Theory of Computation
Class Notes
Fall 2002

Karl Winklmann
Carolyn J. C. Schauble

Department of Computer Science
University of Colorado at Boulder

September 24, 2002

Preface

What is different

Unlike most textbooks on theory of computation these notes try to build as much as possible on knowledge that juniors or seniors in computer science tend to have. For example, the introduction of undecidability issues in the first chapter starts with examples of programs that read and process other programs that are familiar to the students: Unix utilities like `grep`, `cat`, or `diff`. As another example, the proof of Cook's Theorem (NP-completeness of Boolean satisfiability) [Coo71] is done in terms of "compilation": instead of the usual approach of turning a nondeterministic Turing machine into a Boolean expression, we compile a nondeterministic C++ program into machine language, and the resulting "circuit satisfiability" problem into the problem of determining satisfiability of Boolean expressions.

All material is presented without unnecessary notational clutter. To get to strong results quickly, the text starts with the "machine model" students already know as plain "programs."

When a proof requires a substantial new concept, the concept is first presented in isolation. For example, before we prove that *all* problems in P can be transformed into SAT, we show transformations for a few concrete examples. This allows the students to understand the concept of transformation into Boolean expressions before we present the generic argument that covers a whole class of such transformations. Pedagogically this makes sense even though the examples get superseded quickly by the more general result.

Prerequisites

Students should know the basic concepts from a course on algorithms, including the different types of algorithms (greedy, divide-and-conquer, etc.) and they should have some understanding of algorithm design and analysis and of basic concepts of discrete mathematics such as relations, functions, and Boolean operations. Introductions to such concepts can of course be found in many books, e.g., [Sch96] or [Dea96].

Don't try this at home

These notes are not meant for self-study. The concepts are often too subtle, and these notes are generally too terse for that. They are meant to be read by students in conjunction with classroom discussion.

Karl Winklmann
Carolyn Schauble

Boulder, August 5, 2002

Contents

1 Computability **7**
- 1.1 Strings, Properties, Programs . . . 7
- 1.2 A Proof By Contradiction . . . 11
- 1.3 A Proof of Undecidability . . . 12
- 1.4 Diagonalization . . . 13
- 1.5 Transformations . . . 15
- 1.6 The Undecidability of All I/0-Properties . . . 21
- 1.7 The Unsolvability of the Halting Problem . . . 25
- 1.8 Gödel's Incompleteness Theorem . . . 25

2 Finite-State Machines **31**
- 2.1 State Transition Diagrams . . . 31
- 2.2 The Myhill-Nerode Theorem . . . 41
- 2.3 The Pumping Lemma for Regular Languages . . . 62

3 Context-Free Languages **69**
- 3.1 Context-Free Grammars . . . 69
- 3.2 Parsing . . . 71
- 3.3 The Pumping Lemma for Context-Free Grammars . . . 76
- 3.4 A Characterization of Context-Free Languages . . . 84

4 Time Complexity **89**
- 4.1 The State of Knowledge Before NP-Completeness . . . 89
- 4.2 NP . . . 93
- 4.3 The NP-completeness of SATISFIABILITY . . . 95
- 4.4 More "NP-Complete" Problems . . . 98
- 4.5 The "Holy Grail": P = NP? . . . 101

Bibliography **111**

Chapter 1

Computability

We write programs in order to solve problems. But if a problem is "too hard" and our program is "too simple," this won't work. The sophistication of our program needs to match the intrinsic difficulty of the problem.

Can this be made precise? Theory of computation attempts just that. We will study classes of programs, trying to characterize the classes of problems they solve. The approach is mathematical: we try to prove theorems.

In this first chapter we do not place any restrictions on the programs; later we will restrict the kinds of data structures the programs use.

Maybe surprisingly, even without any restrictions on the programs we will be able to prove in this first chapter that many problems cannot be solved.

The central piece of mathematics in this first chapter is a proof technique called "proof by contradiction." The central piece of computer science is the notion that the input to a program can be a program.

1.1 Strings, Properties, Programs

Programs often operate on other programs. Examples of such "tools" are editors, compilers, interpreters, debuggers, utilities such as — to pick some `Unix` examples — `diff`, `lint`, `rcs`, `grep`, `cat`, `more` and lots of others. Some such programs determine properties of programs they are operating on. For example, a compiler among other things determines whether or not the source code of a program contains any syntax errors. Figure 1.1 shows a much more modest example. The program "`oddlength.cpp`" determines whether or not its input consists of an odd number of characters. If we ran the program in Figure 1.1 on some text file, e.g. by issuing the two commands:

```
CC -o oddlength oddlength.cpp
oddlength < phonelist
```

the response would be one of two words, `Yes` or `No` (assuming that the program

```
void main( void )
{
    if (oddlength())
        cout << "Yes";
    else
        cout << "No";
}

boolean oddlength( void )
{
    boolean toggle = FALSE;

    while (getchar() != EOF)
    {
       toggle = !toggle;
    }
    return( toggle );
}
```

Figure 1.1: A program "`oddlength.cpp`" that determines whether or not its input consists of an odd number of characters

was stored in `oddlength.cpp` and that there was indeed a text file `phonelist` and that this all is taking place on a system on which these commands make sense[1]). We could of course run this program on the source texts of programs, even on its own source text:

```
CC -o oddlength oddlength.cpp
oddlength < oddlength.cpp
```

The response in this case would be `No` because it so happens that the source code of this program consists of an even number of characters.

Mathematically speaking, writing this program was a "constructive" proof of the following theorem. To show that there was a program as claimed by the theorem we constructed one.

Theorem 1 *There is a program that outputs* `Yes` *or* `No` *depending on whether or not its input contains an odd number of characters. (Equivalently, there is a boolean function that returns* `TRUE` *or* `FALSE` *depending on whether or not its input*[2] *contains an odd number of characters.)*

In theory of computation the word "decidable" is used to describe this situation, as in

[1] For our examples of programs we assume a Unix environment and a programming language like C or C++. To avoid clutter we leave out the usual "`#include`" and "`#define`" statements and assume that `TRUE` and `FALSE` have been defined to behave the way you would expect them to.

[2] The "input" to a function is that part of the standard input to the program that has not been read yet at the time the function gets called.

```
void main( void )
{
    char c;

    c = getchar();
    while (c != EOF)
    {
       while (c == 'X'); // loop forever if an 'X' is found
       c = getchar();
    }
}
```

Figure 1.2: A program "`x.cpp`" that loops on some inputs

Theorem 1 (new wording) *Membership in the set* $L_{ODDLENGTH} = \{x : |x| \text{ is odd}\}$ *is decidable.*

Properties vs. sets

The notion of a "property" is the same as the notion of a "set": x having property Π is the same as x being in the set $\{z : z \text{ has property } \Pi\}$. This suggests yet another rewording of Theorem 1 which uses standard terminology of theory of computation. If we use the name ODDLENGTH for the property of consisting of an odd number of characters then the theorem reads as follows.

Theorem 1 (yet another wording) ODDLENGTH *is decidable.*

More interesting properties of programs

Concern about whether or not a program contains an odd number of characters is not usually uppermost on programmers' minds. An example of a property of much greater concern is whether or not a program might not always terminate. Let's say a program has property LOOPING if there exists an input on which the program will not terminate. Two examples of programs with property LOOPING are shown in Figures 1.2 and 1.3. In terms of sets, having property LOOPING means being a member of the set $L_{\text{LOOPING}} = \{x : x \text{ is a syntactically correct program and there is an input } y \text{ such that } x, \text{ when run on } y, \text{ will not terminate}\}$.

Instead of being concerned with whether or not there is some input (among the infinitely many possible inputs) on which a given program loops, let us simplify matters and worry about only one input per program. Let's make that one input the (source code of the) program itself. Let SELFLOOPING be the property that a program will not terminate when run with its own source text as input. In terms of sets, SELFLOOPING means being a member of the set $L_{\text{SELFLOOPING}} = \{x : x \text{ is a syntactically correct program and, when run on } x, \text{ will not terminate}\}$.

The programs in Figures 1.2 and 1.3 both have this property. The program in Figure 1.1 does not.

The bad news about SELFLOOPING is spelled out in the following theorem.

Theorem 2 SELFLOOPING *is not decidable.*

```
void main( void )
{
    while (TRUE);
}
```

Figure 1.3: Another program, "`useless.cpp`," that loops on some (in fact, all) inputs

What this means is that there is no way to write a program which will print `"Yes"` if the input it reads is a program that has the property SELFLOOPING, and will print `"No"` otherwise. Surprising, maybe, but true and not even very hard to prove, as we will do a little later, in Section 1.3, which starts on page 12.

Strings vs. programs The definitions of LOOPING and SELFLOOPING use the notion of a "syntactically correct program." The choice of programming language does not matter for any of our discussions. Our results are not about peculiarities of somebody's favorite programming language, they are about fundamental aspects of any form of computation. In fact, to mention syntax at all is a red herring. We could do all our programming in a language, let's call it C−−, in which *every* string is a program. If the string is a C or C++ program, we compile it with our trusty old C or C++ compiler. If not, we compile it into the same object code as the program "`main () {}`." Then every string is a program and "running x on y" now makes sense for any strings x and y. It means that we store the string x in a file named, say, `x.cmm`, and the string y in a file named, say, `y`, and then we use our new C−− compiler and do

```
CC-- -o x x.cmm
x < y
```

So let's assume from now on that we are programming in "C−−," and thus avoid having to mention "syntactically correct" all the time.

One convenient extension to this language is to regard Unix shell constructs like

```
grep Boulder | sort
```

as programs. This program can be run on some input by

```
cat phonebook | grep Boulder | sort
grep Boulder phonebook | sort
```

or by

```
( grep Boulder | sort ) < phonebook
```

1.2 A Proof By Contradiction

Before we prove our first undecidability result here is an example that illustrates the technique of proving something "by contradiction."

Proving a theorem by contradiction starts with the assumption that the theorem is not true. From this assumption we derive something that we know is not true. The conclusion then is that the assumption must have been false, the theorem true. The "something" that we know is not true often takes the form of "*A and not A*."

Theorem 3 $\sqrt{2}$ *is irrational.*

(A rational number is one that is equal to the quotient of two integers. An irrational number is one that is not rational.)

Proof (Pythagoreans, 5th century B.C.) [3] For the sake of deriving a contradiction, assume that $\sqrt{2}$ is rational.

Then

$$\sqrt{2} = \frac{p}{q} \tag{1.1}$$

for two integers p and q which have no factors in common: they are "relatively prime." Squaring both sides of (1.1) yields

$$2 = \frac{p^2}{q^2} \tag{1.2}$$

and therefore

$$2 \times q^2 = p^2 \tag{1.3}$$

This shows that p^2 contains a factor of 2. But p^2 cannot have a factor of 2 unless p did. Therefore p itself must contain a factor of 2, which gives p^2 at least two factors of 2:

$$p^2 = 2 \times 2 \times r \tag{1.4}$$

for some integer r. Combining (1.3) and (1.4) yields

$$2 \times q^2 = 2 \times 2 \times r \tag{1.5}$$

which simplifies to

$$q^2 = 2 \times r \tag{1.6}$$

which shows that q also contains a factor of 2, contradicting the fact that p and q were relatively prime. □

Next we apply this proof technique to get our first undecidability result.

[3] according to Kline, *Mathematical Thought from Ancient to Modern Times*, New York, 1972, pp.32-33

```
void main( void )
{
    if (selflooping())
        ;                    // stop, don't loop
    else
        while (TRUE);        // do loop forever
}

boolean selflooping( void )  // returns TRUE if the input
{                            // is a program which, when
                             // run with its own source code
    ...                      // as input, does not terminate;
                             // FALSE otherwise
}
```

Figure 1.4: A program "`diagonal.cpp`" that would exist if SELFLOOPING were decidable

1.3 A Proof of Undecidability

Proof of Theorem 2 For the sake of deriving a contradiction, assume that there is a programs that decides if its input has property SELFLOOPING. If such a program existed we could re-write it as a function `boolean selflooping()` that decides whether or not its input has property SELFLOOPING. We could then write the program `diagonal`[4] shown in Figure 1.4.

Does the program in Figure 1.4 itself have property SELFLOOPING?

Claim 1 It does not.

Proof of Claim 1 Assume it did. Then, when running the program with its own source code as input, the function `selflooping` would return `TRUE` to the main program, making it terminate and hence not have property SELFLOOPING, a contradiction. □

Claim 2 It does.

Proof of Claim 2 Assume it did not. Then, when running the program with its own source code as input, the function `selflooping` would return `FALSE` to the main program, sending it into an infinite loop and hence have property SELFLOOPING, a contradiction. □

These two claims form a contradiction, proving our initial assumption wrong; there cannot be a program that decides if its input has property SELFLOOPING. □

[4] Why `diagonal` is a fitting name for this program will become clear in the next section.

	digits →											
numbers ↓				$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{10^3}$	$\frac{1}{10^4}$	$\frac{1}{10^5}$	$\frac{1}{10^6}$	$\frac{1}{10^7}$	$\frac{1}{10^8}$	$\cdots$
d	$=$	0	.	1	1	9	1	1	9	9	9	$\cdots$
r_1	$=$	0	.	**6**	2	8	4	1	1	0	4	$\cdots$
r_2	$=$	0	.	0	**8**	8	4	1	1	3	8	$\cdots$
r_3	$=$	0	.	8	4	**1**	1	0	8	4	1	$\cdots$
r_4	$=$	0	.	1	0	0	**0**	0	1	0	0	$\cdots$
r_5	$=$	0	.	0	9	9	9	**9**	0	9	9	$\cdots$
r_6	$=$	0	.	0	4	6	2	6	**1**	4	6	$\cdots$
r_7	$=$	0	.	1	1	0	0	2	1	**1**	0	$\cdots$
r_8	$=$	0	.	4	4	9	4	3	4	4	**1**	$\cdots$
$\vdots$												

Figure 1.5: The diagonal in the proof of Theorem 4

1.4 Diagonalization

The proof of Theorem 2 was a special kind of proof by contradiction called a "diagonalization proof." To understand that aspect of the proof, let's look at a classical theorem with a diagonalization proof.

Theorem 4 (Cantor, 1873) *The real numbers are not countable.*

(A set is countable if there is a function from the natural numbers onto the set. In other words, there is a sequence $r_1, r_2, r_3, r_4, r_5, \ldots$ that contains all members of the set.)

Proof (Cantor, 1890/1891) [5] Assume the real numbers are countable.

Then there is a sequence that contains all reals and a subsequence r_1, r_2, $r_3, \ldots$ that contains all reals from the interval $(0, 1]$. Consider the infinite matrix that contains in row i a decimal expansion of r_i, one digit per column. (See Figure 1.5 for an illustration.) Consider the number d whose i^{th} digit is 9 if the matrix entry in row i and column i is 1, and whose i^{th} digit is 1 otherwise.

The number d thus defined differs from every number in the sequence r_1, r_2, $r_3, \ldots$, contradicting the assumption that all reals from $[0, 1)$ were in the sequence.[6] □

[5] Cantor gave a different first proof in 1873, according to Kline, p.997.

[6] There is a subtlety in this proof stemming from the fact that two different decimal expansions can represent the same number: $0.10000\ldots = 0.09999\ldots$. Why is this of concern? And why is the proof ok anyway?

programs ↓		p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	$\cdots$
`diagonal`	:	1	1	0	1	1	0	0	0	$\cdots$
p_1	:	**0**	0	0	0	1	1	0	0	$\cdots$
p_2	:	0	**0**	0	0	1	1	0	0	$\cdots$
p_3	:	0	0	**1**	1	0	0	0	1	$\cdots$
p_4	:	1	0	0	**0**	0	1	0	0	$\cdots$
p_5	:	0	0	0	0	**0**	0	0	0	$\cdots$
p_6	:	0	0	0	0	0	**1**	0	0	$\cdots$
p_7	:	1	1	0	0	0	1	**1**	0	$\cdots$
p_8	:	0	0	0	0	0	0	0	**1**	$\cdots$
$\vdots$										

Figure 1.6: The diagonal in the proof of Theorem 2

How does this proof of Cantor's Theorem relate to our proof of Theorem 2? Programs are finite strings and hence countable. Let $p_1, p_2, p_3, \ldots$ be a sequence of all programs. (Unlike a sequence of all reals, such a sequence does exist since programs are finite strings.) Consider the infinite boolean matrix which has a 0 in row i and column j if program i, when run with the source code of program j as input, does not terminate; and a 1 if it does. (See Figure 1.6 for an illustration.)

