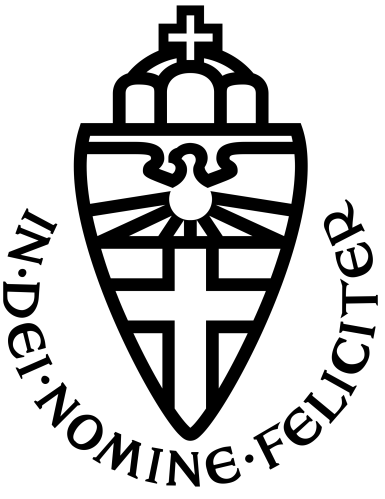
2-Way Finite Automata

Radboud University, Nijmegen



Writer: Serena Rietbergen, s4182804
Supervisor: Herman Geuvers

Academic Year 2017-2018

Contents

Chapter 1

Introduction

Automata are in its most pure form an elegant mechanism for deciding whether a word belongs to a specific language. When picturing a language through an automaton, it makes it easier to understand what the language actually is. We can, after all, pick a certain input and follow its path. It will either get accepted or rejected. There are however a few problems when actually creating an automaton. One can always simply create a language that one finds interesting. The real challenge lies in creating an actual automaton that (optimally) represents the chosen language.

We have seen a few methods of simplifying this process. Among other options, one can use a "Non-Deterministic Finite Automaton(NFA)" or a "NFA-λ". All these constructs are used so that we can better understand how some automata work. When viewing certain Finite Automata that are not Deterministic Finite Automata, one may get the impression that when adding these options we can accept more languages. However, this is not true. It has been shown that the class of languages that NFAs or NFA-λs accept are not different from the class of languages that simple DFAs can accept. There are algorithms for transforming a NFA to an equivalent DFA, a NFA-λ to an equivalent NFA, which in turn can be transformed into an equivalent DFA again.

All these variations help us to better understand the concept of Finite Automata. An important issue is the number of states an automaton uses. For example, when we have a NFA with few states we can obtain a DFA with relatively many states by exercising the method for transforming one. When one sees the DFA, one could have a harder time imagining which language it accepts.

In this thesis, we shall study another method which is used for optimizing DFA. We will see the beauty of a Two-Way Finite Automaton, either Deterministic or Nondeterministic. They were introduced by Rabin and Scott[1] and Shepherdson[2] in 1959. Once again, these two-way automata do not imply that these automata can accept more or different classes of languages. You see, the classes of languages that can be accepted by two-way finite automata, can also be accepted by one-way finite automata. According to Shepherdson, transforming a two-way finite automaton to a one-way deterministic finite automaton can be done in $\mathcal{O}((n+1)^{n+1})$ steps where n is the number of states of the two-way finite automaton.

DFAs and NFAs can only walk in one direction, while two-way automata can walk multiple times through the word by going back and forth. DFAs and NFAs, when accepting a difficult language, can have a number of states. This can be reduced this by allowing the automaton to walk back and forth through the word. By doing this, you also have a better overview of the automaton, making them easier to understand.

Even though there is a basic definition for a two-way deterministic finite automaton, there have been slight adaptations made to this definition. These alterations were used for understanding these automata better. Even though these changes made the automata look different, they still accept the same classes of languages.

In this thesis we will see how to turn a two-way automaton, both deterministic as non-deterministic, into a one-way deterministic automaton and what the costs are of that process in number of states. As previously stated, a two-way automaton is more likely to have less states. We will also see what other methods of converting a two-way automaton into a one-way automaton are solely used for unary languages. That is, languages over a one-symbol alphabet.

Chapter 2

One way automata, deterministic and non-deterministic

A finite automaton is a simple machine to recognize a pattern. More precisely, a pattern of words over an alphabet.

Definition 2.1. An *alphabet* Σ is a finite set of symbols.

Definition 2.2. A *word* $w \in \Sigma^*$ is a concatenation of letters from the alphabet Σ with any length.

Definition 2.3. We will denote the *empty word* as λ, it has length 0.

Definition 2.4. An *language* L is a set of words, $L \subseteq \Sigma^*$

Definition 2.5. A *deterministic finite automaton* is a quintuple $M = (Q, \Sigma, \delta, s, F)$ where:

- Q is a finite set of states;
- Σ is a finite set that we call the input alphabet;
- $\delta : Q \times \Sigma \to Q$ is the so called transition function;
- $s \in Q$ is the start state;
- $F \subseteq Q$, the set of final states.

The automaton that is depicted below accepts the language where every word needs to have an odd length. By alternating between the two states, it is effectively counting the letters in the word. When the length of the word is even, it will end in q_0 and therefore not be accepted.

Remark 2.6. In an automaton the start-state is depicted with an arrow coming from "start". The final states in an automaton are depicted with a double ring around the state.

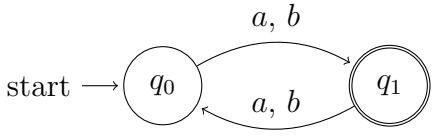


Figure 2.1: A DFA

When reading a word $w \in \Sigma^*$ in a certain DFA, we often would like to know in which state we end up after n steps when beginning at the begin-state q_0. For this we will use the following definition:

Definition 2.7. Let $M = \{Q, \Sigma, \delta, q_0, F\}$ be a DFA, $w \in \Sigma^*$, $n \in \mathbb{N}$ and $q \in Q$. We define the relation $q_0 \xrightarrow[w]{n} q$ inductively for $n > 0$:

- $q_0 \xrightarrow[w]{1} q$ if $\delta(q_0, k) = q$ with k being the first letter of w; and
- $q_0 \xrightarrow[w]{n} q$ if $\exists q' \in Q : q_0 \xrightarrow[w']{n-1} q'$ and $q' \xrightarrow[a]{1} q$ with $w = w'a$.

Definition 2.8. Let $M = \{Q, \Sigma, \delta, q_0, F\}$ be a DFA, $w \in \Sigma^*$ and n the length of w. If $q_0 \xrightarrow[w]{n} q$, it is said that w has a *run on the DFA* M starting in q_0 and ending in q. A *run* is *accepting* if $q \in F$.

Definition 2.9. Let $M = \{Q, \Sigma, \delta, q_0, F\}$ be a DFA and $w = a_1 a_2 \dots a_n \in \Sigma^*$.

1. w is accepted if it has an accepting run on M.
2. w is rejected if it doesn't have an accepting run on M.

The set of words accepted by the DFA M is called $\mathcal{L}(M)$.

Definition 2.10. A *non-deterministic finite automaton* is a quintuple $M = (Q, \Sigma, \delta, s, F)$ where:

- Q is a finite set of states;
- Σ is a finite set that we call the input alphabet;
- $\delta : Q \times \Sigma \to 2^Q$ is the so called transition function;
- $s \in Q$ is the start state;
- $F \subseteq Q$, the set of final states.

The automaton as seen below accepts only the language where the last letter of a word is an 'a'. It walks through the word and when encountering an 'a' the automaton can choose whether it stays in the loop of q_0 or goes to q_1, but will only accept it when the automaton is done reading the whole word.

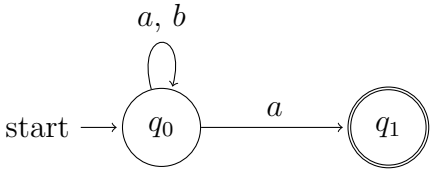


Figure 2.2: Example of a NFA

Definition 2.11. Let $M = \{Q, \Sigma, \delta, q_0, F\}$ be a NFA, $w \in \Sigma^*$ and n the length of w. If $q_0 \xrightarrow[w]{n} q$, it is said that w has a *run on the NFA* M starting in q_0 and ending in q. A *run* is *accepting* if the end-state is a final state. Talking about a NFA, a word can have multiple runs.

The acceptance of a word w in a NFA is the same as the acceptance of a DFA:

Definition 2.12. Let $M = \{Q, \Sigma, \delta, q_0, F\}$ be a DFA and $w = a_1 a_2 \dots a_n \in \Sigma^*$.

1. w is accepted if it has an accepting run on M.
2. w is rejected if it doesn't have an accepting run on M.

The set of words accepted by the NFA M is called $\mathcal{L}(M)$.

Definition 2.13. When we view letters as words of length 1, we can view $\hat{\delta}$ as an extension of δ for words of arbitrary length. We will define $\hat{\delta}(q, y)$ inductively with $y \in \Sigma^*$ as follows:

$$\begin{aligned} \hat{\delta}(q, \lambda) &= q \\ \hat{\delta}(q, ya) &= \delta(\hat{\delta}(q, y), a) \end{aligned}$$

Chapter 3

Overhead for converting NFA to DFA

We now consider the costs when transforming a NFA to a DFA.

Definition 3.1. $I_n = L((a+b)^*a(a+b)^{n-1})$ for $n \in \mathbb{N}$.

For $n \in \mathbb{N}$, I_n is the language of words where the n'th letter from the end is an 'a'. The class of languages is extremely useful when giving an example of transforming a NFA to a DFA. For $(I_n)_{n\in\mathbb{N}}$ we have an elegant NFA, one that simply walks until the wanted 'a' is read, and then reads the next n letters. This automaton blows up exponentially in number of states when it is transformed to a DFA. This class is the prime example as to why the complexity of transforming a NFA to a DFA is in $\mathcal{O}(2^n)$.

Lemma 3.2. *Let $\mathbb{A}$ be a DFA that accepts I_n. If $v, w \in \Sigma^n$, $q_0 \xrightarrow[w]{n} q$ and $q_0 \xrightarrow[v]{n} q$, then $v = w$.*

Proof. Let $\mathbb{A} = (Q, \Sigma, \delta, q_0, F)$. Assume $v \neq w$, $q_0 \xrightarrow[v]{n} q$ and $q_0 \xrightarrow[w]{n} q$. Assume the $(n-k+1)^{\text{th}}$ letter of w and v is 'a' and 'b' respectively. Define $v' = vb^{n-k}$ and $w' = wb^{n-k}$. We see that there is a unique $q' \in Q$ such that $q \xrightarrow[v']{n-k} q'$ and $q \xrightarrow[w']{n-k} q'$. Now we see that there are two possibilities:

1. $q' \in F$. This means that both w' and v' are accepted. However $w' \notin \mathcal{L}(\mathbb{A})$, because w' must not be accepted by this machine. This is because the n^{th} letter from the end of w' is a 'b' and not a 'a'.↯

2. $q' \in Q \backslash F$. This signifies that both v' and w' are not accepted. However $v' \in \mathcal{L}(\mathbb{A})$, because v' must be accepted by this machine. This is because the n^{th} letter from the end for v' is a 'a'. ↯

Therefore $v = w$. □

Theorem 3.3. *Let $\mathbb{A}$ be a DFA that accepts I_n, then $|Q| \geq 2^n$.*

Proof. For $\Sigma = \{a, b\}$ there are 2^n different words of length n. After reading a word of length n, $\mathbb{A}$ is in a unique state.
Assume you have less than 2^n states, then there are two words that will end in the same state according to the Pigeonhole Principle. According to the Lemma 3.2. these two words are the same. ↯
Hence at least 2^n states are necessary. □

Chapter 4

Two way automata, deterministic and non-deterministic

Two-Way Finite Automata are used for efficiently accepting a language. There are multiple ways to denote the exact definition of a two-way deterministic finite automaton.

