UNDERGRADUATE TEXTS IN COMPUTER SCIENCE

Automata and Computability

DEXTER C. KOZEN

Springer

Dexter C. Kozen

Automata and Computability

Springer

Dexter C. Kozen
Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
USA

Series Editors
David Gries
Department of Computer Science
415 Boyd Studies Research Center
The University of Georgia
Athens, Georgia 30602
USA

Fred B. Schneider
Department of Computer Science
Cornell University
4115C Upson Hall
Ithaca, NY 14853-7501
USA

On the cover: Cover photo taken by John Still/Photonica.

With 1 figure.

Library of Congress Cataloging-in-Publication Data
Kozen, Dexter, 1951-
Automata and computability/Dexter C. Kozen.
p. cm. — (Undergraduate texts in computer science)
Includes bibliographical references and index.
ISBN 0-387-94907-0 (hardcover: alk. paper)
1. Machine theory. 2. Computable functions. I. Title.
II. Series.
QA267.K69 1997
511.3—dc21 96-37409

Printed on acid-free paper.

© 1997 Springer Science+Business Media, Inc.
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, Inc., 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now know or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks and similar terms, even if the are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed in the United States of America. (SBA)

9 8 7

ISBN 0-387-94907-0

springeronline.com

To Juris

Preface

These are my lecture notes from CS381/481: Automata and Computability Theory, a one-semester senior-level course I have taught at Cornell University for many years. I took this course myself in the fall of 1974 as a first-year Ph.D. student at Cornell from Juris Hartmanis and have been in love with the subject ever since.

The course is required for computer science majors at Cornell. It exists in two forms: CS481, an honors version; and CS381, a somewhat gentler-paced version. The syllabus is roughly the same, but CS481 goes deeper into the subject, covers more material, and is taught at a more abstract level. Students are encouraged to start off in one or the other, then switch within the first few weeks if they find the other version more suitable to their level of mathematical skill.

The purpose of the course is twofold: to introduce computer science students to the rich heritage of models and abstractions that have arisen over the years; and to develop the capacity to form abstractions of their own and reason in terms of them.

The course is quite mathematical in flavor, and a certain degree of previous mathematical experience is essential for survival. Students should already be conversant with elementary discrete mathematics, including the notions of set, function, relation, product, partial order, equivalence relation, graph, and tree. They should have a repertoire of basic proof techniques at their disposal, including a thorough understanding of the principle of mathematical induction.

The material covered in this text is somewhat more than can be covered in a one-semester course. It is also a mix of elementary and advanced topics. The basic course consists of the lectures numbered 1 through 39. Additionally, I have included several supplementary lectures numbered A through K on various more advanced topics. These can be included or omitted at the instructor's discretion or assigned as extra reading. They appear in roughly the order in which they should be covered.

At first these notes were meant to supplement and not supplant a textbook, but over the years they gradually took on a life of their own. In addition to the notes, I depended on various texts at one time or another: Cutland [30], Harrison [55], Hopcroft and Ullman [60], Lewis and Papadimitriou [79], Machtey and Young [81], and Manna [82]. In particular, the Hopcroft and Ullman text was the standard textbook for the course for many years, and for me it has been an indispensable source of knowledge and insight. All of these texts are excellent references, and I recommend them highly.

In addition to the lectures, I have included 12 homework sets and several miscellaneous exercises. Some of the exercises come with hints and/or solutions; these are indicated by the annotations "H" and "S," respectively. In addition, I have annotated exercises with zero to three stars to indicate relative difficulty.

I have stuck with the format of my previous textbook [72], in which the main text is divided into more or less self-contained lectures, each 4 to 8 pages. Although this format is rather unusual for a textbook, I have found it quite successful. Many readers have commented that they like it because it partitions the subject into bite-sized chunks that can be covered more or less independently.

I owe a supreme debt of gratitude to my wife Frances for her constant love, support, and superhuman patience, especially during the final throes of this project. I am also indebted to the many teachers, colleagues, teaching assistants, and students who over the years have shared the delights of this subject with me and from whom I have learned so much. I would especially like to thank Rick Aaron, Arash Baratloo, Jim Baumgartner, Steve Bloom, Manuel Blum, Amy Briggs, Ashok Chandra, Wilfred Chen, Allan Cheng, Francis Chu, Bob Constable, Devdatt Dubhashi, Peter van Emde Boas, Allen Emerson, András Ferencz, Jeff Foster, Sophia Georgiakaki, David Gries, Joe Halpern, David Harel, Basil Hayek, Tom Henzinger, John Hopcroft, Nick Howe, Doug Ierardi, Tibor Janosi, Jim Jennings, Shyam Kapur, Steve Kautz, Nils Klarlund, Peter Kopke, Vladimir Kotlyar, Alan Kwan, Georges Lauri, Michael Leventon, Jake Levirne, David Liben-Nowell, Yvonne Lo, Steve Mahaney, Nikolay Mateev, Frank McSherry, Albert Meyer, Bob Milnikel, Francesmary Modugno, Anil Nerode, Damian Niwiński, David de la Nuez, Dan Oberlin, Jens Palsberg, Rohit

Parikh, David Pearson, Paul Pedersen, Vaughan Pratt, Zulfikar Ramzan, Jon Rosenberger, Jonathan Rynd, Erik Schmidt, Michael Schwartzbach, Amitabh Shah, Frederick Smith, Kjartan Stefánsson, Colin Stirling, Larry Stockmeyer, Aaron Stump, Jurek Tiuryn, Alex Tsow, Moshe Vardi, Igor Walukiewicz, Rafael Weinstein, Jim Wen, Dan Wineman, Thomas Yan, Paul Zimmons, and many others too numerous to mention. Of course, the greatest of these is Juris Hartmanis, whose boundless enthusiasm for the subject is the ultimate source of my own.

I would be most grateful for suggestions and criticism from readers.

Note added for the third printing. I am indebted to Chris Jeuell for pointing out several typographical errors, which have been corrected in this printing.

Ithaca, New York Dexter C. Kozen

Contents

Lectures

Lecture 1

Course Roadmap and Historical Perspective

The goal of this course is to understand the foundations of computation. We will ask some very basic questions, such as

- What does it mean for a function to be computable?
- Are there any noncomputable functions?
- How does computational power depend on programming constructs?

These questions may appear simple, but they are not. They have intrigued scientists for decades, and the subject is still far from closed.

In the quest for answers to these questions, we will encounter some fundamental and pervasive concepts along the way: *state, transition, nondeterminism, reduction*, and *undecidability*, to name a few. Some of the most important achievements in theoretical computer science have been the crystallization of these concepts. They have shown a remarkable persistence, even as technology changes from day to day. They are crucial for every good computer scientist to know, so that they can be recognized when they are encountered, as they surely will be.

Various models of computation have been proposed over the years, all of which capture some fundamental aspect of computation. We will concentrate on the following three classes of models, in order of increasing power:

(i) finite memory: finite automata, regular expressions;

(ii) finite memory with stack: pushdown automata;

(iii) unrestricted:

- Turing machines (Alan Turing [120]),
- Post systems (Emil Post [99, 100]),
- μ-recursive functions (Kurt Gödel [51], Jacques Herbrand),
- λ-calculus (Alonzo Church [23], Stephen C. Kleene [66]),
- combinatory logic (Moses Schönfinkel [111], Haskell B. Curry [29]).

These systems were developed long before computers existed. Nowadays one could add PASCAL, FORTRAN, BASIC, LISP, SCHEME, C++, JAVA, or any sufficiently powerful programming language to this list.

In parallel with and independent of the development of these models of computation, the linguist Noam Chomsky attempted to formalize the notion of *grammar* and *language*. This effort resulted in the definition of the *Chomsky hierarchy*, a hierarchy of language classes defined by grammars of increasing complexity:

(i) right-linear grammars;

(ii) context-free grammars;

(iii) unrestricted grammars.

Although grammars and machine models appear quite different on a superficial level, the process of parsing a sentence in a language bears a strong resemblance to computation. Upon closer inspection, it turns out that each of the grammar types (i), (ii), and (iii) are equivalent in computational power to the machine models (i), (ii), and (iii) above, respectively. There is even a fourth natural class called the *context-sensitive* grammars and languages, which fits in between (ii) and (iii) and which corresponds to a certain natural class of machine models called *linear bounded automata.*

It is quite surprising that a naturally defined hierarchy in one field should correspond so closely to a naturally defined hierarchy in a completely different field. Could this be mere coincidence?

Abstraction

The machine models mentioned above were first identified in the same way that theories in physics or any other scientific discipline arise. When studying real-world phenomena, one becomes aware of recurring patterns and themes that appear in various guises. These guises may differ substantially on a superficial level but may bear enough resemblance to one another to suggest that there are common underlying principles at work. When this happens, it makes sense to try to construct an abstract model that captures these underlying principles in the simplest possible way, devoid of the unimportant details of each particular manifestation. This is the process of *abstraction.* Abstraction is the essence of scientific progress, because it focuses attention on the important principles, unencumbered by irrelevant details.

Perhaps the most striking example of this phenomenon we will see is the formalization of the concept of *effective computability.* This quest started around the beginning of the twentieth century with the development of the *formalist* school of mathematics, championed by the philosopher Bertrand Russell and the mathematician David Hilbert. They wanted to reduce all of mathematics to the formal manipulation of symbols.

Of course, the formal manipulation of symbols is a form of computation, although there were no computers around at the time. However, there certainly existed an awareness of computation and algorithms. Mathematicians, logicians, and philosophers knew a constructive method when they saw it. There followed several attempts to come to grips with the general notion of *effective computability.* Several definitions emerged (Turing machines, Post systems, etc.), each with its own peculiarities and differing radically in appearance. However, it turned out that as different as all these formalisms appeared to be, they could all simulate one another, thus they were all computationally equivalent.

The formalist program was eventually shattered by Kurt Gödel's incompleteness theorem, which states that no matter how strong a deductive system for number theory you take, it will always be possible to construct simple statements that are true but unprovable. This theorem is widely regarded as one of the crowning intellectual achievements of twentieth century mathematics. It is essentially a statement about computability, and we will be in a position to give a full account of it by the end of the course.

The process of abstraction is inherently mathematical. It involves building models that capture observed behavior in the simplest possible way. Although we will consider plenty of concrete examples and applications of these models, we will work primarily in terms of their mathematical properties. We will always be as explicit as possible about these properties.

We will usually start with definitions, then subsequently reason purely in terms of those definitions. For some, this will undoubtedly be a new way of thinking, but it is a skill that is worth cultivating.

Keep in mind that a large intellectual effort often goes into coming up with just the right definition or model that captures the essence of the principle at hand with the least amount of extraneous baggage. After the fact, the reader often sees only the finished product and is not exposed to all the misguided false attempts and pitfalls that were encountered along the way. Remember that it took many years of intellectual struggle to arrive at the theory as it exists today. This is not to say that the book is closed—far from it!

Lecture 2

Strings and Sets

Decision Problems Versus Functions

A *decision problem* is a function with a one-bit output: "yes" or "no." To specify a decision problem, one must specify

- the set A of possible inputs, and
- the subset $B \subseteq A$ of "yes" instances.

For example, to decide if a given graph is connected, the set of possible inputs is the set of all (encodings of) graphs, and the "yes" instances are the connected graphs. To decide if a given number is a prime, the set of possible inputs is the set of all (binary encodings of) integers, and the "yes" instances are the primes.

In this course we will mostly consider decision problems as opposed to functions with more general outputs. We do this for mathematical simplicity and because the behavior we want to study is already present at this level.

Strings

Now to our first abstraction: we will always take the set of possible inputs to a decision problem to be the set of finite-length strings over some fixed finite

alphabet (formal definitions below). We do this for uniformity and simplicity. Other types of data—graphs, the natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$, trees, even programs—can be encoded naturally as strings. By making this abstraction, we have to deal with only one data type and a few basic operations.

Definition 2.1

- An *alphabet* is any finite set. For example, we might use the alphabet $\{0, 1, 2, \ldots, 9\}$ if we are talking about decimal numbers; the set of all ASCII characters if talking about text; $\{0, 1\}$ if talking about bit strings. The only restriction is that the alphabet be finite. When speaking about an arbitrary finite alphabet abstractly, we usually denote it by the Greek letter Σ. We call elements of Σ *letters* or *symbols* and denote them by $a, b, c, \ldots$. We usually do not care at all about the nature of the elements of Σ, only that there are finitely many of them.

- A *string* over Σ is any finite-length sequence of elements of Σ. Example: if $\Sigma = \{a, b\}$, then $aabab$ is a string over Σ of length five. We use $x, y, z, \ldots$ to refer to strings.

- The *length* of a string x is the number of symbols in x. The length of x is denoted $|x|$. For example, $|aabab| = 5$.

- There is a unique string of length 0 over Σ called the *null string* or *empty string* and denoted by ϵ (Greek epsilon, not to be confused with the symbol for set containment $\in$). Thus $|\epsilon| = 0$.

- We write a^n for a string of a's of length n. For example, $a^5 = aaaaa$, $a^1 = a$, and $a^0 = \epsilon$. Formally, a^n is defined inductively:

$$\begin{aligned} a^0 &\stackrel{\text{def}}{=} \epsilon, \\ a^{n+1} &\stackrel{\text{def}}{=} a^n a. \end{aligned}$$

- The set of all strings over alphabet Σ is denoted Σ^*. For example,

$$\begin{aligned} \{a, b\}^* &= \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}, \\ \{a\}^* &= \{\epsilon, a, aa, aaa, aaaa, \ldots\} \\ &= \{a^n \mid n \geq 0\}. \end{aligned}$$

□

By convention, we take

$$\varnothing^* \stackrel{\text{def}}{=} \{\epsilon\},$$

where $\varnothing$ denotes the empty set. This may seem a bit strange, but there is good mathematical justification for it, which will become apparent shortly.

If Σ is nonempty, then Σ^* is an infinite set of finite-length strings. Be careful not to confuse strings and sets. We won't see any infinite strings until much later in the course. Here are some differences between strings and sets:

- $\{a, b\} = \{b, a\}$, but $ab \neq ba$;
- $\{a, a, b\} = \{a, b\}$, but $aab \neq ab$.

Note also that $\varnothing$, $\{\epsilon\}$, and ϵ are three different things. The first is a set with no elements; the second is a set with one element, namely ϵ; and the last is a string, not a set.

Operations on Strings

The operation of *concatenation* takes two strings x and y and makes a new string xy by putting them together end to end. The string xy is called the *concatenation* of x and y. Note that xy and yx are different in general. Here are some useful properties of concatenation.

- concatenation is *associative*: $(xy)z = x(yz)$;
- the null string ϵ is an *identity* for concatenation: $\epsilon x = x\epsilon = x$;
- $|xy| = |x| + |y|$.

A special case of the last equation is $a^m a^n = a^{m+n}$ for all $m, n \geq 0$.

A *monoid* is any algebraic structure consisting of a set with an associative binary operation and an identity for that operation. By our definitions above, the set Σ^* with string concatenation as the binary operation and ϵ as the identity is a monoid. We will see some other examples later in the course.

Definition 2.2

- We write x^n for the string obtained by concatenating n copies of x. For example, $(aab)^5 = aabaabaabaabaab$, $(aab)^1 = aab$, and $(aab)^0 = \epsilon$. Formally, x^n is defined inductively:

$$x^0 \stackrel{\text{def}}{=} \epsilon,$$
$$x^{n+1} \stackrel{\text{def}}{=} x^n x.$$

- If $a \in \Sigma$ and $x \in \Sigma^*$, we write $\#a(x)$ for the number of a's in x. For example, $\#0(001101001000) = 8$ and $\#1(00000) = 0$.
- A *prefix* of a string x is an initial substring of x; that is, a string y for which there exists a string z such that $x = yz$. For example, *abaab* is a prefix of *abaababa*. The null string is a prefix of every string, and

every string is a prefix of itself. A prefix y of x is a *proper* prefix of x if $y \neq \epsilon$ and $y \neq x$. □

Operations on Sets

We usually denote sets of strings (subsets of Σ^*) by $A, B, C, \ldots$. The *cardinality* (number of elements) of set A is denoted $|A|$. The empty set $\varnothing$ is the unique set of cardinality 0.

Let's define some useful operations on sets. Some of these you have probably seen before, some probably not.

- *Set union:*

  $$A \cup B \stackrel{\text{def}}{=} \{x \mid x \in A \text{ or } x \in B\}.$$

  In other words, x is in the union of A and B iff[1] either x is in A or x is in B. For example, $\{a, ab\} \cup \{ab, aab\} = \{a, ab, aab\}$.

- *Set intersection:*

  $$A \cap B \stackrel{\text{def}}{=} \{x \mid x \in A \text{ and } x \in B\}.$$

  In other words, x is in the intersection of A and B iff x is in both A and B. For example, $\{a, ab\} \cap \{ab, aab\} = \{ab\}$.

- *Complement in* Σ^*:

  $$\sim A \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid x \notin A\}.$$

  For example,

  $$\sim \{\text{strings in } \Sigma^* \text{ of even length}\} = \{\text{strings in } \Sigma^* \text{ of odd length}\}.$$

  Unlike $\cup$ and $\cap$, the definition of $\sim$ depends on Σ^*. The set $\sim A$ is sometimes denoted $\Sigma^* - A$ to emphasize this dependence.

- *Set concatenation:*

  $$AB \stackrel{\text{def}}{=} \{xy \mid x \in A \text{ and } y \in B\}.$$

  In other words, z is in AB iff z can be written as a concatenation of two strings x and y, where $x \in A$ and $y \in B$. For example, $\{a, ab\}\{b, ba\} = \{ab, aba, abb, abba\}$. When forming a set concatenation, you include *all* strings that can be obtained in this way. Note that AB and BA are different sets in general. For example, $\{b, ba\}\{a, ab\} = \{ba, bab, baa, baab\}$.

[1] iff = if and only if.

- The *powers* A^n of a set A are defined inductively as follows:

$$A^0 \stackrel{\text{def}}{=} \{\epsilon\},$$
$$A^{n+1} \stackrel{\text{def}}{=} AA^n.$$

In other words, A^n is formed by concatenating n copies of A together. Taking $A^0 = \{\epsilon\}$ makes the property $A^{m+n} = A^m A^n$ hold, even when one of m or n is 0. For example,

$$\{ab, aab\}^0 = \{\epsilon\},$$
$$\{ab, aab\}^1 = \{ab, aab\},$$
$$\{ab, aab\}^2 = \{abab, abaab, aabab, aabaab\},$$
$$\{ab, aab\}^3 = \{ababab, ababaab, abaabab, aababab, abaabaab, aababaab, aabaabab, aabaabaab\}.$$

Also,

$$\{a, b\}^n = \{x \in \{a, b\}^* \mid |x| = n\}$$
$$= \{\text{strings over } \{a, b\} \text{ of length } n\}.$$

- The *asterate* A^* of a set A is the union of all finite powers of A:

$$A^* \stackrel{\text{def}}{=} \bigcup_{n \geq 0} A^n$$
$$= A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots.$$

Another way to say this is

$$A^* = \{x_1 x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in A,\ 1 \leq i \leq n\}.$$

Note that n can be 0; thus the null string ϵ is in A^* for any A.

We previously defined Σ^* to be the set of all finite-length strings over the alphabet Σ. This is exactly the asterate of the set Σ, so our notation is consistent.

- We define A^+ to be the union of all *nonzero* powers of A:

$$A^+ \stackrel{\text{def}}{=} AA^* = \bigcup_{n \geq 1} A^n.$$

Here are some useful properties of these set operations:

- Set union, set intersection, and set concatenation are *associative*:

$$(A \cup B) \cup C = A \cup (B \cup C),$$
$$(A \cap B) \cap C = A \cap (B \cap C),$$
$$(AB)C = A(BC).$$

- Set union and set intersection are *commutative*:

$$A \cup B = B \cup A,$$
$$A \cap B = B \cap A.$$

As noted above, set concatenation is not.

- The null set $\varnothing$ is an *identity* for $\cup$:

$$A \cup \varnothing = \varnothing \cup A = A.$$

- The set $\{\epsilon\}$ is an identity for set concatenation:

$$\{\epsilon\}A = A\{\epsilon\} = A.$$

- The null set $\varnothing$ is an *annihilator* for set concatenation:

$$A\varnothing = \varnothing A = \varnothing.$$

- Set union and intersection *distribute* over each other:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C),$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

- Set concatenation distributes over union:

$$A(B \cup C) = AB \cup AC,$$
$$(A \cup B)C = AC \cup BC.$$

In fact, concatenation distributes over the union of any family of sets. If $\{B_i \mid i \in I\}$ is any family of sets indexed by another set I, finite or infinite, then

$$A(\bigcup_{i \in I} B_i) = \bigcup_{i \in I} AB_i,$$
$$(\bigcup_{i \in I} B_i)A = \bigcup_{i \in I} B_i A.$$

Here $\bigcup_{i \in I} B_i$ denotes the union of all the sets B_i for $i \in I$. An element x is in this union iff it is in one of the B_i.

