CSC444

October 26, 2014

Contents

1 Language acceptors vs. language generators

The model that we will see next has a different flavor than finite automata, and talking about how it differs is instructive.

There are two distinct ways to approach the representation of languages.

The first way is to give an algorithm for recognizing the language.

Example 1.1. To discover if a string is a palindrome, reverse the string and save it in another variable. Compare characters from the front of the original string and the back of the reversed string. If any character does not match, then the string is not a palindrome. Otherwise, the string is a palindrome.

A representation that does this is called a *language recognizer.*

The second way is to give a description of how to generate strings in the language.

Example 1.2. You can generate a bipartite graph by taking n nodes, splitting them into two groups and then adding edges only between nodes in different groups.

A representation of this form is called a *language generator.*

Question 1.3.
What sort of representation is a finite automaton?

Next, we will see a representation that is a language generator.

2 Regular expressions

A regular expression describes a language using only singletons (that is, symbols in the underlying alphabet), $\emptyset$, $\cup$, parentheses and the Kleene star.

More formally, the *regular expressions* over an alphabet Σ are all strings over the alphabet $\Sigma \cup \{(,), \emptyset, \cup, \}$ that can be obtained as follows:

- $\emptyset$ and each member of Σ is a regular expression
- If α and β are regular expressions, then so is $(\alpha\beta)$
- If α and β are regular expressions, then so is $(\alpha \cup \beta)$
- If α is a regular expression, then so is α^*
- Nothing is a regular expression unless it follows from (1) through (4)

Note. Different textbooks use $+$ instead of $\cup$, and ε instead of $\emptyset$. They may also assume that $*$ has higher precedence than concatenation and that concatenation has higher precedence than $\cup$. This eliminates the need for the parentheses.

Example 2.1. Let $\Sigma = \{0, 1\}$

- $((0 \cup 1)^* 11 (0 \cup 1)^*)$
- $(0(0 \cup 1)^*)$
- $((0 \cup 1)^* 011)$
- $((1 \cup 10)^*)$ — the set of all strings beginning with 1 and having no consecutive 0's

2.1 Syntax vs. semantics

When we write a program in C++ or some other high-level language, we use a set of keywords and variables to construct a sequence of characters.

That program only has symbolic meaning with respect to the actual computer. Until we run it through an interpreter or compiler that translates it into something that is meaningful for the computer, it remains merely a sequence of characters.

In this way, we make a distinction between the syntax of a program (the actual characters used to construct it) and the semantics of the program (the machine instructions that it is intended to produce).

There is a similar distinction with respect to regular expressions.

Each regular expression is simply a sequence of characters, some from the underlying alphabet and others representing operations on sets.

In order to understand the expression, we have to come up with a way to assign those characters meaning.

We do that by formally associating each regular expression with a language over the specified alphabet.

The *meaning of a regular expression* is established by defining a function L such that if α is any regular expression, then $\mathcal{L}(\alpha)$ is the language represented by α.

The function L is defined as follows:

- $\mathcal{L}(\emptyset) = \emptyset$ and $\mathcal{L}(a) = \{a\}$ for each $a \in \Sigma$
- If α and β are regular expressions, then $\mathcal{L}((\alpha\beta)) = \mathcal{L}(\alpha)\mathcal{L}(\beta)$
- If α and β are regular expressions, then $\mathcal{L}((\alpha \cup \beta) = \mathcal{L}(\alpha) \cup \mathcal{L}(\beta)$
- If α is a regular expression, then $\mathcal{L}(\alpha^*) = (\alpha)^*$

This definition is not mysterious, since it simply says that we should interpret each regular expression in the way that we expect.

It is, however, important to understand that a regular expression is simply a string, devoid of meaning, until we apply the function L to it to find the associated language.

Let's go through the formal evaluation procedure for one example so that we can see how it works.

$$\begin{aligned}\mathcal{L}(((0\cup 1)^*11(0\cup 1)^*)) &= \mathcal{L}((0\cup 1)^*)\mathcal{L}((11))\mathcal{L}((0\cup 1)^*)\\ &= \mathcal{L}((0\cup 1))^*\mathcal{L}(1)\mathcal{L}(1)\mathcal{L}((0\cup 1))^*\\ &= (\mathcal{L}(0)\cup\mathcal{L}(1))^*\{1\}\{1\}(\mathcal{L}(0)\cup\mathcal{L}(1))^*\\ &= (\{0\}\cup\{1\})^*\{1\}\{1\}(\{0\}\cup\{1\})^*\\ &= \{0,1\}^*\{1\}\{1\}\{0,1\}^*\\ &= \{\, w\in\{0,1\}^* \mid w \text{ has 11 as a substring}\,\}\end{aligned}$$

It is important to know how to go through these formal steps, although we will not do it that often in this course.

