, which is not equivalent to state 17, so state 17 is added to the register. State 16 is popped. Again, $D_{16} = 1$ but there is no equivalent state in the register, so state 16 is added to R. State 15 undergoes the same actions. State 14 is popped from the stack. It also has $D_{14} = 1$, and this time, there is an equivalent state --- state 17. The incoming transition of state 14 is redirected to state 17, and state 14 is deleted. State 13 is popped. With $D_{13} = 1$, the state is found to be equivalent to state 16, so the incoming transition of state 13 is redirected to state 16, and state 13 is deleted. State 12 is popped. With $D_{12} = 1$, it is found equivalent to state 15, so the incoming transition of state 12 is redirected towards state 15, and state 12 is deleted. State 11 is popped. Since $D_{11} = 2$, the state is added to the register. State 10 is popped. As $D_{10} = 4$, the state is added to the register.

There are no more words on the input. Function BuildStack is called again, this time with state 0 as its last parameter. State 0 is put onto stack, then BuildStack calls itself with state 1, which is put onto stack. Repeated invocations put states 2, 3, 4, 5, 6, 7, and 9 onto the stack. The stack is returned to procedure PseudoMinim. State 9 is popped from the stack. With $D_9 = 1$, it is found to be equivalent to state 17, so the incoming transition of state 9 is redirected to state 17, and state 9 is deleted. State 7 is popped. With $D_7 = 1$, it is found being equivalent to state 16, so the incoming transition of state 7 is redirected towards state 16, and state 7 is deleted. State 6 is popped. With $D_6 = 1$, it is found to be equivalent to state 15, so the incoming transition of state 6 is redirected to state 15, and state 6 is deleted. State 5 is popped. It is equivalent to state 11, but as $D_5 = D_{11} = 2$, state 5 is only added to the register. Note that we have two equivalent states in the register now. It has no adverse effects as divergent states are never searched for in the register. State 4 is popped, but as $D_4 = 2$, it is added to the register. The same happens with states 3, 2, 1, and 0. See Figure 5.15.

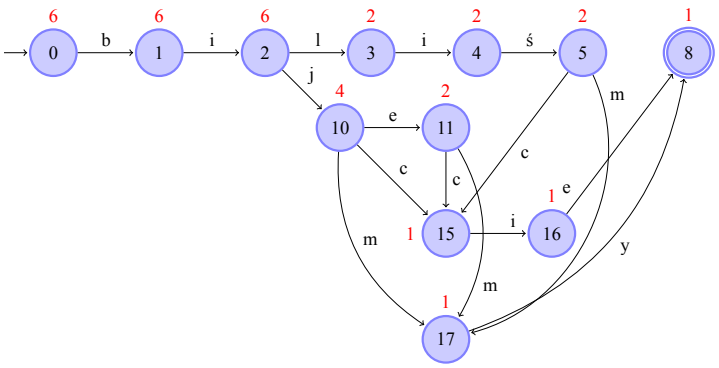


Figure 5.15: The pseudo-minimal automaton recognizing words *biliście biliśmy*, *bijecie*, *bijemy*, *bijcie*, and *bij*.

Note that only final states (except for one) have been merged into one before the final minimization. This is in sharp contrast with incremental methods.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an automaton before a next word w is added, let R be the register at that time, let $M' = (Q', \Sigma, \delta', q'_0, F')$ be the automaton right after function PseudoAddWord returns, let R' be the register at that time, let $M'' = (Q'', \Sigma, \delta'', q''_0, F'')$ be the automaton after the invocation of PseudoMinim right after the word had been added was finished. To prove that the algorithm is correct, we need to prove that:

1. $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$
2. $\mathcal{L}(M'') = \mathcal{L}(M')$
3. $|M''| \leq |M'|$
4. final minimization gives the minimal automaton for the language of the automaton.

The proof of the first point is exactly the same as in the previous algorithm. The difference between PseudoAddWord (page 138) and AddWord (page 132) is that PseudoAddWord maintains counters D_q. But this does not change the language of the automaton. Also, procedure PseudoMinim cannot create new incoming transitions for the states visited in function PseudoAddWord.

For the second point, we also make reference to the main algorithm. We replace states with equivalent ones. When states are deleted, transitions that lead to them are redirected towards equivalent states, so states remain co-reachable. Deleted states have their equivalent counterparts, so every target of an outgoing transition of a deleted state is also a target of an outgoing transition of its equivalent counterpart. Thus, PseudoMinim does not create unreachable states.

The third point is obvious. Local pseudo-minimization deletes states while keeping the language of the automaton intact, and it does not create any new states, so the size of the automaton can only shrink.

As to the fourth point, notice that states are passed to function BuildStack and then popped from the stack in procedure PseudoMinim in such a way that equivalence (pseudo-equivalence is equivalence with an additional condition) can use the recursive definition of the right language where equivalence of target states of transitions is replaced with identity of target states of transitions. This holds for both the invocation after each word has been added and the final invocation of procedure PseudoMinim. As all states are passed to the procedure, and the language of the automaton does not change, the automaton is pseudo-minimal.

The `while` loop in function PseudoWatsonConstruction runs n times, where n is the number of words on the input. Function PseudoAddWord is called n times. Inside it, all simple actions run in constant time (finality should be implemented as a flag, not as a set). The two `while` loops run at most $|w_{max}|$ times, where w_{max} is the longest word on the input. Function BuildStack is invoked at most $n|w_{max}|$ times including recursive calls, as it is invoked for each state in a trie (the trie is never represented in full, as it is minimized on the fly). Each call of BuildStack runs in constant time excluding the time spent in subordinate calls. The `while` loop in procedure PseudoMinim runs as many times in total, as function BuildStack was called in total. Since we assume as before that register operations take constant time, this gives us $\mathcal{O}(n|w_{max}|)$ for the whole algorithm.

This algorithm was developed by the author with Denis Maurel and Agata Savary, and it was first published in [17].

5.1.2 Extension to Cyclic Automata

This extension relies on the same concept as extensions of the incremental algorithms. The initial state is cloned, which creates confluence states that form a barrier between the old and the new, modified part of the automaton. As a cyclic automaton cannot be created by this algorithm, we start with a non-empty, cyclic automaton, and we add words to it. Confluence states may be encountered, and they are cloned.

Algorithm 5.8 Extension of Watson's construction algorithm to cyclic automata.

```
 1: function CyclicWatsonConstruction(M,R)
 2:     Create an empty stack X
 3:     q' ← q0
 4:     q0 ← Clone(q0)
 5:     while there are words on the input do
 6:         w ← next word on the input
 7:         Minim(M,R,CyclicBuildStack(M,X,CyclicAddWord(M,w)))
 8:     end while
 9:     Minim(M,R,CyclicBuildStack(M,X,q0))
10:     if q' ≠ q0 then
11:         Increment incoming transition counter for q'
12:         DeleteBranchUp(M, q')
13:     end if
14:     return M
15: end function
```

The main function of the extension is given as Algorithm 5.8. The original initial state is stored in q'. It is then cloned. The `while` loop resembles that of the original algorithm, but names of functions are different to underline modifications. Just as in other extensions to cyclic automata, a final check is done whether the initial state is different from the original initial state stored in q'. If it is so, then procedure DeleteBranchUp is called to delete unreachable states, if any.

Function CyclicAddWord, given here as Algorithm 5.9, differs from function AddWord (page 132) by an additional loop where confluence states are cloned. The first `while` loop contains an additional condition checking whether the state in question is confluence. The function now closely resembles the first part of a similar function in the unsorted incremental construction (see page 65, function AddStrUnsorted). However, it is used differently. The second `while` loop, i.e. the one that clones confluence states, is run only if states of the original automaton are encountered. Otherwise, the function behaves exactly like the original one. Procedure Minim is used without modifications, as function CyclicBuildStack assures that only new and modified states are on the stack.

Let us examine how the algorithm works using an example. The initial automaton will be the same as in Figure 3.35, repeated here as Figure 5.16. On the input we have words: *bijecie, bijemy, bijcie, bijmy*, and *bij* --- the same set as in the example for the original algorithm.

An empty stack X is created, the original initial state is stored in q', q_0 is cloned as state 6, and q_0 becomes state 6. Word *bijecie* is read form the input, and function CyclicAddWord(M, w=*bijecie*) is called. Variable q becomes state 6, i becomes 1. By cloning state 0, transitions

Algorithm 5.9 Function CyclicAddWord adds a word w to the language of an automaton M. It does not assume that there are no confluence on the path recognizing the word. If there are any, they are cloned. The function returns the final state created to recognize the word.

```
1:  function CyclicAddWord(M, w)
2:      q ← q_0
3:      i ← 1
4:      while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ ∧ FanIn(δ(q, w_i)) = 1 do
5:          q ← δ(q, w_i)
6:          i ← i + 1
7:      end while
8:      while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
9:          δ(q, w_i) ← Clone(δ(q, w_i))
10:         q ← δ(q, w_i)
11:         i ← i + 1
12:     end while
13:     while i ≤ |w| do
14:         δ(q, w_i) ← new state
15:         q ← δ(q, w_i)
16:         i ← i + 1
17:     end while
18:     F ← F ∪ {q}
19:     return q
20: end function
```

Algorithm 5.10 Function CyclicBuildStack builds a stack X of states starting from state q. Confluence states that form a boundary between the old, intact part of the automaton, and the new, modified one, are never put onto stack or used as arguments of recursive calls. The stack is returned by the function.

```
1:  function CyclicBuildStack(M, X, q)
2:      push q onto X
3:      for σinΣ_q do
4:          if δ(q, σ) ∉ F ∧ FanIn(δ(q, σ)) = 1 then
5:              X ← CyclicBuildStack(M, X, δ(q, σ))
6:          end if
7:      end for
8:  end function
```

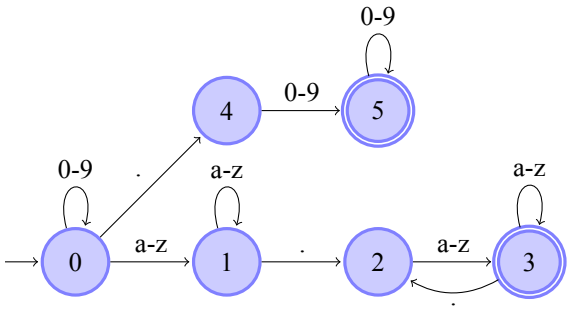


Figure 5.16: A minimal automaton recognizing simplified host names in URLs following the pattern $[a-z]^+(\.[a-z]^+)^+$, and recognizing real numbers in format $[0-9]^*\.[0-9]^+$. A transition labeled with *a-z* stands for many transitions labeled with consecutive letters *a, b, c...z*.

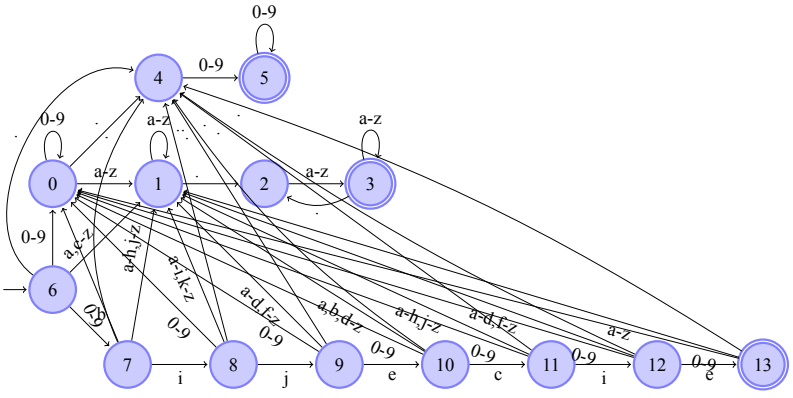


Figure 5.17: Word *bijecie* has just been added using Watson's algorithm extended to cyclic automata to a minimal automaton recognizing simplified host names in URLs following the pattern $[a-z]^+(\.[a-z]^+)^+$, and recognizing real numbers in format $[0-9]^*\.[0-9]^+$.

from state 6 to state 1 were created, so even if state 1 were not confluence in the original automaton, it becomes confluence after the cloning. As $\delta(6, b) = 1$ and state 1 is confluence, the first `while` loop in function CyclicAddWord does not run. In the second `while` loop, state 1 is cloned as state 7, q becomes state 7, and i is incremented to 2. Now $\delta(7, i)$ points again to state 1, so it is cloned again as state 8, q becomes state 8, and i is incremented to 3. Transitions labeled with subsequent letters of w always point to state 1, which is continually cloned as state 9, 10, 11, 12, and 13. State 13 is made final, and it is returned by the function to function CyclicBuildStack. The resulting automaton is shown in Figure 5.17. Function CyclicBuildStack puts state 13 onto the stack. Transitions from state 13 lead to states 0, 1, and 4. They are all confluence, so no more states are put onto the stack, which is returned to procedure Minim. In the procedure, state 13 is found to be unique, so it is added to the register, which now contains states 0, 1, ..., 6, and 13.

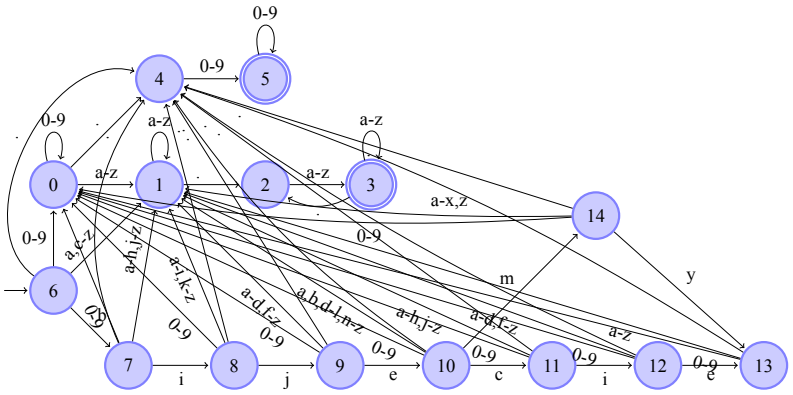


Figure 5.18: Words *bijecie* and *bijemy* have just been added using Watson's algorithm extended to cyclic automata to a minimal automaton recognizing simplified host names in URLs following the pattern $[a-z]^+(\backslash.[a-z]^+)^+$, recognizing real numbers in format $[0-9]^*\backslash.[0-9]^+$.

The next word on the input is *biliśmy*. Function CyclicAddWord(M, w=*biliśmy*) is called. Variable q is set to state 6, and i is set to 1. In the first `while` loop, transitions are traversed until state 11 is reached, so that $q = 11$ and $i = 6$. Transition $\delta(11, m) = 1$, and state 1 is confluence, so state 1 is cloned as state 14 in the second `while` loop, and a transition from state 11 to state 1 is labeled with *m* is redirected to state 14. In the next run of the loop, state 1 is cloned as state 15, and a transition from state 14 to state 1 labeled with *y* is redirected towards state 15. As there are no more letters in w, the loop ends, and state 15 is made final. It is returned to function CyclicBuildStack, which puts it onto the stack. Transitions from state 15 lead to states 0, 1, and 4, which are all confluence, so no more states are put onto the stack. The stack is returned to procedure Minim. In the procedure, state 15 is popped from the stack. State 13, which is present in the register, is found equivalent to state 15, so the incoming transition of state 15 is redirected towards state 13, and state 15 is deleted, as shown in Figure 5.18.

The next word from the input is read, and function CyclicAddWord(M, w=*bijcie*) is called. In the first `while` loop, transitions are traversed until state 9 is reached with $i = 4$. In the second `while` loop, state 1 is cloned as state 15, and the transition from state 9 to state 1 labeled with *c* is redirected to state 15. In the second run of the loop, state 1 is cloned again as state 16, and the transition from state 15 to state 1 labeled with *i* is redirected to state 16. In the last run of state loop, state 1 is cloned yet again as state 17, and the transition from state 16 to state 1 labeled with *e* is redirected to state 17. State 17 is made final, and it is returned to function CyclicBuildStack, which puts it onto the stack. Transitions from state 17 go to states 0, 1, and 4, which are all confluence, so no further states are put onto the stack, and it is returned to procedure Minim. In that procedure, state 17 is popped from the stack. It is found equivalent to state 13, so the transition from state 16 to state 17 is redirected to state 13, and state 17 is deleted, as depicted in Figure 5.19.

The next word on the input is read, and function CyclicAddWord(M, w=*bijcie*) is called. In the first `while` loop, transitions up to and including state 9 are traversed. In the second

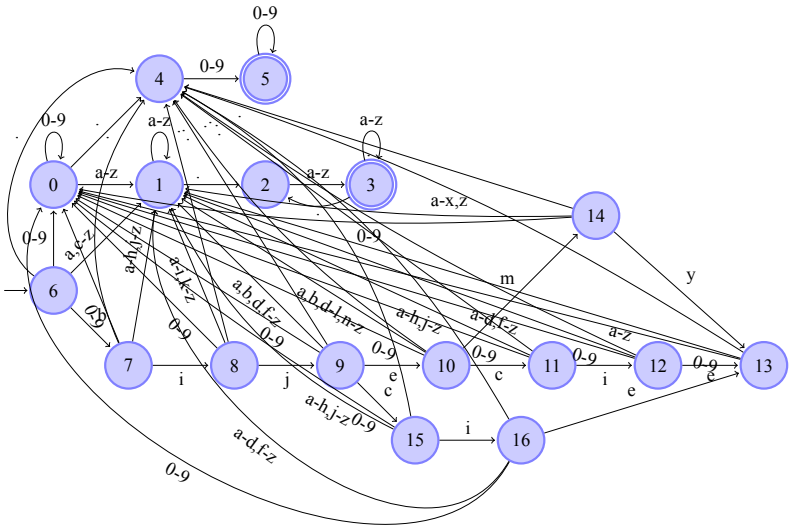


Figure 5.19: Words *bijecie*, *bijemy*, and *bijcie* have just been added using Watson's algorithm extended to cyclic automata to a minimal automaton recognizing simplified host names in URLs following the pattern $[a-z]^+(\backslash.[a-z]^+)^+$, recognizing real numbers in format $[0-9]^*\backslash.[0-9]^+$.

`while` loop, state 1 is cloned as state 17, and the transition from state 9 to state 1 labeled with *m* is redirected towards state 17. In the second run of that loop, state 1 is cloned again as state 18, and the transition from state 17 to state 1 labeled with *y* is redirected towards state 18. State 18 is made final, and it is returned to function CyclicBuildStack. State 18 is put onto the stack. All transitions from state 18 lead to states 0, 1, and 4, which are all confluence, so no more states are put onto the stack. In procedure Minim, state 18 is popped from the stack. It is found equivalent to state 13, so the transition from state 17 to state 18 is redirected to state 13, and state 18 is deleted. The situation is shown in Figure 5.20.

The next word on the input is read, and function CyclicAddWord(M, w=*bij*) is called. In the first `while` loop, state 9 is reached. As there are no more letters in w, the second `while` loop does not run. State 9 is made final, and it is returned to function CyclicBuildStack. In that function, state 9 is put onto the stack. Transitions from state 9 go to states 0, 1, 4, 10, 15, and 17. The first three are confluence, so they are skipped. Next, CyclicBuildStack(M, $X = (9)$, 10) is called. State 10 is put onto the stack. Transitions from state 10 go to states 0, 1, 4, 11, and 14. The first three are confluence, so they are skipped, and CyclicBuildStack(M, $X = (9, 10)$, 11) is called. It puts state 11 onto the stack. Transitions from state 11 go to states 0, 1, 4, and 12. The first three are confluence, so they are skipped, and CyclicBuildStack(M, $X = (9, 10, 11)$, 12) is called. It puts state 12 onto the stack. Transitions from state 12 go to states 0, 1, 4, and 13, they are all confluence, and additionally, state 13 is final. Two

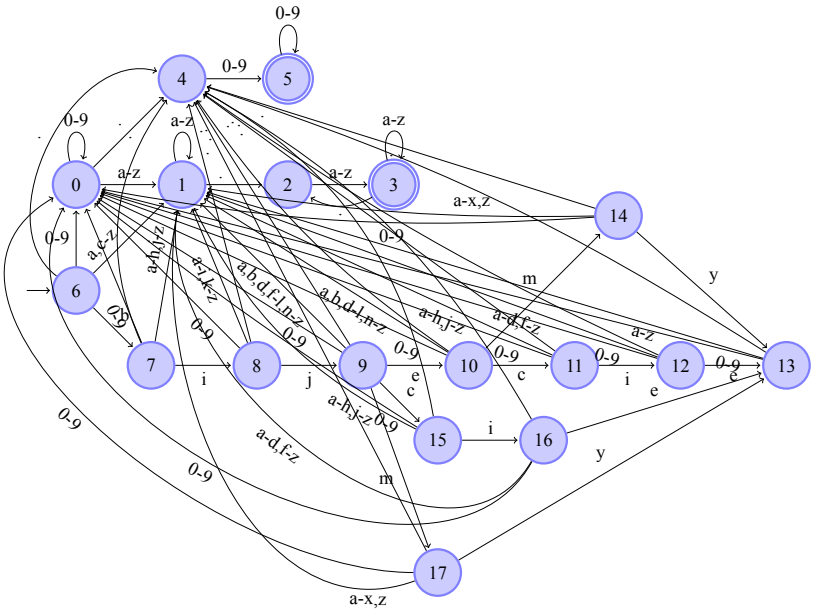


Figure 5.20: Words *bijecie*, *bijemy*, *bijcie*, and *bijmy* have just been added using Watson's algorithm extended to cyclic automata to a minimal automaton recognizing simplified host names in URLs following the pattern $[a-z]^+(\.[a-z]^+)^+$, recognizing real numbers in format $[0-9]^*\.[0-9]^+$.

recursive calls end, and CyclicBuildStack(M, $X = (9, 10, 11, 12)$, 14) is called. It puts state 14 onto the stack. Targets of all transitions leaving state 14 are confluence. Recursive calls terminate so that we are back in state 9, and CyclicBuildStack(M, $X = (9, 10, 11, 12, 14)$, 15) is called. State 15 is put onto the stack. Transitions from state 15 go to states 0, 1, 4, and 16, and only state 16 is not confluence on that list. Function CyclicBuildStack(M, $X = (9, 10, 11, 12, 14, 15)$, 16) is called. State 16 is put onto the stack. All states that are targets of transitions going out from state 16 are confluence. Two recursive calls end, and CyclicBuildStack(M, $X = (9, 10, 11, 12, 14, 15, 16)$, 17) is called. State 17 is put onto the stack. Since all transitions from state 17 go to confluence states, no more states are put onto the stack, and all recursive calls of CyclicBuildStack terminate. The stack is returned to procedure Minim.

In procedure Minim, state 17 is popped from the stack. There is no state in the register that is equivalent to state 17, so state 17 is added to the register. State 16 is popped from the stack. It is also found unique, so it is added to the register. State 15 is popped from the register, and yet again, it is found unique, so it is added to the register. State 14 is popped from the register. It is equivalent to state 17, which is already in the register. The transition from state 10 to state 14 labeled with *m* is redirected to state 17. State 14 is deleted. State 12 is popped from the stack. It is equivalent to state 16, so the transition from state 11 to state 12 labeled with *i* is redirected towards state 16, and state 12 is deleted. State 11 is popped from the stack. It is

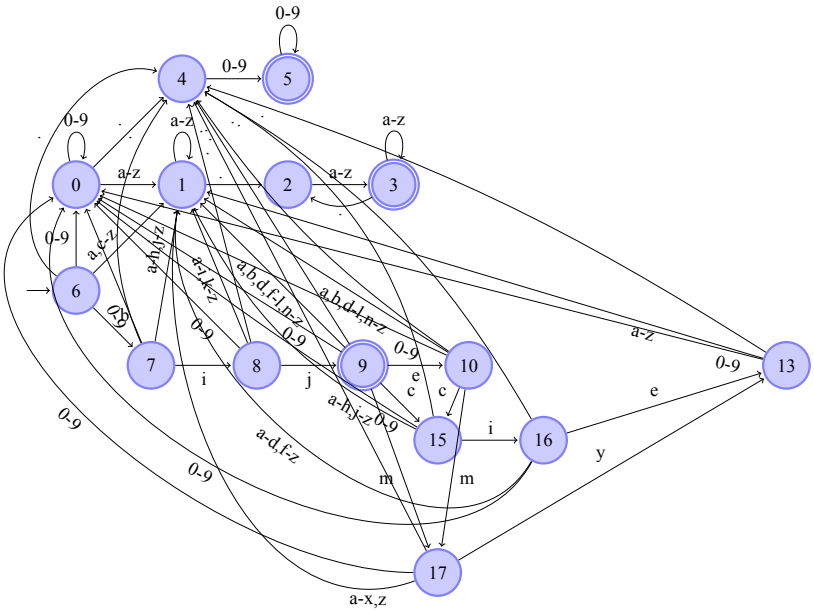


Figure 5.21: Words *bijecie*, *bijemy*, *bijcie*, *bijmy*, and *bij* have just been added using Watson's algorithm extended to cyclic automata to a minimal automaton recognizing simplified host names in URLs following the pattern $[a-z]^+(\backslash.[a-z]^+)^+$, recognizing real numbers in format $[0-9]^*\backslash.[0-9]^+$.

equivalent to state 15, so the transition from state 10 to state 11 labeled with *c* is redirected to state 15, and state 11 is deleted. State 10 is popped from the stack. It is unique, so it is added to the register. State 9 is popped from the register, and it is also unique, so it is also added to the register. The resulting automaton is shown in Figure 5.21.

There are no more words on the input. The loop in function CyclicWatsonConstruction terminates. Function CyclicBuildStack(M, $X = ()$, 6) is called. It puts state 6 onto the stack, and it calls CyclicBuildStack(M, $X = (6)$, 7), as states 0, 1, and 4, which are also targets of transitions going out from state 6, are confluence. State 7 is put onto the stack. States 0, 1, 4 that are targets of transitions going out from state 7 are skipped as they are confluence, and CyclicBuildStack(M, $X = (6, 7)$, 8) is called. It puts state 8 onto the stack. All the recursive calls of CyclicBuildStack terminate, and the stack is returned to procedure Minim. In that procedure, state 8 is popped from the stack. It is unique, so it is added to the register. The same thing happens to states 7 and 6. The automaton in Figure 5.21 is minimal.

Let $M = (Q, \Sigma, \delta, q_0, F)$ be an automaton before addition of a new word w. Let R be the contents of the register before that event. Let $M' = (Q', \Sigma, \delta', q_0', F')$ be the automaton after function CyclicAddWord has been called to add w to the language of the automaton. Let $M'' = (Q'', \Sigma, \delta'', q_0'', F'')$ be the automaton after a call to procedure Minim has acted on some states added by function CyclicAddWord. Let $M''' = (Q''', \Sigma, \delta''', q_0''', F''')$ be the automaton after the last call to procedure Minim inside the loop in function CyclicWatson-

Construction. Let $M'''' = (Q'''', \Sigma, \delta'''', q_0'''', F'''')$ be the automaton obtained after the final call to procedure Minim after the `while` loop in function CyclicWatsonConstruction.