Set concatenation does *not* distribute over intersection. For example, take $A = \{a, ab\}$, $B = \{b\}$, $C = \{\epsilon\}$, and see what you get when you compute $A(B \cap C)$ and $AB \cap AC$.

- The *De Morgan laws* hold:

$$\sim(A \cup B) = \sim A \cap \sim B,$$
$$\sim(A \cap B) = \sim A \cup \sim B.$$

- The asterate operation * satisfies the following properties:

$$A^*A^* = A^*,$$
$$A^{**} = A^*,$$
$$A^* = \{\epsilon\} \cup AA^* = \{\epsilon\} \cup A^*A,$$
$$\varnothing^* = \{\epsilon\}.$$

Lecture 3

Finite Automata and Regular Sets

States and Transitions

Intuitively, a *state* of a system is an instantaneous description of that sy
tem, a snapshot of reality frozen in time. A state gives all relevant info
mation necessary to determine how the system can evolve from that poi
on. *Transitions* are changes of state; they can happen spontaneously or i
response to external inputs.

We assume that state transitions are instantaneous. This is a mathematic
abstraction. In reality, transitions usually take time. Clock cycles in dig
tal computers enforce this abstraction and allow us to treat computers
digital instead of analog devices.

There are innumerable examples of state transition systems in the re
world: electronic circuits, digital watches, elevators, Rubik's cube (54!/9
states and 12 transitions, not counting peeling the little sticky squares off
the game of Life (2^k states on a screen with k cells, one transition).

A system that consists of only finitely many states and transitions amo
them is called a *finite-state transition system.* We model these abstract
by a mathematical model called a *finite automaton.*

Finite Automata

Formally, a *deterministic finite automaton* (DFA) is a structure

$$M = (Q, \Sigma, \delta, s, F),$$

where

- Q is a finite set; elements of Q are called *states*;
- Σ is a finite set, the *input alphabet*;
- $\delta : Q \times \Sigma \to Q$ is the *transition function* (recall that $Q \times \Sigma$ is the set of ordered pairs $\{(q, a) \mid q \in Q \text{ and } a \in \Sigma\}$). Intuitively, δ is a function that tells which state to move to in response to an input: if M is in state q and sees input a, it moves to state $\delta(q, a)$.
- $s \in Q$ is the *start state*;
- F is a subset of Q; elements of F are called *accept* or *final states*.

When you specify a finite automaton, you must give all five parts. Automata may be specified in this set-theoretic form or as a transition diagram or table as in the following example.

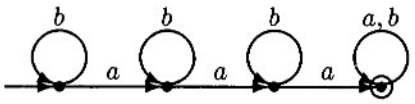
Example 3.1 Here is an example of a simple four-state finite automaton. We'll take the set of states to be $\{0, 1, 2, 3\}$; the input alphabet to be $\{a, b\}$; the start state to be 0; the set of accept states to be $\{3\}$; and the transition function to be

$$\begin{aligned}
&\delta(0, a) = 1,\\
&\delta(1, a) = 2,\\
&\delta(2, a) = \delta(3, a) = 3,\\
&\delta(q, b) = q, \quad q \in \{0, 1, 2, 3\}.
\end{aligned}$$

All parts of the automaton are completely specified. We can also specify the automaton by means of a table

		a	b
$\to$	0	1	0
	1	2	1
	2	3	2
	$3F$	3	3

or transition diagram



The final states are indicated by an F in the table and by a circle in the transition diagram. In both, the start state is indicated by $\to$. The states in

the transition diagram from left to right correspond to the states $0, 1, 2, 3$ in the table. One advantage of transition diagrams is that you don't have to name the states. □

Another convenient representation of finite automata is transition matrices; see Miscellaneous Exercise 7.

Informally, here is how a finite automaton operates. An input can be any string $x \in \Sigma^*$. Put a pebble down on the start state s. Scan the input string x from left to right, one symbol at a time, moving the pebble according to δ: if the next symbol of x is b and the pebble is on state q, move the pebble to $\delta(q, b)$. When we come to the end of the input string, the pebble is on some state p. The string x is said to be *accepted* by the machine M if $p \in F$ and *rejected* if $p \notin F$. There is no formal mechanism for scanning or moving the pebble; these are just intuitive devices.

For example, the automaton of Example 3.1, beginning in its start state 0, will be in state 3 after scanning the input string *baabbaab*, so that string is accepted; however, it will be in state 2 after scanning the string *babbbab*, so that string is rejected. For this automaton, a moment's thought reveals that when scanning any input string, the automaton will be in state 0 if it has seen no a's, state 1 if it has seen one a, state 2 if it has seen two a's, and state 3 if it has seen three or more a's.

This is how we do formally what we just described informally above. We first define a function

$$\widehat{\delta} : Q \times \Sigma^* \to Q$$

from δ by induction on the length of x:

$$\widehat{\delta}(q, \epsilon) \stackrel{\text{def}}{=} q, \tag{3.1}$$

$$\widehat{\delta}(q, xa) \stackrel{\text{def}}{=} \delta(\widehat{\delta}(q, x), a). \tag{3.2}$$

The function $\widehat{\delta}$ maps a state q and a string x to a new state $\widehat{\delta}(q, x)$. Intuitively, $\widehat{\delta}$ is the multistep version of δ. The state $\widehat{\delta}(q, x)$ is the state M ends up in when started in state q and fed the input x, moving in response to each symbol of x according to δ. Equation (3.1) is the basis of the inductive definition; it says that the machine doesn't move anywhere under the null input. Equation (3.2) is the induction step; it says that the state reachable from q under input string xa is the state reachable from p under input symbol a, where p is the state reachable from q under input string x.

Note that the second argument to $\widehat{\delta}$ can be any string in Σ^*, not just a string of length one as with δ; but $\widehat{\delta}$ and δ agree on strings of length one:

$$\begin{aligned} \widehat{\delta}(q, a) &= \widehat{\delta}(q, \epsilon a) && \text{since } a = \epsilon a \\ &= \delta(\widehat{\delta}(q, \epsilon), a) && \text{by (3.2), taking } x = \epsilon \end{aligned}$$

$$= \delta(q, a) \qquad \text{by (3.1).}$$

Formally, a string x is said to be *accepted* by the automaton M if

$$\widehat{\delta}(s, x) \in F$$

and *rejected* by the automaton M if

$$\widehat{\delta}(s, x) \notin F,$$

where s is the start state and F is the set of accept states. This captures formally the intuitive notion of acceptance and rejection described above.

The *set* or *language accepted by* M is the set of all strings accepted by M and is denoted $L(M)$:

$$L(M) \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid \widehat{\delta}(s, x) \in F\}.$$

A subset $A \subseteq \Sigma^*$ is said to be *regular* if $A = L(M)$ for some finite automaton M. The set of strings accepted by the automaton of Example 3.1 is the set

$$\{x \in \{a, b\}^* \mid x \text{ contains at least three } a\text{'s}\},$$

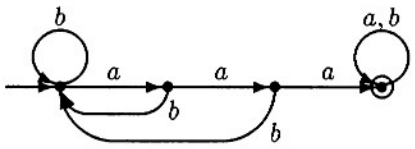
so this is a regular set.

Example 3.2 Here is another example of a regular set and a finite automaton accepting it. Consider the set

$$\{xaaay \mid x, y \in \{a, b\}^*\}$$
$$= \{x \in \{a, b\}^* \mid x \text{ contains a substring of three consecutive } a\text{'s}\}.$$

For example, *baabaaaab* is in the set and should be accepted, whereas *babbabab* is not in the set and should be rejected (because the three a's are not consecutive). Here is an automaton for this set, specified in both table and transition diagram form:

		a	b
$\rightarrow$	0	1	0
	1	2	0
	2	3	0
	$3F$	3	3



□

The idea here is that you use the states to count the number of consecutive a's you have seen. If you haven't seen three a's in a row and you see a b, you must go back to the start. Once you have seen three a's in a row, though, you stay in the accept state.

Lecture 4

More on Regular Sets

Here is another example of a regular set that is a little harder than the example given last time. Consider the set

$$\{x \in \{0,1\}^* \mid x \text{ represents a multiple of three in binary}\} \tag{4.1}$$

(leading zeros permitted, ϵ represents the number 0). For example, the following binary strings represent multiples of three and should be accepted:

Binary	*Decimal equivalent*
0	0
11	3
110	6
1001	9
1100	12
1111	15
10010	18
⋮	⋮

Strings not representing multiples of three should be rejected. Here is an automaton accepting the set (4.1):

		0	1
→	**0***F*	**0**	**1**
	1	**2**	**0**
	2	**1**	**2**

The states **0**, **1**, **2** are written in boldface to distinguish them from the input symbols $0, 1$.

In the diagram, the states are **0**, **1**, **2** from left to right. We prove that this automaton accepts exactly the set (4.1) by induction on the length of the input string. First we associate a meaning to each state:

if the number represented by the string scanned so far is[1]	*then the machine will be in state*
0 mod 3	**0**
1 mod 3	**1**
2 mod 3	**2**

Let $\#x$ denote the number represented by string x in binary. For example,

$$\#\epsilon = 0,$$
$$\#0 = 0,$$
$$\#11 = 3,$$
$$\#100 = 4,$$

and so on. Formally, we want to show that for any string x in $\{0,1\}^*$,

$$\widehat{\delta}(\mathbf{0}, x) = \mathbf{0} \text{ iff } \#x \equiv 0 \bmod 3, \tag{4.2}$$
$$\widehat{\delta}(\mathbf{0}, x) = \mathbf{1} \text{ iff } \#x \equiv 1 \bmod 3,$$
$$\widehat{\delta}(\mathbf{0}, x) = \mathbf{2} \text{ iff } \#x \equiv 2 \bmod 3,$$

or in short,

$$\widehat{\delta}(\mathbf{0}, x) = \#x \bmod 3. \tag{4.3}$$

This will be our induction hypothesis. The final result we want, namely (4.2), is a weaker consequence of (4.3), but we need the more general statement (4.3) for the induction hypothesis.

We have by elementary number theory that

$$\#(x0) = 2(\#x) + 0,$$

[1] Here $a \bmod n$ denotes the remainder when dividing a by n using ordinary integer division. We also write $a \equiv b \bmod n$ (read: a is congruent to b modulo n) to mean that a and b have the same remainder when divided by n; in other words, that n divides $b - a$. Note that $a \equiv b \bmod n$ should be parsed $(a \equiv b) \bmod n$, and that in general $a \equiv b \bmod n$ and $a = b \bmod n$ mean different things. For example, $7 \equiv 2 \bmod 5$ but not $7 = 2 \bmod 5$.

$$\#(x1) = 2(\#x) + 1,$$

or in short,

$$\#(xc) = 2(\#x) + c \tag{4.4}$$

for $c \in \{0,1\}$. From the machine above, we see that for any state $q \in \{\mathbf{0},\mathbf{1},\mathbf{2}\}$ and input symbol $c \in \{0,1\}$,

$$\delta(q,c) = (2q + c) \bmod 3. \tag{4.5}$$

This can be verified by checking all six cases corresponding to possible choices of q and c. (In fact, (4.5) would have been a great way to *define* the transition function formally—then we wouldn't have had to prove it!) Now we use the inductive definition of $\widehat{\delta}$ to show (4.3) by induction on $|x|$.

Basis

For $x = \epsilon$,

$$\begin{aligned} \widehat{\delta}(\mathbf{0},\epsilon) &= \mathbf{0} && \text{by definition of } \widehat{\delta} \\ &= \#\epsilon && \text{since } \#\epsilon = 0 \\ &= \#\epsilon \bmod 3. \end{aligned}$$

Induction step

Assuming that (4.3) is true for $x \in \{0,1\}^*$, we show that it is true for xc, where $c \in \{0,1\}$.

$$\begin{aligned} \widehat{\delta}(\mathbf{0},xc) &= \delta(\widehat{\delta}(\mathbf{0},x),c) && \text{definition of } \widehat{\delta} \\ &= \delta(\#x \bmod 3, c) && \text{induction hypothesis} \\ &= (2(\#x \bmod 3) + c) \bmod 3 && \text{by (4.5)} \\ &= (2(\#x) + c) \bmod 3 && \text{elementary number theory} \\ &= \#xc \bmod 3 && \text{by (4.4).} \end{aligned}$$

Note that each step has its reason. We used the definition of δ, which is specific to this automaton; the definition of $\widehat{\delta}$ from δ, which is the same for all automata; and elementary properties of numbers and strings.

Some Closure Properties of Regular Sets

For $A, B \subseteq \Sigma^*$, recall the following definitions:

$$\begin{aligned} A \cup B &= \{x \mid x \in A \text{ or } x \in B\} && \text{union} \\ A \cap B &= \{x \mid x \in A \text{ and } x \in B\} && \text{intersection} \\ \sim A &= \{x \in \Sigma^* \mid x \notin A\} && \text{complement} \end{aligned}$$

$$AB = \{xy \mid x \in A \text{ and } y \in B\} \qquad \text{concatenation}$$
$$A^* = \{x_1 x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in A,\ 1 \leq i \leq n\}$$
$$= A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots \qquad \text{asterate.}$$

Do not confuse set concatenation with string concatenation. Sometimes $\sim A$ is written $\Sigma^* - A$.

We show below that if A and B are regular, then so are $A \cup B$, $A \cap B$, and $\sim A$. We'll show later that AB and A^* are also regular.

The Product Construction

Assume that A and B are regular. Then there are automata

$$M_1 = (Q_1,\ \Sigma,\ \delta_1,\ s_1,\ F_1),$$
$$M_2 = (Q_2,\ \Sigma,\ \delta_2,\ s_2,\ F_2)$$

with $L(M_1) = A$ and $L(M_2) = B$. To show that $A \cap B$ is regular, we will build an automaton M_3 such that $L(M_3) = A \cap B$.

Intuitively, M_3 will have the states of M_1 and M_2 encoded somehow in its states. On input $x \in \Sigma^*$, it will simulate M_1 and M_2 simultaneously on x, accepting iff both M_1 and M_2 would accept. Think about putting a pebble down on the start state of M_1 and another on the start state of M_2. As the input symbols come in, move both pebbles according to the rules of each machine. Accept if both pebbles occupy accept states in their respective machines when the end of the input string is reached.

Formally, let

$$M_3 = (Q_3,\ \Sigma,\ \delta_3,\ s_3,\ F_3),$$

where

$$Q_3 = Q_1 \times Q_2 = \{(p, q) \mid p \in Q_1 \text{ and } q \in Q_2\},$$
$$F_3 = F_1 \times F_2 = \{(p, q) \mid p \in F_1 \text{ and } q \in F_2\},$$
$$s_3 = (s_1, s_2),$$

and let

$$\delta_3 : Q_3 \times \Sigma \to Q_3$$

be the transition function defined by

$$\delta_3((p, q), a) = (\delta_1(p, a), \delta_2(q, a)).$$

The automaton M_3 is called the *product* of M_1 and M_2. A state (p, q) of M_3 encodes a configuration of pebbles on M_1 and M_2.

Recall the inductive definition (3.1) and (3.2) of the extended transition function $\widehat{\delta}$ from Lecture 2. Applied to δ_3, this gives

$$\widehat{\delta}_3((p,q),\epsilon) = (p,q),$$
$$\widehat{\delta}_3((p,q),xa) = \delta_3(\widehat{\delta}_3((p,q),x),a).$$

Lemma 4.1 *For all $x \in \Sigma^*$,*

$$\widehat{\delta}_3((p,q),x) = (\widehat{\delta}_1(p,x),\widehat{\delta}_2(q,x)).$$

Proof. By induction on $|x|$.

Basis

For $x = \epsilon$,

$$\widehat{\delta}_3((p,q),\epsilon) = (p,q) = (\widehat{\delta}_1(p,\epsilon),\widehat{\delta}_2(q,\epsilon)).$$

Induction step

Assuming the lemma holds for $x \in \Sigma^*$, we show that it holds for xa, where $a \in \Sigma$.

$$\begin{aligned}
&\widehat{\delta}_3((p,q),xa) \\
&= \delta_3(\widehat{\delta}_3((p,q),x),a) && \text{definition of } \widehat{\delta}_3 \\
&= \delta_3((\widehat{\delta}_1(p,x),\widehat{\delta}_2(q,x)),a) && \text{induction hypothesis} \\
&= (\delta_1(\widehat{\delta}_1(p,x),a),\delta_2(\widehat{\delta}_2(q,x),a)) && \text{definition of } \delta_3 \\
&= (\widehat{\delta}_1(p,xa),\widehat{\delta}_2(q,xa)) && \text{definition of } \widehat{\delta}_1 \text{ and } \widehat{\delta}_2. \qquad \square
\end{aligned}$$

Theorem 4.2 $L(M_3) = L(M_1) \cap L(M_2)$.

Proof. For all $x \in \Sigma^*$,

$$\begin{aligned}
&x \in L(M_3) \\
&\iff \widehat{\delta}_3(s_3,x) \in F_3 && \text{definition of acceptance} \\
&\iff \widehat{\delta}_3((s_1,s_2),x) \in F_1 \times F_2 && \text{definition of } s_3 \text{ and } F_3 \\
&\iff (\widehat{\delta}_1(s_1,x),\widehat{\delta}_2(s_2,x)) \in F_1 \times F_2 && \text{Lemma 4.1} \\
&\iff \widehat{\delta}_1(s_1,x) \in F_1 \text{ and } \widehat{\delta}_2(s_2,x) \in F_2 && \text{definition of set product} \\
&\iff x \in L(M_1) \text{ and } x \in L(M_2) && \text{definition of acceptance} \\
&\iff x \in L(M_1) \cap L(M_2) && \text{definition of intersection.} \quad \square
\end{aligned}$$

To show that regular sets are closed under complement, take a deterministic automaton accepting A and interchange the set of accept and nonaccept states. The resulting automaton accepts exactly when the original automaton would reject, so the set accepted is $\sim A$.

Once we know regular sets are closed under $\cap$ and $\sim$, it follows that they are closed under $\cup$ by one of the De Morgan laws:

$$A \cup B = \sim(\sim A \cap \sim B).$$

If you use the constructions for $\cap$ and $\sim$ given above, this gives an automaton for $A \cup B$ that looks exactly like the product automaton for $A \cap B$, except that the accept states are

$$F_3 = \{(p, q) \mid p \in F_1 \text{ or } q \in F_2\} = (F_1 \times Q_2) \cup (Q_1 \times F_2)$$

instead of $F_1 \times F_2$.

Historical Notes

Finite-state transition systems were introduced by McCulloch and Pitts in 1943 [84]. Deterministic finite automata in the form presented here were studied by Kleene [70]. Our notation is borrowed from Hopcroft and Ullman [60].

Lecture 5

Nondeterministic Finite Automata

Nondeterminism

Nondeterminism is an important abstraction in computer science. It refers to situations in which the next state of a computation is not uniquely determined by the current state. Nondeterminism arises in real life when there is incomplete information about the state or when there are external forces at work that can affect the course of a computation. For example, the behavior of a process in a distributed system might depend on messages from other processes that arrive at unpredictable times with unpredictable contents.

Nondeterminism is also important in the design of efficient algorithms. There are many instances of important combinatorial problems with efficient nondeterministic solutions but no known efficient deterministic solution. The famous $P = NP$ problem—whether all problems solvable in nondeterministic polynomial time can be solved in deterministic polynomial time—is a major open problem in computer science and arguably one of the most important open problems in all of mathematics.

In nondeterministic situations, we may not know how a computation will evolve, but we may have some idea of the range of possibilities. This is modeled formally by allowing automata to have multiple-valued transition functions.

In this lecture and the next, we will show how nondeterminism is incorporated naturally in the context of finite automata. One might think that adding nondeterminism might increase expressive power, but in fact for finite automata it does not: in terms of the sets accepted, nondeterministic finite automata are no more powerful than deterministic ones. In other words, for every nondeterministic finite automaton, there is a deterministic one accepting the same set. However, nondeterministic machines may be exponentially more succinct.

Nondeterministic Finite Automata

A *nondeterministic finite automaton* (NFA) is one for which the next state is not necessarily uniquely determined by the current state and input symbol. In a deterministic automaton, there is exactly one start state and exactly one transition out of each state for each symbol in Σ. In a nondeterministic automaton, there may be one, more than one, or zero. The set of *possible* next states that the automaton may move to from a particular state q in response to a particular input symbol a is part of the specification of the automaton, but there is no mechanism for deciding which one will actually be taken. Formally, we won't be able to represent this with a function $\delta : Q \times \Sigma \to Q$ anymore; we will have to use something more general. Also, a nondeterministic automaton may have many start states and may start in any one of them.