Once you've had some practice, it becomes easy to determine the language that a given regular expression represents.

2.2 Translating languages into regular expressions

Let's now look at how to take a language and come up with a regular expression representing that language.

Let $\Sigma = \{0,1\}$. We will assume that $w \in \Sigma^*$ in the following examples.

Example 2.2. $L_1 = \{\, w \mid w \text{ has at least one 1 followed by at least one 0}\,\}$

$L_1 = (0\cup 1)^*10(0\cup 1)^*$

Example 2.3. $L_2 = \{\, w \mid w \text{ does not contain the substring } 11\,\}$

Break down the string into parts. It can either contain all 0's or have some 1's.

If it has some 1's, then we can break it into sections in the following way:

<optional zeros> 10 <optional zeros> 10 <optional zeros> 10 <optional zeros> ... <zero or one>

This looks like:

<optional zeros> (10 <optional zeros>)* <zero or one>

So the whole thing looks like:

$$L_2 = (0^* \cup (0^*(10^+)^*(0\cup 1)))$$

2.2.1 Sloppy notation

Throughout the rest of the course we may choose to be sloppy about our notation when referring to languages or regular expressions, provided that what we are writing is correct.

For example, we may leave off parentheses, brackets or the concatenation operator in various expressions.

If you have questions about what is meant by a particular expression, ask!

2.2.2 Multiple representations

Every language that can be represented by a regular expression, can be represented by infinitely many regular expressions.

We simply add symbols that don't change the underlying language.

Example 2.4. $(01)^*, ((01)^* \cup \emptyset), ((01)^* \cup (01)^*), ...$

2.3 The regular languages

Definition 2.5. A *regular language* over an alphabet Σ is a language L s.t. $L = \mathcal{L}(\alpha)$ for some regular expression α.

Have we captured something different from what we had before using finite automata?

Answer: No

What remains is to show that finite automata (either deterministic or non-deterministic) recognize precisely the regular languages.

2.4 Closure properties of finite automata

Recall that the class of regular languages was defined as the *closure* of certain finite languages under the operations of union, concatenation and Kleene star.

Before we can prove that finite automata and regular expressions have the same computing power, we need to show that finite automata are also closed under these operations (plus a few others).

Theorem 2.6. *The class of languages accepted by finite automata is closed under*

- *union*
- *concatenation*
- *Kleene star*
- *complementation*
- *intersection*

Proof. We will show in each case how to construct an automaton M that accepts the appropriate language, given two automata M_1 and M_2. (Note: We will only need M_1 for Kleene star and complementation). $\square$

Note. I will not give the formal definition or a proof for any of these cases. I assume that you could produce the 5-tuples if needed.

We can also assume without loss of generality that *the set of states of* M_1 *and* M_2 *are disjoint.*

- *Union*: Given M_1 and M_2 we want to construct a FA M s.t. $\mathcal{L}(M) = \mathcal{L}(M_1) \cup \mathcal{L}(M_2)$.

  The idea: Use nondeterminism to "guess" whether the string in question is in $\mathcal{L}(M_1)$ or $\mathcal{L}(M_2)$ and move the appropriate starting state.

  Let s_1 be the starting state of M_1 and s_2 be the starting state of M_2. Let s be a new state not found in either automaton. s will be the starting state of M and will have ε-transitions into both s_1 and s_2.

- *Concatenation*: Given M_1 and M_2 we want to construct M s.t. $\mathcal{L}(M) = \mathcal{L}(M_1) \cdot \mathcal{L}(M_2)$.

  The idea: Simulate M_1 for a while and then nondeterministically "jump" from a final state of M_1 to a starting state of M_2.

Let s_1 be the starting state for both M_1 and M. Connect each final state of M_1 to the starting state of M_2 using an ε-transition. Let the final states of M be the final states of M_2.