The program `diagonal.cpp` of Figure 1.4, which we assumed to exist at the start of the proof of Theorem 2, then differs from each program in the sequence, contradicting the fact that all programs were in the sequence. The reason is that if program p_i does terminate on input p_i, then `diagonal.cpp` does not terminate on input p_i, and vice versa.

Other proof techniques? It is remarkable that without diagonalization, nobody would know how to prove Theorem 2 nor any of the other undecidability results in this chapter.

This monopoly of one proof technique will cause problems in the study of computational complexity in Chapter 4, which injects a concern about efficiency into the discussion. Diagonalization does not work nearly as well in that context. Without any other technique available, we will not be able to prove some very basic conjectures.

Heuristics Every computational problem, whether unsolvable in theory or not, is open to a heuristic approach. The theory presented here does not address the feasibility of heuristic methods. It does suggest that heuristics would be the way to go if one wanted to create software tools to solve in some practically useful sense any of the problems which this theory shows "unsolvable."

Self-reference in other domains

Digressing for a moment from our focus on what programs can and cannot do, programs that process programs are not the only example of self-referencing causing trouble, and far from the oldest one. Here are a few others.

An example known to Greek philosophers over 2000 years ago is the self-referencing statement that says "This statement is a lie." This may look like a pretty silly statement to consider, but it is true that the variant that says "This is a theorem for which there is no proof," underlies what many regard as the most important theorem of 20^{th}-century mathematics, Gödel's Incompleteness Theorem. More on that in Section 1.8 on page 25.

A (seemingly) non-mathematical example arises when you try to compile a "Catalog of all catalogs," which is not a problem yet, but what about a "Catalog of all those catalogs that do *not* contain a reference to themselves?" Does it contain a reference to itself? If we call catalogs that contain a reference to themselves "self-promoting," the trouble stems from trying to compile a "catalog of all non-self-promoting catalogs." Note the similarity to considering a program that halts on all those programs that are "selflooping."

1.5 Transformations

SELFLOOPING is not the only undecidable property of programs. To prove other properties undecidable, we will exploit the observation that they have strong connections to SELFLOOPING, connections such that the undecidability of SELFLOOPING "rubs off" on them. Specifically, we will show that there are "transformations" of SELFLOOPING into other problems.

Before we apply the concept of transformations to our study of (un)decidability, let's look at some examples of transformations in other settings.

Consider the Unix command

```
sort
```

It can be used like

```
sort < phonebook
```

to sort the lines of a text file. "`sort`" solves a certain problem, called SORTING, a problem that is so common and important that whole chapters of books have been written about it (and a solution comes with every computer system you might want to buy).

But what if the problem you want to solve is not SORTING, but the problem of SORTING-THOSE-LINES-THAT-CONTAIN-THE-WORD-"Boulder", STLTCTWB for short? You could write a program "`stltctwb`" to solve STLTCTWB and use it like

```
stltctwb < phonebook
```

But probably you would rather do

```
cat phonebook | grep Boulder | sort
```

This is an example of a transformation. The command "`grep Boulder`" *transforms* the STLTCTWB problem *into* the SORTING problem. The command "`grep Boulder | sort`" solves the STLTCTWB problem. (The "`cat phonebook | `" merely provides the input.)

The general pattern is that if "`b`" is a program that solves problem B, and

```
a_to_b | b
```

solves problem A, then

```
a_to_b
```

is called a *transformation of problem A into problem B*.

The reason for doing such transformations is that they are a way to solve problems. If we had to formulate a theorem about the matter it would read as follows. (Let's make it an "Observation." That way we don't have to spell out a proof.)

Observation 1 *If there is a program that transforms problem A into problem B and there is a program that solves problem B, then there is a program that solves problem A.*

Since at the moment we are more interested in proving that things cannot be done than in proving that things can be done, the following rephrasing of the above observation will be more useful to us.

A problem is *solvable* if there exists a program for solving it.

Observation 2 *If there is a program that transforms problem A into problem B and problem A is unsolvable, then problem B is unsolvable as well.*

"Solvable" or "decidable?"

In good English, "problems" get "solved" and "questions" get "decided." Following this usage, we speak of "solvable and unsolvable problems" and "decidable and undecidable questions." For the substance of our discussion this distinction is of no consequence. For example, *solving* the SELFLOOPING *problem* for a program x is the same as *deciding* whether or programs have the *property* SELFLOOPING. In either case, the prefix "un-" means that there is no program to do the job.

Transforming binary addition into Boolean satisfiability (SAT)

For another example of a transformation, consider the following two problems.

Let BINARY ADDITION be the problem of deciding, given a string like

$$1\,0\,+\,1\,1\,=\,0\,1\,1 \tag{1.7}$$

whether or not it is a correct equation between binary numbers. (This one isn't.) Let's restrict the problem to strings of the form

$$x_2\ x_1\ +\ y_2\ y_1\ \ =\ \ z_3\ z_2\ z_1 \tag{1.8}$$

with each x_i, y_i, and z_i being 0 or 1. Let's call this restricted version 2-ADD.

Let Boolean Satisfiability (SAT) be the problem of deciding, given a Boolean expression, whether or not there is an assignment of truth values to the variables in the expression which makes the whole expression *true*.

A *transformation of* 2-ADD *to* SAT is a function t such that for all strings α,

$$\begin{array}{c}\alpha \text{ represents a correct addition of two binary numbers}\\ \Leftrightarrow\\ t(\alpha) \text{ is a satisfiable Boolean expression.}\end{array}$$

How can we construct such a transformation? Consider the Boolean circuit in Figure 1.7. It adds up two 2-bit binary numbers. Instead of drawing the picture we can describe this circuit equally precisely by the Boolean expression

$$\begin{array}{lllllll}
((\ & x_1 & \oplus & y_1 &)\Leftrightarrow & z_1 &)\ \wedge\\
((\ & x_1 & \wedge & y_1 &)\Leftrightarrow & c_2 &)\ \wedge\\
((\ & x_2 & \oplus & y_2 &)\Leftrightarrow & z_2' &)\ \wedge\\
((\ & x_2 & \wedge & y_2 &)\Leftrightarrow & c_3' &)\ \wedge\\
((\ & c_2 & \oplus & z_2' &)\Leftrightarrow & z_2 &)\ \wedge\\
((\ & c_2 & \wedge & z_2' &)\Leftrightarrow & c_3'' &)\ \wedge\\
((\ & c_3' & \vee & c_3'' &)\Leftrightarrow & z_3 &)
\end{array}$$

This suggests transformation of 2-ADD to SAT that maps the string (1.7) to the expression

$$\begin{array}{lllllll}
((\ & 0 & \oplus & 1 &)\Leftrightarrow & 1 &)\ \wedge\\
((\ & 0 & \wedge & 1 &)\Leftrightarrow & c_2 &)\ \wedge\\
((\ & 1 & \oplus & 1 &)\Leftrightarrow & z_2' &)\ \wedge\\
((\ & 1 & \wedge & 1 &)\Leftrightarrow & c_3' &)\ \wedge\\
((\ & c_2 & \oplus & z_2' &)\Leftrightarrow & 1 &)\ \wedge\\
((\ & c_2 & \wedge & z_2' &)\Leftrightarrow & c_3'' &)\ \wedge\\
((\ & c_3' & \vee & c_3'' &)\Leftrightarrow & 0 &)
\end{array}$$

which, appropriately, is not satisfiable. There are four variables to choose values for, c_2, z_2', c_3', and c_3'', but no choice of values for them will make the whole expression true. Note that every single one of the seven "conjuncts" can be made true with some assignment of truth values to c_2, z_2', c_3', and c_3'', but not all seven can be made true with one and the same assignment. In terms of the circuit this means that if the inputs and outputs are fixed to reflect the values given in (1.7), at least one gate in the circuit has to end up with inconsistent values on its input and output wires, or at least one wire has to have different values at its two ends. The circuit is "not satisfiable."

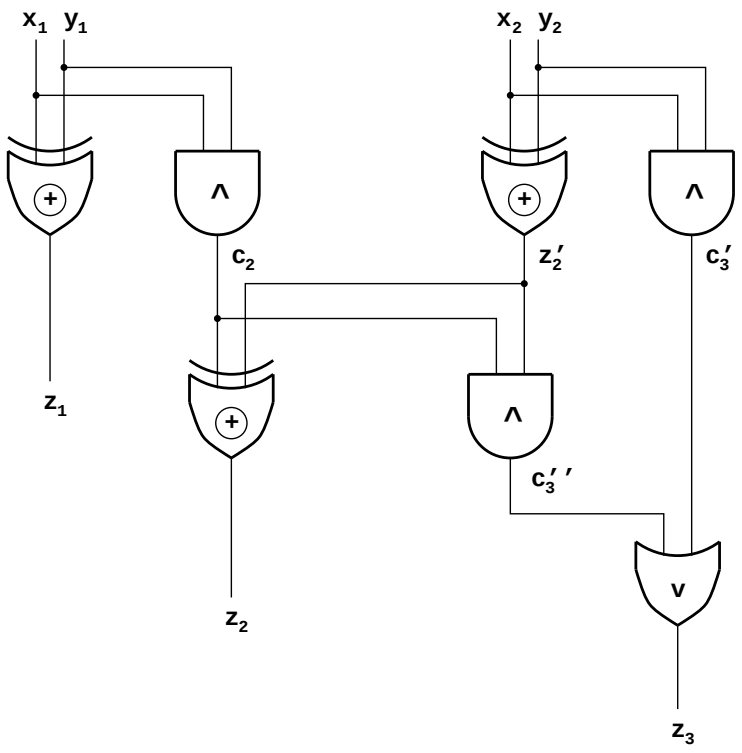


Figure 1.7: A circuit for adding two 2-bit binary numbers

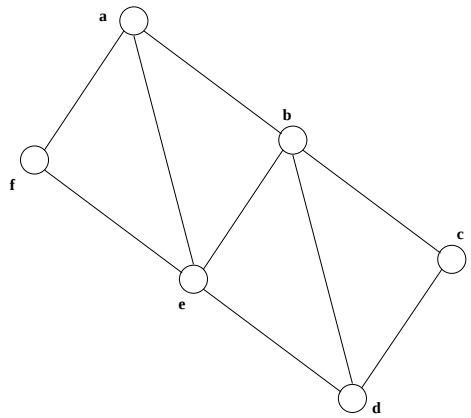


Figure 1.8: A graph of activities and conflicts

Note that transformations do not always go from a harder problem to an easier one.[7] In fact, the opposite is closer to the truth. The destination problem can be easier than the source problem only to the extent of the computational effort expended in carrying out the transformation. But the destination problem can be much harder than the source problem. For example, you can transform ODDLENGTH to LOOPING, but not LOOPING to ODDLENGTH.

Transforming GRAPH 3-COLORABILITY **into** SAT

Graphs like the one shown in Figure 1.8 can be used to represent scheduling problems. Each node represents an activity, e.g. a meeting of some group, and an edge between two nodes means that the two activities must not be scheduled at the same time, e.g. because there is a person who needs to attend both.

Since activities a, b, and e are all conflicting with each other, there is no way to schedule all the activities into fewer than three time slots. Are three slots enough? Instead of activities being scheduled into time slots, graph theorists talk about "nodes" being "colored." The question then becomes, are three different colors enough to color all the nodes, without adjacent nodes getting the same color? This problem is known as "GRAPH 3-COLORABILITY," or "3-COLOR" for short.

The idea behind the transformation is to make the variable a_{GREEN} (a_{RED}, a_{BLUE}) true if and only if the vertex a gets colored green (red, blue). The statement that a and b get different colors can then be made in the "language of Boolean expressions" as

$$\overline{(a_{\text{GREEN}} \wedge b_{\text{GREEN}})} \wedge \overline{(a_{\text{RED}} \wedge b_{\text{RED}})} \wedge \overline{(a_{\text{BLUE}} \wedge b_{\text{BLUE}})} \tag{1.9}$$

We need to make one such statement for each of the nine edges in the graph. The

[7] "Harder" and "easier" mean "needing more and less of a computational resource." Most often the resource we care about is computing time. "Harder" and "easier" do not mean harder or easier to program. Programming effort is not addressed by this theory.

conjunction of these nine statements does not yet complete the transformation because it could always be satisfied by making all the variables *false*, a choice that corresponds to avoiding color conflicts by not coloring the nodes at all. We can complete the transformation by insisting that each node get exactly one color assigned to it. For node a this can be done by the expression

$$(a_{\text{GREEN}} \wedge \overline{a_{\text{RED}}} \wedge \overline{a_{\text{BLUE}}}) \vee (\overline{a_{\text{GREEN}}} \wedge a_{\text{RED}} \wedge \overline{a_{\text{BLUE}}}) \vee (\overline{a_{\text{GREEN}}} \wedge \overline{a_{\text{RED}}} \wedge a_{\text{BLUE}}) \tag{1.10}$$

In general, to carry out the transformation for any graph $G = G(V, E)$, we just have to make a statement like (1.10) for every node of the graph and a statement like (1.9) for every edge, i.e., we have to write this expression E_G:

$$\bigwedge_{x \in V} ((x_{\text{GREEN}} \wedge \overline{x_{\text{RED}}} \wedge \overline{x_{\text{BLUE}}}) \vee (\overline{x_{\text{GREEN}}} \wedge x_{\text{RED}} \wedge \overline{x_{\text{BLUE}}}) \vee (\overline{x_{\text{GREEN}}} \wedge \overline{x_{\text{RED}}} \wedge x_{\text{BLUE}}))$$

$$\bigwedge$$

$$\bigwedge_{(x,y) \in E} (\overline{(x_{\text{GREEN}} \wedge y_{\text{GREEN}})} \wedge \overline{(x_{\text{RED}} \wedge y_{\text{RED}})} \wedge \overline{(x_{\text{BLUE}} \wedge y_{\text{BLUE}})})$$

This expression E_G is satisfiable if and only if the graph G is 3-colorable.

Editing programs To make Observation 2 useful for proving undecidability results, we need to find transformations from one property of programs into another.

Lemma 1 *There is a program* `edit` *which, given as input a program x, creates as output a program y with the following two properties.*

1. *y never reads any input; and*
2. *y, on any input (which it ignores — see previous item), runs like x would on input x.*

Proof We can achieve this transformation "by hand" in a session with our favorite text editor. As an example, if we start with x being the program "`x.cpp`" from Figure 1.2 we could create the program in Figure 1.10. We can even do this in a way where the editing commands do not depend on the program x that is being editing. Therefore we could take a copy of the editor and hard-code those commands into it (or, preferably and equivalently, put the commands into a "script file" or a "macro" for our editor[8]). □

[8] Better yet, we can simplify this editing task greatly by making y the one-line program "`( x < x.cpp )`," which has all the right properties. To see that it ignores its input, do "`( x < x.cpp ) < input`."

```
void main( void )
{
    x_on_x_no_input();
}

void x_on_x_no_input( void )  // runs like x, but
{                             // instead of reading
    ...                       // input it ‘‘reads’’
}                             // from a string constant
```

Figure 1.9: The result of editing program x

1.6 The Undecidability of All I/0-Properties

Recall that LOOPING is the property that there exists an input on which the program will not terminate.

Theorem 5 LOOPING *is undecidable.*

Proof The program edit of Lemma 1 transforms SELFLOOPING into LOOPING. By Observation 2 this proves LOOPING undecidable. □

What other properties of programs are often of interest? The prime consideration usually is whether or not a program "works correctly." The exact meaning of this depends on what you wanted the program to do. If you were working on an early homework assignment in Programming 101, "working correctly" may have meant printing the numbers 1, 2, 3, . . . , 10.

Let ONE TO TEN be the problem of deciding whether or not a program prints the numbers 1, 2, 3, . . . , 10.

Theorem 6 ONE TO TEN *is undecidable.*

Proof Figure 1.11 illustrates a transformation of SELFLOOPING into the negation of property ONE TO TEN.

