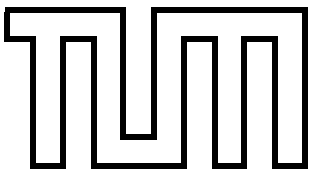


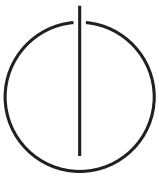
FAKULTÄT FÜR INFORMATIK

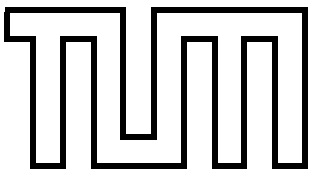
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Master Z-Automata for Regular Languages

Tobias Klenze





FAKULTÄT FÜR INFORMATIK

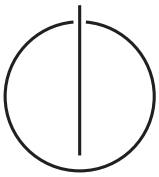
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelorarbeit in Informatik

Master Z-Automata for Regular Languages

Master Z-Automaten für Reguläre Sprachen

Author:	Tobias Klenze
Supervisor:	Prof. Dr. Javier Esparza
Advisor:	Prof. Dr. Javier Esparza
Date:	September 15, 2012



I assure the single handed composition of this bachelor thesis only supported by declared resources

München, 10. September 2012 Tobias Klenze

Acknowledgments

I would sincerely like to thank my supervisor and advisor Prof. Dr. Javier Esparza for his continued effort and time, for his supportive and friendly manner and for choosing the excellent subject matter of this bachelor thesis. Not only did he come up with the main idea of this work, it was also his research on fixed-length languages that it is founded on.

I would also like to express my gratitude to my brother Philipp, who – although not a computer scientist – has a keen sense and deep understanding in automata theory and whose comments helped to make the work more understandable. Lastly, I thank Wiebke Köpp, who helped me with her experience of writing a bachelor thesis.

Abstract

Modern applications of automata theory require handling of huge finite-state machines. While improvement of the algorithms of operations on automata may help to cope with them, this work discusses another approach: two optimizations are presented in conjunction that reduce the overall number of states and transition, thus reducing the complexity of large automata. The ideas, which originate in the field of Binary Decision Diagrams are applied to the more general case of regular languages. Algorithms for some common automata operations are presented for these optimizations and the differences to their non-optimized counterparts are discussed.

Zusammenfassung

Moderne Anwendungen der Automatentheorie machen die Handhabung riesiger endlicher Automaten notwendig. Zwar können verbesserte Algorithmen für Operationen auf Automaten dazu beitragen, dies zu bewältigen, diese Arbeit wird jedoch einen anderen Ansatz diskutieren: Zwei Optimierungen, die die Zahl der Zustände und Transitionen und damit die Komplexität von Automaton verringern, werden in Verbindung miteinander vorgestellt. Die Ideen, die aus dem Bereich der binären Entscheidungsdiagramme stammen, werden auf den allgemeineren Fall regulärer Sprachen angewandt. Algorithmen für einige häufige Operationen auf Automaten werden für diese Optimierungen vorgestellt und die Unterschiede zu ihren nicht-optimierten Pendants diskutiert.

Contents

Acknowledgements **7**

Abstract **9**

1. Introduction and Motivation **13**
1.1. Scope of this work 16

2. Definitions and Proofs **19**
2.1. Master Automaton for Regular Languages 19
2.2. Z-Automaton 21
2.3. Master Z-Automaton for Regular Languages 24

3. Master Automata for Regular Languages **27**
3.1. Projection 27
3.2. Minimization 27

4. Operations on Z-Automata **31**
4.1. Membership 31
4.2. BinOp 31
4.3. Complement 35
4.4. Minimization 37
4.4.1. Indirect minimization 38
4.4.2. Approach one: No expansion of z-transitions with arbitrary zip splits 38
4.4.3. Approach two: Partial expansion of z-transitions with arbitrary splits 40
4.4.4. Approach three: Partial expansion of z-transitions with single-letter-splits 44
4.4.5. Conclusions 49
4.5. Merging 50

5. Conclusions **55**

Glossary **57**

Bibliography **59**

A. Alternative definition of Z-Automata **61**
A.1. Operations 64

A.2. Evaluation 64

B. User manual for JFLAP-Z **67**
B.1. Adding and manipulating z-automata 68
B.2. Complement 68
B.3. Binary operation 68
B.4. Minimization 69
B.5. Merging 69
B.6. Projection 69

1. Introduction and Motivation

Finite automata are a common concept in computer science. They have been used to develop parsers, compilers and numerous other components of software systems since the late 1960s. In the 1980s, formal verification became on of the major applications of automata theory.[1] In verification, one needs to evaluate and handle large boolean functions. Binary Decision Diagrams (BDDs) are a data structure for such functions (see *Bryant 1992*). As we will see, they can be also expressed as automata.

A simple and common way of storing small boolean functions are truth tables. However, they grow exponentially with the number of variables, while BDDs are much more compact. BDDs also have advantages over another alternative representation: While it is convenient to express functions as formulas of propositional logic, formulas are hard to check for satisfiability. One can avoid this problem by enforcing a normal form, but then combining formulas with respect to at least some boolean operations is hard. With BDDs – more specifically, reduced and ordered BDDs – it is possible to represent boolean functions compactly while letting important operations (such as checking for satisfiability and combining functions) be feasible.[2] We now present the concept of BDDs and reduced BDDs (ordered BDDs will not be discussed in this work).

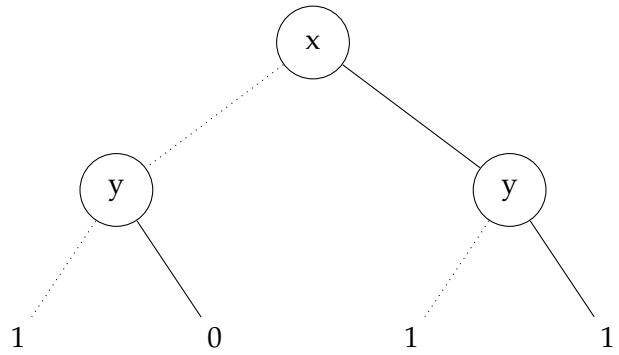


Figure 1.1.: Simple BDD.

Figure 1.1 is an example of a small BDD. At the bottom there are nodes labeled 0 and 1 (the "terminal nodes"), which represent the return values under the chosen truth value assignments. All other nodes (x and y) stand for the decision of the truth value of a variable. The solid line represents "the variable is true", the dotted line "the variable is false". So for example, in Figure 1.1, only when x is false and y is true, the

[1] *Esparza 2012*: Notes on his upcoming book *Automata Theory: An Algorithmic Approach*. See `http://www7.in.tum.de/~esparza/automatanotes.html`

[2] See *Huth / Ryan 2004*, p. 361.

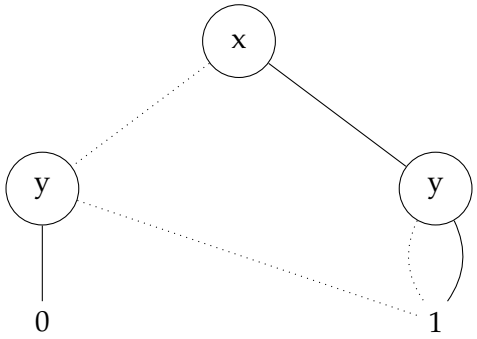


Figure 1.2.: Simple BDD with equivalent terminal nodes merged.

function returns 0. The BDD is somewhat redundant, since all the terminal nodes "1" are equal. Therefor a simple revision of merging all equivalent terminal nodes yields the BDD in Figure 1.2 which has only two terminal nodes. This might seem like a minor optimization, but it almost halves the number of states for large BDDs.

This BDD can be further reduced. Since the truth value of y does not affect the result of the binary function if x is true, it can be omitted in the right branch of the graph. The resulting "reduced" BDD is displayed in Figure 1.3.

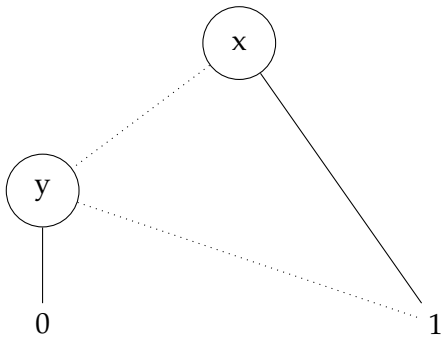


Figure 1.3.: Reduced BDD.

We can also interpret the BDDs as automata (Deterministic finite automaton, DFA) on fixed-length languages. The automaton accepts a word $w = b_1 b_2 \dots b_n \in \{0,1\}^n$ iff the function returns true for the corresponding variable assignments, i.e. iff $f(b_1, \dots, b_n) = 1$.[3]

Figure 1.4 shows the automaton corresponding to the BDD in Figure 1.2. The automaton accepts only three words: $01, 10, 11$, since $f(0,1) = f(1,0) = f(1,1) = 1$ and for all other parameters, f is 0.

As with BDDs, it is possible to reduce the automaton. The resulting automaton should accept the same language, thus the transitions have to be modified to read not only a single letter, but a regular expression $a \cdot \Sigma^i$ for some $a \in \Sigma, i \in \mathbb{N}$ (where Σ is the alphabet). Figure 1.5 displays the reduced automaton with a "$1 \cdot \Sigma$" transition, which

[3] See *Esparza 2012*, Section 6.6

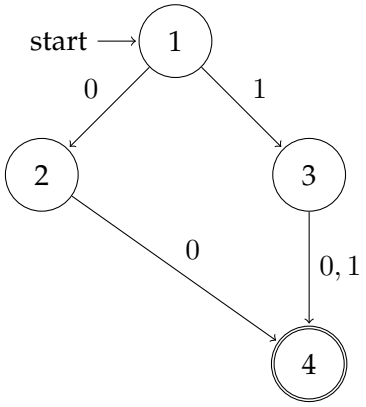


Figure 1.4.: Finite automaton corresponding to the BDD in Figure 1.2 (without the trap state).

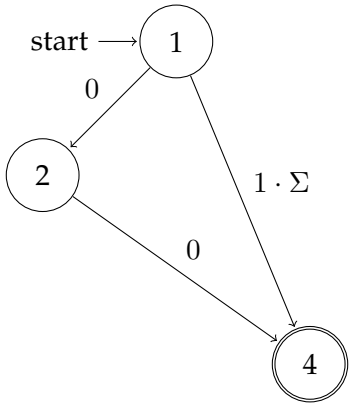


Figure 1.5.: Reduced automaton recognizing the same language as the DFA in Figure 1.4.

stands for "read '1' and a single letter". We call a transition of the form "$a \cdot \Sigma^i$" a "zip" (since $a \cdot \Sigma\Sigma \ldots \Sigma\Sigma$ looks like a zip). In addition, automata with such "z-transitions" are called "z-automata".

With this reduced automaton, we get a compact representation of boolean functions – or, more generally, of fixed-length languages. In traditional automata, one has to have one automaton per referenced language. When there are many automata, it is likely that they could share some of their structure. For example, the automaton recognizing the language $\{a, b\}$ is contained in the automaton for $\{aa, ab, ba\}$. It is possible to build a so-called "Master Automaton"[4], which incorporates all fixed-length languages under an alphabet. In the master automaton, states are languages and the automaton starting from a state L recognizes L. In practice, one always has a fragment of the master automaton and multiple initial states which stand for the languages one wishes to represent. Figure 1.6 is a fragment of the master automaton for $\Sigma = \{a, b\}$ (which will be the default alphabet for most of this work) with pointers to two initial states.

[4]See *Esparza 2012*, Section 6.1

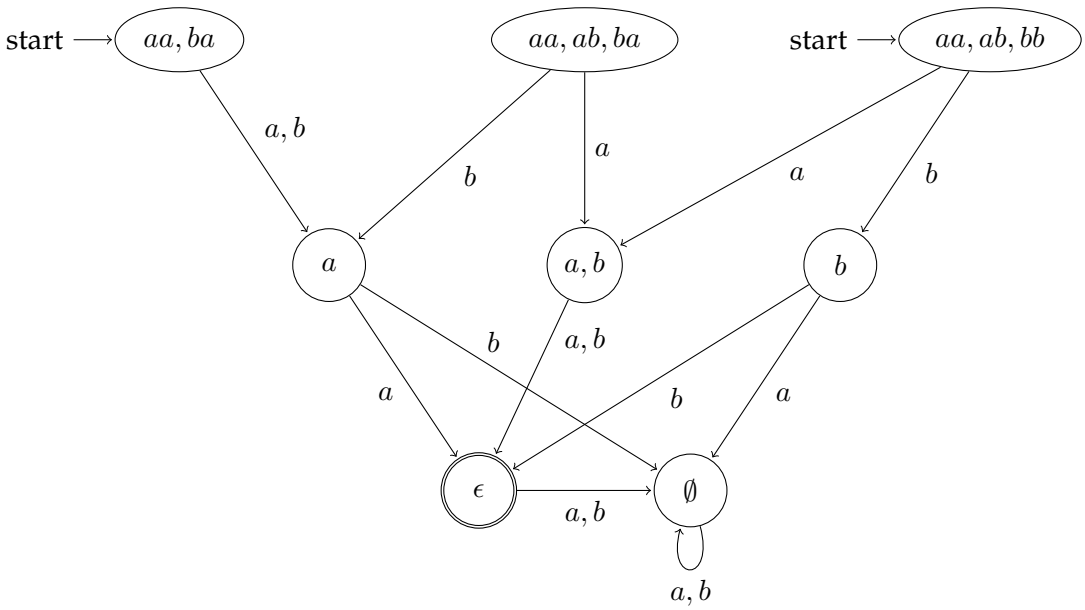


Figure 1.6.: Fragment of the master automaton for fixed-length languages under $\Sigma = \{a, b\}$.

To get the automaton that recognizes $L = \{aa, ba\}$ from the master automaton, one simply sets L as the unique initial state and removes all states not reachable from L (this is called *Projection*). It can be shown that this yields the unique minimal DFA recognizing L.

1.1. Scope of this work

All of the above has been described in *Esparza 2012*. The algorithms and proofs for the fixed-length case will not be presented here. Instead, this work extends the concepts of z-automata and master automata to regular languages. As we will see, while workable in principle, the algorithms are a bit more complex than for fixed-length languages.

While we don't have a specific problem or group of problems in mind, it might be that there applications where z-automata are specifically useful, due to the large number of states that can be reduced. Since we are not aware of such problems, we won't assume that reducible states are exceptionally frequent.

Instead, the general benefits and detriments of z-automata to normal deterministic automata are discussed.

In this work, we define master automata for regular languages ("rMA") and then apply a slightly modified version of the z-optimization[5] on DFAs (yielding the concept of a

[5]The term "z-optimization" refers to "z-automata", i.e. automata with "z-transitions", that may read more than just one letter.

"zDFA"[6]) and master automata for regular languages ("rMZ"). Our goal is to show this is feasible and that certain properties that hold for z-automata in fixed-length languages also hold in the more general case, such as existence of a unique minimal z-automaton for regular languages. We also examine some common algorithms on deterministic z-automata, such as their complement and binary conjunction.

As we have seen, z-automata are deterministic finite automata with a special type of transition. While traditional transitions read a particular letter of an alphabet Σ, z-transitions read a particular letter plus a specific number of letters from Σ. This is expressed as $a \cdot \Sigma^i$, which stands for a transition where first the letter a is read and then i letters from Σ. Z-automata are able to express some languages more efficiently than traditional automata, namely those which contain many states with transitions to states under the whole alphabet. This affects both storage space and time complexity when executing certain operations on the automaton. For some automata, the z-optimization may have little benefits (i.e. they may have many transitions of the form $a \cdot \Sigma^0$ – which indicates only few states could be reduced).

Beside z-automata, we discuss master automata, which – as we have seen – are used to efficiently store multiple languages in a single automaton. A master automaton is a single structure that contains multiple initial states. Thus, it can represent any number of languages and since it is always kept minimal, there is no redundancy in the data structure.

We extend the idea of master automata to the more general case of regular languages. In contrast to fixed-length languages they are a lot more diverse, so the saving effects of this optimization might not carry the same importance. Nevertheless, when there is a much redundancy, master automata might be a worthwhile consideration.

Our paper will discuss both optimizations – z-automata and master automata – in conjunction. It will be shown that a major efficiency problem of the z-optimization (which is minimization) does not obstruct the master automaton optimization.

The first part will be concerned with formal definitions and proofs. Among the latter is the proof for existence of a unique minimal zDFA. The second part shortly discusses some details on the implementation as a data structure. Next, different algorithms and approaches to common problems are presented. The problem of minimization is discussed in great detail, because of the difficulties z-transitions introduce to the traditional approach of minimization via a language partition. It should be noted, that though the algorithms have been tested with a number of automata, this paper does not include formal proofs of their correctness. The last part will be a conclusion on zDFAs and master automata.

Although we are concerned with an optimization, this paper does not include performance tests of DFAs and zDFAs. Neither are there calculations of the complexities of the individual algorithms – it is difficult to estimate them, since they strongly depend on how many states can be reduced. However, the benefits and detriments of the optimizations will be discussed.

There are multiple ways of applying the z-optimization on finite state machines. Ap-

[6]We only examine deterministic machines, therefor the terms "z-automaton" and "zDFA" will be used interchangeably, although technically "z-automaton" could also refer to non-deterministic machines.

pendix A discusses an alternative approach that allows for the reduction of even more states than the approach presented in this paper. However, as will be discussed, this alternative definition of a z-automaton on regular languages introduces some problems.

This paper comes with the proof-of-concept software JFLAP-Z, a modified version of the automaton tool JFLAP.[7] JFLAP-Z includes support for zDFAs and master automata and most of the algorithms discussed in this paper. A user manual for JFLAP-Z can be found in Appendix B.

This paper has few citations and references. Though it builds on the widely studied BDDs, the concepts of z-automata and master automata for regular languages is (to the best of our knowledge) new.

For those without a strong background in automata theory, the glossary explains a lot of the terms used in this paper.

[7] See `http://www.jflap.org` for more information on JFLAP.

2. Definitions and Proofs

2.1. Master Automaton for Regular Languages

We begin by extending the idea of a master automaton for fixed-length languages to a master automaton for all regular languages (abbreviated as rMA).

Definition 1. *The regular master automaton (rMA) over the alphabet Σ is the tuple $M = (Q_M, \Sigma, \delta_M, F_M)$, where*

- *Q_M is the set of all regular languages over Σ;*
- *$\delta_M : Q_M \times \Sigma \to Q_M$ is given by $\delta_M(L, a) = L^a$ for every $L \in Q_M$ and $a \in \Sigma$;*[1]
- *F_M, the set of final state states, is the set of all regular languages over Σ containing the empty language ϵ.*

This definition is almost similar to that of a deterministic finite automaton (DFA). However, the set of regular languages is infinite and we explicitly name the states by the language they recognize – which differs from a simple enumeration of the states. Also, there is not a distinct initial state. Instead, there are pointers to states, which each stand for a single DFA. Figure 2.1 is a fragment of the master automaton for regular languages under $\Sigma = \{a, b\}$.