- *Kleene star*: Given M_1 we want to construct M s.t. $\mathcal{L}(M) = \mathcal{L}(M_1)^*$.

  M consists of all the states of M_1 and all the transitions of M_1. Any final state of M_1 is a final state of M.

  In addition, M has an extra state s not found in M_1. This state has the following properties:

  - s is the initial state for M
  - s is also final so that $\varepsilon \in \mathcal{L}(M)$
  - From s there is an ε-transition to the initial state s_1 of M_1

  Finally, ε-transitions are added from each final state of M_1 back to s_1. This is so that once a string in $\mathcal{L}(M_1)$ has been read, computation can resume from the initial state of M_1.

- *Complementation*: Let M be a DFA. (If we want to complement a NFA then we can first convert it into a DFA). Then $\bar{L} = \Sigma^* - \mathcal{L}(M)$ is accepted by the DFA $\bar{M}$ where $\bar{M}$ is M with all of its final and non-final states reversed. (For this to work, the initial DFA must be complete over Σ.)

- *Intersection*: Let $L_1 = \mathcal{L}(M_1)$ and $L_2 = \mathcal{L}(M_2)$.

  Note that:
  $$L_1 \cap L_2 = \Sigma^* - ((\Sigma^* - L_1) \cup (\Sigma^* - L_2))$$
  Since automata are closed under Kleene star, complementation and union, we know that automata are closed under intersection.

2.5 RLs and FAs

Theorem 2.7. *A language is regular if and only if it is accepted by a finite automaton.*

Proof:

$\Rightarrow$ Suppose that a language L is regular.

Recall. The class of regular languages is the smallest class of languages containing the empty set $\emptyset$ and the singletons a for all $a \in \Sigma$ that is closed under union, concatentation, and Kleene star.

- There is a finite automaton that accepts $\emptyset \to$ No final states.
- There is a finite automaton accepting a for each $a \in \Sigma \to$ Start state with a single transition to the final state labeled by a
- By the previous theorem the languages accepted by finite automata are closed under union, concatenation, and Kleene star.

So L is accepted by some finite automaton.

$\Leftarrow$ Let $M = (K, \Sigma, \Delta, s, F)$ be a finite automaton. We will construct a regular expression R s.t. $\mathcal{L}(R) = \mathcal{L}(M)$.

The idea: We will represent $\mathcal{L}(M)$ as the union of (finitely) many simple languages.

Let $K = \{q_1, q_2, ..., q_n\}$ and $s = q_1$.

Definition 2.8.

$$R(i,j,k) = \{\, w \in \Sigma^* \mid w \text{ causes } M \text{ to move from state } q_i \text{ to state } q_j \text{ without passing through any intermediate state numbered } k+1 \text{ or higher} \,\}$$

Note. i and j, that is, the values of the endpoints, are allowed to be larger than k.

Another way to say the above is that $R(i,j,k)$ is the set of strings spelled by all paths from q_i to q_j of rank k.

When $k = n$, it follows that $R(i,j,n) = \{\, w \in \Sigma^* \mid (q_i, w) \mapsto_M^* (q_j, \varepsilon) \,\}$.

Thus $\mathcal{L}(M) = \cup\{\, R(1,j,n) \mid q_j \in F \,\}$.

Since all of these sets are regular, and we are taking a finite union of them, the language $\mathcal{L}(M)$ is regular.

We must now show that each of the $R(i,j,k)$ is regular.

Claim 2.9. *$R(i,j,k)$ is regular for each i, j, k.*

Proof. By induction on k.

Base: $k = 0$

$R(i, j, 0)$ is either $\{ a \in \Sigma \cup \{\varepsilon\} \mid (q_i, a, q_j \in \Delta) \}$ if $i \neq j$ or it is $\{\varepsilon\} \cup \{ a \in \Sigma \mid (q_i, a, q_j \in \Delta) \quad \}$ if $i = j$.

Each of these is a finite set of alphabet symbols and is therefore regular.

Inductive step: Suppose that $R(i, j, k-1)$ for all i, j have been defined as regular languages.