First is the variant Kozen uses in his lecture notes[3]. This variant is interesting, because it uses endmarkers, whereas Rabin and Scott[1] did not. The reason why is because it gives us a better representation about what we actually would like to accomplish. For example, take the following language.

$$I_3 = \{w \in \Sigma^* | \text{ the third letter from the back is '}a\text{'}\}$$

When an automaton accepts this language, we want to know if the third letter from the end is an 'a' or not. One can simply achieve this by going to the end of the word and then walk back. To do this, we would need to know when a word is finished. For this Kozen introduces endmarkers.

Definition 4.1 (Kozen[3])**.** A *Kozen-2DFA* is an octuple $M = (Q, \Sigma, \vdash, \dashv, \delta, s, t, r)$ where:

- Q is a finite set of states;
- Σ is a finite set that we call the input alphabet;
- $\vdash$ is the left endmarker, $\vdash \notin \Sigma$;
- $\dashv$ is the right endmarker, $\dashv \notin \Sigma$;
- $\delta : Q \times (\Sigma \cup \{\vdash, \dashv\}) \to (Q \times \{L, R\})$ is the transition function (L, R stand for left and right, respectively);
- $s \in Q$ is the initial state;
- $t \in Q$ is the accept state;
- $r \in Q$ is the reject state, $r \neq t$;

such that for all states q,

$$\delta(q, \vdash) = (u, R) \text{ for some } u \in Q$$
$$\delta(q, \dashv) = (v, L) \text{ for some } v \in Q,$$

and for all symbols $b \in \Sigma \cup \{\vdash\}$,

$$\delta(t, b) = (t, R), \qquad \delta(r, b) = (r, R),$$
$$\delta(t, \dashv) = (t, L), \qquad \delta(r, \dashv) = (r, L).$$

When reading a word $w \in \Sigma^*$ in a Kozen-2DFA, we often would like to know in which state we end up after n steps when beginning at the initial state q_0. For this we will use the following definitions:

Definition 4.2. Let $M = (Q, \Sigma, \vdash, \dashv, \delta, s, t, r)$ be a Kozen-2DFA and $w = a_1 a_2 \dots a_n$. The relation $(q, i) \xrightarrow[w]{1} (q', i)$ with $q, q' \in Q$ and $i, j \in \mathbb{N}$ is defined on a 2DFA M as follows:

$$\delta(p, a_i) = (q, L) \Rightarrow (p, i) \xrightarrow[w]{1} (q, i-1)$$
$$\delta(p, a_i) = (q, R) \Rightarrow (p, i) \xrightarrow[w]{1} (q, i+1)$$

Here a_i is the i^{th} letter of the word $w \in \Sigma^*$, $p, q \in Q$.

Definition 4.3. We define the relation $\xrightarrow[w]{n}$ inductively on $n \geq 1$ as follows:

- $(p, i) \xrightarrow[w]{0} (p, i)$; and
- $(p, i) \xrightarrow[w]{n} (u, k)$, if $\exists q \in Q : (p, i) \xrightarrow[w]{n-1} (q, j)$ and $(q, j) \xrightarrow[w]{1} (u, k)$ for some $p \in Q$ and $j \in \mathbb{N}$.

Definition 4.4. We define $\xrightarrow[w]{*}$ as follows:

$$(p, i) \xrightarrow[w]{*} (q, j) \overset{\text{def}}{\Leftrightarrow} \exists n \geq 0 : (p, i) \xrightarrow[w]{n} (q, j)$$

Definition 4.5. There are three possibilities:

1. A word $w \in \Sigma^*$ is said to be *accepted* if $(s, 0) \xrightarrow[w]{*} (t, i)$ for some $i \in \mathbb{N}$.
2. A word w is said to be *rejected* when $(s, 0) \xrightarrow[w]{*} (r, i)$ for some $i \in \mathbb{N}$
3. A word w it also said to be *rejected* when none of (1) or (2) holds.

$L(M) \subseteq \Sigma^*$ is the set of words that are accepted by a Kozen-2DFA M.

Note that w cannot be both accepted and rejected at the same time, because of the construction of Kozen. When a word enters r or t, it can never leave that state.

The example below accepts only words where the number of 'a's in w is a multiple of three and the number of 'b' is a multiple of two. It first checks if the number

of 'a's is correct when walking through the word by going repeatedly right. Only when the automaton ends in q_0 while being at the end of the word, it will read the end-marker and go left through the word while checking if the number of 'b's is a multiple of 2. Only when that is the case, the automaton will end in p_0 and will read the end-marker. Then it will go to t, the "terminating state". Let's say the word we want to check does not have the correct amount of 'a's, it will either end in q_1 or q_2. Then the automaton will read $\dashv$ and go to r, the "rejecting state".

Example 4.6. The 2DFA as seen below represents the language over $\Sigma = \{a, b\}$, namely $\mathbb{L} = \{w \in \Sigma^* | \#a(w) = 3k \text{ and } \#b(w) = 2i \text{ with } k, i \in \mathbb{N}\}$.

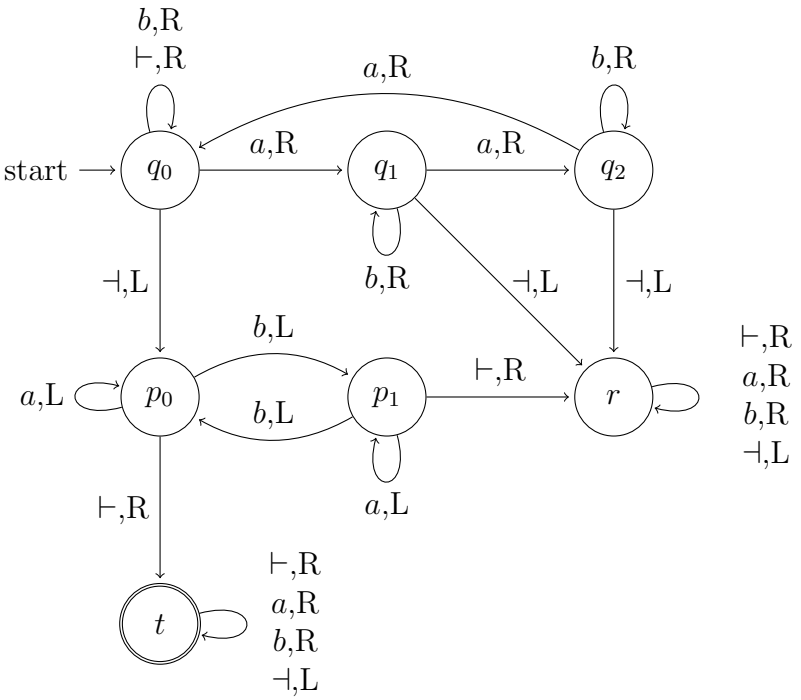


Figure 4.1: A 2DFA by Dexter Kozen

Remark 4.7. An automaton is deterministic when every state has a transition arrow for every letter in the language Σ, $\vdash$ and $\dashv$. Note that not every state has 4 transition arrows in this example. A lot of those are omitted, because those situations will never be reached.

Definition 4.8 (Shepherdson[2])**.** A two-way deterministic finite automaton (Shepherdson-2DFA) over Σ is a system $M = (Q, \delta, s, F)$ where:

- Q, the final set of states;
- $\delta \colon \Sigma \times Q \to L \times Q$, the transition function with $L = \{-1, 0, 1\}$ the direction;
- s, the initial state and
- F, the set of final states.

If for all $q \in Q$ and $a \in \Sigma$, we have that $|\delta(q, a)| \leq 1$, then M is said to be deterministic.

Definition 4.9. A word $w \in \Sigma^*$ in a Shepherdson-2DFA is said to be:

- *accepted* if M is done reading w and it moves off the word at the right-hand-edge of w while M is in a state belonging to F.
- *rejected* in any other case.

The set of words that are accepted by M is called $\mathcal{L}(M)$.

Example 4.10. The automaton accepts the language $L((a|ab)^*)$. Note that this automaton is deterministic, because $|\delta(q, a)| \leq 1$ holds for all $q \in Q$ and $a \in \Sigma$.

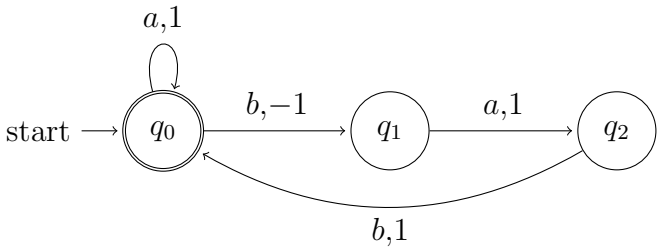


Figure 4.2: A Shepherdson-2DFA accepting $L((a|ab)^*)$

We have now only seen examples of Two-Way Deterministic Finite Automata. A Non-Deterministic variant is also possible. To make a 2NFA one would need to make a few adjustments:

- The transition function needs to be in the form of $\delta\colon \Sigma \times Q \to 2^{Q \times L}$. Instead of δ being a function that gives every pair (a, q) a single value in the form of (q, d), it will now give a set as answer. This needs to happen because in a 2NFA it is possible for an automaton to move to different states when reading a letter, or to no states at all.
- It is possible, but not necessary, to give the 2NFA the option of having multiple initial states.

For an exact definition of an variant of a two-way non-deterministic finite automaton, see definition 6.3.

Chapter 5

Converting 2DFA to 1DFA

5.1 The algorithm

For this part we will use the Two-Way Deterministic Finite Automaton as described by a Kozen-2DFA in Definition 4.1.

