Contents

Preface                                               v

1 Introduction                                      1

2 Finite Automata                                   3
   2.1 Turing Machines                               3
   2.2 Introduction to Finite Automata               3
   2.3 More Examples                                 9
   2.4 Formal Definition                             13
   2.5 Closure Properties                            19
Preface

This document contains solutions to the exercises of the course notes *Automata and Computability*. These notes were written for the course CS345 *Automata Theory and Formal Languages* taught at Clarkson University. The course is also listed as MA345 and CS541. The solutions are organized according to the same chapters and sections as the notes.

Here’s some advice. Whether you are studying these notes as a student in a course or in self-directed study, your goal should be to understand the material well enough that you can do the exercises on your own. Simply studying the solutions is not the best way to achieve this. It is much better to spend a reasonable amount of time and effort trying to do the exercises yourself before looking at the solutions.

If you can’t do an exercise on your own, you should study the notes some more. If that doesn’t work, seek help from another student or from your instructor. Look at the solutions only to check your answer once you think you know how to do an exercise.

If you needed help doing an exercise, try redoing the same exercise later on your own. And do additional exercises.

If your solution to an exercise is different from the solution in this document, take the time to figure out why. Did you make a mistake? Did you forget some-
thing? Did you discover another correct solution? If you’re not sure, ask for help from another student or the instructor. If your solution turns out to be incorrect, fix it, after maybe getting some help, then try redoing the same exercise later on your own and do additional exercises.

Feedback on the notes and solutions is welcome. Please send comments to alexis@clarkson.edu.
Chapter 1

Introduction

There are no exercises in this chapter.
Chapter 2

Finite Automata

2.1 Turing Machines

There are no exercises in this section.

2.2 Introduction to Finite Automata

2.2.3.

Missing edges go to a garbage state. In other words, the full DFA looks like this:
The transition label *other* means any character that’s not a dash or a digit.

2.2.4.

Missing edges go to a garbage state.
2.2.5.
2.2. INTRODUCTION TO FINITE AUTOMATA

2.2.7.

starting_state() { return q0 }

is_accepting(q) { return true iff q is q1 }

next_state(q, c) {
    if (q is q0)
        if (c is underscore or letter)
            return q1
        else
            return q2
    else if (q is q1)
        if (c is underscore, letter or digit)
            return q1
        else
            return q2
    else /* q is q2 */
        return q2
}

2.2.8. The following assumes that the garbage state is labeled $q_9$. In the pseudocode algorithm, states are stored as integers. This is more convenient here.

starting_state() { return 0 }

is_accepting(q) { return true iff q is 8 }
next_state(q, c) {
    if (q in {0, 1, 2} or {4, 5, 6, 7})
        if (c is digit)
            return q + 1
        else
            return 9
    else if (q is 3)
        if (c is digit)
            return 5
        else if (c is dash)
            return 4
        else
            return 9
    else if (q is 8 or 9)
        return 9
}
2.3. More Examples

2.3.5.

2.3.6. In all cases, missing edges go to a garbage state.

a)

b)
2.3.7.

a)

b)

c)
2.3. **MORE EXAMPLES**

d) 

\[ \begin{array}{c}
0 & 0 & 0,1 \\
q_0 & 1 & q_1 & 1 & q_2 \\
\end{array} \]


e) 

\[ \begin{array}{c}
0 & 0 & 0,1 \\
q_0 & 1 & q_1 & 1 & q_2 & \cdots & 1 & q_k \\
\end{array} \]

2.3.8. In all cases, missing edges go to a garbage state.

a) 

\[ \begin{array}{c}
0 & 0 \\
q_0 & 1 & q_1 & 1 \\
\end{array} \]
b) The idea is for the DFA to remember the last two symbols it has seen.

c) Again, the idea is for the DFA to remember the last two symbols it has seen. We could simply change the accepting states of the previous DFA to \( \{q_{10}, q_{11}\} \). But we can also simplify this DFA by assuming that strings of length less than two are preceded by 00.
2.4.5. The DFA is \( (\{q_0, q_1, q_2, \ldots, q_9\}, \Sigma, \delta, q_0, \{q_8\}) \) where \( \Sigma \) is the set of all characters that appear on a standard keyboard and \( \delta \) is defined as follows:

\[
\delta(q_i, c) = \begin{cases} 
q_{i+1} & \text{if } i \notin \{3, 8, 9\} \text{ and } c \text{ is digit} \\
q_9 & \text{if } i \notin \{3, 8, 9\} \text{ and } c \text{ is not digit} 
\end{cases}
\]

\[
\delta(q_3, c) = \begin{cases} 
q_4 & \text{if } c \text{ is dash} \\
q_5 & \text{if } c \text{ is digit} \\
q_9 & \text{otherwise} 
\end{cases}
\]