Informally, a nondeterministic automaton is said to *accept* its input x if it is possible to start in some start state and scan x, moving according to the transition rules and making choices along the way whenever the next state is not uniquely determined, such that when the end of x is reached, the machine is in an accept state. Because the start state is not determined and because of the choices along the way, there may be several possible paths through the automaton in response to the input x; some may lead to accept states while others may lead to reject states. The automaton is said to *accept* x if *at least one* computation path on input x starting from *at least one* start state leads to an accept state. The automaton is said to *reject* x if *no* computation path on input x from *any* start state leads to an accept state. Another way of saying this is that x is accepted iff there exists a path with label x from some start state to some accept state. Again, there is no mechanism for determining which state to start in or which of the possible next moves to take in response to an input symbol.

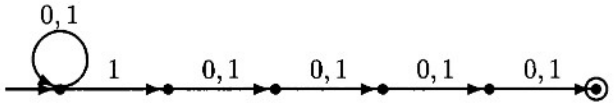
It is helpful to think about this process in terms of *guessing* and *verifying*. On a given input, imagine the automaton *guessing* a successful computation or proof that the input is a "yes" instance of the decision problem, then *verifying* that its guess was indeed correct.

For example, consider the set

$$A = \{x \in \{0,1\}^* \mid \text{the fifth symbol from the right is } 1\}.$$

Thus $11010010 \in A$ but $11000010 \notin A$.

Here is a six-state nondeterministic automaton accepting A:



There is only one start state, namely the leftmost, and only one accept state, namely the rightmost. The automaton is not deterministic, because there are two transitions from the leftmost state labeled 1 (one back to itself and one to the second state) and no transitions from the rightmost state. This automaton accepts the set A, because for any string x whose fifth symbol from the right is 1, *there exists* a sequence of legal transitions leading from the start state to the accept state (it moves from the first state to the second when it scans the fifth symbol from the right); and for any string x whose fifth symbol from the right is 0, there is *no possible* sequence of legal transitions leading to the accept state, no matter what choices it makes (recall that to accept, the machine must be in an accept state when the end of the input string is reached).

Intuitively, we can think of the machine in the leftmost state as *guessing*, every time it sees a 1, whether that 1 is the fifth letter from the right. It might be and it might not be—the machine doesn't know, and there is no way for it to tell at that point. If it guesses that it is not, then it goes around the loop again. If it guesses that it is, then it commits to that guess by moving to the second state, an irrevocable decision. Now it must *verify* that its guess was correct; this is the purpose of the tail of the automaton leading to the accept state. If the 1 that it guessed was fifth from the right really is fifth from the right, then the machine will be in its accept state exactly when it comes to the end of the input string, therefore it will accept the string. If not, then maybe the symbol fifth from the right is a 0, and *no* guess would have worked; or maybe the symbol fifth from the right was a 1, but the machine just guessed the wrong 1.

Note, however, that for any string $x \in A$ (that is, for any string with a 1 fifth from the right), *there is* a lucky guess that leads to acceptance; whereas for any string $x \notin A$ (that is, for any string with a 0 fifth from the right), *no* guess can possibly lead to acceptance, no matter how lucky the automaton is.

In general, to show that a nondeterministic machine accepts a set B, we must argue that for any string $x \in B$, there is a lucky sequence of guesses that leads from a start state to an accept state when the end of x is reached;

but for any string $x \notin B$, *no* sequence of guesses leads to an accept state when the end of x is reached, no matter how lucky the automaton is.

Keep in mind that this process of *guessing and verifying* is just an intuitive aid. The formal definition of nondeterministic acceptance will be given in Lecture 6.

There does exist a deterministic automaton accepting the set A, but any such automaton must have at least $2^5 = 32$ states, since a deterministic machine essentially has to remember the last five symbols seen.

The Subset Construction

We will prove a rather remarkable fact: in terms of the sets accepted, nondeterministic finite automata are no more powerful than deterministic ones. In other words, for every nondeterministic finite automaton, there is a deterministic one accepting the same set. The deterministic automaton, however, may require more states.

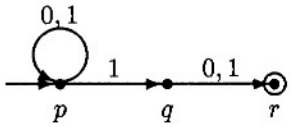
This theorem can be proved using the *subset construction.* Here is the intuitive idea; we will give a formal treatment in Lecture 6. Given a nondeterministic machine N, think of putting pebbles on the states to keep track of all the states N could possibly be in after scanning a prefix of the input. We start with pebbles on all the start states of the nondeterministic machine. Say after scanning some prefix y of the input string, we have pebbles on some set P of states, and say P is the set of all states N could possibly be in after scanning y, depending on the nondeterministic choices that N could have made so far. If input symbol b comes in, pick the pebbles up off the states of P and put a pebble down on each state reachable from a state in P under input symbol b. Let P' be the new set of states covered by pebbles. Then P' is the set of states that N could possibly be in after scanning yb.

Although for a state q of N, there may be many possible next states after scanning b, note that the set P' is uniquely determined by b and the set P. We will thus build a deterministic automaton M *whose states are these sets.* That is, a state of M will be a *set* of states of N. The start state of M will be the *set* of start states of N, indicating that we start with one pebble on each of the start states of N. A final state of M will be any set P containing a final state of N, since we want to accept x if it is possible for N to have made choices while scanning x that lead to an accept state of N.

It takes a stretch of the imagination to regard a set of states of N as a single state of M. Let's illustrate the construction with a shortened version of the example above.

Example 5.1 Consider the set

$$A = \{x \in \{0,1\}^* \mid \text{the second symbol from the right is } 1\}.$$



Label the states p, q, r from left to right, as illustrated. The states of M will be *subsets* of the set of states of N. In this example there are eight such subsets:

$$\varnothing,\ \{p\},\ \{q\},\ \{r\},\ \{p,q\},\ \{p,r\},\ \{q,r\},\ \{p,q,r\}.$$

Here is the deterministic automaton M:

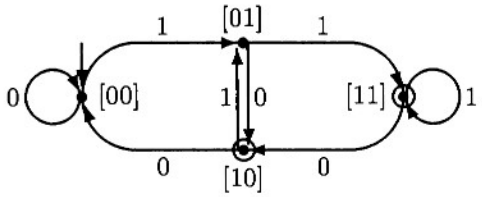
	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\rightarrow \{p\}$	$\{p\}$	$\{p,q\}$
$\{q\}$	$\{r\}$	$\{r\}$
$\{r\}F$	$\varnothing$	$\varnothing$
$\{p,q\}$	$\{p,r\}$	$\{p,q,r\}$
$\{p,r\}F$	$\{p\}$	$\{p,q\}$
$\{q,r\}F$	$\{r\}$	$\{r\}$
$\{p,q,r\}F$	$\{p,r\}$	$\{p,q,r\}$

For example, if we have pebbles on p and q (the fifth row of the table), and if we see input symbol 0 (first column), then in the next step there will be pebbles on p and r. This is because in the automaton N, p is reachable from p under input 0 and r is reachable from q under input 0, and these are the only states reachable from p and q under input 0. The accept states of M (marked F in the table) are those sets containing an accept state of N. The start state of M is $\{p\}$, the set of all start states of N.

Following 0 and 1 transitions from the start state $\{p\}$ of M, one can see that states $\{q,r\}$, $\{q\}$, $\{r\}$, $\varnothing$ of M can never be reached. These states of M are *inaccessible*, and we might as well throw them out. This leaves

	0	1
$\rightarrow \{p\}$	$\{p\}$	$\{p,q\}$
$\{p,q\}$	$\{p,r\}$	$\{p,q,r\}$
$\{p,r\}F$	$\{p\}$	$\{p,q\}$
$\{p,q,r\}F$	$\{p,r\}$	$\{p,q,r\}$

This four-state automaton is exactly the one you would have come up with if you had built a deterministic automaton directly to remember the last two bits seen and accept if the next-to-last bit is a 1:



Here the state labels $[bc]$ indicate the last two bits seen (for our purposes the null string is as good as having just seen two 0's). Note that these two automata are isomorphic (i.e., they are the same automaton up to the renaming of states):

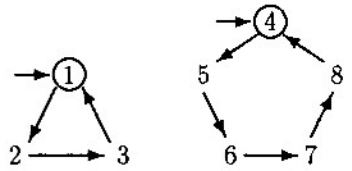
$$\begin{aligned} \{p\} &\approx [00], \\ \{p,q\} &\approx [01], \\ \{p,r\} &\approx [10], \\ \{p,q,r\} &\approx [11]. \end{aligned}$$

□

Example 5.2 Consider the set

$$\{x \in \{a\}^* \mid |x| \text{ is divisible by 3 or 5}\}. \tag{5.1}$$

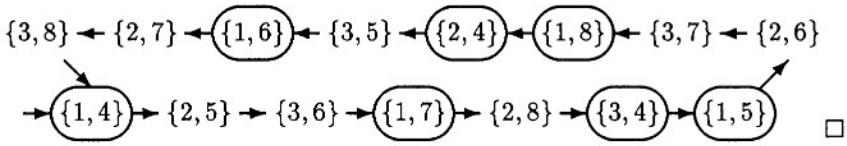
Here is an eight-state nondeterministic automaton N with two start states accepting this set (labels a on transitions are omitted since there is only one input symbol).



The only nondeterminism is in the choice of start state. The machine guesses at the outset whether to check for divisibility by 3 or 5. After that, the computation is deterministic.

Let Q be the states of N. We will build a deterministic machine M whose states are subsets of Q. There are $2^8 = 256$ of these in all, but most will be inaccessible (not reachable from the start state of M under any input). Think about moving pebbles—for this particular automaton, if you start with pebbles on the start states and move pebbles to mark all states the machine could possibly be in, you always have exactly two pebbles on N. This says that only subsets of Q with two elements will be accessible as states of M.

The subset construction gives the following deterministic automaton M with 15 accessible states:



□

In the next lecture we will give a formal definition of nondeterministic finite automata and a general account of the subset construction.

Lecture 6

The Subset Construction

Formal Definition of Nondeterministic Finite Automata

A *nondeterministic finite automaton (NFA)* is a five-tuple

$$N = (Q, \Sigma, \Delta, S, F),$$

where everything is the same as in a deterministic automaton, except for the following two differences.

- S is a *set* of states, that is, $S \subseteq Q$, instead of a single state. The elements of S are called *start states.*
- Δ is a function

  $$\Delta : Q \times \Sigma \to 2^Q,$$

  where 2^Q denotes the *power set* of Q or the set of all subsets of Q:

  $$2^Q \stackrel{\text{def}}{=} \{A \mid A \subseteq Q\}.$$

Intuitively, $\Delta(p, a)$ gives the set of all states that N is allowed to move to from p in one step under input symbol a. We often write

$$p \xrightarrow{a} q$$

if $q \in \Delta(p,a)$. The set $\Delta(p,a)$ can be the empty set $\varnothing$. The function Δ is called the *transition function.*

Now we define acceptance for NFAs. The function Δ extends in a natural way by induction to a function

$$\widehat{\Delta} : 2^Q \times \Sigma^* \to 2^Q$$

according to the rules

$$\widehat{\Delta}(A,\epsilon) \stackrel{\text{def}}{=} A, \tag{6.1}$$

$$\widehat{\Delta}(A,xa) \stackrel{\text{def}}{=} \bigcup_{q \in \widehat{\Delta}(A,x)} \Delta(q,a). \tag{6.2}$$

Intuitively, for $A \subseteq Q$ and $x \in \Sigma^*$, $\widehat{\Delta}(A,x)$ is the set of all states reachable under input string x from *some* state in A. Note that Δ takes a single state as its first argument and a single symbol as its second argument, whereas $\widehat{\Delta}$ takes a *set* of states as its first argument and a *string* of symbols as its second argument.

Equation (6.1) says that the set of all states reachable from a state in A under the null input is just A. In (6.2), the notation on the right-hand side means the union of all the sets $\Delta(q,a)$ for $q \in \widehat{\Delta}(A,x)$; in other words, $r \in \widehat{\Delta}(A,xa)$ if there exists $q \in \widehat{\Delta}(A,x)$ such that $r \in \Delta(q,a)$.

Thus $q \in \widehat{\Delta}(A,x)$ if N can move from some state $p \in A$ to state q under input x. This is the nondeterministic analog of the construction of $\widehat{\delta}$ for deterministic automata we have already seen.

Note that for $a \in \Sigma$,

$$\begin{aligned}\widehat{\Delta}(A,a) &= \bigcup_{p \in \widehat{\Delta}(A,\epsilon)} \Delta(p,a) \\ &= \bigcup_{p \in A} \Delta(p,a).\end{aligned}$$

The automaton N is said to *accept* $x \in \Sigma^*$ if

$$\widehat{\Delta}(S,x) \cap F \neq \varnothing.$$

In other words, N accepts x if there exists an accept state q (i.e., $q \in F$) such that q is reachable from a start state under input string x (i.e., $q \in \widehat{\Delta}(S,x)$).

We define $L(N)$ to be the set of all strings accepted by N:

$$L(N) = \{x \in \Sigma^* \mid N \text{ accepts } x\}.$$

Under this definition, every DFA

$$(Q, \Sigma, \delta, s, F)$$

is equivalent to an NFA

$$(Q, \Sigma, \Delta, \{s\}, F),$$

where $\Delta(p,a) \stackrel{\text{def}}{=} \{\delta(p,a)\}$. Below we will show that the converse holds as well: every NFA is equivalent to some DFA.

Here are some basic lemmas that we will find useful when dealing with NFAs. The first corresponds to Exercise 3 of Homework 1 for deterministic automata.

Lemma 6.1 *For any $x, y \in \Sigma^*$ and $A \subseteq Q$,*

$$\widehat{\Delta}(A, xy) = \widehat{\Delta}(\widehat{\Delta}(A,x), y).$$

Proof. The proof is by induction on $|y|$.

Basis

For $y = \epsilon$,

$$\begin{aligned}\widehat{\Delta}(A, x\epsilon) &= \widehat{\Delta}(A,x) \\ &= \widehat{\Delta}(\widehat{\Delta}(A,x), \epsilon) \quad \text{by (6.1).}\end{aligned}$$

Induction step

For any $y \in \Sigma^*$ and $a \in \Sigma$,

$$\begin{aligned}\widehat{\Delta}(A, xya) &= \bigcup_{q \in \widehat{\Delta}(A,xy)} \Delta(q,a) && \text{by (6.2)} \\ &= \bigcup_{q \in \widehat{\Delta}(\widehat{\Delta}(A,x),y)} \Delta(q,a) && \text{induction hypothesis} \\ &= \widehat{\Delta}(\widehat{\Delta}(A,x), ya) && \text{by (6.2).}\end{aligned}$$

□

Lemma 6.2 *The function $\widehat{\Delta}$ commutes with set union: for any indexed family A_i of subsets of Q and $x \in \Sigma^*$,*

$$\widehat{\Delta}(\bigcup_i A_i, x) = \bigcup_i \widehat{\Delta}(A_i, x).$$

Proof. By induction on $|x|$.

Basis

By (6.1),

$$\widehat{\Delta}(\bigcup_i A_i, \epsilon) = \bigcup_i A_i = \bigcup_i \widehat{\Delta}(A_i, \epsilon).$$

Induction step

$$\begin{aligned}
\widehat{\Delta}(\bigcup_i A_i, xa) &= \bigcup_{p \in \widehat{\Delta}(\bigcup_i A_i, x)} \Delta(p, a) && \text{by (6.2)} \\
&= \bigcup_{p \in \bigcup_i \widehat{\Delta}(A_i, x)} \Delta(p, a) && \text{induction hypothesis} \\
&= \bigcup_i \bigcup_{p \in \widehat{\Delta}(A_i, x)} \Delta(p, a) && \text{basic set theory} \\
&= \bigcup_i \widehat{\Delta}(A_i, xa) && \text{by (6.2).}
\end{aligned}$$

□

In particular, expressing a set as the union of its singleton subsets,

$$\widehat{\Delta}(A, x) = \bigcup_{p \in A} \widehat{\Delta}(\{p\}, x). \tag{6.3}$$

The Subset Construction: General Account

The subset construction works in general. Let

$$N = (Q_N, \Sigma, \Delta_N, S_N, F_N)$$

be an arbitrary NFA. We will use the subset construction to produce an equivalent DFA. Let M be the DFA

$$M = (Q_M, \Sigma, \delta_M, s_M, F_M),$$

where

$$\begin{aligned}
Q_M &\stackrel{\text{def}}{=} 2^{Q_N}, \\
\delta_M(A, a) &\stackrel{\text{def}}{=} \widehat{\Delta}_N(A, a), \\
s_M &\stackrel{\text{def}}{=} S_N, \\
F_M &\stackrel{\text{def}}{=} \{A \subseteq Q_N \mid A \cap F_N \neq \varnothing\}.
\end{aligned}$$

Note that δ_M is a function from states of M and input symbols to states of M, as it should be, because states of M are *sets* of states of N.

Lemma 6.3 *For any $A \subseteq Q_N$ and $x \in \Sigma^*$,*

$$\widehat{\delta}_M(A, x) = \widehat{\Delta}_N(A, x).$$

Proof. Induction on $|x|$.

Basis

For $x = \epsilon$, we want to show

$$\widehat{\delta}_M(A, \epsilon) = \widehat{\Delta}_N(A, \epsilon).$$

But both of these are A, by definition of $\widehat{\delta}_M$ and $\widehat{\Delta}_N$.

Induction step

Assume that

$$\widehat{\delta}_M(A, x) = \widehat{\Delta}_N(A, x).$$

We want to show the same is true for xa, $a \in \Sigma$.

$$\begin{aligned}
\widehat{\delta}_M(A, xa) &= \delta_M(\widehat{\delta}_M(A, x), a) && \text{definition of } \widehat{\delta}_M \\
&= \delta_M(\widehat{\Delta}_N(A, x), a) && \text{induction hypothesis} \\
&= \widehat{\Delta}_N(\widehat{\Delta}_N(A, x), a) && \text{definition of } \delta_M \\
&= \widehat{\Delta}_N(A, xa) && \text{Lemma 6.1.}
\end{aligned}$$

□

Theorem 6.4 *The automata M and N accept the same set.*

Proof. For any $x \in \Sigma^*$,

$$\begin{aligned}
& x \in L(M) \\
\iff & \widehat{\delta}_M(s_M, x) \in F_M && \text{definition of acceptance for } M \\
\iff & \widehat{\Delta}_N(S_N, x) \cap F_N \neq \varnothing && \text{definition of } s_M \text{ and } F_M, \text{ Lemma 6.3} \\
\iff & x \in L(N) && \text{definition of acceptance for } N.
\end{aligned}$$

□

ϵ-Transitions

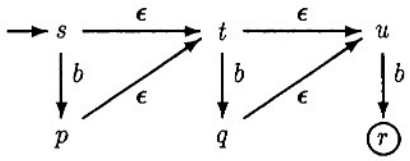
Here is another extension of finite automata that turns out to be quite useful but really adds no more power.

An *ϵ-transition* is a transition with label ϵ, a letter that stands for the null string ϵ:

$$p \xrightarrow{\epsilon} q.$$

The automaton can take such a transition anytime without reading an input symbol.

Example 6.5

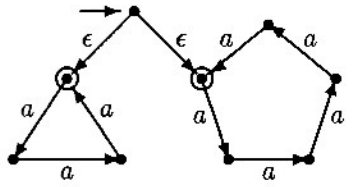


If the machine is in state s and the next input symbol is b, it can nondeterministically decide to do one of three things:

- read the b and move to state p;
- slide to t without reading an input symbol, then read the b and move to state q; or
- slide to t without reading an input symbol, then slide to u without reading an input symbol, then read the b and move to state r.

The set of strings accepted by this automaton is $\{b, bb, bbb\}$. □

Example 6.6 Here is a nondeterministic automaton with ϵ-transitions accepting the set $\{x \in \{a\}^* \mid |x| \text{ is divisible by 3 or 5}\}$:



The automaton chooses at the outset which of the two conditions to check for (divisibility by 3 or 5) and slides to one of the two loops accordingly without reading an input symbol. □

The main benefit of ϵ-transitions is convenience. They do not really add any power: a modified subset construction involving the notion of *ϵ-closure* can be used to show that every NFA with ϵ-transitions can be simulated by a DFA without ϵ-transitions (Miscellaneous Exercise 10); thus all sets accepted by nondeterministic automata with ϵ-transitions are regular. We will also give an alternative treatment in Lecture 10 using homomorphisms.