This master automaton can be projected to a language L, which yields a DFA A_L recognizing the language L and having L as initial state. The construction is as follows (see Figure 2.2, p. 21, for an example): Take a language L. Then construct a DFA $A_L = (Q_L, \Sigma, \delta_L, q_{0L}, F_L)$, where Q_L is the set of states reachable from the state $L \in Q_M$ in the master automaton; q_{0L} is the state L; δ_L is the projection of δ_M onto Q_L; and $F_L = F_M \cap Q_L$.[2] As we will show, this DFA recognizes L and is also minimal. Since languages in the rMA correspond to automata, we will use terms like *recognize* and *accept* directly on languages in the rMA.

So, it is very easy to derive the DFA A_L given a language L in the master automaton. We now show that A_L corresponds to L, because it recognizes L.

Proposition 1. *Let L be a regular language. The language recognized from the state $L \in Q_M$ of the master automaton is L.*

Proof. We first proof the following, by induction on n: For all L, for all words w of length n: $w \in L \Leftrightarrow L$ accepts w.

[1] L^w is the residual of a language, that is the language resulting from cutting off w from all words in L. Or, more formally: $L^w = \{u \in \Sigma^* | wu \in L\}$

[2] This definition is taken in part from *Esparza 2012*, p. 112.

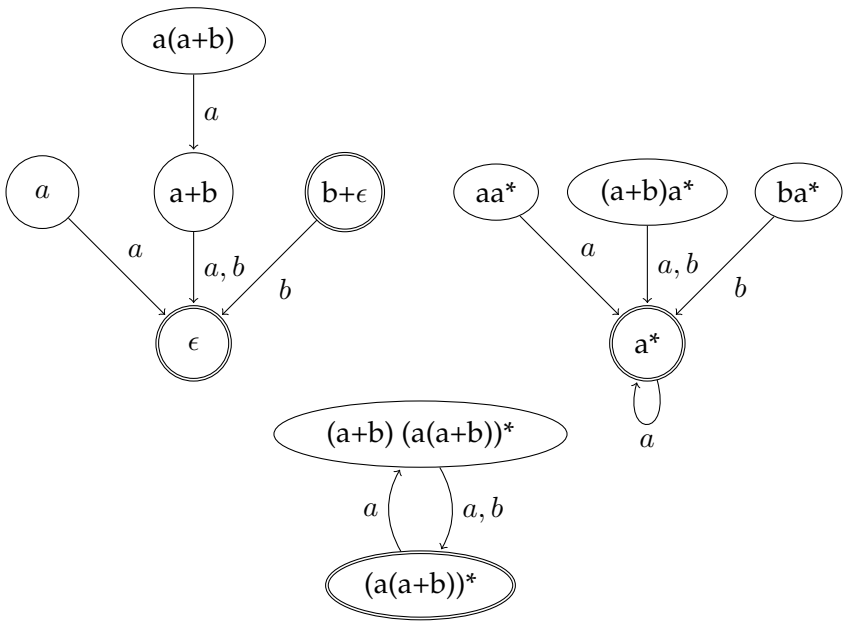


Figure 2.1.: Example excerpt of the rMA. Regular expressions labels exist only for convenience and are not stored in the actual data structure.

If $n = 0$, then $w = \epsilon$. State L accepts w iff $L \in F_L$, which is the case iff $w \in L$.

For $n > 0$ we observe that the successors of the state L are the languages L^a for every $a \in \Sigma$. Therefore, for every a, L accepts aw iff L^a accepts w. By definition of the residual of a language, $aw \in L \Leftrightarrow w \in L^a$. Also, by induction hypothesis, L^a accepts w iff $w \in L^a$, so L accepts aw iff $aw \in L$.

So, for all L, w: $w \in L \Leftrightarrow L$ accepts w. But this means that every state L recognizes the language L. □

It is not only the case that A_L recognizes L, but also that it is the unique minimal DFA recognizing L. This is important, since it means there are no redundant states in the master automaton.

Proposition 2. [3] *For every regular language L, the automaton A_L is the minimal DFA recognizing L.*

Proof. This proof is identical to that for fixed-length languages. By definition, distinct states of the master automaton are distinct languages. By Proposition 1, distinct states of A_L recognize different languages. Since a DFA is minimal if and only if distinct states recognized different languages (see Corollary 3.8 of *Esparza 2012*) A_L is minimal. □

[3] The proposition and proof are similar to *Esparza 2012*, p. 113

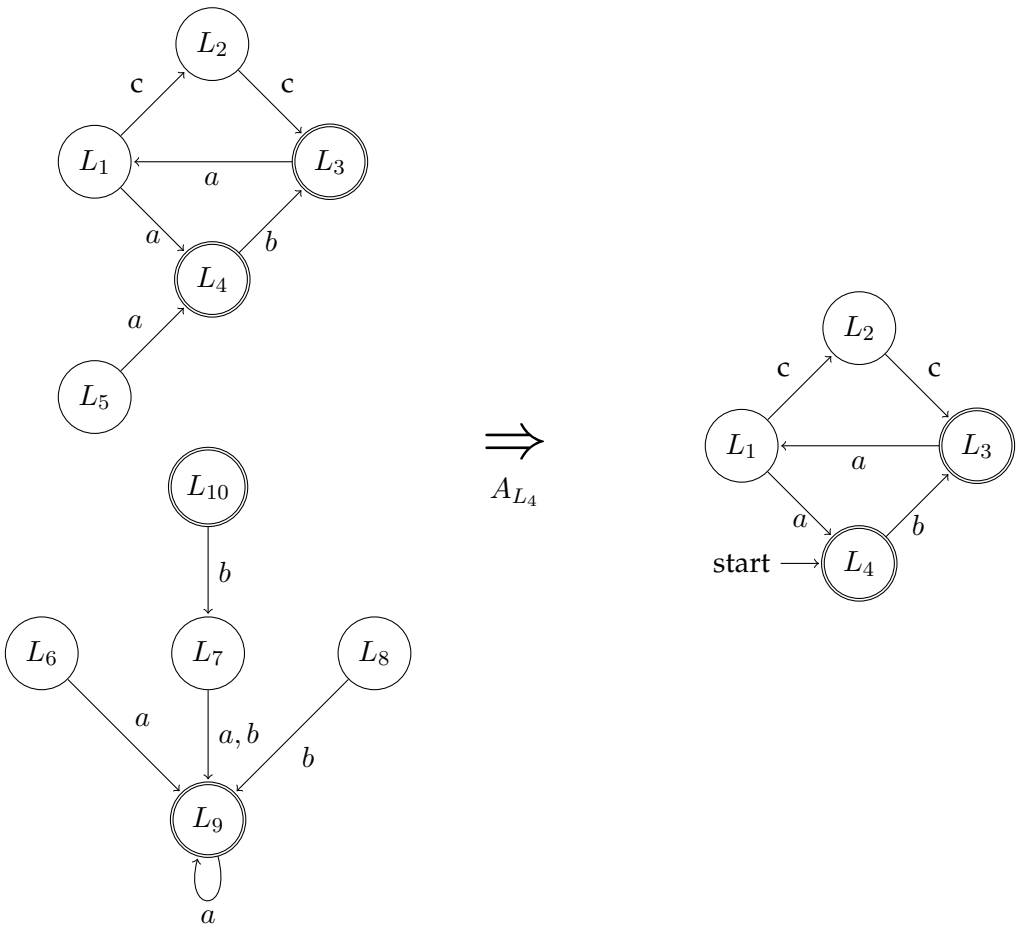


Figure 2.2.: On the left: Excerpt of the rMA. On the right: A_{L_4}

2.2. Z-Automaton

We now introduce the concept of z-automata for regular languages, which can be used to optimize the data structure of automata. We define the "zip" of length n over an alphabet Σ as the regular expression $a \cdot \Sigma^n$ for all $a \in \Sigma$.[4] The set of all zips is denoted by $Z(\Sigma)$.[5] A transition in a z-automaton reads a single letter plus a zip of a certain length, i.e. a specific number of letters from Σ. Figure 2.3 shows a simple zDFA.

Definition 2. *A deterministic z-automaton (zDFA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is the alphabet, δ is the transition function, q_0 is the initial state and F is a set of final (or: accepting) states. Let $\delta : Q \times Z(\Sigma) \to Q$ be so that for every $a \in \Sigma$ and $q \in Q$ there is at exactly one $k \in \mathbb{N}$ and $q' \in Q$ such that $(q, a \cdot \Sigma^k, q') \in \delta$.*
We define a function $\gamma : Q \times \Sigma \to \mathbb{N}$ where $\gamma(q, a) = k$ iff $(q, a \cdot \Sigma^k, q') \in \delta$ for some $q' \in Q$.

[4]Note that in this definition the initial letter (e.g. a) does not count towards the zip length. E.g. $a \cdot \Sigma^2$ has zip length of 2, not 3.
[5]This definition and the following definitions are similar to *Esparza 2012*, p. 128ff

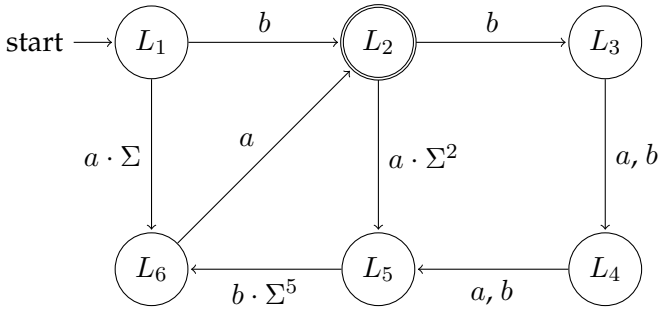


Figure 2.3.: Example zDFA

The main difference in the definition of a zDFA to that of a DFA is the transition function δ, which instead of a single letter a has a regular expression of the form $a \cdot \Sigma^i$ as parameter. The function $\gamma(q, a)$ is used to extract the zip length i of such a transition (or: *z-transition*).

With the exception of the trap state,[6] there is a unique transition from each (q, a) pair which leads to a state recognizing a non-empty language. Thus, zDFAs are deterministic machines, where the membership algorithm has only to take one path through the automaton.

However, zDFAs as defined above might not be fully reduced. This means that there are states which could be eliminated by changing certain transitions. For example, in Figure 2.3 L_3 could be reduced by replacing its incoming and outgoing transition by a single transition $(L_2, b \cdot \Sigma, L_4)$. And even that yields an automaton that can be further reduced.

Furthermore, a zDFA may contain multiple states that recognize the same language.

To reference z-automata that are fully reduced, we introduce the concept of kernels. In automata where states are kernels, the states *are* the languages that they recognize (i.e. that an automaton starting from a state recognizes). This also implies that each language can only be represented by at most one state. As we will see, automata with kernels as states are minimal and fully reduced.

Definition 3. *Let $L \subseteq \Sigma^*$ be a nonempty, regular language and let k be the largest number such that $L = \Sigma^k \cdot K$ for some regular $K \subseteq \Sigma^*$, or $k = 0$ if no such number exists.*

The language K is the kernel of L, denoted by $K = ker(L)$. If $ker(L) = L$, then L is a kernel.

In Figure 2.3, the languages L_1, L_2, L_5, L_6 are kernels, whereas L_3 and L_4 are not, since $L_3 = \Sigma^2 \cdot L_5$ and $L_4 = \Sigma \cdot L_5$.

In order to get an automaton which only contains kernels, the **reduction rule** (see Figure 2.4) is applied. It will – if exhaustively applied – turn any DFA into a zDFA with kernels as states (which we will call the *reduced zDFA*). So, a zDFA is fully reduced if and only if all of its states are kernels. The reverse operation, which turns a zDFA into a DFA will be called *expanding* the z-transitions. It is important to note that *reduced*

[6] The trap state is the unique state that is both non-final and has no transitions to other states.

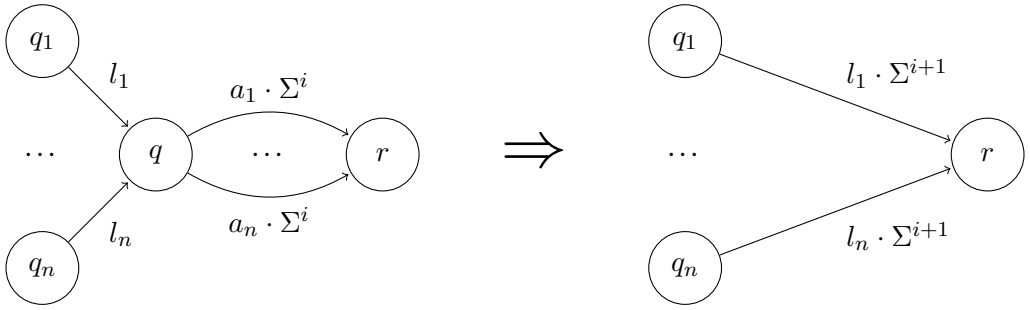


Figure 2.4.: Reduction rule, where $\Sigma = \{a_1, \dots, a_n\}$ and all l_k are of the form $a \cdot \Sigma^j$ ($a \in \Sigma$, $i, j \in \mathbb{N}$). State q must not be final.

and *minimized* are two distinct properties. The first refers to an automaton where the reduction rule has been applied, whereas the second specifies that there is no automaton with fewer states recognizing the same language. Therefore, if an automaton is minimal, it is also reduced.

The initial state of a zDFA must correspond to a language that is a kernel, since it cannot be reduced. Since in practice one also has to reference languages that are not kernels, one can specify a number that is the prefix zip length in conjunction with an automaton recognizing the kernel of the referenced language. E.g. $(3, A)$ refers to $\{w_1 w_2 | w_1 \in \Sigma^3, A \textit{ accepts } w_2\}$.

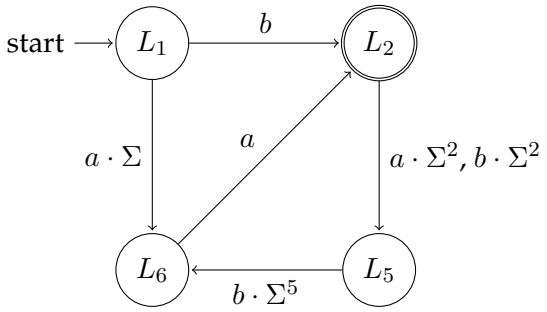


Figure 2.5.: zDFA of Figure 2.3 fully reduced.

If one exhaustively applies the reduction rule on the automaton in Figure 2.3, the non-kernels L_3 and L_4 will be removed and their transitions be replaced by a single transition $(L_2, b \cdot \Sigma^2, L_5)$ (see Figure 2.5). One might ask, why the state L_2 cannot be removed, since it then has transitions $(L_2, x \cdot \Sigma^2, L_5)$ for all $x \in \Sigma$. The answer is, that final states of automata are always kernels[7] and thus cannot be reduced.

The idea of master automata for DFAs can also be applied on zDFAs:

[7]This follows immediately from the definition of kernels. Suppose a state recognizing L is final and not a kernel. So, L contains the empty word ϵ of length 0. However, it must be the case that $L = \Sigma^k \cdot K$ for some $k > 0$ (since $k = 0$ would mean that $L = K$ is its own kernel) and thus all words in L have a length > 0. Contradiction.

2.3. Master Z-Automaton for Regular Languages

Definition 4. *The master z-automaton for regular languages over Σ is the tuple rMZ $= (Q_M, \Sigma, \delta_M, F_M)$, where*

- *Q_M is the set of all kernels;*
- *$(K, a \cdot \Sigma^k, K') \in \delta_M$ iff $K' = ker(K^a)$ and $K^a = \Sigma^k \cdot K'$; and*
- *F_M is the set of all kernels that contain the empty word.*

This definition is a combination of the definitions for master automata and zDFAs. As mentioned, a z-automaton is minimal and reduced if and only if all of its states are distinct kernels. So it follows from the definition, that the master automaton is both minimal and fully reduced.

Since the rMZ contains the zDFAs for all kernels, it is possible to construct the zDFA $A_K = (Q_K, \Sigma, \delta_K, q_{0K}, F_K)$ for a given kernel K using the rMZ: Q_K is the set of states reachable from K, δ_K is the projection of δ_M onto Q_K, $q_{0K} = K$, and $F_K = Q_K \cap F_M$.

We now prove that the automata A_K derived from the rMZ are indeed minimal.

Proposition 3. [8] *(1) A zDFA A is minimal if and only if (a) every state of A recognizes a kernel, and (b) distinct states of A recognize distinct kernels.*

(2) A_K is the unique minimal zDFA recognizing a kernel K.

(3) The result exhaustively applying the reduction rule to the minimal DFA recognizing a language L is the minimal zDFA recognizing $ker(L)$.

Proof. (1 $\Rightarrow$ a) For (a), assume that the language L_q recognized from a state q of A is not a kernel. Then A has a transition $(q, a \cdot \Sigma^{k_a}, q_a)$ for every $a \in \Sigma$, and moreover $(L(q))^a = (L(q))^b$ for every $a, b \in \Sigma$. Since $(L(q))^a = \Sigma^{k_a} \cdot L(q_a)$ for every $a \in \Sigma$, we have $\Sigma^{k_a} \cdot L(q_a) = \Sigma^{k_b} \cdot L(q_b)$ for every $a, b \in \Sigma$. Let m be a letter of Σ such that k_m is minimal. Then $(L(q))^a = \Sigma^{k_m} \cdot L(q_m)$ for every $a \in \Sigma$, and so $L(q) = \Sigma^{k_m+1} \cdot L(q_m)$. Consider now the zDFA A' obtained from A by applying the reduction rule at the beginning of the section. A' and A recognize the same language, and so A is not minimal.

(1 $\Rightarrow$ b) For (b), observe that the quotienting operation can be defined as for DFAs; if two distinct states recognize the same kernel then the quotient with respect to the language partition has fewer states than A, and so A is not minimal.

(1 $\Leftarrow$ a,b) Let K be the language recognized by A. We prove that any zDFA A' recognizing K and satisfying (a) and (b) is isomorphic to A. As we have just shown, a zDFA must satisfy (a) and (b) if it is minimal, thus the isomorphism implies that A is minimal.

First note, that A and A' are both fully reduced, since otherwise there exists a state q that has transitions to a state r under $x \cdot \Sigma^i$ for some i and for every $x \in \Sigma$. But then q is equal to $\Sigma^{i+1} \cdot r$ and thus it is not a kernel, which contradicts (a).