To construct a 1DFA M' from a 2DFA M we need to find the set of states, which will be found through the creation T_x for $x \in \Sigma^*$. To every x we associate a function $T_x \colon (Q \cup \{\bullet\}) \to (Q \cup \{\perp\})$, which will be later shown in the form of a table.
To construct T_x we first need to introduce the symbols $\bullet$ and $\perp$. Throughout this construction let $y \in \Sigma^*$ be the concatenation of $x \in \Sigma^*$ and $z \in \Sigma^*$. On this input the machine M will start reading the word y on the left side and will move through it from left to right. Therefore, it may cross the border from x to z. Here we say that M crosses the border from x to z if the last letter that is read is the last letter of x and the next letter that will be read is the first letter of z.
Note that there are two possibilities, M either crosses the border from x to z or it doesn't:

1. M crosses the border from x to z. It does so for the **first time** while going from $q \in Q$ (the last letter of x is read) to $q' \in Q$ (the first letter of z is read). This will be denoted as $T_x(\bullet) = q$.

2. M does not cross the border. This will be denoted as $T_x(\bullet) = \perp$.

Let's assume that the machine does cross the border from x to z and enters into z. Similarly to the above, M will either cross the border again from z to x or it will not. Say it does cross the border between z and x and enters back into x, entering into q_i (the first letter from z is read). We can repeat the distinction from above:

1. M crosses the border again from x to z in some state $q_j \in Q$. This will be denoted as $T_x(q_i) = q_j$.

2. M never crosses the border between x and z again. This will be denoted as $T_x(q_i) = \perp$.

Notice that after crossing the border into x from z in state q_i, the state q_j in which it emerges only depends on x and q_i and not on z, motivating the used notation.

The explicit construction of making a 1DFA from a 2DFA consists of a few steps. The largest step in the algorithm is determining all function T_x, which will be represented by tables. This can be done in various ways. Since there are finitely many tables possible, namely $(n+1)^{n+1}$ with n being the number of states in Q, one can construct all tables and see how they interact with one another when constructing the transition function. Seeing that a lot of the tables will never be reached, this is a highly inefficient method. Consider at the following algorithm.

Definition 5.1. This algorithm lets us construct all T_x, starting at step 1.

1. Let $w = \lambda$.
2. Construct T_w as stated above.
3. If $T_w(\bullet) = \perp$ or T_w already exists, stop at this branch.
   Else, save T_w. For every $a \in \Sigma$ let $w := wa$ and repeat from step 2.

Definition 5.2. Given the 2DFA $M = (Q, \Sigma, \vdash, \dashv, \delta, s, t, r)$ we define the 1DFA $M' = (Q', \Sigma, s', \delta', F')$ as follows:

- $Q' = \{T_x | x \in \Sigma^*\}$, a finite set of states;
- $s' = T_\lambda$, the start-state;
- $\delta'(T_x, a) = T_{xa}$, the transition-function; and
- $F' = \{T_x | x \in L(M)\}$, the set of finite states.

Before we are able to prove that the 2DFA M and the 1DFA M' accept the same language, we first state the following lemmas.

Lemma 5.3. *Let $x, y \in \Sigma^*$: $\hat{\delta}'(T_x, y) = T_{xy}$.*

Proof. We will prove this by induction over y.

- Induction base:
  $y = \lambda$. $\hat{\delta}'(T_x, \lambda) \stackrel{def}{=} T_x = T_{x\lambda}$.
- Induction hypotheses:
  $\hat{\delta}'(T_x, y) = T_{xy}$.
- Inductive step:
  $y' = ya$, which will be the concatenation of a word y and a letter $a \in \Sigma$.
  $\hat{\delta}'(T_x, ya) \stackrel{def}{=} \delta'(\hat{\delta}'(T_x, y), a) \stackrel{IH}{=} \delta'(T_{xy}, a) = T_{xya}$.

Since both the base and the inductive step have been performed and are correct, by mathematical induction, $\hat{\delta}'(T_x, y) = T_{xy}$ holds for all $x \in \Sigma^*$. □

Lemma 5.4. *Let $x \in \Sigma^*$: $T_x \in F' \Leftrightarrow x \in L(M)$.*

Proof. $\Leftarrow$: this follows from the definition of F'.
$\Rightarrow$: If $T_x \in F'$, then there exists some $y \in L(M)$ with $T_x = T_y$. Now we prove that x is also an element of $L(M)$: let us look at the sequences of states M finds itself in when crossing the boundary between x and $\dashv$, and y and $\dashv$. These sequences are

identical because they are determined by the identical tables T_x and T_y. We know that $y \in L(M)$, so the sequence of y contains t at some point. This indicates that the sequence of x also contains t at some point, meaning that the word is accepted by M. After all, when the 2DFA M reaches the state t it will never leave this state and thus the word x is accepted. Therefore, $x \in L(M)$. □

We can now finally show that the Kozen-2DFA M and the 1DFA M' accept the same languages. This proves that when a Kozen-2DFA accepts a certain language, a 1DFA can be created that accepts the same language. It is merely another method of describing an automaton that accepts said language.

Theorem 5.5. $L(M) = L(M')$

Proof.

$$\begin{aligned} x \in L(M') &\Leftrightarrow \hat{\delta}'(s', x) \in F' \\ &\Leftrightarrow \hat{\delta}'(T_\lambda, x) \in F' \\ &\Leftrightarrow T_x \in F' \\ &\Leftrightarrow x \in L(M) \end{aligned}$$

Now we see that $w \in L(M) \Leftrightarrow w \in L(M')$ and therefore $L(M) = L(M')$. □

5.2 Examples

This algorithm gives an upper bound of $\mathcal{O}((n+1)^{(n+1)})$, because there are $(n+1)^{(n+1)}$ tables as mentioned in Section 5.1 Here is an elegant example and a not so elegant example, in order for the reader to get an idea of how this algorithm works.

The algorithm essentially consists of constructing the individual elements of a 1DFA.

5.2.1 Example 1

We will now see a fairly elegant example in which Kozen's algorithm is used. This automaton does not have a blow-up in the number of states.

$\mathbb{L} = \{w \in \Sigma^* | \#a(w) = 3k \text{ and } \#b(w) = 2i \text{ with } i, k \in \mathbb{N}\}$ with $\Sigma = \{a, b\}$.

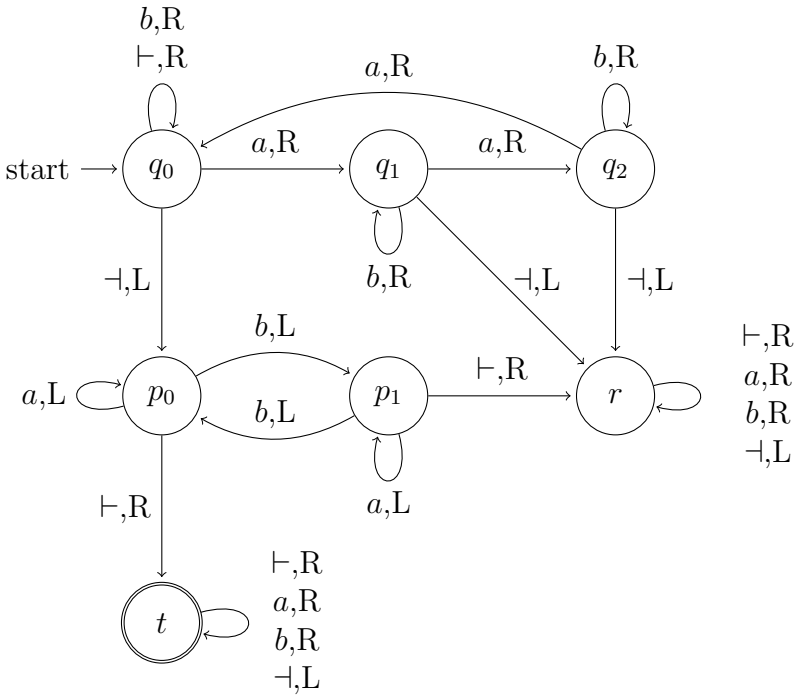


Figure 5.1: A 2DFA accepting $\mathbb{L}$

We can now compute all different T_x.

T_{aaa}	
•	q_0
p_0	t
p_1	r
u_0	

T_{aa}	
•	q_2
p_0	t
p_1	r
u_1	

T_a	
•	q_1
p_0	t
p_1	r
u_2	

T_b	
•	q_0
p_0	r
p_1	t
u_3	

T_{ab}	
•	q_1
p_0	r
p_1	t
u_4	

T_{aab}	
•	q_2
p_0	r
p_1	t
u_5	

Note that the following always holds for this automaton:

T_x	
q_0	q_0
q_1	q_1
q_2	q_2
r	r
t	t

States that are the same for every T_x will be left out in the examples that follow. T_{aaa} can be read as follows:

- $T_{aaa}(\bullet) = q_0$
  When reading aaa starting from the left starting in q_0, we see that after reading the first a the automaton goes to q_1. After reading the second a the automaton

goes to q_2. After reading the third a the automaton goes to q_0 again. So after reading aaa, the automaton ends in q_0. Note that whether an a,b,$\vdash$ or $\dashv$ follows, the automaton will leave the state q_0. Therefore $T_{aaa}(\bullet) = q_0$.

- $T_{aaa}(q_0) = q_0$
  We start reading aaa starting from the right in q_0. After reading the a, we go to q_1. So the state we left before crossing the border is q_0. Therefore $T_{aaa}(q_0) = q_0$.

- $T_{aaa}(q_1) = q_1$
  We start reading aaa starting from the right in q_1. After reading the a, we go to q_2. So the state we left before crossing the border is q_1. Therefore $T_{aaa}(q_1) = q_1$.

- $T_{aaa}(q_2) = q_2$
  We start reading aaa starting from the right in q_2. After reading the a, we go to q_0. So the state we left before crossing the border is q_2. Therefore $T_{aaa}(q_2) = q_2$.

- $T_{aaa}(p_0) = t$
  Now we will start reading from the right side of aaa starting in q_0. We read the "third" a and go to p_0. We read the "second" a and go to p_0. We read the first a and go to q_0. Then we read the $\vdash$ and go to t. In state state we will read the three a's, before we cross the rest of the word, all while staying in t. Therefore, $T_{aaa}(p_0) = t$.

- $T_{aaa}(p_0) = r$
  Now we will start reading from the right side of aaa starting in q_1. We read the "third" a and go to p_1. We read the "second" a and go to p_1. We read the first a and go to q_1. Then we read the $\vdash$ and go to r. In state state we will read the three a's, before we cross the rest of the word, all while staying in r. Therefore, $T_{aaa}(p_0) = r$.