1. $\mathcal{L}(M') = \mathcal{L}(M) \cup \{w\}$
2. $\mathcal{L}(M'') = \mathcal{L}(M')$
3. $\mathcal{L}(M'''') = \mathcal{L}(M''')$
4. M'''' is minimal

Initial cloning of the initial state does not change the language of the automaton. Function CyclicAddWord recognizes a (possibly empty) prefix of w already contained in the automaton, and then it creates a (possibly empty) chain of states and transitions recognizing the (possibly empty) so far unrecognized part of w. Thus $w \in \mathcal{L}(M')$. The question is whether something else is deleted or added. Deletion could only happen in CyclicAddWord when redirecting transitions. However, transitions are redirected to equivalent states that are clones of other states. Additionally, cloned states that lose one of their incoming transitions have other incoming transitions, since they are always confluence states. Addition of strings, other than w, could happen if any state in the path recognizing w were confluence. This is impossible, as all confluence states in that path are cloned, and their clones have only one incoming transition. Thus the first point is proven.

In function Minim, targets of transitions are replaced with equivalent states, with the original targets deleted. Replacing states with equivalent states does not change the language of the automaton. Additionally, since the state being deleted has an equivalent state, no states become unreachable. Thus, point 2 is proven.

Point 3 is analogous to point 2: procedure Minim replaces states with equivalent states, which cannot change the language of the automaton.

To prove point 4, notice that each state created in function CyclicAddWord (either a new state, or a clone of an existing one) is processed exactly once. Function CyclicBuildStack never puts unmodified states onto the stack, as they are always confluence. States are put onto the stack in preorder, so they are processed in postorder. This also concerns states in different invocations of function CyclicBuildStack. As each state created in function CyclicAddWord is processed, all states of the unmodified part of the automaton are mutually inequivalent, and during the processing in procedure Minim, states are replaced with equivalent ones if they exist, each equivalence class in M'''' has exactly one representative. Therefore, M'''' is minimal.

Operations outside the `while` loop in function CyclicWatsonConstruction take constant time, except for procedure Minim and function CyclicBuildStack, as well as procedure DeleteBranchUp. Procedure DeleteBranchUp is executed up to $n|w_{max}|$ times, where n is the number of words on the input, and w_{max} is the longest word on the input. We will count invocations of function CyclicBuildStack and procedure Minim outside the loop together with their invocations inside the loop.

The loop runs n times. Inside the loop, function CyclicAddWord is invoked once per each run. Inside the function, there are three loops that run as many times in total as there are characters in the word being added. All operations inside and outside the loops take constant time. Thus, function CyclicAddWord runs in $\mathcal{O}(n|w_{max}|)$ in total. Function CyclicBuildStack is invoked as many times as there are states created in function CyclicAddWord --- $n|w_{max}|$

in all. This is the same number as the number of runs of the contents of the `while` loop in procedure Minim across all its invocations. Under the assumption of constant-time register operations, all operations inside that loop run in constant time. This gives us $\mathcal{O}(n|w_{max}|)$ for the whole algorithm.

Note that the problem with greediness of function CyclicBuildStack can be solved also by marking words processed by CyclicAddWord. In that case, cloning the initial state is necessary only when it has some incoming transitions, and invocation of procedure DeleteBranchUp is not necessary.

This algorithm was developed by the author. It was published in [16].

5.1.3 Extension to Minimal Bottom-Up Tree Automata

The algorithm described in Section 4.4 has small memory footprint. However, states can be reprocessed many times, and the cost of that processing is much higher in DTAs than in FSAs because of much more complex cloning. An extension of Watson's algorithm can offer higher speed at the cost of memory efficiency. As we have seen in examples of various versions of the algorithm, the final invocation of local minimization has still a lot to do.

Algorithm 5.11 Extension of Watson's construction algorithm to construction of deterministic bottom-up tree automata.

```
 1: function WatsonDTAConstruction
 2:     Create an empty automaton A, register R, and a stack X
 3:     while There are trees on the input do
 4:         t ← next tree on the input
 5:         LocTreeMinim(A,R,BuildTreeStack(A,X,AddWatsonTree(A,t)))
 6:     end while
 7:     for σ ∈ Σ do
 8:         if δ0(σ) ≠ ⊥ ∧ δ0(σ) ∉ F then
 9:             LocTreeMinim(A,R,BuildTreeStack(A, X, δ0(σ)))
10:         end if
11:     end for
12:     return A
13: end function
```

The main algorithm is implemented as function WatsonDTAConstruction, and is presented as Algorithm 5.11. In comparison to the basic algorithm in Section 5.1, (Algorithm 5.1, page 131), there are slightly different function and procedure names, as they deal with tree automata. The final minimization cannot start in the initial state, as there is no initial state in DTAs. Therefore we start in states whose language is a single symbol in the alphabet, unless they are final, i.e. there are trees in the language of the automaton (trees read from the input) being just single symbols.

Function AddWatsonTree (Algorithm 5.12) calls function WatsonSplit (a simplified version of function Split, given as Algorithm 4.12, page 100). As Watson's algorithm is much simpler than the incremental algorithm for unsorted data, the only thing that remains to be done in function AddWatsonTree is to make the returned state final.

Algorithm 5.12 Function AddWatsonTree adds a tree to the language of a DTA. The main work is done by a call to function WatsonSplit. The remaining work consists only of making the returned state final.

```
1: function AddWatsonTree(A,R,X,t = σ(t_1, ..., t_m))
2:     q ← WatsonSplit(A, t)
3:     F ← F ∪ {q}
4:     return q
5: end function
```

Algorithm 5.13 Function WatsonSplit adds a tree t to the automaton A without making state $\delta_A(t)$ final (this is done in function AddWatsonTree.

```
1: function WatsonSplit(A,t = σ(t_1, ..., t_m))
2:     for k ∈ 1, ..., m do
3:         r_k ← WatsonSplit(A, t_k)
4:     end for
5:     q ← δ_m(σ, r_1, ..., r_m)
6:     if q = ⊥ then
7:         n ← new state
8:         Δ ← Δ ∪ {(σ, r_1, ..., r_m, n)}
9:         q ← n
10:        B_q ← (σ, r_1, ..., r_m, n)
11:    end if
12:    return q
13: end function
```

Function Split has been simplified to WatsonSplit as there are no confluence states to be handled there. Therefore, there is one case less. The function calls itself on the subtrees of the parameter tree t, and if $\delta_A(t) \neq \bot$, $\delta_A(t)$ is returned. If not, the state and a transition leading to it have to be created, and the back transition needs to be stored.

Algorithm 5.14 Function BuildTreeStack.

```
1: function BuildTreeStack(A, X, q)
2:     push q onto X
3:     Mark(q)
4:     for d = (σ, q_1, ..., q_m, n) ∈ Δ such that ∃_{1≤i≤m} q_i = q do
5:         if n ∉ F ∧ ¬Marked(n) then
6:             X = BuildTreeStack(A, X, n)
7:         end if
8:     end for
9:     return X
10: end function
```

Function BuildTreeStack differs from function BuildStack (Algorithm 5.3, page 132) in that it operates on states of a DTA, and ---as a consequence--- that it needs to mark states in order not to reprocess them. In a DFA, a state that did not undergo the process of minimization

had one predecessor as one symbol in a word may be immediately preceded by at most a single symbol. In a DTA, a tree may have several subtrees that are shared among different trees. Therefore, even in an automaton built like a trie with no minimization whatsoever, states may have one incoming transition with more than one source state, i.e. they may have more than one preceding state.

Algorithm 5.15 Procedure LocTreeMinim performs local minimization on states of a DTA A stored on a stack X using a register R as a set of unique states.

```
 1: procedure LocTreeMinim(A, R, X)
 2:     while X ≠ ∅ do
 3:         pop n from X
 4:         q ← FindEquiv(A, R, n)
 5:         if q ≠ n then
 6:             for τ = (σ, r_1, ..., r_m, p) ∈ Δ such that ∃_{1≤i≤m} q_i = q do
 7:                 Δ ← Δ \ {τ}
 8:             end for
 9:             τ = (σ, r_1, ..., q_m, n) ← B_n
10:             Δ ← Δ \ {τ} ∪ {(σ, r_1, ..., r_m, q)}
11:             delete state n
12:         else
13:             R ← R ∪ {n}
14:         end if
15:     end while
16: end procedure
```

Procedure LocTreeMinim is also simple. States are popped from the stack until it becomes empty. For each state an equivalent state is searched for. If found, all transitions that have the popped state as a source state are removed, its incoming transition is redirected towards the equivalent state, and the popped state is deleted. If not, the state is added to the register.

Let us see how the algorithm works. We start with an empty automaton, and we add *a(b(a,b),b(a,b))*. Function WatsonDTAConstruction creates the empty automaton and an empty stack X. Then it reads the tree, and calls function AddWatsonTree(A, t = *a(b(a,b),b(a,b))*), which immediately calls function WatsonSplit(A, t = *a(b(a,b),b(a,b))*). The latter calls itself as WatsonSplit(A, t = *b(a,b)*), which calls WatsonSplit(A, *t=a*). As $\delta_0(a) = \perp$, a new state 1 is created, and a new transition $(a, 1)$ is added to Δ. That transition is also assigned to B_1. The function returns state 1, which is assigned to variable r_1 one level higher. Function WatsonSplit(A, *t=b*) is called. As $\delta_0(b) = \perp$, a new state 1 is created, and a new transition $(b, 2)$ is added to Δ. That transition is also assigned to B_2. State 2 is returned, and it is assigned to variable r_2 one level of recursion up. As $\delta_2(b, 1, 2) = \perp$, a new state 3 is created, and a new transition $(b, 1, 2, 3)$ is added to Δ. That transition is also assigned to B_3. The function returns state 3, which is assigned to variable r_1 in the top-level call to WatsonSplit. Then WatsonSplit(A, *t=*b(a,b)) is called again. It calls WatsonSplit(A, *t=a*). This time, $\delta_0(a) = 1$, so state 1 is returned and assigned to variable r_1 one level up. Function WatsonSplit(A, *t=b*) is called, and as $\delta_0(b) = 2$, it returns state 2, which is assigned to variable r_2 one level up. At that level, as $\delta_2(b, 1, 2) = 3$, state 3 is returned, which is assigned to variable r_2 at the top-level invocation of function WatsonSplit. As $\delta_2(a, 3, 3) = \perp$, a new state 4 is created,

and a new transition $(a, 3, 3, 4)$ is added to Δ. That transition is also assigned to B_4. The function returns state 4, which is made final in AddWatsonTree.

Function BuildTreeStack(A, $X = ()$, 4) is called. It puts state 4 onto the stack. As there are no transitions with state 4 as a source state, the function returns the stack. Procedure LocTreeMinim(A, $R = \emptyset$, $X = (4)$) is called. State 4 is popped from the stack. As the register R is empty, the state is added to the register, and the call is completed. The resulting automaton is shown in Figure 5.22.

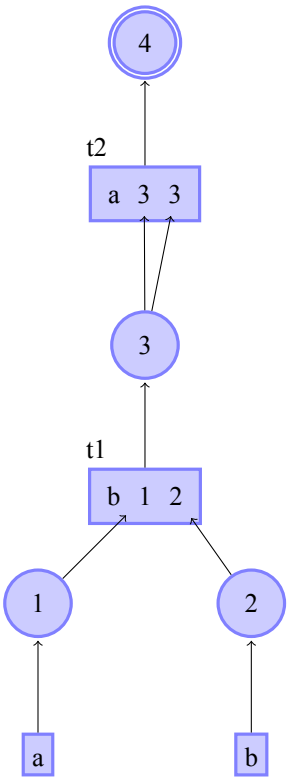


Figure 5.22: Extension of Watson construction to minimal bottom-up DTAs: a tree *a(b(a,b),b(a,b))* has just been added to an empty DTA.

Function WatsonDTAConstruction reads another tree from the input, and calls function AddWatsonTree(A, *t=b(b(a,b),a(b,a),b)*), which calls function WatsonSplit(A, *t=b(b(a,b), a(b,a),b)*). It calls itself as WatsonSplit(A, *t=b(a,b)*), which calls WatsonSplit(A, *t=a*). That call returns state 1, which is assigned to variable r_1 one level up. A following call to WatsonSplit(A, *t=b*) returns state 2, which is assigned to variable r_2 one level up. At that level, as $\delta_2(b, 1, 2) = 3$, state 3 is returned and assigned to variable r_1 at the top-level invocation of function WatsonSplit. Then, WatsonSplit(A, *t=a(b,a)*) is called, which calls WatsonSplit(A, *t=b*). The latter call returns state 2, which is assigned to variable r_1 one level up. A following call to WatsonSplit(A, *t=a*) returns state 1, which is assigned to variable r_2 one level up. At that level, as $\delta_2(a(2, 1)) = \perp$, a new state 5 is created, and a new transition $(a, 2, 1, 5)$ is

added to Δ. That transition is also assigned to B_5. The function returns state 5, which is assigned to variable r_2 in the top-level invocation of function WatsonSplit. The function is called again as WatsonSplit(A, *t=b*). It returns state 2 to the top-level, where it is assigned to variable r_3. As $\delta_3(b, 3, 5, 2) = \perp$, a new state 6 is created, and a new transition $(b, 3, 5, 2, 6)$ is added to Δ. That transition is also assigned to B_6. The function returns state 6 to function AddWatsonTree, which makes it final, and returns it to function BuildTreeStack.

Function BuildTreeStack puts state 6 onto the stack. As there are no transitions leaving state 6, the stack is returned to procedure LocTreeMinim. State 6 is popped from the stack, and function FindEquiv returns state 4. Transition $\tau = (b, 3, 5, 2, 6)$ is replaced in Δ with transition $(b, 3, 5, 2, 4)$. State 6 is deleted, as depicted in Figure 5.23.

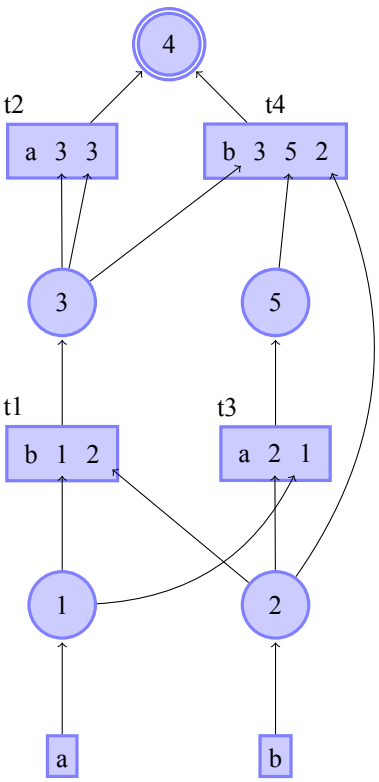


Figure 5.23: Extension of Watson construction to minimal bottom-up DTAs: a tree *b(b(a,b),a(b,a),b)* has just been added to a DTA containing *a(b(a,b),b(a,b))*.

Function WatsonDTAConstruction reads tree *a(a(a,a),a(a,a))* from the input, and it calls function AddWatsonTree(A, *t=a(a(a,a),a(a,a))*), which calls function WatsonSplit(A, *t=a(a(a, a),a(a,a))*). The latter calls itself as WatsonSplit(A, *t=a(a,a)*), which calls WatsonSplit(A, *t=a*). The last call returns state 1, which is assigned to variable r_1 one level up in the call hierarchy. A following identical call also returns state 1, which is assigned to variable r_2 one level up. At that level, as $\delta_2(1, 1) = \perp$, a new state 6 is created, and a new transition $(a, 1, 1, 6)$ is added to Δ and assigned to B_6. State 6 is returned to the top-level invocation of

WatsonSplit to be assigned to variable r_1. Another call to WatsonSplit(A, *t=a(a,a)*) follows. It calls WatsonSplit(A, *t=a*) twice, which results is assigning state 1 to variables r_1 and r_2, and it returns state 6 to the top-level invocation of function WatsonSplit. At that level, as $\delta_2(a, 6, 6) = \bot$, a new state 7 is created, and a new transition $(a, 6, 6, 7)$ is added to Δ, and assigned to B_7. State 7 is made final, and it is returned to function BuildTreeStack.

Function BuildTreeStack puts state 7 onto the stack X. As state 7 has no outgoing transitions, the function returns the stack to procedure LocTreeMinim, which pops state 7 from the top of the stack. Function FindEquiv returns state 4. Transition $(a, 6, 6, 7)$ is replaced with transition $(a, 6, 6, 4)$, and state 7 is deleted. The situation is shown in Figure 5.24.

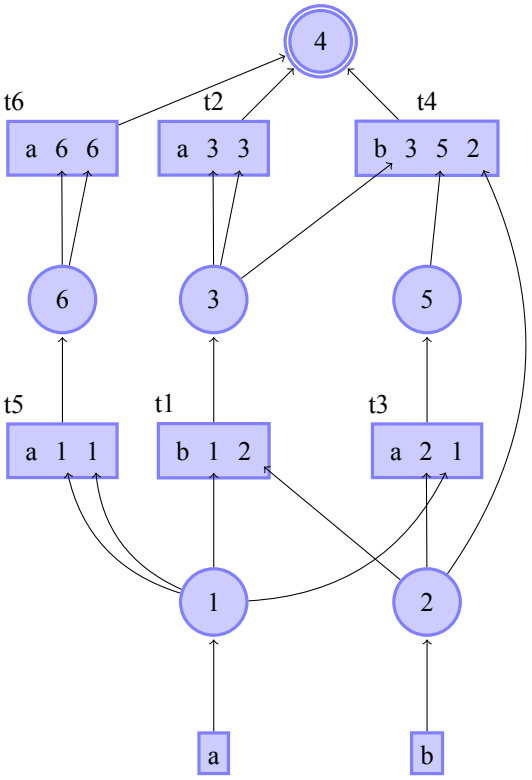


Figure 5.24: Extension of Watson construction to minimal bottom-up DTAs: a tree *a(a(a,a),a(a,a))* has just been added to a DTA containing *a(b(a,b),b(a,b))* and *b(b(a,b),a(b,a),b)*.

Function WatsonDTAConstruction reads another tree from the input, and it calls function AddWatsonTree(A, *t=a(a(a,a),b(a,b))*), which calls function WatsonSplit(A, *t=a(a(a,a),b(a,b))*). The last function calls itself as WatsonSplit(A, *t=a(a,a)*), which calls WatsonSplit(A, *t=a*). The function returns state 1, which is assigned to variable r_1 one level up in the call hierarchy. An identical call returns state 1 again, which is assigned to variable r_2 one level up. At that level, state 6 is returned and assigned to variable r_1 at the top level. A call to WatsonSplit(A, *t=b(a,b)*) follows, which calls WatsonSplit(A, *t=a*). State 1 is returned and

assigned to variable r_1 one level up. A call to WatsonSplit(A, *t=b*) returns state 2, which is assigned to variable r_2 one level up. At that level, state 3 is returned, which is assigned to variable r_2 at the top level. There, as $\delta_2(a, 6, 3) = \bot$, a new state 7 is created, and a new transition $(a, 6, 3, 7$ is added to Δ and assigned to B_7. State 7 is returned to function AddWatsonTree, which makes it final, and returns it to function BuildTreeStack.

In function BuildTreeStack, state 7 is pushed onto the stack. As state 7 had no outgoing transitions, the stack is returned to procedure LocTreeMinim. Function FindEquiv returns state 4. Transition $(a, 6, 3, 7)$ is redirected to state 4, and state 7 is deleted. The stack is empty. The resulting automaton is shown in Figure 5.25.

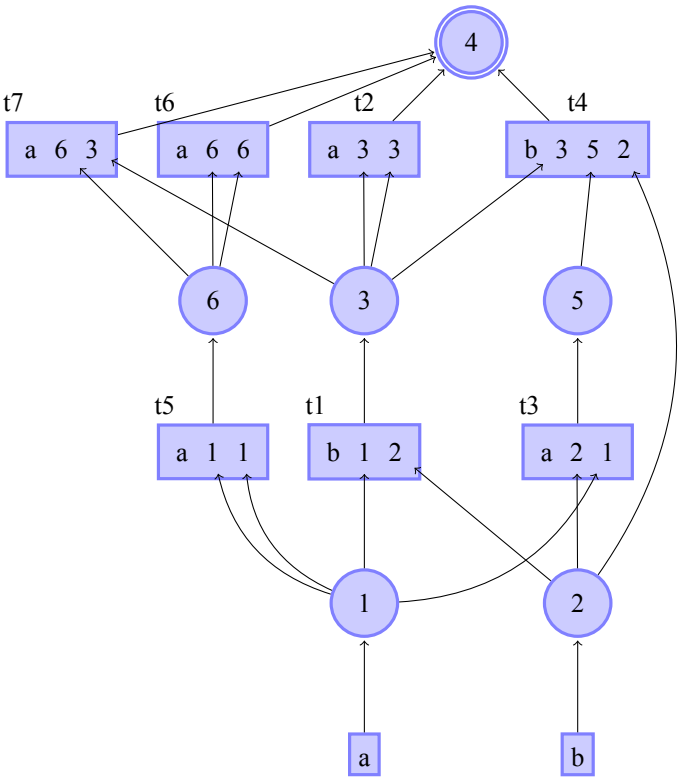


Figure 5.25: Extension of Watson construction to minimal bottom-up DTAs: a tree *a(a(a,a),b(a,b))* has just been added to a DTA containing *a(b(a,b),b(a,b))* *b(b(a,b),a(b,a),b)*, and *a(a(a,a),a(a,a))*.

Function WatsonDTAConstruction reads another tree from the input, and it calls function AddWatsonTree(A, *t=a(b(a,b),a(a,a))*). The latter calls function WatsonSplit(A, *t=a(b(a,b), a(a,a))*), which calls itself as WatsonSplit(A, *t=b(a,b)*), which calls WatsonSplit(A, *t=a*). The last function call returns state 1, which is assigned to variable r_1 one level up in the call hierarchy. Then WatsonSplit(A, *t=b*) returns state 2 assigned to variable r_2. State 3 is returned to the top-level WatsonSplit invocation, where it is assigned to variable r_1 Function WatsonSplit(A, *t=a(a,a)*) is called, which calls WatsonSplit(A, *t=a*). The last function returns state 1,

which is assigned to variable r_1 one level up. An identical call returns state 1 again, this time assigned to variable r_2 one level up. State 6 is returned to the top-level invocation of function WatsonSplit. As $\delta_2(a, 3, 6) = \perp$, a new state 7 is created, and a new transition $(a, 3, 6, 7)$ is added to Δ and assigned to B_7. State 7 is returned to function AddWatsonTree that makes the state final and returns it to function BuildTreeStack.

Function BuildTreeStack pushes state 7 onto the stack, and returns the stack, as state 7 has no outgoing transitions. Function LocTreeMinim pops state 7 from the stack. Function FindEquiv returns state 4, so transition $(a, 3, 6, 7)$ is redirected towards state 4, and state 7 is deleted, as depicted in Figure 5.26.

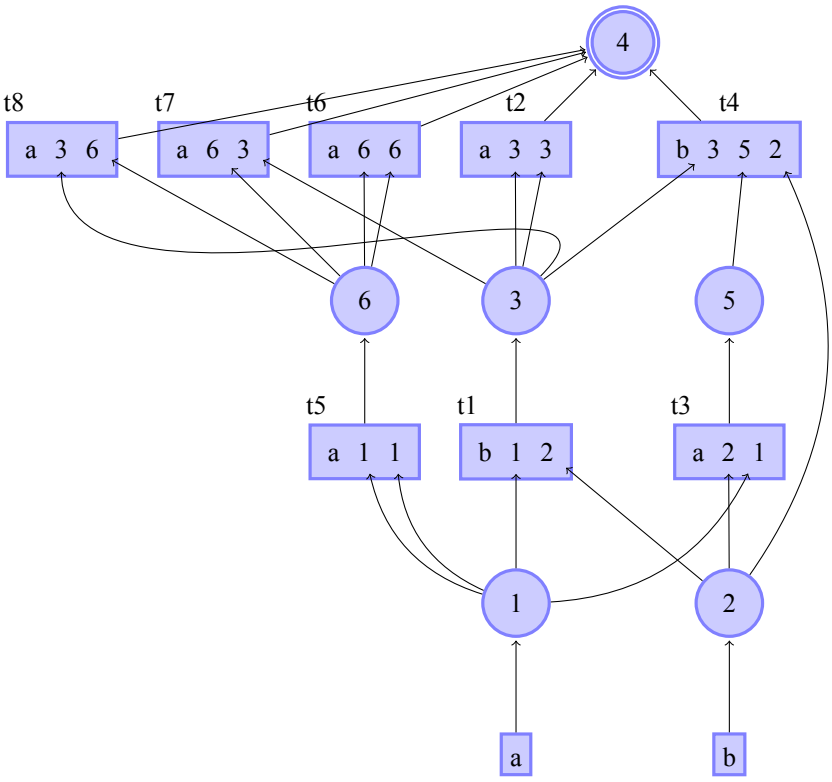


Figure 5.26: Extension of Watson construction to minimal bottom-up DTAs: a tree *a(b(a,b),a(a,a))* has just been added to a DTA containing *a(b(a,b),b(a,b))* *b(b(a,b),a(b,a),b)*, *a(a(a,a),a(a,a))*, and *a(a(a,a),b(a,b))*.

Function WatsonDTAConstruction reads another tree from the input, and it calls function AddWatsonTree(A, *t=b(a(a,a),a(b,a),b)*). The latter calls function WatsonSplit(A, *t=b(a(a,a), a(b,a),b)*), which calls itself as WatsonSplit(A, *t=a(a,a)*), which calls WatsonSplit(A, *t=a*). The last function returns state 1, that is assigned to variable r_1. An identical call that follows returns state 1, which is assigned to variable r_2. State 6 is returned to the top-level invocation of WatsonSplit, where it is assigned to variable r_1. Then WatsonSplit(A, *t=a(b,a)*) is called, which calls WatsonSplit(A, *t=b*). The last call returns state 2, which is assigned to variable

r_1 one level up. A call to WatsonSplit(A, *t=a*) returns state 1 again, which is assigned to variable r_2 one level up. State 5 is returned to the top-level invocation of WatsonSplit. A call to function WatsonSplit(A, *t=b*) returns state 2 to the top-level invocation of WatsonSplit. As $\delta_3(b, 6, 5, 2) = \bot$, a new state 7 is created, and a new transition $(b, 6, 5, 2, 7)$ is added to Δ and assigned to B_7. State 7 is returned to function AddWatsonTree, where it is made final, and where it is returned to function BuildTreeStack.

Function BuildTreeStack pushes state 7 onto the stack, and as there are no outgoing transitions in state 7, the stack is returned to procedure LocTreeMinim, where state 7 is popped from the stack. Function FindEquiv returns state 4, so the transition $(b, 6, 5, 2, 7)$ is redirected towards state 4, and state 7 is deleted, as shown in Figure 5.27.