A and A' have initial states q_K and r_K that recognize K. For all $a \in \Sigma$, A has $(q_K, a \cdot \Sigma^i, q_{ker(K^a)})$ for some i as transition, and by analogy A' has $(r_K, a \cdot \Sigma^i, r_{ker(K^a)})$ for the same i as transition. Since A and A' satisfy (b), $q_{ker(K^a)}$ and $r_{ker(K^a)}$ are the only states

[8] The proposition and proofs (1 $\Rightarrow$ a,b), (2), (3) are similar to *Esparza 2012*, p. 130

recognizing $ker(K^a)$. So with respect to vertices reachable by one transition from the initial states, A and A' are isomorphic. However, the same reasoning can be applied recursively to the kernels of all successors of q_K and r_K. Thus A and A' are isomorphic with respect to the whole automaton.

(2) A_K recognizes K and it satisfies conditions (a) and (b) of part (1) by definition, and so it is a minimal zDFA. Uniqueness follows immediately from the proof of (1 $\Leftarrow$).

(3) Let A be the minimal DFA recognizing K. Then distinct states of A recognize distinct languages. We show that after exhaustively applying the reduction rule, every state of the resulting zDFA recognizes a kernel, which is then the minimal zDFA by (1). Assume that after exhaustively applying the reduction rule some state q does not recognize a kernel. Without loss of generality, we can assume that $L(q)$ is a language of minimal length. It follows that the target state of all the outgoing transitions of q is the same state q recognizing the kernel of $L(q)$. But then the reduction rule can be applied to eliminate q, contradicting the hypothesis. □

For some of the algorithms presented in this paper, the following is useful: Define $\hat{\delta}$ as a superset of δ, such that if $(q, a \cdot \Sigma^k, p)$ is in δ, for all $i \leq k$: $(q, a \cdot \Sigma^i, \Sigma^{k-i} \cdot p)$ is in $\hat{\delta}$. $\Sigma^0 \cdot x$ is defined as equivalent to x. Also:

$$\forall q \in Q, a \in \Sigma, i, j : j > i \Rightarrow (\Sigma^j \cdot q, a \cdot \Sigma^i, \Sigma^{j-i-1} \cdot q) \in \hat{\delta}$$

This enables us to reference languages that are not kernels.

By analogy, we define the remaining zip length $\hat{\gamma}$ of transitions such that $\hat{\gamma}(q, a) = k$ iff $(q, a \cdot \Sigma^k, q') \in \hat{\delta}$ for some $q' \in Q$.

3. Master Automata for Regular Languages

The master automaton for regular languages can be thought of as a collection of multiple automata, with the graph of the automaton containing circles and multiple connected components. It also contains not only one, but many accepting states.

This distinguishes it from the master automaton for fixed-length languages, as described in *Esparza 2012*. In the fixed-length language case, a well-structured master automaton for a given alphabet can be found, which is not possible in the case of regular languages.

Figure 2.1 on page 20 shows a part of the rMA (without the trap state).

In the master automaton states represent a language. The language of a state is given by the language its automaton A_L recognizes. Pointers to states correspond to DFAs and thus languages. To extract the DFA corresponding to an initial state in the master automaton, one can use the projection algorithm.

A language, given by its DFA, can be added by copying it into the rMA and then applying the minimization algorithm. This is different from the approach of fixed-length language master automata, where it is possible to directly construct states in the master automaton, thus omitting states of language already present in the automaton and averting the need for minimization.

Since the minimization algorithm will be of great importance for z-automata, we will recapitulate how it works. Like other algorithms to some common problems on rMAs (which will not be discussed here), the minimization algorithm is as for DFA, except that it is required to update all the pointers to initial states, which don't exist in the case of DFAs.

3.1. Projection

The projection of a master automaton to a language L (as discussed in section 2.1) is a way of retrieving the DFA recognizing L from a master automaton. In practice, L is a pointer to an initial state s, so we will use s as an input for the algorithm.

The algorithm starts at the designated initial state and recursively visits all of its successors (breadth-first search), using a worklist of unvisited states *(l. 2)*. All states visited and their transitions are added to the new automaton. The resulting automaton is the minimal automaton recognizing L.

3.2. Minimization

After adding a language to the rMA by copying its states into the rMA, one might get a master automaton, which is not minimal, i.e. where it is not the case, that distinct states

Procedure 1 Project(A_M, s)

Input: Master Automaton $A_M = (Q_M, \Sigma, \delta_M, F_M)$
Output: Automaton $A = (Q, \Sigma, \delta, s, F)$

```
1: W ← {s}, Q, δ, F ← ∅
2: while W ≠ ∅ do
3:     pick q from W
4:     add q to Q
5:     if q ∈ F_M then add q to F
6:     for all a ∈ Σ do
7:         q' ← δ̂_M(q, a)
8:         add (q, a, q') to δ
9:         if q' ∉ Q then add q' to W
10: return A
```

recognize distinct languages.

In order to get a minimal rMA, the minimization algorithm has to be applied. The algorithm described in Procedure 3 (p. 29) is quite similar to that for DFAs (see *Hopcroft 2007*). First, of all, the language partition of the automaton is computed.[1]

Definition 5. *The language partition P_l is a set of blocks, where two states are in the same block if and only if they recognize the same language. So, $[q]_{P_l} = [q']_{P_l}$ iff $L(q) = L(q')$, where $[q]_{P_l}$ is the block containing the state q. The language partition of an automaton is the language partition of the language it recognizes.*

Procedure 2 LanPar(A)

Input: non-minimal rMA $A = (Q_M, \Sigma, \delta_M, F_M)$
Output: The language partition P_l

```
1: if F_M = ∅ or Q_M\F_M = ∅ then return {Q_M}.
2: else P ← {F_M, Q_M\F_M}
3: while P is unstable do
4:     pick B, B' ∈ P and a ∈ Σ such that (a, B') splits B
5:     P ← Ref_P[B, a, B']
6: return P
```

The algorithm *LanPar* (Procedure 2, example on page 30) returns the language partition of an rMA which may be non-minimal. In the trivial case where all states are accepting or all are not accepting, the language partition consists of the single block Q_M, which is the set of all states *(l. 1)*.

If all other cases case, the algorithm initializes with the partition of the language into accepting and non-accepting states *(l. 2)*. It then refines the partition by splitting blocks *(l. 6)*, as long as the partition is unstable:

[1] The definitions of language partitions, instability, splitting, refinement, quotient automata and the LanPar algorithm are similar to and in part taken from *Esparza 2012*, p. 42ff

Procedure 3 Minimize(A)

Input: non-minimal rMA $A = (Q_M, \Sigma, \delta_M, F_M)$, Pointer set I to languages
Output: rMA $A/P_l = (Q_P, \Sigma, \delta_P, F_P)$

1: $Q_P \leftarrow LanPar(A)$
2: **for** $B, B' \in Q_P$, $q \in B$, $q' \in B'$, $a \in \Sigma$ **do**
3: **if** $(q, a, q') \in \delta_M$ **then add** (B, a, B') **to** δ_P
4: **for** $q \in F_M$ **do**
5: **add** $[q]_{P_l}$ **to** F_P
6: **for** $i \in I$ **do**
7: Point i to $[q]_{P_l}$, where q is its current target
8: **return** A/P_l

Definition 6. *Let B, B' be (not necessarily distinct) blocks of a partition P and let $a \in \Sigma$. The pair (a, B') splits B if there are $q_1, q_2 \in B$ such that $\delta(q_1, a) \in B'$ and $\delta(q_2, a) \notin B'$. The result of the split is the partition $Ref_P[B, a, B'] = (P \backslash \{B\}) \cup \{B_0, B_1\}$, where*

$$B_0 = \{q \in B | \delta(q, a) \notin B'\} \text{ and } B_1 = \{q \in B | \delta(q, a) \in B'\}.$$

A partition is unstable if and only if it contains blocks B, B', such that (a, B') splits B for some $a \in \Sigma$. $Ref_P[B, a, B']$ returns a refined partition P' that splits B into two blocks.

The algorithm does not specify any order in cases where more than one block can be split or where a block can be split by different letters.

The minimization algorithms calls the *LanPar* procedure to get the language partition of its input automaton *(l. 1)*. It then builds the quotient automaton. The states of the quotient automaton are the blocks of the original automaton's language partition.

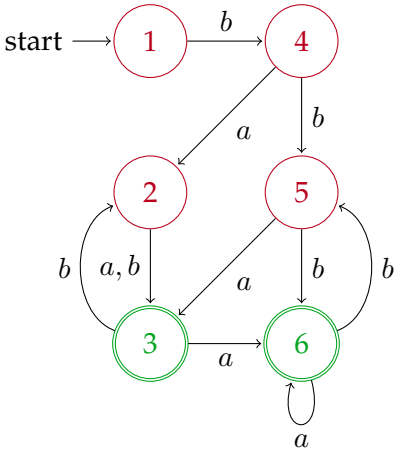
Next, transitions are added to the quotient automaton *(l. 2f)*. If in the original automaton there is a transition between two states, then the quotient automaton has a transition between the blocks of these states.

All blocks which consist of accepting states are added to the list of final states themselves *(l. 5f)*.

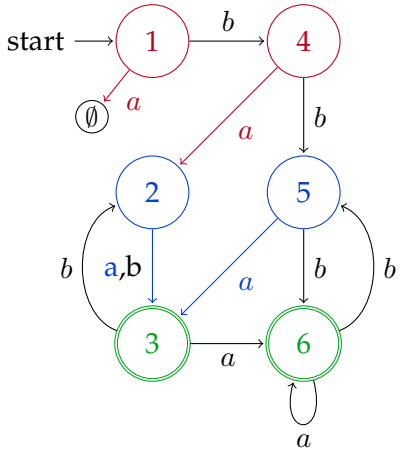
The only substantial difference of the minimization algorithm for rMAs in contrast to that for DFAs is in line 8f: There is more than one possible initial state, so all state pointers must be set to the corresponding blocks.

The quotient automaton is indeed minimal: By definition of the language partition, different blocks represent different languages. By Corollary 3.8 of *Esparza 2012* this implies the minimality of the automaton.

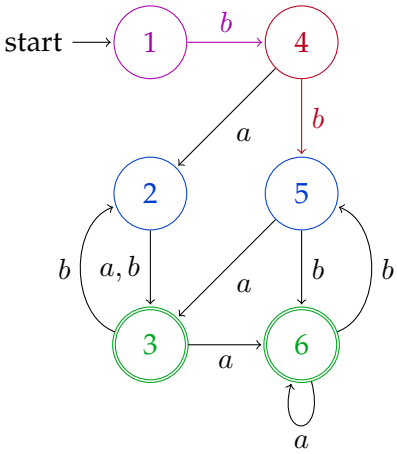
Figure 3.1 (p. 30) is an example run of the minimization algorithm.



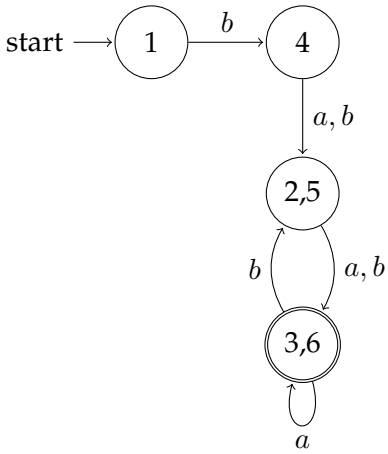
(a) Initial automaton



(b) Red block split by "a". Edge $1 \to \emptyset$ shown for clarity



(c) Red block split by "b"



(d) Resulting automaton with blocks as states

Figure 3.1.: Example of the minimization algorithm. (a)-(c) show the *LanPar* algorithm, which is a part of the minimization algorithm.

4. Operations on Z-Automata

The introduction of z-transitions necessitates the modification of algorithms to all common problems on automata. Its impact varies from small modifications (as in the *membership* algorithm) to severe problems in transforming the algorithms to work for z-automata (which we will see in the case of *minimization*). It will be shown that some algorithms can be applied seamlessly to z-automata, while others require expanding a large part of the z-transitions and thus giving up a lot of the gain that z-automata offer.

As with DFAs, the complement algorithm preserves minimality, while the BinOp algorithm does not.

Most of this section is concerned with algorithms on zDFAs. However, they can be easily applied to the master z-automaton for regular languages. At the end, we look at the minimization problem, which is specifically important for the master z-automaton. As with the master automaton, adding a language to the rMZ works by first copying the automaton representing the language to the rMZ and then minimizing the result. We will see that minimization of z-automata is in general hard to do. Thus, instead of adding an automaton directly to the rMZ, it is minimized and then added to the rMZ via a merging algorithm.

4.1. Membership

The concept of membership (described in procedure 4) in zDFAs is similar to that of DFAs. However, not only one letter $a \in \Sigma$ is read in each transition and iteration, but generally multiple letters.

The algorithm initializes at $q = q_0$ with the full word. Each iteration in the while loop represents a transition $(q, a \cdot \Sigma^i, q') \in \delta$ and cuts off the letters $a \cdot \Sigma^i$ of the word. It might be the case that the remaining word has fewer letters than the transition, in which case the initial word is not accepted and the algorithm returns *false (l. 6)*.

If this does not happen, then the automaton is in a defined state after the entire word has been read. The algorithm then simply returns whether that state is final *(l. 9)*.

Figure 4.1 is an example run of the membership algorithm on a zDFA.

4.2. BinOp

The *BinOp* algorithm (Procedure 5, p. 34) can be used to get the automaton that recognizes the result of a binary operation on the words of two languages (which are given by zDFAs). Much like the *BinOp* algorithm for DFAs, the states of both automata are combined: If, after reading a certain word, the first automaton is in state x and the second in y, then the resulting automaton is in (x, y). Since transitions of zDFAs read in general

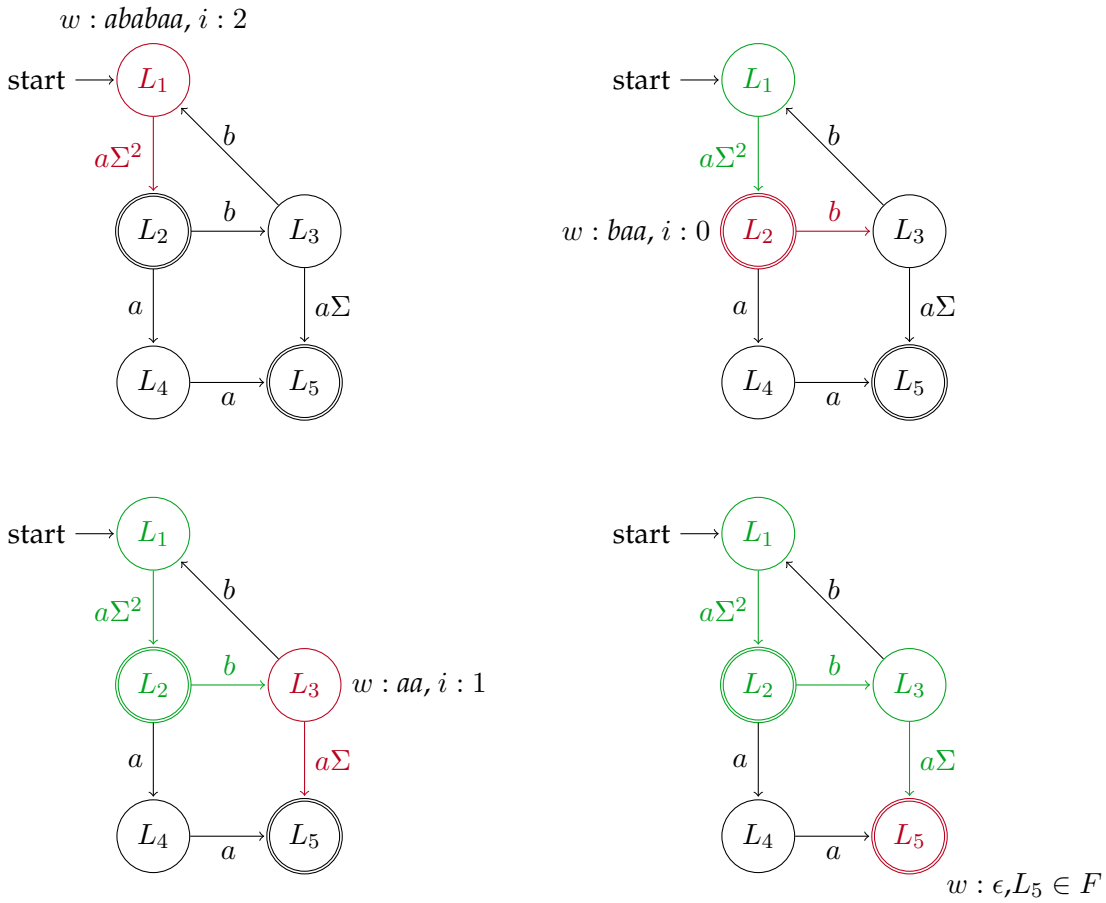
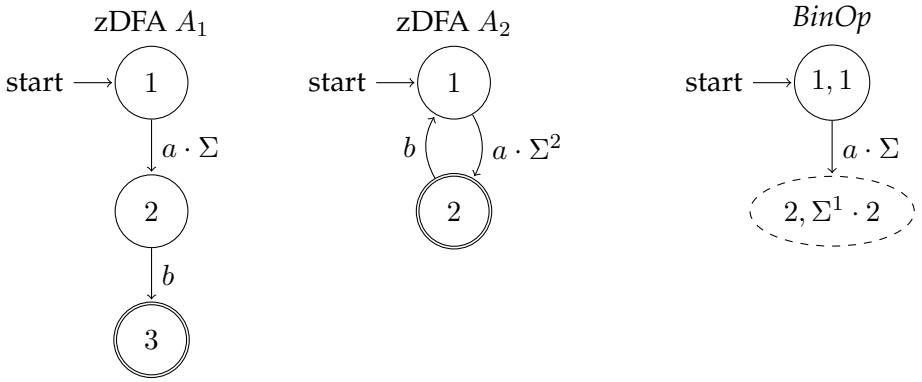


Figure 4.1.: Example membership test. The word *ababaa* is accepted by the automaton.