- $T_{aaa}(t) = t$
  We start reading aaa starting from the right in t. After reading the t, we go to t. So the state we left before crossing the border is t. Therefore $T_{aaa}(t) = t$.

- $T_{aaa}(r) = r$
  We start reading aaa starting from the right in r. After reading the r, we go to r. So the state we left before crossing the border is r. Therefore $T_{aaa}(r) = r$.

To construct the DFA corresponding to the 2DFA we need to take several steps. We have to give the set of states Q', the starting state s', the transitioning function δ' and the set of states that are accepting, F'.

We have already found all different T_x, so Q' is given above.

We see that $s' \stackrel{\text{def}}{=} T_\lambda = T_{aaa} = u_0$.

We see that $\delta'(T_x, a) \stackrel{\text{def}}{=} T_{xa}$. This provides us with the following transition function:

	a	b
u_0	u_2	u_3
u_1	u_0	u_4
u_2	u_1	u_5
u_3	u_5	u_0
u_4	u_3	u_1
u_5	u_4	u_3

Table 5.1: The transition function δ'

Finally we see that $F' \stackrel{\text{def}}{=} \{T_x | x \in L(M)\} = \{T_{aaa}\} = \{u_0\}$, because of all the option as stated above, only $aaa \in L(M)$.
This provides us with the following DFA.

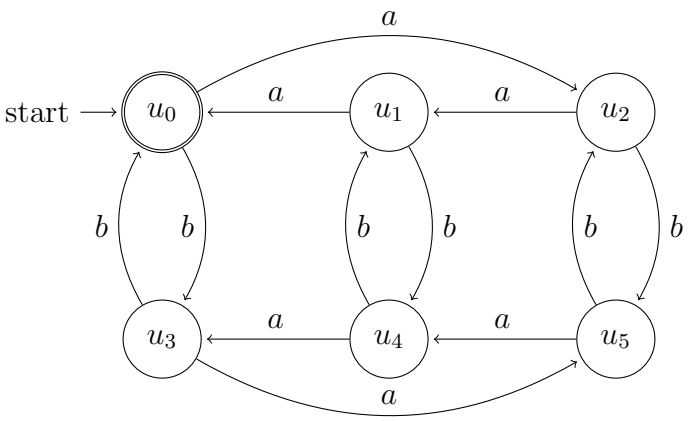


Figure 5.2: The DFA accepting $\mathbb{L}$

5.2.2 Example 2

We will now see an example that isn't quite as elegant as example 1. This example is mainly used to show that the algorithm is indeed capable of a blow-up.

Let us start by considering the language
$I_4 = (a + b)^* a (a + b)^3$.

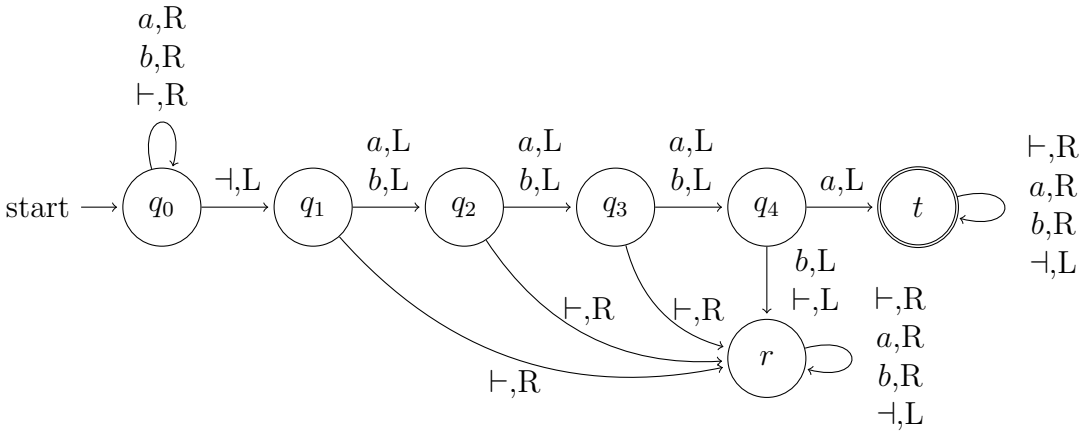


Figure 5.3: A 2DFA accepting the language I_4.

We can now find all T_x:

T_{aaaa}	
•	q_0
q_1	t
q_2	t
q_3	t
q_4	t
u_0	

T_{aaab}	
•	q_0
q_1	t
q_2	t
q_3	t
q_4	r
u_1	

T_{aaba}	
•	q_0
q_1	t
q_2	t
q_3	r
q_4	t
u_2	

T_{aabb}	
•	q_0
q_1	t
q_2	t
q_3	r
q_4	r
u_3	

T_{abaa}	
•	q_0
q_1	t
q_2	r
q_3	t
q_4	t
u_4	

T_{abab}	
•	q_0
q_1	t
q_2	r
q_3	t
q_4	r
u_5	

T_{abba}	
•	q_0
q_1	t
q_2	r
q_3	r
q_4	t
u_6	

T_{abbb}	
•	q_0
q_1	t
q_2	r
q_3	r
q_4	r
u_7	

T_{baaa}	
•	q_0
q_1	r
q_2	t
q_3	t
q_4	t
u_8	

T_{baab}	
•	q_0
q_1	r
q_2	t
q_3	t
q_4	r
u_9	

T_{baba}	
•	q_0
q_1	r
q_2	t
q_3	r
q_4	t
u_{10}	

T_{babb}	
•	q_0
q_1	r
q_2	t
q_3	r
q_4	r
u_{11}	

T_{bbaa}	
•	q_0
q_1	r
q_2	r
q_3	t
q_4	t
u_{12}	

T_{bbab}	
•	q_0
q_1	r
q_2	r
q_3	t
q_4	r
u_{13}	

T_{bbba}	
•	q_0
q_1	r
q_2	r
q_3	r
q_4	t
u_{14}	

T_{bbbb}	
•	q_0
q_1	r
q_2	r
q_3	r
q_4	r
u_{15}	

The set of states of the 1DFA Q' is as stated above.

The starting state is equal to T_λ, which in this case equals u_{15}.

This provides us with the following transition function:

	a	b
u_0	u_0	u_1
u_1	u_2	u_3
u_2	u_4	u_5
u_3	u_6	u_7
u_4	u_8	u_9
u_5	u_{10}	u_{11}
u_6	u_{12}	u_{13}
u_7	u_{14}	u_{15}
u_8	u_0	u_1
u_9	u_2	u_3
u_{10}	u_4	u_5
u_{11}	u_6	u_7
u_{12}	u_8	u_9
u_{13}	u_{10}	u_{11}
u_{14}	u_{12}	u_{13}
u_{15}	u_{14}	u_{15}

Table 5.2: The transition function δ'.

The final thing that rests us to do, is to compute the set of final states: $F' = \{T_x | x \in L(M)\} = \{u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7\}$. This is because $aaaa, aaab, aaba, aabb, abaa, abab, abba, abbb \in L(M)$.

Now that we have computed all the parts of the 1DFA, we can actually construct it. The 1DFA M' is as follows:

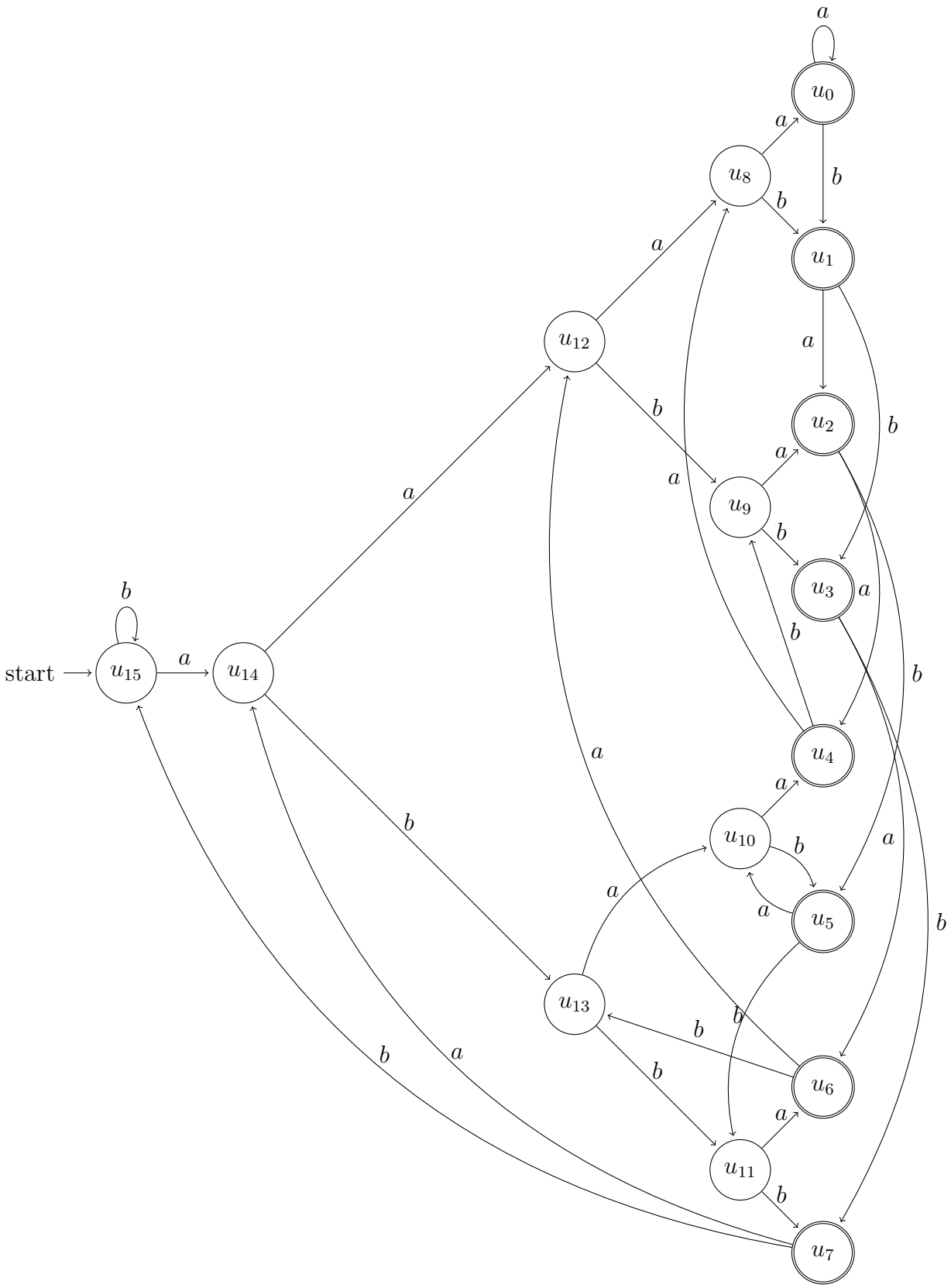


Figure 5.4: A 1DFA accepting the language I_4.