More Closure Properties

Recall that the concatenation of sets A and B is the set

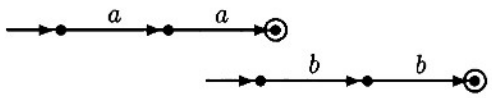
$$AB = \{xy \mid x \in A \text{ and } y \in B\}.$$

For example,

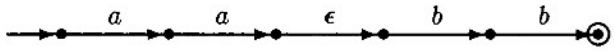
$$\{a, ab\}\{b, ba\} = \{ab, aba, abb, abba\}.$$

If A and B are regular, then so is AB. To see this, let M be an automaton for A and N an automaton for B. Make a new automaton P whose states are the union of the state sets of M and N, and take all the transitions of M and N as transitions of P. Make the start states of M the start states of P and the final states of N the final states of P. Finally, put ϵ-transitions from all the final states of M to all the start states of N. Then $L(P) = AB$.

Example 6.7 Let $A = \{aa\}$, $B = \{bb\}$. Here are automata for A and B:



Here is the automaton you get by the construction above for AB:



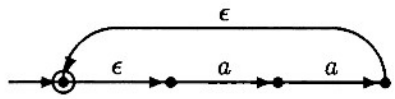
□

If A is regular, then so is its asterate:

$$\begin{aligned} A^* &= \{\epsilon\} \cup A \cup A^2 \cup A^3 \cup \cdots \\ &= \{x_1 x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in A,\ 1 \leq i \leq n\}. \end{aligned}$$

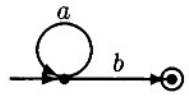
To see this, take an automaton M for A. Build an automaton P for A^* as follows. Start with all the states and transitions of M. Add a new state s. Add ϵ-transitions from s to all the start states of M and from all the final states of M to s. Make s the only start state of P and also the only final state of P (thus the start and final states of M are *not* start and final states of P). Then P accepts exactly the set A^*.

Example 6.8 Let $A = \{aa\}$. Consider the three-state automaton for A in Example 6.7. Here is the automaton you get for A^* by the construction above:



□

In this construction, you must add the new start/final state s. You might think that it suffices to put in ϵ-transitions from the old final states back to the old start states and make the old start states final states, but this doesn't always work. Here's a counterexample:



The set accepted is $\{a^n b \mid n \geq 0\}$. The asterate of this set is

$$\{\epsilon\} \cup \{\text{strings ending with } b\},$$

but if you put in an ϵ-transition from the final state back to the start state and made the start state a final state, then the set accepted would be $\{a, b\}^*$.

Historical Notes

Rabin and Scott [102] introduced nondeterministic finite automata and showed using the subset construction that they were no more powerful than deterministic finite automata.

Closure properties of regular sets were studied by Ginsburg and Rose [46, 48], Ginsburg [43], McNaughton and Yamada [85], and Rabin and Scott [102], among others.

Lecture 7

Pattern Matching

What happens when one types `rm *` in UNIX? (If you don't know, don't try it to find out!) What if the current directory contains the files

```
a.tex  bc.tex  a.dvi  bc.dvi
```

and one types `rm *.dvi`? What would happen if there were a file named `.dvi`?

What is going on here is *pattern matching*. The * in UNIX is a pattern that matches any string of symbols, including the null string.

Pattern matching is an important application of finite automata. The UNIX commands **grep**, **fgrep**, and **egrep** are basic pattern-matching utilities that use finite automata in their implementation.

Let Σ be a finite alphabet. A *pattern* is a string of symbols of a certain form representing a (possibly infinite) set of strings in Σ^*. The set of patterns is defined formally by induction below. They are either *atomic patterns* or *compound patterns* built up inductively from atomic patterns using certain *operators*. We'll denote patterns by Greek letters α, β, γ, ...

As we define patterns, we will tell which strings $x \in \Sigma^*$ *match* them. The set of strings in Σ^* matching a given pattern α will be denoted $L(\alpha)$. Thus

$$L(\alpha) = \{x \in \Sigma^* \mid x \text{ matches } \alpha\}.$$

In the following, forget the UNIX definition of *. We will use the symbol * for something else.

The *atomic patterns* are

- a for each $a \in \Sigma$, matched by the symbol a only; in symbols, $L(a) = \{a\}$;
- ϵ, matched only by ϵ, the null string; in symbols, $L(\epsilon) = \{\epsilon\}$;
- $\emptyset$, matched by nothing; in symbols, $L(\emptyset) = \varnothing$, the empty set;
- #, matched by any symbol in Σ; in symbols, $L(\#) = \Sigma$;
- @, matched by any string in Σ^*; in symbols, $L(@) = \Sigma^*$.

Compound patterns are formed inductively using binary operators $+$, $\cap$, and $\cdot$ (usually not written) and unary operators $^+$, *, and $\sim$. If α and β are patterns, then so are $\alpha + \beta$, $\alpha \cap \beta$, α^*, α^+, $\sim\alpha$, and $\alpha\beta$. The last of these is short for $\alpha \cdot \beta$.

We also define inductively which strings match each pattern. We have already said which strings match the atomic patterns. This is the basis of the inductive definition. Now suppose we have already defined the sets of strings $L(\alpha)$ and $L(\beta)$ matching α and β, respectively. Then we'll say that

- x matches $\alpha + \beta$ if x matches either α or β:

$$L(\alpha + \beta) = L(\alpha) \cup L(\beta);$$

- x matches $\alpha \cap \beta$ if x matches both α and β:

$$L(\alpha \cap \beta) = L(\alpha) \cap L(\beta);$$

- x matches $\alpha\beta$ if x can be broken down as $x = yz$ such that y matches α and z matches β:

$$\begin{aligned} L(\alpha\beta) &= L(\alpha)L(\beta) \\ &= \{yz \mid y \in L(\alpha) \text{ and } z \in L(\beta)\}; \end{aligned}$$

- x matches $\sim\alpha$ if x does not match α:

$$\begin{aligned} L(\sim\alpha) &= \sim L(\alpha) \\ &= \Sigma^* - L(\alpha); \end{aligned}$$

- x matches α^* if x can be expressed as a concatenation of zero or more strings, all of which match α:

$$\begin{aligned} L(\alpha^*) &= \{x_1 x_2 \cdots x_n \mid n \geq 0 \text{ and } x_i \in L(\alpha),\ 1 \leq i \leq n\} \\ &= L(\alpha)^0 \cup L(\alpha)^1 \cup L(\alpha)^2 \cup \cdots \end{aligned}$$

$$= L(\alpha)^*.$$

The null string ϵ always matches α^*, since ϵ is a concatenation of zero strings, all of which (vacuously) match α.

- x matches α^+ if x can be expressed as a concatenation of one or more strings, all of which match α:

$$\begin{aligned} L(\alpha^+) &= \{x_1 x_2 \cdots x_n \mid n \geq 1 \text{ and } x_i \in L(\alpha),\ 1 \leq i \leq n\} \\ &= L(\alpha)^1 \cup L(\alpha)^2 \cup L(\alpha)^3 \cup \cdots \\ &= L(\alpha)^+. \end{aligned}$$

Note that patterns are just certain strings of symbols over the alphabet

$$\Sigma \cup \{\, \epsilon,\ \emptyset,\ \#,\ @,\ +,\ \cap,\ \sim,\ ^*,\ ^+,\ (,\ )\,\}.$$

Note also that the meanings of $\#$, $@$, and $\sim$ depend on Σ. For example, if $\Sigma = \{a, b, c\}$ then $L(\#) = \{a, b, c\}$, but if $\Sigma = \{a\}$ then $L(\#) = \{a\}$.

Example 7.1

- $\Sigma^* = L(@) = L(\#^*)$.
- Singleton sets: if $x \in \Sigma^*$, then x itself is a pattern and is matched only by the string x; i.e., $\{x\} = L(x)$.
- Finite sets: if $x_1, \ldots, x_m \in \Sigma^*$, then

$$\{x_1, x_2, \ldots, x_m\} = L(x_1 + x_2 + \cdots + x_m). \qquad \square$$

Note that we can write the last pattern $x_1 + x_2 + \cdots + x_m$ without parentheses, since the two patterns $(\alpha + \beta) + \gamma$ and $\alpha + (\beta + \gamma)$ are matched by the same set of strings; i.e.,

$$L((\alpha + \beta) + \gamma) = L(\alpha + (\beta + \gamma)).$$

Mathematically speaking, the operator $+$ is *associative*. The concatenation operator $\cdot$ is associative, too. Hence we can also unambiguously write $\alpha\beta\gamma$ without parentheses.

Example 7.2

- strings containing at least three occurrences of a:

$$@a@a@a@;$$

- strings containing an a followed later by a b; that is, strings of the form $xaybz$ for some x, y, z:

$$@a@b@;$$

- all single letters except a:

$$\# \cap \sim a;$$

- strings with no occurrence of the letter a:

  $$(\# \cap \sim a)^*;$$

- strings in which every occurrence of a is followed sometime later by an occurrence of b; in other words, strings in which there are either no occurrences of a, or there is an occurrence of b followed by no occurrence of a; for example, aab matches but bba doesn't:

  $$(\# \cap \sim a)^* + @b(\# \cap \sim a)^*.$$

  If the alphabet is $\{a, b\}$, then this takes a much simpler form:

  $$\epsilon + @b.$$ □

Before we go too much further, there is a subtlety that needs to be mentioned. Note the slight difference in appearance between ϵ and ϵ and between $\emptyset$ and $\varnothing$. The objects ϵ and $\emptyset$ are *symbols* in the language of patterns, whereas ϵ and $\varnothing$ are *metasymbols* that we are using to name the null string and the empty set, respectively. These are different sorts of things: ϵ and $\emptyset$ are symbols, that is, strings of length one, whereas ϵ is a string of length zero and $\varnothing$ isn't even a string.

We'll maintain the distinction for a few lectures until we get used to the idea, but at some point in the near future we'll drop the boldface and use ϵ and $\varnothing$ exclusively. We'll always be able to infer from context whether we mean the symbols or the metasymbols. This is a little more convenient and conforms to standard usage, but bear in mind that they are still different things.

While we're on the subject of abuse of notation, we should also mention that very often you will see things like $x \in a^*b^*$ in texts and articles. Strictly speaking, one should write $x \in L(a^*b^*)$, since a^*b^* is a pattern, not a set of strings. But as long as you know what you really mean and can stand the guilt, it is okay to write $x \in a^*b^*$.

Lecture 8

Pattern Matching and Regular Expressions

Here are some interesting and important questions:

- How hard is it to determine whether a given string x matches a given pattern α? This is an important practical question. There are very efficient algorithms, as we will see.
- Is every set represented by some pattern? Answer: no. For example, the set

  $$\{a^n b^n \mid n \geq 0\}$$

  is not represented by any pattern. We'll prove this later.
- Patterns α and β are *equivalent* if $L(\alpha) = L(\beta)$. How do you tell whether α and β are equivalent? Sometimes it is obvious and sometimes not.
- Which operators are redundant? For example, we can get rid of ϵ since it is equivalent to $\sim(\#@)$ and also to $\emptyset^*$. We can get rid of @ since it is equivalent to $\#^*$. We can get rid of unary $^+$ since α^+ is equivalent to $\alpha\alpha^*$. We can get rid of #, since if $\Sigma = \{a_1, \ldots, a_n\}$ then # is equivalent to the pattern

  $$a_1 + a_2 + \cdots + a_n.$$

The operator $\cap$ is also redundant, by one of the De Morgan laws:

$\alpha \cap \beta$ is equivalent to $\sim(\sim\alpha + \sim\beta)$.

Redundancy is an important question. From a user's point of view, we would like to have a lot of operators since this lets us write more succinct patterns; but from a programmer's point of view, we would like to have as few as possible since there is less code to write. Also, from a theoretical point of view, fewer operators mean fewer cases we have to treat in giving formal semantics and proofs of correctness.

An amazing and difficult-to-prove fact is that the operator $\sim$ is redundant. Thus every pattern is equivalent to one using only atomic patterns $a \in \Sigma$, ϵ, $\emptyset$, and operators $+$, $\cdot$, and *. Patterns using only these symbols are called *regular expressions*. Actually, as we have observed, even ϵ is redundant, but we include it in the definition of regular expressions because it occurs so often.

Our goal for this lecture and the next will be to show that the family of subsets of Σ^* represented by patterns is exactly the family of regular sets. Thus as a way of describing subsets of Σ^*, finite automata, patterns, and regular expressions are equally expressive.

Some Notational Conveniences

Since the binary operators $+$ and $\cdot$ are associative, that is,

$$L(\alpha + (\beta + \gamma)) = L((\alpha + \beta) + \gamma),$$
$$L(\alpha(\beta\gamma)) = L((\alpha\beta)\gamma),$$

we can write

$$\alpha + \beta + \gamma \quad \text{and} \quad \alpha\beta\gamma$$

without ambiguity. To resolve ambiguity in other situations, we assign precedence to operators. For example,

$$\alpha + \beta\gamma$$

could be interpreted as either

$$\alpha + (\beta\gamma) \quad \text{or} \quad (\alpha + \beta)\gamma,$$

which are not equivalent. We adopt the convention that the concatenation operator $\cdot$ has higher precedence than $+$, so that we would prefer the former interpretation. Similarly, we assign * higher precedence than $+$ or $\cdot$, so that

$$\alpha + \beta^*$$

is interpreted as

$$\alpha + (\beta^*)$$

and not as

$$(\alpha + \beta)^*.$$

All else failing, use parentheses.

Equivalence of Patterns, Regular Expressions, and Finite Automata

Patterns, regular expressions (patterns built from atomic patterns $a \in \Sigma$, ϵ, $\emptyset$, and operators $+$, *, and $\cdot$ only), and finite automata are all equivalent in expressive power: they all represent the regular sets.

Theorem 8.1 *Let $A \subseteq \Sigma^*$. The following three statements are equivalent:*

(i) A is regular; that is, $A = L(M)$ for some finite automaton M;

(ii) $A = L(\alpha)$ for some pattern α;

(iii) $A = L(\alpha)$ for some regular expression α.

Proof. The implication (iii) $\Rightarrow$ (ii) is trivial, since every regular expression is a pattern. We prove (ii) $\Rightarrow$ (i) here and (i) $\Rightarrow$ (iii) in Lecture 9.

The heart of the proof (ii) $\Rightarrow$ (i) involves showing that certain basic sets (corresponding to atomic patterns) are regular, and the regular sets are closed under certain closure operations corresponding to the operators used to build patterns. Note that

- the singleton set $\{a\}$ is regular, $a \in \Sigma$,
- the singleton set $\{\epsilon\}$ is regular, and
- the empty set $\varnothing$ is regular,

since each of these sets is the set accepted by some automaton. Here are nondeterministic automata for these three sets, respectively:

Also, we have previously shown that the regular sets are closed under the set operations $\cup$, $\cap$, $\sim$, $\cdot$, *, and $^+$; that is, if A and B are regular sets, then so are $A \cup B$, $A \cap B$, $\sim A = \Sigma^* - A$, AB, A^*, and A^+.

These facts can be used to prove inductively that (ii) $\Rightarrow$ (i). Let α be a given pattern. We wish to show that $L(\alpha)$ is a regular set. We proceed by

induction on the structure of α. The pattern α is of one of the following forms:

(i)	a, where $a \in \Sigma$;	(vi)	$\beta + \gamma$;
(ii)	ϵ;	(vii)	$\beta \cap \gamma$;
(iii)	$\emptyset$;	(viii)	$\beta\gamma$;
(iv)	$\#$;	(ix)	$\sim\beta$;
(v)	@;	(x)	β^*;
		(xi)	β^+.

There are five base cases (i) through (v) corresponding to the atomic patterns and six induction cases (vi) through (xi) corresponding to compound patterns. Each of these cases uses a closure property of the regular sets previously observed.

For (i), (ii), and (iii), we have $L(a) = \{a\}$ for $a \in \Sigma$, $L(\epsilon) = \{\epsilon\}$, and $L(\emptyset) = \varnothing$, and these are regular sets.

For (iv), (v), and (xi), we observed earlier that the operators $\#$, @, and $^+$ were redundant, so we may disregard these cases since they are already covered by the other cases.

For (vi), recall that $L(\beta+\gamma) = L(\beta) \cup L(\gamma)$ by definition of the $+$ operator. By the induction hypothesis, $L(\beta)$ and $L(\gamma)$ are regular. Since the regular sets are closed under union, $L(\beta + \gamma) = L(\beta) \cup L(\gamma)$ is also regular.

The arguments for the remaining cases (vii) through (x) are similar to the argument for (vi). Each of these cases uses a closure property of the regular sets that we have observed previously in Lectures 4 and 6. □

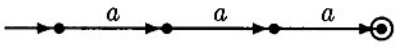
Example 8.2 Let's convert the regular expression

$$(aaa)^* + (aaaaa)^*$$

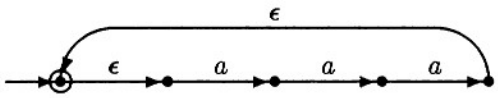
for the set

$$\{x \in \{a\}^* \mid |x| \text{ is divisible by either 3 or 5}\}$$

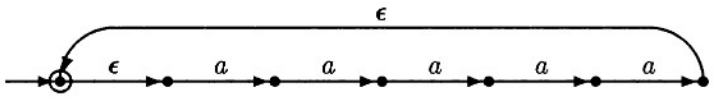
to an equivalent NFA. First we show how to construct an automaton for $(aaa)^*$. We take an automaton accepting only the string aaa, say



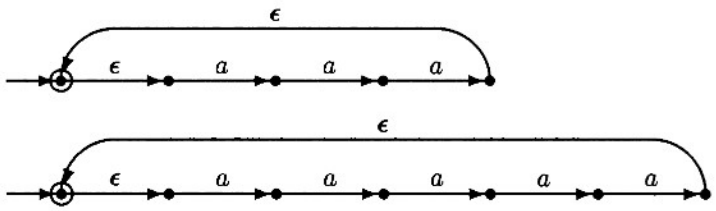
Applying the construction of Lecture 6, we add a new start state and ϵ-transitions from the new start state to all the old start states and from all the old accept states to the new start state. We let the new start state be the only accept state of the new automaton. This gives



The construction for $(aaaaa)^*$ is similar, giving



To get an NFA for $(aaa)^* + (aaaaa)^*$, we can simply take the disjoint union of these two automata:



□

Lecture 9

Regular Expressions and Finite Automata

Simplification of Expressions

For small regular expressions, one can often see how to construct an equivalent automaton directly without going through the mechanical procedure of the previous lecture. It is therefore useful to try to simplify the expression first.

For regular expressions α, β, if $L(\alpha) = L(\beta)$, we write $\alpha \equiv \beta$ and say that α and β are *equivalent*. The relation $\equiv$ on regular expressions is an equivalence relation; that is, it is

- reflexive: $\alpha \equiv \alpha$ for all α;
- symmetric: if $\alpha \equiv \beta$, then $\beta \equiv \alpha$; and
- transitive: if $\alpha \equiv \beta$ and $\beta \equiv \gamma$, then $\alpha \equiv \gamma$.

If $\alpha \equiv \beta$, one can substitute α for β (or vice versa) in any regular expression, and the resulting expression will be equivalent to the original.

Here are a few laws that can be used to simplify regular expressions.

$$\alpha + (\beta + \gamma) \equiv (\alpha + \beta) + \gamma \tag{9.1}$$

$$\alpha + \beta \equiv \beta + \alpha \tag{9.2}$$

$$\alpha + \emptyset \equiv \alpha \tag{9.3}$$
$$\alpha + \alpha \equiv \alpha \tag{9.4}$$
$$\alpha(\beta\gamma) \equiv (\alpha\beta)\gamma \tag{9.5}$$
$$\epsilon\alpha \equiv \alpha\epsilon \equiv \alpha \tag{9.6}$$
$$\alpha(\beta + \gamma) \equiv \alpha\beta + \alpha\gamma \tag{9.7}$$
$$(\alpha + \beta)\gamma \equiv \alpha\gamma + \beta\gamma \tag{9.8}$$
$$\emptyset\alpha \equiv \alpha\emptyset \equiv \emptyset \tag{9.9}$$
$$\epsilon + \alpha\alpha^* \equiv \alpha^* \tag{9.10}$$
$$\epsilon + \alpha^*\alpha \equiv \alpha^* \tag{9.11}$$
$$\beta + \alpha\gamma \leq \gamma \Rightarrow \alpha^*\beta \leq \gamma \tag{9.12}$$
$$\beta + \gamma\alpha \leq \gamma \Rightarrow \beta\alpha^* \leq \gamma \tag{9.13}$$

In (9.12) and (9.13), $\leq$ refers to the subset order:

$$\begin{aligned} \alpha \leq \beta &\stackrel{\text{def}}{\iff} L(\alpha) \subseteq L(\beta) \\ &\iff L(\alpha + \beta) = L(\beta) \\ &\iff \alpha + \beta \equiv \beta. \end{aligned}$$

Laws (9.12) and (9.13) are not equations but rules from which one can derive equations from other equations. Laws (9.1) through (9.13) can be justified by replacing each expression by its definition and reasoning set theoretically.