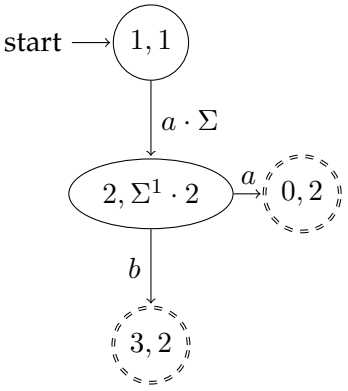
Procedure 4 Membership(A, w)

Input: zDFAs $A = (Q, \Sigma, \delta, q_0, F)$, word w
Output: *true*, iff A accepts w

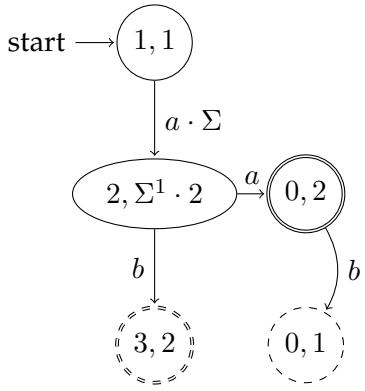
1: $q \leftarrow q_0$
2: **while** $w.length \neq 0$ **do**
3: $\quad a \leftarrow firstLetter(w)$
4: $\quad i \leftarrow \gamma(q, a)$
5: $\quad q \leftarrow \delta(q, a \cdot \Sigma^i)$
6: $\quad$ **if** $w.length < i + 1$ **then return** false
7: $\quad w \leftarrow cutOffLetters(w, i + 1)$
8: **return** $q \in F$



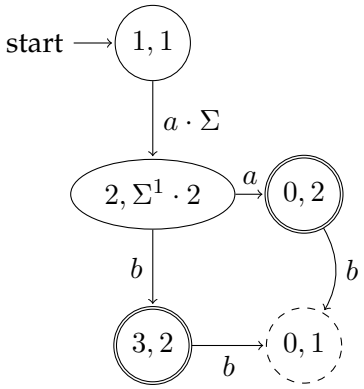
(a) From left to right: zDFA A_1, zDFA A_2, First step of BinOp$[\vee](A_1, A_2)$
Q: $\{(1,1)\}$, W: $\{(2, \Sigma^1 \cdot 2)\}$ For a: $\hat{\gamma}_1(1,a) = 1$, $\hat{\gamma}_2(1,a) = 2$, $m = 1$



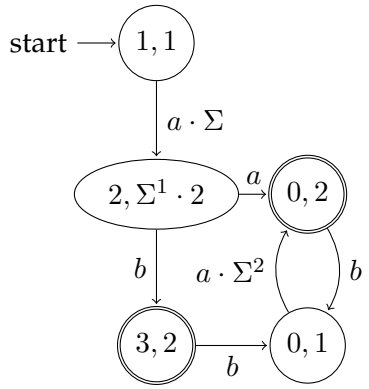
(b) Q: $\{(1,1), (2, \Sigma^1 \cdot 2)\}$, W: $\{(0,2), (3,2)\}$
For a: $\hat{\gamma}_1(2,a) = 0$, $\hat{\gamma}_2(\Sigma^1 \cdot 2, a) = 0$, $m = 0$
For b: $\hat{\gamma}_1(2,b) = 0$, $\hat{\gamma}_2(\Sigma^1 \cdot 2, b) = 0$, $m = 0$



(c) Q: $\{(1,1), (2, \Sigma^1 \cdot 2), (0,2)\}$,
W: $\{(3,2), (0,1)\}$
For b: $\hat{\gamma}_1(0,b) = 0$, $\hat{\gamma}_2(2,b) = 0$, $m = 0$



(d) Q: $\{(1,1), (2, \Sigma^1 \cdot 2), (0,2), (3,2)\}$,
W: $\{(0,1)\}$ For b: $\hat{\gamma}_1(3,b) = 0$,
$\hat{\gamma}_2(\Sigma^1 \cdot 2, b) = 0$, $m = 0$



(e) Final automaton. Q: $\{(1,1), (2, \Sigma^1 \cdot 2),$
$(0,2), (3,2), (0,1)\}$, W: $\emptyset$
For a: $\hat{\gamma}_1(0,a) = 0$, $\hat{\gamma}_2(1,a) = 2$, $m = 2$

Figure 4.2.: Example BinOp. From left to right: A_1, A_2, $A_1 \cup A_2$

Procedure 5 BinOp$[\odot](A_1, A_2)$

Input: zDFAs $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
Output: zDFA $A = (Q, \Sigma, \delta, q_0, F)$ with $L(A) = L(A_1)\hat{\odot}L(A_2)$

```
1:  Q, δ, F ← ∅
2:  q_0 ← [q_01, q_02]
3:  W ← {q_0}
4:  while W ≠ ∅ do
5:      pick [q_1, q_2] from W
6:      add [q_1, q_2] to Q
7:      if (q_1 ∈ F_1) ⊙ (q_2 ∈ F_2) then add [q_1, q_2] to F
8:      for all a ∈ Σ do
9:          m ← min(γ̂_1(q_1, a), γ̂_2(q_2, a))
10:         if q_1 = Σ* or q_1 = ∅ then m ← γ̂_2(q_2, a)
11:         if q_2 = Σ* or q_2 = ∅ then m ← γ̂_1(q_1, a)
12:         q'_1 ← δ̂_1(q_1, a · Σ^m)
13:         q'_2 ← δ̂_2(q_2, a · Σ^m)
14:         if [q'_1, q'_2] ∉ Q then add [q'_1, q'_2] to W
15:         add ([q_1, q_2], a · Σ^m, [q'_1, q'_2]) to δ
16:     apply reduction rule
17: return A
```

more than one letter, automata may also be "in between two states". For example, the "pseudo state" $\Sigma^2 \cdot q$ means after reading two letters, the automaton will be in state q.

The binary operation $\odot$ is a logical "or" for the union of two languages and "and" for intersection. $\hat{\odot}$ is defined directly on languages: $L_1\hat{\odot}L_2 = \{w \in \Sigma^* | (w \in L_1) \odot (w \in L_2)\}$,[1] so $\hat{\odot}$ can be "union", "intersection", "set difference" and so on.

The main loop of the algorithm is as for DFAs: Starting from both initial states[2], a breadth-first search is done in parallel on both automata *(l. 3)*. New states are added to a worklist and when a state is picked from the worklist, it is added to the set of states and its successors are determined.

In contrast to the *Complement* algorithm, which will be discussed in the next section, *BinOp* does not dissolve all of the z-optimizations. However, if the zip lengths differ, the shorter one is chosen *(l. 9)*, since the combined state might be final and thus a kernel.

In the algorithm, new states are created, which might not be kernels. Therefore, the reduction rule is applied after adding a state *(l. 17)*. Whether it can be successfully applied depends – among other things – on whether the state to be removed is final or not. Since that is determined by the binary operation *(l. 7)*, the structure of the resulting automaton is not the same for all operations.

There are two special cases which the algorithm takes into account *(l. 10-11)*: First of all, it might be that one of the automata is in the trap state. The zip length should then

[1] This definition is taken from *Esparza 2012*, p. 65.

[2] BinOp cannot have master automata as input, since it requires a distinct initial state for each input automaton.

not be set to zero, but be determined by the other automaton, since a trapped automaton can read an arbitrary number of letters without difference. Secondly, the same is true of the state which recognizes Σ^*. If one of these properties is true of both automata, then the resulting zip length will be zero.

The consideration of this special case saves time especially if one of the automata is in the trap state. However, it is not enough to make the reduction rule superfluous. Figure 4.3 shows an example where the state $(2, \Sigma \cdot 2)$ is created by the algorithm and later reduced from the resulting automaton.

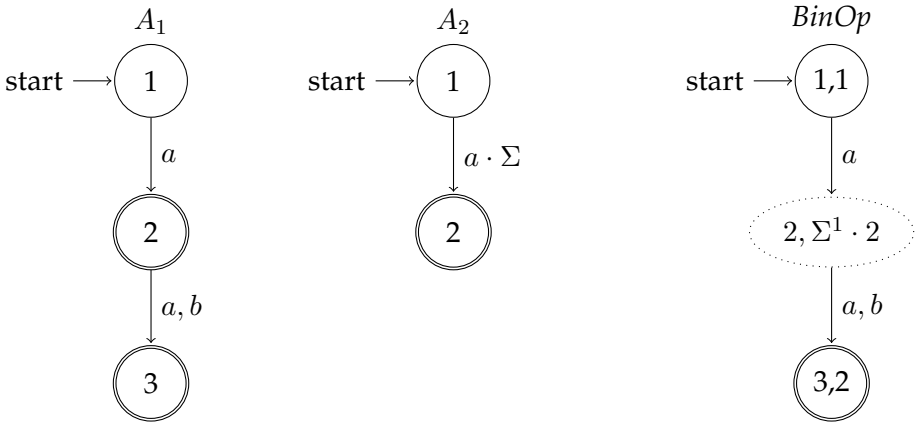


Figure 4.3.: An example where *BinOp*[$\wedge$], after adding $(3, 2)$, will apply the reduction rule and remove $(2, \Sigma^1 \cdot 2)$. Note that $A_1 \cap A_2 = A_2$

An easier algorithm would set the zip length of new transitions to one and extensively apply the reduction rule. By comparison, our algorithm saves most of that work by choosing a higher, more suitable zip length. Both approaches ensure that never the entire automaton (which may be very large) has to be stored in memory, since the reduction rule is applied after a state has been added and its successors determined.

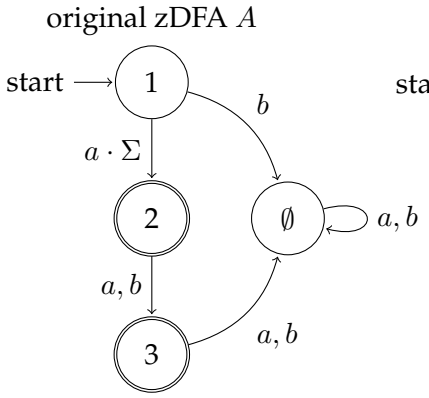
Figure 4.2 on page 33 shows an example run on two zDFAs and each iteration of the while loop.

4.3. Complement

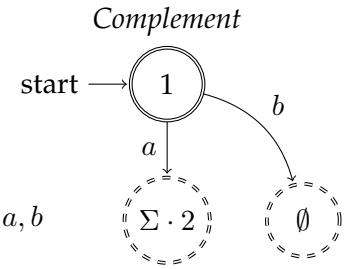
Given a language L by its zDFA A_L the *Complement* algorithm (Procedure 6) returns a zDFA $A_{\bar{L}}$, which recognizes the complement language $\bar{L}$, i.e. $\forall w \in \Sigma^* : w \in L \Leftrightarrow w \notin \bar{L}$.

In contrast to the *BinOp* algorithm, it is impossible to retain any z-transitions of the automaton. Since the non-kernel languages that are omitted from the set of states are non-final, their complement is final. Final states however, are by definition always kernels and thus must be present in the automaton.

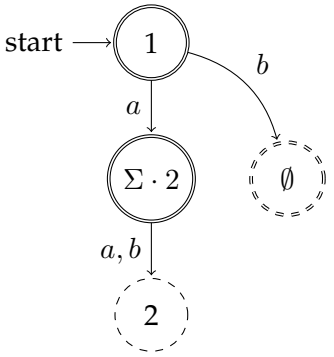
It is not necessary to save the whole DFA in memory though: The complement algorithm applies the reduction rule (if possible) right after a new state has been added.



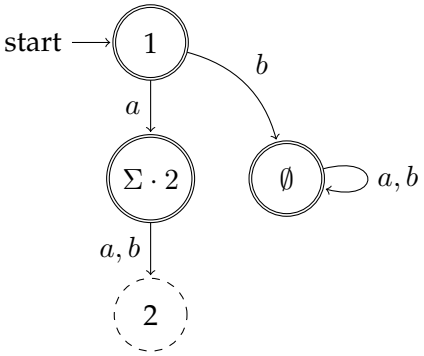
(a) Original zDFA with trap state



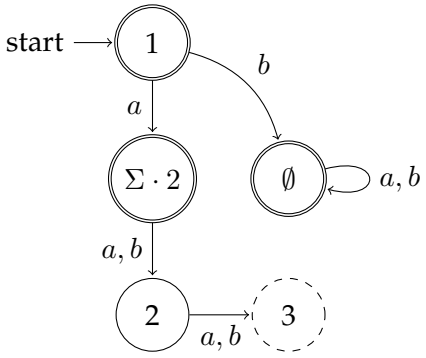
(b) Q: $\{1\}$, W: $\{\Sigma \cdot 2, \emptyset\}$
$\hat{\delta}(1, a) = \Sigma \cdot 2$, $\hat{\delta}(1, b) = \emptyset$



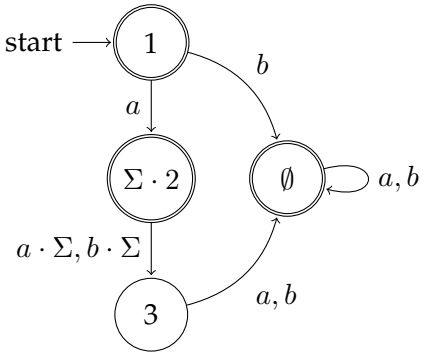
(c) Q: $\{1, \Sigma \cdot 2\}$, W: $\{\emptyset, 2\}$
$\hat{\delta}(\Sigma \cdot 2, a) = 2$, $\hat{\delta}(\Sigma \cdot 2, b) = 2$



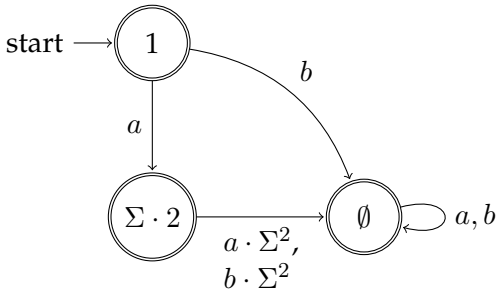
(d) Q: $\{1, \Sigma \cdot 2, \emptyset\}$, W: $\{2\}$
$\hat{\delta}(\emptyset, a) = \emptyset$, $\hat{\delta}(\emptyset, b) = \emptyset$



(e) Q: $\{1, \Sigma \cdot 2, \emptyset, 2\}$, W: $\{3\}$
$\hat{\delta}(2, a) = 3$, $\hat{\delta}(2, b) = 3$



(f) Q: $\{1, \Sigma \cdot 2, \emptyset, 2, 3\}$, W: $\emptyset$
$\hat{\delta}(3, a) = \emptyset$, $\hat{\delta}(3, b) = \emptyset$



(g) Final result.

Figure 4.4.: Sample run of the *Complement* algorithm. Between (e) and (f), as well as (f) and (g), the reduction rule has been applied.

Procedure 6 Complement(A)

Input: zDFAs $A = (Q, \Sigma, \delta, q_0, F)$
Output: zDFA $A' = (Q', \Sigma, \delta', q_0, F')$ with $L(A') = \overline{L(A)}$

```
 1: Q', δ', F' ← ∅
 2: W ← {q_0}
 3: while W ≠ ∅ do
 4:     pick q from W
 5:     add q to Q'
 6:     if q ∉ F then add q to F'
 7:     for all a ∈ Σ do
 8:         q' ← δ̂(q, a)
 9:         if q' ∉ Q then add q' to W
10:         add (q, a, q') to δ'
11:     apply reduction rule
12: return A'
```

The trap state, which we usually omit to draw for simplicity in this paper, has to be specifically taken into account in the *Complement* algorithm. Figure 4.4 is an example run of *Complement*.

The algorithm does a breadth-first search and adds each state to the worklist, unless it is already a part of $A_{\bar{L}}$ *(l. 9)*. The state is added to the set of final states if and only if it was not a final state in the original automaton *(l. 6)*. Since z-transitions have to be dissolved, $\hat{\delta}$ is used to reference non-kernels of the original language. For instance, in the example of Figure 4.4, $\hat{\delta}(1, a) = \Sigma \cdot 2$ and $\Sigma \cdot 2$ is a state of the newly created automaton and a kernel of its language.

All function calls $\hat{\delta}(q, a)$ in *Complement (l. 8)* are defined, since q is either of the form $\Sigma^i \cdot x$ ($i > 0$, x is a kernel of L), so that $\hat{\delta}(q, a) = \Sigma^{i-1} \cdot x$. Or q is a kernel of L, in which case $\hat{\delta}(q, a)$ will be $\Sigma^i \cdot x$, where $i = \hat{\gamma}(q, a)$.

4.4. Minimization

The indirect minimization algorithm for zDFAs is very simple and will be discussed in the first section. However, it is much harder to find a direct method.

One direct minimization algorithm for zDFAs and – in the more general case – rMZs is given by Procedure 10 (p. 47). It is based on the algorithm for DFAs and rMAs. However, due to different zip lengths of otherwise equivalent states, the language partition algorithm given by Procedure 9 (p. 47) is a a lot more complex than its DFA/rMA equivalent. In addition to its conceptional difficulty, a major result of this section will be that the zip-transitions have to be at least partially expanded to minimize. This is problematic, since it increases the computational complexity and limits the gains of the z-optimization.

A simple idea for a minimization algorithm on zDFAs would be to apply the reduction rule on all non-kernels in the input automaton before minimizing. This would solve

the problem of different zip lengths of equivalent states. Figure 4.5 (p. 39) shows that in general this is not possible, because in some cases the reduction rule can only be applied to a non-kernel in the minimal automaton. The second section will explain this problem and show that a non-expanding algorithm would be very inefficient.

The next approach is to allow for partial expansion of z-transitions, wherever required. This may come as a surprise, since the final automaton will consist only of kernels and expanding transitions creates non-kernels. However, they are needed to compute the language partition, which is required by the minimization algorithm. In this approach, arbitrary splits are permitted (so, for example, splitting a block by $a \cdot \Sigma^4$). It will be shown, that an algorithm following this concept would also have profound problems.

In contrast, the last approach only allows for splits by a single letter of the alphabet (e.g. a or b, but not $a \cdot \Sigma^4$). A working algorithm will be presented and some of the difficulties discussed.

4.4.1. Indirect minimization

The trivial approach to minimizing zDFAs is this: First, convert the given zDFA to a DFA. Then, minimize the DFA by expanding all z-transitions. Finally, exhaustively apply the reduction rule, which yields the minimal zDFA. While this approach is conceptionally easy, it is also inefficient: It requires expanding all z-transitions and thus giving up a large part of the benefits of zDFAs.

Proposition 4. *Given a zDFA A, converting it to a DFA A_{DFA}, minimizing A_{DFA} and exhaustively applying the reduction rule on the result A_{MinDFA} yields the unique minimal zDFA A'.*

Proof. First of all, note that expanding all z-transitions does indeed yield the DFA recognizing the same language, since it is just the reverse application of the reduction rule. Secondly, the minimization algorithm yields the minimal DFA (see section 3.2.1 in *Esparza 2011*). By Proposition 3 (c) (p. 24), exhaustively applying the reduction rule on a minimal DFA yields the minimal zDFA. □

4.4.2. Approach one: No expansion of z-transitions with arbitrary zip splits

A desirable preprocessing operation for the minimization algorithm on zDFA would be the reduction of all non-kernel states according to the reduction rule (Figure 2.4, p. 23). The idea is that this would solve the problem that there may be different zip lengths of equivalent states (i.e. states that recognize the same language). Take for example Figure 4.5: State 1 and state 4 have different zip lengths in their transitions under a ($a \cdot \Sigma^2$ from state 1 and $a \cdot \Sigma^3$ from state 4), even though they are in the same block. This difficulty would be solved by reducing state 2 – however, this is not possible until the states 3 and 5 are merged. So, Figure 4.5 shows that in general it is not possible to reduce before minimizing, because in some cases the reduction rule can only be applied in the minimal automaton to a non-kernel.