The idea: To get from q_i to q_j without passing through a state numbered greater than k, M may either:

- go from q_i to q_j without passing through a state numbered greater than $k - 1$, or
- go (a) from q_i to q_k then (b) from q_k to q_k zero or more times then (c) from q_k to q_j; in each of these cases M may not pass through any intermediate states numbered greater than $k - 1$

Thus $R(i, j, k) = R(i, j, k-1) \cup R(i, k, k-1)R(k, k, k-1)^*R(k, j, k-1)$.

This is a regular expression since each of the $R(i, j, k)$'s is regular by the inductive hypothesis, and since regular expressions are closed under union, concatenation and Kleene star. ? $\square$

2.6 Construction of regular expressions

Now we want to see how we can move back and forth between regular expressions and finite automata using this notation and the construction from the proof.

2.6.1 Constructing a FA for a regular expression

Example 2.10. Let $R = (0 \cup 01)^*$. Let's construct a NFA to accept $\mathcal{L}(R)$ using the ideas from Theorem 2.4.2.

- Construct a FA accepting 0 and 1.

- Construct one accepting 01 by connecting them with an ε-transition and removing the accepting state at the end of the 0 machine.
- Construct the union of the two by defining a new starting state with ε-transitions to each of the other starting states.
- Produce the Kleene star by adding another starting state that is also an accepting state, linking it to the starting state of the union, and adding with ε-transitions from the accepting states to the initial state of the union.

2.6.2 Finding a regular expression for a FA

We can simplify the construction of the regular expression if we assume that the finite automaton has *two properties*:

- It has a single final state $F = \{f\}$.
- If $(q, u, p) \in \Delta$, then $q \neq f$ and $p \neq s$, that is, there are no transitions into the initial state nor out of the final state.

This special form is *not a loss of generality.* Why?

We can add to any automaton M a new initial state s and a new final state f, together with ε-transitions from s to the initial state of M and from all final states of M to f.

If we number the states of the automaton $q_1, q_2, \ldots, q_n$ so that $s = q_{n-1}$ and $f = q_n$, then the regular expression for M is simply $R(n-1, n, n)$.

Our computation will go as follows: $R(i, j, 0) \rightarrow R(i, j, 1) \rightarrow ...R(i, j, k)$.

At each stage we will depict each $R(i, j, k)$ *as a label on an arrow* going from state q_i to state q_j.

This will have the effect of transforming the finite automaton into an equivalent *generalized finite automaton*, with transitions that may be labeled not only by symbols in Σ or ε but by entire regular expressions.

Note that once we've finished with computing $R(i, j, k)$ we can *eliminate* state q_k. This is because each string that leads M to acceptance by passing through q_k has been taken into account in the $R(i, j, k)$'s.

How do we *eliminate a state* q in general?

For each pair of states $q_i \neq q$ and $q_j \neq q$, s.t. there is an arrow from q_i to q labeled α, an arrow labeled β from q to q_j, and an arrow labeled γ from q to itself, we add an arrow from q_i to q_j labeled $\alpha\gamma^*\beta$.

If there is no arrow from q to itself, we simply label the new arrow $\alpha\beta$.

If there was an arrow from q_i to q_j with label λ, then the new arrow's label becomes $\lambda \cup \alpha\gamma^*\beta$.

Once we have eliminated all states except the initial and final states, then we have a regular expression that represents the finite automaton.

Example 2.11. Let M be a DFA accepting $\{\, w \in \{a, b\}^* \mid w \text{ has } 3k + 1b\text{'s for some } k \in N \,\}$. In particular, M is given by $K = \{q_1, q_2, q_3\}$, $\Sigma = \{a, b\}$, $s = q_1$, $F = \{q_3\}$ with transition function δ given below:

q	σ	δ (q,σ)
q_1	a	q_1
q_1	b	q_3
q_2	a	q_2
q_2	b	q_1
q_3	a	q_3
q_3	b	q_2

Notation. We will omit arrows labeled by $\emptyset$ and self-loops labeled $\{\ \varepsilon\ \}$.

With this assumption, the initial automaton has the correct values of the $R(i, j, 0)$'s.

This is not true in general. In the automaton given there is at most one transition of the form (q_i, u, q_j) in Δ for each i and j. If there was more than one transition for a particular i and j, we would have to merge it into a single transition with the union of states on the arrow.

- Put the finite automaton in the correct form.
- Compute each $R(i, j, 1)$:
  - $q_4 \to q_3$ labeled by a^*b

  - $q_2 \to q_3$ labeled by ba^*b

Now we eliminate state q_1.