Note that unlike the function x_on_x_no_input in the previous proof, the function x_on_x_no_input_no_output suppresses whatever output the program x might have generated when run on its own source code as input. This is necessary because otherwise the transformation would not work in the case of programs x that print the numbers 1, 2, 3, . . . , 10.

Since the negation of ONE TO TEN is undecidable, so is the property ONE TO TEN. □

How many more undecidability results can we derive this way?

Figure 1.13 shows an example of an "input-output table" IO_x of a program x. For every input, the table shows the output that the program produces. If the program loops forever on some input and keeps producing output, the entry in the right-hand column is an infinite string.

```
        #define EOF NULL  // NULL is the proper end marker since
                          // we are ``reading'' from a string.

        void main( void )
        {
            x_on_x_no_input();
        }

        char mygetchar( void )
        {
            static int i = 0;
            static char* inputstring =
// start of input string ////////////////////////////////
"\
\n\
        void main( void )\n\
        {\n\
            char c;\n\
\n\
            c = getchar();\n\
            while (c != EOF)\n\
            {\n\
               while (c == 'X'); // loop forever if an 'X' is found\n\
               c = getchar();\n\
            }\n\
        };\n\
";
// end of input string //////////////////////////////////

            return( inputstring[i++] );
        }

        void x_on_x_no_input( void )
        {
            char c;

            c = mygetchar();
            while (c != EOF)
            {
               while (c == 'X'); // loop forever if an 'X' is found
               c = mygetchar();
            }
        }
```

Figure 1.10: The result of editing the program "`x.cpp`" of Figure 1.2

```
void main( void )
{
    x_on_x_no_input_no_output();
    one_to_ten();
}

void x_on_x_no_input_no_output( void )
{                               // runs like x, but
                                // instead of reading
    ...                         // input it ‘‘reads’’
                                // from a string constant,
                                // and it does not
}                               // generate any output

void one_to_ten( void )
{
    int i;
    for (i = 1; i <= 10; i++)
        cout << i << endl;
}
```

Figure 1.11: Illustrating the proof of Theorem 6

```
void main( void )
{
    x_on_x_no_input_no_output();
    notPI();
}

void x_on_x_no_input_no_output( void )
{
    ...
}

void notPI( void )          // runs like a program
{                           // that does not have
    ...                     // property PI
}
```

Figure 1.12: The result of the editing step `rice` on input `x`

Inputs	*Outputs*
λ	`Empty input file.`
0	`Okay.`
1	
00	`Yes.`
01	`Yes. Yes. Yes. ...`
10	`No.`
11	
000	`Yes. No. Yes. No. Yes. No.` $\cdots$
001	`Segmentation fault.`
010	`Core dumped.`
$\vdots$	$\vdots$

Figure 1.13: The "input-output function" of a lousy program

These tables are not anything we would want to store nor do we want to "compute" them in any sense. They merely serve to define what an "input-output property" is. Informally speaking, a property of programs is an input-output property if the information that is necessary to determine whether or not a program x has the property is always present in the program's input-output table IO_x. For example, the property of making a recursive function call on some input, let's call it RECURSIVE, is not an input-output property. Even if you knew the whole input-output table such as the one in Figure 1.13, you would still not know whether the program was recursive. The property of running in linear time is not an input-output property, nor is the property of always producing some output within execution of the first one million statements, nor the property of being a syntactically correct C program. Examples of input-output properties include

- the property of generating the same output for all inputs,
- the property of not looping on any inputs of length less that 1000,
- the property of printing the words `Core dumped` on some input.

More formally, a property Π of programs is an *input-output property* if the following is true:

> For any two programs a and b, if $IO_a = IO_b$ then either both a and b have property Π, or neither does.

A property of programs is *trivial* if either all programs have it, or if none do. Thus a property of programs is *nontrivial* if there is at least one program that has the property and one that doesn't.[9]

[9]There are only two trivial properties of programs. One is the property of being a program

Theorem 7 (Rice's Theorem) *No nontrivial input-output property of programs is decidable.*

Proof Let Π be a nontrivial input-output property of programs. There are two possibilities. Either the program

```
main () { while (TRUE); }
```
(1.11)

has the property Π or it doesn't. Let's first prove the theorem for properties Π which this program (1.11) does have.

Let `notPI` be a function that runs like a program that does not have property Π. Consider the program that is outlined in Figure 1.12. It can be created from x by a program, call it `rice`, much like the program in Figure 1.9 was created from x by the program `edit` of Lemma 1.

The function `rice` transforms SELFLOOPING into Π, which proves Π undecidable.

What if the infinite-loop program in (1.11) does not have property Π? In that case we prove Rice's Theorem for the negation of Π — by the argument in the first part of this proof — and then observe that if the negation of a property is undecidable, so is the property itself. □

1.7 The Unsolvability of the Halting Problem

The following result is probably the best-known undecidability result.

Theorem 8 (Unsolvability of the Halting Problem) *There is no program that meets the following specification. When given two inputs, a program x and a string y, print "Yes" if x, when run on y, terminates, and print "No" otherwise.*

Proof Assume HALTING is decidable. I.e. assume there is a program `halting` which takes two file names, `x` and `y`, as arguments and prints `Yes` if `x`, when run on `y`, halts, and prints `No` otherwise. Then SELFLOOPING can be solved by running `halting` with the two arguments being the same, and then reversing the answer, i.e., changing `Yes` to `No` and vice versa. □

1.8 Gödel's Incompleteness Theorem

Once more digressing a bit from our focus on programs and their power, it is worth pointing out that our proof techniques easily let us prove what is certainly the most famous theorem in mathematical logic and one of the most famous in

— all programs have that property. The other is the property of not being a program — no program has that property. In terms of sets, these are the set of all programs and the empty set.

all of mathematics. The result was published in 1931 by Kurt Gödel and put an end to the notion that if only we could find the right axioms and proof rules, we could prove in principle all true conjectures. Put another way, it put an end to the notion that mathematics could somehow be perfectly "automated."

Theorem 9 (Gödel's Incompleteness Theorem, 1931)[10] *For any system of formal proofs that includes at least the usual axioms about natural numbers, there are theorems that are true but do not have a proof in that system.*

Proof If every true statement had a proof, SELFLOOPING would be decidable, by the following algorithm.

For any given program x the algorithm generates all strings in some systematic order until it finds either a proof that x has property SELFLOOPING or a proof that x does not have property SELFLOOPING. □

It is crucial for the above argument that verification of formal proofs can be automated, i.e., done by a program. It is not the verification of formal proofs that is impossible. What is impossible is the design of an axiom system that is strong enough to provide proofs for all true conjectures.

Does this mean that once we choose a set of axioms and proof rules, our mathematical results will forever be limited to what can be proven from those axioms and rules? Not at all. The following lemma is an example.

Lemma 2 *For any system of formal proofs, the program in Figure 1.14 does have property* SELFLOOPING *but there is no formal proof of that fact within the system.*

Proof The program in Figure 1.14 is a modification of the program we used to prove SELFLOOPING undecidable. Now we prove SELFLOOPING "unprovable." When given its own source as input, the program searches systematically for a proof that it does loop and if it finds one it terminates. Thus the existence of such a proof leads to a contradiction. Since no such proof exists the program keeps looking forever and thus does in fact have property SELFLOOPING. □

Note that there is no contradiction between the fact that there is no proof of this program having property SELFLOOPING *within the given axiom system* and the fact that we just proved that the program has property SELFLOOPING. It is just that our (informal) proof cannot be formalized within the given system.

What if we enlarged the system, for example by adding one new axiom that says that this program has property SELFLOOPING? Then we could prove that fact (with a one-line proof no less). This is correct but note that the program had the axiom system "hard coded" in it. So if we change the system we get a new program and the lemma is again true — for the new program.

[10] Kurt Gödel, Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I, *Monatshefte für Mathematik und Physik,* 38, 1931, 173–98.

```
void main( void )
{
              // systematically generates all proofs
      ...     // in the proof system under consideration
              // and stops if and when it finds a proof
              // that its input has property SELFLOOPING
}
```

Figure 1.14: Illustrating the proof of Lemma 2

Exercises

Ex. 1.1. The rational numbers are countable (as argued in class). Why does the diagonalization proof that shows that the real numbers are not countable not work for the rational numbers? **Diagonalization**

Ex. 1.2. Is it decidable if a program has a "memory leak?"

Ex. 1.3. Is it decidable if a program will try to "de-reference a null pointer?"

Ex. 1.4. Let SELFRECURSIVE be the property of programs that they make a recursive function call when running on their own source code as input. Describe a diagonalization proof that SELFRECURSIVE is not decidable.

Ex. 1.5. Somebody who has not taken this course cannot believe that there is no way to write a program that finds out, given a program p and input x, whether or not p when run on input x goes into an infinite loop. That person hands you a floppy with the source code of a program that they claim does in fact carry out that analysis.

Can you produce a program p and input x on which their program fails?

Ex. 1.6. Describe a transformation from SELFLOOPING to the property of not reading any input. **Transformations**

Ex. 1.7. The function `edit` that is used in the proof of Theorem 5 transforms SELFLOOPING not only to LOOPING but also to other properties. Give three such properties.

Ex. 1.8. Consider the following problem. Given a graph, is there a set of edges so that each node is an endpoint of exactly one of the edges in that set. (Such a set of edges is called a "complete matching".) Describe a transformation of this problem to the satisfiability problem for Boolean expressions. You may do this by showing (and explaining, if it's complicated) the expression for one of the above graphs.

Ex. 1.9. If G is a graph that you are trying to find a 3-coloring for, if $t(G)$ is the Boolean expression the graph gets translated into by our transformation of 3-COLOR into SAT, and, finally, if someone handed you an assignment of truth values to the variables of $t(G)$ that satisfied the expression, describe how you can translate that assignment of truth values back to a 3-coloring of the graph.

Ex. 1.10. Consider trying to transform 3-COLOR into SAT but using, in place of

$$(a_{\text{GREEN}} \wedge \overline{a_{\text{RED}}} \wedge \overline{a_{\text{BLUE}}}) \vee (\overline{a_{\text{GREEN}}} \wedge a_{\text{RED}} \wedge \overline{a_{\text{BLUE}}}) \vee (\overline{a_{\text{GREEN}}} \wedge \overline{a_{\text{RED}}} \wedge a_{\text{BLUE}})$$

the expression

$$x_{\text{GREEN}} \vee x_{\text{RED}} \vee x_{\text{BLUE}} .$$

(a) Would this still be a transformation of 3-COLOR into SAT? Explain why, or why not.

(b) Could we still translate a satisfying assignment of truth values back into a 3-coloring of the graph? Explain how, or why not.

Ex. 1.11. *(a)* Transform SELFLOOPING into the property of halting on those inputs that contain the word `Halt`, and looping on all others. *(b)* What does this prove?

Ex. 1.12. *(a)* Transform ODDLENGTH into SELFLOOPING. *(b)* What does this prove?

I/0-Properties

Ex. 1.13. Give three undecidable properties of programs that are not input-output properties and that were not mentioned in the notes. Give three others that are nontrivial I/O properties, hence undecidable, but were not mentioned in class.

Ex. 1.14. We needed to suppress the output in the `x_on_x_no_input_no_output` function used in the proof of Rice's Theorem. Which line of the proof could otherwise be wrong?

Ex. 1.15. Is SELFLOOPING an input-output property?

More Undecidability Results

Ex. 1.16. Formulate and prove a theorem about the impossibility of program optimization.

Ex. 1.17. "Autograders" are programs that check other programs for correctness. For example, an instructor in a programming course might give a programming assignment and announce that your programs will be graded by a program she wrote.

[10] If having a choice is not acceptable in a real "algorithm," we can always program the algorithm to prefer RED over GREEN over BLUE.

What does theory of computing have to say about this? Choose one of the three answers below and elaborate in no more than two or three short sentences.

1. "Autograding" is not possible.
2. "Autograding" is possible.
3. It depends on the programming assignment.

Ex. 1.18. Which of the following five properties of programs are decidable? Which are IO-properties?

1. Containing comments.
2. Containing only correct comments.
3. Being a correct parser for C (i.e., a program which outputs "Yes" if the input is a syntactically correct C program, and "No" otherwise).
4. Being a correct compiler for C (i.e., a program which, when the input is a C program, outputs an equivalent machine-language version of the input. "Equivalent" means having the same IO table.).
5. The output not depending on the input.

Chapter 2

Finite-State Machines

In contrast to the preceding chapter, where we did not place any restrictions on the kinds of programs we were considering, in this chapter we restrict the programs under consideration very severely. We consider only programs whose storage requirements do not grow with their inputs. This rules out dynamically growing arrays or linked structures, as well as recursion beyond a limited depth.

The central piece of mathematics for this chapter is the notion of an "equivalence relation." It will allow us to arrive at a complete characterization of what these restricted programs can do. Applying mathematical induction will provide us with a second complete characterization.

2.1 State Transition Diagrams

The program `oddlength`, once more, in a picture

Consider the program `oddlength` from Figure 1.1 on page 8. Let's use a debugger on it and set a breakpoint just before the `getchar()` statement. When the program stops for the first time at that breakpoint, we can examine the values of its lone variable. The value of `toggle` is `FALSE` at that time and the "instruction pointer" is of course just in front of that `getchar()` statement. Let's refer to this situation as state A.

If we resume execution and provide an input character, then at the next break the value of `toggle` will be `TRUE` and the instruction pointer will of course again be just in front of that `getchar()`. This is another "state" the program can be in. Let's call it state B.

Let's give the program another input character. The next time our debugger lets us look at things the program is back in state A. As long as we keep feeding it input characters, it bounces back and forth between state A and state B.

What does the program do when an `EOF` happens to come around? From state A the program will output `No`, from state B `Yes`.

Figure 2.1 summarizes all of this, in a picture. The double circle means that an `EOF` produces a `Yes`, the single circle, a `No`. Σ is the set of all possible input

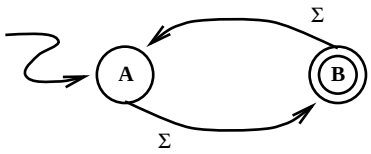


Figure 2.1: The program `oddlength` as a finite-state machine

characters, the *input alphabet*. Labeling a transition with Σ means that any input character will cause that transition to be made.

Terminology A picture such as the one in Figure 2.1 describes a *finite-state machine* or a *finite-state automaton*. States drawn as double circles are *accepting states*, single circles are *rejecting states*. Input characters cause *transitions* between states. Arrows that go from state to state and are labeled with input characters indicate which input characters cause what transitions. For every state and every input character there must be an arrow starting at that state and labeled by the input character. The initial state, or *start state*, is indicated by an unlabeled arrow.[1]

An input string that makes the machine end up in an accepting (rejecting) state is said to be *accepted* (*rejected*) by the machine. If M is a finite-state machine M then $L_M = \{x : x \text{ is accepted by } M\}$ is the *language accepted by* M.

Another Example Let L_{MATCH} be the language of all binary strings that start and end with a 0 or start and end with a 1. Thus $L_{\text{MATCH}} = \{0, 1, 00, 11, 000, 010, 101, 111, 0000, 0010, \ldots\}$ or, more formally,

$$L_{\text{MATCH}} = \{c, cxc : x \in \{0,1\}^\star, c \in \{0,1\}\}. \tag{2.1}$$

Can we construct a finite-state machine for this language? To have a chance to make the right decision between accepting and rejecting a string when its last character has been read, it is necessary to remember what the first character and the last character were. How can a finite-state machine "remember" or "know" this information? The only thing a finite-state machine "knows" at any one time is which state it is in. We need four states for the four combinations of values for those two characters (assuming, again, that we are dealing only with inputs consisting entirely of 0s and 1s). None of those four states would be appropriate for the machine to be in at the start, when no characters have been read yet. A fifth state will serve as start state.

[1] Purists would call the picture the "transition diagram" of the machine, and define the machine itself to be the mathematical structure that the picture represents. This structure consists of a set of states, a function from states and input symbols to states, a start state, and a set of accepting states.