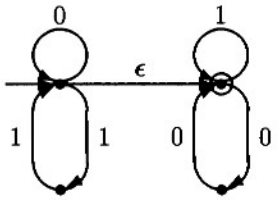
Here are some useful equations that follow from (9.1) through (9.13) that you can use to simplify expressions.

$$(\alpha\beta)^*\alpha \equiv \alpha(\beta\alpha)^* \tag{9.14}$$
$$(\alpha^*\beta)^*\alpha^* \equiv (\alpha + \beta)^* \tag{9.15}$$
$$\alpha^*(\beta\alpha^*)^* \equiv (\alpha + \beta)^* \tag{9.16}$$
$$(\epsilon + \alpha)^* \equiv \alpha^* \tag{9.17}$$
$$\alpha\alpha^* \equiv \alpha^*\alpha \tag{9.18}$$

An interesting fact that is beyond the scope of this course is that all true equations between regular expressions can be proved purely algebraically from the axioms and rules (9.1) through (9.13) plus the laws of equational logic [73].

To illustrate, let's convert some regular expressions to finite automata.

Example 9.1 $(11+0)^*(00+1)^*$



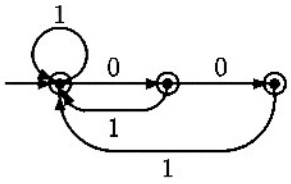
This expression is simple enough that the easiest thing to do is eyeball it. The mechanical method described in Lecture 8 would give more states and ϵ-transitions than shown here. The two states connected by an ϵ-transition cannot be collapsed into one state, since then 10 would be accepted, which does not match the regular expression. □

Example 9.2 $(1+01+001)^*(\epsilon+0+00)$

Using the algebraic laws above, we can rewrite the expression:

$$\begin{aligned}(1+01+001)^*(\epsilon+0+00) &\equiv ((\epsilon+0+00)1)^*(\epsilon+0+00)\\ &\equiv ((\epsilon+0)(\epsilon+0)1)^*(\epsilon+0)(\epsilon+0).\end{aligned}$$

It is now easier to see that the set represented is the set of all strings over $\{0,1\}$ with no substring of more than two adjacent 0's.



□

Just because all states of an NFA are accept states doesn't mean that all strings are accepted! Note that in Example 9.2, 000 is not accepted.

Converting Automata to Regular Expressions

To finish the proof of Theorem 8.1, it remains to show how to convert a given finite automaton M to an equivalent regular expression.

Given an NFA

$$M = (Q, \Sigma, \Delta, S, F),$$

a subset $X \subseteq Q$, and states $u, v \in Q$, we show how to construct a regular expression

$$\alpha_{uv}^{X}$$

representing the set of all strings x such that there is a path from u to v in M labeled x (i.e., such that $v \in \widehat{\Delta}(\{u\}, x)$) and all states along that path, with the possible exception of u and v, lie in X.

The expressions are constructed inductively on the size of X. For the basis $X = \varnothing$, let $a_1, \ldots, a_k$ be all the symbols in Σ such that $v \in \Delta(u, a_i)$. For $u \neq v$, take

$$\alpha_{uv}^{\varnothing} \stackrel{\text{def}}{=} \begin{cases} a_1 + \cdots + a_k & \text{if } k \geq 1, \\ \emptyset & \text{if } k = 0; \end{cases}$$

and for $u = v$, take

$$\alpha_{uv}^{\varnothing} \stackrel{\text{def}}{=} \begin{cases} a_1 + \cdots + a_k + \epsilon & \text{if } k \geq 1, \\ \epsilon & \text{if } k = 0. \end{cases}$$

For nonempty X, we can choose any element $q \in X$ and take

$$\alpha_{uv}^{X} \stackrel{\text{def}}{=} \alpha_{uv}^{X-\{q\}} + \alpha_{uq}^{X-\{q\}} (\alpha_{qq}^{X-\{q\}})^* \alpha_{qv}^{X-\{q\}}. \tag{9.19}$$

To justify the definition (9.19), note that any path from u to v with all intermediate states in X either (i) never visits q, hence the expression

$$\alpha_{uv}^{X-\{q\}}$$

on the right-hand side of (9.19); or (ii) visits q for the first time, hence the expression

$$\alpha_{uq}^{X-\{q\}},$$

followed by a finite number (possibly zero) of loops from q back to itself without visiting q in between and staying in X, hence the expression

$$(\alpha_{qq}^{X-\{q\}})^*,$$

followed by a path from q to v after leaving q for the last time, hence the expression

$$\alpha_{qv}^{X-\{q\}}.$$

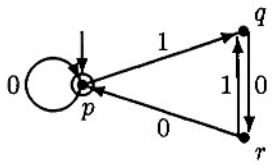
The sum of all expressions of the form

$$\alpha_{sf}^{Q},$$

where s is a start state and f is a final state, represents the set of strings accepted by M.

As a practical rule of thumb when doing homework exercises, when choosing the $q \in X$ to drop out in (9.19), it is best to try to choose one that disconnects the automaton as much as possible.

Example 9.3 Let's convert the automaton



to an equivalent regular expression. The set accepted by this automaton will be represented by the inductively defined regular expression

$$\alpha_{pp}^{\{p,q,r\}},$$

since p is the only start and the only accept state. Removing the state q (we can choose any state we like here), we can take

$$\alpha_{pp}^{\{p,q,r\}} = \alpha_{pp}^{\{p,r\}} + \alpha_{pq}^{\{p,r\}}(\alpha_{qq}^{\{p,r\}})^*\alpha_{qp}^{\{p,r\}}.$$

Looking at the automaton, the only paths going from p to p and staying in the states $\{p,r\}$ are paths going around the single loop labeled 0 from p to p some finite number of times; thus we can take

$$\alpha_{pp}^{\{p,r\}} = 0^*.$$

By similar informal reasoning, we can take

$$\begin{aligned}
\alpha_{pq}^{\{p,r\}} &= 0^*1, \\
\alpha_{qq}^{\{p,r\}} &= \epsilon + 01 + 000^*1 \\
&\equiv \epsilon + 0(\epsilon + 00^*)1 \\
&\equiv \epsilon + 00^*1, \\
\alpha_{qp}^{\{p,r\}} &= 000^*.
\end{aligned}$$

Thus we can take

$$\alpha_{pp}^{\{p,q,r\}} = 0^* + 0^*1(\epsilon + 00^*1)^*000^*.$$

This is matched by the set of all strings accepted by the automaton. We can further simplify the expression using the algebraic laws (9.1) through (9.18):

$$\begin{aligned}
&0^* + 0^*1(\epsilon + 00^*1)^*000^* \\
&\equiv 0^* + 0^*1(00^*1)^*000^* && \text{by (9.17)} \\
&\equiv \epsilon + 00^* + 0^*10(0^*10)^*00^* && \text{by (9.10) and (9.14)} \\
&\equiv \epsilon + (\epsilon + 0^*10(0^*10)^*)00^* && \text{by (9.8)} \\
&\equiv \epsilon + (0^*10)^*00^* && \text{by (9.10)} \\
&\equiv \epsilon + (0^*10)^*0^*0 && \text{by (9.18)} \\
&\equiv \epsilon + (0 + 10)^*0 && \text{by (9.15).}
\end{aligned}$$

□

Historical Notes

Kleene [70] proved that deterministic finite automata and regular expressions are equivalent. A shorter proof was given by McNaughton and Yamada [85].

The relationship between right- and left-linear grammars and regular sets (Homework 5, Exercise 1) was observed by Chomsky and Miller [21].

Supplementary Lecture A

Kleene Algebra and Regular Expressions

In Lecture 9, we gave a combinatorial proof that every finite automaton has an equivalent regular expression. Here is an algebraic proof that generalizes that argument. It is worth looking at because it introduces the notion of *Kleene algebra* and the use of matrices. We will show how to use matrices and Kleene algebra to solve systems of linear equations involving sets of strings.

Kleene algebra is named after Stephen C. Kleene, who invented the regular sets [70].

Kleene Algebra

We have already observed in Lecture 9 that the set operations $\cup$, $\cdot$, and * on subsets of Σ^*, along with the distinguished subsets $\varnothing$ and $\{\epsilon\}$, satisfy certain important algebraic properties. These were listed in Lecture 9, axioms (9.1) through (9.13). Let us call any algebraic structure satisfying these properties a *Kleene algebra*. In general, a Kleene algebra $\mathcal{K}$ consists of a nonempty set with two distinguished constants 0 and 1, two binary operations $+$ and $\cdot$ (usually omitted in expressions), and a unary operation * satisfying the following axioms.

$$a + (b + c) = (a + b) + c \qquad \text{associativity of } + \tag{A.1}$$

$$a + b = b + a \qquad \text{commutativity of } + \tag{A.2}$$

$a + a = a$	idempotence of $+$	(A.3)
$a + 0 = a$	0 is an identity for $+$	(A.4)
$a(bc) = (ab)c$	associativity of $\cdot$	(A.5)
$a1 = 1a = a$	1 is an identity for $\cdot$	(A.6)
$a0 = 0a = 0$	0 is an annihilator for $\cdot$	(A.7)
$a(b + c) = ab + ac$	distributivity	(A.8)
$(a + b)c = ac + bc$	distributivity	(A.9)
$1 + aa^* = a^*$		(A.10)
$1 + a^*a = a^*$		(A.11)
$b + ac \leq c \Rightarrow a^*b \leq c$		(A.12)
$b + ca \leq c \Rightarrow ba^* \leq c$		(A.13)

In (A.12) and (A.13), $\leq$ refers to the naturally defined order

$$a \leq b \stackrel{\text{def}}{\Longleftrightarrow} a + b = b.$$

In 2^{Σ^*}, $\leq$ is just set inclusion $\subseteq$.

Axioms (A.1) through (A.9) discuss the properties of addition and multiplication in a Kleene algebra. These properties are the same as those of ordinary addition and multiplication, with the addition of the idempotence axiom (A.3). These axioms can be summed up briefly by saying that $\mathcal{K}$ is an *idempotent semiring*. The remaining axioms (A.10) through (A.13) discuss the properties of the operator *. They say essentially that * behaves like the asterate operator on sets of strings or the reflexive transitive closure operator on binary relations.

It follows quite easily from the axioms that $\leq$ is a partial order; that is, it is reflexive ($a \leq a$), transitive ($a \leq b$ and $b \leq c$ imply $a \leq c$), and antisymmetric ($a \leq b$ and $b \leq a$ imply $a = b$). Moreover, $a + b$ is the least upper bound of a and b with respect to $\leq$. All the operators are monotone with respect to $\leq$; in other words, if $a \leq b$, then $ac \leq bc$, $ca \leq cb$, $a + c \leq b + c$, and $a^* \leq b^*$.

By (A.10) and distributivity, we have

$$b + aa^*b \leq a^*b,$$

which says that a^*b satisfies the inequality $b + ac \leq c$ when substituted for c. The implication (A.12) says that a^*b is the $\leq$-least element of $\mathcal{K}$ for which this is true. It follows that

Lemma A.1 *In any Kleene algebra, a^*b is the $\leq$-least solution of the equation $x = ax + b$.*

Proof. Miscellaneous Exercise 21. □

Instead of (A.12) and (A.13), we might take the equivalent axioms

$$ac \leq c \Rightarrow a^*c \leq c, \tag{A.14}$$

$$ca \leq c \Rightarrow ca^* \leq c \tag{A.15}$$

(see Miscellaneous Exercise 22).

Here are some typical theorems of Kleene algebra. These can be derived by purely equational reasoning from the axioms above (Miscellaneous Exercise 20).

$$a^*a^* = a^*$$

$$a^{**} = a^*$$

$$(a^*b)^*a^* = (a+b)^* \qquad \text{denesting rule} \tag{A.16}$$

$$a(ba)^* = (ab)^*a \qquad \text{shifting rule} \tag{A.17}$$

$$a^* = (aa)^* + a(aa)^*$$

Equations (A.16) and (A.17), the *denesting rule* and the *shifting rule*, respectively, turn out to be particularly useful in simplifying regular expressions.

The family 2^{Σ^*} of all subsets of Σ^* with constants $\varnothing$ and $\{\epsilon\}$ and operations $\cup, \cdot$, and * forms a Kleene algebra, as does the family of all regular subsets of Σ^* with the same operations. As mentioned in Lecture 9, it can be shown that an equation $\alpha = \beta$ is a theorem of Kleene algebra, that is, is derivable from axioms (A.1) through (A.13), if and only if α and β are equivalent as regular expressions [73].

Another example of a Kleene algebra is the family of all binary relations on a set X with the empty relation for 0, the identity relation

$$\iota \stackrel{\text{def}}{=} \{(u,u) \mid u \in X\}$$

for 1, $\cup$ for $+$, relational composition

$$R \circ S \stackrel{\text{def}}{=} \{(u,w) \mid \exists v \in X \ (u,v) \in R \text{ and } (v,w) \in S\}$$

for $\cdot$, and reflexive transitive closure for *:

$$R^* \stackrel{\text{def}}{=} \bigcup_{n \geq 0} R^n,$$

where

$$R^0 \stackrel{\text{def}}{=} \iota,$$

$$R^{n+1} \stackrel{\text{def}}{=} R^n \circ R.$$

Still another example is the family of $n \times n$ Boolean matrices with the zero matrix for 0, the identity matrix for 1, componentwise Boolean matrix

addition and multiplication for $+$ and $\cdot$, respectively, and reflexive transitive closure for *. This is really the same as the previous example, where the set X has n elements.

Matrices

Given an arbitrary Kleene algebra $\mathcal{K}$, the set of $n \times n$ matrices over $\mathcal{K}$, which we will denote by $\mathcal{M}(n, \mathcal{K})$, also forms a Kleene algebra. In $\mathcal{M}(2, \mathcal{K})$, for example, the identity elements for $+$ and $\cdot$ are

$$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \text{ and } \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

respectively, and the operations $+$, $\cdot$, and * are given by

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} e & f \\ g & h \end{bmatrix} \stackrel{\text{def}}{=} \begin{bmatrix} a+e & b+f \\ c+g & d+h \end{bmatrix},$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} e & f \\ g & h \end{bmatrix} \stackrel{\text{def}}{=} \begin{bmatrix} ae+bg & af+bh \\ ce+dg & cf+dh \end{bmatrix}, \text{ and}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* \stackrel{\text{def}}{=} \begin{bmatrix} (a+bd^*c)^* & (a+bd^*c)^*bd^* \\ (d+ca^*b)^*ca^* & (d+ca^*b)^* \end{bmatrix}, \tag{A.18}$$

respectively. In general, $+$ and $\cdot$ in $\mathcal{M}(n, \mathcal{K})$ are ordinary matrix addition and multiplication, respectively, the identity for $+$ is the zero matrix, and the identity for $\cdot$ is the identity matrix.

To define E^* for a given $n \times n$ matrix E over $\mathcal{K}$, we proceed by induction on n. If $n = 1$, the structure $\mathcal{M}(n, \mathcal{K})$ is just $\mathcal{K}$, so we are done. For $n > 1$, break E up into four submatrices

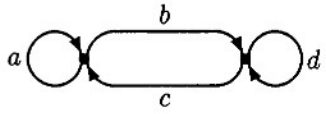
$$E = \left[\begin{array}{c|c} A & B \\ \hline C & D \end{array}\right]$$

such that A and D are square, say $m \times m$ and $(n-m) \times (n-m)$, respectively. By the induction hypothesis, $\mathcal{M}(m, \mathcal{K})$ and $\mathcal{M}(n-m, \mathcal{K})$ are Kleene algebras, so it makes sense to form the asterates of any $m \times m$ or $(n-m) \times (n-m)$ matrix over $\mathcal{K}$, and these matrices will satisfy all the axioms for *. This allows us to define

$$E^* \stackrel{\text{def}}{=} \left[\begin{array}{c|c} (A+BD^*C)^* & (A+BD^*C)^*BD^* \\ \hline (D+CA^*B)^*CA^* & (D+CA^*B)^* \end{array}\right]. \tag{A.19}$$

Compare this definition to (A.18).

The expressions on the right-hand sides of (A.18) and (A.19) may look like they were pulled out of thin air. Where did we get them from? The answer will come to you if you stare really hard at the following mandala:



It can be shown that $\mathcal{M}(n, \mathcal{K})$ is a Kleene algebra under these definitions:

Lemma A.2 *If $\mathcal{K}$ is a Kleene algebra, then so is $\mathcal{M}(n, \mathcal{K})$.*

Proof. Miscellaneous Exercise 24. We must verify that $\mathcal{M}(n, \mathcal{K})$ satisfies the axioms (A.1) through (A.13) of Kleene algebra assuming only that $\mathcal{K}$ does. □

If E is a matrix of indeterminates, and if the inductive construction of E^* given in (A.19) is carried out *symbolically*, then the entries of the resulting matrix E^* will be regular expressions in those indeterminates. This construction generalizes the construction of Lecture 9, which corresponds to the case $m = 1$.

Systems of Linear Equations

It is possible to solve systems of linear equations over a Kleene algebra $\mathcal{K}$. Suppose we are given a set of n variables $x_1, \ldots, x_n$ ranging over $\mathcal{K}$ and a system of n equations of the form

$$x_i = a_{i1}x_1 + \cdots + a_{in}x_n + b_i, \quad 1 \leq i \leq n,$$

where the a_{ij} and b_i are elements of $\mathcal{K}$. Arranging the a_{ij} in an $n \times n$ matrix A, the b_i in a vector b of length n, and the x_i in a vector x of length n, we obtain the matrix-vector equation

$$x = Ax + b. \tag{A.20}$$

It is now not hard to show

Theorem A.3 *The vector A^*b is a solution to (A.20); moreover, it is the $\leq$-least solution in $\mathcal{K}^n$.*

Proof. Miscellaneous Exercise 25. □

Now we use this to give a regular expression equivalent to an arbitrarily given deterministic finite automaton

$$M = (Q, \Sigma, \delta, s, F).$$

Assume without loss of generality that $Q = \{1, 2, \ldots, n\}$. For each $q \in Q$, let X_q denote the set of strings in Σ^* that would be accepted by M if q were the start state; that is,

$$X_q \stackrel{\text{def}}{=} \{x \in \Sigma^* \mid \widehat{\delta}(q, x) \in F\}.$$

The X_q satisfy the following system of equations:

$$X_q = \begin{cases} \sum_{a \in \Sigma} a X_{\delta(q,a)} & \text{if } q \notin F, \\ \sum_{a \in \Sigma} a X_{\delta(q,a)} + 1 & \text{if } q \in F. \end{cases}$$

Moreover, the X_q give the least solution with respect to $\subseteq$. As above, these equations can be arranged in a single matrix-vector equation of the form

$$X = AX + b, \tag{A.21}$$

where A is an $n \times n$ matrix containing sums of elements of Σ, b is a 0-1 vector of length n, and X is a vector consisting of $X_1, \ldots, X_n$. The vector X is the least solution of (A.21). By Theorem A.3,

$$X = A^* b.$$

Compute the matrix A^* symbolically according to (A.19), so that its entries are regular expressions, then multiply by b. A regular expression for $L(M)$ can then be read off from the sth entry of A^*b, where s is the start state of M.

Historical Notes

Salomaa [108] gave the first complete axiomatization of the algebra of regular sets. The algebraic theory was developed extensively in the monograph of Conway [27]. Many others have contributed to the theory, including Redko [103], Backhouse [6], Bloom and Ésik [10], Boffa [11, 12], Gécseg and Peák [41], Krob [74], Kuich and Salomaa [76], and Salomaa and Soittola [109]. The definition of Kleene algebra and the complete axiomatization given here is from Kozen [73].

Lecture 10

Homomorphisms

A *homomorphism* is a map $h : \Sigma^* \to \Gamma^*$ such that for all $x, y \in \Sigma^*$,

$$h(xy) = h(x)h(y), \tag{10.1}$$
$$h(\epsilon) = \epsilon. \tag{10.2}$$

Actually, (10.2) is a consequence of (10.1):

$$\begin{aligned} |h(\epsilon)| &= |h(\epsilon\epsilon)| \\ &= |h(\epsilon)h(\epsilon)| \\ &= |h(\epsilon)| + |h(\epsilon)|; \end{aligned}$$

subtracting $|h(\epsilon)|$ from both sides, we have $|h(\epsilon)| = 0$, therefore $h(\epsilon) = \epsilon$.