5.3 Complexity

When constructing a 1DFA from a Kozen-2DFA, we see that the number of states equals the number of tables. Therefore, the maximum number of states is bound by the number of possible tables. When we take a look at the function $T_x \colon (Q \cup \{\bullet\}) \to (Q \cup \{\bot\})$, we see that there is a maximum of $(n+1)^{(n+1)}$ tables, with n being the number of states the Kozen-2DFA has. We can now easily see that the upper bound for the complexity of this algorithm is $\mathcal{O}((n+1)^{(n+1)})$. It is not clear whether this algorithm has the lowest complexity, but at least we can show that such an algorithm cannot be polynomial.

Theorem 5.6. *There exists no polynomial algorithm that can convert a general Kozen-2DFA into a 1DFA.*

Proof. An example was given in which a Kozen-2DFA has a linear number of states and the corresponding 1DFA has 2^n states when converting the class of languages I_n. Let's assume a polynomial algorithm is given for converting a Kozen-2DFA into a 1DFA. The created 1DFA will always have at least 2^n states according to Lemma 3.2. Therefore, the algorithm will never be polynomial. □

Chapter 6

Converting 2NFA to 1DFA

To show we can turn a 2NFA into a 1DFA we will use an variation of the algorithm that Shepherdson describes is his paper[2]. We will see that the algorithm Shepherdson uses is basically the same as the algorithm Kozen uses. The reason for not using the algorithm of Kozen when dealing with 2NFA's is because of his usage of end-markers, which his method is based on.

6.1 Kozen versus Shepherdson

We will compare the method Kozen uses for constructing a DFA out of an Kozen-2DFA (see Definition 4.1) with the method Shepherdson uses to construct a DFA from a Shepherdson-2DFA(see Definition 4.8). Note that the method Shepherdson uses shows great similarity to the method Dexter Kozen uses. We will see that Shepherdson mainly uses a different notation. The only difference that really pops out is the construction of the tables that will be the finite set of states in the 1DFA. Kozen uses T_x for this, while Shepherdson uses τ_t.

Definition 6.1. The construction of $\tau_t \colon (Q \cup \{\bullet\}) \to (Q \cup \{\perp\})$ is as follows:

1. For all $t \in \Sigma^*$ with $t \neq \lambda$, τ_t is defined as follows:
   - $\tau_t(q) = q'$ if M, started in the state q while reading the rightmost letter of t, leaves the word t on the right side in state q'.
   - $\tau_t(q) = \perp$ when M leaves t at the left or when it is stuck in a loop, starting in the state q.
   - $\tau_t(\bullet) = q'$ if M, started in the state q_0 while reading the leftmost letter of t, leaves the word t on the right side in state q'.
   - $\tau_t(\bullet) = \perp$ when M leaves t on the left edge or when M is stuck in a loop, starting in the state q_0.
2. τ_λ is defined as follows:
   - $\tau_\lambda(q) = \perp$ and
   - $\tau_\lambda(\bullet) = s$.

The rest of the construction is as follows:

	Shepherdson	Kozen
Q'	$\{\tau_t \mid t \in \Sigma^*\}$	$\{T_x \mid x \in \Sigma^*\}$
F'	$\{\tau_t \mid \tau_t(\bullet) \in F\}$	$\{T_x \mid x \in L(M)\}$
s'	τ_λ	T_λ
δ'	$: \Sigma \times Q \to Q$	$: Q \times \Sigma \to Q$

One can see that most of these attributes are just simply rewritten, the only attribute that looks significantly different is that of F.

To actually show that these two methods are equal, it remains to prove that the set of end-states of the method of Shepherdson equals those of the method of Kozen:

Lemma 6.2. $\{\tau_t \mid \tau_t(\bullet) \in F\} = \{T_x \mid x \in L(M)\}$

Proof. We need to prove that $\tau_t(\bullet) \in F \Leftrightarrow t \in L(M)$.
Both of these statements are equivalent to the following: After M is done reading t it is in some state $q \in F$ □

6.2 2NFAs

We will first need a definition of a Two-Way Non-Deterministic Finite Automaton. For that, we will use the following definition, which is a simple adjustment of the the standard definition Shepherdson uses for a 2DFA.

Definition 6.3. A Shepherdson-2NFA $M = (Q, s, \delta, F)$ over an alphabet Σ is a automaton that holds the following properties:

- Q, the set of states;
- s, the begin-state;
- $\delta \colon \Sigma \times Q \to 2^{Q \times L}$;
- F, the set of final states.

To know whether a word $w \in \Sigma^*$ is accepted by Shepherdson-2FA M, we first need to look at a few useful definitions.

Definition 6.4. A *configuration* of M is a pair $(q, i) \in (Q \times \mathbb{N})$ consisting of a state q and a position i.

Definition 6.5. A *run* is a sequence of configurations. It is an element of $(Q \times \mathbb{N})^*$. Formally, $(q_0, i_0), \ldots, (q_m, i_m)$ is a run of M on a word $w = a_0, \ldots, a_n \in \Sigma^*$ if the following holds:

1. $q_0 = s$, the run starts with the begin state;
2. $i_0 = 0$, the run starts on position 0;
3. $i_m \leq n + 1$, the final configuration does not exceed the position $n + 1$;
4. $\forall j \colon 0 \leq j < m$, we have:

- $\diamond$ $0 \leq i_j \leq n$, except for maybe the final configuration, the run never leaves the bounds of the word;
- $\diamond$ there exists a configuration $(t, k) \in \delta(q_j, a_{i_j})$ such that $s_{j+1} = t$ and $i_{j+1} = i_j + k$.

Definition 6.6. A run is accepting if $i_m = n + 1$ and $q_m \in F$.

Definition 6.7. A word $w \in \Sigma^*$ is said to be:

- accepted by M if it has an accepting run on w;
- rejected by M if it does not have an accepting run on w.

The set of words that are accepted by M is called $L(M)$.

The second requirement to convert a Shepherdson-2NFA into a 1DFA, is an adjustment to the construction of all τ_t with $t \in \Sigma^*$.

Definition 6.8. The function $\tau_t \colon Q \cup \{\bullet\} \to 2^{Q \cup \{\perp\}}$ is constructed as follows:

1. When talking about a state q we see the following
   - $\tau_t(q) = \{q' \mid$ M starts in q on the rightmost letter of t, there is a run where M leaves t at the right in state q'$\}$.
   - $\perp$ is added to $\tau_t(q)$ otherwise. This includes when M leaves t at the left or when M is stuck in a loop.
2. When talking about $\bullet$ we see the following
   - $\tau_t(\bullet) = \{q' \mid$ M starts in q on the leftmost letter of t, there is a run where M leaves t at the right in state q'$\}$.
   - $\perp$ is added to $\tau_t(\bullet)$ otherwise. This includes when M leaves t at the left or when M is stuck in a loop.

Definition 6.9. Given the 2NFA $M = (Q, s, \delta, F)$ we define the 1DFA $M' = (Q', s', \delta', F')$ as follows:

- $Q' = \{\tau_t | t \in \Sigma^*\}$, a finite set of states;
- $s' = \tau_\lambda$, the start-state;
- $\delta'(\tau_t, a) = \tau_{ta}$, the transition-function; and
- $F' = \{\tau_t | \exists q \in F \colon q \in \tau_t(\bullet)\}$ the set of finite states.

Before we can actually see that these two automata accept the same language, we have to introduce a few help functions.

Definition 6.10. Considering letters as words of length 1, we can view $\hat{\delta}'$ as an extension of δ' for words of arbitrary length. We will define $\hat{\delta}'(\tau_x, y)$ inductively with $y \in \Sigma^*$ as follows:
$\hat{\delta}'(\tau_x, \lambda) = \tau_x$
$\hat{\delta}'(\tau_x, ya) = \delta'(\hat{\delta}'(\tau_x, y), a)$

Lemma 6.11. *Let $x, y \in \Sigma^*$: $\hat{\delta}'(\tau_x, y) = \tau_{xy}$.*

Proof. We will prove this by induction over y.

- Induction base:
  $y = \lambda$. $\hat{\delta}'(\tau_x, \lambda) \stackrel{def}{=} \tau_x = \tau_{x\lambda}$.

- Induction hypothesis:
  $\hat{\delta}'(\tau_x, y) = \tau_{xy}$.

- Inductive step:
  $y' = ya$, which will be the concatenation of a word y and a letter $a \in \Sigma$.
  $\hat{\delta}'(\tau_x, ya) \stackrel{def}{=} \delta'(\hat{\delta}'(\tau_x, y), a) \stackrel{IH}{=} \delta'(\tau_{xy}, a) = \tau_{xya}$.