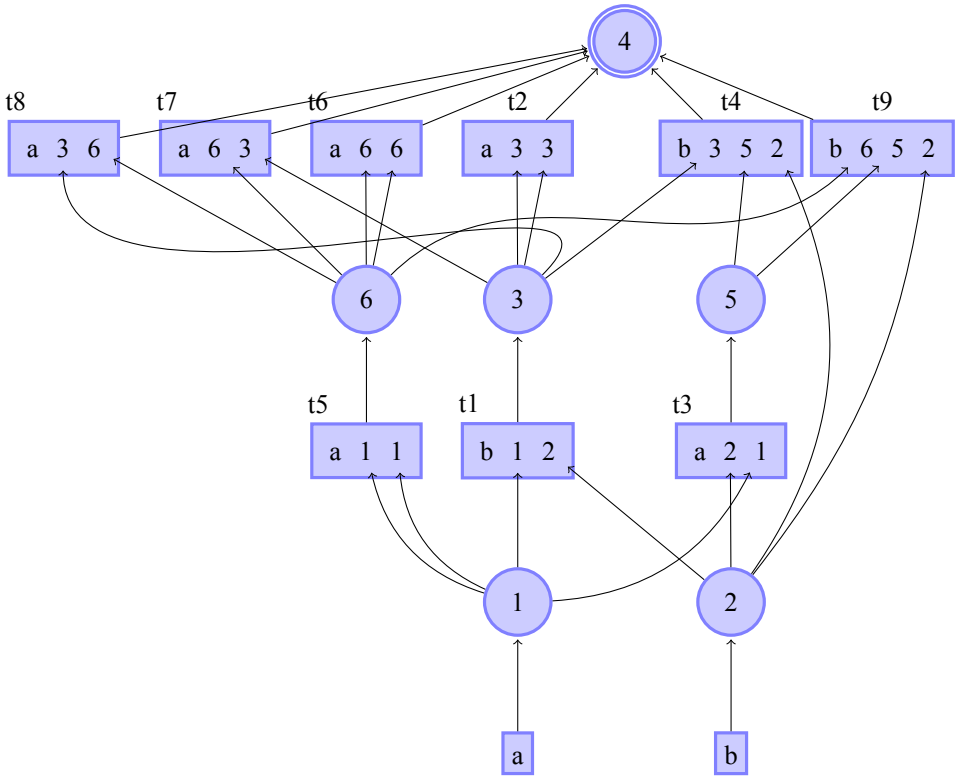


Figure 5.27: Extension of Watson construction to minimal bottom-up DTAs: a tree *b(a(a,a),a(b,a),b)* has just been added to a DTA containing *a(b(a,b),b(a,b))* *b(b(a,b),a(b,a),b)*, *a(a(a,a),a(a,a))*, *a(a(a,a),b(a,b))*, and *a(b(a,b),a(a,a))*

Function WatsonDTAConstruction reads another tree from the input, and it calls function AddWatsonTree(A, *t=a(a,a)*). The latter calls function WatsonSplit(A, *t=a(a,a)*). That one calls itself as WatsonSplit(A, *t=a*), which returns state 1 that is assigned to variable r_1 in the top-level invocation of function WatsonSplit. State 1 is also returned from an identical call, and then assigned to variable r_2. Function WatsonSplit returns state 6 to function AddWatsonTree, where it is made final, and it is returned to function BuildTreeStack.

Function BuildTreeStack pushes state 6 onto the stack. State 6 has a few outgoing transitions, but all of them lead to the same state 4, which is final, so it cannot be a parameter of another BuildTreeStack call. The function returns the stack to procedure LocTreeMinim. State 6 is popped from the stack. Function FindEquiv returns the same state (state 3 is not final), so this completes the procedure.

There are no more trees on the input, so the `for` loop is executed. It sets σ to a and calls BuildTreeStack(A, X, 1). State 1 is pushed onto the stack, and state 1 is marked. Transition $(a, 1, 1, 6)$ leads to state 6, which is final. Transition $(b, 1, 2, 3)$ leads to state 3, which is not final, so BuildTreeStack(A, $X = (1), 3$) is called. State 3 is pushed onto the stack and marked. All outgoing transitions of state 3 go to final state 4, so the call finishes. The last transition from state 1 leads to non-final state 5, so BuildTreeStack(A, $X = (1, 3), 5$) is called. State 5 is pushed onto the stack, and it is marked. All transitions from state 5 lead to final state 4. All invocations of BuildTreeStack terminate. The stack is returned to procedure LocTreeMinim. State 5 is popped from the stack. Function FindEquiv returns the same state, so the state is added to the register, and no further handling of that state is needed. State 3 is popped from the stack, and it is also found to be unique, so it is added to the register. The same happens to state 1.

The `for` loop sets σ to b and calls BuildTreeStack(A, $X = ()$, 2). State 2 is pushed onto the stack and marked. All outgoing transitions from state 2 lead to states that are either final or already marked. The call terminates with the stack returned to procedure LocTreeMinim. State 2 is popped from the stack. It is unique, so it is added to the register, and procedure LocTreeMinim terminates. So does WatsonDTAConstruction. The resulting automaton is presented in Figure 5.28.

Let $A = (Q, \Sigma, \Delta, F)$ be a minimal DTA just before a new tree t is added to the language of the automaton, i.e. before function AddWatsonTree is called. Let $A' = (Q', \Sigma, \Delta', F')$ be the automaton right after function AddWatsonTree has returned a state to function BuildTreeStack. Let $A'' = (Q'', \Sigma, \Delta'', F'')$ be the DTA after function LocTreeMinim has finished working on t. Let $A''' = (Q''', \Sigma, \Delta''', F''')$ be the DTA returned by function WatsonDTAConstruction. We assume that there are no redundant trees on the input, i.e. there are no duplicates. We show that:

1. $\mathcal{L}(A') = \mathcal{L}(A) \cup \{t\}$
2. $\mathcal{L}(A'') = \mathcal{L}(A')$
3. $|Q''| \leq |Q'|$
4. During the whole construction process, each state is put onto the stack exactly once.
5. States are put onto the stack, and then undergo minimization in such order, that states that are reachable from any given state are processed before that state.
6. $\mathcal{L}(A''')$ is the sum of all input trees.
7. A''' is minimal.

To prove the first point, let us first notice that $t \in \mathcal{L}(A')$. Function WatsonSplit makes sure that $L_A(t) \neq \bot$. It first calls itself recursively to ensure that subtrees $t_1, \ldots, t_m$ (if any) get their corresponding states $r_1, \ldots, r_m$. Then it verifies whether there is a state q such that

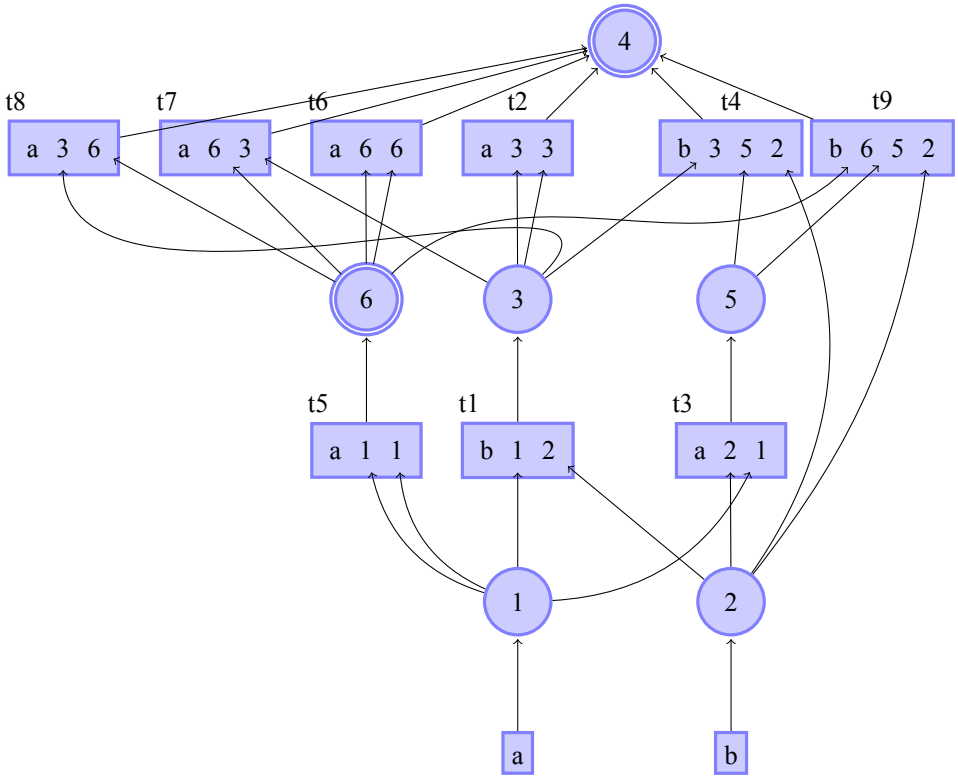


Figure 5.28: Extension of Watson construction to minimal bottom-up DTAs: a tree *b(a(a,a),a(b,a),b)* has just been added to a DTA containing *a(b(a,b),b(a,b))* *b(b(a,b),a(b,a),b)*, *a(a(a,a),a(a,a))*, *a(a(a,a),b(a,b))*, and *a(b(a,b),a(a,a))*

$L_A(q) = \{t\}$ (we could write $t \in L_A(q)$, but states visited in function WatsonSplit are never subject of minimization before that call). If there is such state, it is simply returned by the function as nothing else needs to be done. If there is none, than it is created along with a transition from the source states $r_1, \ldots, r_m$. When $m = 0$, then no recursion takes place, and an initial transition from a leaf is created. Once the entire structure is built, function AddWatsonTree makes $L_A(t)$ final, thus ensuring directly from definition (2.19) that $t \in \mathcal{L}(A)$. Secondly, it can easily be seen that $\mathcal{L}(A) \subseteq \mathcal{L}(A')$. Indeed, no states or transitions are deleted or redirected in functions WatsonSplit and AddWatsonTree. Thirdly, notice that no additional trees are recognized in $\mathcal{L}(A')$. As function WatsonSplit creates only states and transitions needed directly to recognize t, additional trees can be added only by creating transitions from confluence states. But confluence states are created in procedure LocTreeMinim that acts on states put onto stack by function BuildTreeStack. The latter is called initially with final states that correspond to roots of trees that just have been added by function AddWatsonTree, and then it can add additional states that are reachable from that state. Because the input trees are sorted on height, and the DTA is acyclic, function WatsonSplit can never reach such states.

To prove the second point, let us see that function BuildTreeStack only selects states for

minimization, and procedure LocTreeMinim replaces states with equivalent ones, redirecting transitions accordingly, and deleting redundant states along with their outgoing transitions. Function FindEquiv used for finding an equivalent state uses the assumption that states are processed in an appropriate order. However, this only weakens the condition for replacement, and replacing a state with an equivalent one does not change the language of the automaton.

The third point is obvious, as states in procedure LocTreeMinim can only be deleted.

When a unique tree is added to a DTA that recognizes trees already processed, there are two cases. Either a tree is a subtree of at least one other tree that was added earlier (as the bigger trees have greater height and must have come earlier in the input), or not. In the second case, function WatsonSplit must create a new state for the root of such a tree. The state is made final in function AddWatsonTree and handed over to function BuildTreeStack. Since the tree cannot be a subtree of a tree that comes later that bigger tree would have a greater height, and the state is final, it cannot be revisited in function BuildTreeStack, and consequently in procedure LocTreeMinim. In the first case, the state handed over to function BuildTreeStack was not created during addition of the current tree. It was created earlier during addition of a bigger tree. There may be some other bigger trees that also share the same subtree. States that correspond to such subtrees of those trees that are not in the language of the DTA. They fall into two categories: those that have not yet been processed by function BuildTreeStack and procedure LocTreeMinim, and those that are already marked as processed. Note that none of them can be revisited in function WatsonSplit. Marking ensures that states are processed at most once. Depth-first visiting order in function BuildTreeStack ensures that each state is indeed processed, as all states reachable from marked states are already marked. The final loop in function WatsonDTAConstruction calls function BuildTreeStack with all states $L_A(\sigma)$ such that $\sigma \in \Sigma$ is a leaf of some tree in $\mathcal{L}(A''')$ and $\sigma \notin \mathcal{L}(A''')$. Those states and states reachable from them up to but excluding final states are the only states in the automaton not yet processed. The reasoning why they are processed once is exactly the same as above. this concludes point 4.

Point 5 is a direct consequence of depth-first order of putting states onto the stack in function BuildTreeStack. Point 6 can be concluded from point 2, and from the observation that minimization does not change the language of the automaton. Point 7 results from points 4, 5, and 6.

Reading a tree takes time proportional to its size. Function AddWatsonTree is called once for every input tree. Making a state final takes a constant amount of time (the information is stored in states). Function WatsonSplit is called in total a number of times proportional to the total size of the input trees (once for every transition). It either traverses or creates states and transitions of the size of the input data. Function BuildTreeStack is called as many times in total as function WatsonSplit. Marking a state, checking those marks or finality, and pushing a state onto a stack is done in constant time. Procedure LocTreeMinim is called as many times as there are trees on the input plus one, but the `while` loop in that procedure is executed a number of times limited by the size of the input data. It is the size of the input data except that common subtrees are counted once. Since each state is processed once in procedure LocTreeMinim, each transition can be redirected (in constant time) or deleted (in constant time) only once. We assume, as we do everywhere in this book, that register operations take constant amount of time. Then function FindEquiv executes in constant time. This gives us $\mathcal{O}(\sum_i |t_i|)$ as the overall complexity of the algorithm, where $\sum_i |t_i|$ is total size of the input trees. Under a different assumption about register operations, the complexity of such

operations would become a factor in the overall complexity.

Notice that the algorithm is much faster than its incremental counterpart, because it does not have to clone states, and in general, it does not have to reprocess states. However, this comes at a cost of much greater intermediate size of the automaton.

This algorithm is yet to be benchmarked and then to be submitted to a conference or a journal. It was developed by the author of the book.

5.1.4 Extension to Pseudo-Minimal Bottom-Up Tree Automata

Just like the incremental construction algorithm for sorted data has an extension for building pseudo-minimal DTAs, so Watson algorithm does. Transforming an algorithm for constructing minimal DTAs into an algorithm for constructing pseudo-minimal DTAs is very similar. The extension of Watson algorithm is simpler. The algorithm is also faster, but it needs much more memory for an intermediate automaton. As this algorithm is an extension of Watson's algorithm, trees have to be sorted on their height, so that bigger trees come first.

Algorithm 5.16 Extension of Watson's construction algorithm to construction of pseudo-minimal deterministic bottom-up tree automata.

```
 1: function WatsonPseudoDTAConstruction
 2:     Create an empty automaton A, register R, and a stack X
 3:     while there are trees on the input do
 4:         t ← next tree on the input
 5:         LocTreePseudoMinim(A,R,BuildTreeStack(A,X,PseudoAddWatsonTree(A,t)))
 6:     end while
 7:     for σ ∈ Σ do
 8:         if δ0(σ) ≠ ⊥ ∧ δ0(σ) ∉ F then
 9:             LocTreePseudoMinim(A,R,BuildTreeStack(A, X, δ0(σ)))
10:         end if
11:     end for
12:     return A
13: end function
```

The main algorithm is implemented as function WatsonPseudoDTAConstruction, and is presented as Algorithm 5.16. In comparison to the algorithm in the previous section, a bit different function and procedure names are used. They reflect changes that are made inside them.

Function PseudoAddWatsonTree (Algorithm 5.12) calls function WatsonPseudoSplit instead of function WatsonSplit. It also empties a set M of states that updated their counters. Otherwise, it is identical to function AddWatsonTree.

Compared to function WatsonSplit, function WatsonPseudoSplit has additional handling of counters T_q of the number of trees that need to visit a state q while a tree is being recognized. That handling also involves a set M of states that have already participated in counting that prevents recounting the same tree when it has identical subtrees. Note that we previously used markers for the same task. They were cleared in the procedure that performed local minimization. However, that procedure processed the whole tree right after it has been added.

Algorithm 5.17 Function PseudoAddWatsonTree adds a tree to the language of a DTA. The main work is done by a call to function WatsonPseudoSplit. The remaining work consists only in making the returned state final.

```
1: function PseudoAddWatsonTree(A,R,X,t = σ(t_1, ..., t_m))
2:     M ← ∅
3:     q ← WatsonSplit(A, M, t)
4:     F ← F ∪ {q}
5:     return q
6: end function
```

Algorithm 5.18 Function WatsonPseudoSplit adds a tree t to the automaton A without making state $\delta_A(t)$ final (this is done in function PseudoAddWatsonTree.

```
1: function WatsonPseudoSplit(A,M,t = σ(t_1, ..., t_m))
2:     for k ∈ 1, ..., m do
3:         r_k ← WatsonPseudoSplit(A, t_k)
4:     end for
5:     q ← δ_m(σ, r_1, ..., r_m)
6:     if q = ⊥ then
7:         n ← new state
8:         Δ ← Δ ∪ {(σ, r_1, ..., r_m, n)}
9:         q ← n
10:        B_q ← (σ, r_1, ..., r_m, n)
11:        M ← M ∪ {n}
12:        T_n ← 1
13:    else if q ∉ M then
14:        M ← M ∪ {q}
15:        T_q ← T_q + 1
16:    end if
17:    return q
18: end function
```

Here, it is not the case, so we use a set that can be cleared before each tree is added in function PseudoAddWatsonTree.

Procedure LocTreePseudoMinim differs from procedure LocTreeMinim only by a test $T_n \leq 1$ that transforms equivalence into pseudo-equivalence.

We leave the proof of correctness as an easy exercise for the reader. The difference between this algorithm and the algorithm described in the previous section is that equivalence has been replaced with pseudo-equivalence (which is also an equivalence relation), and that we maintain counters (handled in a trivial way) that help in implementing pseudo-equivalence.

This algorithm has been developed by the author of the book, and it still needs to be benchmarked and to be submitted to a conference or a journal.

Algorithm 5.19 Procedure LocTreePseudoMinim performs local pseudo-minimization on states of a DTA A stored on a stack X using a register R as a set of unique states.

```
 1: procedure LocTreePseudoMinim(A, R, X)
 2:     while X ≠ ∅ do
 3:         pop n from X
 4:         if T_n ≤ 1 then
 5:             q ← FindEquiv(A, R, n)
 6:         else
 7:             q ← n
 8:         end if
 9:         if q ≠ n then
10:             for τ = (σ, r_1, ..., r_m, p) ∈ Δ such that ∃_{1≤i≤m} q_i = q do
11:                 Δ ← Δ \ {τ}
12:             end for
13:             τ = (σ, r_1, ..., q_m, n) ← B_n
14:             Δ ← Δ \ {τ} ∪ {(σ, r_1, ..., r_m, q)}
15:             delete state n
16:         else
17:             R ← R ∪ {n}
18:         end if
19:     end while
20: end procedure
```

5.2 Revuz's Algorithm

Dominique Revuz wanted to develop an algorithm for construction of minimal deterministic acyclic automata. He wrote: "Il n'existe pas d'algorithme rapide pour construire directement l'automate déterministe minimal d'un language fini donné, à partir du language sous forme de liste. Il est nécessaire de procéder en deux étapes."[2] We now know that it is indeed possible to construct such an automaton directly; this is what incremental algorithms do. However, the pseudo-minimal automaton that is the result of the first phase of his algorithm can be useful for implementing arbitrary hashing. Therefore, we present his algorithm as an algorithm for constructing a pseudo-minimal automaton. The original has also a final minimization phase that we skip here.

The algorithm is presented here without any proof, as it is given here only to complete the inventory of construction algorithms. The main function creates an empty automaton, and then adds subsequent words from the input one by one using procedure RevuzAddWord as shown in Algorithm 5.20.

Procedure RevuzAddWord looks quite complicated, and it is indeed more complex than the incremental algorithms, and more complex than Watson's algorithm. The first part looks familiar; those are the same two `while` loops that we see in the incremental construction algorithm for sorted data. The only difference is the invocation of procedure Mark that marks states that have been visited there. If the end of the input word has been reached, the path

[2] There is no fast algorithm for direct construction of a minimal, deterministic automaton of a given finite language from a language in form of a list.

Algorithm 5.20 Function RevuzConstruction takes a list of words sorted lexicographically on reversals of words.

```
 1: function RevuzConstruction
 2:     Create an empty automaton M = ({q_0, q_f, }, Σ, ∅, q_0, {q_f})
 3:     P ← ()
 4:     w' ← ε
 5:     while there are words on the input do
 6:         w ← next word on the input
 7:         RevuzAddWord(M, P, w, w')
 8:         w' ← w
 9:     end while
10:     return M
11: end function
```

P is cleared, the current state is made final, and that is it. Otherwise, the suffix of the word w is examined. In lines 19--22, we look for the longest part of it that is also a suffix of the previous word. This search only takes place if the previous word was not a prefix of any other word in the automaton (checked in the line immediately preceding the loop). In the loop, we also check whether the suffix overlaps with the prefix of w already found in the automaton (states in the prefix are marked), and whether states in the suffix have more than one outgoing transition. If the second condition is met, the suffix is shortened to exclude such state. The next `while` loop in lines 24--28 builds a chain of states and transitions that link the state representing the end of the prefix of w with the state starting the suffix. The remaining part of the procedure does housekeeping. It clears the marks in the path recognizing w, and it stores that path in P for use with the next word.

Since the algorithm is provided here only for reference, and it has no extensions, we give no examples of its use, and no proof of correctness. Our description of the algorithm differs much from the original, as we give details about implementation that are totally lacking in Revuz's text. The algorithm is linear in respect of its input data. It was designed by Dominique Revuz and presented in his Ph.D. thesis [31].

5.3 Summary

In the last three chapters, various construction algorithms are presented. This presentation differs from earlier ones. Algorithms are grouped into families forming extensions of a main algorithm. Implementation details are provided. There are four families: two incremental algorithms --- algorithm for sorted data and algorithm for unsorted data, and two semi-incremental algorithms --- Watson's algorithm and Revuz's algorithm. The last one is presented here as an incremental algorithm for constructing pseudo-minimal automata, because we skip its second phase. However, the intention of Dominique Revuz was to construct minimal automata, and the whole algorithm should be classified as semi-incremental.

The main algorithms in the families construct minimal acyclic DFAs. Extensions construct pseudo-minimal acyclic DFAs, add words to minimal cyclic DFAs, construct minimal DTAs, and construct pseudo-minimal DTAs. Each of the algorithms has its niche, where it can be considered the best solution.

Algorithm 5.21 Procedure RevuzAddWord adds word w to the language of an acyclic DFA M. States of the previous word w' are stored in vector P.

```
 1: procedure RevuzAddWord(M, P, w, w')
 2:     q ← q_0; i ← 1
 3:     Mark(q)
 4:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ ∧ FanIn(δ(q, w_i)) = 1 do
 5:         q ← δ(q, w_i); i ← i + 1
 6:         Mark(q)
 7:     end while
 8:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
 9:         δ(q, w_i) ← Clone(δ(q, w_i))
10:         q ← δ(q, w_i); i ← i + 1
11:         Mark(q)
12:     end while
13:     if i > |w| then
14:         P ← ()
15:         F ← F ∪ {q}
16:     else
17:         j ← 1; p ← P_{|w'|}
18:         if P_{|w'|} = q_f then
19:             while j ≤ |w'| ∧ i + j − 2 < |w| ∧ w_{|w|−j+1} = w'_{|w'|−j+1} ∧ |Σ_p| =
    1 ∧ ¬Marked(p) do
20:                 j ← j + 1
21:                 p ← P_{|w'|−j+1}
22:             end while
23:         end if
24:         while i + j − 3 < |w| do
25:             δ(q, w_i) ← new state
26:             q ← δ(q, w_i)
27:             i ← i + 1
28:         end while
29:         δ(q, w_i) ← q_f
30:     end if
31:     P ← (); i ← 1; q ← q_0
32:     Unmark(q)
33:     P_1 ← q_0
34:     while i ≤ |w| do
35:         q ← δ(q, w_i)
36:         Unmark(q)
37:         i ← i + 1
38:         P_i ← q
39:     end while
40: end procedure
```

- For construction of minimal, acyclic DFAs:
  - The incremental algorithm for sorted data should be used whenever data can be presorted. This holds for quasi-static dictionaries that are updated with small supplemental data that can easily be inserted into correct positions in the input data file. This algorithm offers both the highest speed and the smallest memory footprint.
  - The incremental algorithm for unsorted data should be used whenever data comes in an arbitrary order. The algorithm is slower than the algorithm for sorted data, but sorting also takes both time and memory.
  - When only speed matters, Watson's algorithm may be used. This algorithm has much bigger memory footprint than incremental algorithms. In morphological dictionaries, only the leaves of a trie are compressed to one node. It should be noted that this algorithm requires (simple, linear in time) presorting that also takes time, and only some sources indicate that that it may be faster than the incremental algorithm for sorted data.
- For adding words to minimal, cyclic DFAs:
  - The extension of the incremental algorithm for sorted data should be used whenever data comes in sorted series.
  - The extension of the incremental algorithm for unsorted data should be used whenever data comes in arbitrary order and sorting offline cannot be used.
  - The extension of Watson's algorithm should be used when data can be presorted and the whole emphasis is on speed, and not on memory use.
- For construction of pseudo-minimal, acyclic DFAs:
  - The extension of the incremental algorithm should be used whenever the data is quasi-static and can be presorted.
  - The extension of the incremental algorithm for unsorted data should be used whenever the data comes unsorted.
  - The extension of Watson's algorithm can be used when speed is of the highest priority.
- For construction of minimal and pseudo-minimal DTAs:
  - The extension of the incremental algorithm should be used whenever the data cannot be presorted or when small memory requirements are of the highest priority. Cloning states is very expensive, so very small memory footprint of the algorithm comes at a cost of slower processing.
  - Watson's algorithm should be used when speed is the highest concern. It should be noted that intermediate automata are much bigger than minimal.

Some of the algorithms presented in this chapter have been benchmarked. This includes a study by the author of this book [14]. This book is focused on the ease of implementation. For a more formal review of algorithms for constructing minimal, acyclic DFAs, see [40].

Some of the algorithms can be adapted for deletion of strings or trees from automata. Those adaptations are quite simple, but fall outside the scope of this book.

Chapter 6

Hashing

Finite-state automata are used in many domains. In natural language processing, they are used mainly as dictionaries, which does not exclude other uses, like e.g. tagging. Dictionaries can be seen as mappings [20]. Words can be associated with other words (for spelling correction), with morphological features, with frequency counts, translations, sense, etc. When the mapping is not determined by morphology of words, storing it only by means of an automaton would lead to a huge waste of memory, because suffixes of words could not share space, and the automaton would have a form of a trie. A better solution is to keep that sort of information outside the automaton. The minimal (and thus small) DFA would then provide a mapping between words and integer numbers (hash values) that could serve as indexes in external storage. An inverse mapping, i.e. from integer numbers (hash values) to words, can also be easily implemented with minimal automata. This mapping is called minimal perfect hashing. It is hashing as it maps items from a very large domain (here: arbitrary strings) to integer numbers. It is perfect as computing the hash values is injective --- different words always get different numbers, or in other words: there are no collisions. It is minimal as it uses n consecutive numbers for n words. Notice that minimal perfect hashing is possibly only for acyclic automata as their languages are finite sets of finite words.

