1 Decidable Problems on Finite Automata

Problems 4.15, 4.16 from the text.

Comments: In 4.15, you simply have to describe an algorithm for solving the problem. In 4.16, the author essentially tells you what the algorithm is, and you just have to figure out at what point you can stop and say that the two languages are equal. Unless I’m missing something, it is probably an error to try to work directly with the regular expression in 4.15—instead you should convert to a different representation.

1.1 Solutions.

4.15. We show decidability by describing an algorithm for determining whether a regular expression \( R \) has the desired property.

- Convert \( R \) into an NFA \( M \) such that \( L(M) \) is the language generated by \( R \). The algorithm for carrying this out is described in Lemma 1.55 of the text.

- For each pair of states \((q, q')\) of \( M \)
  - Determine if there is a path from the initial state to \( q \) and from \( q' \) to one of the accepting states. (This can be determined using a standard graph algorithm such as depth-first search).
  - If such paths exist, determine if there is a path labeled 111 from \( q \) to \( q' \). (Note that if such a path exists, then there is one whose length is no more than the number of states of \( M \), even if \( M \) has \( \epsilon \)-transitions, so we can indeed determine this.) If this path exists, then terminate and answer “yes”.

- If no pair of states gives the desired three paths, terminate and answer “no”.

4.16. Let \( M_1 \) and \( M_2 \) be DFA’s with \( n_1 \) and \( n_2 \) states, respectively, and where \( q_0^{(1)} \) and \( q_0^{(2)} \) are the respective initial states.
Suppose we read a string \( w = a_1 \cdots a_m \) with both automata, in effect reading it in the product automaton as described in Theorem 1.25. The successive letters of \( w \) will give rise to a sequence of pairs of states

\[
(q_0^{(1)},q_0^{(2)}), (q_1^{(1)},q_1^{(2)}), \ldots, (q_m^{(1)},q_m^{(2)})
\]

If \( m \geq n_1 n_2 \), then there will be a repeated pair of states—a loop in the product automaton. We can excise this loop and find a shorter string \( w' \) that leads to the same pair of states at the end.

Thus if there is a string \( w \) that is accepted by one the two DFAs and not by the other, then there is such a string with length less than \( n_1 n_2 \). This gives us the algorithm to check equivalence of the two automata: It suffices to test all the strings less than this length. If each of these strings is accepted by both automata or rejected by both automata, then \( L(M_1) = L(M_2) \). If the two automata disagree on one of these strings, then of course \( L(M_1) \neq L(M_2) \).

2 Problems About Turing Machines

1. Problem 5.9 from the text.
2. Consider the problem of determining, given a single-tape Turing machine \( M \) and a tape symbol \( a \), whether \( M \), when started on a blank tape, ever writes the symbol \( a \). Show that this problem is undecidable.

Comment. In both these problems, the solution is a reduction from \( A_{TM} \). In 5.9, it’s already in the book, if you know where to look!

3. Problem 5.13 from the text.

2.1 Solutions.

1. The problem asks us to show that the problem of determining whether the language recognized by a given Turing machine is closed under reversal is undecidable. We prove this by reducing \( A_{TM} \) to the problem. We use exactly the same reduction as in the proof of Theorem 5.3. That reduction begins with a TM \( M \) and a string \( w \), and constructs a TM \( M' \) such that

- If \( M \) accepts \( w \) then \( L(M') = \{0,1\}^* \).
- If \( M \) does not accept \( w \), then \( L(M') = \{0^n1^n : n \geq 0\} \).

The first of these two languages is closed under reversal, while the second is not. Thus if we had an algorithm to determine whether the language recognized by a TM is closed under reversal, we would have an algorithm for \( A_{TM} \).

3. Again, we describe a reduction from \( A_{TM} \).

2. Note that the word “every” that appears in this problem should have been “ever”!

We show this problem is undecidable by reducing \( A_{TM} \) to it: Let \( M \) be a Turing machine, and let \( w \) be a string. We define a new Turing machine \( M' \)
whose tape alphabet is the same as that of $M$ with one additional symbol $a$ adjoined. $M'$ works as follows: When started, it erases its input tape, writes the string $w$ and proceeds to simulate $M$. However the transitions are altered so that whenever $M$ enters an accepting state, $M'$ writes the symbol $a$.

As a result, $M$ accepts $w$ if and only $M'$ eventually writes the symbol $a$ when started on a blank tape. Thus if we had an algorithm for deciding the latter condition, we would have an algorithm for $A_{TM}$.