- Compute each $R(i, j, 2)$:
  - $q_3 \to q_3$ labeled by a $\cup ba^*ba^*b$

We then eliminate state q_2.

- Compute each $R(i, j, 3)$:
  - $q_4 \to q_5$ labeled by $a^*b(a \cup ba^*ba^*ba^*)^*$

We then eliminate state q_3.

- Read off the regular expression from the start state to the final state.

3 State minimization

As we can show using the pumping lemma, the computing power of finite automata is very restricted.

However, finite automata can be useful in many ways including the following:

- As a piece of an algorithm or system, for example, in a model checker
- As a component of a cellular automaton
- As the underlying mechanism for stronger models of computation, such as pushdown automata and Turing machines

Recall that we can construct more than one finite automaton to accept the same regular language.

Since they are primarily used as parts of a larger object, it is important to use the simplest automaton possible for a given language.

We usually measure the simplicity of a finite automaton by the number of states that it has.

We will introduce a result that shows us how to *minimize the number of states* of a given DFA.

Given a particular DFA, this algorithm will allow us to find an equivalent DFA that has a few states as possible.

3.1 Unreachable states

The first step toward decreasing the size of a DFA is to eliminate the unreachable states.

These states do not change the language accepted by the DFA, so eliminating them gives you an equivalent DFA.

Example 3.1. Let $M = (K, \{0, 1\}, \delta, s, F)$, where $K = \{q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8\}$, $s = q_1$, $F = \{q_1, q_3, q_7\}$ and the transition function δ is given by the table below:

q	σ	δ(q,σ)
q_1	0	q_2
q_1	1	q_4
q_2	0	q_5
q_2	1	q_3
q_3	0	q_2
q_3	1	q_6
q_4	0	q_1
q_4	1	q_5
q_5	0	q_5
q_5	1	q_5
q_6	0	q_3
q_6	1	q_5
q_7	0	q_6
q_7	1	q_8
q_8	0	q_7
q_8	1	q_3

Draw the *state diagram* found on page 93 of the Lewis and Papidimitriou textbook.

$\mathcal{L}(M) = (01 \cup 10)^*$

Both state q_7 and q_8 are unreachable since there is no directed path from q_1 to either state.

So we can eliminate both q_7 and q_8 without changing $\mathcal{L}(M)$.

Once we have removed both these states, there are still unnecessary states. Showing this requires a more subtle argument.

3.2 Minimization algorithm

The next two definitions require a quick review of relations.

3.2.1 Equivalence relations

Defns: A relation $R \subseteq A \times A$ is:

- *Reflexive* if $(a, a) \in R$ for each $a \in A$.
- *Symmetric* if $(b, a) \in R$ whenever $(a, b) \in R$.
- *Transitive* if whenever $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$.
- An *equivalence relation* if R is reflexive, symmetric and transitive.

Definition 3.2. A *partition* of a non-empty set A is a decomposition of A into subsets s.t. each subset is disjoint and the union of all the subsets is A.

Theorem 3.3. *Let R be an equivalence relation on a non-empty set A. Then the equivalence classes of R constitute a partition of A.*

Notation. The partition of A is called the *equivalence classes* of A. We write $[a]$ to denote the equivalence class containing the element a. More formally, $[a] = \{ b \mid (a, b) \in R \}$.

We now want to talk about an algorithm that given a particular DFA M will *construct the minimal finite automaton* associated with M.

3.2.2 Definitions

Definition 3.4. Let $M = (K, \Sigma, \delta, s, F)$ be a DFA. Let $A_M \subseteq K \times \Sigma^*$ be a relation defined as follows: $(q, w) \in A_M \Leftrightarrow (q, w) \mapsto_M^* (f, \varepsilon)$, for some $f \in F$.

Thus if $(q, w) \in A_M$ then w *drives* M *from* q *to an accepting state.*

Definition 3.5. Two states q and p are *equivalent*, denoted $q \equiv p$, if the following holds for all $z \in \Sigma^* : (q, z) \in A_M \Leftrightarrow (p, z) \in A_M$.

Note. $\equiv$ is an equivalence relation.

We must merge each of the equivalence classes under $\equiv$ into a single state in the minimal automaton for $\mathcal{L}(M)$.