\[
\delta(q_8, c) = q_9 \quad \text{for every } c
\]

\[
\delta(q_9, c) = q_9 \quad \text{for every } c
\]
b) The DFA is \( \{q_0, q_1, q_2, q_3\}, \Sigma, \delta, q_0, \{q_2\} \) where \( \Sigma \) is the set of all characters that appear on a standard keyboard and \( \delta \) is defined as follows:

\[
\delta(q_0, c) = \begin{cases} 
  q_1 & \text{if } c \text{ is dash} \\
  q_2 & \text{if } c \text{ is digit} \\
  q_3 & \text{otherwise}
\end{cases}
\]

\[
\delta(q_i, c) = \begin{cases} 
  q_2 & \text{if } i \in \{1, 2\} \text{ and } c \text{ is digit} \\
  q_3 & \text{if } i \in \{1, 2\} \text{ and } c \text{ is not digit}
\end{cases}
\]

\[\delta(q_3, c) = q_3 \text{ for every } c\]

c) The DFA is \( \{q_0, q_1, q_2, \ldots, q_5\}, \Sigma, \delta, q_0, \{q_2, q_4\} \) where \( \Sigma \) is the set of all characters that appear on a standard keyboard and \( \delta \) is defined as follows:

\[
\delta(q_0, c) = \begin{cases} 
  q_1 & \text{if } c \text{ is dash} \\
  q_2 & \text{if } c \text{ is digit} \\
  q_3 & \text{if } c \text{ is decimal point} \\
  q_5 & \text{otherwise}
\end{cases}
\]

\[
\delta(q_i, c) = \begin{cases} 
  q_2 & \text{if } i \in \{1, 2\} \text{ and } c \text{ is digit} \\
  q_3 & \text{if } i \in \{1, 2\} \text{ and } c \text{ is decimal point} \\
  q_5 & \text{if } i \in \{1, 2\} \text{ and } c \text{ is not digit or decimal point}
\end{cases}
\]

\[
\delta(q_i, c) = \begin{cases} 
  q_4 & \text{if } i \in \{3, 4\} \text{ and } c \text{ is digit} \\
  q_5 & \text{if } i \in \{3, 4\} \text{ and } c \text{ is not digit}
\end{cases}
\]

\[\delta(q_5, c) = q_5 \text{ for every } c\]
2.4.6. The idea is for the DFA to remember the last $k$ symbols it has seen. But this is too difficult to draw clearly, so here’s a formal description of the DFA: $(Q, \{0, 1\}, \delta, q_0, F)$ where

\[
Q = \{q_w \mid w \in \{0, 1\}^* \text{ and } w \text{ has length } k\}
\]

\[
q_0 = q_{w_0} \quad \text{where } w_0 = 0^k \text{ (that is, a string of } k \text{ 0’s)}
\]

\[
F = \{q_w \in Q \mid w \text{ starts with a } 1\}
\]

and $\delta$ is defined as follows:

\[
\delta(q_{au}, b) = q_{ub}
\]

where $a \in \Sigma$, $u$ is a string of length $k - 1$ and $b \in \Sigma$.

2.4.7.

a) The idea is for the DFA to store the value, modulo 3, of the portion of the number it has seen so far, and then update that value for every additional digit that is read. To update the value, the current value is multiplied by 10, the new digit is added and the result is reduced modulo 3.
(Note that this is exactly the same DFA we designed in an example of this section for the language of strings that have the property that the sum of their digits is a multiple of 3. This is because $10 \mod 3 = 1$ so that when we multiply the current value by 10 and reduce modulo 3, we are really just multiplying by 1. Which implies that the strategy we described above is equivalent to simply adding the digits of the number, modulo 3.)

b) We use the same strategy that was described in the first part, but this time, we reduce modulo $k$. Here's a formal description of the DFA: $(Q, \Sigma, \delta, q_0, F)$ where

$$Q = \{q_0, q_1, q_2, \ldots, q_{k-1}\}$$

$$\Sigma = \{0, 1, 2, \ldots, 9\}$$

$$F = \{q_0\}$$

and $\delta$ is defined as follows: for every $i \in Q$ and $c \in \Sigma$,

$$\delta(q_i, c) = q_j \quad \text{where} \quad j = (i \cdot 10 + c) \mod k.$$
a) The idea is for the DFA to verify, for each input symbol, that the third digit is the sum of the first two plus any carry that was previously generated, as well as determine if a carry is generated. All that the DFA needs to remember is the value of the carry (0 or 1). The DFA accepts if no carry is generated when processing the last input symbol. Here's a formal description of the DFA, where state $q_2$ is a garbage state: $(Q, \Sigma, \delta, q_0, F)$ where

\[
Q = \{q_0, q_1, q_2\} \\
\Sigma = \{[abc] \mid a, b, c \in \{0, 1, 2, \ldots, 9\}\} \\
F = \{q_0\}
\]

and $\delta$ is defined as follows:

\[
\delta(q_d, [abc]) = \begin{cases} 
q_0 & \text{if } d \in \{0, 1\} \text{ and } d + a + b = c \\
q_1 & \text{if } d \in \{0, 1\}, d + a + b \geq 10 \text{ and } (d + a + b) \mod 10 = c \\
q_2 & \text{otherwise}
\end{cases}
\]