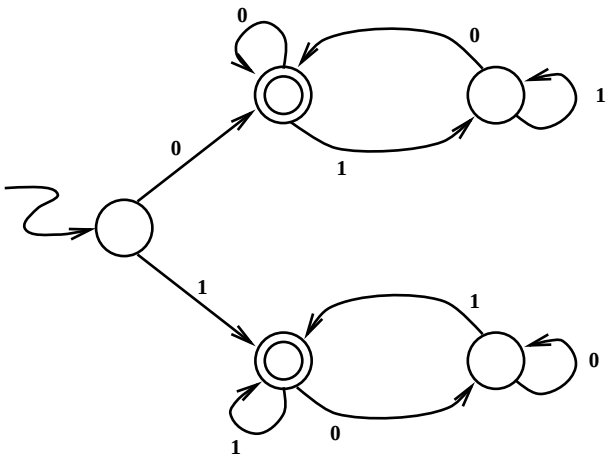


Figure 2.2: A finite-state machine that accepts L_{MATCH}

Now that we know what each state means, putting in the right transitions is straightforward. Figure 2.2 shows the finished product.

(Note that we could have arrived at the same machine by starting with the program `match` in Figure 2.3 and translating it into a finite-state machine by the same process that we used earlier with the program `oddlength`.)

A third example

Consider the language $L_{\text{EQ-4}}$ of all nonempty bitstrings of length 4 or less which contain the same number of 0s as 1s. Thus $L_{\text{EQ-4}} = \{01, 10, 0011, 0101, 0110, 1001, 1010, 1100\}$.

What does a finite-state machine need to remember about its input to handle this language? We could make it remember exactly what the input was, up to length 4. This results in a finite-state machine with the structure of a binary tree of depth 4. See Figure 2.4. Such a binary tree structure corresponds to a "table lookup" in a more conventional language. It works for any finite language.

The preceding example proves

Theorem 10 *Every finite language is accepted by a finite-state machine.*

Minimizing Finite-State Machines

The finite-state machine of Figure 2.4 is not as small as it could be. For example, if we "merged" the states H, P, Q, and FF — just draw a line surrounding them and regard the enclosed area as a new single state —, we would have reduced the size of this machine by three states (without changing the language it accepts).

Why was this merger possible? Because from any of these four states, only rejecting states can be reached. Being in any one of these four states, no matter what the rest of the input, the outcome of the computation is always the same, *viz.* rejecting the input string.

```
void main( void )
{
    if (match())
        cout << "Yes";
    else
        cout << "No";
}

boolean match( void )
{
    char first, next, last;

    first = last = next = getchar();
    while (next != EOF)
    {
       last = next;
       next = getchar();
    }
    return ((first == last) && (first != EOF)):
}
```

Figure 2.3: Another "finite-state" program, `match`

Can J and K be merged? No, because a subsequent input 1 leads to an accepting state from J but a rejecting state from K.

What about A and D? They both have only rejecting states as their immediate successors. This is not a sufficient reason to allow a merger, though. What if the subsequent input is 01? Out of A the string 01 drives the machine into state E — a fact usually written as $\delta(A, 01) = E$ — and out of D the string 01 drives the machine into state $\delta(D, 01) = Q$. One of E and Q is rejecting, the other accepting. This prevents us from merging A and D.

When is a merger of two states p and q feasible? *When it is true that for all input strings s, the two states $\delta(p, s)$ and $\delta(q, s)$ are either both accepting or both rejecting.* (Note that s could be the empty string, which prevents a merger of an accepting with a rejecting state.) This condition can be rephrased to talk about when a merger is *not* feasible, *viz. if there is an input string s such that of the two states $\delta(p, s)$ and $\delta(q, s)$ one is rejecting and one is accepting.* Let's call such a string s "a reason why p and q cannot be merged." For example, the string 011 is a reason why states B and C in Figure 2.4 cannot be merged. The shortest reasons for this are the strings 0 and 1. A reason why states D and E cannot be merged is the string 11. The shortest reason for this is the empty string.

Figure 2.5 is an illustration for the following observation, which is the basis for an algorithm for finding all the states that can be merged.

Observation 3 *If c is a character and s a string, and cs is a shortest reason why p and q cannot be merged, then s is a shortest reason why $\delta(p, c)$ and $\delta(q, c)$*

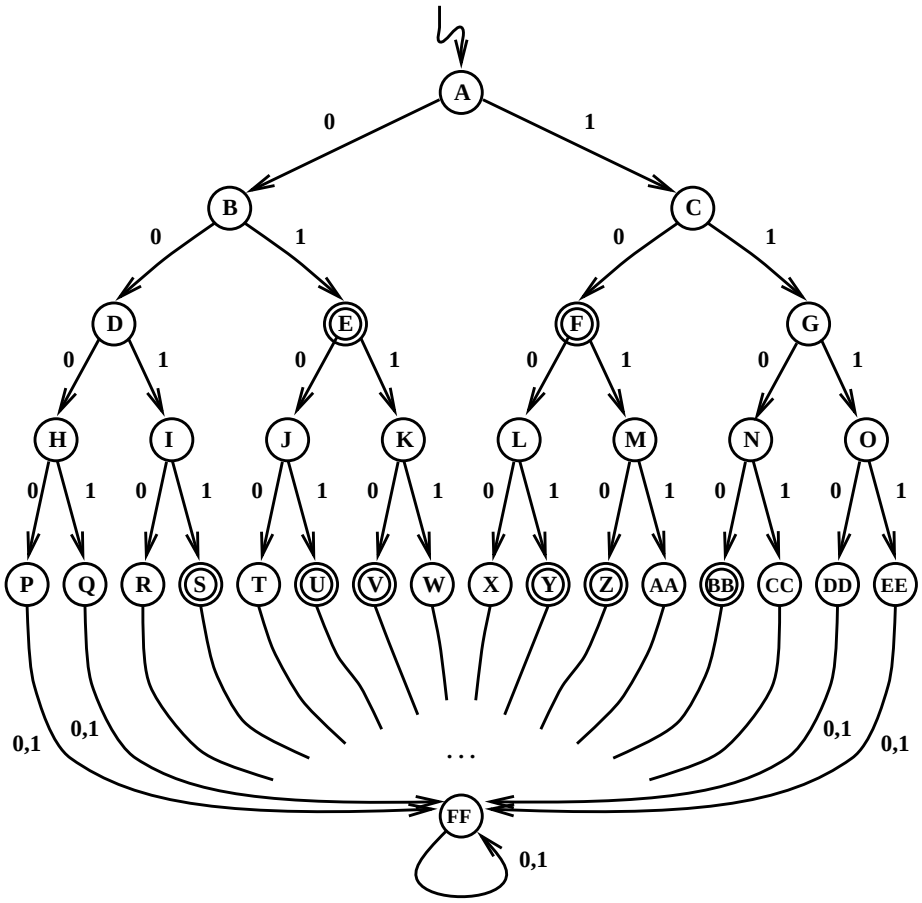


Figure 2.4: A finite-state machine for $L_{\text{EQ-4}}$

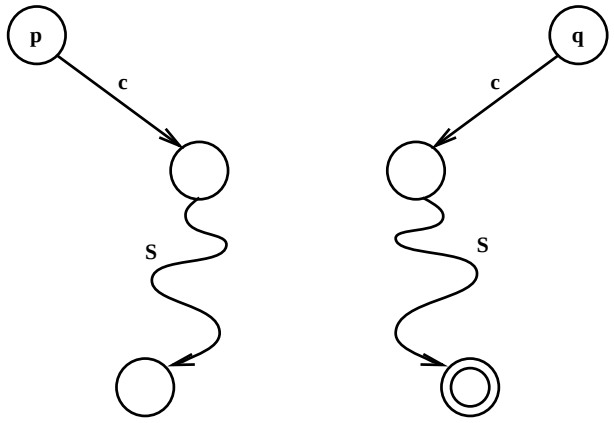


Figure 2.5: Illustrating Observation 3

cannot be merged.

Observation 3 implies that the algorithm is Figure 2.6 is correct.

When carrying out this algorithms by hand, it is convenient to keep track of clusters of states.[2] Two states p and q are in the same cluster if $(p, q) \in$ `MergablePairs`. As we find reasons for not merging states, i.e., as we find the condition of the `if`-statement to be true, we break up the clusters. If p and q are two states that cannot be merged and the shortest reason for that is a string of length $k > 0$, then p and q will remain together through the first $k - 1$ passes through the `repeat`-loop and get separated in the k^{th} pass.

For example, consider the finite-state machine in Figure 2.7. It accepts "identifiers with one subscript" such as `A[2]`, `TEMP[I]`, `LINE1[W3]`, `X034[1001]`. Figure 2.8 shows how the clusters of states develop during execution of the minimization algorithm. States a and b get separated in the third pass, which is consistent with the fact that the shortest strings that are reasons not to merge a and b are of length 3, for example "`[0]`."

Other examples are the finite-state machines in Figures 2.9 and 2.4. On the finite-state machine of Figure 2.9 the algorithm ends after one pass through the loop, resulting in the machine shown in Figure 2.10. The binary-tree-shaped machine of Figure 2.4 takes three passes, resulting in the machine shown in Figure 2.11.

[2]Note: This works because the binary relation MergablePairs is always an equivalence relation.

```
MergablePairs = ( States × States )
                − ( AcceptingStates × RejectingStates )
repeat
    OldMergablePairs = MergablePairs
        for all (p, q) ∈ MergablePairs
            for all input characters c
                if ( (δ(p, c), δ(q, c)) ∉ OldMergablePairs )
                    MergablePairs = MergablePairs − (p, q)
until
    MergablePairs = OldMergablePairs
```

Figure 2.6: An algorithm for finding all mergable states in a finite-state machine