It follows from these properties that any homomorphism defined on Σ^* is uniquely determined by its values on Σ. For example, if $h(a) = ccc$ and $h(b) = dd$, then

$$h(abaab) = h(a)h(b)h(a)h(a)h(b) = cccddcccccdd.$$

Moreover, any map $h : \Sigma \to \Gamma^*$ extends uniquely by induction to a homomorphism defined on all of Σ^*. Therefore, in order to specify a homomorphism completely, we need only say what values it takes on elements of Σ.

If $A \subseteq \Sigma^*$, define

$$h(A) \stackrel{\text{def}}{=} \{h(x) \mid x \in A\} \subseteq \Gamma^*,$$

and if $B \subseteq \Gamma^*$, define

$$h^{-1}(B) \stackrel{\text{def}}{=} \{x \mid h(x) \in B\} \subseteq \Sigma^*.$$

The set $h(A)$ is called the *image* of A under h, and the set $h^{-1}(B)$ is called the *preimage* of B under h.

We will show two useful closure properties of the regular sets: any homomorphic image or homomorphic preimage of a regular set is regular.

Theorem 10.1 *Let $h : \Sigma^* \to \Gamma^*$ be a homomorphism. If $B \subseteq \Gamma^*$ is regular, then so is its preimage $h^{-1}(B)$ under h.*

Proof. Let $M = (Q, \Gamma, \delta, s, F)$ be a DFA such that $L(M) = B$. Create a new DFA $M' = (Q, \Sigma, \delta', s, F)$ for $h^{-1}(B)$ as follows. The set of states, start state, and final states of M' are the same as in M. The input alphabet is Σ instead of Γ. The transition function δ' is defined by

$$\delta'(q, a) \stackrel{\text{def}}{=} \widehat{\delta}(q, h(a)).$$

Note that we have to use $\widehat{\delta}$ on the right-hand side, since $h(a)$ need not be a single letter.

Now it follows by induction on $|x|$ that for all $x \in \Sigma^*$,

$$\widehat{\delta'}(q, x) = \widehat{\delta}(q, h(x)). \tag{10.3}$$

For the basis $x = \epsilon$, using (10.1),

$$\widehat{\delta'}(q, \epsilon) = q = \widehat{\delta}(q, \epsilon) = \widehat{\delta}(q, h(\epsilon)).$$

For the induction step, assume that $\widehat{\delta'}(q, x) = \widehat{\delta}(q, h(x))$. Then

$$\begin{aligned}
\widehat{\delta'}(q, xa) &= \delta'(\widehat{\delta'}(q, x), a) && \text{definition of } \widehat{\delta'} \\
&= \delta'(\widehat{\delta}(q, h(x)), a) && \text{induction hypothesis} \\
&= \widehat{\delta}(\widehat{\delta}(q, h(x)), h(a)) && \text{definition of } \delta' \\
&= \widehat{\delta}(q, h(x)h(a)) && \text{Homework 1, Exercise 3} \\
&= \widehat{\delta}(q, h(xa)) && \text{property (10.2) of homomorphisms.}
\end{aligned}$$

Now we can use (10.3) to prove that $L(M') = h^{-1}(L(M))$. For any $x \in \Sigma^*$,

$$\begin{aligned}
x \in L(M') &\iff \widehat{\delta'}(s, x) \in F && \text{definition of acceptance} \\
&\iff \widehat{\delta}(s, h(x)) \in F && \text{by (10.3)} \\
&\iff h(x) \in L(M) && \text{definition of acceptance} \\
&\iff x \in h^{-1}(L(M)) && \text{definition of } h^{-1}(L(M)).
\end{aligned}$$

□

Theorem 10.2 *Let $h : \Sigma^* \to \Gamma^*$ be a homomorphism. If $A \subseteq \Sigma^*$ is regular, then so is its image $h(A)$ under h.*

Proof. For this proof, we will use regular expressions. Let α be a regular expression over Σ such that $L(\alpha) = A$. Let α' be the regular expression obtained by replacing each letter $a \in \Sigma$ appearing in α with the string $h(a) \in \Gamma^*$. For example, if $h(a) = ccc$ and $h(b) = dd$, then

$$((a+b)^*ab)' = (ccc+dd)^*cccdd.$$

Formally, α' is defined by induction:

$$\begin{aligned} a' &= h(a), \quad a \in \Sigma, \\ \emptyset' &= \emptyset, \\ (\beta+\gamma)' &= \beta' + \gamma', \\ (\beta\gamma)' &= \beta'\gamma', \\ \beta^{*\prime} &= \beta'^*. \end{aligned}$$

We claim that for any regular expression β over Σ,

$$L(\beta') = h(L(\beta)); \tag{10.4}$$

in particular, $L(\alpha') = h(A)$. This can be proved by induction on the structure of β. To do this, we will need two facts about homomorphisms: for any pair of subsets $C, D \subseteq \Sigma^*$ and any family of subsets $C_i \subseteq \Sigma^*$, $i \in I$,

$$h(CD) = h(C)h(D), \tag{10.5}$$

$$h\Big(\bigcup_{i\in I} C_i\Big) = \bigcup_{i\in I} h(C_i). \tag{10.6}$$

To prove (10.5),

$$\begin{aligned} h(CD) &= \{h(w) \mid w \in CD\} \\ &= \{h(yz) \mid y \in C,\ z \in D\} \\ &= \{h(y)h(z) \mid y \in C,\ z \in D\} \\ &= \{uv \mid u \in h(C),\ v \in h(D)\} \\ &= h(C)h(D). \end{aligned}$$

To prove (10.6),

$$\begin{aligned} h\Big(\bigcup_i C_i\Big) &= \{h(w) \mid w \in \bigcup_i C_i\} \\ &= \{h(w) \mid \exists i\ w \in C_i\} \\ &= \bigcup_i \{h(w) \mid w \in C_i\} \\ &= \bigcup_i h(C_i). \end{aligned}$$

Now we prove (10.4) by induction. There are two base cases:

$$L(a') = L(h(a)) = \{h(a)\} = h(\{a\}) = h(L(a))$$

and

$$L(\emptyset') = L(\emptyset) = \varnothing = h(\varnothing) = h(L(\emptyset)).$$

The case of ϵ is covered by the other cases, since $\epsilon = \varnothing^*$.

There are three induction cases, one for each of the operators $+$, $\cdot$, and *. For $+$,

$$\begin{aligned}
L((\beta+\gamma)') &= L(\beta' + \gamma') && \text{definition of } ' \\
&= L(\beta') \cup L(\gamma') && \text{definition of } + \\
&= h(L(\beta)) \cup h(L(\gamma)) && \text{induction hypothesis} \\
&= h(L(\beta) \cup L(\gamma)) && \text{property (10.6)} \\
&= h(L(\beta+\gamma)) && \text{definition of } +.
\end{aligned}$$

The proof for $\cdot$ is similar, using property (10.5) instead of (10.6). Finally, for *,

$$\begin{aligned}
&L(\beta^{*\prime}) \\
&= L(\beta'^*) && \text{definition of } ' \\
&= L(\beta')^* && \text{definition of regular expression operator } ^* \\
&= h(L(\beta))^* && \text{induction hypothesis} \\
&= \bigcup_{n\geq 0} h(L(\beta))^n && \text{definition of set operator } ^* \\
&= \bigcup_{n\geq 0} h(L(\beta)^n) && \text{property (10.5)} \\
&= h(\bigcup_{n\geq 0} L(\beta)^n) && \text{property (10.6)} \\
&= h(L(\beta)^*) && \text{definition of set operator } ^* \\
&= h(L(\beta^*)) && \text{definition of regular expression operator } ^*. \qquad \square
\end{aligned}$$

Warning: It is *not* true that A is regular whenever $h(A)$ is. This is not what Theorem 10.1 says. We will show later that the set $\{a^n b^n \mid n \geq 0\}$ is not regular, but the image of this set under the homomorphism $h(a) = h(b) = a$ is the regular set $\{a^n \mid n \text{ is even}\}$. The preimage $h^{-1}(\{a^n \mid n \text{ is even}\})$ is not $\{a^n b^n \mid n \geq 0\}$, but $\{x \in \{a,b\}^* \mid |x| \text{ is even}\}$, which is regular.

Automata with ϵ-transitions

Here is an example of how to use homomorphisms to give a clean treatment of ϵ-transitions. Define an *NFA with ϵ-transitions* to be a structure

$$M = (Q, \Sigma, \epsilon, \Delta, S, F)$$

such that ϵ is a special symbol not in Σ and

$$M_\epsilon = (Q, \Sigma \cup \{\epsilon\}, \Delta, S, F)$$

is an ordinary NFA over the alphabet $\Sigma \cup \{\epsilon\}$. We define acceptance for automata with ϵ-transitions as follows: for any $x \in \Sigma^*$, M *accepts* x if there exists $y \in (\Sigma \cup \{\epsilon\})^*$ such that

- M_ϵ accepts y under the ordinary definition of acceptance for NFAs, and
- x is obtained from y by erasing all occurrences of the symbol ϵ; that is, $x = h(y)$, where

  $$h : (\Sigma \cup \{\epsilon\})^* \to \Sigma^*$$

  is the homomorphism defined by

  $$h(a) \stackrel{\text{def}}{=} a, \quad a \in \Sigma,$$
  $$h(\epsilon) \stackrel{\text{def}}{=} \epsilon.$$

In other words,

$$L(M) \stackrel{\text{def}}{=} h(L(M_\epsilon)).$$

This definition and the definition involving ϵ-closure described in Lecture 6 are equivalent (Miscellaneous Exercise 10). It is immediate from this definition and Theorem 10.2 that the set accepted by any finite automaton with ϵ-transitions is regular.

Hamming Distance

Here is another example of the use of homomorphisms. We can use them to give slick solutions to Exercise 3 of Homework 2 and Miscellaneous Exercise 8, the problems involving Hamming distance. Let $\Sigma = \{0, 1\}$ and consider the alphabet

$$\Sigma \times \Sigma = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

The elements of $\Sigma \times \Sigma$ are ordered pairs, but we write the components one on top of the other. Let $\mathbf{top} : \Sigma \times \Sigma \to \Sigma$ and $\mathbf{bottom} : \Sigma \times \Sigma \to \Sigma$ be the

two projections

$$\mathbf{top}\left(\begin{array}{|c|}\hline a \\ \hline b \\ \hline\end{array}\right) = a,$$

$$\mathbf{bottom}\left(\begin{array}{|c|}\hline a \\ \hline b \\ \hline\end{array}\right) = b.$$

These maps extend uniquely to homomorphisms $(\Sigma \times \Sigma)^* \to \Sigma^*$, which we also denote by **top** and **bottom**. For example,

$$\mathbf{top}\left(\begin{array}{|c|c|c|c|}\hline 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 1 & 1 \\ \hline\end{array}\right) = 0010,$$

$$\mathbf{bottom}\left(\begin{array}{|c|c|c|c|}\hline 0 & 0 & 1 & 0 \\ \hline 0 & 1 & 1 & 1 \\ \hline\end{array}\right) = 0111.$$

Thus we can think of strings in $(\Sigma \times \Sigma)^*$ as consisting of two tracks, and the homomorphisms **top** and **bottom** give the contents of the top and bottom track, respectively.

For fixed k, let D_k be the set of all strings in $(\Sigma \times \Sigma)^*$ containing no more than k occurrences of

$$\begin{array}{|c|}\hline 0 \\ \hline 1 \\ \hline\end{array} \text{ or } \begin{array}{|c|}\hline 1 \\ \hline 0 \\ \hline\end{array}.$$

This is certainly a regular set. Note also that

$$D_k = \{x \in (\Sigma \times \Sigma)^* \mid H(\mathbf{top}(x), \mathbf{bottom}(x)) \leq k\},$$

where H is the Hamming distance function. Now take any regular set $A \subseteq \Sigma^*$, and consider the set

$$\mathbf{top}(\mathbf{bottom}^{-1}(A) \cap D_k). \tag{10.7}$$

Believe it or not, this set is exactly $N_k(A)$, the set of strings in Σ^* of Hamming distance at most k from some string in A. The set $\mathbf{bottom}^{-1}(A)$ is the set of strings whose bottom track is in A; the set $\mathbf{bottom}^{-1}(A) \cap D_k$ is the set of strings whose bottom track is in A and whose top track is of Hamming distance at most k from the bottom track; and the set (10.7) is the set of top tracks of all such strings.

Moreover, the set (10.7) is a regular set, because the regular sets are closed under intersection, homomorphic image, and homomorphic preimage.

Lecture 11

Limitations of Finite Automata

We have studied what finite automata can do; let's see what they cannot do. The canonical example of a nonregular set (one accepted by no finite automaton) is

$$B = \{a^n b^n \mid n \geq 0\} = \{\epsilon, ab, aabb, aaabbb, aaaabbbb, \ldots\},$$

the set of all strings of the form a^*b^* with equally many a's and b's.

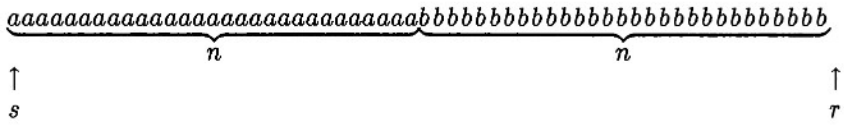
Intuitively, in order to accept the set B, an automaton scanning a string of the form a^*b^* would have to remember when passing the center point between the a's and b's how many a's it has seen, since it would have to compare that with the number of b's and accept iff the two numbers are the same.

$$aaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbb$$
$$\uparrow$$
$$q$$

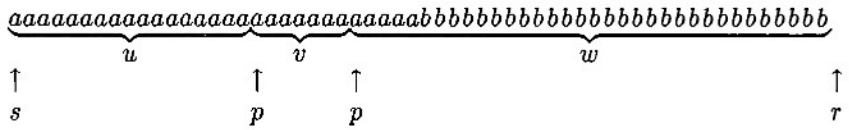
Moreover, it would have to do this for arbitrarily long strings of a's and b's, much longer than the number of states. This is an unbounded amount of information, and there is no way it can remember this with only finite memory. All it "knows" at that point is represented in the state q it is in, which is only a finite amount of information. You might at first think there may be some clever strategy, such as counting mod 3, 5, and 7, or something similar. But any such attempt is doomed to failure: you cannot

distinguish between infinitely many different cases with only finitely many states.

This is just an informal argument. But we can easily give a formal proof by contradiction that B is not regular. Assuming that B were regular, there would be a DFA M such that $L(M) = B$. Let k be the number of states of this alleged M. Consider the action of M on input $a^n b^n$, where $n \gg k$. It starts in its start state s. Since the string $a^n b^n$ is in B, M must accept it, thus M must be in some final state r after scanning $a^n b^n$.



Since $n \gg k$, by the pigeonhole principle there must be some state p that the automaton enters more than once while scanning the initial sequence of a's. Break up the string $a^n b^n$ into three pieces u, v, w, where v is the string of a's scanned between two occurrences of the state p, as illustrated in the following picture:

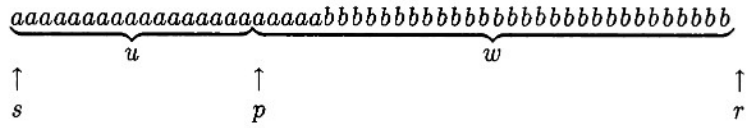


Let $j = |v| > 0$. In this example, $j = 7$. Then

$$\begin{aligned}
&\widehat{\delta}(s, u) = p, \\
&\widehat{\delta}(p, v) = p, \\
&\widehat{\delta}(p, w) = r \in F.
\end{aligned}$$

The string v could be deleted and the resulting string would be erroneously accepted:

$$\begin{aligned}
\widehat{\delta}(s, uw) &= \widehat{\delta}(\widehat{\delta}(s, u), w) \\
&= \widehat{\delta}(p, w) \\
&= r \in F.
\end{aligned}$$



It's erroneous because after deleting v, the number of a's is strictly less than the number of b's: $uw = a^{n-j} b^n \in L(M)$, but $uw \notin B$. This contradicts our assumption that $L(M) = B$.

We could also insert extra copies of v and the resulting string would be erroneously accepted. For example, $uv^3w = a^{n+2j}b^n$ is erroneously accepted:

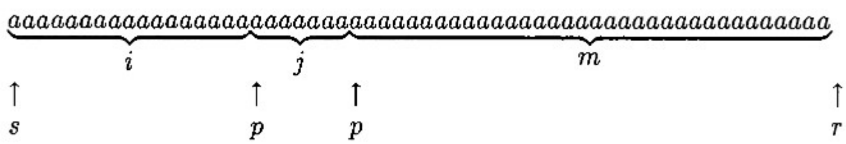
$$\begin{aligned}\widehat{\delta}(s, uvvvw) &= \widehat{\delta}(\widehat{\delta}(\widehat{\delta}(\widehat{\delta}(\widehat{\delta}(s,u),v),v),v),w) \\ &= \widehat{\delta}(\widehat{\delta}(\widehat{\delta}(\widehat{\delta}(p,v),v),v),w) \\ &= \widehat{\delta}(\widehat{\delta}(\widehat{\delta}(p,v),v),w) \\ &= \widehat{\delta}(\widehat{\delta}(p,v),w) \\ &= \widehat{\delta}(p,w) \\ &= r \in F.\end{aligned}$$

For another example of a nonregular set, consider

$$\begin{aligned}C &= \{a^{2^n} \mid n \geq 0\} \\ &= \{x \in \{a\}^* \mid |x| \text{ is a power of } 2\} \\ &= \{a, a^2, a^4, a^8, a^{16}, \ldots\}.\end{aligned}$$

This set is also nonregular. Suppose (again for a contradiction) that $L(M) = C$ for some DFA M. Let k be the number of states of M. Let $n \gg k$ and consider the action of M on input $a^{2^n} \in C$. Since $n \gg k$, by the pigeonhole principle the automaton must repeat a state p while scanning the first n symbols of a^{2^n}. Thus $2^n = i + j + m$ for some i, j, m with $0 < j \leq n$ and

$$\begin{aligned}\widehat{\delta}(s, a^i) &= p, \\ \widehat{\delta}(p, a^j) &= p, \\ \widehat{\delta}(p, a^m) &= r \in F.\end{aligned}$$



As above, we could insert an extra a^j to get a^{2^n+j}, and this string would be erroneously accepted:

$$\begin{aligned}\widehat{\delta}(s, a^{2^n+j}) &= \widehat{\delta}(s, a^i a^j a^j a^m) \\ &= \widehat{\delta}(\widehat{\delta}(\widehat{\delta}(\widehat{\delta}(s,a^i),a^j),a^j),a^m) \\ &= \widehat{\delta}(\widehat{\delta}(\widehat{\delta}(p,a^j),a^j),a^m) \\ &= \widehat{\delta}(\widehat{\delta}(p,a^j),a^m) \\ &= \widehat{\delta}(p,a^m) \\ &= r \in F.\end{aligned}$$

This is erroneous because $2^n + j$ is not a power of 2:

$$2^n + j \leq 2^n + n$$

$$< 2^n + 2^n$$
$$= 2^{n+1}$$

and 2^{n+1} is the next power of 2 greater than 2^n.

The Pumping Lemma

We can encapsulate the arguments above in a general theorem called the *pumping lemma*. This lemma is very useful in proving sets nonregular. The idea is that whenever an automaton scans a long string (longer than the number of states) and accepts, there must be a repeated state, and extra copies of the segment of the input between the two occurrences of that state can be inserted and the resulting string is still accepted.

Theorem 11.1 **(Pumping lemma)** *Let A be a regular set. Then the following property holds of A:*

(P) *There exists $k \geq 0$ such that for any strings x, y, z with $xyz \in A$ and $|y| \geq k$, there exist strings u, v, w such that $y = uvw$, $v \neq \epsilon$, and for all $i \geq 0$, the string $xuv^iwz \in A$.*

Informally, if A is regular, then for any string in A and any sufficiently long substring y of that string, y has a nonnull substring v of which you can pump in as many copies as you like and the resulting string is still in A.

We have essentially already proved this theorem. Think of k as the number of states of a DFA accepting A. Since y is at least as long as the number of states, there must be a repeated state while scanning y. The string v is the substring between the two occurrences of that state. We can pump in as many copies of v as we want (or delete v—this would be the case $i = 0$), and the resulting string is still accepted.