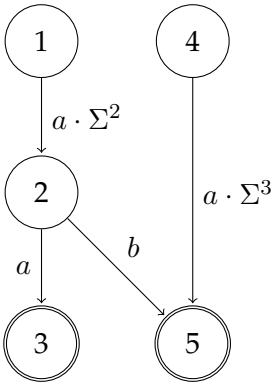


Figure 4.5.: Example zDFA where the reduction rule cannot be applied to state 2 until the states 3 and 5 have been merged by minimizing the automaton.

Instead, one has to cope with the fact that a reduction of all non-kernels is impossible without first minimizing the automaton. Figure 4.5 also shows the problem this fact leads to:

Assume we start the partitioning with the blocks $\{1, 2, 4\}$ and $\{3, 5\}$. One could try to split the block $\{1, 2, 4\}$ by any of the transition labels a, b, $a \cdot \Sigma^2$ or $a \cdot \Sigma^3$. The zip lengths of all transitions of states present in a block must be considered. Why is an ambiguity of zip lengths in a block under a letter a problem? Suppose, we pick a: We then have to decide whether the successors of $1, 2, 4$ under a are in different blocks. However, the successors not always part of the automaton. In our case, the successors of $1, 2$ and 4 under a are $\Sigma^2 \cdot 2, 3$ and $\Sigma^3 \cdot 5$ (where the pseudo-state "$\Sigma^2 \cdot 2$" means "after any two letters, we are in state 2"). There is no simple way of deciding whether $\Sigma^2 \cdot 2$ and $\Sigma^3 \cdot 5$ are in the same block.

In this example it might be possible to circumvent the problem, by first merging 3 and 5 (which is easy), applying the reduction rule on 2 and then merging 1 and 4. However, Figure 4.6 displays another example, where there is no such simple resolution. The block $\{2, 3\}$ should not be split under the transition b.

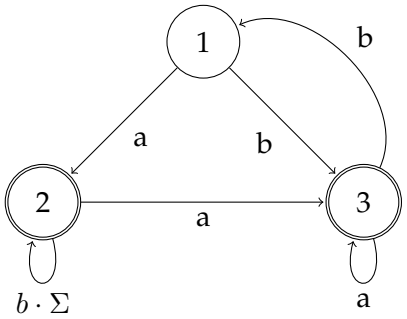


Figure 4.6.: 2 and 3 should not be split under $(\{2, 3\}, b)$.

On the other hand, there are similar cases where a split is appropriate. In Figure 4.7,

the block $\{1, 2\}$ can't be split by a b-transition, so an algorithm would try to split by some a-transition. $a \cdot \Sigma$ will lead to $\{2, \Sigma \cdot 1\}$ and $a \cdot \Sigma^2$ to $\{\Sigma^2 \cdot 1, 2\}$ or $\{\Sigma \cdot 3, 2\}$ (depending on what path we take). In all of these cases it is unclear whether a split is possible, because we can't know if the non-kernel state (e.g. $\Sigma \cdot 1$) is in the same block as the kernel-state (e.g. 2).

Unless we explicitly do the language partition also for pseudo-states (such as $\Sigma^2 \cdot 1$), it may be hard to figure out whether two states are equal.

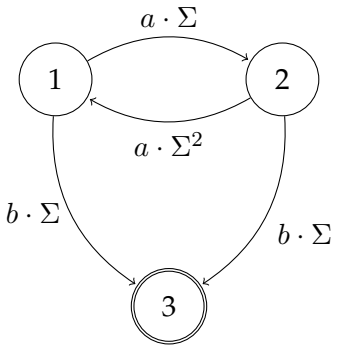


Figure 4.7.: 1 and 2 should be split.

It seems that the only way of finding out whether two states q_1, q_2 of the same block are unequal (and thus the block has to be split), is to find a word $w \in \Sigma^*$ such that $L(q_1)^w$ and $L(q_2)^w$ are different languages. This is either indicated by the fact that their states are in different blocks. For that, the successors of both states under w have to be in the automaton (i.e. they can't be pseudo-states like $\Sigma \cdot 1$). Or, one of the states has to be final and the other one non-final (e.g. state 1, state 2 or a pseudo-state in Figure 4.7).

A short word to split $\{1, 2\}$ in Figure 4.7 is $w = aba$, which leads to state 3 from state 1 and state $\Sigma \cdot 2$ from state 2. Since the former is final and the latter isn't (even though we have no additional information on it), it is possible split $\{1, 2\}$.

Figure 4.8 shows an example, where it is difficult to compute whether the states 1 and 1' are unequal. For all short words, one of the successors is a state, the other a pseudo-state. E.g. if we pick $w = abb$, the successor of 1 with respect to w is the state 3ab, whereas the successor of 1' with respect to w is the pseudo-state $\Sigma \cdot 3$ab. In the worst case, this means there are $O(2^n)$ words to consider, which renders this first approach impractical.

4.4.3. Approach two: Partial expansion of z-transitions with arbitrary splits

As previously demonstrated, it is very hard to directly construct the minimal zDFA of a given zDFA. The trivial workaround would be to expand all z-transitions (in other words, to turn the zDFA into its corresponding DFA), minimizing and exhaustively applying the reduction rule (i.e. turning the minimal DFA into a zDFA). As can be easily shown, this would yield the minimal zDFA.

However, a total expansion of all z-transitions would give up most of the profit gained by the z-optimization. So, instead the aim will be to partially expand z-transitions

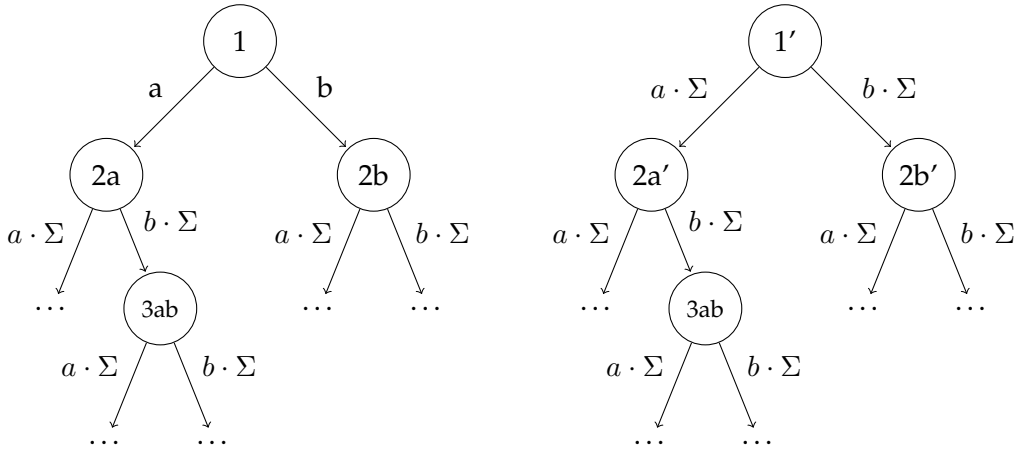


Figure 4.8.: Are states 1 and 1′ equal? Note that both parts of the automaton are equal, except for the initial transitions' zip length from 1, respectively 1′.

during the process of building the language partition, and only to the extend required by the language partition algorithm.

The idea for such an algorithm would be to start with a zDFA, and as usual, the final states and the non-final states as the first two blocks. Then, to figure out whether a block can be split, two of its states are compared with respect to one of their transitions (e.g. $a \cdot \Sigma^2$). If the other state has the same transition, then a quick comparison is possible – if the successors belong to different blocks, a split can be made. In case the other state does not have a similar transition (e.g. it has $a \cdot \Sigma^3$), the missing intermediate state has to be added and the z-transition partially expanded, such that there is a transition with the same zip length as the first one ($a \cdot \Sigma^2$). However, since the newly added state is not a part of any block in the partition, it has to be subsequently added to the block it belongs to.

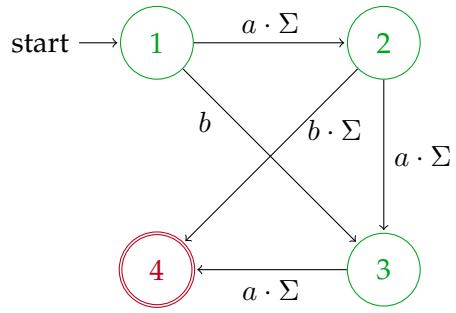
To add states subsequently to blocks, the entire history of blocks and splits has to be preserved. For example, if the initial block of non-final states has been split by a certain transition, the newly added state has to be examined with respect to this splitting transition, to determine which of the sub-blocks it belongs to. This has to be repeated, until it is clear in which of the current blocks the state belongs to.

During this process of finding the appropriate block of a new state, it might be required to add even more states. For example, when trying to figure out which block a newly created state belongs to, it might be that the initial block has been split by a transition a, but the new state has only a transition $a \cdot \Sigma^2$. Then, the immediate successor under a of the newly created state has to be added, before it can be figured out, which block the former state belongs to.

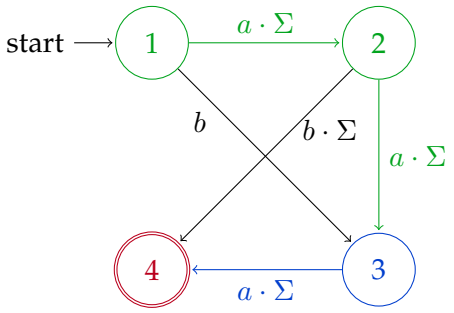
Figure 4.9 shows an example of this algorithm concept.

The complexity of this process is aggravated even more, since there might be circular dependencies.

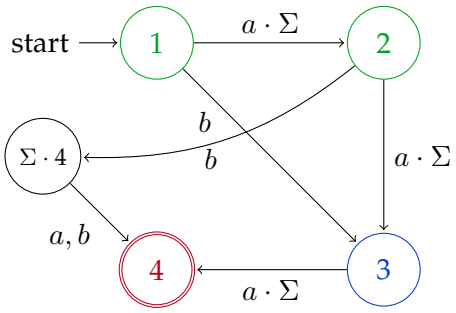
To illustrate, consider Figure 4.10, a constructed example for circular dependencies. The circular dependency is in the upper part, whereas the lower part is a helper struc-



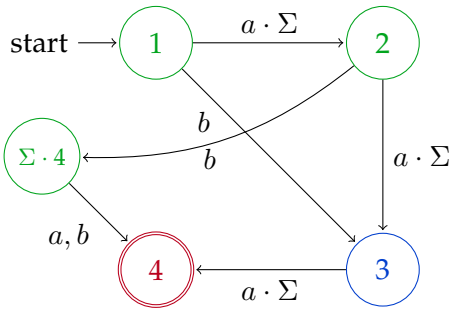
(a) Initial automaton and blocks



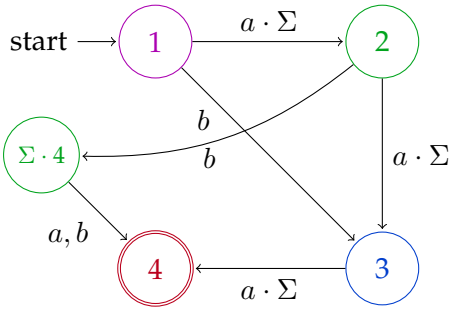
(b) Split $\{1, 2, 3\}$ by "$a \cdot \Sigma$"



(c) Adding $\Sigma{\cdot}4$ in preparation for split of $\{1, 2\}$ by "b"



(d) Adding $\Sigma \cdot 4$ to green block of (b), since it leads to $\emptyset$ under $a \cdot \Sigma$, which is not in the red block of (a).



(e) Splitting $\{1, 2, \Sigma{\cdot}4\}$ into $\{1\}, \{2, \Sigma{\cdot}4\}$ under b, since the successor of 1 under b is blue in (c) and the successors of 2 and $\Sigma \cdot 4$ aren't.

Figure 4.9.: Example of how minimization may take place (only the beginning is shown).

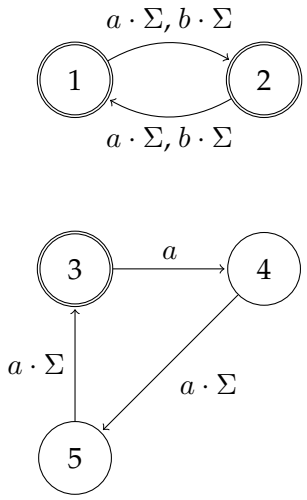


Figure 4.10.: Example zDFA where under a certain split sequence there will be circular dependencies.

ture to construct a certain split sequence.

Say, we initially split the block $\{4, 5\}$ by $a \cdot \Sigma$ into $\{5\}$ (leading to a final state) and $\{4\}$. Then, split the block $\{1, 2, 3\}$ by a. Starting at state 1, we have to create the intermediate state $\Sigma \cdot 2$. In order to figure out what block it belongs to, we have to look at what block the successor with respect to $a \cdot \Sigma$ belongs to, since that was the first split operation for non-final states. However, that successor does also not exist and has to be created as state $\Sigma \cdot 1$ (see Figure 4.11 for an illustration). Now, we have to find out which block $\Sigma \cdot 1$ belongs to. Again, the first split operation has to be applied, which was $a \cdot \Sigma$, for the initial non-final block. The successor of $\Sigma \cdot 1$ with respect to $a \cdot \Sigma$ is $\Sigma \cdot 2$ however. So, in total the block of $\Sigma \cdot 2$ is required and depends on the computation of $\Sigma \cdot 2$'s block.

This circular dependency is enough to dismiss the approach of arbitrary splits, even when z-transitions can be expanded in the algorithm.

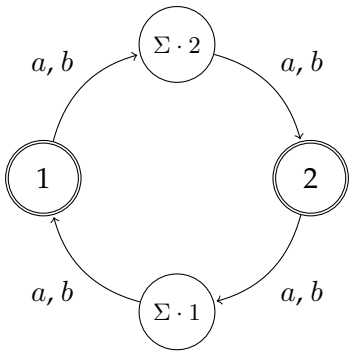


Figure 4.11.: Excerpt of the automaton in Figure 4.10 during the minimizing algorithm: The block of $\Sigma \cdot 1$ depends on $\Sigma \cdot 2$ and the other way around.

4.4.4. Approach three: Partial expansion of z-transitions with single-letter-splits

Instead of allowing splits by arbitrary $a \cdot \Sigma^i$ transitions, this next approach is going to only allow splits by a single letter. These have the advantage that circles cannot occur, since the "jumping over a state", as in Figure 4.10 can't take place. On the other hand, they preserve some z-transitions that do not have to be expanded.

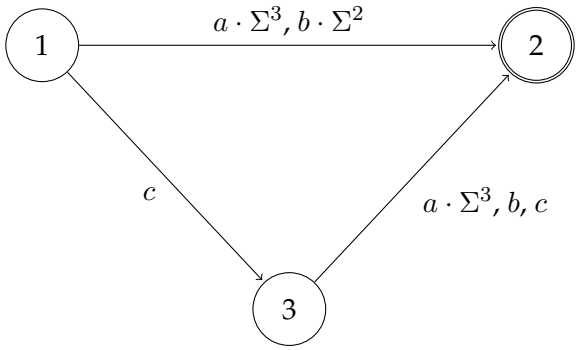


Figure 4.12.: In the minimization of this automaton, there are be multiple (potential) ways to split $\{1, 3\}$. Choosing a as a candidate for splitting is inefficient (and will not even result in a split), a fairly efficient split is possible with b and an efficient split with c.

As Figure 4.12 demonstrates, one can attempt a split of a block for every $x \in \Sigma$. Under some letters, a block might split and under others not (i.e. because all states lead to the same block). Unfortunately, there is now way of knowing whether a potential split succeeds until the z-transitions have been expanded.

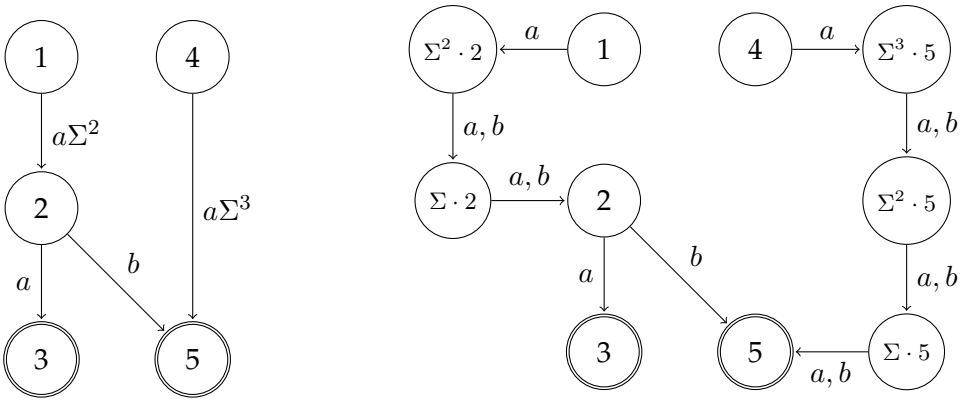
The algorithm of this approach is given by Procedure 10 (p. 47). There are various helper functions:

- *IsIsomorphic* (p. 46): Checks whether the states of a block are already isomorphic, that is: For a given block b and a letter a: Return true if and only if all states in the block have the same zip length i under a and lead to states of the same block under a.

- *Expand* (p. 46): Completely expands a given z-transition $a \cdot \Sigma^i$ by creating i intermediate states which are connected by single-letter-transitions *(l. 2)*. Since these new states are not part of any block, they are subsequently added, starting with the last one *(l. 5)*. They are guaranteed to have existing successors under all single-letter transitions, so during the process of finding their appropriate block, no new states have to be added. However, the complete history of partition is required to find the correct block.

- *LanPar* (p. 47): As for DFAs, this algorithm returns the language partition of an automaton. It is conceptionally much more difficult, however, since the complete history of partitions has to be preserved. This is required to subsequently add new

states.
A partition is unstable if there is a pair (B, a) that satisfies *l. 8* of *LanPar*.



(a) Initial automaton

(b) When trying to split $\{1, 2, 4\}$ under a, the z-transitions are fully expanded.

Figure 4.13.: Excerpt of the minimization algorithm, where complete expansion of transitions is required, when splitting the initial block of non-final states under a.

Some notes on the variables used in the various algorithms:

- Most of the variables should be understood as unique global variable with multiple references. Calls are usually *call-by-reference*, not *call-by-value* (see *Expand, l. 8* for an example).

- $P = \{P_0, P_1, \dots\}$ is the history of all partitions. P_0 is the initial partition of final and non-final states.