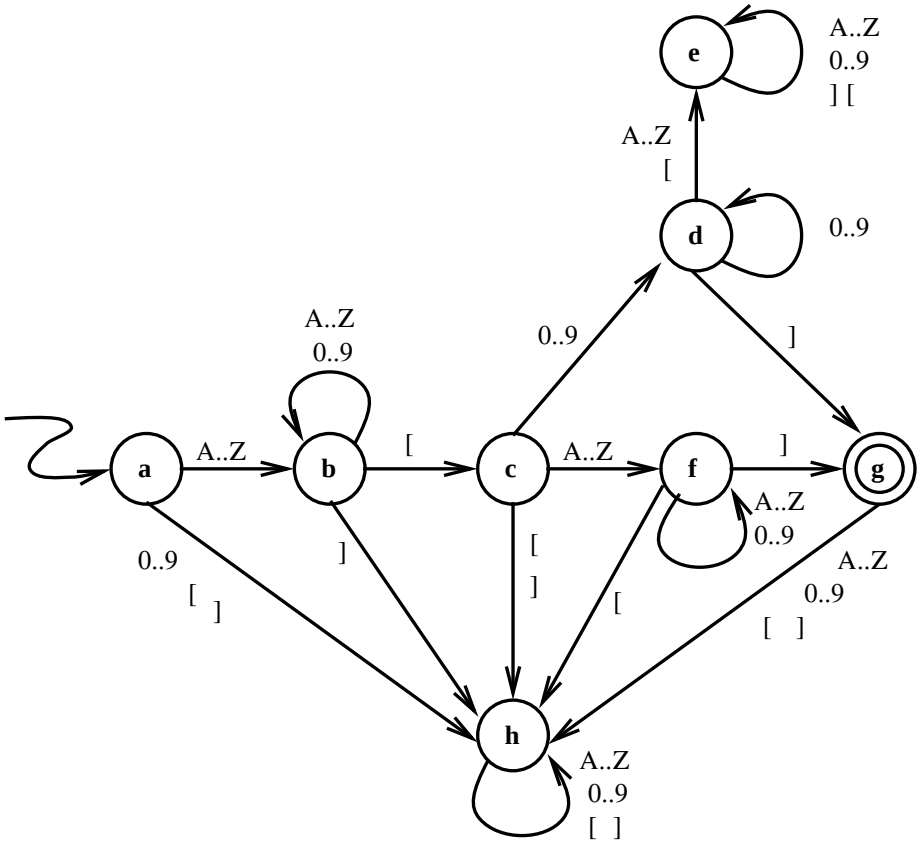


Figure 2.7: A finite-state machine for accepting identifiers with one subscript

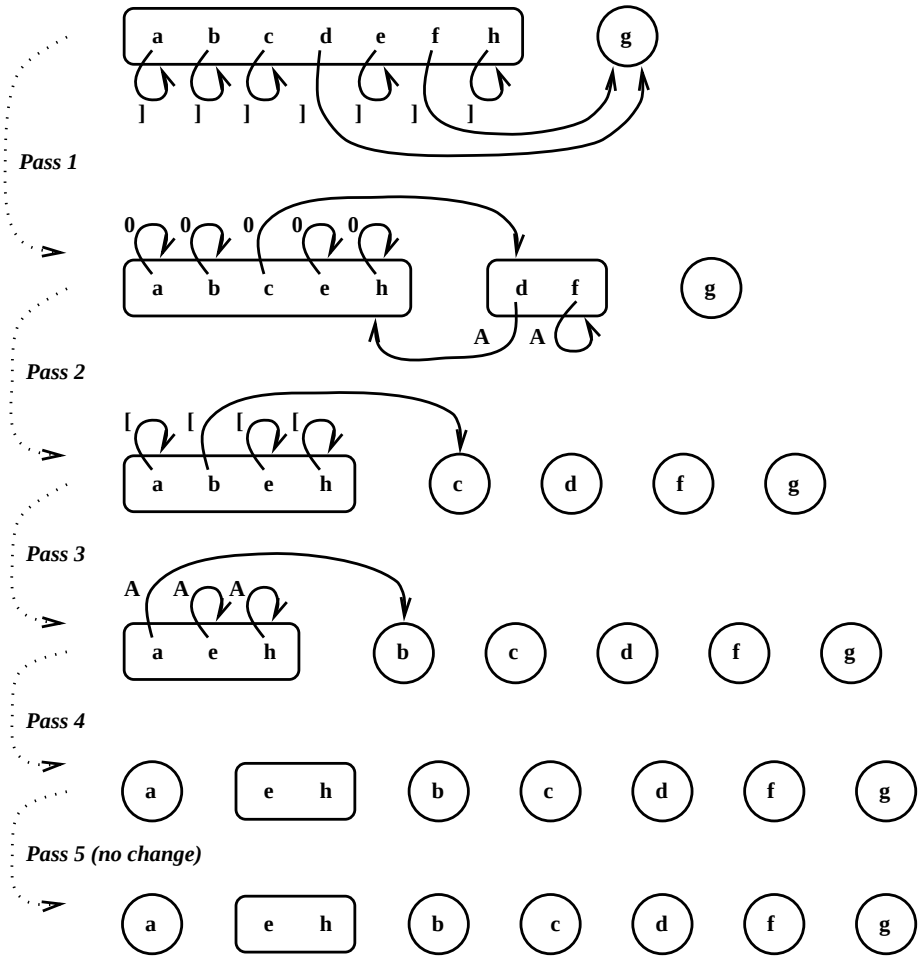


Figure 2.8: Minimizing the finite-state machine of Figure 2.7

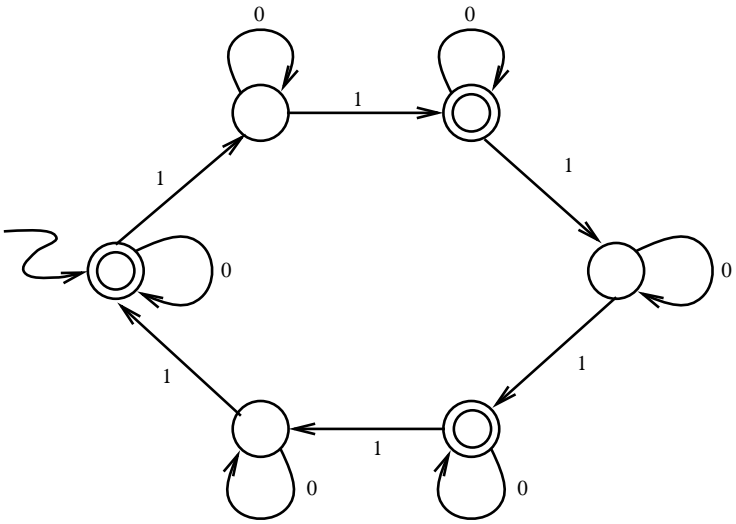


Figure 2.9: A finite-state machine for checking parity

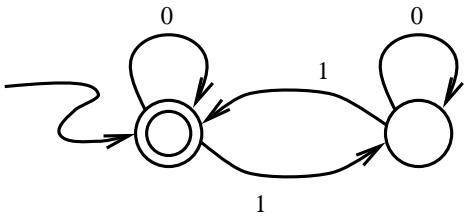


Figure 2.10: A minimal finite-state machine for checking parity

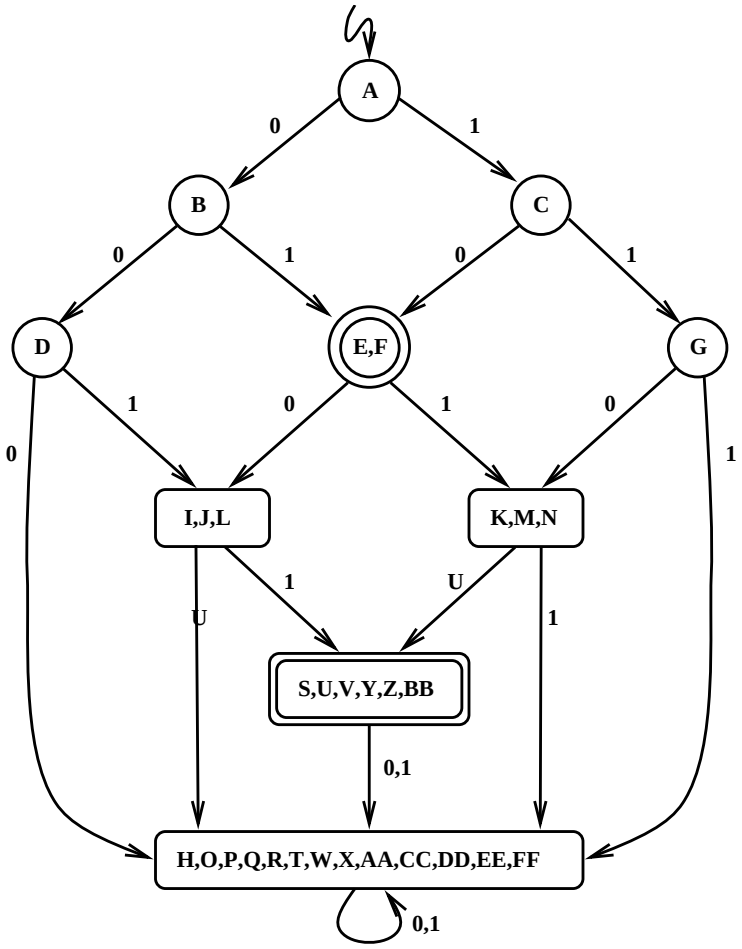


Figure 2.11: A minimal finite-state machine for $L_{\text{EQ-4}}$

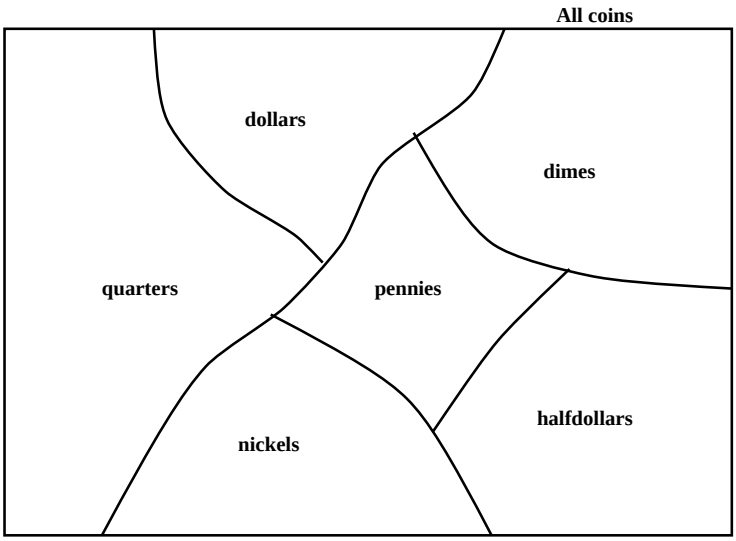


Figure 2.12: Equivalence classes of coins

2.2 The Myhill-Nerode Theorem

Equivalence Relations, Partitions

We start with a brief introduction of equivalence relations and partitions because these are central to the discussion of finite-state machines.

Informally, two objects are equivalent if distinguishing between them is of no interest. At the checkout counter of a supermarket, when you are getting 25 cents back, one quarter looks as good as the next. But the difference between a dime and a quarter is of interest. The set of all quarters is what mathematicians call an "equivalence class." The set of all dimes is another such class, as is the set of all pennies, and so on. The set of all coins gets carved up, or "partitioned," into disjoint subsets, as illustrated in Figure 2.12. Each subset is called an "equivalence class." Collectively, they form a "partition" of the set of all coins: every coin belongs to one and only one subset. There are finitely many such subsets in this example (6 to be precise).

Partitions ≡ Equivalence Relations

Given a partition, there is a unique equivalence relation that goes with it: the relation of belonging to the same subset.

And given an equivalence relation, there is a unique partition that goes with it. Draw the graph of the relation. It consists of disconnected complete subgraphs whose node sets form the sets of the partition.[3]

[3]Note: This is easier to visualize when the set of nodes is finite, but works also for infinite

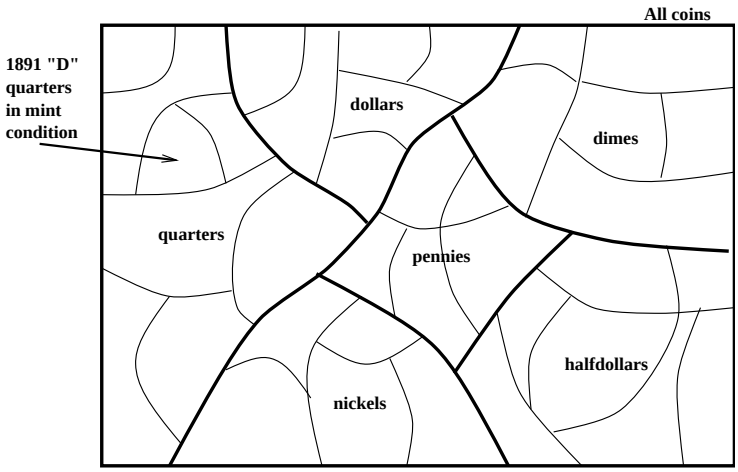


Figure 2.13: A refinement of a partition

Refinements Just exactly what distinctions are of interest may vary. To someone who is collecting rare coins, one quarter is not always going to look as good as the next, but one quarter that was minted in 1891 and has a D on it and is in mint condition looks as good as the next such quarter. Our original equivalence class of all quarters gets subdivided, capturing the finer distinctions that are of interest to a coin collector. The picture of Figure 2.12 gets more boundary lines added to it. See Figure 2.13.

Equivalence Relations Among Programs What differences between program do we normally ignore? It depends on what we are interested in doing with the programs. If all we want to do is put a printout of their source under the short leg of a wobbly table, programs with equally many pages of source code are equivalent.

A very natural notion of equivalence between programs came up in Chapter 1: $IO_a = IO_b$. A further refinement could take running time into account: $IO_a = IO_b$ and $t_a = t_b$ ("t" for time). A further refinement could in addition include memory usage: $IO_a = IO_b$ and $t_a = t_b$ and $s_a = s_b$ ("s" for space). (Both t and s are functions of the input.) The refinements need not stop here. We might insist that the two programs compile to the same object code. If all we want to do with the program is run it, we won't care about further refinements. What else would we want to do with a program but run it? We might want to modify it. Then two programs, one with comments and one without, are not going to be equivalent to us even if they are indistinguishable at run time. So there is a sequence of half a dozen or so successive refinements of notions of

and even uncountably infinite sets.

program equivalence, each of which is the right one for some circumstances.

Next we will apply the concept of partitions and equivalence relations to finite-state machines.

A Game

Let PARITY be the set consisting of all binary strings with an even number of 1s. For example, 10100011 ∈ PARITY, 100 ∉ PARITY. Consider the following game, played by Bob and Alice.

1. Bob writes down a string x without showing it to Alice.
2. Alice asks a question Q about the string x. The question must be such that there are only finitely many different possible answers. E.g., "What is x?" is out, but "What are the last three digits of x?" is allowed.
3. Bob gives a truthful answer $A_Q(x)$ and writes down a string y for Alice to see. (Alice still has not seen x.)
4. If Alice at this point has sufficient information to know whether or not $xy \in$ PARITY, she wins. If not, she loses.

Can Alice be sure to win this game? For example, if her question was *"What is the last bit of x?"* she might lose the game because an answer *"1"* and a subsequent choice of $y = 1$ would not allow her to know if $xy \in$ PARITY since x might have been 1 or 11. What would be a better question?

Alice asks *"Is the number of 1s in x even?"*. She is sure to win.

Tired of losing Bob suggests they play the same game with a different language: MORE-0S-THAN-1S, the set of bitstrings that contain more 0s than 1s.

Alice asks *"How many more 0s than 1s does x have?"*. Bob points out that the question is against the rules. Alice instead asks *"Does x have more 0s than 1s?"*. Bob answers *"Yes."*. Alice realizes that x could be any one of 0, 00, 000, 0000, ... and concedes defeat.

Theorem 11 *Alice can't win the* MORE-0S-THAN-1S*-game.*

Proof Whatever finite-answer question Q Alice asks, the answer will be the same for some different strings among the infinite collection {0, 00, 000, 0000, ...}. Let 0^i[4] and 0^j be two strings such that $0 \leq i < j$ and $A_Q(0^i) = A_Q(0^j)$. The Bob can write down $x = 0^i$ or $x = 0^j$, answer Alice's question, and then write $y = 1^i$. Since $0^i 1^i \notin$ MORE-0S-THAN-1S and $0^j 1^i \in$ MORE-0S-THAN-1S, Alice cannot know whether or not $x1^i \in$ MORE-0S-THAN-1S — she loses. □

What does this game have to do with finite-state machines? It provides a complete characterization of the languages that are accepted by finite-state machines ...

[4]Note: 0^i means a string consisting of i 0s.

Theorem 12 *Consider Bob and Alice playing their game with a language L.*[5] *The following two statements are equivalent.*

1. *Alice can be sure to win.*
2. *L is accepted by some finite-state machine.*

In fact, the connection between the game and finite-state machines is even stronger.

Theorem 13 *Consider Bob and Alice playing their game with the language L. The following two statements are equivalent.*

1. *There is a question that lets Alice win and that can always be answered with one of k different answers.*
2. *There is a finite-state machine with k states that accepts L.*

To see that *(2)* implies *(1)* is the easier of the two parts of the proof. Alice draws a picture of a k-state machine for L and asks which state the machine is in after reading x. When Bob writes down y, Alice runs M to see whether or not it ends up in an accepting state.

The key to proving that *(1)* implies *(2)* is to understand why some questions that Alice might ask will make her win the game while others won't.

The Relation $\equiv_L$ Why was it that in the PARITY game asking whether the last bit of x was a 1 did not ensure a win? Because there were two different choices for x, $x_1 = 1$ and $x_2 = 11$, which draw the same answer, but for which there exists a string y with $(x_1y \in \text{PARITY}) \not\Leftrightarrow (x_2y \in \text{PARITY})$. In this sense the two strings $x_1 = 1$ and $x_2 = 11$ were "not equivalent" and Alice's question made her lose because it failed to distinguish between two nonequivalent strings. Such nonequivalence of two strings is usually written as $x_1 \not\equiv_{\text{PARITY}} x_2$.

Conversely, two strings x_1 and x_2 are equivalent, $x_1 \equiv_{\text{PARITY}} x_2$, if for all strings y, $(x_1y \in \text{PARITY}) \Leftrightarrow (x_2y \in \text{PARITY})$.

Definition 1 *Let L be any language. Two strings x and x' are* equivalent with respect to L, *$x \equiv_L x'$, if for all strings y, $(xy \in L) \Leftrightarrow (x'y \in L)$.*

The Relation $\equiv_Q$ Consider the question Q that made Alice win the DIVISIBLE-BY-8-game: *"What are the last three digits of x, or, if x has fewer than three digits, what is x?"*.

Definition 2 *Let Q be any question about strings. Two strings x and x' are* equivalent with respect to Q, *$x \equiv_Q x'$, if they draw the same answer, i.e., $A_Q(x) = A_Q(x')$.*

The relation $\equiv_Q$ "partitions" all strings into disjoint subset. This partition "induced by $\equiv_Q$" is illustrated in Figure 2.14.

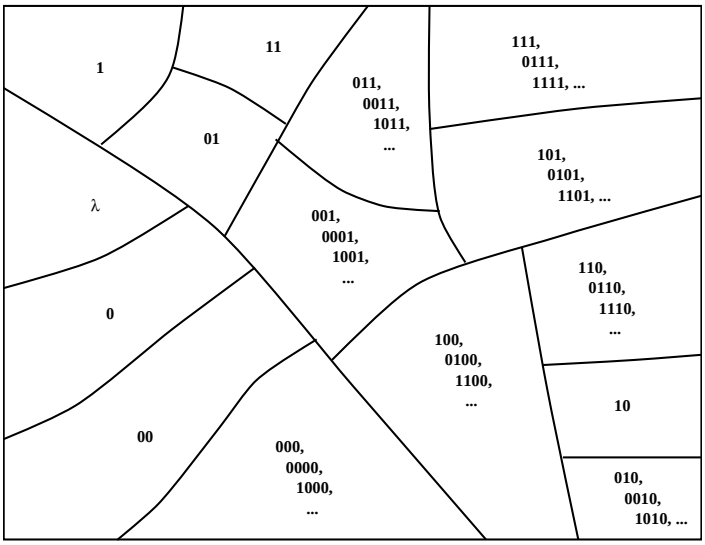


Figure 2.14: The partition induced by Alice's question

The subsets of the partition induced by $\equiv_Q$ then become the state of a finite-state machine. This is illustrated in Figure 2.15. The start state is the state containing the empty string. States containing strings in the language are accepting states. There is a transition labeled 1 from the state that contains 0 to the state that contains 01. Generally, there is a transition labeled c from the state containing s to the state containing sc.

Could it happen that this construction results in more than one transition with the same label out of one state? In this example, we got lucky — Alice asked a very nice question —, and this won't happen. In general it might, though. As an example, Alice might have asked the question, *"If x is empty, say so; if x is nonempty but contains only* 0*s, say so; otherwise, what are the last three digits of x, or, if x has fewer than three digits, what is x?"*. Kind of messy, but better than the original question in the sense that it takes fewer different answers. The problem is that the two strings 0 and 00 draw the same answer, but 01 and 001 don't. Figure 2.16 shows where this causes a problem in the construction of a finite-state machine. The resolution is easy. We arbitrarily choose one of the conflicting transitions and throw away the other(s). Why is this safe? Because $\equiv_Q$ is a "refinement" of $\equiv_L$ and it is safe to ignore distinctions between strings that are equivalent under $\equiv_L$.

The best question

It seems that the finite-state machine in Figure 2.15 is not minimal. We could run our minimization algorithm on it, but let's instead try to work on "mini-

[5]Note: A "language" in this theory means a set of strings.