The solution with minimal automata uses integer numbers stored in all states or in all transitions to compute the word number and to find the word associated with a number. An alternative solution is to use pseudo-minimal automata. In such an automaton, under certain condition, each word has its own, proper transition that is not shared with any other word. Such transition can then be used to store some information associated with the word. It can be a number, but it can also be any other information. When it is a number, it can be any number, i.e. the same number can be assigned to different words, and the range of number is arbitrary. This comes at a cost: a pseudo-minimal automaton is not minimal; it can be much larger than the minimal one. Additionally, the inverse mapping, i.e. from numbers (or any other information) to words cannot be easily and efficiently implemented.

Hashing is not limited to words accepted by DFAs. Tree automata can provide a mapping between trees and numbers and *vice versa*. Just like in case of DFAs, minimal and pseudo-minimal can be used, with the same advantages and disadvantages. XML files are collections of trees. Perfect hashing with DTAs can number those trees. This can lead to efficient storage techniques for XML files.

In minimal DFAs, perfect hashing can be implemented in two ways:

1. by storing integer numbers in states,
2. by storing integer numbers in transitions.

The first method can lead to more memory efficient implementations. The second method is faster, and it can more easily be used in sparse-matrix representations of automata.

6.1 Perfect Hashing with Minimal DFAs --- Numbers in States

In this method, cardinalities of the right languages of states are stored in those states. Figure 6.1 shows a minimal automaton recognizing forms of a Polish verb *pić* with numbers by each state showing cardinality of the right language of that state.

The numbers can be efficiently computed using recursive definition of the right language (2.6):

$$|\vec{\mathcal{L}}(q)| = \sum_{a \in \Sigma_q} |\vec{\mathcal{L}}(\delta(q,a))| + \begin{cases} 1 & \text{if } q \in F \\ 0 & \text{if } q \notin F \end{cases} \tag{6.1}$$

A minimal perfect hashing function effectively numbers words in an automaton. We assume that words are ordered, e.g. lexicographically using some ordering of the alphabet Σ. A hash value of a word is its ordinal number among all words recognized by the DTA. In other words, if we count words from 0, it is the number of words that precede the word in the automaton. An implementation of that hashing (word to hash value mapping) relies on that definition. We assume that states are visited in preorder, following outgoing transitions in the order of their labels. At each state visited during recognition of a word, we count the number of words recognized when visiting states immediately reachable from the state by transitions labeled with symbols that precede the current symbol in the word being mapped. Final states receive a special treatment. They are treated as if a final state had an additional transition labeled with an end-of-word marker symbol that precedes all other symbols. That transition would lead to a final state with no outgoing transitions.

Function Word2Number that converts a word to its hash value is shown as Algorithm 6.1. Apart from the automaton M and the word w, it has two additional parameters: q is the state where counting should start (initially the initial state), i is the number of the first symbol in the word to be considered (initially one).

An inverse mapping can also be easily computed. Algorithm 6.2 lists function Number2Word that finds a word associated with the given number n in an automaton M. It should be invoked with the third parameter set to the initial state, and then it calls itself with successive states along the path that recognizes the word.

Figure 6.1 shows the numbers in states for the automaton in Figure 3.9 recognizing inflected forms of a Polish verb *bić*. Let us compute the hash value for the word *biłyśmy*. We call function Word2Number(M, w=*biłyśmy*, 0, 1). The first symbol on the input is *b*. There is only one transition leaving the initial state 0, and it is labeled with *b*. No transitions have labels that precede *b*, the initial state is not final, so s is set 0, and Word2Number(M, w, 1, 2). There is a single transition leaving non-final state 1, and it has label *i*, so s is set to 0, and Word2Number(M, w, 2, 3) is called. In state 2, there are outgoing transitions labeled

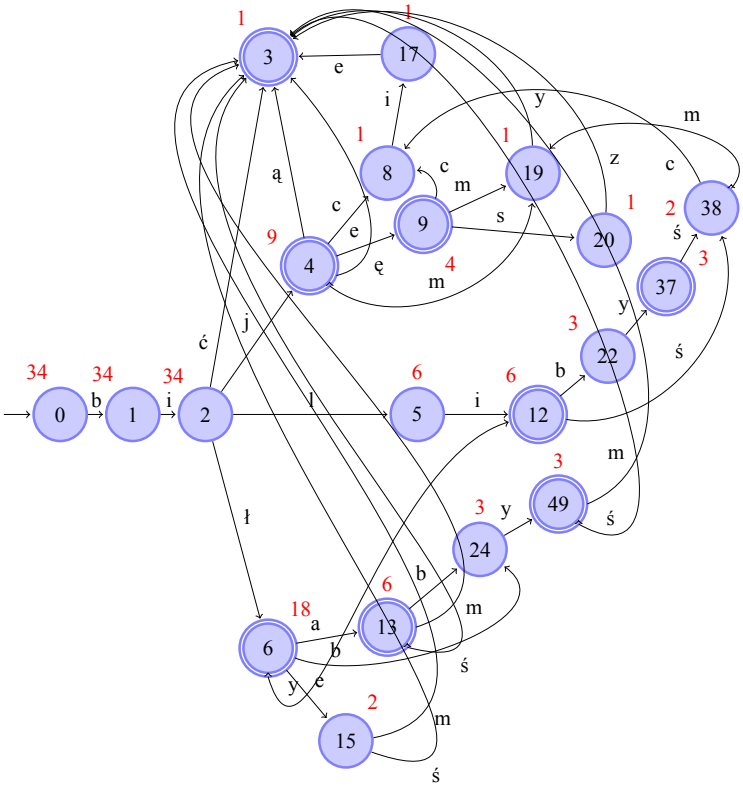


Figure 6.1: Minimal automaton recognizing the same language as the trie automaton in Figure 3.1. A number close to a state shows cardinality of the right language of the state.

Algorithm 6.1 Function Word2Number returns a hash value associated with a word w in a dictionary (a DTA) M. The function calls itself recursively. In the top-level invocation, q should be set to the initial state of the automaton, and i should be set to 1.

```
1:  function Word2Number(M,w,q,i)
2:      if i > |w| then
3:          if q ∈ F then
4:              return 0
5:          else
6:              Error("Word not found")
7:          end if
8:      end if
9:      s ← Σ_{σ∈Σ_q, σ<w_i} | L⃗(δ(q,σ))|
10:     if q ∈ F then
11:         s ← s + 1
12:     end if
13:     if δ(q,w_i) ≠ ⊥ then
14:         return s + Word2Number(M, w, δ(q,w_i), i+1)
15:     else
16:         Error("Word not found")
17:     end if
18: end function
```

Algorithm 6.2 Function Number2Word returns a word associated with the given hash value n in an automaton M. The search starts in state q, which should be the initial state of the automaton in the top-level invocation.

```
1:  function Number2Word(M, n, q)
2:      if n = 0 ∧ q ∈ F then
3:          return ε
4:      else
5:          if q ∈ F then
6:              s ← 1
7:          else
8:              s ← 0
9:          end if
10:         for σ ∈ Σ_q do
11:             if n < s + | L⃗(δ(q,σ))| then
12:                 return σ · Number2Word(M, n − s, δ(q,σ))
13:             else
14:                 s ← s + | L⃗(δ(q,σ))|
15:             end if
16:         end for
17:         Error("Number too big")
18:     end if
19: end function
```

with *ć*, *j*, *l*, and *ł*. As w_i=*ł*, we compute s as the sum of numbers associated with targets of outgoing transitions labeled with *ć* (1), *j* (9), and *l* (6), which gives us 16. We do not have to increment that number as state 2 is not final. At this point, we know that any word in the language of the automaton starting with *bił* must have a hash value greater or equal to 16. We call Word2Number(M, w, 6, 4). The next symbol in the word is y. The sum of numbers associated with targets of outgoing transitions labeled with symbols that precede y in the alphabet is $6 + 3 + 2 = 11$. As state 6 is a final state, we increment s to 12. Then we call Word2Number(M, w, 12, 5). One transition with label *b* leading to a state with number 3 precedes a transition labeled with *ś*, and state 12 is final, so s is set to 4. We call Word2Number(M, w, 38, 6). In state 38, there is one transition preceding a transition labeled with *m*. It is labeled with *c*, and it leads to a state with number 1. As 36 is not final, s is set to 1. We call Word2Number(M, w, 19, 7). In non-final state 19, no transition precedes a transitions labeled with *y*, so s is set to 0, and Word2Number(M, w, 3, 8) is called. As $i > |w|$ and state 3 is final, the function returns 0. One level up in the call hierarchy, in state 19, that 0 is added to the overall result, and returned. One level up, in state 38, the returned value is added to 1, which gives 1. That value is returned. In state 12, that value is added to 4, so the returned value is 5. In state 6, that value is incremented by 12, which gives 17. In state 2, 16 is added to make 33. In states 2 and 1, we also add 0 to the result, so *biłyśmy* (the last word in the language) has number 33.

Let us try the other direction. Which word has number 15? We call Number2Word(M, n=15, q=0). Variable s is set to 0, and the first transition meets the criterion $15 < 0 + 35$, so we call Number2Word(M, n=15, q=1), while remembering σ=*b*. In state 1, we have the same situation with σ=*i*, so we call Number2Word(M, n=15, q=2). We set s to 0. The first transition does not meet the condition as $15 \geq 0 + 1$, so we increment s to 1. With the second transition, we have $15 \geq 1 + 9$, so we increment s to 10. The third transition finally meets the condition $15 < 10 + 6$, so we call Number2Word(M, n=15 - 10, q=5) with σ=*l*. In state 5, the first transition meets the condition $5 < 0 + 6$, we call Number2Word(M, n=5, q=12) with σ=*i*. In state 12, we set s to 1. The first transition does not meet the condition, so we increment s to 4. The second transition does, so we call Number2Word(M, n=1, q=39) with σ=*ś*. In state 38, we set s to 0. For the first transition, we have $1 \geq 0 + 1$, so we increment s to 1. We set σ=*m*, and we call Number2Word(M, n=0, q=19). In state 19, the condition is obviously true for the unique transition there, so we set σ to *y*, and call Number2Word(M, n=0, q=3). As state 3 is final and $n = 0$, the function returns ε. In state 19, we prepend *y* to ε, which gives us *y* that is returned. In state 38, we prepend *m* to *y* to return *my*. In state 12, we prepend *ś* to return *śmy*. While going back to state 0, we prepend other letters that give us the final result --- the word *biliśmy*.

Cardinalities of the right language can also be computed during construction of an automaton regardless of the actual algorithm in use. For example, in procedure AddStrSorted (Algorithm 3.7, page 32), counters should be incremented by one for the initial state, and for each state visited in the first `while` loop. Counters should be set to one for each state created in the second `while` loop.

Various compression methods may change the order of transitions in an automaton. Minimal perfect hashing would still number words in the DFA, but the order of words could be different.

6.2 Perfect Hashing with Minimal DFAs --- Numbers in Transitions

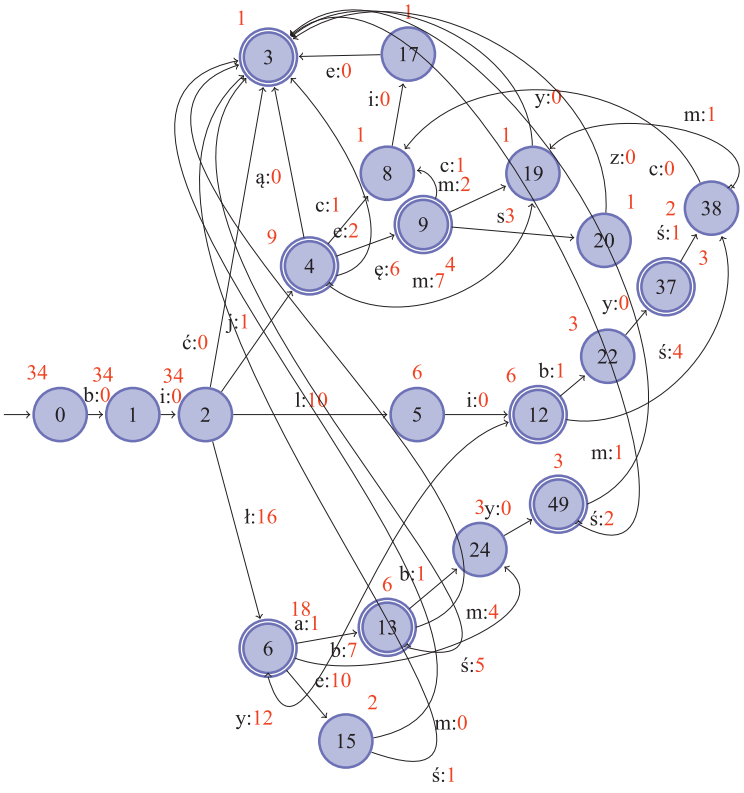


Figure 6.2: Minimal automaton recognizing the same language as the trie automaton in Figure 3.1. Number after a colon in labels of transitions indicate the ordinal number of the lexicographically first word that can be recognized while following the transition.

The method described in the previous section requires visiting states that are not normally visited during recognition of a word that we want to convert to a hash value. This slows down processing. Additionally, a representation method for DTAs that is fast in recognition (sparse matrix representation) would be particularly slow with it. However, it is easy to notice that the value of variable s right before a recursive invocation of function Word2Number can be precomputed and stored directly in an appropriate transition in the automaton. Such numbers are shown in Figure 6.2.

Algorithm 6.3 Function Word2NumberT returns a hash value associated with a word w in a dictionary (a DTA) M using precomputed number stored in transitions. A number stored in a transition $\delta(q, w_i)$ can be retrieved as $\psi(q, w_i)$. The function calls itself recursively. In the top-level invocation, q should be set to the initial state of the automaton, and i should be set to one.

```
1:  function Word2NumberT(M, w, q, i)
2:      if i > |w| then
3:          if q ∈ F then
4:              return 0
5:          else
6:              Error("Word not found")
7:          end if
8:      end if
9:      if δ(q, w_i) ≠ ⊥ then
10:         return ψ(q, w_i) + Word2NumberT(M, w, δ(q, w_i), i + 1)
11:     else
12:         Error("Word not found")
13:     end if
14: end function
```

Algorithm 6.3 lists function Word2NumberT that computes hash values for words using numbers stored in transitions. Function $\psi(q, \sigma)$ returns a number associated with a transition $\delta(q, \sigma)$.

Let us compute again the hash value for the word *biłyśmy*. As the last executed instruction in function Word2NumberT is returning a sum of number and the result of a recursive call, we will compute the sum while going forwards instead of doing it at the end while going backwards. We start by calling Word2NumberT(M, w, 0, 1). We follow $\delta(0, b)$, add 0 to the total sum, and call Word2NumberT(M, w, 1, 2). In state 1, we add 0 to the total sum, and call Word2NumberT(M, w, 2, 3). In state 3, we add 16, and we call Word2NumberT(M, w, 6, 4). In state 6, we add 12, and call Word2NumberT(M, w, 12, 5). In state 12, we add 4, and call Word2NumberT(M, w, 38, 6). In state 38, we add 1, and call Word2NumberT(M, w, 19, 7). In state 19, we add 0, and call Word2NumberT(M, w, 3, 8). In state 3, we have reached the end of the word, and the state is final, so we add 0 to the total sum, which is 0 + 0 + 16 + 12 + 4 + 1 + 0 + 0 = 33. This is consistent with the result in the previous section.

Computing the inverse mapping does require visiting other transitions. We need to remember the symbol on the previous transition as we check which transition leads to words that have a greater ordinal number than the one that is searched for. Function Number2WordT is shown as Algorithm 6.4.

Numbers on transitions can be computed in the same manner as in the previous method, i.e. they can be updated after addition of each new word, but that would mean updating several transitions for a given state instead of just one. Therefore, it is more efficient to compute and store the numbers once the construction process is completed, i.e. non-incrementally.

Various compression methods may change the order of transitions in an automaton. Minimal perfect hashing would still number words in the DFA, but the order of words would different.

Algorithm 6.4 Function Number2WordT returns a word associated with the given hash value n in a DFA M. The search starts in state q, which should be the initial state of the automaton in the top-level call.

```
 1: function Number2WordT(M, n, q)
 2:     if n = 0 ∧ q ∈ F then
 3:         return ε
 4:     else
 5:         σ' ← min σ ∈ Σ_q
 6:         for σ ∈ Σ_q do
 7:             if ψ(q, σ) > n then
 8:                 return σ' · Number2WordT(M, n − ψ(q, σ'), δ(q, σ'))
 9:             else
10:                 σ' ← σ
11:             end if
12:         end for
13:         return σ' · Number2WordT(M, n − ψ(q, σ'), δ(q, σ'))
14:     end if
15: end function
```

6.3 Arbitrary Hashing with Pseudo-Minimal DFAs

Minimal perfect hashing numbers words in the language of an automaton. The numbering scheme is imposed by the automaton; normally it is the lexicographical order of words. This is usually what is wanted, but there are situations when something else is required. We may want to associate specific numbers with specific words. In dynamic perfect hashing, the contents of the automaton change, but we want to keep the mapping for those items that do not change. Finally, we may want to tag words, so that tags do not depend on morphology of words, and there are many different tags, but fewer than the number of words. In those situations, minimal perfect hashing may not be the optimal solution.

Pseudo-minimal automata can be used for implementation of arbitrary hashing. If words have an end-of-word marker appended, such automata have proper transitions, i.e. for each word, the path in the automaton visited during recognition of that word has at least one transition that is not shared with any other word. Such transition is called a proper transition. As it is not shared with other words, it can be used to store *any* number. Actually, it even does not have to be a number; it can be e.g. string. This freedom comes at a price. Pseudo-minimal automata are not minimal --- they can be much greater than the minimal ones.

There can be more than one proper transition for a word. In such case, they form a chain. A number can be stored on any of them, but to facilitate compression, it should be stored on the first such transition counting from the initial state.

The numbers (or other objects that are the image of the mapping) can be stored during construction. In the extension of the incremental algorithm for sorted data, in procedure PseudoAddStrSorted (Algorithm 3.10, page 41), putting the item onto the transition from state q labeled with w'_i should be done just before invoking function PseudoReplOrReg in line 10. By the way, the test $i \leq |w'|$ is needed only for the first word. Afterwards, it will always be true. The item should be kept along with the previous word w' before it can be stored onto

a transition. It is done once for each word. The last item is stored just before the final invocation of function PseudoReplOrReg. An updated version of function PseudoAddStrSorted that handles hashing is presented as function PseudoAddStrSortedH in Algorithm 6.5.

Algorithm 6.5 Procedure PseudoAddStrSortedH adds a string w to the language of an automaton M. Strings must come sorted. The previous word w' has an associated item x'.

```
 1: procedure PseudoAddStrSortedH(M, w, w', x')
 2:     q ← q_0
 3:     i ← 1
 4:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
 5:         q ← δ(q, w_i)
 6:         i ← i + 1
 7:     end while
 8:     if i ≤ |w'| then
 9:         d ← false
10:         put x' onto transition (q, w'_i, δ(q, w_i))
11:         δ(q, w'_i) ← PseudoReplOrReg(M, δ(q, w'_i), w'_{i+1...|w'|}, d)
12:     end if
13:     while i ≤ |w| do
14:         δ(q, w_i) ← new state
15:         q ← δ(q, w_i)
16:         i ← i + 1
17:     end while
18:     F ← F ∪ {q}
19: end procedure
```

The main algorithm is updated as function PseudoSortedIncrementalConstructionH and presented as Algorithm 6.6. Storing the last item requires going to the state q reachable from q_0 with the longest common prefix of the last two words, and putting it onto a transition labeled with the next symbol of the last word.

In the extension of the incremental algorithm for sorted data and in the extension of the Watson algorithm, the items should be placed on the first new transition created for the word with which the item should be associated. When a new word reuses the transition, the item should be moved along the path of its word past the state for which a new transition is created for the new word.

Function PseudoAddStrUnsortedH presented as Algorithm 6.7 is an adaptation of function PseudoAddStrUnsorted for arbitrary hashing. The function has an additional parameter x, which is the item to be associated with the word w. The item is stored on the first new transition created for the word w. Note that there must be at least one new transition created for each word assuming that each word ends with an end-of-word symbol and there are no duplicate words. An item x' associated with some previous word that shares an initial path with the current word w may need to be relocated. The situation happens when w shares the longest common prefix with that word among all words in the language of the automaton. Then either in the first or in the second `while` loop, the item x' is removed from a transition it was stored on. The item is put onto the other outgoing transition of the same state that we have put x. We use the notation $x \leftarrow \emptyset$ to indicate that an item x has already been stored in

Algorithm 6.6 Algorithm for incremental construction of pseudo-minimal automata from sorted data implementing arbitrary hashing.

```
 1: function PseudoSortedIncrementalConstructionH
 2:     Create empty automaton M = ({q_0}, Σ, ∅, q_0, ∅)
 3:     R ← ∅                                          ▷ set empty register
 4:     w' ← ε                                         ▷ previous word
 5:     while input not empty do
 6:         (w, x) ← next word and item
 7:         PseudoAddStrSorted(M, w, w', x')
 8:         (w', x') ← (w, x)
 9:     end while
10:     d ← false
11:     put x' onto the last branching transition in the path of w'
12:     PseudoReplOrReg(M, q_0, w', d)
13:     return M;
14: end function
```

an appropriate place. The main algorithm has to be modified accordingly. As those changes are trivial, we do not list them here.

Similar modifications have to be carried out in Watson's algorithm in function PseudoAddWord (Algorithm 5.6, page 138). As there are no confluence states there, picking up an item for some previous word is handled in one loop instead of two. Otherwise, the changes are almost identical, so we do not list them here.

Contrary to minimal perfect hashing with minimal automata, implementing the inverse mapping in arbitrary hashing with pseudo-minimal automata does not seem to be possible.

6.4 Variable Output Transducers

The name *variable output transducer*, introduced by Denis Maurel, is misleading. Of course, all automata implementing some form of hashing can be seen as transducers, i.e. automata with output. However, it is possible to implement hashing in true transducers, e.g. in transducers implementing morphological rules. Therefore, we treat "variable output transducers" here as just simple automata-recognizers with hashing, or *numbered* automata. Hash values stored on transitions act as a sort of weights.

The main idea behind variable output transducers is that we can ignore earlier weights on a path, and just keep the latest as the output for the mapping. Originally, Denis Maurel used an adaptation of the incremental construction algorithm for sorted data (Chapter 3, page 19) for construction of such automata, but any construction algorithm family described in the previous chapters can be used for that purpose. During the construction process, the hash value associated with a given word is put onto the first transition that is created for that word. The difference between this approach and that for minimal perfect hashing using the incremental algorithm for unsorted data or Watson algorithm is that there is no moving of existing hash values along transitions. The hash values are treated as part of labels for the purpose of minimization.

Algorithm 6.7 Procedure PseudoAddStrUnsortedH adds a string w associated with item x to the language of an acyclic, pseudo-minimal automaton M. No assumption about previously added strings is made.

```
1:  procedure PseudoAddStrUnsortedH(M, w, x)
2:      q ← q_0; x' ← ∅
3:      i ← 1
4:      P ← ε                          ▷ path of non-confluence states in the common prefix
5:      while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ ∧ FanIn(δ(q, w_i)) = 1 do
6:          if transition (q, w_i, δ(q, w_i)) has an item then
7:              x' ← the item
8:              remove the item from the transition
9:          end if
10:         push q onto P
11:         q ← δ(q, w_i)
12:         i ← i + 1
13:     end while
14:     p ← q
15:     j ← i
16:     R ← R \ {p}
17:     while i ≤ |w| ∧ δ(q, w_i) ≠ ⊥ do
18:         if transition (q, w_i, δ(q, w_i)) has an item then
19:             x' ← the item
20:             remove the item from the transition
21:         end if
22:         δ(q, w_i) ← Clone(δ(q, w_i))
23:         q ← δ(q, w_i)
24:         i ← i + 1
25:     end while
26:     if x' ≠ ∅ then
27:         put x' onto transition (q, w'_i, δ(q, w'_i))
28:     end if
29:     while i ≤ |w| do
30:         δ(q, w_i) ← new state
31:         if x ≠ ∅ then
32:             put x onto transition (q, w_i, δ(q, w_i)); x ← ∅
33:         end if
34:         q ← δ(q, w_i)
35:         i ← i + 1
36:     end while
37:     F ← F ∪ {q}
38:     d ← false
39:     PseudoLocalMinimization(M, P, w, PseudoReplOrReg(M, p, w_{j...|w|}, d), p, j, d)
40: end procedure
```

Results reported in [29] show that this implementation of hashing can offer smallest number of states and transitions for at least some data where the number of different hash values is relatively small when compared with the number of words. This method requires more investigation before any definitive conclusions are drown.

6.5 Dynamic Perfect Hashing

When a new word is added to a set of words, and an automaton that recognizes that set is rebuilt, numbering of words done using minimal perfect hashing also changes, unless the word follows all other words in lexicographical order. This situation may be undesirable in many applications. One possible solution is to use pseudo-minimal automata. However, it is also possible to use minimal automata with minimal perfect hashing, and with an additional translation vector. Another solution is to include perfect hashing weights into labels of transitions, and another one is to use variable out transducers.

The choice of a particular solution depends on the characteristics of the data, and on the profile of the desired application. We investigate the choices in more detail.

Dynamic perfect hashing is a mapping from n words to numbers $0, \ldots, n-1$ (or equivalently $1, \ldots, n$) that does not change for words that stay in the language of the automaton even when new words are added or some other words are deleted.

Minimal perfect hashing using minimal DFAs is described in the first two sections of this chapter. It provides static mapping. When a new word is added, the mapping changes. Words that come later in lexicographical order than the newly added word have their numbers incremented by one. To adapt it to dynamic perfect hashing requirements, an additional vector is needed. We call it *translation vector*. Initially, it contains a sequence of numbers $0, \ldots, n-1$, so that if t is a translation vector, then $t[i] = i$. When a new word comes, and it takes the j-th position among all words, then $|\mathcal{L}(M)| - j$ numbers in the vector need to be moved. It will be $|\mathcal{L}(M)|/2$ on average, and $|\mathcal{L}(M)|$ in the worst case. When words come in groups, then adding m words would require moving $\mathcal{O}(|\mathcal{L}(M)|)$ numbers in the vector, and inserting exactly m of them. If the automaton is maintained on-line, which means no compression, then updating the automaton takes time proportional to the sum of the lengths of additional words (and also of words to be deleted, as deletion is very similar to addition, but we do not cover deletion in this book). At each visited state of the automaton, its right language counters need to be incremented by one (in the first method described in Section 6.1), or numbers on a transition that is taken and all outgoing transitions of the state that follow it need to be incremented by one. Rebuilding the automaton from scratch takes linear time with regard to the input size (see the previous chapter). The translation vector occupies space of $|\mathcal{L}(M)|$ integers.