- $L = \{L_1, L_2, \dots\}$ is the history of all splitting letters. L_1 for example is the letter of the first block split.

- The function f is a reference to the resulting block after a split. It has two parameters. The first one is the current block (which should be split). The second one is the target block (where it should lead to). The value of the function is the resulting block, i.e. the block that leads to the block of the second parameter under L_n.

- N_i are the resulting blocks of a new split. In contrast to the minimization algorithm for DFAs, we split into any number of blocks, not just two.

Procedure 7 IsIsomorphic(P_n, B, a)

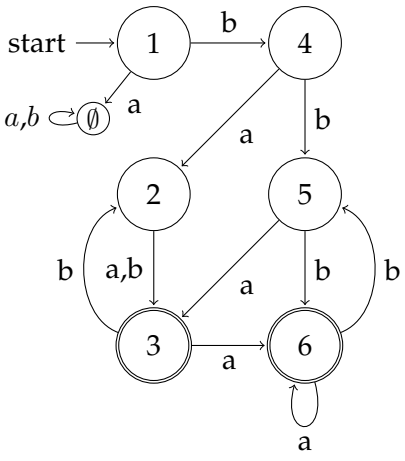
Input: Tuple (P_n, B, a)
Output: *true*, iff the transitions under a are isomorphic for all $q \in B$

1: **pick** $p \in B$
2: $i \leftarrow \gamma(p, a)$, $b \leftarrow [\delta(p, a \cdot \Sigma^i)]_{P_n}$
3: **for** $q \in B$ **do**
4: $\quad j \leftarrow \gamma(q, a)$
5: $\quad$ **if** $j \neq i \vee [\delta(q, a \cdot \Sigma^j)]_{P_n} \neq b$ **then return** *false*
6: **return** *true*

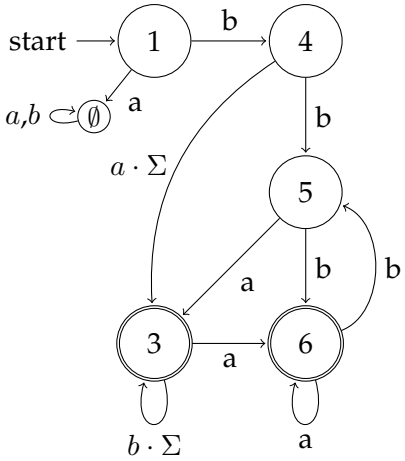
Procedure 8 Expand(A, P, L, n, q, a)

Input: Tuple (A, P, L, n, q, a)
Output: Tuple (A, P)

1: $i \leftarrow \gamma(q, a)$, $r \leftarrow \delta(q, a \cdot \Sigma^i)$
2: $\delta \leftarrow (\delta \backslash \{(q, a \cdot \Sigma^i, r)\}) \cup \{(q, a, \Sigma^i \cdot r)\} \cup \{\Sigma^j \cdot r, x, \Sigma^{j-1} \cdot r | x \in \Sigma, 1 \leq j \leq i\}$
3: **for** $j \leftarrow 1$ **to** i **do**
4: $\quad b_j \leftarrow Q_M \backslash F_M$
5: $\quad$ **for** $m \leftarrow 1$ **to** $n - 1$ **do**
6: $\quad\quad$ **if** $f(b_j, [\delta(\Sigma^j \cdot r, L_m)]_{P_m})$ **is defined then**
7: $\quad\quad\quad b_j \leftarrow f(b_j, [\delta(\Sigma^j \cdot r, L_m)]_{P_m})$
8: $\quad\quad\quad$ **add state** $\Sigma^j \cdot r$ **to** b_j
9: $\quad$ **add state** $\Sigma^j \cdot r$ **to** Q
10: **return** *(A, P)*



(a) Initial automaton



(b) Exhaustively applied reduction rule *(Minimize, l. 2)*.

Procedure 9 LanPar(A)

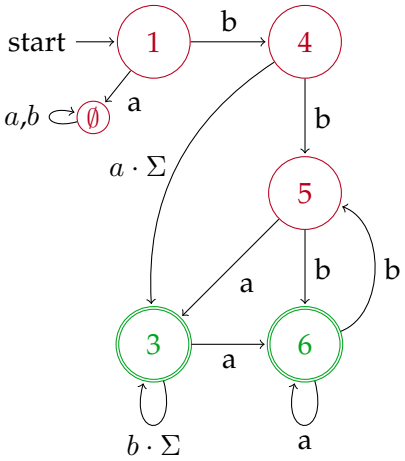
Input: zDFA $A = (Q, \Sigma, \delta, q_0, F)$
Output: The language partition P_l

for all $i \in \mathbb{N}$: $P_i \leftarrow \emptyset$; $P \leftarrow \{P_i | i \in \mathbb{N}\}$, $L \leftarrow \{L_i | i \in \mathbb{N}\}$, $f \leftarrow \emptyset$
if $F = \emptyset$ or $Q \backslash F = \emptyset$ **then return** $\{Q\}$.
else $P_0 \leftarrow \{F, Q \backslash F\}$
$n \leftarrow 0$
while P_n is unstable **do**
    $n \leftarrow n + 1$
    for all $i \in \mathbb{N}$: $N_i \leftarrow \emptyset$;
    pick $B \in P_n$, $a \in \Sigma$ **such that** !*IsIsomorphic*(P_n, B, a) **and** $|B| > 1$
    copy B **to** $B_{original}$
    for $q \in B$ **do**
        if $\gamma(q, a) \neq 0$ **then** $(A, P) \leftarrow$ *Expand*(A, P, L, n, q, a)
    for $q \in B$ **do**
        put q **into** N_i, **where** $\delta(q, a) \in B_i$
    for all nonempty N_i: **add** (B, B_i, N_i) to f.
    $P_n \leftarrow P_{n-1} \backslash \{B_{original}\} \cup \{N_i | i \in \mathbb{N}, N_i \neq \emptyset\}$
    $L_n \leftarrow a$
return P

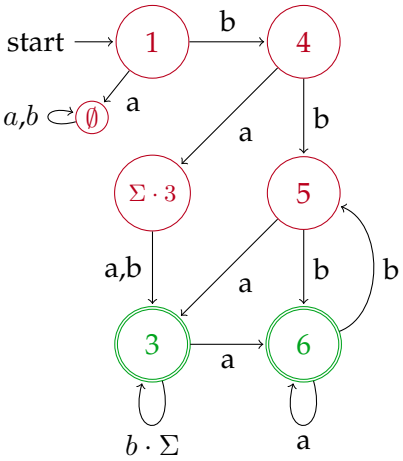
Procedure 10 Minimize(A)

Input: zDFA $A = (Q, \Sigma, \delta, q_0, F)$, Pointer set I to languages
Output: zDFA $A/P_l = (Q_P, \Sigma, \delta_P, q_{0P}, F_P)$

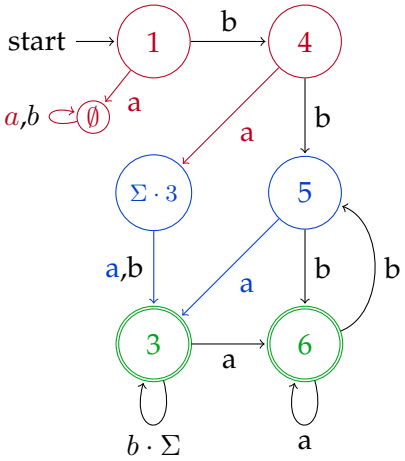
Exhaustively apply reduction rule on A.
$Q_P \leftarrow LanPar(A)$
for $B, B' \in Q_P$, $q \in B$, $q' \in B'$, $a \in \Sigma$ **do**
    if $(q, a \cdot \Sigma^i, q') \in \delta$ **for some** i **then add** $(B, a \cdot \Sigma^i, B')$ **to** δ_P
for $q \in F$ **do**
    add $[q]_{P_l}$ **to** F_P
$q_{0P} \leftarrow [q_0]_{P_l}$
Exhaustively apply reduction rule on A/P_l.
return A/P_l



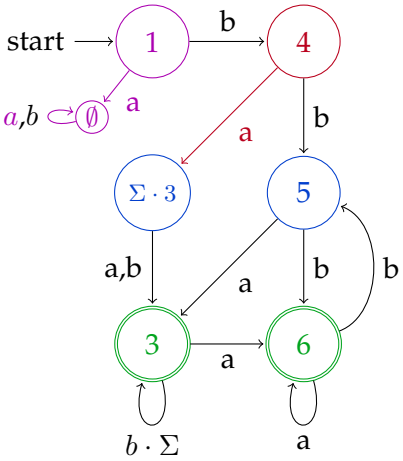
(c) Initialize *LanPar (LanPar, l. 3)*.



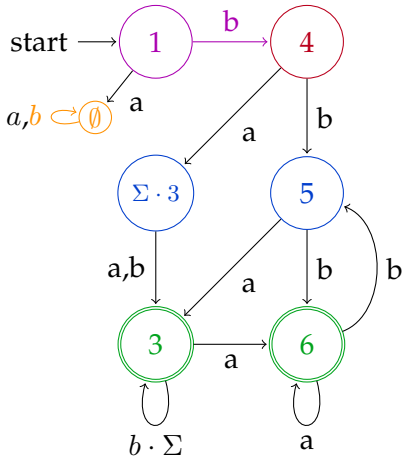
(d) *Expand(A, P, L, 0, 4, a)*. $\Sigma \cdot 3$ is added to Q *(Expand, l. 9)* and thus the first partition: $P_0 = \{\{1, \Sigma \cdot 3, 4, 5, \emptyset\}, \{3, 6\}\}$



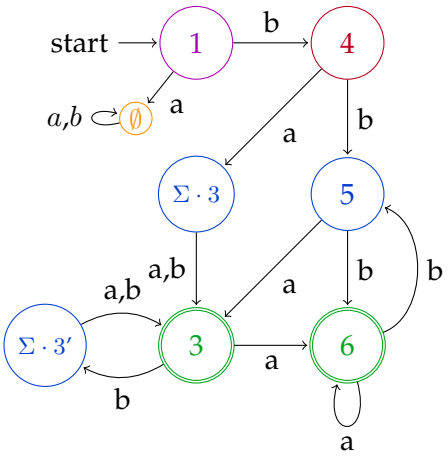
(e) Split $\{1, \Sigma \cdot 3, 4, 5, \emptyset\}$ under a.
$f(\{1, \Sigma \cdot 3, 4, 5, \emptyset\}, \{3, 6\}) = \{\Sigma \cdot 3, 5\}$
$f(\{1, \Sigma \cdot 3, 4, 5, \emptyset\}, \{1, \Sigma \cdot 3, 4, 5, \emptyset\}) = \{1, 4, \emptyset\}$ *(LanPar, l. 14)*.
$P_1 = \{\{1, 4, 5, \emptyset\}, \{3, 6\}\} \setminus \{\{1, 4, 5, \emptyset\}\} \cup \{\{\Sigma \cdot 3, 5\}, \{1, 4, \emptyset\}\}$
$= \{\{1, 4, \emptyset\}, \{\Sigma \cdot 3, 5\}, \{3, 6\}\}$



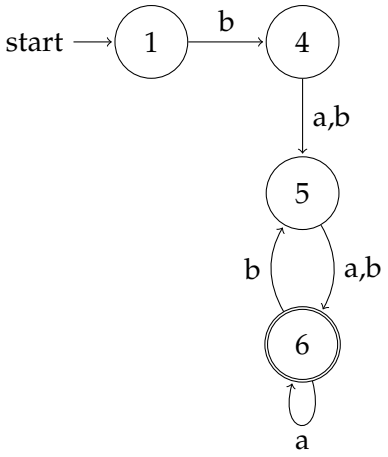
(f) Split $\{1, 4, \emptyset\}$ under a.
$f(\{1, 4, \emptyset\}, \{\Sigma \cdot 3, 5\}) = \{4\}$,
$f(\{1, 4, \emptyset\}, \{1, 4, \emptyset\}) = \{1, \emptyset\}$ *(LanPar, l. 14)*.
$P_2 = \{\{1, \emptyset\}, \{\Sigma \cdot 3, 5\}, \{3, 6\}, \{4\}\}$



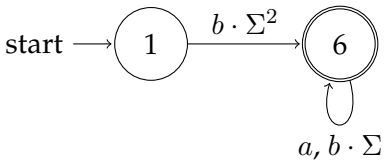
(g) Split $\{1, \emptyset\}$ under b.
$f(\{1, \emptyset\}, \{4\}) = \{1\}$,
$f(\{1, \emptyset\}, \{\emptyset\}) = \{\emptyset\}$ *(LanPar, l. 14)*.
$P_3 = \{\{1\}, \{\emptyset\}, \{\Sigma \cdot 3, 5\}, \{3, 6\}, \{4\}\}$



(h) *Expand(A, P, L, 4, 3, b)*. $\Sigma \cdot 3'$ is subsequentially added to Q, $\{1, \Sigma{\cdot}3, 4, 5, \emptyset\}$ and $\{\Sigma{\cdot}3, 5\}$ *(Expand, l. 8-9)*, thus the partition is:
$P_4 = \{\{1\}, \{\emptyset\}, \{\Sigma \cdot 3, \Sigma \cdot 3', 5\}, \{3, 6\}, \{4\}\}$



(i) Automaton A/P_l with blocks as states



(j) A/P_l, fully reduced

Figure 4.14.: Example of the z-minimization algorithm.

4.4.5. Conclusions

Minimizing zDFAs directly is both conceptually and computationally difficult. It requires expanding a large number of zips.

In addition to this loss of a major part of the gain that zip-transitions offer, sometimes

identical states are added. In Figure 4.14 (starting on page 45), in (h) a state $\Sigma \cdot 3'$ is created in addition to the already-existing $\Sigma \cdot 3$. This problem can be solved by adding a reference for pseudo-states, to which state they belong (so $\Sigma \cdot 3$ has a reference to 3). Then, whenever a state is expanded, it is first checked whether there already is an equivalent pseudo-state.

This further consideration would again increase the complexity of the minimization algorithm. The lack of a simple direct minimization is not as drastic for master automata as one might assume: Though it is hard to minimize a master automaton, there are more efficient ways for handling master automata. As we will see in the next section, *merging* two minimal z-automata into a minimal automaton can be done both efficiently and in a simple manner.

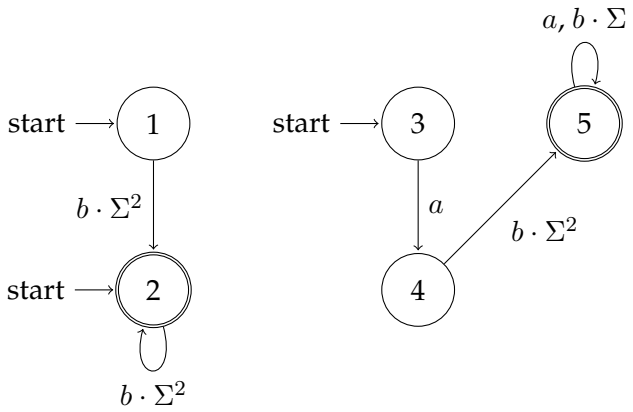
Since merging requires the existing automata being minimal, any automaton that should be added to the master automaton has to be minimized first. The options for doing so are the simple indirect method and the (conceptually) complex direct method of minimization (as discussed in the previous section). We choose the former, since the minimization algorithm for DFAs is well known and easy to implement. This is not done without any doubts, because there are both benefits (in some situations, z-transitions don't have to be expanded and thus the number of states considered is smaller) and detriments (saving of the entire history of blocks and splits is required, subsequential adding of states to blocks is required) of the direct method to indirect minimization.

4.5. Merging

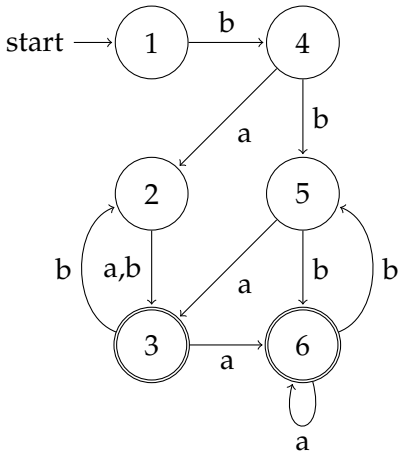
In the context of master automata, as discussed in section 3, adding a language works by copying its automaton to the master automaton and then minimizing the result. We've seen that minimizing zDFAs is hard. So instead, an approach of merging automata will be discussed.

The idea is that minimization is easier, if the task is to merge two minimal and fully reduced automata into one minimal automaton. Since each input automaton is minimal, the zip lengths from two states which recognize the same language are equivalent. This approach resolves a major obstacle that was present in the minimization problem: that of different zip lengths of equivalent states.

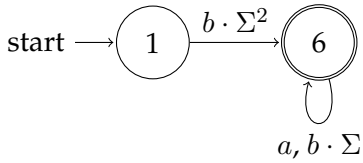
One of the automata can be the master automaton, the other one represents the language we wish to add. Since the master automaton is minimal by an invariant and the other automaton is typically small (and thus can be minimized indirectly via DFAs), presumably not a huge effort has to be invested to ensure that the input automata to the merging algorithm are minimal. Procedure 12 (p. 54) is such an algorithm and Figure 4.5 (p. 50) is an example run.



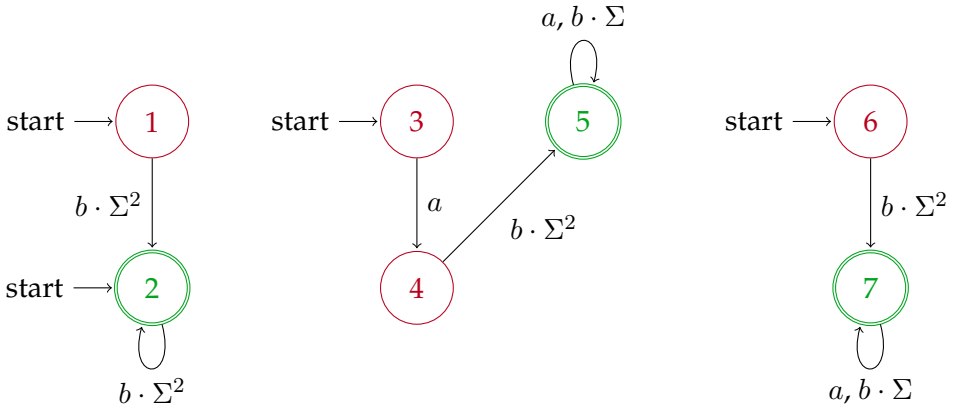
(a) Input automaton 1: rMZ.