How will we do this?

Definition 3.6. For two states q and p, $q \equiv_n p$ if the following is true: $(q, z) \in A_M \Leftrightarrow (p, z) \in A_M$ for all strings z such that $|z| \leq n$.

This is a coarser definition of equivalence that says two states are the same if they behave the same way with respect to acceptance when driven by strings of length no greater than n.

We will compute the equivalence classes of $\equiv$ as the limit of $\equiv_0$, $\equiv_1$, $\equiv_2$, etc.

Clearly, when we increase the value of n, we are refining $\equiv_n$. This is because a string of length n may be able to distinguish two states that are both in $\equiv_{n-1}$.

3.2.3 Algorithm for computing $\equiv_{n+1}$ from $\equiv_n$

It is easy to compute $\equiv_0$: $q \equiv_0 p$ iff q and p are either both accepting or both non-accepting states.

This means that there are precisely two equivalence classes of $\equiv_0$: F and $K - F$ (assuming that both are non-empty).

We now have to determine how to compute $\equiv_{n+1}$ given $\equiv_n$. The following lemma gives us the relationship between the two relations that we need.

Lemma 3.7. *For any two states $q, p \in K$ and any integer $n \geq 1$, $q \equiv_n p$ implies (a) $q \equiv_{n-1} p$, and (b) for all $a \in \Sigma$, $\delta(q, a) \equiv_{n-1} \delta(p, a)$.*

Proof. By definition. $\square$ *Algorithm* for computing $\equiv$ (and thus the standard automaton for a given language):

- Let F and $K - F$ be the classes in $\equiv_0$
- Repeat for $n = 1, 2, ...$

Compute the equivalence classes of $\equiv_n$ from those of $\equiv_{n-1}$

Until $\equiv_n$ is the same as $\equiv_{n-1}$

Certainly this is the correct thing to do given the lemma. But does this process actually *terminate*?

For each iteration at which the termination condition is not satisfied, $\equiv_n$ is a proper refinement of $\equiv_{n-1}$, so that $\equiv_n$ has at least one more equivalence class than $\equiv_{n-1}$.

But since the language under consideration cannot have more equivalence classes than it has states, this will terminate after no more than $|K| - 1$ iterations.

3.2.4 Example

Let's apply the algorithm to the DFA in Example 2, that is, $M = (K, \{0, 1\}, \delta, s, F)$, where $K = \{q_1, q_2, q_3, q_4, q_5, q_6\}$, $s = q_1$, $F = \{q_1, q_3\}$ and the transition function δ is given by the table below:

q	σ	δ(q,σ)
q_1	0	q_2
q_1	1	q_4
q_2	0	q_5
q_2	1	q_3
q_3	0	q_2
q_3	1	q_6
q_4	0	q_1
q_4	1	q_5
q_5	0	q_5
q_5	1	q_5
q_6	0	q_3
q_6	1	q_5

- The equivalence classes of $\equiv_0$ are $\{q_1, q_3\}$ and $\{q_2, q_4, q_5, q_6\}$.
- The equivalence classes of $\equiv_1$ are $\{q_1, q_3\}$, $\{q_2\}$, $\{q_4, q_6\}$, and $\{q_5\}$. The split happened because $\delta(q_2, b)$ is an accepting state but the states $\delta(q_4, b)$, $\delta(q_5, b)$ are not accepting. Also $\delta(q_4, a)$ is an accepting state, but $\delta(q_5, a)$ is not an accepting state.
- After the second step, there is no further splitting of the classes. We now have the 4-state automaton that we have seen before.

3.2.5 Running Time

What about the running time of this algorithm?

For each iteration at which the termination condition is not satisfied, $\equiv_n$ is a proper refinement of $\equiv_{n-1}$, so that $\equiv_n$ has at least one more equivalence class than $\equiv_{n-1}$.

Since the language under consideration cannot have more equivalence classes than it has states, this will terminate after no more than $|K| - 1$ iterations.

How much work is required at each iteration?

We must determine, for each pair of states, whether they are related by $\equiv_n$.

This takes $O(|\Sigma|)$ tests to determine if two states are related by a previously computed equivalence relation $\equiv_{n-1}$,

This requires $O(|\Sigma| \cdot |K|^3)$ time. This is polynomial in the size of the input, namely, the original automaton.