Arbitrary hashing with pseudo-minimal automata is described in the previous section. Maintaining the automaton on-line takes time proportional to the size of the additional input (or words to be deleted), rebuilding the automaton takes time proportional to the size of the whole input data (see the previous chapter). The size of a pseudo-minimal automaton depends heavily on "regularity" of its language. Experiments reported in [22] show that pseudo-minimal automata were from 1.11 (for random data) to 33.47 times greater than minimal ones. The second greatest ratio among 21 automata was 15.36 (for a dictionary of Polish, a highly inflectional language). Differences between various languages strongly influence the ratio,

e.g. for a dictionary of English (a language with hardly any inflection) was 2.39. The worst case scenario for pseudo-minimal automata is Σ^n. It requires $2 + \sum_{i=0}^{n-1} |\Sigma|^i = 2 + \frac{|\Sigma|^n - 1}{|\Sigma| - 1}$ states, and $\sum_{i=0}^{n} |\Sigma|^i = \frac{|\Sigma|^{n+1} - 1}{|\Sigma| - 1}$ transitions. A pseudo-minimal automaton recognizing the language $\{a, b, c\}^3$ is shown in Figure 6.3. The minimal automaton for the same language has $n+1$ states forming a single chain, with $|\Sigma|$ transitions linking each consecutive pair of states, making $n|\Sigma|$ in total. Note that pseudo-minimal automata cannot easily implement inverted mapping.

The third solution consist in including weights (numbers needed for implementing perfect hashing) into labels. This can be explained best with an example. Suppose we want to recognize a language $\{aa, ab, ba, bb\}$. The minimal automaton for that language is shown in Figure 6.4. It is easy to put numbers either on states or on transitions to implement minimal perfect hashing with it. However, the mapping created in that way is determined by the order of words in the automaton and cannot be easily changed. Suppose we want the mapping to be *aa* $\rightarrow$ 0, *ab* $\rightarrow$ 3, *ba* $\rightarrow$ 1, *bb* $\rightarrow$ 2. The minimal automaton cannot provide such mapping on the basis of simple addition of numbers. Therefore, we need to differentiate a state with the right language $\{a,b\}$ reachable with *a* from a state with exactly the same right language reachable from the initial state with *b*. One way of achieving that is to treat weights as parts of labels, so that would be included in the right language of states during minimization. This would prevent merger of states with the same right language in terms of symbols, but different right language in terms of symbols and weights.

To avoid problems with negative weights, an end-of-word marker should be used exactly in the same manner it is used in pseudo-minimal automata. The symbols is appended to each word. As a consequence, the DFA has only one final state.

While it is possible to use other methods of constructing and maintaining such automata, we will describe the simplest one. Words would simply receive consecutive numbers as they are read. In that setting, we can use the incremental algorithm for unsorted data (see Chapter 4, page 63). When adding a new word, we traverse the transitions already existing in the automaton recognizing a prefix of the word. At the same time, we calculate the sum of weights on those transitions. When we need to create a new transition, we put there the difference between the number associated with the word and the calculated sum.

We have no experimental results on this method. However, we can determine what the best case and worst case scenario is. The best scenario is when the mapping is the same as in perfect hashing. The resulting automaton would be minimal. The worst case scenario is similar to that for pseudo-minimal automata. It is the language $|\Sigma|^n$. However, for pseudo-minimal automata, the worst case is achieved for all permutations of the mapping $\mathcal{L}(M) \rightarrow 0, \ldots, n-1$. For automata with weights included into labels, we can obtain both a minimal automaton, and an automaton of size equal to that of the pseudo-minimal automaton. One of several possible mappings that gives the worst case scenario is shown in Figure 6.6. Note that the inverse mapping for this kind of automaton cannot be easily implemented.

The worst case scenario for variable output automata is exactly the same as for pseudo-minimal automata, i.e. it is the language Σ^n. The reason is that that each $|\Sigma|$ words share exactly the same prefix. Therefore, if all those words are to have different hash values, at least $|\Sigma| - 1$ of them must have those values stored on transition labeled with their last symbol. Because hash values are incorporated into labels for the purpose of minimization, states reachable with prefixes $n-1$ long have all the same right language (Σ) cannot be replaced

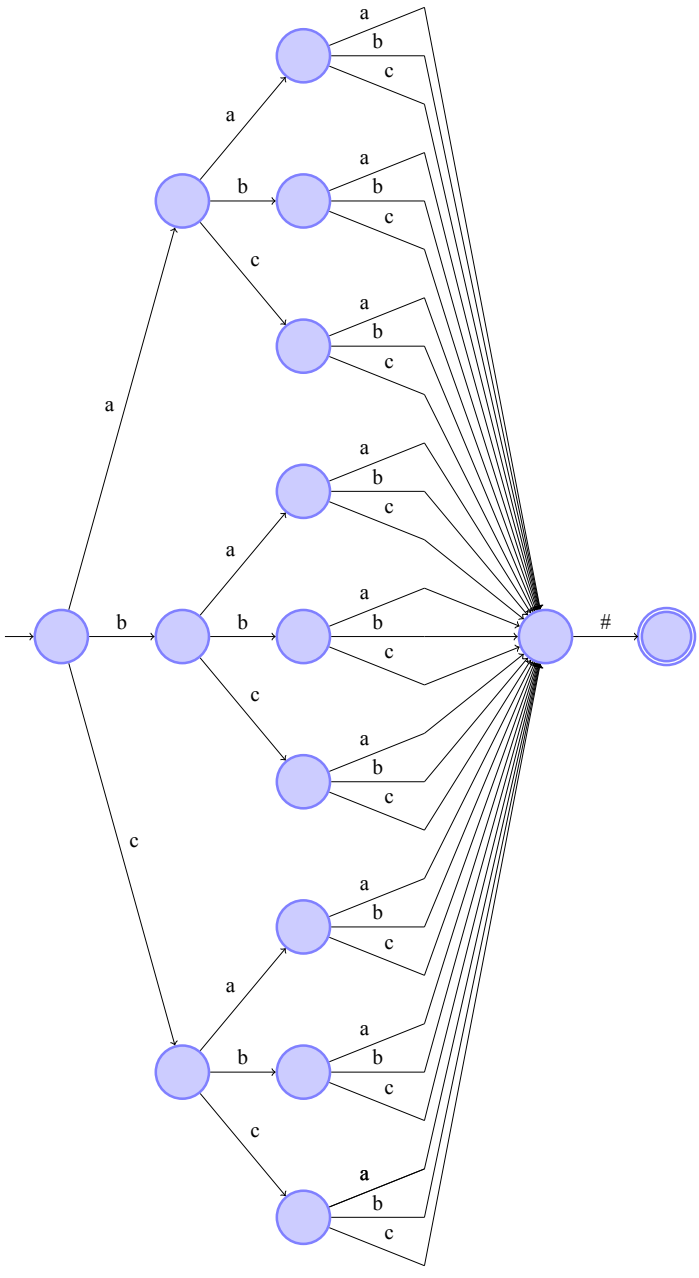


Figure 6.3: Pseudo-minimal automaton recognizing the language Σ^n for $n = 3$.

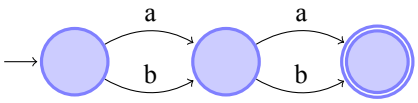


Figure 6.4: Minimal automaton accepting $\{aa, ab, ba, bb\}$

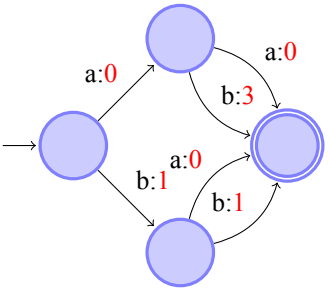


Figure 6.5: Automaton with weights assigning consecutive numbers (starting from 0) to words $\{aa, ba, bb, ab\}$

with a single state.

Theoretical results show limits of various methods. A particular method should be chosen empirically. The size of an automaton measured in the number of states or transitions is only an indication of the size in bytes. Various compression methods interfere with hashing techniques. For example, if we choose variable length encoding in automaton representation, then methods that store fewer numbers (weights) somewhere in the automaton require less bytes for their representation. Also, the smaller the numbers, the fewer bytes they need to represent them. A minimal automaton implementing minimal perfect hashing has numbers in every state or on every transition. Other kinds of automata need fewer of them. Therefore, experiments are vital.

6.6 Perfect Hashing with Minimal DTAs

Just as finite automata recognize words or strings, tree automata recognize trees. Trees play a vital role in many applications. Tree automata are primarily used for recognition and storage of Extensible Markup Language (XML) trees. XML is used for example to store various configuration files. In NLP, it is used e.g. for storing annotated corpora. We may want e.g. to look for sentences where particular syntactic constructions are used. Syntax is mainly expressed with trees, so the task is to recognize a tree, and then to find information that is associated with it. That information does not depend on the shape or labels of the tree. It needs to be stored outside the tree automaton. Just as in case of finite automata, we have access to that information by computing an index in that other data structure. It can be done by means of perfect hashing with DTAs.

There are similarities between hashing with DFAs and with DTAs, but there are also differences. To compute a tree hash value (to determine a tree number), we need to compute how many trees precede it in some ordering imposed by the automaton. To calculate that number in DFAs, we started in the initial state. In bottom-up DTAs, there is no initial state. However, we can compute the number of trees that precede the current tree t in the language of the state $\delta_A(t)$:

$$\iota_A(t) = |\{t' : \delta_A(t') = \delta_A(t) \wedge t' \prec_A t\} \tag{6.2}$$

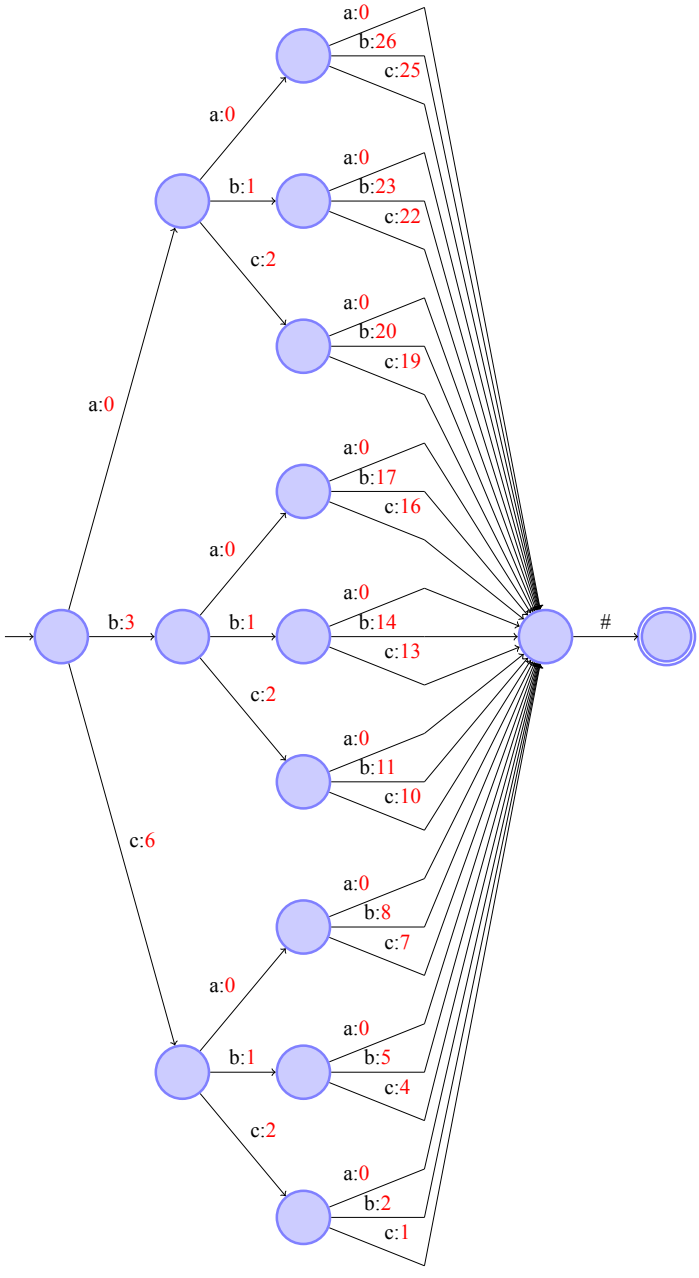


Figure 6.6: Automaton with weights included into labels recognizing the language Σ^n for $n = 3$.

That number can be expressed as a sum of two numbers. Let $q = \delta_A(t)$, and $t = (\sigma, t_1, \ldots, t_m)$. Let $\tau = (\sigma, \delta_A(t_1), \ldots, \delta_A(t_m), \delta_A(t)) = (\sigma, q_1, \ldots, q_m, q)$. The first number is the number of trees $\rho_A(t)$ that precede t but follow the same transition τ. The second one is the number of trees while following transitions that precede τ:

$$\iota_A(t) = \rho_A(t) + \sum_{\tau'=(\sigma', q'_1, \ldots, q'_m, q) \prec_A \tau} |L_A(\tau)| \tag{6.3}$$

where $L_A(\tau)$ is the language of a transition τ defined in Equation (2.32) (page 18). Its cardinality is:

$$|L_A(\tau)| = \begin{cases} 1 & \text{if } \tau = (\sigma, q) \\ \prod_{i=1}^{m} |L_A(q_i)| & \text{if } \tau = (\sigma, q_1, \ldots, q_m, q), m > 0 \end{cases} \tag{6.4}$$

The language of a state can be expressed in terms of the languages of its incoming transitions:

$$L_A(q) = \bigcup_{\tau=(\sigma, q_1, \ldots, q_m, q), m \geq 0} L_A(\tau) \tag{6.5}$$

so its cardinality is:

$$|L_A(q)| = \sum_{\tau=(\sigma, q_1, \ldots, q_m, q), m \geq 0} |L_A(\tau)| \tag{6.6}$$

Before we formalize computation of $\rho_A(t)$, let us see an example. Figure 6.7 shows a DTA that recognizes the language: {*a(a,a)*, *b(a,b)*, *a(a(a,a),a(a,a))*, *a(a(a,a),b(a,b))*, *a(b(a,b),a(a, a))*, *a(b(a,b),b(a,b))*, *b(a(a,a),a(b,a),b)*, *b(a(a,a),a(b,b),b)*, *b(a(a,a),b(b,b))*, *b(b(a,b),a(b,a),b)*, *b(b(a, b),a(b,b),b)*, *b(b(a,b),b(b,b),b)*}. Languages of states and transitions are: $L_A(1) = \{a\}, L_A(b) = \{b\}, L_A(t_5) = \{a(a,a)\}, L_A(t_1) = \{b(a,b)\}, L_A(t_3) = \{a(b,a)\}, L_A(t_6) = \{a(b,b)\}, L_A(t_7 = \{b(b,b)\}, L_A(3) = \{a(a,a), b(a,b)\}, L_A(5) = \{a(b,a), a(b,b), b(b,b)\}, L_A(t_2) = \{a(a(a,a), a(a,a)), a(a(a,a), b(a,b)), a(b(a,b), a(a,a)), a(b(a,b), b(a,b))\}$, and $L_A(t_4) = \{b(a(a,\ a), a(b,a), b),\ b(a(a,a),\}, L_A(4) = L_A(t_2) \cup L_A(t_4)$. Let us compute how many trees precede a tree *t=b(b(a,b),a(b,b),b)* in state 4. We can immediately see that t follows transition t_4. Transition t_2 precedes transition t_4. We can use equations (6.6) and (6.4) to find out, that $|L_A(t_2)| = 4$. We can also see that $t_4 = (b, 3, 5, 2)$. Our tree $t = (b, t_1, t_2, t_3)$ has t_1=*b(a,b)*, which is the second tree recognized in state 2. This means at least $(|L_A(5)||L_A(2)|) = 3$ trees precede t in transition t_4. Tree t_2=*a(b,b)* is the second tree recognized in state 5. This means that $|L_A(2)| = 1$ trees additionally precede t in t_4. Tree t_3=*b* is the only tree recognized in state 2. It follows that our tree *t=b(b(a,b),a(b,b),b)* has number 4+3+1=8.

The number of trees that precede t but follow the same transition τ can be computed as:

$$\rho_A(t) = \begin{cases} 0 & \text{if } t \in \Sigma \\ \sum_{i=1}^{m} \iota_A(t_i) \cdot \prod_{j=i+1}^{m} |L_A(\delta_A(t_j))| & \text{if } t = \sigma(t_1, \ldots, t_m) \in T_\Sigma - \Sigma \end{cases} \tag{6.7}$$

It is more convenient and more efficient to compute $\rho_A(t)$ by computing $\rho^i_A(t)$ defined as:

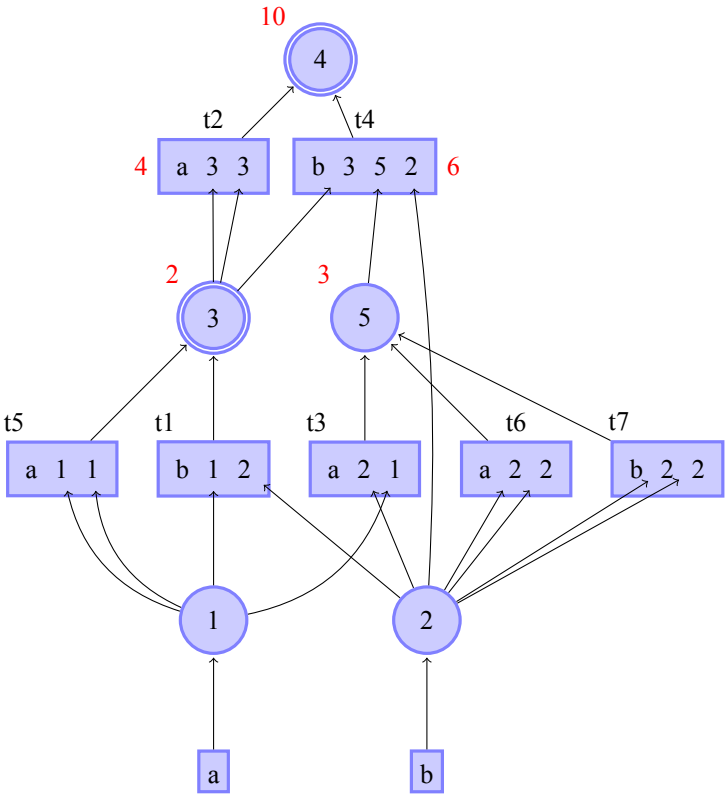


Figure 6.7: Minimal DTA recognizing trees: *a(a,a)*, *b(a,b)*, *a(a(a,a),a(a,a))*, *a(a(a,a),b(a,b))*, *a(b(a,b),a(a,a))*, *a(b(a,b),b(a,b))*, *b(a(a,a),a(b,a),b)*, *b(a(a,a),a(b,b),b)*, *b(a(a,a),b(b,b))*, *b(b(a,b),a(b,a),b)*, *b(b(a,b),a(b,b),b)*, *b(b(a,b),b(b,b),b)*.

$$\rho_A^i(t) = \begin{cases} 1 & \text{if } i = 0 \\ \iota_A^i(t) = \rho_A^{i-1}(t) \cdot |L(\delta_A(t_i))| + \iota_A(t_i) & \text{if } 1 \leq i \leq m \end{cases} \tag{6.8}$$

Thus, $\rho_A(t) = \rho_A^m(t)$.

When state q is final, then $\iota_A(q)$ refers to numbering of trees in the language of the automaton as expressed with Equation (2.19) on page 17. The tree number for t is the sum of $\iota_A(\delta_A(t))$ and $\sum_{f \in F : \delta_A(f) \prec_A \delta_A(t)} |L_A(f)|$, assuming some ordering $\prec_A$ among final states of the DTA.

We can now define function $h_A(t)$ which returns a hash value (a tree number) for a given tree t. It is given as Algorithm 6.8. The function returns -1 if the word is not present in the language of the automaton. The main work in function $h_A(t)$ is done in function rh(t) given as Algorithm 6.9. The rest of function $h_A(t)$ deals with incrementing the returned value with the cardinality of languages of those final states that precede $\delta_A(t)$.

In function rh, the first loop calculates tree numbers v_i for subtrees t_i of the parameter tree t among trees in the language of $q_i = \delta_A(t_i)$. Those values are used to compute $\rho_A^i(t)$. Once the

Algorithm 6.8 Function h_A returns a tree number for a given tree t. If the tree is not present in the language of the automaton, the function returns -1.

```
1:  function h_A(t)
2:      (q, v) ← rh(t)
3:      if q ∈ F ∧ v ≥ 0 then
4:          for f ∈ F : f ≺_A q do
5:              v ← v + |L_A(f)|
6:          end for
7:      else
8:          return -1
9:      end if
10:     return v
11: end function
```

Algorithm 6.9 Function rh calculates a tree number for a given tree t among trees in the language $L_A(\delta_A(t))$. If the tree is not recognized, the function returns -1.

```
1:  function rh(t = σ(t_1, ..., t_m))
2:      h ← 0
3:      for i ∈ 1, ..., m do
4:          (q_i, v_i) ← rh(t_i)
5:          if q_i = ⊥ ∨ v_i = −1 then
6:              return -1
7:          else
8:              h ← h · |L_A(q_{i−1})| + v_i
9:          end if
10:     end for
11:     q ← δ(σ, q_1, ..., q_m)
12:     for Δ ∋ τ = (σ', q'_1, ..., q'_m, q) ≺_A (σ, q_1, ..., q_m, q) do
13:         h ← h + |L_A(τ)|
14:     end for
15:     return (q, h)
16: end function
```

value of $\rho_A(t)$ is computed, it is incremented in the second loop by the cardinality of languages of the transitions that precede transition $(\sigma, q_1, \ldots, q_m, q)$ among incoming transitions of state q. We assume for simplicity that q_0 exists, and that $|L_A(q_0)| = 0$, otherwise we would have to make the code a bit more complex.

Algorithm 6.10 Function h_A^{-1} returns a tree number n in the language of the tree automaton. If there is no such tree, the function returns ε.

```
 1: function h_A^{-1}(n)
 2:     h ← 0
 3:     for i ∈ 1, ..., |F| do                    ▷ F = (f_1, ..., f_{|F|} : f_1 ≺_A ··· ≺_A f_{|F|}
 4:         if h + |L_A(f_i)| > n then
 5:             return rh^{-1}(f_i, n − h)
 6:         else
 7:             h ← h + |L_A(f_i)|
 8:         end if
 9:     end for
10:     return ε
11: end function
```

Let us look at our example again. We call function $h_A(t = b(b(a, b), a(b, b), b))$, which immediately calls function rh(*t=b(b(a,b),a(b,b),b)*). Variable h is set to 0, and the tree is divided into subtrees. Function rh(*t=b(a,b)*) is called. It sets h to 0 at its level, and then it calls rh(*t=a*). Variable h for this invocation is set to 0. As $m = 0$, the first `for` loop does not run. Variable q is set to state 1. State 1 has only one incoming transition, and that transition is used by t, so there are no preceding transitions, and the second `for` loop does not run either. The function returns $(1, 0)$. One level up, q_1 is set to state 1, and v_1 is set to 0. As $v_1 = 0$, h stays equal to 0. Function rh(*t=b*) is called. Variable h is set to 0. As $m = 0$, the first `for` loop does not run. Variable q is set to state 2. State 2 has only one incoming transition, and that transition is used by t, so there are no preceding transitions, and the second `for` loop does not run either. The function returns $(2, 0)$. One level up, q_2 is set to state 2, and v_2 is set to 0. As $v_2 = 0$, h stays equal to 0. Variable q is set to the value of $\delta(b, 1, 2)$, i.e. to state 3. In state 3, there is one transition t_5 that precedes transition t_1 used by the tree. As $|L_A(t_1)| = 1$, h is incremented to 1. The function returns $(3, 1)$. One level up, q_1 is set to 3, and v_1 is set to 1. Then h is incremented to 1. Function rh(*t=a(b,b)*) is called. It sets its local variable h to 0, an calls rh(*t=b*). The function returns $(2, 0)$ as before. One level up, q_1 is set to 2, and v_1 is set to 0. Variable h remains equal to 0. An identical call follows with an identical result. This time, q_2 is set to 2, and v_2 is set to 0. Variable h is still equal to 0. Variable q is set to state 5. There is one transition ---transition t_3--- that precedes transition t_6 among incoming transitions of state 5. As $|L_A(t_3)| = 1$, variable h is incremented to 1. The function returns $(5, 1)$. One level up, at the top level invocation of function rh, q_2 is set to state 5, and v_2 is set to 1. Variable h is set to its previous value (1) multiplied by $|L_A(5)| = 3$, and incremented by $v_2 = 1$, which means that h is set to 4. A call to rh(*t=b*) follows. It returns $(2, 0)$. One level up, variable q_3 is set to state 2, and v_3 is set to 0. Variable h is set to 4 * 1 + 0 = 4. Variable q is set to state 4. There is one transition ---transition t_2--- that precedes transition t_4 used by the tree. As $|L_A(t_2)| = 4$, variable h is incremented by 4 to reach 8. The function returns $(4, 8)$. Back in function h_A, variable q is set to state 4, and variable v is set to 8. State 3 with

$|L_A(3)| = 2$ precedes state 4, so v is incremented by 2 to reach 10. That value is returned by the function as the hash value for tree *b(b(a,b),a(b,b),b)*.

The inverse function can also be computed. Function h_A^{-1} is given as Algorithm 6.10. Just like function h_A, it adjusts the result taking into account other final state, while the actual work is done in function rh^{-1}, which is given as Algorithm 6.11. During invocation of rh^{-1} in h_A^{-1}, h takes the sum of cardinality of languages of final states that precede state f_i such that $f_i = \delta_A(h_A^{-1}(n))$. Tree number n has number $n - h$ among $L_A(f_i)$.