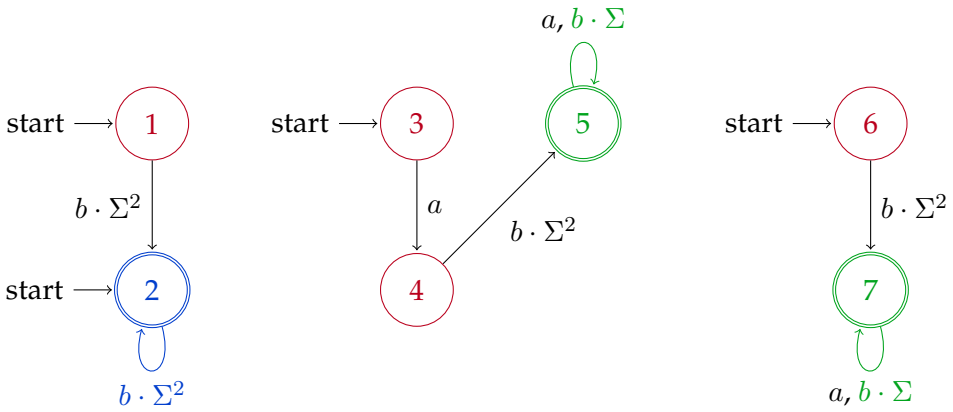
(b) DFA that should be added.



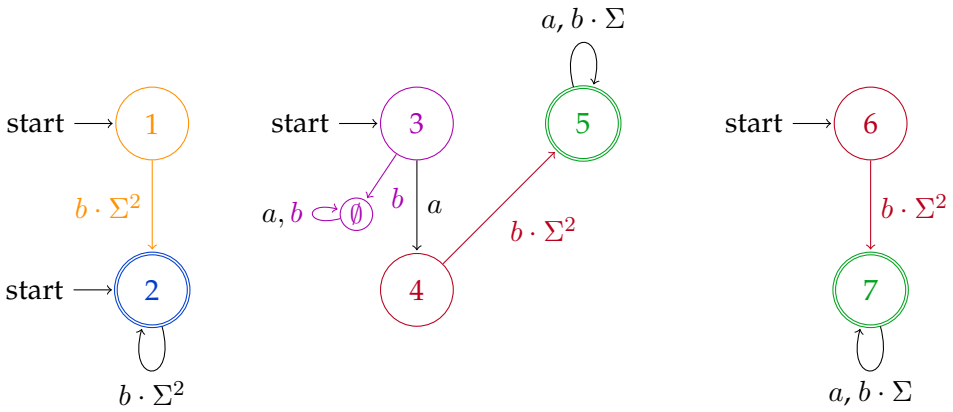
(c) Input automaton 2: Minimal and fully reduced zDFA based on (b).



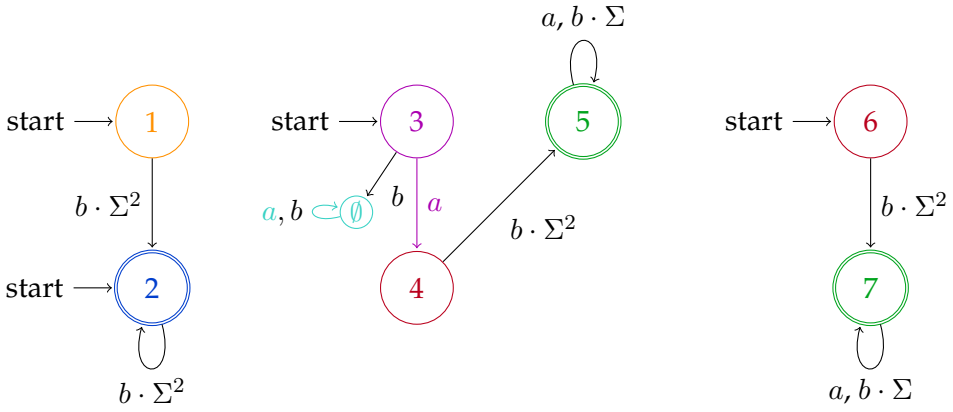
(d) Merged automaton of (a) and (c) that has to be minimized.



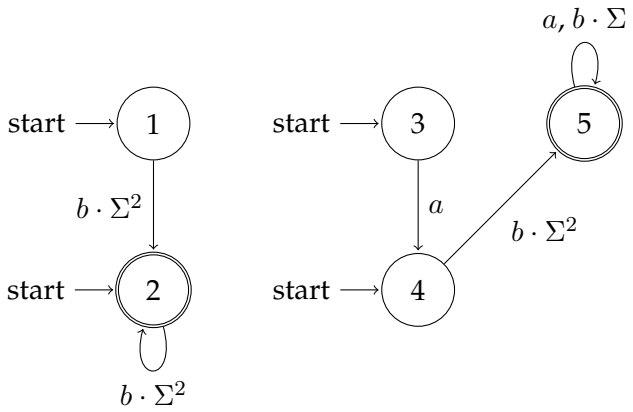
(e) Splitting $\{2, 5, 7\}$ by letter b. All states lead to the same block, but the zip lengths in $\{5, 7\}$ differ from that in $\{2\}$.



(f) Splitting $\{1, 3, 4, 6, \emptyset\}$ by letter b into $\{1\}, \{4, 6\}, \{3, \emptyset\}$ due to the different blocks the states lead to.



(g) Splitting $\{3, \emptyset\}$ by letter a into $\{3\}, \{\emptyset\}$ due to the different blocks the states reach.



(h) Final rMZ with the blocks of (g) merged.

Figure 4.15.: Example of the merging algorithm that is a common scenario when using a rMZ as data structure. We have an existing rMZ (a) and want to add a language (given by a DFA (b)) to it. First, we minimize the DFA in (b)-(c), then we apply the merging algorithm in (d)-(h).

Procedure 11 LanPar(A)

Input: non-minimal rMA $A = (Q_M, \Sigma, \delta_M, F_M)$
Output: The language partition P_l

1: **if** $F_M = \emptyset$ or $Q_M \backslash F_M = \emptyset$ **then return** $\{Q_M\}$.
2: **else** $P \leftarrow \{F_M, Q_M \backslash F_M\}$
3: **while** P is unstable **do**
4: pick $B, B' \in P$ and $a \in \Sigma$ such that (a, B') splits B
5: $P \leftarrow \mathit{Ref}_P[B, a, B']$
6: **return** P

The language partition algorithm is almost equivalent to that for DFAs. However, the definition of stability varies. A block is not only unstable if transitions under a letter lead to different blocks, but also if transitions under a letter differ in their zip length. This is a valid extension of the stability criterion, because states with different zip lengths can't recognize the same language and be minimal at the same time, due to the fact that there is a unique minimal zDFA.

We allow blocks to be split into more than two blocks. So first, blocks are split under a letter iff they lead to different blocks. The resulting blocks are then split again, if their states differ in their zip length under that letter. This two-stage split may speed up the whole process, since fewer iterations of the loop in *l. 3* of the *LanPar* algorithm are required.

The merging algorithm also has a lot in common with the minimization algorithm for DFAs. The difference lies in the input, which contains two automata instead of one and both have to be minimal (which is not verified, since it is computationally inefficient). They are immediately merged *(l. 1)*. A pointer to the block of A_2's initial state is

Procedure 12 Merge(A_1, A_2)

Input: minimal rMZ $A_1 = (Q_1, \Sigma, \delta_1, F_1)$, minimal zDFA $A_2 = (Q_2, \Sigma, \delta_2, q_0, F_2)$, Pointer set I to languages
Output: minimal rMZ $A/P_l = (Q_P, \Sigma, \delta_P, F_P)$, Pointer set I to languages

1: **copy** A_1 **and** A_2 **into** $A = (Q, \Sigma, \delta, F)$
2: $Q_P \leftarrow LanPar(A)$
3: **for** $B, B' \in Q$, $q \in B$, $q' \in B'$, $a \in \Sigma$ **do**
4: **if** $(q, a, q') \in \delta$ **then add** (B, a, B') **to** δ_P
5: **for** $q \in F$ **do**
6: **add** $[q]_{P_l}$ **to** F_P
7: **for** $i \in I$ **do**
8: **point** i **to** $[q]_{P_l}$, where q is its current target
9: **add new pointer** $[q_0]_{P_l}$ **to** I
10: **return** A/P_l, I

added to the set of pointers *(l. 9)*.

This approach copes well with problem of an inefficient minimization algorithm. The master automaton does not have to be minimized at any time and thus the master automaton optimization performs well. However, the problem remains that individual z-automata (which are also heavily used when employing master automata) have to be minimized, which are (put together) at least as large as the master automaton. So introducing the merging algorithm does not solve the problem of inefficient minimization, it just lessens impact for master automata to a level one would normally have with z-automata.

5. Conclusions

Z-automata for regular languages are a worthwhile endeavor wherever one expects to have automata with many reducible states or perform lots of membership operations. We have proven an important property for automata: existence of a unique minimal automaton. While z-automata introduce some difficulties, they offer the advantage of a reduced number of states and transitions. This not only leads to a more compact representation, but also to faster handling of automata. For example, the membership algorithm profits greatly when there are transitions with large zips.

This is especially true for fixed-length languages. For them, a common field of application has been described in the introduction. When one needs to represent and manipulate large boolean functions, z-transitions can to help reduce the size of automata.

Automata that recognize regular languages have a more diverse field of application, so it is difficult to determine whether the introduction of z-transitions for them has equally strong advantages. However, it clearly has problematic disadvantages: With most operations it is difficult or impossible to retain z-transitions. For example, the complement algorithm requires the expansion of all z-transitions. This not only means that the computational complexity is as great as for DFAs, but because of the additional overhead of expanding z-transitions, complementing a zDFA is even less efficient than complementing the original DFA. It is only a small comfort to know that the space complexity is better, due to the fact that at no point the fully-expanded automaton has to be saved, since the reduction rule can be applied on the fly.

The BinOp algorithm can only retain zips to a certain degree. For example, if a state with $a \cdot \Sigma^{23}$ and a state with $a \cdot \Sigma^{42}$ are combined, the new state has zip length of the minimum of both, 23, under a. For this operation, the z-optimization might be worth the effort.

On the other hand, the basic operation of membership is very efficient, like its fixed-length language equivalent.

The largest conceptual problem with z-automata lies in the minimization algorithm. We have considered multiple approaches which turned out to be flawed, because ultimately the membership of a state to a block in a partition can only be decided if it is explicitly added. One has to add pseudo-states of the form $\Sigma^i \cdot q$ to the automaton and the partition just to *check* whether one of the current blocks can be split. This inefficient method is the only way of solving the "chicken or egg" problem of Figure 4.5 (p. 39): in order to minimize we have to have a fully reduced automaton (so that transitions of equivalent states don't differ) and in order to fully reduce, we have to have a minimal automaton (so that the reduction rule can reduce states that lead to states which, though they are distinct states, recognize the same language).

Fortunately, this problem does not affect the conception of master automata too much. An inefficient minimization algorithm would be fatal for a master automaton,

since we can assume that it is much larger than any ordinary zDFAs. The master automaton should always be minimal after a language has been added. The idea we presented with the merging algorithm is this: minimizing the combined automaton (master automaton plus a new automaton) is much easier if each automaton is itself already minimal.

To elaborate, instead of adding an automaton to the master automaton directly, we first minimize it. This is costly, because there is no efficient way that conserves the z-transitions, but it is much better than minimizing the combined automaton that includes the large master automaton. After the automaton is minimized, it is added to the (already minimal) master automaton using the merging algorithm, which detects and merges equivalent states. As we have seen, the merging algorithm does not require expanding any z-transition and is as fast or faster than the minimization algorithm.

So, at no time the whole master automaton has to be minimized. To apply operations on languages in the master automaton, first the master automaton is projected to the initial states of those languages. Next, the operation is applied. Then, the minimization algorithm has to be applied. Finally, the resulting automaton is added to the master automaton via the merging algorithm.

The question remains: Are there problems which are solved by automata with many reducible states? In those instances, z-automata have a plain advantage over ordinary DFAs, at least if one refrains from employing operations that require the expansion of a large number of zips.

The same consideration of the operations required is valid for the more general case, when one does not expect to have lots of reducible states. However, the advantages and thus the arguments for z-automata carry less weight. When one has to perform lots of membership tests on an automaton, without too many inefficient operations, z-automata are a good choice.

It should be noted that these are not formal examinations of the computational complexity of the individual algorithms. In the worst case, all of these algorithms perform at least as badly as their DFA counterparts, since in some cases no reduction is possible. The average case is also difficult to determine, since it depends on how well an "average" automaton is reducible (which leads to the almost philosophical question, what an average automaton is). These are first results that do not immediately translate to quantitatively measurable differences in the execution time of implementations. Further research could include a closer examination of the complexity of the algorithms or real-life performance tests.

To summarize, we have shown that z-automata and master automata for regular languages are feasible and that algorithms for common operations exist, though they vary in their efficiency. While both of the optimizations discussed bring about a reduced number of states and transitions, their impact on the performance of the algorithms must be further researched before it can be definitely concluded whether they are an option for real-life applications.

Glossary

Σ The alphabet of an automaton or a language. It is a set containing letters, e.g. $\Sigma = \{a, b\}$.

$\emptyset$ $\rightarrow$ trap state.

ϵ the empty word, i.e. the unique word of length zero.

accept An automaton is said to accept a word w if there is a path from the initial state to a final state that reads w. An automaton is said to accept a language L if it accepts all words in it. For an accepting state, see $\rightarrow$ final state.

BDD $\rightarrow$ Binary Decision Diagram.

Binary Decision Diagram a data structure for boolean functions.

DFA a deterministic finite automaton, i.e. an automaton with states (among them one initial state and a number of final or accepting states) and transitions between states, that read letters of an alphabet $\rightarrow$ Sigma. An automaton is deterministic if for each state and letter, there is a unique transition to another state.

expanding applying the $\rightarrow$ reduction rule in the reverse direction. Required to convert a $\rightarrow$ zDFA to a $\rightarrow$ DFA.

final state is a state accepting the empty word ϵ.

JFLAP-Z A modified version of JFLAP, an automaton learning tool. JFLAP-Z includes support for $\rightarrow$ z-transitions and $\rightarrow$ master automata.

kernel the kernel K of a language L is given by "cutting off" the Σ^i prefix of a language, i.e. $L = \Sigma^k \cdot K$ for the highest k or $k = 0$ if no such number exists. A kernel language is its own kernel.

Master Automaton a single automaton that includes a multitude of languages and initial states. A master automaton is used instead of multiple automata, that may share some of their structure and are thus redundant.

pseudo state a state that is not part of the original automaton but might be considered during an algorithm (for example, the complement algorithm). It is represented by $\Sigma^i \cdot r$, which means "after reading i letters, the automaton is in state r".

recognize An automaton is said to recognize a language if it accepts all words in it and no other words.

reduction rule a simple scheme for removing a suitable state from an automaton and replacing all incoming and outgoing transitions with a z-transition of a higher zip length. See also Figure 2.4 (p. 23).

rMA *master automaton for regular languages*, an automaton that incorporates many automata by having multiple initial states. The set of states is the set of regular languages.

rMZ *master z-automaton for regular languages*, an automaton that incorporates many z-automata by having multiple initial states. The set of states is the set of $\to$ kernels.

trap state The trap state is the unique state that is both non-final and has no transitions to other states. It is required for some operations, such as minimization. It is often abbreviated with $\emptyset$, since $\emptyset$ is the language it recognizes.

z-automaton an automaton with $\to$ z-transitions, i.e. transitions that have the form $a \cdot \Sigma^i$ for some $a \in \Sigma$, $i \in \mathbb{N}$.

z-optimization An automaton utilizing $\to$ z-transitions (i.e. a z-automaton). The automaton is optimized, since the number of states and transitions is reduced.

z-transition A transition that reads the regular expression $a \cdot \Sigma^i$ for some $a \in \Sigma$, $i \in \mathbb{N}$.

zDFA a deterministic z-automaton.

zip a regular expression of the form $a \cdot \Sigma^i$ (for some $a \in \Sigma, i \in \mathbb{N}$). The zip length is i.

Bibliography

[1] Bryant, Randal: *Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams*, ACM Computing Surveys, Vol. 24, No. 3 (September, 1992), pp. 293–318.

[2] Esparza, Javier: *Automata theory. An algorithmic approach.* Lecture Notes, July 20, 2012.[1]

[3] Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation.* Pearson International Edition, 2007

[4] Huth, Michael and Ryan, Mark: *Logic in Computer Science. Modelling and Reasoning about Systems.* Cambridge University Press, second edition, Cambridge 2004

[1] Note that this is a work in progress and thus the page numbers in this work may differ from those in the current version.

A. Alternative definition of Z-Automata

This section briefly discusses an alternative application of the z-optimization on finite state machines and its implication for the algorithms presented in this paper.

A difference between a fixed-length language z-automaton and a regular language z-automaton is that the former has only one final state, whereas the latter can have a multitude of final states. So, a question that did not come up with fixed-length languages is this: Should final states be reducible?

According to the definition given in Definition 2 (p. 21) for zDFAs, final states cannot be kernels. To illustrate, an example introduced in Section 2.2:

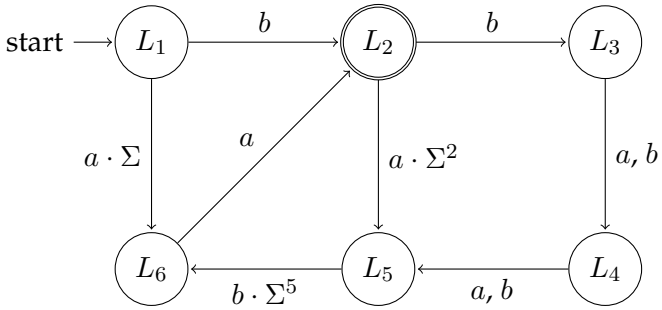


Figure A.1.: Sample zDFA

As discussed earlier, L_3 and L_4 of Figure A.1 can be reduced, but L_2 can't, because it follows from the definition of kernels, that accepting states are always kernels.

We can change the definition, so that final states can be reduced as well. This will yield a higher savings in the number of states and transitions we have to store. To distinguish automata following the alternate definition from the zDFAs as defined above, we will call the former "arzFA" (for "**a**ccepting states **r**educible"):

Definition 7. *A z-automaton with accepting states reducible (arzFA) is a tuple $A = (Q, \Sigma, \delta, q_0, f)$, where Q is a finite set of states, Σ is the alphabet, δ is the transition function and q_0 is the initial state. f is the unique final state of the automaton. Let $\delta : Q \times Z(\Sigma) \to Q$ be so that for every $a \in \Sigma$ and $q \in Q$ there is at exactly one $k \in \mathbb{N}$ and $q' \in Q \backslash \{f\}$ such that $(q, a \cdot \Sigma^k, q') \in \delta$ and there is an arbitrary number of elements in δ, where $q' = f$.*
We define a function $\gamma : Q \times \Sigma \to \mathbb{N}$ where $\gamma(q, a) = k$ iff $(q, a \cdot \Sigma^k, q') \in \delta$ for some $q' \in Q \backslash \{f\}$.