3. Again we show this problem is undecidable by reducing $A_{TM}$ to it. Given an Turing Machine $M$ and a string $w$, we construct a new Turing Machine $M'$ that works as follows: $M'$ begins by erasing its input tape and writing the string $w$, then simulating $M$ on $w$. If the simulation ever enters the accept state of $w$, $M'$ executes a sequence of transitions that visits every state of $M$, then halts. (Note that the states of $M'$ are the states of $M$ together with any additional states needed to carry out the initial phase of erasing the input and writing $w$.

Observe that, as is the case in my such reductions, $M'$ behaves the same on all inputs. If $M$ does not accept $w$, then the simulation never enters the accept state of $M$, and thus $M'$ has a useless state. Conversely, if $M$ does not accept $w$, then $M'$ visits every one of its states, and thus has no useless states. Thus if we had an algorithm for detecting the presence of useless states, we would have one for $A_{TM}$.

3 Problems About Time Complexity and NP-Completeness

The textbook shows that the problem $COMPOSITES$ is in $NP$, and in fact, it is now known to be in $P$. An outstanding open problem is whether it is possible to find the prime factors of an integer, given in binary, in polynomial time. It is believed that no polynomial time algorithm exists, and in fact, the security of many cryptographic systems in wide use depends on this belief. This problem ($FACTORIZATION$) does not have an obvious formulation as a language (this is what we will address below). It should be noted that the polynomial-time algorithm for $COMPOSITES$ gives no hint about the factors of the number.

Consider the language $L$ consisting of all pairs of strings $<x,y>$, where $x$ and $y$ are binary encodings of integers, such that $y$ has a factor $z$ with $1 < z < x$. For example $<7,25>$ (with the integers encoded in binary) belongs to $L$, because $5 < 7$ and 5 is a factor of 25.

(a) Show that $L$ is in $NP$.

(b) Show that if $P = NP$, then factoring can be done in polynomial time. (That is, if you are given an $n$-bit number, it can be factored in time polynomial in $n$.) HINT: If $P = NP$, then there is a polynomial-time algorithm for testing membership in $L$. Couple this with binary search to find a polynomial-time algorithm for factoring. Give an example to illustrate how your algorithm works.
3.1 Solutions.

(a) Consider the nondeterministic algorithm that given \( <x, y> \), guesses a factor \( z \). The standard long division algorithm for dividing \( z \) into \( y \) takes time \( O(n^2) \), where \( n \) is the number of bits of \( y \), so we can verify in polynomial time if the guess is indeed a factor. So \( L \in NP \).

(b) If \( P = NP \), then there is an algorithm for deciding membership in \( L \) whose running time is bounded above by \( n^r \), where \( n \) is the number of bits in the input and \( r \) is some constant. We can then use this algorithm as follows to determine a factor of a given number \( y \). We first set \( z = y \) and test if \( <y, y> \in L \). This will tell us whether \( y \) is composite or prime. If it is composite, we proceed to test membership in \( L \) repeatedly with \( z = y/2 \), and then using binary search to narrow down the location of the factor. The number of iterations in binary search is proportional to \( \log y \) which is a constant times \( n \), and each iteration, by our assumption, requires \( (2n)^r \) steps, at most. So we find a factor in \( O(n^{r+1}) \) steps. Once we find a factor \( z \), we repeat the procedure with \( y/z \). There can be at most \( n = \log y \) repetitions of this, since \( y < 2^n \), and thus its prime factorization contains fewer than \( n \) factors. Thus we obtain the prime factorization of \( y \) in \( O(n^{r+2}) \) steps.

For example, let \( y = 31349 \). (For purposes of complexity, working with the decimal representation is no different from working with the binary representation.) We would find that \( y \) has factors less than 15674, 7837, 3918, 1959, 979, 489, 244, 122, 61, 30, but not less than 15. We now test halfway between 15 and 30 (22) and find no factor less than 22, then halfway between 22 and 30 (26), halfway between 22 and 26 (24), and halfway between 22 and 24 and arrive at the factor 23. The number of calls to the algorithm for \( L \) we had to make in order to get to this point (fifteen) looks large, but is really bounded above by the ceiling of the base 2 logarithm of \( y \) and thus linear in the input size.

We now divide 31349 by 23 and repeat the procedure with the quotient 1363 and test the following values of \( z \): 681, 340, 170, 85, 42, 21, 26, 34, 30, 28, 29. Again, we had to perform \( 11 = \lceil \log_2 y \rceil \) calls to the algorithm for \( L \). We would now find that 1363/29=47, and we get have complete prime factorization of \( y \).