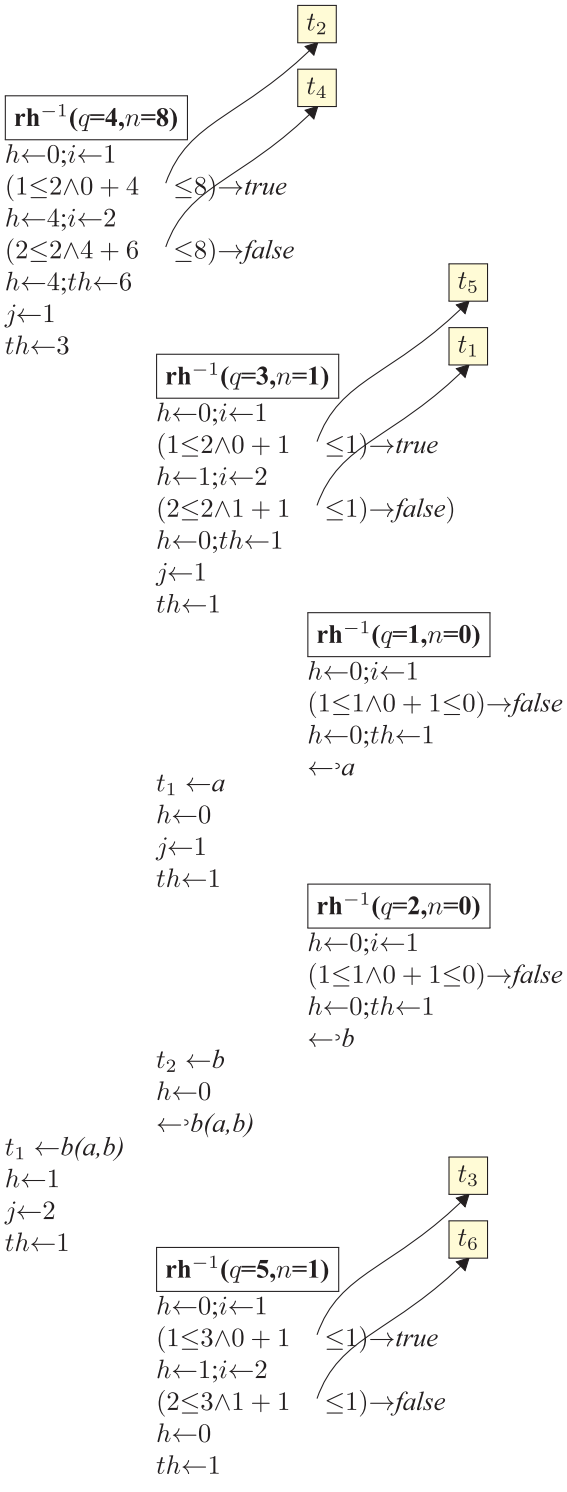
Algorithm 6.11 Function rh^{-1} returns tree number n in $L_A(q)$.

```
 1: function rh^-1(q,n)
 2:     Let N = |{(σ, q_1, ..., q_m, q) ∈ Δ}|
 3:     τ_1 ≺_A ... ≺_A τ_N, τ_i = (σ_i, q_{1_i}, ..., q_{m_i}, q)
 4:     h ← 0; i ← 1
 5:     while i ≤ N ∧ h + |L_A(τ_i)| ≤ n do
 6:         h ← h + |L_A(τ_i)|; i ← i + 1
 7:     end while
 8:     h ← n − h
 9:     th ← |L_A(τ_i = (σ_i, q_1, ..., q_{m_i}, q))|
10:     for j ∈ 1, ..., m_i do
11:         th ← th/|L_A(q_j)|
12:         t_j ← rh^-1(q_j,⌊h/th⌋)
13:         h ← h mod th
14:     end for
15:     return σ_i(t_1, ..., t_{m_i})
16: end function
```

In function rh^{-1}, we must first find the transition that is used by n^{th} tree. We do that by trying them one by one and calculating the sum of languages of the transitions tried so far. Once we have found the correct transition, we decompose the tree to be found into subtrees recognized in the source states of the transition. As in each source state q_j there are $|L_A(q_j)|$ trees recognized there. To calculate a subtree t_j number among the trees in $L_A(q_j)$, we invert Equation (6.8).

Let us see how it works by looking at an example. We take the same automaton as in the previous example. What tree has number 10? We call $h_A^{-1}(10)$. Function h_A^{-1} calls function rh^{-1}. Recursive calls of function h_A^{-1} follow. As the function has a few local variables that keep their different values at different levels of the call hierarchy, a verbal description can be hard to read. Therefore, we decided to use a different method.

$\boxed{h_A^{-1}(n = 10)}$
$h \leftarrow 0$
$j \leftarrow 1$; $f_1 \leftarrow 3$
$(0 + 2 > 10) \rightarrow$ *false*
$h \leftarrow 2$
$j \leftarrow 2$; $f_2 \leftarrow 4$
$(2 + 10 > 10) \rightarrow$ *true*


t_2
t_4
rh^{-1}(q=**4**,n=**8**)
$h\leftarrow 0; i\leftarrow 1$
$(1\leq 2\wedge 0+4 \quad \leq 8)\rightarrow$*true*
$h\leftarrow 4; i\leftarrow 2$
$(2\leq 2\wedge 4+6 \quad \leq 8)\rightarrow$*false*
$h\leftarrow 4; th\leftarrow 6$
$j\leftarrow 1$
$th\leftarrow 3$
t_5
t_1
rh^{-1}(q=**3**,n=**1**)
$h\leftarrow 0; i\leftarrow 1$
$(1\leq 2\wedge 0+1 \quad \leq 1)\rightarrow$*true*
$h\leftarrow 1; i\leftarrow 2$
$(2\leq 2\wedge 1+1 \quad \leq 1)\rightarrow$*false*)
$h\leftarrow 0; th\leftarrow 1$
$j\leftarrow 1$
$th\leftarrow 1$
rh^{-1}(q=**1**,n=**0**)
$h\leftarrow 0; i\leftarrow 1$
$(1\leq 1\wedge 0+1\leq 0)\rightarrow$*false*
$h\leftarrow 0; th\leftarrow 1$
$\leftarrow$*a*
$t_1\leftarrow$*a*
$h\leftarrow 0$
$j\leftarrow 1$
$th\leftarrow 1$
rh^{-1}(q=**2**,n=**0**)
$h\leftarrow 0; i\leftarrow 1$
$(1\leq 1\wedge 0+1\leq 0)\rightarrow$*false*
$h\leftarrow 0; th\leftarrow 1$
$\leftarrow$*b*
$t_2\leftarrow$*b*
$h\leftarrow 0$
$\leftarrow$*b(a,b)*
$t_1\leftarrow$*b(a,b)*
$h\leftarrow 1$
$j\leftarrow 2$
$th\leftarrow 1$
t_3
t_6
rh^{-1}(q=**5**,n=**1**)
$h\leftarrow 0; i\leftarrow 1$
$(1\leq 3\wedge 0+1 \quad \leq 1)\rightarrow$*true*
$h\leftarrow 1; i\leftarrow 2$
$(2\leq 3\wedge 1+1 \quad \leq 1)\rightarrow$*false*
$h\leftarrow 0$
$th\leftarrow 1$

$j \leftarrow 1$
$th \leftarrow 1$
rh^{-1}(*q*=**2**,*n*=**0**)
$\cdots$ (as before)
$\hookleftarrow b$
$t_1 \leftarrow b$
$h \leftarrow 0$
$j \leftarrow 2$
$th \leftarrow 1$
rh^{-1}(*q*=**2**,*n*=**0**)
$\cdots$ (as before)
$\hookleftarrow b$
$t_2 \leftarrow b$
$\hookleftarrow a(b,b)$
$t_2 \leftarrow a(b,b)$
$h \leftarrow 0$
$j \leftarrow 3$
$th \leftarrow 1$
rh^{-1}(*q*=**2**,*n*=**0**)
$\cdots$ (as before)
$\hookleftarrow b$
$t_3 \leftarrow b$
$h \leftarrow 0$
$\hookleftarrow b(b(a,b),a(b,b),b$
$\hookleftarrow b(b(a,b),a(b,b),b$

The time complexity of computing a tree hash value is $\mathcal{O}(|t| \cdot |\Delta| + |F|)$, where $|t|$ is the number of tree nodes calculated as:

$$|t| = \begin{cases} 1 & \text{if } t = \sigma \in \Sigma \\ 1 + \sum_{i=1}^{m} |t_i| & \text{if } t = \sigma(t_1, \ldots, t_m) \in (T_\Sigma \setminus \Sigma) \end{cases} \tag{6.9}$$

$|F|$ is the number of final states, and $|\Delta|$ is the number of transitions in the DTA. There is a loop on final states in function h_A. Its contents is a constant-time operation. Function rh is called $|t|$ times for a tree t. Inside the function, there may be up to $|\Delta| - 1$ previous transitions that have to be included in calculations.

It is possible to significantly improve that complexity by:

1. storing the number of trees recognized by following previous transitions in the current transition (this also eliminates the need for storing back transitions),
2. using an artificial super-root for every tree, so that only one final state will be present (alternatively, one can store the number of tree recognized in previous final states in the current final state).

These modifications lead to complexity of $\mathcal{O}(|t|)$, i.e. hashing is linear with regard to the size of the input.

For inverse of perfect hashing, the complexity is $\mathcal{O}(|t| \cdot |\Delta| + |F|)$. The $|F|$ component comes from a loop on final states in function h_A^{-1}. The $|t|$ component comes from the fact that function rh^{-1} is called exactly once for each node of the resulting tree. The $|\Delta|$ factor comes from the fact that we have to count incoming transitions for every visited node. Unfortunately, that complexity is not easy to reduce. Also note that we have to store back transitions for every node.

In the construction algorithms in Chapter 4 and Chapter 5, we saw how to calculate values of $|L_A(q)|$. Values of $|L_A(\tau)|$ can be calculated in a similar way. Values of $\sum_{\tau' \prec_A \tau} |L_A(\tau')|$ needed for improving time complexity of perfect hashing are best calculated after the automaton has been built.

6.7 Arbitrary Hashing with Pseudo-Minimal DTAs

Pseudo-minimal DTAs have the property that each tree belonging to the language of an automaton has its own state or transition that is traversed during its recognition, that is not shared with any other tree recognized by the automaton. When an additional super-root for a tree is provided, e.g. for a tree t, we store the tree $\lambda(t)$, then each tree has its own transition. That transition is called a *proper transition*, and it can be used to store an arbitrary value of a hash function.

Section 4.5, page 116, describes an extension of the construction algorithm for sorted data to the case of pseudo-minimal DTAs. Section 5.1.4, page 164, describes a similar extension of the Watson algorithm. The only remaining issue is the placement of hash values onto the transitions. There may be more than one proper transition for a given tree. The hash value can be placed onto any proper transition. However, for better compression, it is a good practice to place the values on the first available (closest to leaves) transition. Since we cannot predict, which transitions will be proper, we can either move the hash values as new trees arrive, or we can find them once the construction is completed. The latter does not seem like a good choice. In order to maintain the tree-hash value relation, we would have to reread the collection of trees with their hash values after the construction is completed. The former is a better solution. In function PseudoSplit (Algorithm 4.18, page 119), when a new state and a new transition are created (after line 13), the hash code can be put onto the new transition. In order to prevent that the code is put onto more than one transition (which would not harm perfect hashing, but it would harm compression), a flag should be used. When a code is encountered when traversing existing transitions, it should be removed from its original place, and put onto the other outgoing transition of a state q_k then the first new transition created in line 13. An identical solution can be used in function WatsonPseudoSplit (Algorithm 5.18, page 5.18), only the function is simpler. Figure 6.8 shows an equivalent of a trie for DTAs, where the trees all have a super-root labeled λ. The reader is encouraged to use it to visualize the movement of hash codes.

6.8 Summary

We presented implementation of perfect hashing and arbitrary hashing for both deterministic, acyclic finite-state automata, and for deterministic, bottom-up tree automata. We showed solutions for dynamic perfect hashing and theoretical limits for it.

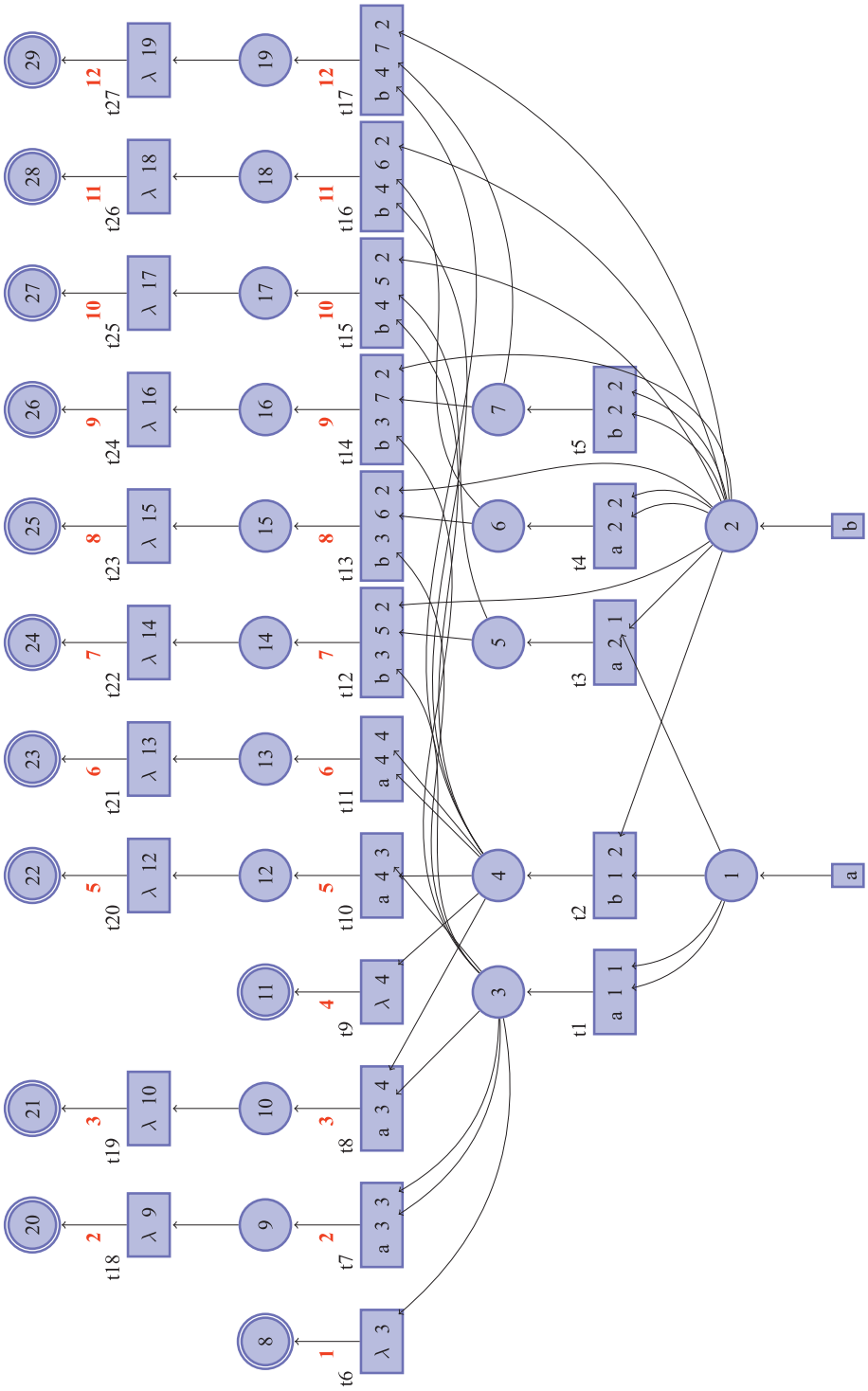


Figure 6.8: An equivalent for a trie for a DTA.

The author of this book implemented all of those methods, and he is also an inventor of hashing with DTAs. DTAs seem to be a promising field for further research, specially in context of annotated corpora.

Chapter 7

Incremental Minimization Algorithms

Among many deterministic finite state automata that recognize the same language there is only one (up to isomorphisms) that has the minimal number of states. This also means that it has the minimal number of transitions. There are several general minimization algorithms for DFAs. Three of them are based on dividing states into two groups: finite and non-finite states, and then refining the groups further based on transitions. The process stops when no divisions can be made. Each group represents a state in a minimal automaton. Those algorithms are:

- Hopcroft algorithm [24] (the fastest in terms of asymptotic worst-case time complexity)
- Aho-Hopcroft-Ullman algorithm [1] (the simplest one)
- Hopcroft-Ullman algorithm [25]

Another algorithm was invented by Janusz Brzozowski [5]. His algorithm differs in several respects from the three listed above. It dos not require a deterministic automaton as its input; the automaton can be nondeterministic. It also does not divide states into groups. An input automaton is reversed so that it recognizes reversals of words that form the language of the original automaton, determinized, reversed again, and determinized again.

Brzozowski's algorithm and the three algorithms based on refinements of groups have something in common: intermediate results are not usable. Either we get the final result at the end of the processing, or we get nothing at all. Bruce Watson [39] came with a solution that is different. It can be interrupted at any time, so that e.g. one can check whether a word belongs to the language of a DFA during minimization of that DFA.

7.1 Original Algorithm by Bruce Watson

The incremental algorithm developed by Bruce Watson starts from a simple observation that if two different states are equivalent, then they can be merged into one state, and the language of the automaton does not change. It is therefore sufficient to check all pairs of states for

equivalence, and merge them if possible, to arrive at the minimal automaton. In the minimal automaton, no pairs of states are equivalent.

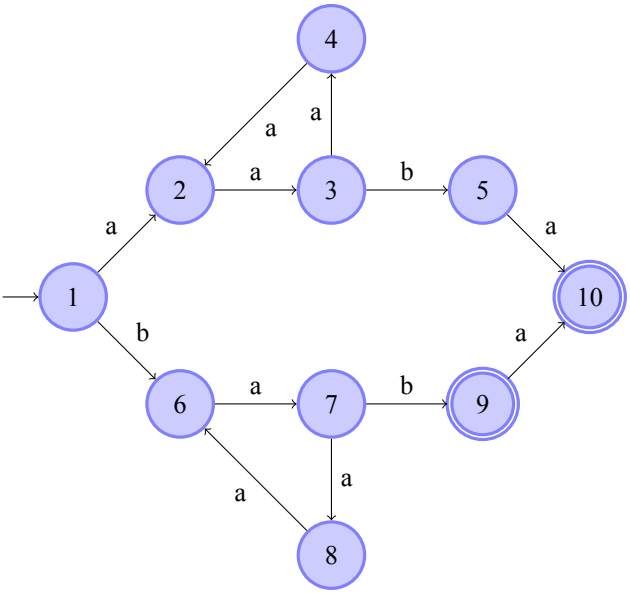


Figure 7.1: A DFA during incremental minimization.

The idea is simple, but the details are not. Let us look at the automaton in Figure 7.1. Suppose we want to check the equivalence of states 2 and 6. Their equivalence depends on the equivalence of states 3 and 7. However, the equivalence of states 3 and 7 depends on the equivalence of states 4 and 8 as well as on the equivalence of states 5 and 9. The equivalence of states 4 and 8 depends on the equivalence of states 2 and 6. There are chains of dependencies that can be very long and complicated.

Two states are equivalent when their right languages are equal. We can use the recursive definition (2.6) of the right language.

$$(p \equiv q) \Leftrightarrow \big((p \in F \equiv q \in F) \wedge (\Sigma_p = \Sigma_q) \wedge \forall_{\sigma \in \Sigma_p} \delta(p, \sigma) = \delta(q, \sigma)\big) \tag{7.1}$$

This leads to the first (basic) version of a function Equiv that checks equivalence of states pairwise. The function is given as Algorithm 7.1.

The most important item in the function is the third parameter S. If in course of checking equivalence we arrive at the same pair of states (regardless of their order inside the pair), then the states are considered equivalent for the purpose of the current evaluation. In other words, if other conditions hold, then the pair must be equivalent. This part is not explained well in [41], so let us focus on it. The purpose of the variable S is to detect cycles. When a pair of parameters is already in S, it means that there is a cycle in the automaton that starts at the pair. It also means that the right language of a state in the pair that is the parameter of the top-level call to function Equiv has the form $u \cup vw^*x_i \cup y_i$, where

- $u \in \Sigma^*$ is any language that is already found the same for both states of the pair,

Algorithm 7.1 First version of function Equiv that checks equivalence of a pair of states p and q. Parameter S contains pairs of states already under examination. The function returns *true* if states p and q are equivalent, and *false* otherwise.

```
1:  function Equiv(p, q, S)
2:      if {p, q} ∈ S then
3:          return true
4:      else if (p ∈ F ≢ q ∈ F) ∨ (Σ_p ≠ Σ_q) then
5:          return false
6:      else
7:          for σ ∈ Σ_p do
8:              if ¬Equiv(σ(p, σ), δ(q, σ), S ∪ {p, q}) then
9:                  return false
10:             end if
11:         end for
12:         return true
13:     end if
14: end function
```

- $v \in \Sigma^*$ is a string leading from the states of the pair in the top-level call to function Equiv to the states of the pair $\{p, q\}$ that starts the cycle,
- $w \in \Sigma^+$ is string that leads from the pair starting the cycle back to the same pair,
- $x_i \in \Sigma^*$ is the right language of a state in the cycle-starting pair without a w^* prefix --- it may be different for those two states, and
- $y_i \in \Sigma^*$ is the rest of the right language of the states in the top-level pair --- again it may be different for the two states in the pair.

So $\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)$ and $p \equiv q$ when $x_p = x_q$.

Let us see an example. We check equivalence of states 2 and 6 in the automaton in Figure 7.1. Function Equiv(2, 6, $\emptyset$) checks that both states are not final and that $\Sigma_2 = \{a\} = \Sigma_6$, and calls Equiv(3, 7, $\{\{2, 6\}\}$). As $\{3, 7\} \notin S$, and both states are not final, and $\Sigma_3 = \{a, b\} = \Sigma_7$, Equiv(4, 8, $\{\{2, 6\}, \{3, 7\}\}$) is called. As $\{4, 8\} \notin \{\{2, 6\}, \{3, 7\}\}$, and both states are not final, and $\Sigma_4 = \{a\} = \Sigma_8$, Equiv(2, 6, $\{\{2, 6\}, \{3, 7\}, \{4, 8\}\}$) is called. This time $\{2, 6\} \in \{\{2, 6\}, \{3, 7\}, \{4, 8\}\}$, so the pair is considered equivalent unless proved otherwise, and the function returns *true*. Since there are no more symbols in Σ_4, Equiv(4, 8, $\{\{2, 6\}, \{3, 7\}\}$) returns true. In Equiv(3, 7, $\{\{2, 6\}\}$), Equiv(5, 9, $\{\{2, 6\}, \{3, 7\}\}$) is called. This time $\{5, 9\} \notin \{\{2, 6\}, \{3, 7\}\}$, but state 9 is final and state 5 is not final, so the function returns *false*. So does Equiv(3, 7, $\{\{2, 6\}\}$) and Equiv(2, 6, $\emptyset$). States 2 and 6 are not equivalent.

Function IncrMin presented as Algorithm 7.2 is the first version of the incremental minimization algorithm. Set U contains unordered pairs of states that can possibly be equivalent and that have not yet been checked for equivalence. Since only pairs with both final states or both non-final states can be equivalent, the set is initialized accordingly. Set E holds pairs of states that we know they are equivalent. The original algorithm also contained a set of states

Algorithm 7.2 Function IncrMin performs incremental minimization of an automaton M.

```
1:  function IncrMin(M = (Q, Σ, δ, q_0, F))
2:      E ← ∅
3:      U ← (F × F ∪ (Q \ F) × (Q \ F)) \ {{p, p} : p ∈ Q}
4:      for {p, q} ∈ U do
5:          if Equiv(p, q, ∅) then E ← E ∪ {{p, q}}
6:          end if
7:          U ← U \ {{p.q}}
8:      end for
9:      Merge pairs of states in E
10:     return M;
11: end function
```

that were not equivalent ---updated with the result of function Equiv--- but the set was not used in minimization, so maintaining it had no sense. Assuming constant time for set operations, this version of the algorithm runs in $\mathcal{O}(|Q|^4)$ time, because there may be up to $|Q|^2$ pairs checked in each top-level invocation of function Equiv, and there may be up to $|Q|^2$ such invocations.

The original algorithm contains two improvements. The set S is made a global variable rather than a parameter. While this leads to faster programs, we prefer to consider it an implementation detail rather than a major modification of the algorithm. The other improvement is a limit on recursion depth. It can be proved that the depth of recursion can be bounded by $\max(|Q| - 2, 0)$, so that value is provided as the third parameter (instead of S), and it is decreased in every recursive call. Since this modification is incompatible with further improvements, we will not use it.

7.2 Faster Minimization

The author introduced three improvements to the algorithm. The first two are simple and result in minor speed increase. The last one is crucial to enhanced time complexity of the algorithm. The improvements are:

1. presorting of states
2. saving only the initial pair of states and states with more than one incoming transitions in S
3. full memoization

7.2.1 Presorting of States

Initialization of variable U in Algorithm 7.2 can be interpreted as division of states into two classes, and using only states from the same class for comparison. Such approach can be enhanced by including more features for distinguishing classes: the number of outgoing transitions and the labels of those transitions. This has two advantages:

1. As there are more smaller classes, there are fewer invocations of function Equiv. Presorting also takes time, but it can be done in $\mathcal{O}(|Q|)$ using bucket sort.

2. Instead of comparing finality, the number of outgoing transitions and labels on those transitions, we compare only class numbers, so comparisons are much faster.

7.2.2 Saving Fewer State Pairs in S

Let us examine parameter S of function Equiv more closely. Its purpose is to detect cycles. The cycles can be formed in two ways:

1. a chain of transitions can lead back to the pair in the initial call,

2. at least one of the states in the pair has more than one incoming transition; one of them leads from a state in the initial pair, another one closes the cycle.

Saving fewer states in S can have two advantages, depending on the implementation of S. It can either speed up the search in S, save memory required for S, or both.

7.2.3 Full Memoization

The original algorithm uses only (positive) results of top-level invocations of function Equiv. However, for a single top-level call, up to $|Q|^2$ pairs of states may be examined. The result of their evaluation is simply thrown away in the original algorithm. The reason for that is that the results from deeper levels of nested calls may not always be directly usable. Let us look again at Figure 7.1. Function Equiv finds that states 4 and 8 are equivalent. However, this is not true. The right language of state 4 is $\overrightarrow{\mathcal{L}}\ (4) = aa\left(ba \cup a\ \overrightarrow{\mathcal{L}}\ (4)\right) = aaa^*ba$ while the right language of state 8 is $\overrightarrow{\mathcal{L}}\ (8) = aa\left(b \cup ba \cup a\ \overrightarrow{\mathcal{L}}\ (8)\right) = aaa^*b \cup aaa^*ba$.

In the following, we will use the notation:

$$\delta^*(\{p,q\},w) = \{\delta^*(p,w),\delta^*(q,w)\},\ p,q \in Q, w \in \Sigma^* \tag{7.2}$$

Results returned by function Equiv(p, q, S) can fall into three categories:

1. Conclusive true. This happens when:

   (a) $p = q$, i.e. p and q are actually the same state, or when the states have no outgoing transitions and they belong to the same class

   (b) Function Equiv returns conclusive true on all pairs of states directly reachable from (p,q) by the same symbol.

   (c) The pair $\{p,q\}$ depends only on itself, i.e. $\exists_{w\in\Sigma^*}\delta^*(\{p,q\},w) = \{p,q\}$ and $\forall_{w\in\Sigma^*}\delta(\{p,q\},w) \notin (S \setminus \{p,q\})$. In this case all pairs depending on $\{p,q\}$ are merged.

2. Conclusive false. This happens when:

   (a) The states are in different classes

   (b) They are already remembered as inequivalent

(c) Any pair of states directly reachable from (p, q) by any symbol are inequivalent.

3. Inconclusive true. This happens when:

   (a) A pair of states $\{r, s\}$ reachable from $\{p, q\}$ is already a parameter in a higher-level call to Equiv. The pair $\{p, q\}$, and all pairs in the call hierarchy from $\{r, s\}$ up to and including $\{p, q\}$ depend on $\{r, s\}$. If other pairs of states depend on $\{p, q\}$, that dependency is transferred to $\{r, s\}$.

   (b) The pair is already remembered as probably equivalent depending on another pair of states.

Let us identify those cases in Figure 7.2. Case 1a is $\{11, 11\}$. Case 1b is not in the figure, but it would be states 5 and 9 if state 5 were final or state 9 non-final. Under the same condition states 2 and 6 would be case 1c. States 5 and 9 are actually case 2a. Case 2b is not in the figure, as remembered inequivalence would be transferred from another top-level call to function Equiv. When function Equiv detects inequivalence, then all recursive calls are immediately completed and no further pairs are examined until another top-level invocation. States 12 and 13 are an example of case 2c when inequivalence of states 5 and 9 has been detected. States 4 and 8 are an example of case 3a when reached from $\{3, 7\}$, and of case 3b when reached from $\{12, 13\}$.