Since both the base and the inductive step have been performed and are correct, by mathematical induction, $\hat{\delta}'(T_x, y) = T_{xy}$ holds for all $x \in \Sigma^*$. □

Theorem 6.12. $L(M) = L(M')$

Proof.

$$\begin{aligned} t \in L(M') &\Leftrightarrow \hat{\delta}'(s', t) \cap F' \neq \emptyset \\ &\Leftrightarrow \hat{\delta}'(\tau_\lambda, t) \cap F' \neq \emptyset \\ &\Leftrightarrow \tau_t \cap F' \neq \emptyset \\ &\Leftrightarrow \exists q \in F \colon q \in \tau_t(\bullet) \\ &\Leftrightarrow t \in L(M) \end{aligned}$$

And now we see that $L(M) = L(M')$. □

6.3 Example

The example will make use of the following class of languages:
$\mathcal{L}_n = (a+b)^* a (a+b)^{n-1} a (a+b)^*$

Let us take a look at the following automaton representing the language $\mathcal{L}_2 = (a+b)^* a (a+b) a (a+b)^*$:

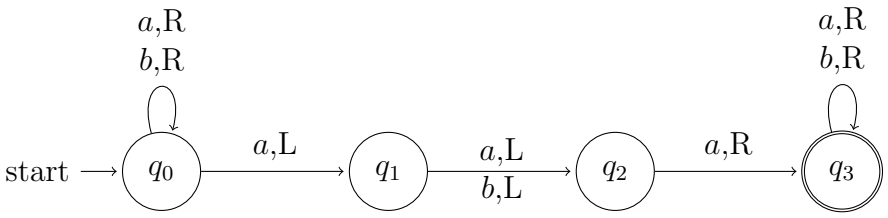


Figure 6.1: A Shepherdson-2NFA accepting $\mathcal{L}_2$

τ_λ	
$\bullet$	q_0
q_0	$\perp$
q_1	$\perp$
q_2	$\perp$
u_0	

τ_a	
$\bullet$	$q_0, \perp$
q_0	$q_0, \perp$
q_1	$\perp$
q_2	q_3
u_1	

τ_b	
$\bullet$	q_0
q_0	q_0
q_1	$\perp$
q_2	$\perp$
u_2	

τ_{aa}	
$\bullet$	$q_0, \perp$
q_0	$q_0, \perp$
q_1	q_3
q_2	q_3
u_3	

τ_{aaa}	
$\bullet$	$q_0, q_3, \perp$
q_0	q_0, q_3
q_1	q_3
q_2	q_3
u_4	

τ_{ab}	
$\bullet$	$q_0, \perp$
q_0	q_0
q_1	q_3
q_2	$\perp$
u_5	

τ_{aaab}	
$\bullet$	$q_0, q_3, \perp$
q_0	q_0
q_1	q_3
q_2	$\perp$
u_6	

τ_{aba}	
$\bullet$	$q_0, q_3, \perp$
q_0	q_0, q_3
q_1	$\perp$
q_2	q_3
u_7	

τ_{abb}	
$\bullet$	$q_0, \perp$
q_0	q_0
q_1	$\perp$
q_2	$\perp$
u_8	

τ_{aaabb}	
$\bullet$	$q_0, q_3, \perp$
q_0	q_0
q_1	$\perp$
q_2	$\perp$
u_9	

τ_{abaa}	
$\bullet$	$q_0, q_3, \perp$
q_0	$q_0, \perp$
q_1	q_3
q_2	q_3
u_{10}	

τ_{abba}	
$\bullet$	$q_0, \perp$
q_0	$q_0, \perp$
q_1	$\perp$
q_2	q_3
u_{11}	

τ_{aaabba}	
$\bullet$	$q_0, q_3, \perp$
q_0	q_0
q_1	$\perp$
q_2	$\perp$
u_{12}	

The set of states is all different τ_t, as stated above. The start-state is $s' = \tau_\lambda = u_0$. The transition function δ' is given below:

	a	b
u_0	u_1	u_2
u_1	u_3	u_5
u_2	u_1	u_2
u_3	u_4	u_5
u_4	u_4	u_6
u_5	u_7	u_8
u_6	u_7	u_9
u_7	u_{10}	u_6
u_8	u_{11}	u_8
u_9	u_{12}	u_9
u_{10}	u_4	u_6
u_{11}	u_3	u_5
u_{12}	u_{10}	u_6

Figure 6.2: The transition function δ'

The set of final states is $F = \{\tau_t | \tau_t(\bullet) \in F\} = \{u_4, u_6, u_7, u_9, u_10, u_12\}$. This gives us the following 1DFA. As you might notice, the states u_4, u_6, u_7, u_9, u_10 and u_12 could be merged into one state. As the word enters through u_4 or u_7 it will never leave that area. Note that the algorithm given is not optimal!

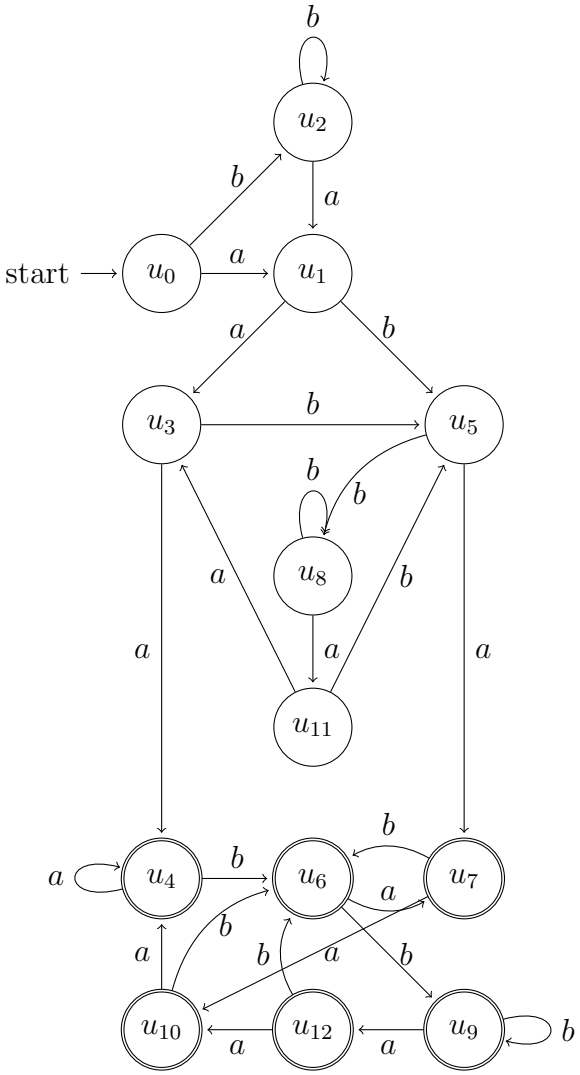


Figure 6.3: A 1DFA accepting $\mathcal{L}_2$

6.4 Complexity

Because this algorithm is an adaptation of the algorithm Dexter Kozen uses, the lower bound remains the same.

The upper bound changes to $\mathcal{O}(2^{(n+1)^{(n+1)}})$, because τ_t gives a maximum of $2^{(n+1)^{(n+1)}}$ tables where n is the number of states of the 2NFA that is being converted.

Chapter 7

Unary Languages and Automata

We now introduce unary finite automata, both deterministic and non-deterministic one-way, and both one-way and two-way. Since we will only be talking about words over a one-letter alphabet in this chapter, input words can be identified as positive integers, as well as 0. Therefore, we will write x instead of a^x.

All of the given transformations in this chapter are from "Finite Automata and Unary Languages"[4] written by M. Chrobak in 1986. As is written there, we will talk about the transformation of a 1NFA to a 1DFA, a 2DFA to a 1DFA and 1NFA to 2DFA. All of these automata will be unary.

Definition 7.1. A unary one-way deterministic finite automaton (U1DFA) $A = (Q, q_0, E, F)$ is an automaton with the following properties.

- Q: the finite set of states;
- $q_0 \in Q$: the begin-state;
- $E \subseteq Q \times Q$: the set of directed edges between states, such that for every $q_i \in Q$ there exists a unique $q_j \in Q$ with $(q_i, q_j) \in E$;
- F: the set of final states, with $F \subseteq Q$.

Note that when we are talking about a deterministic unary finite automaton, the number of edges is equal to the number of states, $|E| = |Q|$.

Definition 7.2. The acceptance of a word x for a unary deterministic finite automaton is the same as the acceptance of a word w for a deterministic finite automaton. The only difference is that when we are talking about a unary 1DFA, we don't have a transition function, but purely the set of edges. When we have an edge (q_i, q_j), we consider this the same as $\delta(q_i, a) = q_j$.

Definition 7.3. A unary one-way non-deterministic finite automaton (U1NFA) $A = (Q, q_0, E, F)$ is an automaton with the following properties.

- Q: the finite set of states;
- $q_0 \in Q$: the begin-state;
- $E \subseteq Q \times Q$: the set of directed edges between states, such that for every $q_i \in Q$ there exists $q_j \in Q$ with $(q_i, q_j) \in E$;

- F: the set of final states.

Note that the number of edges of a non-deterministic unary finite automaton is at least as large as the number of states, $|E| \geq |Q|$.

When we are talking about the acceptance of a unary non-deterministic finite automaton, this is nearly the same as Definition 7.2. The only difference is of course, that we are talking about unary non-deterministic finite automata instead of unary deterministic finite automata.

Definition 7.4. An unary two-way deterministic finite automaton (U2DFA) $B = (Q, q_0, \delta, F)$ is an automaton with the following properties.

- Q: the finite set of states;
- $q_0 \in Q$: the begin-state;
- $\delta\colon Q \times \{a, \vdash, \dashv\} \to Q \times \{L, R\}$: the transition-function with a being the only letter that is being read. L and R represent left and right: the direction that the we read the input word in. $\dashv$ and $\vdash$ represent the endmakers of the word. A word x will be read af $\vdash x \dashv$.
- F: the set of final states.

Definition 7.5. A word $\vdash x \dashv$:

- is accepted by B if the run of x on B ends in a state $q \in F$.
- is not accepted by B otherwise.

We could also define a two-way non-deterministic automaton, but since we will not be using these automata the definition will be omitted.

Definition 7.6. A sweeping automaton is an automaton that only turns the direction of which it is reading the word at the end (and beginning) of the word. This is the same for "normal" as well as unary automata.

7.1 Normal form 1NFA

Definition 7.7. Let A be a unary one-way non-deterministic finite automaton (U1NFA) with the following properties.

- r of its vertices are in cycles
- s of them are not, so that $r + s$ are equal to $|Q|$

Then we define $S(A) = (r, s)$.

Definition 7.8. A U1NFA $A = (Q, q_0, E, F)$ is in *normal form* if it meets the following requirements.

- The states consist of the states that are the prelude to the cycles and the states that are in the the cycles themselves.
  $Q = \{q_0, q_1, \ldots, q_n\} \cup C_1 \cup C_2 \cup \ldots C_k$, where $C_i = \{p_{i,0}, p_{i,1}, \ldots, p_{i,y_i-1}\}$ for $i \in \{1, \ldots, k\}$, with C_i being the i'th cycle.

- The vertices consist of the path from q_0 to q_n, the paths in the cycles $C_1, \ldots, C_k$ and the connections between the cycles and q_n.

  $E = \{(q_i, q_{i+1}) | i \in \{0, \ldots, n-1\}\} \cup$
  $\{(p_{i,j}, p_{i,j+1}) | i \in \{0, \ldots, y_j - 1\}\} \cup$
  $\{(q_n, p_{i,0}) | i \in \{1, \ldots, k\}\}$

  The addition of $j + 1$ in the second component is mod y_i.