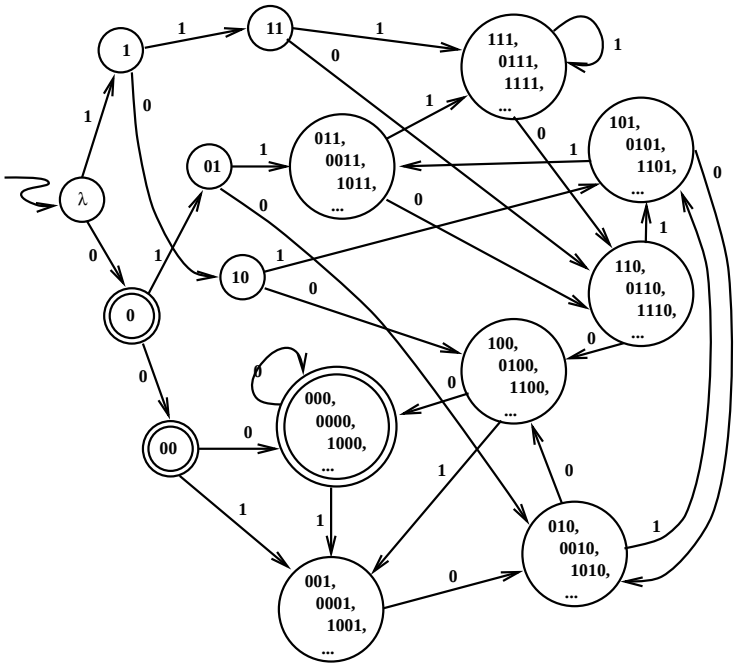


Figure 2.15: The finite-state machine constructed from Alice's question

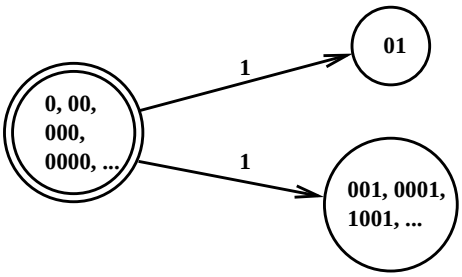


Figure 2.16: A conflict in the construction of a finite-state machine

mizing" the question Q from which the machine was constructed. Minimizing a question for us means minimizing the number of different answers.

The reason why this construction worked was that $\equiv_Q$ was a refinement of $\equiv_L$. This feature of the question Q we need to keep. But what is the refinement of $\equiv_L$ that has the fewest subsets? $\equiv_L$ itself. So the best question is *"Which subset of the partition that is induced by $\equiv_L$ does x belong to?"*. This is easier to phrase in terms not of the subsets but in terms of "representative" strings from the subsets: *"What's a shortest string y with $x \equiv_L y$?"*.

Let's apply this idea to the DIVISIBLE-BY-8 problem. There are four equivalence classes:

$0, 00, 000, 0000, 1000, \ldots$,

$\lambda, 100, 0100, 1100, \ldots$,

$10, 010, 110, 0010, 0110, 1010, 1110, \ldots$, and

$1, 01, 11, 001, 011, 101, 111, 0001, 0011, 0101, 0111, 1001, 1010, 1101, 1111, \ldots$.

The four answers to our best question are $\lambda, 0, 10$, and 1. The resulting finite-state machine has four states and is minimal.

Let's take stock of the equivalence relations (or partitions) that play a role in finite-state machines and the languages accepted by such machines.

The Relations $\equiv_L$, $\equiv_Q$, $\equiv_M$

As mentioned already, every language, i.e., set of strings, L defines ("induces") an equivalence relation $\equiv_L$ between strings:

$$(x \equiv_L y) \stackrel{\text{def}}{\Longleftrightarrow} (\textit{for all strings } z, (xz \in L) \Leftrightarrow (yz \in L)). \tag{2.2}$$

Every question Q about strings defines an equivalence relation $\equiv_Q$ between strings:

$$(x \equiv_Q y) \stackrel{\text{def}}{\Longleftrightarrow} (A_Q(x) = A_Q(y)). \tag{2.3}$$

And every finite-state machine M defines an equivalence relation $\equiv_M$ between strings:

$$(x \equiv_M y) \stackrel{\text{def}}{\Longleftrightarrow} (\delta_M(s_0, x) = \delta_M(s_0, y)). \tag{2.4}$$

s_0 denotes the start state of M. Note that the number of states of M is the number of subsets into which $\equiv_M$ partitions the set of all strings.

As mentioned earlier, for Alice to win the game, her question Q must distinguish between strings x and y for which there exists a string z with $(xz \in L) \not\Leftrightarrow (yz \in L)$. In other words, she must make sure that

$$(x \not\equiv_L y) \Rightarrow (x \not\equiv_Q y) \tag{2.5}$$

i.e., $\equiv_Q$ must be a refinement of $\equiv_L$. If we regard binary relations as sets of pairs, then (2.5) can be written as

$$((x, y) \notin \equiv_L) \Rightarrow ((x, y) \notin \equiv_Q) \tag{2.6}$$

or, for the notationally daring,

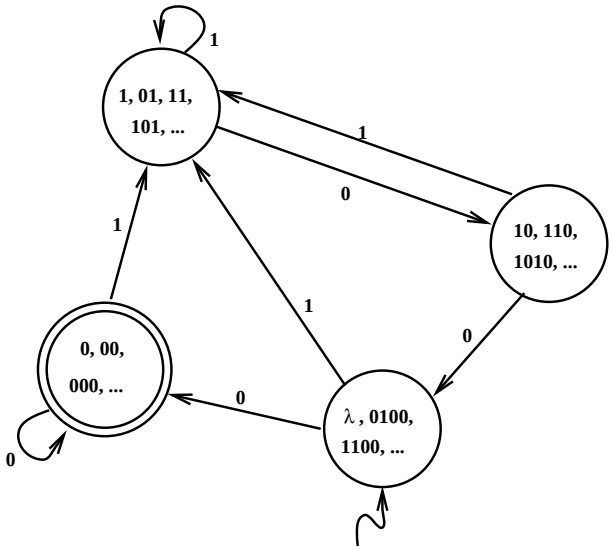


Figure 2.17: Illustrating the uniqueness of a minimal machine

$$\equiv_Q \quad \subseteq \quad \equiv_L \tag{2.7}$$

Instead of talking about Alice and her question, we can talk about a finite-state machine and its states.[6] For the machine M to accept all strings in L and reject all strings that are not in L — a state of affairs usually written as $L(M) = L$ —, M must "distinguish" between strings x and y for which there exists a strings z with $(xz \in L) \not\Leftrightarrow (yz \in L)$. In other words,

$$(x \not\equiv_L y) \Rightarrow (x \not\equiv_M y) \tag{2.8}$$

i.e., $\equiv_M$ must be a refinement of $\equiv_L$. Again, if we regard binary relations as sets of pairs, then (2.8) can be written as

$$\equiv_M \quad \subseteq \quad \equiv_L \tag{2.9}$$

Every machine M with $L(M) = L$ must satisfy (2.9); a smallest machine M_0 will satisfy

$$\equiv_{M_0} \quad = \quad \equiv_L \tag{2.10}$$

Uniqueness How much freedom do we have in constructing a smallest machine for L, given

[6] The analogy becomes clear if you think of a finite-state machine as asking at every step the question *"What state does the input read so far put me in?"*.

that we have to observe (2.10)? None of any substance. The number of states is given, of course. Sure, we can choose arbitrary names for the states. But (2.10) says that the states have to partition the inputs in exactly one way. Each state must correspond to one set from that partition. Figure 2.17 illustrates this for the language DIVISIBLE-BY-8. This then determines all the transitions, it determines which states are accepting, which are rejecting, and which is the start state. Thus, other than choosing different names for the states and laying out the picture differently on the page, there is no choice. This is expressed in the following theorem.

Theorem 14 *All minimal finite-state machines for the same language L are isomorphic.*

Unlike our earlier construction of a finite-state machine from a (non-minimal) winning question of Alice's, this construction will never result in two transitions with the same label from the same state. If that were to happen, it would mean that there are two strings x and y and a character c with $x \equiv_L y$ and $xc \not\equiv_L yc$. Since $xc \not\equiv_L yc$ there is a string s with $(xcs \in L) \not\Leftrightarrow (ycs \in L)$ and therefore a string z $(= cs)$ with $(xz \in L) \not\Leftrightarrow (yz \in L)$, contradicting $x \equiv_L y$.

Figure 2.17 could have been obtained from Figure 2.14 by erasing those lines that are induced by $\equiv_Q$ but not by $\equiv_L$. This is what our minimization algorithm would have done when run on the machine of Figure 2.15.

Summarizing, we have the following theorem.

Theorem 15 (Myhill-Nerode) *Let L be a language. Then the following two statements are equivalent.*

1. *$\equiv_L$ partitions all strings into finitely many classes.*
2. *L is accepted by a finite-state machine.*

Furthermore, every minimal finite-state machine for L is isomorphic to the machine constructed as follows. The states are the subsets of the partition induced by $\equiv_L$. The start state is the set that contains λ. A state A is accepting if $A \subseteq L$. There is a transition from state A to state B labeled c if and only if there is a string x with $x \in A$ and $xc \in B$.

Regular Expressions (Kleene's Theorem)

The above Myhill-Nerode Theorem provides a complete characterization of properties that can be determined with finite-state machines. A totally different characterization is based on "regular expressions," which you may know as search patterns in text editors.

Regular languages and expressions are defined inductively as follows.

Definition 3 (Regular languages) 1. *The empty set $\emptyset$ is regular.*
2. *Any set containing one character is regular.*

3. If A and B are regular then so are $A \cup B$, AB, and $A^\star$.

[The union, concatenation and Kleene star operations are defined as usual:

$$A \cup B = \{x : x \in A \text{ or } x \in B\},$$

$$AB = \{xy : x \in A \text{ and } y \in B\},$$

and

$$A^\star = \{x_1 x_2 \cdots x_k : k \geq 0 \text{ and } x_i \in A \text{ for all } i\}$$

.]

Observation 4 *Finite languages are regular.*

Definition 4 (Regular expressions) *1. The symbol $\emptyset$ is a regular expression and $L(\emptyset) = \emptyset$.*

2. Any single character c is a regular expression and $L(c) = \{c\}$.

3. If α and β are regular expressions then so are $\alpha \cup \beta$, $\alpha\beta$, and $\alpha^\star$, and $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$, $L(\alpha\beta) = L(\alpha)L(\beta)$, and $L(\alpha^\star) = (L(\alpha))^\star$.

Observation 5 *A language L is regular if and only if there is a regular expression E with $L = L(E)$.*

The important result in this section is the following.

Theorem 16 (Kleene's Theorem) *The following two statements are equivalent.*

1. L is regular.

2. L is accepted by some finite-state machine.

Proof Both parts of this proof are inductive constructions.

To construct regular expressions from finite-state machines we number the states and carry out an inductive construction, with the induction being over this numbering of the states. This is a nontrivial example of a proof by induction and a nice illustration of the fact that inductive proofs often become easier once you decide to prove a stronger result than you really wanted.

To construct finite-state machines from regular expressions we do an "induction over the structure" of the expression. This is a nice example of the fact that induction, while strictly speaking a technique that is tied to natural numbers, can be thought of as extending to other sets as well.

Here are the details.

(2) $\Rightarrow$ (1) To construct a regular expression from a given finite-state machine M, first number the states of the machine $1, \ldots, q$, with state 1 being the start

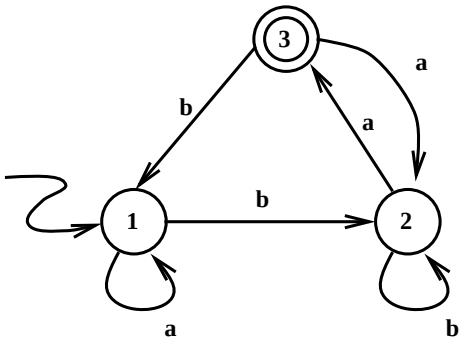


Figure 2.18: Illustrating the notion of a string making a machine "pass through" states

state. Define L_{ij}^k to be the set of all strings which make the machine go from state i into state j without ever "passing through" any state numbered higher than k. For example, if the machine of Figure 2.18 happens to be in state 1 then the string ba will make it first go to state 2 and then to state 3. In this sequence $1 \to 2 \to 3$ we say that the machine "went through" state 2. It did not "go through" states 1 or 3, which are only the starting point and the endpoint in the sequence. Since it did not go through any state numbered higher than 2 we have $ba \in L_{13}^2$. Other examples are $babb \in L_{32}^2$ and $a \in L_{23}^0$, in fact $\{a\} = L_{23}^0$. In this example, the language accepted by the machine is L_{13}^3. In general, the language accepted by a finite-state machine M is

$$L(M) = \bigcup_f L_{1f}^q \tag{2.11}$$

where the union is taken over all accepting states of M. What's left to show is that these languages L_{ij}^k are always regular. This we show by induction on k.

For $k = 0$, L_{ij}^0 is finite. It contains those characters that label a transition from state i to state j. If $i = j$ the empty string gets added to the set.

For $k > 0$,

$$L_{ij}^k = (L_{ik}^{k-1}(L_{kk}^{k-1})^\star L_{kj}^{k-1}) \cup L_{ij}^{k-1}. \tag{2.12}$$

This equation amounts to saying that there can be two ways to get from state i to state j without going through states numbered higher than k. Either we go through k, or we don't, corresponding to the subexpression before the union and the subexpression after the union. Figure 2.19 illustrates this. Figure 2.20 shows some of the steps in the construction of a regular expression from the finite-state machine of Figure 2.18. We simplify the subexpressions as we go. Expression E_{ij}^k describes language L_{ij}^k: "$L(E_{ij}^k) = L_{ij}^k$." Some parentheses are omitted. λ is short for $\emptyset^\star$, hence $L(\lambda) = \{\lambda\}$, by the definition of the *-operation. (We wouldn't want to miss a chance to be obscure, would we?)

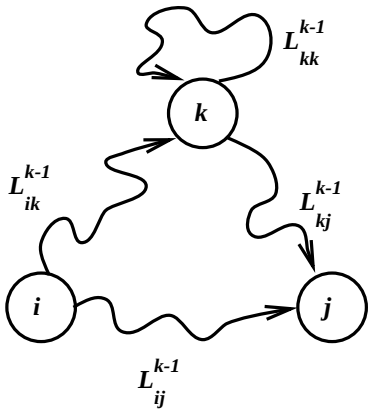


Figure 2.19: The pattern of induction for constructing the regular expressions L_{ij}^k in Kleene's Theorem

(1) ⇒ (2) To construct finite-state machines from regular expressions we use what is called "induction on the structure of the expression" or "structural induction." This means that finite-state machines are first constructed for small subexpressions, then combined to form machines for larger subexpressions and ultimately for the whole expression.

For example, if we want to convert the expression

$$E = (abb) \cup (b(aa)^\star bab)$$

a machine $M_{(aa)^\star}$ for the subexpression $(aa)^\star$ is shown in figure 2.21.

For simplicity let's adopt the convention that if a transition goes to a rejecting state from which the machine cannot escape anymore then we won't draw the transition, nor that "dead" state. This simplifies the picture for $M_{(aa)^\star}$, see Figure 2.22. A machine M_b is easier yet, see Figure 2.23. We can combine M_b and $M_{(aa)^\star}$ to get $M_{b(aa)^\star}$, shown in Figure 2.24. Adding parts that correspond to subexpressions *abb* and *bab* completes the construction, see Figure 2.25. This seems easy enough, but what if the expression had been just slightly different, with an extra b in front:

$$E' = (babb) \cup (b(aa)^\star bab)$$

Then our construction would have gotten us this machine $M_{E'}$ in Figure 2.26.