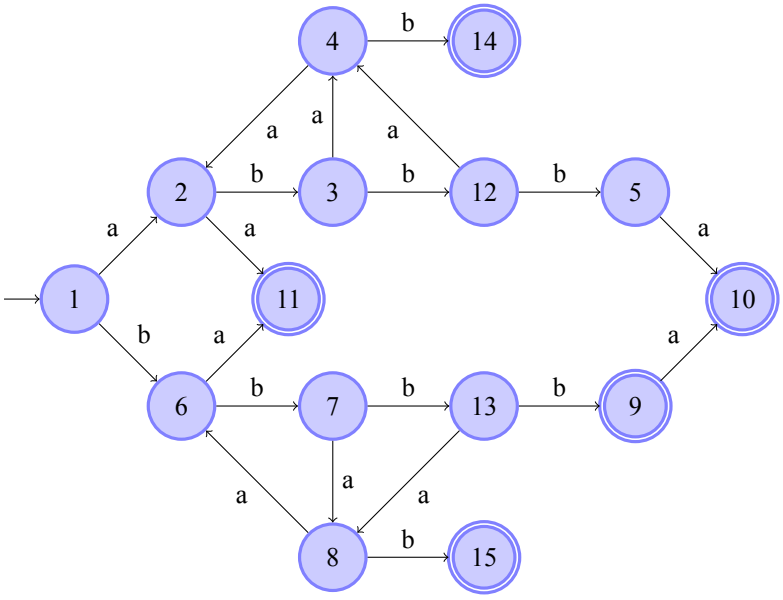


Figure 7.2: Another DFA during incremental minimization.

The case of inconclusive (probable) equivalence requires further examination. When a pair of states is revisited in function Equiv, it means that a cycle has been detected. Until proved otherwise, the pair is considered equivalent. That equivalence is not sure. It is probable, it is inconclusive as we do not have the complete picture. A cycle means that a pair

of states, e.g. $\{r, s\}$ is both at the beginning and at the end of a pair of paths (or a path of pairs). Being at the end of a path means that there are other pairs of states whose equivalence depends on the equivalence of $\{r, s\}$. All pairs in the cycle depend on $\{r, s\}$. They are stored as dependent on $\{r, s\}$. In the course of recursive calls to function Equiv, more dependencies may be detected. When we reach the pair $\{r, s\}$, it may turn out that it depends on no other pair of states. Then all pairs of states that depend on $\{r, s\}$, including $\{r, s\}$, can be merged or can be remembered as inequivalent. If $\{r, s\}$ depends on another pair of states $\{p, q\}$, then states depending on $\{r, s\}$ become dependent on $\{p, q\}$. To prevent formation of chains of dependencies that would require more time to be processed, the union-find algorithm [1] is used to merge dependencies.

7.2.4 Improved Version

The most important part of the algorithm is function Equiv. It is depicted as Algorithm 7.3.

In this improved version, several new variables are used, and an old one is used also in a new way. Let us begin with the variable S. It holds pairs of states that are already under investigation in other invocations of function Equiv that are not yet completed, i.e. they are higher up in the call hierarchy; only those pairs that can start cycles are stored in S. In the original version, S was seen only as a set. In the improved version, it is perceived as both a set and a stack. In particular, it holds information about the recursion level of the invocation in which a pair was a parameter. More precisely, it holds information about the number of pairs stored in S before the pair in question was put there, as not every call to Equiv stores pairs of states in S. Local variable *pushed* is set to *true* if a pair being a parameter of Equiv is stored, and *false* otherwise. Global variable *level* holds the number of pairs stored in S. It can be seen as a stack pointer. Function *index* returns an index of the pair inside the stack. Note that S does not have to be implemented as a structure containing a stack. We can store the index in the stack in a table along with the pair, and instead of looking at the top of the stack, we can find the current pair of states as the pair of two current parameters of function Equiv (actually stored on a program stack).

If the current pair depends on another pair, then global variable *rl* holds an index inside S of that other pair. No dependency is stored as $|Q|^2$. As *rl* is a global variable, it can be used to pass information on the dependency across recursive calls. Local variable *rl′* can collect that information on the local level.

Variable P holds information about dependencies. It is also dual in nature. A list of pairs depending on a pair at the given level l in S is accessible as $P[l]$. However, we can also check whether $\{p, q\} \in P$ regardless of the level of the pair that the pair $\{p, q\}$ depends on. Therefore, P should be implemented as both a table (e.g. a hash table) and a vector of lists. Function *index* returns the level of a pair that the argument pair depends on, or $|Q$ if no such pair exists.

Set I holds pairs of inequivalent states. Equivalent states are merged as soon as their equivalence is discovered. For a given state q, $cl[q]$ gives its class.

In line 3 of function Equiv, we check whether two states are actually a single state. In the top-level invocation of Equiv, the states are always different, but they can be merged at some recursion level. If the states are equal, then Equiv returns *true*. The check is done in constant time. In line 4, it is checked whether the states belong to the same class. This is equivalent to checking finality and the suite of outgoing transitions in the original algorithm, but it is much

Algorithm 7.3 Improved version of function Equiv.

```
 1: function Equiv(p, q)
 2:     rl ← |Q|²; pushed ← false
 3:     if p = q then eq ← true
 4:     else if cl[p] ≠ cl[q] then eq ← false
 5:     else if {p, q} ∈ S then eq ← true; rl ← index({p, q}, S)
 6:     else if {p, q} ∈ I then eq ← false
 7:     else if {p, q} ∈ P then eq ← true; rl ← index({p, q}, P)
 8:     else
 9:         if level = 0 ∨ FanIn(p) > 1 ∨ FanIn(q) > 1 then
10:             S ← S ∪ {{p, q}}; level ← level + 1; pushed ← true
11:         end if
12:         rl′ ← |Q|²; eq ← true
13:         for σ ∈ Σ_p do
14:             eq ← eq ∧ Equiv(δ(p, σ), δ(q, σ))
15:             rl′ ← min(rl′, rl)
16:             if ¬eq then break
17:             end if
18:         end for
19:         rl ← rl′
20:         if pushed then S ← S \ {{p, q}}; level ← level − 1
21:         end if
22:         if eq then
23:             if rl > level then
24:                 Merge{p, q}
25:             else
26:                 P[rl] ← P[rl] ∪ {{p, q}}
27:             end if
28:         else
29:             I ← I ∪ {{p, q}}
30:         end if
31:         if rl = level then rl ← |Q|²
32:         end if
33:         if eq then
34:             if rl = |Q|² then
35:                 ∀_{{r,s}∈P[level]}Merge({r, s})
36:             else
37:                 P[rl] ← P[rl] ∪ P[level]
38:             end if
39:         else
40:             I ← I ∪ P[level]
41:         end if
42:         P[level] ← ∅
43:     end if
44:     return eq
45: end function
```

faster. It is done in constant time. In line 5, cycles are detected by checking whether the pair $\{p, q\}$ is already present in S. If it is so, the function returns *true*, and variable *rl* is set to the index of $\{p, q\}$ in S (seen as a stack). This can be done in constant time. Note that in the earlier instructions, *rl* was not modified; it maintained its value from a higher level *Equiv* invocation. In line 6, it is checked whether the states are already known as inequivalent; in that case, the function returns *false*. The check can also be done in constant time. This completes the checks that can be done at the current recursion level.

In lines 9--11, it is determined whether the pair can start a cycle. If it is so, the pair is stored in S, with the stack pointer *level* incremented, and the variable *pushed* set to true. Line 12 is a preparation for checks done in deeper recursion levels. Local variable *rl'* is set to $|Q|^2$ to indicate that the current pair depends on no other pair, and *eq* is set to true to indicate that the pair is considered equivalent until proved otherwise.

The loop in lines 13--18 launches checks done for targets of transitions going out from the current pair of states. As soon as the states are found to be inequivalent, the loop is interrupted as there is no point in checking more targets of outgoing transitions --- they would not change the result. Function Equiv is called with targets of transitions going out of p and q. Local variable *rl'* is updated so that the global variable *rl* can be reset at deeper levels. At the exit of the loop, the value of *rl* is synchronized with the local variable *rl'* to communicate dependency upwards in the call hierarchy. This is done in constant time.

Once the subordinate calls are completed, the pair $\{p, q\}$ can be removed from S in constant time in lines 20--21. If the result of equivalence check for the pair is positive, there are two positives. Either the result is conclusive, in which case $rl > level$, and the pair can be merged in line 24 (using Union-Find algorithm), or the result is inconclusive, and it depends on the equivalence of a pair at rl level. In the latter case, the pair $\{p, q\}$ is added to pairs depending on the pair at level rl in line 26. This can be done in constant time (adding an item to a list and to a table). If $rl = level$, then the pair depends only on itself, so rl is reset to $|Q|^2$ in line 31 to indicate that there is no longer any dependency. If it is so, and the pair is equivalent, then all pairs depending on $\{p, q\}$ are merged in line 35. If the result is positive, but there is still a dependency on a pair at rl level, the list of pairs depending on $\{p, q\}$ is merged with the list of dependencies of the pair at rl level, and the merged list is stored as a dependency list for the pair at rl level. If the result is negative, pair $\{p, q\}$ is added to I, i.e. to a set of pairs known to be inequivalent. The dependency list for the current level is reset in line 42.

We will give no pseudo-code for the main function for incremental minimization, as it would not shorten the text. First, all states are sorted on their finality and on the suite of their outgoing transitions using bucket sort. Sorting divides states into classes. For each class, states are compared pairwise. As each top-level call to Equiv may compare many pairs from various classes, it also updates the set I of inequivalent states. Information from I is used to limit the comparisons at the top level. It also helps in creating new classes.

Let us minimize the automaton in Figure 7.2. We call bucket sort, and get the classes as in Table 7.1.

0								8	9				13	
2	6	3	4	5	1	7	8	9	10	11	14	15	12	13

Table 7.1: Division of states in automaton in Figure 7.2 into classes after bucket sort.

We set *level* to 0. We call Equiv(2, 6, $S = \emptyset$). We set *rl* to 175. States 2 and 6 belong to the same class, and all S, I and P are empty. As *level* is zero, we put $\{2, 6\}$ into S, associating it with level 0, increment *level*, and set *pushed* to *true*. In preparation for nested calls, *rl'* is set to 175, and *eq* is set to *true*. Variable σ is set to *a*, and Equiv(11, 11) is called with $S = \{\{2, 6\}\}$. It immediately returns true with $rl = |Q|^2$. Then σ changes to *b*, and Equiv(3, 7) is called, with $S = \{\{2, 6\}\}$. The states are different, but in the same class, and they are not in S, I, nor in P. As they are not the initial pair, and they both have a single incoming transition, they are not stored in S, *level* remains unchanged, and *pushed* stays *false*. Variable *rl'* is set to 175, and *eq* is set to *true*. The `for` loop is entered. Variable σ is set to *a*, and Equiv(4, 8) is called, with $S = \{2, 6\}$. States 4 and 8 are in the same class, and they are not in S, I nor in P. They do have more than one incoming transition, so the pair is stored in S at level 1, with variable *label* incremented, and variable *pushed* set to true. Local variable *rl'* is set to 175, and *eq* is set to true. With σ=*a*, Equiv(2, 6) is called, with $S = \{\{2, 6\}, \{4, 8\}\}$. This time, $\{2, 6\} \in S$, so *rl* is set to 0, and the function returns *true*. Back in Equiv(4, 8), *rl'* is set to 0, σ is set to *b*, and Equiv(14, 15) is called, with $S = \{\{2, 6\}, \{4, 8\}\}$. Global variable *rl* is set to 175, *pushed* is set to *false*, states are in the same class, and they are not in S, I or P. They cannot start a cycle, so instructions in line 10 are not executed. There are no outgoing transitions, and nothing was saved in S, so *eq* stays *true*. States 14 and 15 are merged, i.e. state 15 is deleted, and its incoming transition is redirected to state 14. Back in Equiv(4, 8), the loop terminates, *rl* is set to 0, the pair $\{4, 8\}$ is removed from S, and global variable *level* is decremented. Pair $\{4, 8\}$ is made dependent on pair $\{2, 6\}$ in line 26. The function returns *true*. Back in Equiv(3, 7), *rl* is set to 0, σ is set to *b*, and Equiv(12, 13) is called, with $S = \{2, 6\}$. Global variable *rl* is set to 175, and local *pushed* to *false*. The states are in the same class, they are not in S, I, nor they are in P. Since they cannot start a cycle, they are not stored in S. Local variables *rl'* and *eq* are set to $|Q|^2$ and *true*, respectively. Variable σ is set to *a*, and Equiv(4, 8) is called, with $S = \{\{2, 6\}\}$. Variable *rl* is set to 175, and *pushed* is set to *false*. States 4 and 8 are in the same class, they are no longer in S, they are not in I, but they are in P, so *rl* is set to 0, and *true* is returned. Back in Equiv(12, 13), *rl'* is set to 0, σ is set to *b*, and Equiv(5, 9) is called, with $S = \{\{2, 6\}\}$. Global variable *rl* is set to 0. States belong to different classes, so the function returns *false*. Back in Equiv(12, 13), the loop is finished, and *rl* is set to 0. Restoring S is skipped, and I is set to $\{\{12, 13\}\}$. The function returns *false*. Back in Equiv(3, 7), the loop finishes. No restoration of S is needed. The pair $\{3, 7\}$ is added to I, and the function returns *false*. Back in Equiv(2, 6), *rl'* is set to 0, and so is *rl*. States 2 and 6 are removed from S, and *level* reaches 0. The pair $\{2, 6\}$ is added to I. As *rl* and *level* are now equal, *rl* is set to 175. Pair $\{4, 8\}$ is added to I, so that $I = \{\{12, 13\}, \{3, 7\}, \{2, 6\}, \{4, 8\}\}$.

The algorithm has an initialization phase, and comparison done inside classes. In the initialization, a bucket sort is invoked. It runs in time proportional to the number of states. Global variables S, I, and P are either two-dimensional arrays, or structures that contain two-dimensional arrays, or hash tables of length proportional to $|Q|^2$. Their initialization takes $\mathcal{O}(|Q|^2)$ time. Function Equiv is invoked at the top level at most $|Q|^2$ times. For each top-level call, the function may be invoked up to $|\Sigma||Q|^2$ times recursively. Lines 9--42 are executed at most $|Q|^2$ times; previous lines may be executed $|\Sigma|$ times more, and it is possible to limit the number of calls to Equiv like it is done in Section 7.3, however it would not change the time complexity since the same instructions are executed, and $|\Sigma|$ is only a constant factor. When the top-level call is finished, the equivalence or inequivalence of each visited pair is known. Computational complexity of most operations inside function Equiv has been given

when describing the function. Those operations run in constant time. What remains to be explained is:

1. Merging states in lines 24 and 35. One merger can be done in constant time (a vector can be used for redirection to actual structures representing states; redirection means replacement of pointers). Various numbers of mergers can take place during a single call, but there cannot be more mergers than there are states in the automaton. When more mergers take place, one can use Union-Find algorithm to merge merged states (see below).

2. Variable S can be implemented as a two-dimensional array or a hash table indexed with a pair of states, with values being stack pointers for the pairs. The stack itself does not need to be implemented. Adding a pair to S takes a constant amount of time, and so does finding a pair and removing a pair from S.

3. Variable I can be implemented as a two-dimensional array (it can be bit packed) or a hash table indexed with a pair of states. Adding, searching or removing a pair of states takes constant amount of time.

4. Variable P is implemented as a structure containing an array or a hash table closely resembling the one that is used in S, and a vector indexed with the number of pairs stored in S with values being lists of pairs of states. Operations on the array or the hash table take constant amount of time each. Adding a pair to a list also takes a constant amount of time. Moving a list of pairs from P to I in line 40 takes time proportional to the length of the list. There cannot be more than $|Q|^2$ pairs there across all calls to function Equiv. Merging two dependency lists in line 35 can be done using Union-Find algorithm [1]. As there cannot be more than $|Q|^2$ items in all those lists in total, and a pair cannot be on such list in two separate top-level calls to Equiv, executing line 37 across all calls takes $\mathcal{O}(|Q|^2 G(|Q|^2))$, where $G(n)$ is the inverse Ackermann function.

Therefore, the complexity of the whole algorithm is $\mathcal{O}(|Q|^2 G(|Q|^2))$. That complexity is larger than most other minimization algorithms, except for Brzozowski algorithm. It should also be noted that „almost quadratic" complexity does not mean that the algorithm runs almost the same time as e.g. Aho-Hopcroft-Ullman algorithm for the same data. The incremental algorithm is much more complicated, so the constant factor is much larger. However, it is the only algorithm that can be interrupted at any time without loosing intermediate results.

The improvements introduced in this version have various influence on the execution time of the algorithm. Omitting saving in S pairs of states that cannot start cycles had only a small effect. During experiments conducted for [41], most states had more than one incoming transition, and only in one case the greatest number of pairs in S was reduced from 24 to 10. There may be, however, some applications where automata have different characteristics, and where this improvement would induce more savings.

Benefits of presorting of states also depend on characteristics of input automata. The smallest or no improvement was measured on automata with only a few classes. However, for automata with more classes, presorting was always beneficial, with the average reduction of calls to function Equiv equal to 16.95% for real-life automata used in the experiments. In one of the cases, it was reduced by more than a half.

The greatest improvement was registered with full memoization. For some automata, e.g. for an automaton with 1 971 states and 24 633 transitions (26 states in the minimal automaton) as well as for an automaton with 3 186 states and 12 077 transitions (46 states in the minimal automaton) the execution time for the original version of the algorithm was more than 24 hours on the computer used in the experiments at that time, so it was not measured. In cases when the execution time could be measured, the time was reduced up to 6 000 times, although for some automata, improvement was not noticeable.

This version of the algorithm was developed and implemented by the author. It was first published in [41]. That paper contains an error: limiting recursion depth is incompatible with the full memoization. That implementation has been tested on many real-life automata, and many automatically generated random automata. Only recently an example of an automaton that was not minimized correctly has been discovered by Marco Almeida. We corrected the error in this book. The correction involves not only omitting the recursion depth testing, but also changing the value used for indicating no dependency. Originally, it was set to $|Q|$, it is $|Q|^2$ now.

7.3 Simplified Version

Marco Almeida, Nelma Moreira, and Rogério Reis did not stop at detecting the error in the improved version of the algorithm. They developed their own version that is a simplification of the improved version.

Algorithm 7.4 Simplified version of function Equiv proposed by Marco Almeida, Nelma Moreira, and Rogério Reis.

```
 1: function EquivAMR(p, q)
 2:     if {{p, q}} ∈ I then return false
 3:     end if
 4:     if Cl[p] = CL[q] then return true
 5:     end if
 6:     S ← S ∪ {{p, q}}
 7:     for σ ∈ Σ_p do
 8:         {p', q'} ← {Cl[δ(p, σ)], Cl[δ(q, σ)]}
 9:         if p' ≠ q' ∧ {p', q'} ∉ E then
10:             S ← S ∪ {p', q'}
11:             if EquivAMR({p', q'}) then
12:                 return false
13:             end if
14:         end if
15:     end for
16:     S ← S \ {{p, q}}
17:     E ← S ∪ {{p, q}}
18: end function
```

Function Equiv in that version is listed here as function EquivAMR in Algorithm 7.4. It is different from [2] because we introduced corrections discussed with one of the authors; the

same corrections were introduced in FAdo --- the software they used for their experiments. We have also renamed some variables and changed the style so that it resembles more closely our version of the algorithm. Variable E is a set of potentially equivalent states. When the top-level call to function EquivAMR returns *true*, pairs of states in that set are merged. However, the process of merging is different from ours as the states keep their identity after the merger. We introduced variable Cl (different from cl used in the earlier version) to store information about merged states. When the top-level call returns *false*, the pairs in S are added to I. However, contrary to our version of the algorithm, when any nested EquivAMR call returns *false*, variable S stays intact holding the whole path from the initial pair to the first pair that turned out to be inequivalent.

Nelma, Marco and Rogério claim that their version is faster than our version. It may be so because our handling of dependency lists is time-consuming, but it is surprising as some information gained in nested Equiv calls is lost in their version. They claim the time complexity of their algorithm is $|\Sigma||Q|^2\alpha(|Q|)$, where $\alpha(.)$ is related to the inverse Ackermann function. However, their Lemma 5 is incorrect --- if EquivAMR returns *false*, the pair being an argument of a nested EquivAMR call can be reused as an argument in a top-level call as S does not store all visited pairs. That error should be corrected in [3].

7.4 New Version

If the bookkeeping done in our improved version is responsible for longer execution time than the simplification done by Nelma, Marco and Rogério, then we can propose another simplification of the improved version that does not loose so much information computed in nested Equiv calls. Also for simplicity, we do not include our second improvement, but it is always possible to include it.

It is quite similar to our improved version, but it does not fully manage the set of potentially equivalent pairs of states. This should speed up the algorithm if the simplicity hypothesis turns out to be true. The set of potential states depends always on the pair that is the highest in nested call hierarchy. When an inequivalent pair is encountered and the set is not empty, the result for those pairs is waisted, just like in the Portuguese version. However, if the dependencies are resolved before the inequivalent pair is found, the result is kept, i.e. the pairs are merged. Also, the reaction to conclusive results (also when dependency has just been resolved) is immediate --- either the pairs are merged, or they are added to I without waiting until the top-level call is completed.

Since this is only a variant of the earlier algorithm, we leave the proof of the correctness as an exercise for the reader.

Algorithm 7.5 New simplified version of function Equiv.

```
 1: function EquivN({p, q})
 2:     rl ← |Q|²
 3:     if p = q then eq ← true
 4:     else if cl[p] ≠ cl[q] then eq ← false
 5:     else if {p, q} ∈ S then eq ← true; rl ← index({p, q}, S)
 6:     else if {p, q} ∈ I then eq ← false
 7:     else if {p, q} ∈ P then eq ← true; rl ← index({p, q}, S)
 8:     else
 9:         S ← S ∪ {p, q}; level ← level + 1
10:         rl′ ← |Q|²; eq ← true
11:         for σ ∈ Σ_p do
12:             eq ← EquivN({δ(p, σ), δ(q, σ)})
13:             rl′ ← min(rl′, rl)
14:             if ¬eq then
15:                 I ← I ∪ {{p, q}}
16:                 return false
17:             end if
18:         end for
19:         rl ← rl′
20:         S ← S \ {{p, q}}; level ← level − 1;
21:         if rl > level then
22:             Merge(p, q)
23:         else
24:             P ← P ∪ {p, q}
25:         end if
26:         if rl = level then rl ← |Q|²
27:         end if
28:     end if
29:     return eq
30: end function
```

Chapter 8

Memory-Efficient Representations

Two previous chapters discussed algorithms for construction of minimal automata or for minimization of automata. A minimal automaton is smaller than a non-minimal one. However, when the size in bytes is concerned, there are other factors that influence it. The most important one is the representation of the automaton in memory. In this chapter, we present various techniques that reduce memory footprint of deterministic finite-state automata. The techniques can be classified into several families. Some methods are incompatible with others.

Let us take a definition of a DFA $M = (Q, \Sigma, \delta, q_0, F)$. The alphabet Σ usually does not have to be represented explicitly. If it does, it is when the default coding of symbols takes too much space (e.g. two bytes when the number of symbols is smaller than 255), and the symbols have to be recoded. The initial state is either known by its position (the first state) or it is coded as a single integer, so its influence on the size of the whole automaton is negligible.

Final states can theoretically be represented as a set of states, e.g. as a set of integer state numbers. Unfortunately, such representation is impractical, because testing finality of a given state would take too much time. Therefore, finality can be treated as an attribute of a state. It can either be stored in a structure representing a state, or it can be stored apart in a vector. As it is a binary feature, this can be a bit vector, i.e. a single state in that vector would take a single bit. Finality is a concept that is slightly artificial. It can be seen as an output of an automaton. There are two types of automata with output: Moore's automata and Mealy's automata. In the first ones, the output is stored in states, in the latter ones --- in transitions. Analogically, instead of automata with final states, we can have automata with final transitions. Ciura and Deorowicz [10] call them Mealy's recognizers, and that seems to be a very good name. Such automata have fewer states and fewer transitions than the traditional automata. Finality of transitions can be stored in a similar way as for states. Even when representing traditional automata, finality can also be stored in transitions representing finality of their target states. It is also possible to get rid of final states altogether. This is done by appending an end-of-string or end-of-word marker at the end of each input string. The marker cannot belong to the alphabet of the input strings. The resulting minimal automaton would have only one final state, and that state would have no outgoing transitions. In comparison with a traditional minimal automaton recognizing the same language, the new automaton would have one additional state --- the final state, and transitions from the old final states leading to the new final

states labeled with the marker.

The states can be represented as structures holding information about finality and about the suite of outgoing transitions of a state. However, in most efficient representations, states are represented only implicitly; they are sets or lists of transitions. One possible representation of a state is a set of transitions, which is an implementation of a transition table. A state is a vector of transitions, with transitions indexed by their labels. When there is no outgoing transition with a particular label for a state, then the place is left ``empty''. Note that it is normal for an automaton implemented in software to be incomplete rather than complete, so there will normally be a lot of empty room in the transition table. The other implementation is a transition list. The list can either be represented as a vector with a transition counter, or as a sequence. Since the states are implicit, the transition counter would be placed in incoming transitions of a state (the initial state would need an artificial incoming transition). When the list is represented as a sequence, it can be a list with pointers, but it is more efficient to put transitions one after another with a single bit flag indicating the *last* (outgoing) transition of a state.

Transitions carry information about their source state, their label, and their target state. They can also carry additional information related to states. The source state is usually implicit. An automaton is basically represented as a vector of transitions.

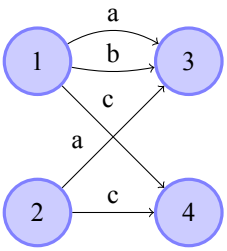
Compression methods fall into several categories:

1. representing values of transition fields with smaller amount of memory
   (a) single bit flags
   (b) fixed size in bytes or bits
   (c) variable size
2. sharing space
   (a) storing states inside other states
   (b) storing parts of states inside other states
   (c) storing transitions of states interdigitated (superimposed coding)
   (d) storing repeated sequences of transitions in one place
3. using frequency --- reordering states or transitions so that either
   (a) more space can be shared
   (b) some more frequent values can be represented with less memory

We will describe those methods with more detail, focusing on those techniques that have been successfully implemented by the author of the book.