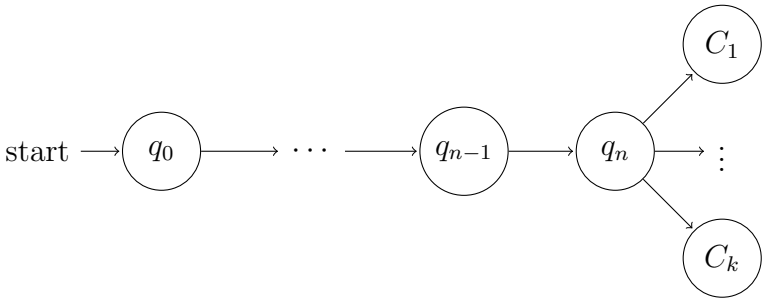


Figure 7.1: The general shape of an normal form U1NFA

We state the following lemma without proof. The proof can be found in the paper "Finite Automata and unary languages"[4], Lemma 4.3.

Lemma 7.9. *For every U1NFA A with n states, there exists a normal form U1NFA A' that accepts the same language such that $S(A') \leq (n, \mathcal{O}(n^2))$.*

7.2 1NFA to 1DFA

Say we want to transform a n-state U1NFA A into a U1DFA B. First we want to transform A into a normal form U1DFA A'. For A' we define $y_1, \ldots, y_k$ and y as follows.

- $y_1, \ldots, y_k$ are the lengths of the cycles $C_1, \ldots, C_k$
- $y = \text{lcm}(y_1, \ldots, y_k)$

The U1DFA $B = (Q, q_0, E, F)$ is then constructed as follows.

- $Q = \{q_0, \ldots, q_{s-1}, q_s, \ldots, q_{y+s-1}\}$
- q_0 is the q_0 of A'.
- $E = \{q_i, q_{i+1}) | i \in \{0, 1, \ldots, y + s - 2\} \cup \{q_{y+s-1}, q_s)\}$
- F is constructed as follows:
  - If q_i with $0 \leq i < s$ is an accepting state in A', then $q_i \in F$.
  - If $p_{i,j}$ is an accepting state in A', then $q_{s+t} \in F$ for each t such that $t - j = cy_i$ for some $c \in \mathbb{N}$.

For a better understanding of this construction actually works, here is a step-by-step breakdown. We will start with the normal form U1NFA, for there is were the relevant steps happen.
The automaton A' below accepts the language $\mathcal{L} = aaaa((aa)^* + (aaa)^*)$.

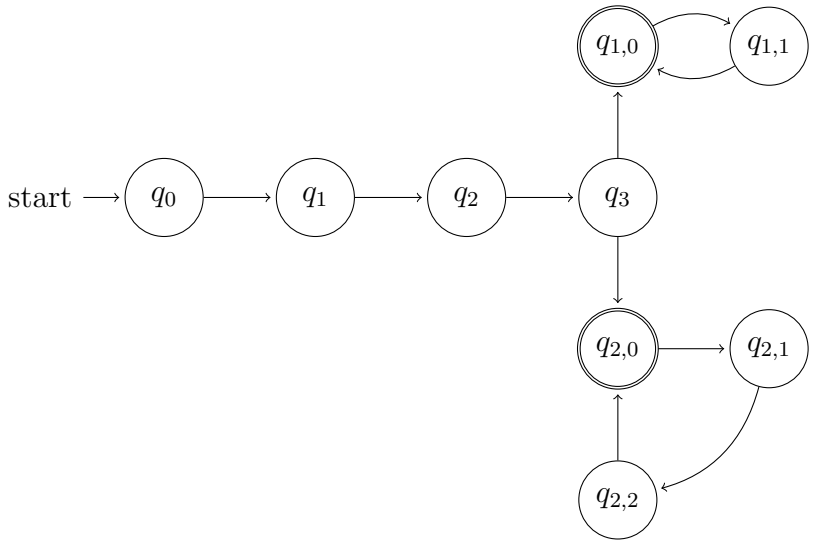


Figure 7.2: A normal form U1NFA

We see that this automaton is in normal form (Definition 7.8) with the following constants.

$$\begin{array}{lll} n = 9 & s = 4 & y_1 = 2 \\ r = 5 & y = 6 & y_2 = 3 \end{array}$$

Then we can start constructing the U1DFA $B = (Q, q_0, E, F)$:

- $Q = \{q_0, \ldots, q_3, q_4, \ldots, q_9\}$
- $E = \{q_i, q_{i+1})|i \in \{0, 1, \ldots, 8\}\} \cup \{q_9, q_4)\}$
- q_0 remains the same.
- For F we have to look at the 2 accepting states of A':
  - $p_{1,0}$ is an accepting state in A', so $q_{s+t} \in F$ for each t such that $t = 2c$ for some $c \in \mathbb{N}$. This is true for $t = 0, 2, 4$. So $q_{4+0}, q_{4+2}, q_{4+4} = q_4, q_6, q_8 \in F$.
  - $p_{2,0}$ is an accepting state in A', so $q_{s+t} \in F$ for each t such that $t = 3c$ for some $c \in \mathbb{N}$. This is true for $t = 0, 3$, so $q_{4+0}, q_{4+3} = q_4, q_7 \in F$.

  This gives $F = \{q_4, q_6, q_7, q_8\}$

This gives us the following U1DFA:

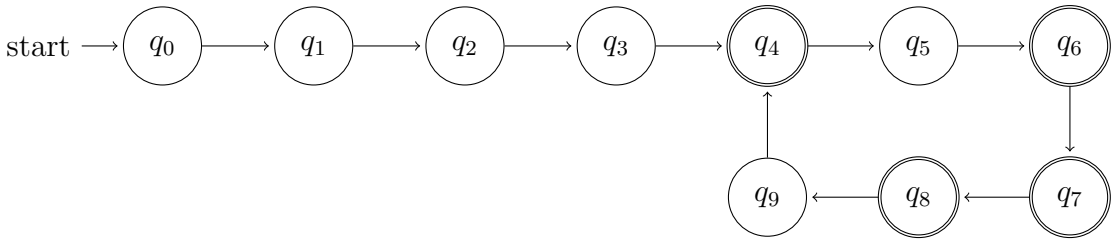


Figure 7.3: U1DFA B that accepts $\mathcal{L} = aaaa((aa)^* + (aaa)^*)$.

7.3 2DFA to 1DFA

We have already seen how to transform a U2DFA into a U1DFA with the use of Kozen's method. There is also a method of transforming a U2DFA into a U1DFA with the use of sweeping automata, this method is purely used for unary automata. We will discuss both these methods and show the reader how the method works with an example of a U2DFA that accepts $\mathcal{L} = \{n | n \equiv 0 \bmod 6\}$.

7.3.1 The Kozen transformation

Please note that an R at an edge denote a,R and L denote a,L.

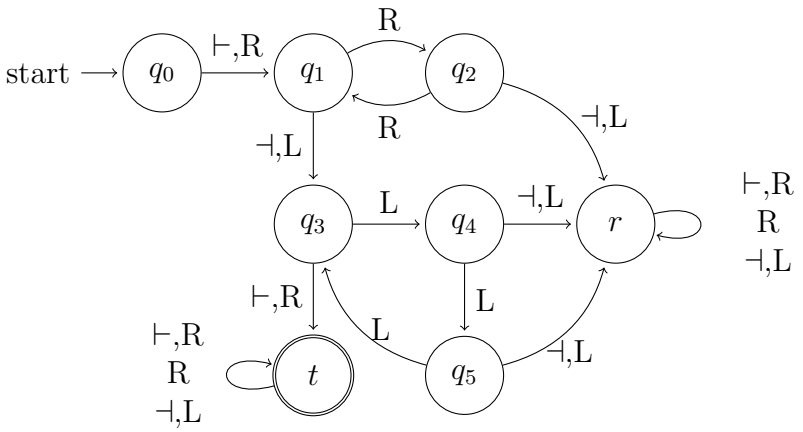


Figure 7.4: U2DFA that accepts $\mathcal{L} = \{a^n | n \equiv 0 \bmod 6\}$

To define the U1DFA corresponding to the U2DFA we need to take several steps. Here we will do the same procedure as in Section 5.1. We have to decide the set of states Q', the starting state s', the transitioning function δ' and the set of states that are accepting F'.

For all the states in Q, see the tables below.

T_λ	
•	q_1
q_3	t
q_4	r
q_5	r
u_0	

T_a	
•	q_2
q_3	r
q_4	r
q_5	t
u_1	

T_{aa}	
•	q_1
q_3	r
q_4	t
q_5	r
u_2	

T_{aaa}	
•	q_2
q_3	t
q_4	r
q_5	r
u_3	

T_{aaaa}	
•	q_1
q_3	r
q_4	r
q_5	t
u_4	

T_{aaaaa}	
•	q_1
q_3	r
q_4	t
q_5	r
u_5	

We have $s' = T_\lambda = u_0$ and $\delta'(T_x, a) = T_{xa}$. This provides us with the following transition function.

	a
u_0	u_1
u_1	u_2
u_2	u_3
u_3	u_4
u_4	u_5
u_5	u_0

Table 7.1: The transition function δ'

Finally we have $F' = \{T_x | x \in L(M)\} = \{T_\lambda\} = \{u_0\}$, because of all the options as stated above, only $\lambda \in L(M)$.

This provides us with the following U1DFA:

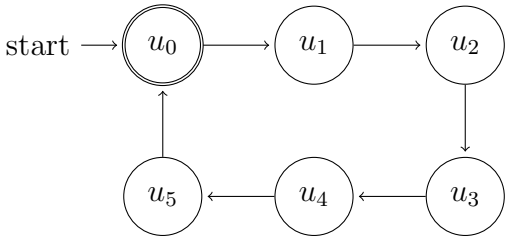


Figure 7.5: U1DFA that accepts $\mathcal{L} = \{a^n | n \equiv 0 \bmod 6\}$

Remark 7.10. When we're talking about a 1DFA that accepts a language of the form of $\{x | x \bmod n \equiv 0\}$, we can note that there is an automaton that has n states. Indeed, the automaton needs to count n occurrences of the letter. Fig 7.3.1 shows that a unary 2DFA for $\{x | x \bmod n \equiv 0\}$ can be made with

$$|Q| = 3 + \sum_{p \text{ prime}} \begin{cases} 0 & v_p(n) = 0 \\ p^{v_p(n)} & \text{otherwise} \end{cases}$$

where $v_p(n)$ is the multiplicity of prime-number p in the number n.