Games with the Demon

The pumping lemma is often used to show that certain sets are nonregular. For this purpose we usually use it in its contrapositive form:

Theorem 11.2 **(Pumping lemma, contrapositive form)** *Let A be a set of strings. Suppose that the following property holds of A.*

($\neg$P) *For all $k \geq 0$ there exist strings x, y, z such that $xyz \in A$, $|y| \geq k$, and for all u, v, w with $y = uvw$ and $v \neq \epsilon$, there exists an $i \geq 0$ such that $xuv^iwz \notin A$.*

Then A is not regular.

To use the pumping lemma to prove that a given set A is nonregular, we need to establish that ($\neg$P) holds of A. Because of the alternating "for all/there exists" form of ($\neg$P), we can think of this as a game between you and a demon. You want to show that A is nonregular, and the demon wants to show that A is regular. The game proceeds as follows:

1. The demon picks k. (If A really is regular, the demon's best strategy here is to pick k to be the number of states of a DFA for A.)
2. You pick x, y, z such that $xyz \in A$ and $|y| \geq k$.
3. The demon picks u, v, w such that $y = uvw$ and $v \neq \epsilon$.
4. You pick $i \geq 0$.

You win if $xuv^iwz \notin A$, and the demon wins if $xuv^iwz \in A$.

The property ($\neg$P) for A is equivalent to saying that you have a *winning strategy* in this game. This means that by playing optimally, you can always win no matter what the demon does in steps 1 and 3.

If you can show that you have a winning strategy, you have essentially shown that the condition ($\neg$P) holds for A, therefore by Theorem 11.2, A is not regular.

We have thus reduced the problem of showing that a given set is nonregular to the puzzle of finding a winning strategy in the corresponding demon game. Each nonregular set gives a different game. We'll give several examples in Lecture 12.

Warning: Although there do exist stronger versions that give necessary and sufficient conditions for regularity (Miscellaneous Exercise 44), the version of the pumping lemma given here gives only a necessary condition; there exist sets satisfying (P) that are nonregular (Miscellaneous Exercise 43). You cannot show that a set *is* regular by showing that it satisfies (P). To show a given set *is* regular, you should construct a finite automaton or regular expression for it.

Historical Notes

The pumping lemma for regular sets is due to Bar-Hillel, Perles, and Shamir [8]. This version gives only a necessary condition for regularity. Necessary and sufficient conditions are given by Stanat and Weiss [117], Jaffe [62], and Ehrenfeucht, Parikh, and Rozenberg [33].

Lecture 12

Using the Pumping Lemma

Example 12.1 Let's use the pumping lemma in the form of the demon game to show that the set

$$A = \{a^n b^m \mid n \geq m\}$$

is not regular. The set A is the set of strings in a^*b^* with no more b's than a's. The demon, who is betting that A is regular, picks some number k. A good response for you is to pick $x = a^k$, $y = b^k$, and $z = \epsilon$. Then $xyz = a^k b^k \in A$ and $|y| = k$; so far you have followed the rules. The demon must now pick u, v, w such that $y = uvw$ and $v \neq \epsilon$. Say the demon picks u, v, w of length j, m, n, respectively, with $k = j + m + n$ and $m > 0$. No matter what the demon picks, you can take $i = 2$ and you win:

$$\begin{aligned} xuv^2wz &= a^k b^j b^m b^m b^n \\ &= a^k b^{j+2m+n} \\ &= a^k b^{k+m}, \end{aligned}$$

which is not in A, because the number of b's is strictly larger than the number of a's.

This strategy always leads to victory for you in the demon game associated with the set A. As we argued in Lecture 11, this is tantamount to showing that A is nonregular. □

Example 12.2 For another example, take the set

$$C = \{a^{n!} \mid n \geq 0\}.$$

We would like to show that this set is not regular. This one is a little harder. It is an example of a nonregular set over a single-letter alphabet. Intuitively, it is not regular because the differences in the lengths of the successive elements of the set grow too fast.

Suppose the demon chooses k. A good choice for you is $x = z = \epsilon$ and $y = a^{k!}$. Then $xyz = a^{k!} \in C$ and $|y| = k! \geq k$, so you have not cheated. The demon must now choose u, v, w such that $y = uvw$ and $v \neq \epsilon$. Say the demon chooses u, v, w of length j, m, n, respectively, with $k! = j + m + n$ and $m > 0$. You now need to find i such that $xuv^iwz \notin C$; in other words, $|xuv^iwz| \neq p!$ for any p. Note that for any i,

$$|xuv^iwz| = j + im + n = k! + (i-1)m,$$

so you will win if you can choose i such that $k! + (i-1)m \neq p!$ for any p. Take $i = (k+1)! + 1$. Then

$$k! + (i-1)m = k! + (k+1)!m = k!(1 + m(k+1)),$$

and we want to show that this cannot be $p!$ for any p. But if

$$p! = k!(1 + m(k+1)),$$

then we could divide both sides by $k!$ to get

$$p(p-1)(p-2)\cdots(k+2)(k+1) = 1 + m(k+1),$$

which is impossible, because the left-hand side is divisible by $k+1$ and the right-hand side is not. □

A Trick

When trying to show that a set is nonregular, one can often simplify the problem by using one of the closure properties of regular sets. This often allows us to reduce a complicated set to a simpler set that is already known to be nonregular, thereby avoiding the use of the pumping lemma.

To illustrate, consider the set

$$D = \{x \in \{a, b\}^* \mid \#a(x) = \#b(x)\}.$$

To show that this set is nonregular, suppose for a contradiction that it were regular. Then the set

$$D \cap a^*b^*$$

would also be regular, since the intersection of two regular sets is always regular (the product construction, remember?). But

$$D \cap L(a^*b^*) = \{a^n b^n \mid n \geq 0\},$$

which we have already shown to be nonregular. This is a contradiction.

For another illustration of this trick, consider the set A of Example 12.1 above:

$$A = \{a^n b^m \mid n \geq m\},$$

the set of strings $x \in L(a^*b^*)$ with no more b's than a's. By Exercise 2 of Homework 2, if A were regular, then so would be the set

$$\mathbf{rev}\ A = \{b^m a^n \mid n \geq m\},$$

and by interchanging a and b, we would get that the set

$$A' = \{a^m b^n \mid n \geq m\}$$

is also regular. Formally, "interchanging a and b" means applying the homomorphism $a \mapsto b$, $b \mapsto a$. But then the intersection

$$A \cap A' = \{a^n b^n \mid n \geq 0\}$$

would be regular. But we have already shown using the pumping lemma that this set is nonregular. This is a contradiction.

Ultimate Periodicity

Let U be a subset of $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$, the natural numbers.

The set U is said to be *ultimately periodic* if there exist numbers $n \geq 0$ and $p > 0$ such that for all $m \geq n$, $m \in U$ if and only if $m + p \in U$. The number p is called a *period* of U.

In other words, except for a finite initial part (the numbers less than n), numbers are in or out of the set U according to a repeating pattern. For example, consider the set

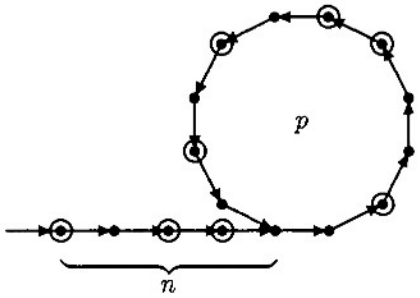
$$\{0, 3, 7, 11, 19, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, \ldots\}.$$

Starting at 20, every third element is in the set, therefore this set is ultimately periodic with $n = 20$ and $p = 3$. Note that neither n nor p is unique; for example, for this set we could also have taken $n = 21$ and $p = 6$, or $n = 100$ and $p = 33$.

Regular sets over a single-letter alphabet $\{a\}$ and ultimately periodic subsets of $\mathbb{N}$ are strongly related:

Theorem 12.3 *Let $A \subseteq \{a\}^*$. Then A is regular if and only if the set $\{m \mid a^m \in A\}$, the set of lengths of strings in A, is ultimately periodic.*

Proof. If A is regular, then any DFA for it consists of a finite tail of some length, say $n \geq 0$, followed by a loop of length $p > 0$ (plus possibly some inaccessible states, which can be thrown out).



To see this, consider any DFA for A. Since the alphabet is $\{a\}$ and the machine is deterministic, there is exactly one edge out of each state, and it has label a. Thus there is a unique path through the automaton starting at the start state. Follow this path until the first time you see a state that you have seen before. Since the collection of states is finite, eventually this must happen. The first time this happens, we have discovered a loop. Let p be the length of the loop, and let n be the length of the initial tail preceding the first time we enter the loop. For all strings a^m with $m \geq n$, the automaton is in the loop part after scanning a^m. Then a^m is accepted iff a^{m+p} is, since the automaton moves around the loop once under the last p a's of a^{m+p}. Thus it is in the same state after scanning both strings. Therefore, the set of lengths of accepted strings is ultimately periodic.

Conversely, given any ultimately periodic set U, let p be the period and let n be the starting point of the periodic behavior. Then one can build an automaton with a tail of length n and loop of length p accepting exactly the set of strings in $\{a\}^*$ whose lengths are in U. For example, for the ultimately periodic set

$$\{0, 3, 7, 11, 19, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, \ldots\}$$

mentioned above, the automaton would be

□

Corollary 12.4 *Let A be any regular set over any finite alphabet Σ, not necessarily consisting of a single letter. Then the set*

$$\textbf{lengths}\ A = \{|x| \mid x \in A\}$$

of lengths of strings in A is ultimately periodic.

Proof. Define the homomorphism $h : \Sigma \to \{a\}$ by $h(b) = a$ for all $b \in \Sigma$. Then $h(x) = a^{|x|}$. Since h preserves length, we have that **lengths** $A =$ **lengths** $h(A)$. But $h(A)$ is a regular subset of $\{a\}^*$, since the regular sets are closed under homomorphic image; therefore, by Theorem 12.3, **lengths** $h(A)$ is ultimately periodic. □

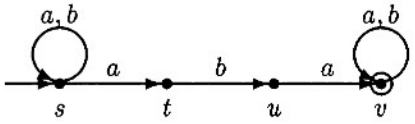
Historical Notes

A general treatment of ultimate periodicity and regularity-preserving functions is given in Seiferas and McNaughton [113]; see Miscellaneous Exercise 34.

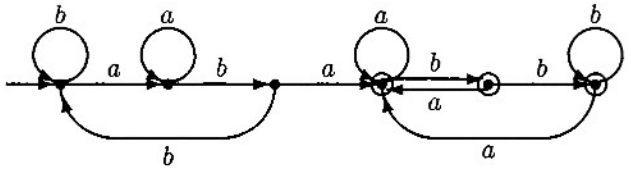
Lecture 13

DFA State Minimization

By now you have probably come across several situations in which you have observed that some automaton could be simplified either by deleting states inaccessible from the start state or by collapsing states that were equivalent in some sense. For example, if you were to apply the subset construction to the NFA

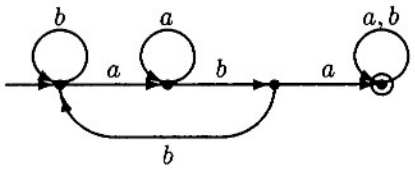


accepting the set of all strings containing the substring aba, you would obtain a DFA with $2^4 = 16$ states. However, all except six of these states are inaccessible. Deleting them, you would obtain the DFA



From left to right, the states of this DFA correspond to the subsets $\{s\}$, $\{s,t\}$, $\{s,u\}$, $\{s,t,v\}$, $\{s,u,v\}$, $\{s,v\}$.

Now, note that the rightmost three states of this DFA might as well be collapsed into a single state, since they are all accept states, and once the machine enters one of them it cannot escape. Thus this DFA is equivalent to



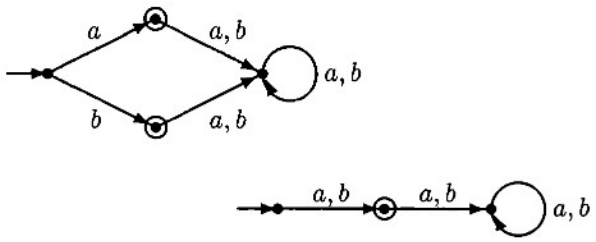
This is a simple example in which the equivalence of states is obvious, but sometimes it is not so obvious. In this and the next lecture we will develop a mechanical method to find all equivalent states of any given DFA and collapse them. This will give a DFA for any given regular set A that has as few states as possible. An amazing fact is that every regular set has a minimal DFA that is unique up to isomorphism, and there is a purely mechanical method for constructing it from any given DFA for A.

Say we are given a DFA $M = (Q, \Sigma, \delta, s, F)$ for A. The minimization process consists of two stages:

1. Get rid of inaccessible states; that is, states q for which there exists no string $x \in \Sigma^*$ such that $\widehat{\delta}(s, x) = q$.
2. Collapse "equivalent" states.

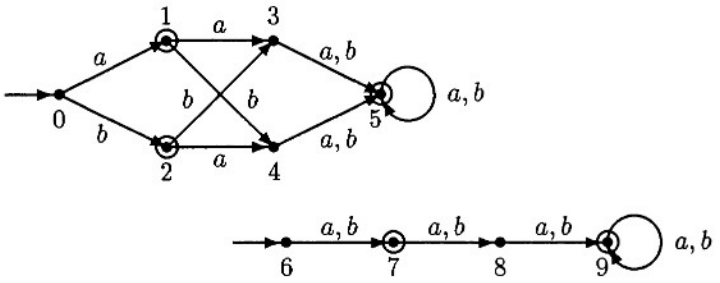
Removing inaccessible states surely does not change the set accepted. It is quite straightforward to see how to do this mechanically using depth-first search on the transition graph. Let us then assume that this has been done. For stage 2, we need to say what we mean by "equivalent" and how we do the collapsing. Let's look at some examples before giving a formal definition.

Example 13.1



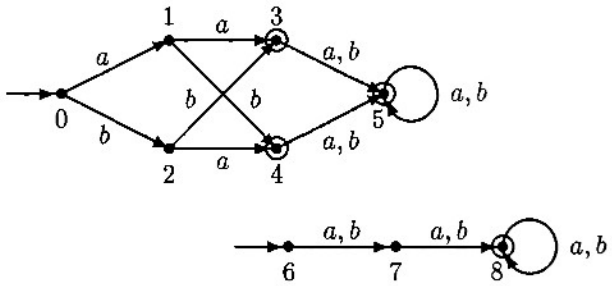
These automata both accept the set $\{a, b\}$. The automaton with four states goes to different states depending on the first input symbol, but there's really no reason for the states to be separate. They are equivalent and can be collapsed into one state, giving the automaton with three states. □

Example 13.2



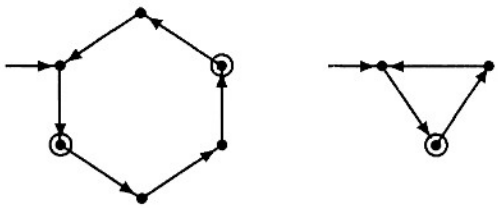
This example is a little more complicated. The automata both accept the set $\{a, b\} \cup \{\text{strings of length 3 or greater}\}$. In the first automaton, states 3 and 4 are equivalent, since they both go to state 5 under both input symbols, so there's no reason to keep them separate. Once we collapse them, we can collapse 1 and 2 for the same reason, giving the second automaton. State 0 becomes state 6; states 1 and 2 collapse to become state 7; states 3 and 4 collapse to become state 8; and state 5 becomes state 9. □

Example 13.3



Here we have modified the first automaton by making states 3, 4 accept states instead of 1, 2. Now states 3, 4, 5 are equivalent and can be collapsed. These become state 8 of the second automaton. The set accepted is the set of all strings of length at least two. □

Example 13.4



These automata both accept the set $\{a^m \mid m \equiv 1 \bmod 3\}$ (edge labels are omitted). In the left automaton, diametrically opposed states are equivalent and can be collapsed, giving the automaton on the right. □

The Quotient Construction

How do we know in general when two states can be collapsed safely without changing the set accepted? How do we do the collapsing formally? Is there a fast algorithm for doing it? How can we determine whether any further collapsing is possible?

Surely we never want to collapse an accept state p and a reject state q, because if $p = \widehat{\delta}(s,x) \in F$ and $q = \widehat{\delta}(s,y) \notin F$, then x must be accepted and y must be rejected even after collapsing, so there is no way to declare the collapsed state to be an accept or reject state without error. Also, if we collapse p and q, then we had better also collapse $\delta(p,a)$ and $\delta(q,a)$ to maintain determinism. These two observations together imply inductively that we cannot collapse p and q if $\widehat{\delta}(p,x) \in F$ and $\widehat{\delta}(q,x) \notin F$ for some string x.

It turns out that this criterion is necessary and sufficient for deciding whether a pair of states can be collapsed. That is, if there exists a string x such that $\widehat{\delta}(p,x) \in F$ and $\widehat{\delta}(q,x) \notin F$ or vice versa, then p and q cannot be safely collapsed; and if no such x exists, then they can.

Here's how we show this formally. We first define an equivalence relation $\approx$ on Q by

$$p \approx q \stackrel{\text{def}}{\iff} \forall x \in \Sigma^* \ (\widehat{\delta}(p,x) \in F \iff \widehat{\delta}(q,x) \in F).$$

This definition is just a formal restatement of the collapsing criterion. It is not hard to argue that the relation $\approx$ is indeed an equivalence relation: it is

- *reflexive*: $p \approx p$ for all p;
- *symmetric*: if $p \approx q$, then $q \approx p$; and
- *transitive*: if $p \approx q$ and $q \approx r$, then $p \approx r$.

As with all equivalence relations, $\approx$ partitions the set on which it is defined into disjoint *equivalence classes*:

$$[p] \stackrel{\text{def}}{=} \{q \mid q \approx p\}.$$

Every element $p \in Q$ is contained in exactly one equivalence class $[p]$, and

$$p \approx q \iff [p] = [q].$$

We now define a DFA $M/\!\approx$ called the *quotient automaton*, whose states correspond to the equivalence classes of $\approx$. This construction is called a *quotient construction* and is quite common in algebra. We will see a more general account of it in Supplementary Lectures C and D.

There is one state of $M/\approx$ for each $\approx$-equivalence class. In fact, formally, the states of $M/\approx$ *are* the equivalence classes; this is the mathematical way of "collapsing" equivalent states.

Define

$$M/\approx \stackrel{\text{def}}{=} (Q', \Sigma, \delta', s', F'),$$

where

$$\begin{aligned} Q' &\stackrel{\text{def}}{=} \{[p] \mid p \in Q\}, \\ \delta'([p], a) &\stackrel{\text{def}}{=} [\delta(p, a)], \\ s' &\stackrel{\text{def}}{=} [s], \\ F' &\stackrel{\text{def}}{=} \{[p] \mid p \in F\}. \end{aligned} \tag{13.1}$$

There is a subtle but important point involving the definition of δ' in (13.1): we need to show that it is *well-defined.* Note that the action of δ' on the equivalence class $[p]$ is defined in terms of p. It is conceivable that a different choice of representative of the class $[p]$ (i.e., some q such that $q \approx p$) might lead to a different right-hand side in (13.1). Lemma 13.5 says exactly that this does not happen.

Lemma 13.5 *If $p \approx q$, then $\delta(p, a) \approx \delta(q, a)$. Equivalently, if $[p] = [q]$, then $[\delta(p, a)] = [\delta(q, a)]$.*

Proof. Suppose $p \approx q$. Let $a \in \Sigma$ and $y \in \Sigma^*$.

$$\begin{aligned} \widehat{\delta}(\delta(p, a), y) \in F &\iff \widehat{\delta}(p, ay) \in F \\ &\iff \widehat{\delta}(q, ay) \in F \qquad \text{since } p \approx q \\ &\iff \widehat{\delta}(\delta(q, a), y) \in F. \end{aligned}$$

Since y was arbitrary, $\delta(p, a) \approx \delta(q, a)$ by definition of $\approx$. □

Lemma 13.6 $p \in F \iff [p] \in F'$.

Proof. The direction $\Rightarrow$ is immediate from the definition of F'. For the direction $\Leftarrow$, we need to show that if $p \approx q$ and $p \in F$, then $q \in F$. In other words, every $\approx$-equivalence class is either a subset of F or disjoint from F. This follows immediately by taking $x = \epsilon$ in the definition of $p \approx q$. □

Lemma 13.7 *For all $x \in \Sigma^*$, $\widehat{\delta'}([p], x) = [\widehat{\delta}(p, x)]$.*

Proof. By induction on $|x|$.