8.1 Sharing Space

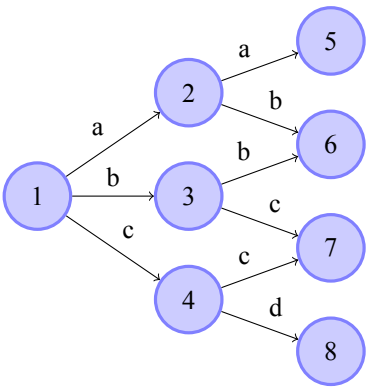
Certain transitions of one state can be stored inside another state. There are several variants of this method.



	l	L	t
1	b		4
2	a		4
3	c	○	7
4	⋮		⋮

Figure 8.1: Compression by sharing space: a state is stored inside another state. Fields of a transition are: label (l), last transition (L) flag, target (t). We assume that state 3 starts at transition 4, and state 4 starts at transition 7.

Let us see an example. Figure 8.1 shows a part (a few states) of an automaton. State 1 has three transitions labeled with *a*, *b*, and *c*. State 2 has two transitions with labels *a* and *c*. Transitions labeled with *a* and *c* have the same targets for both states. State 2 can be stored entirely inside state 1. A state is represented as a list of transitions, with the last transition having a flag `L` (last transition of a state) raised. State 1 starts at transition 1, state 2 starts at transition 2. Both states end with transition 3. Note that in order to use transitions of state 1 to represent also transitions of state 2, transitions of state 1 had to be reordered.

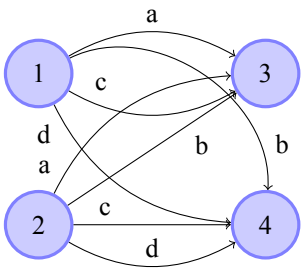


	l	c	t
1	a	2	4
2	b	2	5
3	c	2	6
4	a	1	8
5	b	1	9
6	c	1	10
7	d	1	11

Figure 8.2: Compression by sharing space: a state is stored partially inside one state, and partially in another one. State 3 is entirely stored in states 2 and 4, a half in state 2, and the other half in state 4. Field *c* is the number of transitions in the target state. It is assumed that states 5--8 have a single outgoing transition.

Figure 8.2 shows an example of another variant of the same technique. This time a state is also a list of transitions, but instead of having a marker for the last transition of a state, we have a counter (c) for the number of transitions of the target state. While this representation could be used in the previous example, the figure shows another case that is impossible to implement with the previous representation. The first transition of state 3 is stored as the last transition of state 2, and the second transition of state 3 is stored as the first transition of state 4. Unfortunately, counters take much space in comparison to one bit flags, so this

representation brings less savings than the previous one.



	l	L	T	t
1	c			8
2	b			9
3	a			8
4	d	○		9
5	b			8
6	c			9
7	·		○	3

Figure 8.3: Compression by sharing space: transitions labeled with *a* and *d* are shared by states 1 and 2. State 1 starts at location 1 in the transition vector and ends in location 4. State 2 starts in location 5, and in location 7, it has a *tail flag* T raised and a pointer to location 3. It is assumed that state 3 starts in location 8, and state 4 starts in location 9.

When a last transition marker is used, another space-sharing technique can be used. When two states share a subset of outgoing transitions, and that subset does not constitute the entire set of outgoing transitions of one of the states, then it is possible to store the subset in one state, and provide only a pointer to it in the other one. Figure 8.3 shows an example. Transition labeled with *a* going to state 3, and transition labeled with *d* going to state 4 are common both to state 1 and 2. Those states have also transitions labeled with *b* and *c*, but their targets differ. State 1 is represented normally --- it is a sequence of transitions starting at location 1 in the transition vector, and ending in location 4, with flag L raised for the last transition. State 2 starts at location 5. Two subsequent slots are occupied by transitions labeled with *b* and *c*. The third slot (at location 7) has a *tail flag* (T) raised, with the target field pointing to location 3 in the transition vector. At that location inside state 1, there are two other transitions of state 2 that are also transitions of state 1. The T flag in the transition at location 4 indicates both the end of state 1 and the end of state 2.

Compression techniques described so far made possible sharing subsets of transitions belonging to single states. It is a restriction that can be broken. We have already seen that an automaton is represented as a sequence of transitions. The transitions can be treated as symbols in a large alphabet. Then the sequence of transitions would become text. Such text can be compressed using text compression techniques. One of the most widely used is the LZW compression, and a variant of it can also be used for compressing automata.

The LZ-style compression in automata consists in replacing repeated sequences of transitions with pointers to their first appearance in the transition vector. While the idea is simple and natural, and many researchers tried to follow it, there are several crucial issues to be resolved that decide whether the result is actually a success. The first decision to make is what is the starting point. It turns out that it is better to start with a trie than with the minimal automaton. This looks surprising at first glance, but this compression method automatically includes minimization (minimization replaces certain repeated structures with pointer to their single representative), and minimization makes some patterns of transitions less regular, which is an obstacle.

The second decision is what sequences can be replaced. It is obvious that the sequence

should be longer than a pointer to it. The sequence does not have to form complete states, but there are usually constraints that must be obeyed. In the state-of-the-art solution in [33], incoming transitions for states in the replaced sequence must be located later in the transition vector than the sequence. Outgoing transitions can only point to a location inside the replaced sequence. Note that when we use the last transition (L) flag, and store transitions with subsequent letters of words in subsequent locations, we do not have to store transition addresses there.

The third decision is how to look for repeated sequences. Naive solutions lead to quadratic algorithms, which can be problematic given the size of large dictionaries. Suffix trees [23] provide required speed, but they require huge memory. Suffix arrays take less memory while providing the same construction and search speed. Both suffix trees and suffix arrays can be built in linear time, and searching for a word in them takes time proportional to the length of the word. The search for replacements should be optimized so that the greatest gain is obtained. It should be noted that performing a replacement may change the transitions vector so that some candidates for replacements stored in a suffix tree or suffix array are no longer available. Therefore for the best results, the structure should be updated.

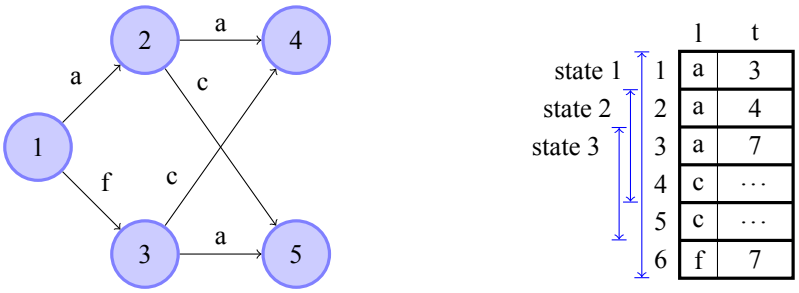


Figure 8.4: Compression by sharing space: State 1 is represented by transitions 1 to 6. However, only locations 1 and 6 are actually occupied. The empty space is taken by two other states: state 2 and state 3. State 2 starts at location 2 and ends in location 4. Location 3 is not filled by any transition of state 2, so state 3 starts there. We assume that state 4 has a transition labeled with *a*, and we make no assumption about state 5.

Another form of sharing space is shown in Figure 8.4. It is possible only when transitions have all the same length, so it is incompatible with some other compression techniques. A state is represented as a full vector of transitions with space for every symbol of the alphabet. Therefore, the state starts a the location destined for a transition with the first symbol of the alphabet. As in natural language dictionaries typical states have few outgoing transitions, most slots in a vector for a single state are not taken by its transitions. That space is reused by other states. Starting locations of different states must be different. When a word is being recognized, a transition is accessed directly by its symbol number, i.e. the zero-based symbol number must be added to the starting location of the state. As the label is determined by the location, it seems at the first glance that the label does not have to be stored. However, the label is used to verify that a given transition belongs to a given state. For example, the transition at location 3 does not belong to state 2, because state 2 begins at location 2, location 3 would hold symbol number 1 (*b*). Location 3 belongs to state 3, as state 3 starts at that location, and

a is symbol number 0. In the figure, it is assumed that state 4 has a transition labeled with *a*. Should its transitions start with a transition labeled with *b*, the state would start at location 6 instead of 7.

That compression technique gives fastest recognition times as we go directly to the right transition or see that it does not exist. By contrast, it is slow in tasks when we explore all transitions of a state, e.g. in spelling correction or listing previous/next word.

8.2 Compressing Fields

Programming languages have integer and Boolean types that can be used to express values of fields of transitions. It is natural that programmers use them for that purpose. However, such solution is not optimal. When all transitions have fixed length, then $\lceil(\log_2(n+1))/8\rceil$ bytes are needed to represent addresses of n transitions. When transitions do not have fixed length, then they can be addressed by specifying their starting byte number, and for n bytes occupied by the transition vector, $\lceil(\log_2(n+1))/8\rceil$ byte addresses would be needed. For typical natural language dictionaries, especially morphological dictionaries, addresses are usually three bytes long.

Stopping at a byte boundary is not always the best solution. One can go one step further and use as few bits as possible. When field boundaries are not bytes, this makes it possible to store more fields in one byte, or to use fewer bytes to store the same number of fields. The disadvantage is that unpacking bits is a bit slower than working at a byte level. This can be compensated by a smaller size of the automaton as a bigger part of the automaton fits into a cache.

For binary values, like certain flags (see the previous section and the beginning of the chapter), one bit is sufficient. One-bit flags can also be used to indicate special situations when a field value can be omitted. For example, in many cases the target state of a transition in a transition list representation of states would be placed directly after the current state. Computing its address is not necessary. One-bit flag *Next* (`N`) can be used to indicate whether the situation takes place.

When represented values are integers that can be more than one byte long, it is possible to use variable-length coding [6]. A value is split into 7-bit chunks accompanied with one-bit continuation flag that indicates whether the chunk is the last one. Smaller values are represented with fewer bytes. This simple technique is very efficient.

8.3 Using Frequency

Many text compression algorithms make extensive use of frequency of compressed items. This can also be applied to automata. For example, we have already seen that sorting transitions made possible sharing more transitions between states. This works for all forms of sharing space.

In the previous chapter, we introduced the *Next* (`N`) flag. If it is raised, then the target state is located right after the transition, and the target field can be omitted. This leads to large savings. The more often the flag is used, the more savings occur. It is possible to sort transitions of states and the states themselves so that the flag is used more often.

Transition labels occur with variable frequency. Actually, the frequency distribution follows Zipf's law. That property can be used for compression. A label in natural language dictionaries usually takes one byte. If UTF-8 is in use, it is possible to treat individual bytes of UTF-8 codes as labels. The most frequent labels can be stored in a most frequent label dictionary, and they can get shorter codes. For example, in the format described in [22], 31 most frequent labels are stored on 5 bytes; 0 indicates that the label is not frequent, and that it is stored directly afterwards. Three other bits in a byte are taken by finality (`F`), next (`N`), and last (`L`) flags. When the `N` flag is raised (which happens often), a transition is represented on a single byte! Of course, the most frequent labels dictionary must be stored with the automaton, and it also occupies some space. However, that space is negligible compared to the savings.

8.4 Summary

Many techniques described in this section are described in [26, 28] --- a primary reference for compression of automata. Interdigitation of states (the last technique described in this section, also known as *superimposed coding*) is described in [27]. It is based on *sparse matrix representation* described in [35]. The technique is also described in [31]. The LZ-style compression is described in [34], and an improved method in [33]. Finding sub-automata [37, 36] can be seen as a restricted variant of LZ-style compression, but finding sub-automata can also serve other purpose than mere compression. Variable-length coding in transition fields including recoding of the most frequent labels was introduced by Dawid Weiss [22]. Comparison of some of techniques can be found in [13], and [22] can be seen as a follow up to that paper, with new techniques. A section on compression techniques can be found in [20]. The author of the present book implemented most of those techniques -- see [22, 20, 14, 19]. He reinvented many of them.

Chapter 9

Summary

The author of this book spent years on developing tools and algorithms for natural language processing. Most of that work was focused on dictionaries. Throughout those years, questions were asked: ``Why are you doing that''? Technological progress in hardware leads to faster computers with larger memories. Does it make incremental algorithms, hashing, compression obsolete? Does it really matter if we use standard databases for dictionaries instead of some fancy compressed automata?

There are several reasons why this work still meats with interest in the natural language processing community. Computers are getting faster and faster, but it does not mean that we give up more advanced sorting algorithms in favor of bubble sort. Compiling an automaton could take almost the whole day fifteen years ago. It can take minutes nowadays, but the speed will not matter when the processing time is under a second. It is simply annoying to wait until some process finishes. Computers have bigger memories, but much part of them is already occupied by the ever-growing resource-hungry operating system, and by other processes (a browser, a multimedia player, a word processor etc.) that run at the same time so that the user does not fall asleep waiting for the results.

Desktop computers and laptops have very fast processors (but their processing power can fully be used in games and multimedia applications, NLP applications usually use a single core), and they have huge memories. However, the computers we use most frequently are not so powerful. They are getting faster and faster, and their memories grow, but mobile phones (smartphones) still cannot match desktop computers. And when they do, smart watches wait for their chance to capture our attention.

Faster, more powerful computers with bigger memories make possible tasks that we could not think of before. Those tasks require more processing power than it used to be available. We gather much more data than before. This is true also for dictionary data --- dictionaries grow with time. They contain more information about individual words, their syntax, semantics, frequency of use, context of use, etc. When the author developed a library for Alpino project, dictionaries in the system dropped their memory requirements by an order of magnitude. But it took Gertjan van Noord little time to reclaim that memory -- he simply used more data.

Modern NLP applications need algorithms and methods that are efficient. This book should provide them to developers. We did not make the book formal; the focus is on im-

plementation. We give many details that are often omitted from journal or conference papers. We hope that the readers will find this work useful.

Bibliography

[1] Aho A.V., Hopcroft J.E., Ullman J.D.: *The Design and Analysis of ComputerAlgorithms*. Addison-Wesley Publishing Company 1974.

[2] Almeida M., Moreira N., Reis R.: Incremental DFA minimization. In: *Proceedings of the 15th International Conference on Implementation and Applicationof Automata*, vol. 6482 of *Lecture Notes in Computer Science*. Berlin, Heidelberg: Springer-Verlag 2011, pp. 39–48.

[3] Almeida M., Moreira N., Reis R.: Incremental DFA minimization. *RAIRO – Theoretical Informatics and Applications*, 2014, 48(2):173–186.

[4] Aoe J., Morimoto K., Hase M.: An algorithm for compressing common suffixes used in trie structures. *Trans. IEICE*, 1992.

[5] Brzozowski J.A.: Canonical regular expressions and minimal state graphs for definite events. In: *Mathematical theory of Automata*. Polytechnic Press, Polytechnic Institute of Brooklyn, N.Y. 1962. Vol. 12 of MRI Symposia Series, pp. 529–561.

[6] Büttcher S., Clarke Ch.L.A., Gordon V.: Cormack. *Information Retrieval. Implementing and Evaluating Search Engines*. MIT Press 2010.

[7] Carrasco R., Daciuk J.: A perfect hashing incremental scheme for unrankedtrees using pseudo-minimal automata. *RAIRO – Theoretical Informatics and Applications*, 43(04):779–790, October 2009. DOI: http://dx.doi.org/10.1051/ita/2009018.

[8] Carrasco R.C., Daciuk J., Forcada M.L.: Incremental construction ofminimal tree automata. *Algorithmica*, 55(1):95–110, September 2009.

[9] Carrasco R.C., Forcada M.L.: Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(2), June 2002.

[10] Ciura M., Deorowicz S.: How to squeeze a lexicon. *Software – Practiceand Experience*, 2001, 31(11):1077–1090.

[11] Comon H., Dauchet M., Gilleron R., Jacquemard F., Lugier D., Tison S., Tommasi M.: Tree automata techniques and applications. Draft book, available at http://www.grappa.univ-lille3.fr/tata, September 2002.

[12] Daciuk J.: *Incremental Construction of Finite-State Automata and Transducers*, *and their Use in the Natural Language Processing*. PhD thesis, Technical University of Gdańsk, Faculty of Electronics, Telecommunications, and Informatics, 1998.

[13] Daciuk J.: Experiments with automata compression. In: Sheng Yu, A. P□un, editors, *Implementation ans Application of Automata. 5th International Conference CIAA 2000.* London, Ontario, July 2000. *Revised Papers*, number 2088 in LNCS. Springer 2001, pp. 105–112.

[14] Daciuk J.: Comparison of construction algorithms for minimal, acyclic, deterministic, finite-state automata from sets of strings. In: *Seventh International Conference on Implementation and Application of Automata CIAA 2002*, Tours, July 2002.

[15] Daciuk J.: Comments on incremental construction and maintenance of minimal finitestate automata by R.C. Carrasco, M. Forcada. *Computational Linguistics*, 30(2):227–235, June 2004.

[16] Daciuk J.: Semi-incremental addition of strings to a cyclic finite automaton. In: K. Trojanowski, M.A. Kłopotek, S.T. Wierzchoń, editor, *Proceedings of the International IIS*: *IIP WM '04 Conference*, Advances in Soft Computing, vol. 25, Springer Zakopane, 17- 20 May 2004, pp. 201–207.

[17] Daciuk J., Maurel D., Savary A.: Incremental and semi-incremental construction of pseudo-minimal automata. In: J. Farre, I. Litovsky, S. Schmitz, editors, *Implementation and Application of Automata*: *10th International Conference*, *CIAA 2005*, vol. 3845 of *LNCS*. Springer 2006, pp. 341–342.

[18] Daciuk J., Mihov S., Watson B., Watson R.: Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April 2000.

[19] Daciuk J., Piskorski J.: Gazetteer compression technique based on substructure recognition. In: *Intelligent Information Processing and Web Mining*, *Proceedings of the International Conference on Intelligent Information Systems.* Ustroń, 19–22 June 2006. Book Series Advances in Soft Computing, Vol. 35/2006, Berlin, Heidelberg: Springer 2006, pp. 87–95.

[20] Daciuk J., Piskorski J., Ristov S.: *Scientific Applications of Language Methods*, chapter Natural Language Dictionaries Implemented as Finite Automata. Imperial College Press 2010, pp. 133–204.

[21] Daciuk J., Watson R.E., Watson B.W.: Incremental construction of acyclic finite-state automata and transducers. In: K. Oflazer, L. Karttunen, editors, *Finite State Methods in Natural Language Processing*. Bilkent University, Ankara, June–July 1998.

[22] Daciuk J., Weiss D.: Smaller representation of finite-state automata. *Theoretical Computer Science*, 450:10–21, September 2013.

[23] Gusfield D.: *Algorithms on Strings*, *Trees*, *and Sequences*. Cambridge University Press 1997.

[24] Hopcroft J.E.: An $n \log n$ algorithm for minimizing the states in a finite automaton. In: Z. Kohavi, editor, *The Theory of Machines and Computations*. Academic Press 1971, pp. 189–196.

[25] Hopcroft J.E., Ullman J.D.: *Introduction to Automata Theory, Languages*, *and Computation*. Massachusets: Adison-Wesley Publishing Company, Reading 1979.

[26] Kowaltowski T., Lucchesi C.L., Stolfi J.: Minimization of binary automata. In: *First South American String Processing Workshop.* Belo Horizonte, 1993.

[27] Liang F.M.: *Word Hy-phen-a-tion by Comp-uter*. PhD thesis, Stanford University, 1983.

[28] Lucchiesi C., Kowaltowski T.: Applications of finite automata representing large vocabularies. *Software Practice and Experience*, 23(1):15–30 Jan. 1993.

[29] Maurel D., Daciuk J.: Les transducteurs a sorties variables. In: P. Mertens, C. Fairon, A. Dister, P. Watrin, editors, *Actes de la 13□me conferenceannuelle sur le traitement des langues naturelles*, vol. 1, Louvain, 10–13 April 2006. UCL Presses Universitaires de Louvain 2006, pp. 237–245.

[30] Mihov S., Maurel D.: Direct construction of minimal acyclic subsequential transducers. In: *Fifth International Conference on Implementation and Application of Automata CIAA 2000*, London, Ontario, July 2000, pp. 1–1.

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

[32] Revuz D.: Dynamic acyclic minimal automaton. In *CIAA 2000*, *Fifth International Conference on Implementation and Application of Automata.* London, Canada, July 2000, pp. 226–232.

[33] Ristov S., Korenčić D.: Fast construction of space-optimized recursive automaton. Submitted to Software: practice and Experience.

[34] Ristov S., Laporte E.: Ziv Lempel compression of huge natural language data tries using suffix arrays. *Journal of Discrete Algorithms*, 1999, pp. 241–256.

[35] Tarjan R.E., Chi-Chih Yao A.: Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.

[36] Tounsi L.: *Sous-automates □ nombre fini d'états. Application □ la compression de dictionnaires électroniques*. PhD thesis, Université Francois Rabelais Tours, 2008.

[37] Tounsi L., Bouchou B., Maurel D.: A compression method for natural language automata. In: *Proceedings of the 7th International Workshop on Finite-State Methods and Natural Language Processing*, Ispra, 2008, pp. 146–157.

[38] Watson B.: A fast new (semi-incremental) algorithm for the construction of minimal acyclic DFAs. In: *Third Workshop on Implementing Automata.* Rouen, September 1998. Lecture Notes in Computer Science, Springer, pp. 91–98.

[39] Watson B.: An incremental DFA minimization algorithm. In: L. Karttunen, K. Koskenniemi, G. van Noord, editors, *proceedings of FSMNLP 2001*, *ESSLLI workshop*, Helsinki, August 2001. Available at odur.let.rug.nl/~vannoord/alp/esslli fsmnlp/watson.pdf.

[40] Watson B.W.: *Constructing Minimal Acyclic Deterministic Finite Automata*. PhD thesis, University of Pretoria, 2010.

[41] Watson B.W., Daciuk J.: An efficient incremental DFA minimization algorithm. *Natural Language Engineering*, 9(1):49–64, March 2003.

Index

`F`
    flag, 162
`L`
    flag, 242--245, 248
`N`
    flag, 247, 248
`T`
    flag, 244

accepting state, 12
acyclic automaton, 11, 17, 18, 27, 51, 195
Alfa Informatica, 8
Almeida, Marco, 8, 237
alphabet, **9**, 12
    output
        of a state, 10
automaton
    acyclic, 11, 17, 18, 27, 51, 195
    complete, 9, 242
    cyclic, 11, 51--66
    deterministic, 5, 9, 17, 18, 51
    incomplete, 9
    minimal, 5, 10, 11, 17, 18, 20, 27, 30, 32, 46, 51, 66
    pseudo-minimal, 11, 14, 41, 85, 188, 204

bottom-up tree automaton, 12
Boullion, Pierrete, 7

Carrasco
    Rafael, 128
Carrasco, Rafael, 8
cloning, 51, **51**, 54--56, 67, 109, 162
co-reachable state, 11, 15, 40, 95, 125, 127
complete automaton, 9, 242
confluence state, 11, 38, 45, 46, 51, **51**, 52, 62, 67, 69, 70, 72, 76, 84, 96, 148, 172, 183
construction
    semi-incremental, 147
cyclic automaton, 11, 51--66

deterministic automaton, 5, 9, 17, 18, 51
deterministic FSA, 9
DFA, 9
divergent state, 11, 14, 41, 46, 86, 89, 93, 155, 160
DTA
    pseudo-minimal, 14
dynamic perfect hashing, 204

equivalent state, 6, 10, 11, 20, 23, 24, 27, 30--32, 35, 36, 38, 40, 42, 44, 51, 52, 55, 62, 66

fin, 11
final state, 12
finality flag, 162
flag, 40, 84, 96, 107, 221, 242, 243, 247
    finality, 162
    last, 242--245, 248
    next, 247, 248
    tail, 244
flag `F`, 162
flag `L`, 242--245, 248
flag `N`, 247, 248
flag `T`, 244
Forcada
    Mikel, 128
Forcada, Mikel, 8
frontier-to-root tree automaton, 12

FSA
    deterministic, 9

Goczyła, Krzysztof, 8

hash value, 195
hashing
    perfect, 195
height, 11

incomplete automaton, 9
initial state, 9, 11, 18, 31--33, 51, 52, 54--56, 58, 60, 62, 67, 68
Intex, 7
ISSCO, 7

Karttunen, Lauri, 7

language of an automaton, **10**
last flag, 242--245, 248
left language, **10**
Lehmann Sabine, 7
local minimization, 27, 68

Maurel
    Denis, 46, 96, 162, 207
Maurel, Denis, 8
minimal automaton, 5, **10**, 11, 17, 18, 20, 27, 30, 32, 46, 51, 66
minimal perfect
    hashing, 195
minimal tree automaton, 13
minimization
    local, 27, 68
Miłkowski, Marcin, 8
Moreira, Nelma, 8
Mrozowski, Michał, 8

next flag, 247, 248

output alphabet of a state, **10**

path, **9**
perfect
    hashing
        minimal, 195
perfect hashing, 195
    dynamic, 204
Petitpierre, Dominique, 7
postorder, **23**, 40
proper transition, 14, 204, 221
pseudo-equivalent state, 11
pseudo-minimal automaton, 11, 14, 41, 85, 188, 204
pseudo-minimal DTA, 14

reachable state, 11, 15, 40, 55, 95, 125
recognizer, **10**
register, **23**
Reis, Rogério, 8
representation
    sparse matrix, 248
Revuz, Dominique, 85
right language, **10**
root-to-frontier tree automaton, 12

Savary
    Agata, 46, 96, 162
Savary, Agata, 8
semi-incremental construction, 147
signature, **11**, **14**, 67, 68, 84, 128
Silberztein, Max, 7
sparse matrix representation, 248
start state, 9, 11, 18, 31--33, 51, 52, 54--56, 60, 62, 67, 68
state
    accepting, 12
    co-reachable, 11, 15, 40, 95, 125, 127
    confluence, 11, 38, 45, 46, 51, 52, 62, 67, 69, 70, 72, 76, 84, 96, 148, 172, 183
    divergent, 11, 14, 41, 46, 86, 89, 93, 155, 160
    equivalent, 6, 10, 11, 20, 23, 24, 27, 30--32, 35, 36, 38, 40, 42, 44, 51, 52, 55, 62, 66
    final, 12
    initial, 9, 11, 18, 31--33, 51, 52, 54--56, 58, 60, 62, 67, 68
    output alphabet, 10
    pseudo-equivalent, 11
    reachable, 11, 15, 40, 55, 95, 125
    start, 9, 11, 18, 31--33, 51, 52, 54--56, 60, 62, 67, 68

unreachable, 11, 15, 127
useless, 40
string, **9**
superimposed coding, 242
Szejko, Stanisław, 7

tail flag, 244
top-down tree automaton, 12
transducer, **10**
transition
proper, 14, 204, 221
transition diagram, **11**
transition function, **9**
translation vector, 208
tree automaton
bottom-up, 12
frontier-to-root, 12
minimal, 13
root-to-frontier, 12
top-down, 12
trie, **18**, 148

unreachable state, 11, 15, 127
useless state, 40

van Noord, Gertjan, 8, 250
Vetulani, Zygmunt, 7

Watson, Bruce, 7, 85, 147, 155, 225, 226
Watson, Richard, 85
Weiss, Dawid, 8, 248

WYDAWNICTWO POLITECHNIKI GDAŃSKIEJ

Wydanie I. Ark. wyd. 12,3, ark. druku 14,25, 151/814

Druk i oprawa: *EXPOL* P. Rybiński, J. Dąbek, Sp. Jawna
ul. Brzeska 4, 87-800 Włocławek, tel. 54 232 37 23