Here's a transition diagram of the DFA that shows only one of the 1,000 transitions that come out of each state.
b) Since the DFA is now reading the numbers from left to right, it can’t compute the carries as it reads the numbers. So it will do the opposite: for each input symbol, the DFA will figure out what carry it needs from the rest of the numbers. For example, if the first symbol that the DFA sees is $[123]$, the DFA will know that there should be no carry generated from the rest of the numbers. But if the symbol is $[124]$, the DFA needs the rest of the number to generate a carry. And if a carry needs to be generated, the next symbol will have to be something like $[561]$ but not $[358]$. The states of the DFA will be used to remember the carry that is needed from the rest of the numbers. The DFA will accept if no carry is needed for the first position of the numbers (which is given by the last symbol of the input string). Here’s a formal description of the DFA, where state $q_2$ is a garbage state: $(Q, \Sigma, \delta, q_0, F)$ where

\[
Q = \{q_0, q_1, q_2\} \\
\Sigma = \{[abc] \mid a, b, c \in \{0, 1, 2, \ldots, 9\}\} \\
F = \{q_0\}
\]

and $\delta$ is defined as follows:

\[
\delta(q_0, [abc]) = \begin{cases} 
q_d & \text{if } d \in \{0, 1\} \text{ and } d + a + b = c \\
q_2 & \text{otherwise}
\end{cases}
\]

\[
\delta(q_1, [abc]) = \begin{cases} 
q_d & \text{if } d \in \{0, 1\}, d + a + b \geq 10 \text{ and } (d + a + b) \mod 10 = c \\
q_2 & \text{otherwise}
\end{cases}
\]

\[
\delta(q_2, [abc]) = q_2, \text{ for every } [abc] \in \Sigma
\]
2.5. Closure Properties

2.5.3. In each case, all we have to do is switch the acceptance status of each state. But we need to remember to do it for the garbage states too.

a)
2.5.4. It is important to include in the pair construction the garbage states of the DFA’s for the simpler languages. (This is actually not needed for intersections but it is critical for unions.) In each case, we give the DFA’s for the two simpler languages followed by the DFA obtained by the pair construction.
2.5. CLOSURE PROPERTIES

a)
b)

\[ q_0 \xrightarrow{0} \quad 1 \quad \xrightarrow{0} \quad 0,1 \]

\[ q_0 \xrightarrow{1} \quad q_1 \xrightarrow{1} \quad q_2 \]

\[ q_0 \xrightarrow{0} \quad q_1 \xrightarrow{1} \quad q_2 \xrightarrow{0,1} \]
2.5. CLOSURE PROPERTIES

c)}
2.5. CLOSURE PROPERTIES

d)

\begin{center}
\begin{tikzpicture}[node distance = 2cm, thick, main/.style = {draw, circle}]
\node[main] (q0) {$q_0$};
\node[main] (q1) [right of=q0] {$q_1$};
\node[main] (q2) [right of=q1] {$q_2$};
\path[->] (q0) edge node {$0$} (q1) edge[bend left] node {$1$} (q0);
\path[->] (q1) edge node {$0$} (q2) edge[bend left] node {$1$} (q1);
\path[->] (q2) edge node {$0,1$} (q2);
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}[node distance = 2cm, thick, main/.style = {draw, circle}]
\node[main] (q0) {$q_0$};
\node[main] (q1) [below of=q0] {$q_1$};
\path[->] (q0) edge node {$0$} (q1) edge[bend left] node {$1$} (q0);
\path[->] (q1) edge[bend left] node {$0$} (q1);
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}[node distance = 2cm, thick, main/.style = {draw, circle}]
\node[main] (q0) {$q_{00}$};
\node[main] (q1) [right of=q0] {$q_{10}$};
\node[main] (q2) [right of=q1] {$q_{20}$};
\path[->] (q0) edge node {$0$} (q1) edge node {$0$} (q0);
\path[->] (q1) edge node {$0$} (q2) edge node {$0$} (q1);
\path[->] (q2) edge node {$0$} (q2);
\end{tikzpicture}
\end{center}
2.5.5. In both cases, missing edges go to a garbage state.

a)

![Diagram 1](image1.png)

b) The dashed state and edge could be deleted.

![Diagram 2](image2.png)