Basis

For $x = \epsilon$,

$$\begin{aligned}\widehat{\delta'}([p], \epsilon) &= [p] && \text{definition of } \widehat{\delta'} \\ &= [\widehat{\delta}(p, \epsilon)] && \text{definition of } \widehat{\delta}.\end{aligned}$$

Induction step

Assume $\widehat{\delta'}([p], x) = [\widehat{\delta}(p, x)]$, and let $a \in \Sigma$.

$$\begin{aligned}\widehat{\delta'}([p], xa) &= \delta'(\widehat{\delta'}([p], x), a) && \text{definition of } \widehat{\delta'} \\ &= \delta'([\widehat{\delta}(p, x)], a) && \text{induction hypothesis} \\ &= [\delta(\widehat{\delta}(p, x), a)] && \text{definition of } \delta' \\ &= [\widehat{\delta}(p, xa)] && \text{definition of } \widehat{\delta}. \qquad \square\end{aligned}$$

Theorem 13.8 $L(M/\approx) = L(M)$.

Proof. For $x \in \Sigma^*$,

$$\begin{aligned}x \in L(M/\approx) &\iff \widehat{\delta'}(s', x) \in F' && \text{definition of acceptance} \\ &\iff \widehat{\delta'}([s], x) \in F' && \text{definition of } s' \\ &\iff [\widehat{\delta}(s, x)] \in F' && \text{Lemma 13.7} \\ &\iff \widehat{\delta}(s, x) \in F && \text{Lemma 13.6} \\ &\iff x \in L(M) && \text{definition of acceptance.} \qquad \square\end{aligned}$$

$M/\approx$ Cannot Be Collapsed Further

It is conceivable that after doing the quotient construction once, we might be able to collapse even further by doing it again. It turns out that once is enough. To see this, let's do the quotient construction a second time. Define

$$[p] \sim [q] \overset{\text{def}}{\iff} \forall x \in \Sigma^* \ (\widehat{\delta'}([p], x) \in F' \iff \widehat{\delta'}([q], x) \in F').$$

This is exactly the same definition as $\approx$ above, only applied to the quotient automaton $M/\approx$. We use the notation $\sim$ for the equivalence relation on Q' to distinguish it from the relation $\approx$ on Q. Now

$$\begin{aligned}&[p] \sim [q] \\ &\Rightarrow \forall x \ (\widehat{\delta'}([p], x) \in F' \iff \widehat{\delta'}([q], x) \in F') && \text{definition of } \sim \\ &\Rightarrow \forall x \ ([\widehat{\delta}(p, x)] \in F' \iff [\widehat{\delta}(q, x)] \in F') && \text{Lemma 13.7}\end{aligned}$$

$\Rightarrow \forall x\ (\widehat{\delta}(p,x) \in F \iff \widehat{\delta}(q,x) \in F)$ Lemma 13.6

$\Rightarrow p \approx q$ definition of $\approx$

$\Rightarrow [p] = [q]$.

Thus any two equivalent states of $M/\approx$ are in fact equal, and the collapsing relation $\sim$ on Q' is just the identity relation $=$.

Lecture 14

A Minimization Algorithm

Here is an algorithm for computing the collapsing relation $\approx$ for a given DFA M with no inaccessible states. Our algorithm will mark (unordered) pairs of states $\{p, q\}$. A pair $\{p, q\}$ will be marked as soon as a reason is discovered why p and q are *not* equivalent.

1. Write down a table of all pairs $\{p, q\}$, initially unmarked.
2. Mark $\{p, q\}$ if $p \in F$ and $q \notin F$ or vice versa.
3. Repeat the following until no more changes occur: if there exists an unmarked pair $\{p, q\}$ such that $\{\delta(p, a), \delta(q, a)\}$ is marked for some $a \in \Sigma$, then mark $\{p, q\}$.
4. When done, $p \approx q$ iff $\{p, q\}$ is not marked.

Here are some things to note about this algorithm:

- If $\{p, q\}$ is marked in step 2, then p and q are surely not equivalent: take $x = \epsilon$ in the definition of $\approx$.
- We may have to look at the same pair $\{p, q\}$ many times in step 3, since any change in the table may suddenly allow $\{p, q\}$ to be marked. We stop only after we make an entire pass through the table with no new marks.

- The algorithm runs for only a finite number of steps, since there are only $\binom{n}{2}$ possible marks that can be made,[1] and we have to make at least one new mark in each pass to keep going.
- Step 4 is really a statement of the theorem that the algorithm correctly computes $\approx$. This requires proof, which we defer until later.

Example 14.1 Let's minimize the automaton of Example 13.2 of Lecture 13.

		a	b
$\rightarrow$	0	1	2
	$1F$	3	4
	$2F$	4	3
	3	5	5
	4	5	5
	$5F$	5	5

Here is the table built in step 1. Initially all pairs are unmarked.

0					
–	1				
–	–	2			
–	–	–	3		
–	–	–	–	4	
–	–	–	–	–	5

After step 2, all pairs consisting of one accept state and one nonaccept state have been marked.

0					
✓	1				
✓	–	2			
–	✓	✓	3		
–	✓	✓	–	4	
✓	–	–	✓	✓	5

Now look at an unmarked pair, say $\{0,3\}$. Under input a, 0 and 3 go to 1 and 5, respectively (write: $\{0,3\} \rightarrow \{1,5\}$). The pair $\{1,5\}$ is not marked, so we don't mark $\{0,3\}$, at least not yet. Under input b, $\{0,3\} \rightarrow \{2,5\}$, which is not marked, so we still don't mark $\{0,3\}$. We then look at unmarked pairs $\{0,4\}$ and $\{1,2\}$ and find out we cannot mark them yet for the same reasons. But for $\{1,5\}$, under input a, $\{1,5\} \rightarrow \{3,5\}$, and $\{3,5\}$ is marked, so we mark $\{1,5\}$. Similarly, under input a, $\{2,5\} \rightarrow \{4,5\}$ which is marked, so we mark $\{2,5\}$. Under both inputs a and b, $\{3,4\} \rightarrow \{5,5\}$, which is never marked (it's not even in the table), so we do not mark $\{3,4\}$. After the first

[1] $\binom{n}{k} \stackrel{\text{def}}{=} \frac{n!}{k!(n-k)!}$, the number of subsets of size k in a set of size n.

pass of step 3, the table looks like

0					
✓	1				
✓	–	2			
–	✓	✓	3		
–	✓	✓	–	4	
✓	✓	✓	✓	✓	5

Now we make another pass through the table. As before, $\{0,3\} \to \{1,5\}$ under input a, but this time $\{1,5\}$ is marked, so we mark $\{0,3\}$. Similarly, $\{0,4\} \to \{2,5\}$ under input b, and $\{2,5\}$ is marked, so we mark $\{0,4\}$. This gives

0					
✓	1				
✓	–	2			
✓	✓	✓	3		
✓	✓	✓	–	4	
✓	✓	✓	✓	✓	5

Now we check the remaining unmarked pairs and find out that $\{1,2\} \to \{3,4\}$ and $\{3,4\} \to \{5,5\}$ under both a and b, and neither $\{3,4\}$ nor $\{5,5\}$ is marked, so there are no new marks. We are left with unmarked pairs $\{1,2\}$ and $\{3,4\}$, indicating that $1 \approx 2$ and $3 \approx 4$. □

Example 14.2 Now let's do Example 13.4 of Lecture 13.

		a
→	0	1
	$1F$	2
	2	3
	3	4
	$4F$	5
	5	0

Here is the table after step 2.

0					
✓	1				
–	✓	2			
–	✓	–	3		
✓	–	✓	✓	4	
–	✓	–	–	✓	5

Then:

- $\{0,2\} \to \{1,3\}$, which is marked, so mark $\{0,2\}$.

- $\{0,3\} \to \{1,4\}$, which is not marked, so do not mark $\{0,3\}$.
- $\{0,5\} \to \{0,1\}$, which is marked, so mark $\{0,5\}$.
- $\{1,4\} \to \{2,5\}$, which is not marked, so do not mark $\{1,4\}$.
- $\{2,3\} \to \{3,4\}$, which is marked, so mark $\{2,3\}$.
- $\{2,5\} \to \{0,3\}$, which is not marked, so do not mark $\{2,5\}$.
- $\{3,5\} \to \{0,4\}$, which is marked, so mark $\{3,5\}$.

After the first pass, the table looks like this:

```
0
✓  1
✓  ✓  2
–  ✓  ✓  3
✓  –  ✓  ✓  4
✓  ✓  –  ✓  ✓  5
```

Now do another pass. We discover that $\{0,3\} \to \{1,4\} \to \{2,5\} \to \{0,3\}$ and none of these are marked, so we are done. Thus $0 \approx 3$, $1 \approx 4$, and $2 \approx 5$. □

Correctness of the Collapsing Algorithm

Theorem 14.3 *The pair $\{p,q\}$ is marked by the above algorithm if and only if there exists $x \in \Sigma^*$ such that $\widehat{\delta}(p,x) \in F$ and $\widehat{\delta}(q,x) \notin F$ or vice versa; i.e., if and only if $p \not\approx q$.*

Proof. This is easily proved by induction. We leave the proof as an exercise (Miscellaneous Exercise 49). □

A nice way to look at the algorithm is as a finite automaton itself. Let

$$\mathcal{Q} = \{\{p,q\} \mid p,q \in Q,\ p \neq q\}.$$

There are $\binom{n}{2}$ elements of $\mathcal{Q}$, where n is the size of Q. Define a nondeterministic "transition function"

$$\Delta : \mathcal{Q} \to 2^{\mathcal{Q}}$$

on $\mathcal{Q}$ as follows:

$$\Delta(\{p,q\},a) = \{\{p',q'\} \mid p = \delta(p',a),\ q = \delta(q',a)\}.$$

Define a set of "start states" $\mathcal{S} \subseteq \mathcal{Q}$ as follows:

$$\mathcal{S} = \{\{p,q\} \mid p \in F,\ q \notin F\}.$$

(We don't need to write "... or vice versa" because $\{p, q\}$ is an unordered pair.) Step 2 of the algorithm marks the elements of $\mathcal{S}$, and step 3 marks pairs in $\Delta(\{p, q\}, a)$ when $\{p, q\}$ is marked for any $a \in \Sigma$. In these terms, Theorem 14.3 says that $p \not\approx q$ iff $\{p, q\}$ is accessible in this automaton.

Lecture 15

Myhill–Nerode Relations

Two deterministic finite automata

$$M = (Q_M, \Sigma, \delta_M, s_M, F_M),$$
$$N = (Q_N, \Sigma, \delta_N, s_N, F_N)$$

are said to be *isomorphic* (Greek for "same form") if there is a one-to-one and onto mapping $f : Q_M \to Q_N$ such that

- $f(s_M) = s_N$,
- $f(\delta_M(p, a)) = \delta_N(f(p), a)$ for all $p \in Q_M$, $a \in \Sigma$, and
- $p \in F_M$ iff $f(p) \in F_N$.

That is, they are essentially the same automaton up to renaming of states. It is easily argued that isomorphic automata accept the same set.

In this lecture and the next we will show that if M and N are any two automata with no inaccessible states accepting the same set, then the quotient automata $M/\approx$ and $N/\approx$ obtained by the collapsing algorithm of Lecture 14 are isomorphic. Thus the DFA obtained by the collapsing algorithm is the minimal DFA for the set it accepts, and this automaton is unique up to isomorphism.

We will do this by exploiting a profound and beautiful correspondence between finite automata with input alphabet Σ and certain equivalence

relations on Σ^*. We will show that the unique minimal DFA for a regular set R can be defined in a natural way *directly from* R, and that any minimal automaton for R is isomorphic to this automaton.

Myhill–Nerode Relations

Let $R \subseteq \Sigma^*$ be a regular set, and let $M = (Q, \Sigma, \delta, s, F)$ be a DFA for R with no inaccessible states. The automaton M induces an equivalence relation $\equiv_M$ on Σ^* defined by

$$x \equiv_M y \stackrel{\text{def}}{\iff} \widehat{\delta}(s,x) = \widehat{\delta}(s,y).$$

(Don't confuse this relation with the collapsing relation $\approx$ of Lecture 13—that relation was defined on Q, whereas $\equiv_M$ is defined on Σ^*.)

One can easily show that the relation $\equiv_M$ is an equivalence relation; that is, that it is reflexive, symmetric, and transitive. In addition, $\equiv_M$ satisfies a few other useful properties:

(i) It is a *right congruence*: for any $x, y \in \Sigma^*$ and $a \in \Sigma$,

$$x \equiv_M y \Rightarrow xa \equiv_M ya.$$

To see this, assume that $x \equiv_M y$. Then

$$\begin{aligned} \widehat{\delta}(s,xa) &= \delta(\widehat{\delta}(s,x),a) \\ &= \delta(\widehat{\delta}(s,y),a) \quad \text{by assumption} \\ &= \widehat{\delta}(s,ya). \end{aligned}$$

(ii) It *refines* R: for any $x, y \in \Sigma^*$,

$$x \equiv_M y \Rightarrow (x \in R \iff y \in R).$$

This is because $\widehat{\delta}(s,x) = \widehat{\delta}(s,y)$, and this is either an accept or a reject state, so either both x and y are accepted or both are rejected. Another way to say this is that every $\equiv_M$-class has either all its elements in R or none of its elements in R; in other words, R is a union of $\equiv_M$-classes.

(iii) It is of *finite index*; that is, it has only finitely many equivalence classes. This is because there is exactly one equivalence class

$$\{x \in \Sigma^* \mid \widehat{\delta}(s,x) = q\}$$

corresponding to each state q of M.

Let us call an equivalence relation $\equiv$ on Σ^* a *Myhill–Nerode relation for* R if it satisfies properties (i), (ii), and (iii); that is, if it is a right congruence of finite index refining R.

The interesting thing about this definition is that it characterizes exactly the relations on Σ^* that are $\equiv_M$ for some automaton M. In other words, we can reconstruct M from $\equiv_M$ using only the fact that $\equiv_M$ is Myhill–Nerode. To see this, we will show how to construct an automaton $M_\equiv$ for R from any given Myhill–Nerode relation $\equiv$ for R. We will show later that the two constructions

$$M \mapsto \equiv_M,$$
$$\equiv \mapsto M_\equiv$$

are inverses up to isomorphism of automata.

Let $R \subseteq \Sigma^*$, and let $\equiv$ be an arbitrary Myhill–Nerode relation for R. Right now we're not assuming that R is regular, only that the relation $\equiv$ satisfies (i), (ii), and (iii). The $\equiv$-class of the string x is

$$[x] \stackrel{\text{def}}{=} \{y \mid y \equiv x\}.$$

Although there are infinitely many strings, there are only finitely many $\equiv$-classes, by property (iii).

Now define the DFA $M_\equiv = (Q, \Sigma, \delta, s, F)$, where

$$Q \stackrel{\text{def}}{=} \{[x] \mid x \in \Sigma^*\},$$
$$s \stackrel{\text{def}}{=} [\epsilon],$$
$$F \stackrel{\text{def}}{=} \{[x] \mid x \in R\},$$
$$\delta([x], a) \stackrel{\text{def}}{=} [xa].$$

It follows from property (i) of Myhill–Nerode relations that δ is well defined. In other words, we have defined the action of δ on an equivalence class $[x]$ in terms of an element x chosen from that class, and it is conceivable that we could have gotten something different had we chosen another $y \in [x]$ such that $[xa] \neq [ya]$. The property of right congruence says exactly that this cannot happen.

Finally, observe that

$$x \in R \iff [x] \in F. \tag{15.1}$$

The implication ($\Rightarrow$) is from the definition of F, and ($\Leftarrow$) follows from the definition of F and property (ii) of Myhill–Nerode relations.

Now we are ready to prove that $L(M_\equiv) = R$.

Lemma 15.1 $\widehat{\delta}([x], y) = [xy]$.

Proof. Induction on $|y|$.

Basis

$$\widehat{\delta}([x], \epsilon) = [x] = [x\epsilon].$$

Induction step

$$\begin{aligned} \widehat{\delta}([x], ya) &= \delta(\widehat{\delta}([x], y), a) && \text{definition of } \widehat{\delta} \\ &= \delta([xy], a) && \text{induction hypothesis} \\ &= [xya] && \text{definition of } \delta. \end{aligned}$$

□

Theorem 15.2 $L(M_{\equiv}) = R$.

Proof.

$$\begin{aligned} x \in L(M_{\equiv}) &\iff \widehat{\delta}([\epsilon], x) \in F && \text{definition of acceptance} \\ &\iff [x] \in F && \text{Lemma 15.1} \\ &\iff x \in R && \text{property (15.1).} \end{aligned}$$

□

$M \mapsto \equiv_M$ and $\equiv \mapsto M_{\equiv}$ Are Inverses

We have described two natural constructions, one taking a given automaton M for R with no inaccessible states to a corresponding Myhill–Nerode relation $\equiv_M$ for R, and one taking a given Myhill–Nerode relation $\equiv$ for R to a DFA $M_{\equiv}$ for R. We now wish to show that these two operations are inverses up to isomorphism.

Lemma 15.3

(i) If $\equiv$ is a Myhill–Nerode relation for R, and if we apply the construction $\equiv \mapsto M_{\equiv}$ and then apply the construction $M \mapsto \equiv_M$ to the result, the resulting relation $\equiv_{M_{\equiv}}$ is identical to $\equiv$.

(ii) If M is a DFA for R with no inaccessible states, and if we apply the construction $M \mapsto \equiv_M$ and then apply the construction $\equiv \mapsto M_{\equiv}$ to the result, the resulting DFA $M_{\equiv_M}$ is isomorphic to M.

Proof. (i) Let $M_{\equiv} = (Q, \Sigma, \delta, s, F)$ be the automaton constructed from $\equiv$ as described above. Then for any $x, y \in \Sigma^*$,

$$\begin{aligned} x \equiv_{M_{\equiv}} y &\iff \widehat{\delta}(s, x) = \widehat{\delta}(s, y) && \text{definition of } \equiv_{M_{\equiv}} \\ &\iff \widehat{\delta}([\epsilon], x) = \widehat{\delta}([\epsilon], y) && \text{definition of } s \\ &\iff [x] = [y] && \text{Lemma 15.1} \\ &\iff x \equiv y. \end{aligned}$$

(ii) Let $M = (Q, \Sigma, \delta, s, F)$ and let $M_{\equiv_M} = (Q', \Sigma, \delta', s', F')$. Recall from the construction that

$$\begin{aligned} [x] &= \{y \mid y \equiv_M x\} = \{y \mid \widehat{\delta}(s,y) = \widehat{\delta}(s,x)\}, \\ Q' &= \{[x] \mid x \in \Sigma^*\}, \\ s' &= [\epsilon], \\ F' &= \{[x] \mid x \in R\}, \\ \delta'([x],a) &= [xa]. \end{aligned}$$

We will show that $M_{\equiv_M}$ and M are isomorphic under the map

$$\begin{aligned} &f : Q' \to Q, \\ &f([x]) = \widehat{\delta}(s,x). \end{aligned}$$

By the definition of $\equiv_M$, $[x] = [y]$ iff $\widehat{\delta}(s,x) = \widehat{\delta}(s,y)$, so the map f is well defined on $\equiv_M$-classes and is one-to-one. Since M has no inaccessible states, f is onto.

To show that f is an isomorphism of automata, we need to show that f preserves all automata-theoretic structure: the start state, transition function, and final states. That is, we need to show

- $f(s') = s$,
- $f(\delta'([x],a)) = \delta(f([x]),a)$,
- $[x] \in F' \iff f([x]) \in F$.

These are argued as follows:

$$\begin{aligned} f(s') &= f([\epsilon]) && \text{definition of } s' \\ &= \widehat{\delta}(s,\epsilon) && \text{definition of } f \\ &= s && \text{definition of } \widehat{\delta}; \end{aligned}$$

$$\begin{aligned} f(\delta'([x],a)) &= f([xa]) && \text{definition of } \delta' \\ &= \widehat{\delta}(s,xa) && \text{definition of } f \\ &= \delta(\widehat{\delta}(s,x),a) && \text{definition of } \widehat{\delta} \\ &= \delta(f([x]),a) && \text{definition of } f; \end{aligned}$$

$$\begin{aligned} [x] \in F' &\iff x \in R && \text{definition of } F \text{ and property (ii)} \\ &\iff \widehat{\delta}(s,x) \in F && \text{since } L(M) = R \\ &\iff f([x]) \in F && \text{definition of } f. \end{aligned}$$

□

We have shown:

Theorem 15.4 *Let Σ be a finite alphabet. Up to isomorphism of automata, there is a one-to-one correspondence between deterministic finite automata over Σ with no inaccessible states accepting R and Myhill–Nerode relations for R on Σ^*.*