This machine $M_{E'}$ has a flaw: there are two transitions labeled b out of the start state. When we run such a machine what will it do? What should it do, presented with such an ambiguity? *Can we define the "semantics," i.e., can we define how such a machine is to run, in such a way that the machine $M_{E'}$ accepts the very same strings that the expression E' describes?*

$$\begin{array}{lcl} E^0_{11} &=& \lambda \cup a \\ E^0_{21} &=& \emptyset \\ E^0_{31} &=& b \end{array} \qquad \begin{array}{lcl} E^0_{12} &=& b \\ E^0_{22} &=& \lambda \cup b \\ E^0_{32} &=& a \end{array} \qquad \begin{array}{lcl} E^0_{13} &=& \emptyset \\ E^0_{23} &=& a \\ E^0_{33} &=& \lambda \end{array}$$

$$E^1_{32} = (E^0_{31}(E^0_{11})^\star E^0_{12}) \cup E^0_{32} = (b(\lambda \cup a)^\star b) \cup a = ba^\star b \cup a$$

$$\begin{array}{lcl} E^1_{11} &=& a^\star \\ E^1_{21} &=& \emptyset \\ E^1_{31} &=& ba^\star \end{array} \qquad \begin{array}{lcl} E^1_{12} &=& a^\star b \\ E^1_{22} &=& \lambda \cup b \\ E^1_{32} &=& ba^\star b \cup a \end{array} \qquad \begin{array}{lcl} E^1_{13} &=& \emptyset \\ E^1_{23} &=& a \\ E^1_{33} &=& \lambda \end{array}$$

$$E^2_{13} = (E^1_{12}(E^1_{22})^\star E^1_{23}) \cup E^1_{13} = (a^\star b(b^\star)^\star a) \cup \emptyset = a^\star bb^\star a$$

$$\begin{array}{lcl} E^2_{11} &=& a^\star \\ E^2_{21} &=& \emptyset \\ E^2_{31} &=& ba^\star \end{array} \qquad \begin{array}{lcl} E^2_{12} &=& a^\star b^+ \\ E^2_{22} &=& b^\star \\ E^2_{32} &=& ba^\star b^+ \end{array} \qquad \begin{array}{lcl} E^2_{13} &=& a^\star bb^\star a \\ E^2_{23} &=& b^\star a \\ E^2_{33} &=& (ba^\star b \cup a)b^\star a \end{array}$$

$$E^3_{11} = (E^2_{13}(E^2_{33})^\star E^2_{31}) \cup E^2_{11} = (a^\star bb^\star a((ba^\star b \cup a)b^\star a)^\star ba^\star) \cup a^\star$$

Figure 2.20: Converting the finite-state machine from Figure 2.18 into a regular expression

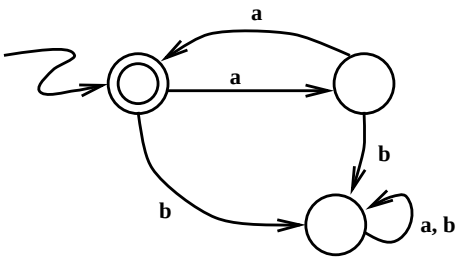


Figure 2.21: The machine $M_{(aa)^\star}$

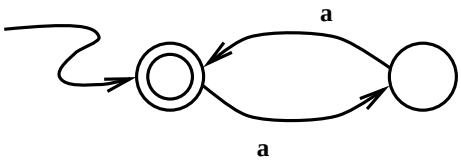


Figure 2.22: The machine $M_{(aa)^\star}$, a simplified drawing

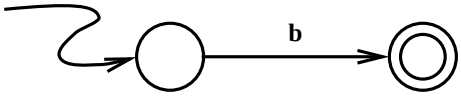


Figure 2.23: The machine M_b

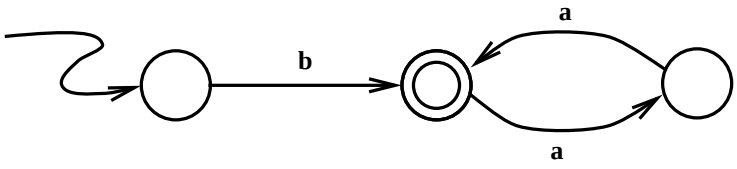


Figure 2.24: The machine $M_{b(aa)^\star}$

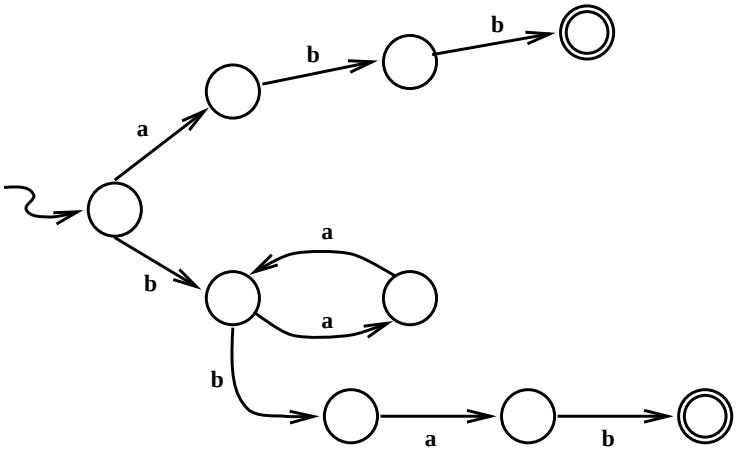


Figure 2.25: The machine M_E

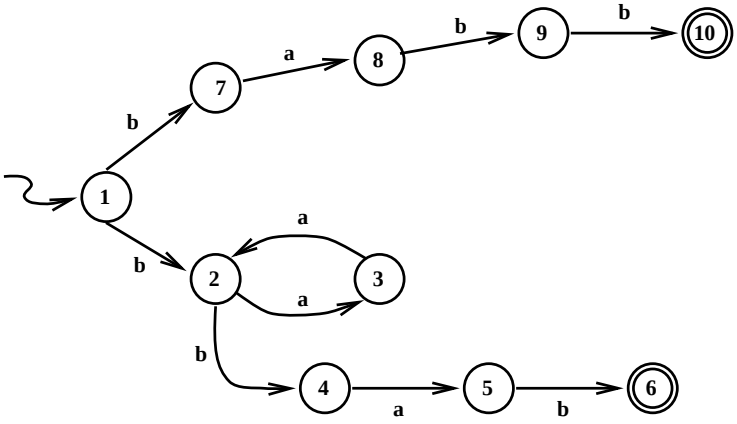


Figure 2.26: The machine $M_{E'}$

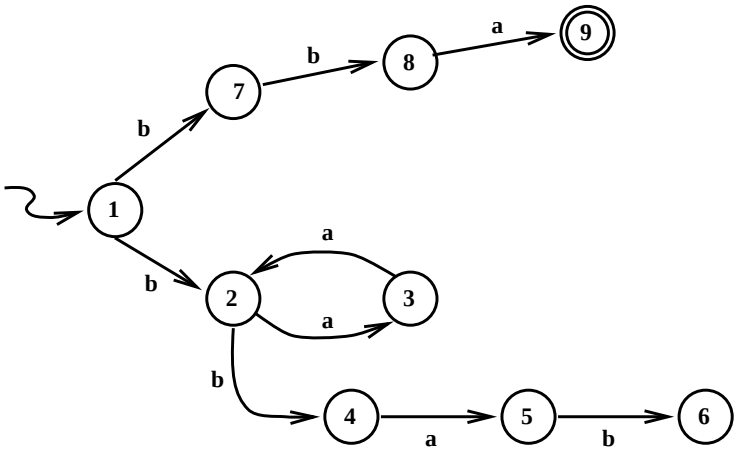


Figure 2.27: The machine $M_{E''}$

After reading an initial b, should $M_{E'}$ be in state 2 or in state 7? We want to accept both *babb* and *baabab*. Being in state 7 after the first b give us a chance to accept *babb*; and being in state 2 gives us a chance to accept *baabab*. This suggests that the right way to run this machine is to have it be *in both states* 2 *and* 7 after reading the first b.

With this way of running the machine, which states would it be in after reading an a as the second character? In states 3 and 8. After reading a b as the third character, it can only be in state 9. [7] After reading another b — for a total input string of *babb* — $M_{E'}$ is in state 10 and accepts the input.

There is one more issue that this example does not address. What if some such machine ends up in a set of states some of which are accepting and some are rejecting? The expression

$$E'' = (bba) \cup (b(aa)^\star bab)$$

provides an example. Our construction will result in the machine $M_{E''}$ of Figure 2.27. After reading *bba*, the machine is in state 5, which is rejecting, and in state 9, which is accepting. Since *bba* is a string described by E'', it should be accepted by $M_{E''}$. That settles the issue: *a string is accepted if at least one of the states it puts the machine in is an accepting state.*

Our conventions about running these machines and knowing when a string is accepted are the "semantics of nondeterministic finite-state machines." These

[7] This could be up to some discussion depending on how we view our convention of not drawing a "dead" state, i.e., a rejecting state from which the machine cannot escape. If we take the view that the state is there — we just don't draw it — then at this point we should say that the machine would be in state 9 and that invisible state, say 11. This is clumsy. It is more elegant — although ultimately of no important consequence — to say that the machine is only in state 9.

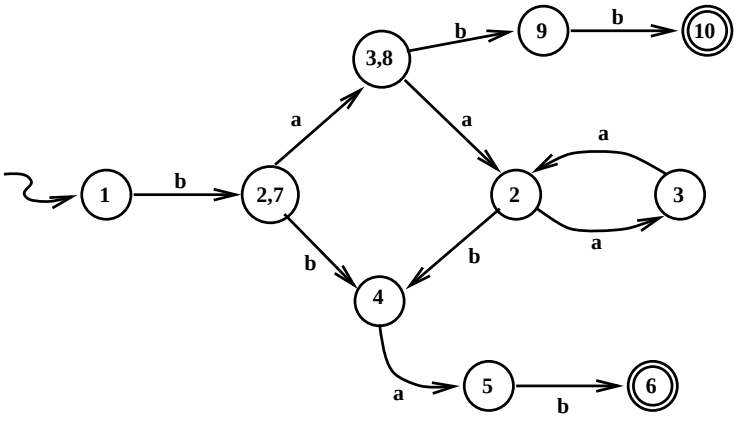


Figure 2.28: Converting $M_{E'}$ into a deterministic machine

conventions can be applied to more powerful programs as well (and we will do so in a later chapter on computational complexity, where the concept of nondeterminism will play a crucial role in defining a class of computational problems). "Nondeterministic C," for example, could be C with the extra feature of a nondeterministic branch:

```
( nondet-branch; x = 0; x = 1; )
```

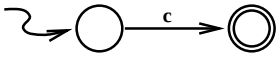
The way to think about the execution of such nondeterministic programs is that there is more than one path of execution. Tracing the execution results in a tree and a program is said to accept its input if at least one of the paths leads to acceptance.

Can we convert this new kind of machine into a standard (deterministic) finite-state machine? This task amounts to a search of the "execution tree," which we do in a breadth-first approach. Figure 2.28 gives an example, converting the machine $M_{E'}$ into a deterministic one. The states of the deterministic machine are the *sets* of states of $M_{E'}$. (Many of these sets of states may not be reachable from the start state and can be discarded.) In the deterministic machine, the state reached from a state A via a transition c, $\delta(A, c)$, is the set of those states b of $M_{E'}$ to which there is a c-transition (in $M_{E'}$) from some state $a \in A$.

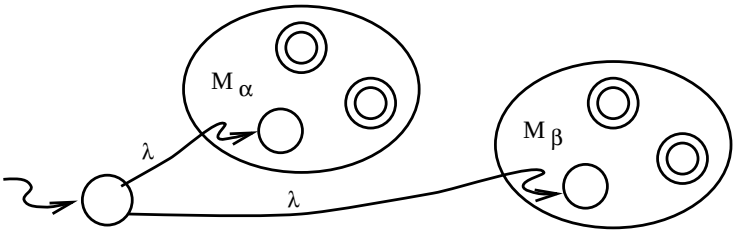
The algorithm for constructing a nondeterministic machine from a regular expression is easier to describe if we allow yet another form of nondeterministic behavior: making a "spontaneous" transition, i.e., a transition that does not consume an input character. The standard term is a "λ-transition." The inductive algorithm for constructing a (nondeterministic) finite-state machine M_E from an arbitrary regular expression E then works as shown in Figure 2.29. An example that uses all the parts of this algorithm is the machine shown in Figure 2.30.

1. $M_\emptyset$ is the following machine:

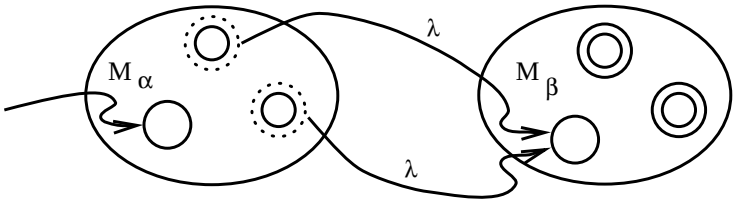
2. For any single character c, M_c is the following machine:



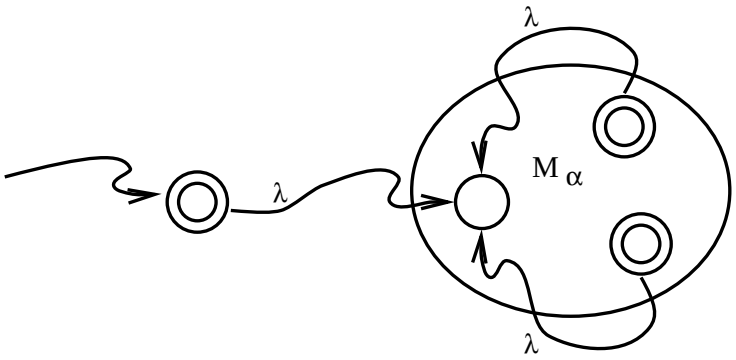
3. If α and β are regular expressions then $M_{\alpha\cup\beta}$, $M_{\alpha\beta}$, and $M_{\alpha^\star}$ are constructed from M_α and M_β as shown below.



Constructing $M_{\alpha\cup\beta}$ from M_α and M_β



Constructing $M_{\alpha\beta}$ from M_α and M_β



Constructing $M_{\alpha^\star}$ from M_α

Figure 2.29: An algorithm for converting regular expressions to finite-state machines

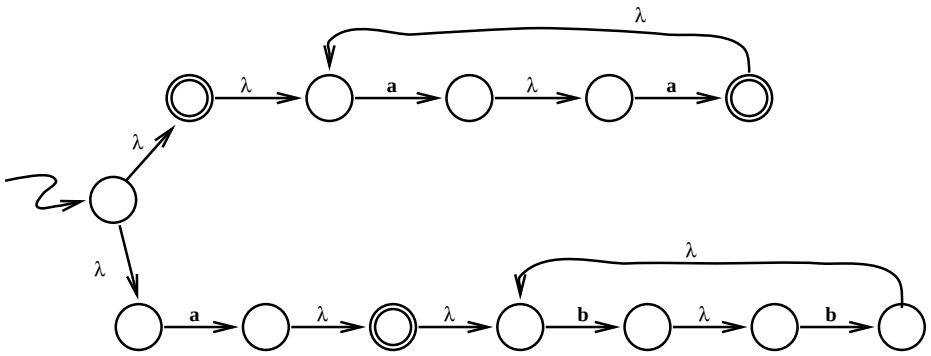


Figure 2.30: The machine $M_{(aa)^\star \cup (a(bb)^\star)}$

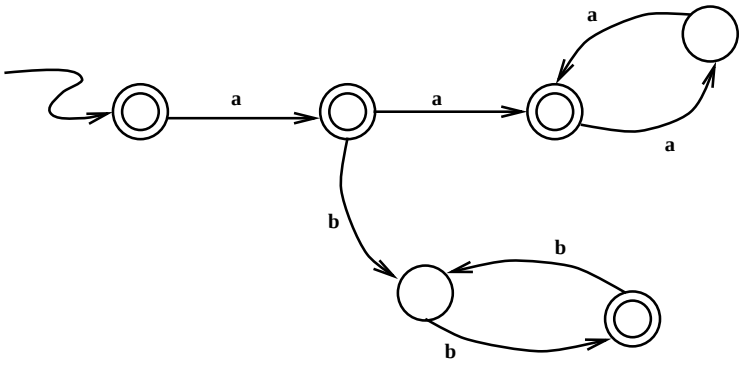


Figure 2.31: A deterministic version of $M_{(aa)^\star \cup (a(bb)^\star)}$

Nondeterministic machine with λ-transitions can be turned into deterministic ones with the same set-of-states approach used earlier. For example, the machine in Figure 2.30 turns into the machine in Figure 2.31.

This concludes the proof of Kleene's Theorem. □

Finite-State Machines or Regular Expressions?

Given that finite-state machines and regular expressions are equivalent concepts, we can choose to approach some problem in terms of one or the other. Some questions are easier to deal with in terms of finite-state machines, others in terms of regular expressions. Here are some examples, all of them variations of the question of which languages are regular:

Question 1. Is the set of binary numbers that are divisible by 2 regular?