7.3.2 Using sweeping automata

We now give a general method for transforming a U2DFA into a U1DFA, now using sweeping automata. This method only applies to unary languages, and is due to [4]. The following is a high-level description of the idea. We first need to turn the U2DFA into a sweeping U2DFA. This can be done easily, according to Chrobak[4]. Let the sweeping automaton A have n states:

The U1DFA B will simulate the sweeping U2DFA $A = (Q, q_0, \delta, F)$ on the input x in the following way.

1. If $x \leq n$ and x has an accepting run on A, then x is accepted by B.

2. If $x > n$, then A will make a cycle when reading x. Let $y_1, \ldots, y_k$ be the lengths of the cycles $C_1 \ldots, C_k$ and $y = lcm(y_1, \ldots, y_k)$. We note that $y_1 + \cdots + y_k \leq n$, because no state can be in more than one cycle.
   For every two words u and v with $v > u > n$ and $v - u = y$:
   $u \in L(A) \leftrightarrow v \in L(A)$.

   In other words, knowing the lowest common multiple y suffices for words that are longer than n. If a word is longer than n, one must only keep subtracting y from x, until the length of x is less than n, so it will fall in category 1.

Here are the details of the description by example using the following sweeping U2DFA:

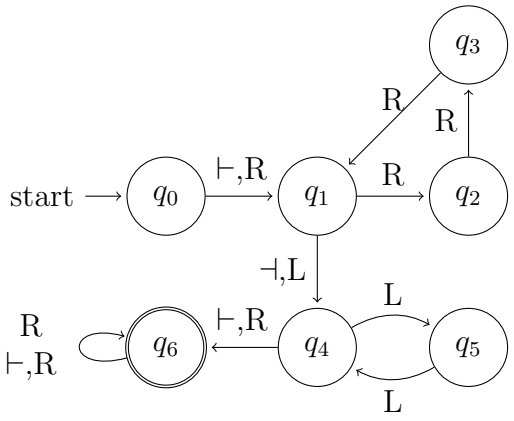


Figure 7.6: U2DFA that accepts $\mathcal{L} = \{a^n | n \equiv 0 \bmod 6\}$

Let us just look at the properties we can decude from this automaton:

1. The length of cycle one is 3: $y_1 = 3$.

2. The length of cycle two is 2: $y_2 = 2$.

3. $y = lcm(2, 3) = 6$

For every x for which $x \leq 6$, only 0 and 6 are accepted by A. Therefore, only $0, 6 \in L(B)$. From that information we see that $0+ky$ and $6+ky$ with $k \in \mathbb{N}$ should be accepted by A. However, we see that those two are equivalent, because they only differ by 6 which is the lowest common multiple.

We now only need to construct a U1DFA that accepts every x for which $x = ky$ with $k \in \mathbb{N}$ holds. This provides us with the following U1DFA:

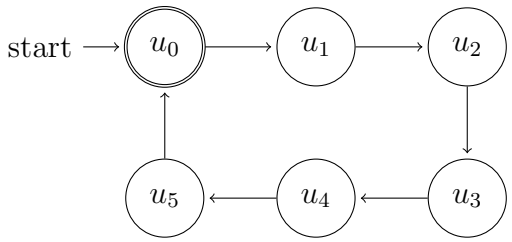


Figure 7.7: U1DFA that accepts $\mathcal{L} = \{a^n | n \equiv 0 \bmod 6\}$

7.4 1NFA to 2DFA

We now give a general method for transforming a U1NFA into a U2DFA using normal form U1NFA. This method only applies to unary languages, and is due to [4]. The following is a high-level description of the idea. We need to make a normal form U1NFA A' from a U1NFA A. We will not define the U2DFA B explicitly, but will show how B simulates the normal form 1NFA A'. It will do so in the following way:

- If $x < s$: $x \in L(B)$ iff q_x is an accepting state of A'

- Otherwise: B must make some passes over the input x, let's say j. The length of the cycle in that pass is y_j. There B will compute if $t = (x - s) \bmod y_j$. B will accept x if and only if $p_{j,t}$ is an accepting state of A'.

Here are the details of the description by example using the following normal form U1NFA:

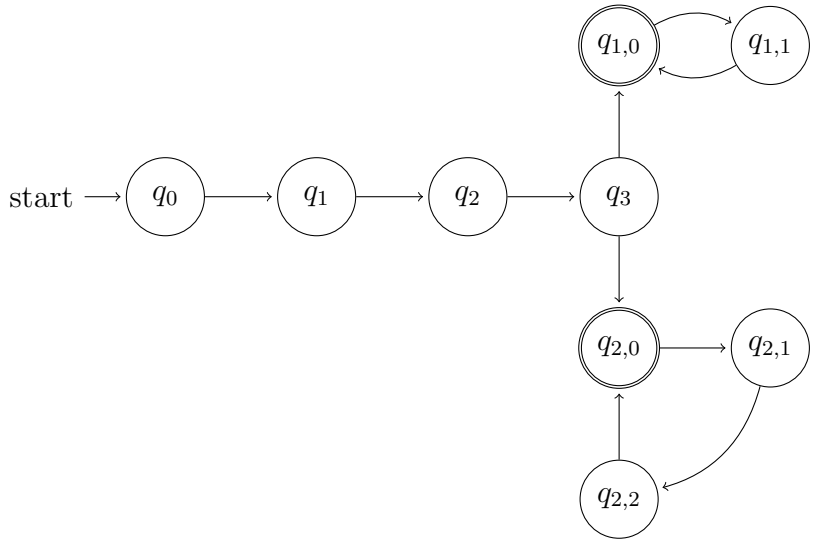


Figure 7.8: A normal form U1NFA

This normal form U1NFA has the following properties:

$$\begin{array}{lll} n = 9 & s = 4 & y_1 = 2 \\ r = 5 & y = 6 & y_2 = 3 \end{array}$$

So we calculate t for C_1 and C_2:

- C_1: $t \equiv (x-4) \bmod 2$.
  t needs to be 0, because $p_{1,0}$ is an accepting state in A'. Then we see that $(x-4) \bmod 2 \equiv 0$
- C_2: $t \equiv (x-4) \bmod 3$.
  t needs to be 0, because $p_{2,0}$ is an accepting state in A'. Then we see that $(x-4) \bmod 3 \equiv 0$

So for every $x > 4$, x will be accepted if

- $(x-4) \bmod 2 \equiv 0$, or
- $(x-4) \bmod 3 \equiv 0$

Remark 7.11. The best way to simulate this into a U2DFA is as follows. You first perform the first r steps and perform the first cycle by going to the right. After that you perform your "first" r steps again and perform cycle C_2, both by going left. This can be done until you are at the final cycle, alternating the direction for every cycle.

One obtains the following U2DFA

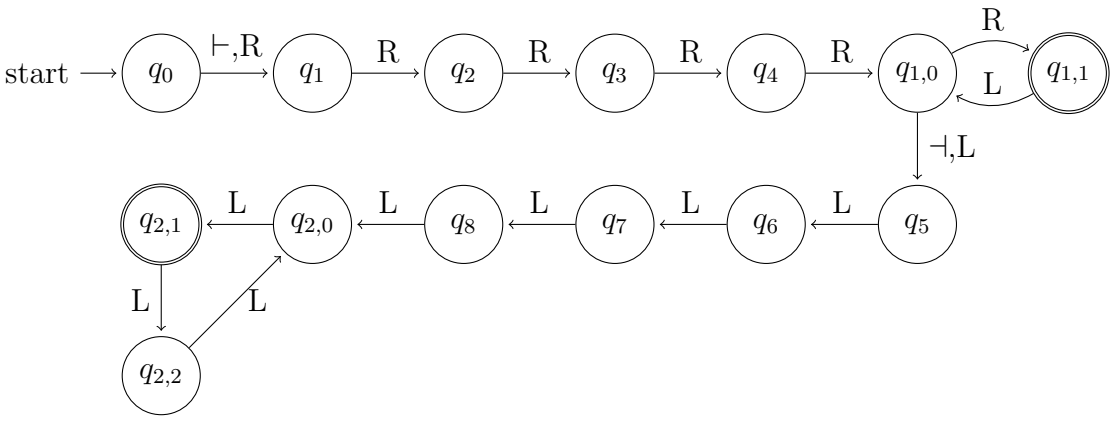


Figure 7.9: U2DFA

Chapter 8

Conclusion

We have seen that two-way deterministic automata accept the same languages as one-way deterministic finite automata. One can transform a 2DFA into a 1DFA using the method Kozen provides in his lecture notes. This algorithm has an upper bound of $\mathcal{O}((n+1)^{(n+1)})$ with n being the number of states the 2DFA has.

Furthermore we have seen that two-way non-deterministic finite automata also accept the same languages as one-way deterministic finite automata. One can transform a 2NFA into a 1DFA using a variation on the method Shepherdson provides. This algorithm has an upper bound of $\mathcal{O}(2^{(n+1)^{(n+1)}})$ with n being the number of states of the provided 2NFA.

As a final point we viewed both unary one-way (non-deterministic) finite automata and unary two-way deterministic finite automata. These kind of automata have unique methods for transforming into one another, because of the fact that they only accept unary languages. A common theme when transforming an unary one-way deterministic automaton into another automaton is the usage of the normal form one-way non-deterministic automaton.

8.1 A note on endmarkers

We notice that Kozen prefers the use of endmarkers, while others might not. Endmarkers are used for a better understanding and construction of certain Two-Way Automata. When one works with 2FA's, one will quickly notice that a lot of the 2FAs are *sweeping* automata. When using an endmarker, the automaton quickly recognizes it's at the end of the word or at the beginning of a word. A plain 2FAs doesn't have a way of recognising it's at the end (or at the beginning) of a word.

When adding something to an automaton, one creates the image that there are more classes of languages that will be accepted than before. This is not true in this case. According to Shepherdson one can transform a language that is based on endmakers to a language that isn't based on endmarkers. Addition of these endmarkers does not create more classes of languages.

Bibliography

[1] M. O. Rabin and D. S. Scott, "Finite automata and their decision problems," *IBM Journal of Research and Development*, vol. 3, no. 2, pp. 114–125, 1959.

[2] J. C. Shepherdson, "The reduction of two-way automata to one-way automata," *IBM Journal of Research and Development*, vol. 3, no. 2, pp. 198–200, 1959.

[3] D. Kozen, *Automata and computability*. Undergraduate texts in computer science, Springer, 1997.

[4] M. Chrobak, "Finite automata and unary languages," *Theor. Comput. Sci.*, vol. 47, no. 3, pp. 149–158, 1986.