Under this definition, automata are *partially* indeterministic. Each state can have indeterminism, but beside a single transition to an arbitrary state, only transitions to the final state are allowed. Figure A.2 shows how such an arzFA looks like. In this instance, there is not much of a savings effect compared to the zDFA (since there is only a single

final state that can be reduced). However, in the more general case final states can be reduced in addition to non-final states.

In order to turn a given DFA into a arzFA, one has to add a new final state f, copy all incoming transitions of final states and direct these copies to f. Lastly, all states except for f are made non-final.

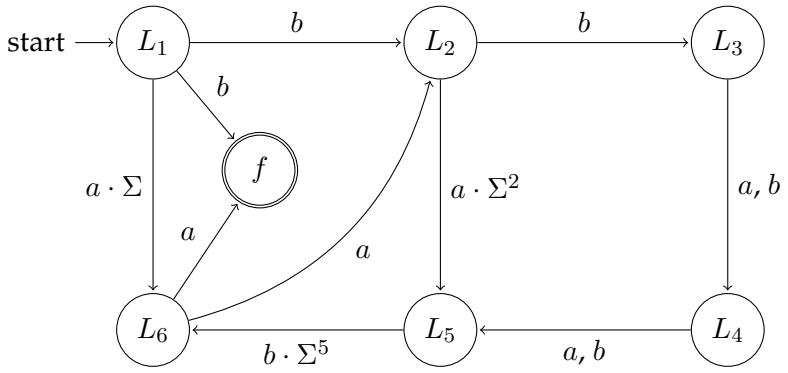


Figure A.2.: arzFA (recognizing the same language as the automaton in Figure A.1)

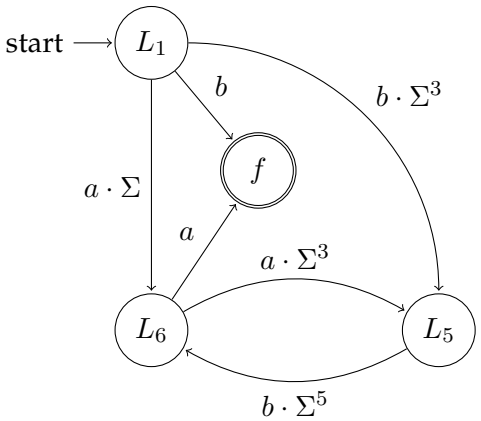


Figure A.3.: reduced arzFA of the arzFA in Figure A.2

In order to get a reduced arzFA, one has to apply reduction rules. The first reduction rule is that for zDFAs (see Figure 2.4, p. 23). The second and new reduction rule for arzFAs is given by Figure A.4.

Another simple example in Figure A.5 shows that one needs to examine closely the transitions from a state to the final state f. The product construction of the reduction rule ($l \cdot \{a_1, \ldots, a_n\}$) suggests that the number of transitions to f is large and might consist of words that are not of the form $a \cdot \Sigma^i$ (but rather arbitrary words). But, in fact all such transitions can always be expressed by z-transitions – in this instance by $l \cdot \Sigma^2$.[1]

[1]Proof sketch: Assume there is transition from a state q to f reading a word ab ($a, b \in \Sigma$). Then there must have been a state q' such that $(q, a, q') \in \delta$. Since q' has been removed, there must have been

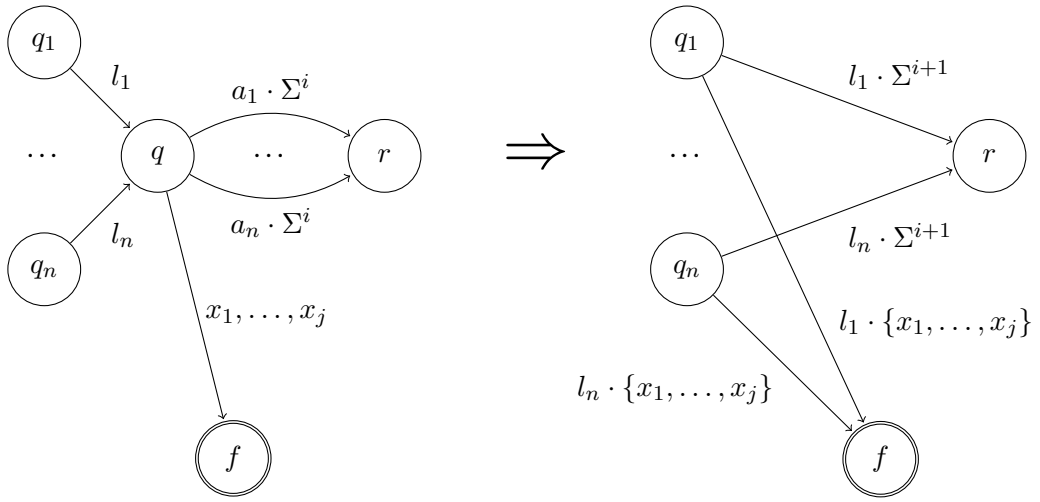


Figure A.4.: Second reduction rules, where $\Sigma = \{a_1, \dots, a_n\}$ and all l_k and x_k are of the form $a \cdot \Sigma^j$ ($a \in \Sigma$, $i, j \in \mathbb{N}$).

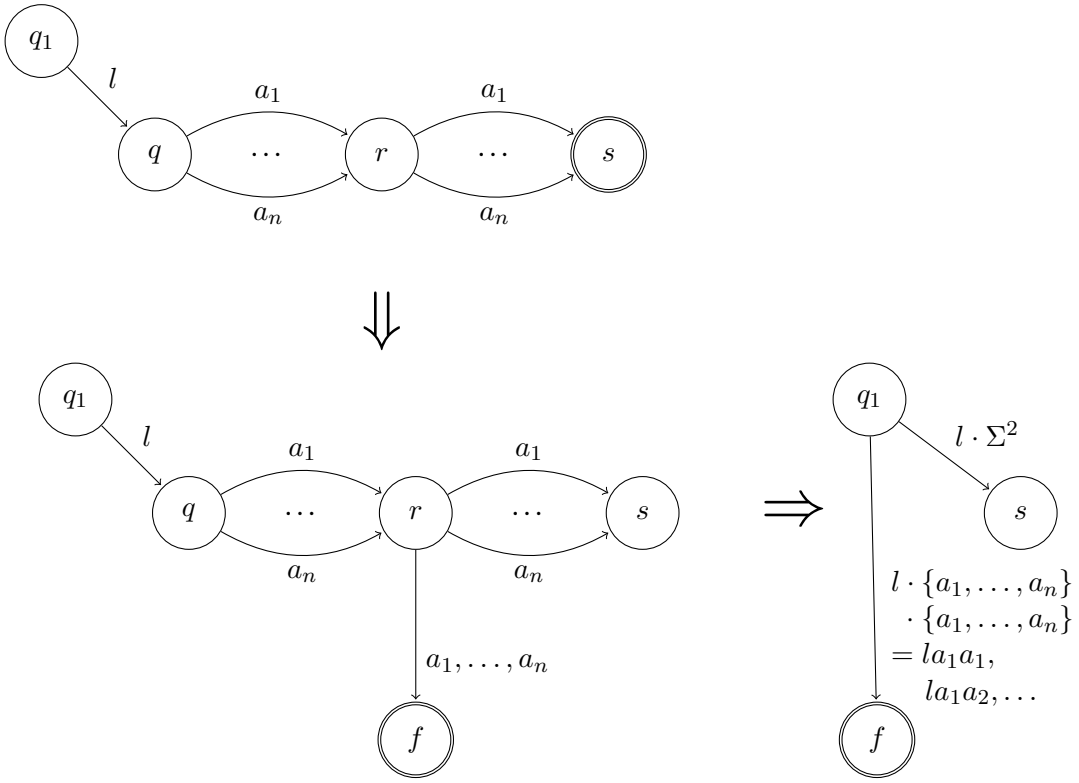


Figure A.5.: arzFA problem: After applying the reduction rule, there are too many transition to the final state (i.e. $|\Sigma^2|$ many). They could be expressed by a single transition $(q_1, l \cdot \Sigma^2, f)$.

A.1. Operations

The membership algorithm has to take into account the slight indeterminacy already mentioned. This can be implemented easily. Assume one has a word aw, is in state q and the unique transition to a non-final state is $(q, a \cdot \Sigma^i, r)$. If the remaining word is longer than $i + 1$, the algorithm takes the unique transition, otherwise the word is accepted iff there is a transition $(q, a \cdot \Sigma^j, r)$, where $j = |w|$.

The complement algorithm should profit greatly from the modified definition of the reduction rule: since final states can be reduced as well, complementing does not require expanding any z-transition. Instead, when there is a transition from q to r under $a \cdot \Sigma^i$, the set of transitions from q to f is complemented (with respect to the set of zips starting with a and having a length up to i): if, for example, $(q, a \cdot \Sigma^3, r), (q, a \cdot \Sigma^2, f) \in \delta$, then in the complement automaton the latter gets replaced by (q, a, f), $(q, a \cdot \Sigma, f)$ and $(q, a \cdot \Sigma^3, f)$.

The BinOp algorithm works similar to that for zDFAs. However, all resulting states except for a single final state f are non-final. If the resulting automaton is in state $[q_1, q_2]$, then a transition $([q_1, q_2], a \cdot \Sigma^i, f)$ is added iff $(q_1, a \cdot \Sigma^i, f_1) \in \delta_1 \odot (q_2, a \cdot \Sigma^i, f_2) \in \delta_2$, where $\odot$ is the binary operation.

The minimization algorithm is probably at least as complicated as for zDFAs, since it has the same problem: pseudo-states have to be added, just to check whether the existing blocks can be split. However, the indirect minimization via DFAs should work equally well.

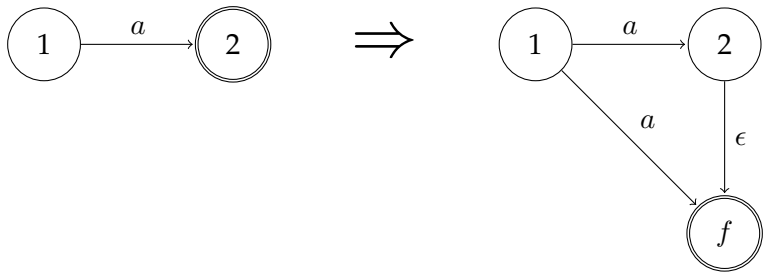


Figure A.6.: Problems persist even after adding ϵ-transitions.

A.2. Evaluation

This alternative definition seems like a viable option for implementing z-automata. The complement algorithm is much faster, the other algorithms are comparably fast. It is possible to minimize even more states (i.e. the final states). However, this approach has a major deficit: It cannot represent languages that accept the empty word ϵ. In the transformation of a DFA to an arzFA, all incoming transitions of final states are considered. The case that the automaton starts from that final state is not considered.

$(q', x, r) \in \delta$, for all $x \in \Sigma$. Because *ab* leads from q to r and f, r was final, and since q leads to r also under *ax* for all $x \in \Sigma$, there is a transition $a \cdot \Sigma$ from q to f.

This problem could be fixed by allowing ϵ-transitions from states to the unique final state. However, this introduces another problem:

Now there a multiple paths the *membership* algorithm could take. In addition, there are problems when one tries to apply the concept of master automata on arzFAs. In the master automaton, one would like to eliminate states like state 2 in Figure A.6, so that the projection to a minimal state yields the minimal automaton. However, eliminating states from the master automaton that are final and have multiple transitions makes it impossible to represent the languages that they recognize.

These problems with arzFAs seem to justify the focus on the definition in Section 2.2 of this work. Nevertheless, it is worthwhile to explore other possible options of representing reduced automata.

B. User manual for JFLAP-Z

JFLAP-Z is based on the popular automata and learning tool JFLAP, which stands for "Java Formal Languages and Automata Package".[1] With JFLAP-Z, this tool has been modified to include finite z-automata, i.e. deterministic machines with $a \cdot \Sigma^i$ transitions. Beside zDFAs, master automata (rMA, rMZ) are also supported. So instead of just a single initial state, automata can have multiple initial states.

This data structure implementation comes with implementation of algorithms for some common problems, such as complement, binary operation, minimization and merging minimal automata.

The other components of JFLAP should be largely unaffected. However, this cannot be guaranteed and this user manual will only focus on the newly implemented features of JFLAP-Z. JFLAP-Z is based on JFLAP 7.0, which was first released in 2009.

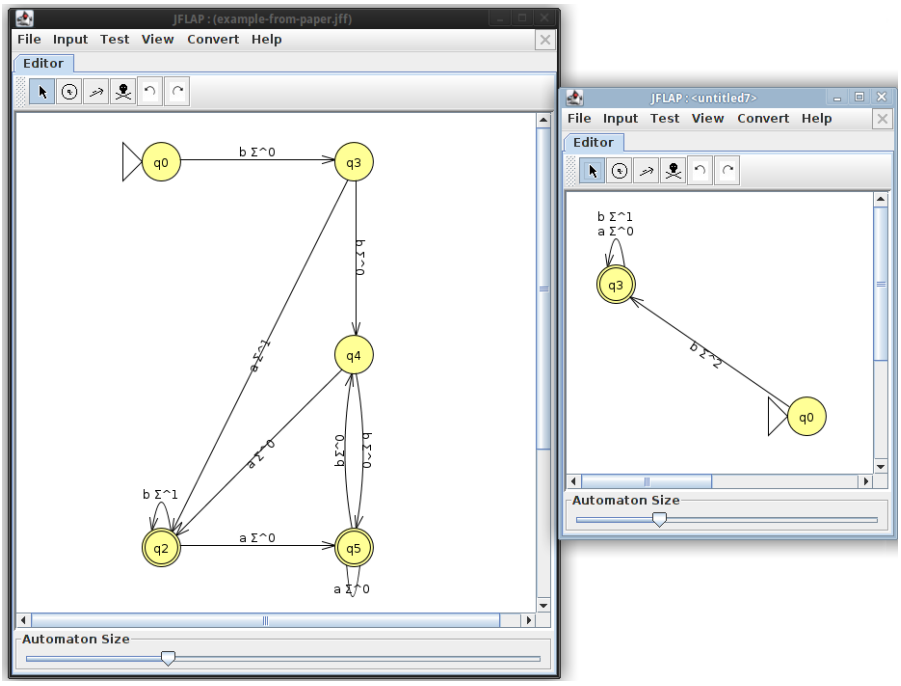


Figure B.1.: Example run of the (indirect) minimization algorithm. On the left, the original z-automaton. On the right, the minimal zDFA.

[1] See `http://www.jflap.org` for more information on JFLAP.

B.1. Adding and manipulating z-automata

To create a new z-automaton, one has to click "Finite Z-Automaton" in the main menu of the program. Alternatively, it is possible to convert a given DFA into a zDFA.

Adding states and manipulating the automaton is similar to that of DFAs. However, when adding a transition, one not only has to specify a letter, but also an integer, i.e. the zip length.

Since it is possible to represent a master automaton with this data structure, one can select more than one state as initial state. When using testing if an input is accepted by the automaton (e.g. *Input* $\rightarrow$ *Multiple Run*), one of the initial states has to be selected.

It is possible to convert a given zDFA back to a DFA (*Convert* $\rightarrow$ *Convert zDFA to DFA*). This expands all z-transitions. Finite automata in JFLAP-Z have been modified to also be able of having more than one initial state and thus being master automata.

The alphabet of an automaton is given implicitly by the transitions present. It is therefor not required to specify it. However, on operations that require a common alphabet, the original automaton is interpreted to have the same alphabet as both automata combined. For example, if you have an automaton with a single transition $a \cdot \Sigma$, then $\Sigma = \{a\}$. But, if it is combined with an automaton that has $\Sigma = \{a, b\}$, then this alphabet also applies to the first transition (so $a \cdot \Sigma$ stands not only for *aa*, but also for *ab*).

It should be noted that all operations are only supported for deterministic machines. Attempting to introduce indeterminacy may yield faulty results. Also, as discussed in the paper, only automata with kernels as initial states can be fully reduced.

B.2. Complement

After applying the complement operation on an automaton (Convert $\rightarrow$ Complement z-automaton), one gets an automaton whose state names are the same as for the original automaton.[2] But, since "pseudo" states like $\Sigma^3 \cdot 2$ also have to be added, multiple states can have the same name. The number in an additional label below the circle of the automaton indicates what the zip of the state is (so with $\Sigma^3 \cdot 2$ this label would be 3), if it is a pseudo-state. Usually, a trap state has to be added to which a new number is assigned.

This operation yields a zDFA that is reduced, but might be non-minimal.

B.3. Binary operation

The binary operation (Convert $\rightarrow$ Binary operation) can be applied to a pair of z-automata. After selecting a second automaton for the operation, one can choose from a number of common binary operations, such as AND, OR, IMPLICATION. For each automaton one has to select one initial state, if there are more than one.

[2] In some cases the ID of a state differs from the name (e.g. after applying some operation), in which case it cannot be guaranteed that the new automaton has the same state names as the original automaton.

Figure B.2.: Resulting state of the BinOp operation. This indicates that the first automaton is in the pseudo-state $\Sigma \cdot 1$ and the second automaton in $\Sigma^3 \cdot 2$.

The resulting automaton also uses the state numbers of the original automata, as described in Figure B.2. Note that usually trap states are created for each automaton which to which a new number is assigned.

This operation yields a zDFA that is reduced, but might be non-minimal.

B.4. Minimization

The *Minimize zDFA (indirectly)* operation (accessible from the *Convert* menu) is an implementation of the simple indirect minimization algorithm for zDFAs. All of the z-transitions will be expanded, then the automaton is minimized and the reduction rule is exhaustively applied.

This function does not require user interaction.

B.5. Merging

Merging required two fully-minimized and fully-reduced z-automata – otherwise it will produce an incorrect result. The procedure will first open a tree interface to enable interactive partitioning by the user. By clicking on the root node of the tree and then "Complete Subtree" the partition is computed automatically. After confirming the tree with "Finish", one gets to a view where one can manually add edges to the new minimal automaton. The button "Complete" enables the user to take the shortcut and after clicking "Done?" and confirming, the final minimal automaton is presented.

Note that each block is not only split according to the different blocks transitions lead to, but also according to different zip lengths.

B.6. Projection

The projection operation turns a master automaton into a regular automaton with a unique initial state. The user can select an initial state an the result is the automaton with all states unreachable from this initial state removed. This is useful to get a smaller automaton from a large master automaton when one has only to perform an operation on one language in the master automaton. Since operations like complement yield non-minimal automata, it is advisable to project, apply the operation, minimize and add the result to the master automaton instead of complementing the whole master automaton.