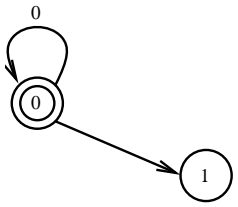


Figure 2.32: A piece of a finite-state machine

This is easy in either domain. The right regular expression is $(0 \cup 1)^\star 0$.

Question 2. Is the set of binary numbers that are divisible by 3 regular?

The answer is yes, but it seems nearly impossible to build a suitable regular expression. After all, what is the common pattern of 11, 110, 10000000010, 1001? Yet it is not all that hard to build a finite-state machine for this language, with the right approach anyway. The Myhill-Nerode Theorem tells us that we would have to figure out exactly what it is that we want to keep track of as the input gets processed.

If we know that the input so far was a number that was divisible by 3, then we know that after reading one more 0, the number is again divisible by 3. (This is so because reading that 0 amounted to multiplying the previous number by 2.) What if a 1 had been read instead? This would have amounted to multiplying the previous number by 2 and adding 1. The resulting number, when divided by 3, would yield a remainder of 1. This translates into a piece of a finite-state machine shown in Figure 2.32. The complete machine is shown in Figure 2.33. The four equivalence classes of $\equiv_L$ (and hence the states of the minimal machine for L) are $\{\lambda\}$ and the three sets of binary numbers which, when divided by three, yield a remainder of 0, 1, and 2. These remainders are used as the labels of the three states.

Question 3. Are regular languages closed under union? I.e., if A and B are regular, is their union $A \cup B$ regular as well?

The answer is yes. It follows from the definition of regular sets and expressions. In terms of finite-state machines, this requires an argument involving nondeterministic machine (as used in the proof of Kleene's Theorem).

Question 4. Are regular languages closed under complement? I.e., if A is regular, is $\overline{A} = \{x : x \notin A\}$ regular as well?

This is most easily dealt with in terms of finite-state machines. If M is a finite-state machine for A then a finite-state machine for $\overline{A}$ can be obtained by turning accepting states into rejecting ones, and vice versa. There is no equally simple argument in terms of regular expressions.

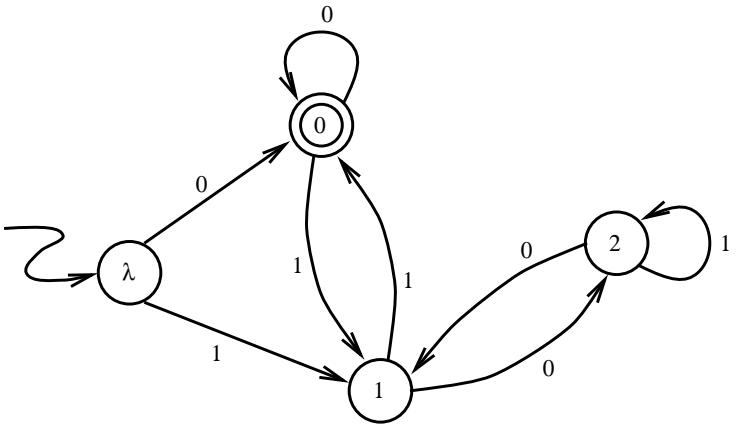


Figure 2.33: A finite-state machine for binary numbers divisible by 3

Question 5. Are regular languages closed under set difference? I.e., if A and B are regular, is their difference $A - B = \{x : x \in A \text{ and } x \notin B\}$ regular as well?

There is a construction based on finite-state machine that proves this closure result and is interesting in its own right.

Question 6. Is the set of syntactically correct C programs regular?

It is not, and the only proof method presented so far is to argue that there are infinitely many equivalence classes induced by $\equiv_L$. Here is an infinite list of nonequivalent strings:

```
main () { int x; x = (
main () { int x; x = ((
main () { int x; x = (((
main () { int x; x = ((((
main () { int x; x = (((((
...
```

This shows that parsing programs, which is a necessary part of compiling them, cannot be done with an amount of memory that does not depend on the input that is being parsed. As we will see in the next chapter, it takes a "stack" to parse.

2.3 The Pumping Lemma for Regular Languages

"Pumping" is a second technique to prove nonregularity. It is interesting in its own right, although no more powerful than the analysis of $\equiv_L$ that we have discussed already in Sections ??. Pumping is important if for no other reason then because unlike the analysis of $\equiv_L$ in Myhill-Nerode, pumping arguments also work for the more powerful programs we will consider in the next chapter.

Lemma 3 Pumping Lemma for Regular Languages. *If L is a regular language then there is a constant k such that for every string w in L of length k or more the following is true.*

There are three strings, u, v, x, such that

1. *$w = uvx$, and*
2. *for all $i \geq 0$, $uv^i x \in L$, and*
3. *$|v| > 0$.*

Let's use this Lemma to prove that the language $L = \{a^n b^n : n \geq 1\}$ is not regular.

Assuming it was. Then the Lemma applies. Let k be the constant said to exist in the Lemma. Consider the string $w = a^k b^k$. The length of w is k or more (in fact, $2k$), hence statements (1) — (3) from the Lemma apply, implying that it is possible to split w into three parts, $w = uvx$, such that $|v| > 0$ and $uv^2x \in L$. But this is not true. No matter what three-way split of w one tries, uv^2x is never of the form of equal numbers of a's and b's.

(One can come up with a more detailed argument for this. The challenge is to cover all ways in which the strings could get divided up into three pieces. An added challenge is to come up with a *concise* argument. Here is an attempt:

The substring v cannot contain more than one type of character because otherwise v^2 contains characters in an order not permitted in L. Hence at most one type of character take part in the "pumping" from uvx to uv^2x, which therefore does not contain equal numbers of both characters anymore.)

This contradiction shows that L is not regular. □

Why is this so messy? The Pumping Lemma can be stated as follows.

$$L \text{ regular}$$

$$\Longrightarrow$$

There is a constant k such that
for all strings $w \in L$ with $|w| \geq k$
there are strings u, v, x with $w = uvx$ and $|v| > 0$ such that
for all $i \geq 0$, $uv^i x \in L$.

The above statement is of the form "*L regular* $\Rightarrow B$," where B is that complicated statement with four alternating quantifiers. Since we are trying to prove that some languages are not regular, a more useful version of the statement is "*L not regular* $\Leftarrow \neg B$." Spelled out in detail, this version reads

$$\begin{array}{c} L \text{ not regular} \\ \Longleftarrow \\ \text{For all constants } k \\ \text{there is a string } w \in L \text{ with } |w| \geq k \text{ such that} \\ \text{for all strings } u, v, x \text{ with } w = uvx \text{ and } |v| > 0 \\ \text{there is an } i \geq 0 \text{ with } uv^i x \notin L. \end{array}$$

Thus, we can prove that some language L is not regular by proving the above four-line statement $\neg B$: *"For all ... $\notin L$"*. This is what we did in the previous example. Here is another one.

Another Example

Let's take the language that consists of strings like

```
1234+111=1345
10000+10000=20000
1020304+505=1020809
11111+22222=33333
```

The language consists of strings of the form $\alpha + \beta = \gamma$ that represent a *correct* equation between decimal integers.

This language L is not regular, a fact we prove by proving the statement "$\neg B$" above.

How do we prove something *"for all constants k"*? By starting out the proof by saying "Let k be an arbitrary constant" and carrying the k along as a parameter of the argument.

What's next? We need to show that *there exists* a string w with certain properties. To show this, all we need to do is exhibit *one* such string. In this example, let's pick w to be the string

$$\underbrace{1\cdots1}_{k} + \underbrace{2\cdots2}_{k} = \underbrace{3\cdots3}_{k}$$

(Note how the k plays its role.) What's the next step? We have to show that *for all* strings u, v, x which represent a three-way breakup of w ("$w = uvx$") and which don't have the second piece empty ("$|v| > 0$"), there is a constant i with $uv^i x \notin L$. This is the hard part of the proof because we somehow have to provide an argument that covers all possible such three-way breakups of w. It can be done, by dealing with an exhaustive set of cases. Here is such an exhaustive set of cases for this example.

Case 1. v contains the =. Then $uv^0 x \notin L$ if for no other reason then because it does not have an = in it.

Case 2. v does not contain the $=$. Then v lies entirely on one side of the $=$ and pumping (with any $i \neq 1$) changes that side of the equation but not the other, thus creating a string outside of L.

Exercises

Ex. 2.1. Draw the smallest finite-state machine M for the language of all bitstrings that contain the substring 011.

Ex. 2.2. Draw the smallest finite-state machine M for the language of all bitstrings that contain the substring 011. (Make sure you provide a 0 and a 1 transition out of every state.)

Minimizing finite-state machines

Ex. 2.3. Carry out the minimization algorithm of Figure 2.6 for the finite-state machine shown below. (Show the development of the "clusters" of states just like Figure 2.8 did.)

Ex. 2.4. Draw a finite-state machine on which the minimization algorithm runs through more than 10 passes.

The Myhill-Nerode Theorem

Ex. 2.5. Let L be the language of all bitstrings that contain a run of three consecutive 0s. Is it true that $01 \equiv_L 001$? Explain.

Ex. 2.6.

1. If L is the language of all bitstrings that start with a 1 and end with a 0, then $01 \equiv_L 00$.
2. If L is the language of all bitstrings of length 8, then $0001 \equiv_L 0011$.
3. Let L be the language of all bitstrings x such that x contains at least one 0. (For example, $01 \in L$, $1111 \notin L$.) How many equivalence classes does $\equiv_L$ induce?
4. Let L be the language of all bitstrings x such that x contains at least one 0 and at least one 1. (For example, $01 \in L$, $1111 \notin L$, $00 \notin L$.) How many equivalence classes does $\equiv_L$ induce?

Ex. 2.7. Let L be the language of binary strings whose third bit from the end is a 1. Give one string from each of the equivalence classes induced by $\equiv_L$.

Ex. 2.8. Same for the language of bitstrings whose first two bits are equal to its last two bits.

Ex. 2.9. Give one representative of each equivalence class of $\equiv_L$ for $L = \{x : x$ *contains the substring* $000\}$.

Ex. 2.10. Consider the following language L over the alphabet $\Sigma = \{a, b, \$\}$:

$$L = \{x\$y : x, y \in \{a, b\}^+ \textit{ and } x \textit{ does not start with the same letter as } y\}$$

For every equivalence class of $\equiv_L$, give one shortest string from the class.

Ex. 2.11. Is it true that $x \equiv_L y$ implies $xz \equiv_L yz$? Is it true that $xz \equiv_L yz$ implies $x \equiv_L y$? When your answer is no, give a counterexample.

Ex. 2.12. Are the two relations $\equiv_L$ and $\equiv_{\bar{L}}$ (that's $\bar{L}$, the complement of L) always the same, for any given L? Either argue that they are, or show an example were they aren't?

Ex. 2.13. *(b)* Do the partitions that go with $\equiv_L$ and with $\equiv_{L^{rev}}$ (that's $L^{rev} = \{x^{rev} : x \in L\}$) always have the same equivalence classes? Either argue that they always do, or show an example were they don't? (x^{rev} is x written backwards.)

Ex. 2.14. Which of the following languages is regular? (Just circle "Yes" or "No" for each language. Do not give explanations.)

1. The language described by $(ab(a^\star)^\star(b^\star \cup (aab)^\star))$
2. Strings containing the word `while`
3. Strings containing two opening parentheses with no closing parenthesis in between the two, e.g. "`(a)b)c(d(e(f`"
4. Strings containing more opening parentheses than closing ones, e.g. "`())(((`"
5. Strings containing all their opening parentheses before all their closing ones, e.g. "`((x(y-/)`"
6. Strings of even length whose first half and second half both start with the same letter, e.g. "`abcabaeeee`"

Ex. 2.15. Consider two finite-state machines that accept languages A and B and consider their cross-product machine. As discussed in class, one can always choose the accepting states of the cross-product machine such that it accepts $A \cap B$.

Can one always choose the accepting states of the cross-product machine such that it accepts ...

1. $A \cup B$?
2. $A - B$? (That's $\{x : x \in A \text{ and } x \notin B\}$.)
3. AB ? (That's A concatenated with B: $\{xy : x \in A \text{ and } y \in B\}$.)
4. Strings in $A \cup B$ of length five or more?
5. A ?

Regular Expressions

Ex. 2.16. Derive an asymptotic bound on the length of the expression constructed from a finite-state machine in the proof of Kleene's Theorem. Assume we are dealing with bitstrings only.

Nondeterministic Machines

Ex. 2.17. Consider the problem of taking an arbitrary nondeterministic finite-state machine that accepts some language A and modifying the machine to accept the complement of A instead (i.e., all the strings that are not in A).

Does the following approach work: Make all rejecting states of the nondeterministic machine into accepting states and all accepting states into rejecting states?

Briefly explain your answer.

Ex. 2.18. For any $k \geq 0$ let L_k be the language of bitstrings whose first k bits equal their last k bits. How many states does a deterministic finite-state machine for L_k need? Same for a nondeterministic machine. How long is a regular expression for L_k?

Ex. 2.19. Give three properties that are decidable for finite-state machines but undecidable for arbitrary programs.

Ex. 2.20. Which of the languages in Figure **??** are regular?

Cross-Product Construction

Ex. 2.21. Consider creating the cross-product machine of two given machines and choosing the accepting states such that the new machine accepts the intersection of the languages of the two old machines.

(a) Give an example where the cross-product machine is minimal.

(b) If the cross-product machine is minimal, does that imply that the two "factors" in the product had to be minimal? Give an argument or a counterexample.

Miscellaneous

Ex. 2.22. For any program p define L_p to be the set of input strings on which the program prints the word "Yes" and terminates. Is it decidable whether or not a program p has a regular language L_p? Briefly justify your answer.

Ex. 2.23. Let FSM-EQUIVALENCE be the problem of deciding, given two finite-state machines P and Q, whether or not $L(P) = L(Q)$. Let FSM-EMPTINESS

be the problem of deciding, given a finite-state machine M, whether or not $L(M) = \emptyset$. Describe how a cross-product construction can be used to transform FSM-EQUIVALENCE into FSM-EMPTINESS.

Ex. 2.24. Construct a finite-state machine for the language $\{x : x \in \{0,1\}^\star$ *and* $x \notin \{000, 001\}^\star\}$. First construct a nondeterministic machine that is as small as possible, then apply the subset construction, then minimization.

Ex. 2.25. For two of the nonregular languages in Figure **??** use the pumping lemma to prove that they are not regular.

Ex. 2.26. Prove that every infinite regular language contains a string whose length is not a prime number.

Ex. 2.27. Prove that the set of strings that are finite prefixes of this infinite string
-0-1-10-11-100-101-110-111-1000-1001-1010-1011- $\cdots$
is not regular.

Ex. 2.28. Formulate a stronger version of the pumping lemma, one that says not only that pumping occurs somewhere in a long enough string but that it occurs within every long enough substring. Use this stronger version to prove two languages from [8] Figure **??** not regular which cannot be proven not regular with the original weaker version.

[8] To do: This is not in the sample solutions yet.

Chapter 3

Context-Free Languages

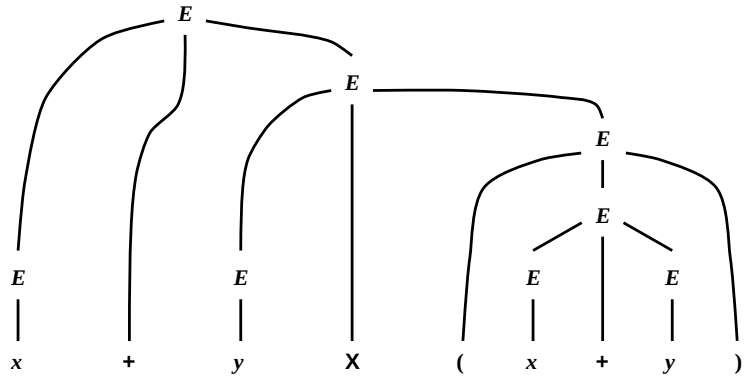
3.1 Context-Free Grammars

As we saw at the end of the previous chapter, one feature of programming languages that makes them nonregular is the arbitrary nesting of parenthesis in expressions. Regular expressions cannot be used to define what syntactically correct expressions look like. "Context-free grammars" can.

An example of a context-free grammar is this:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E \times E \\ E &\rightarrow (E) \\ E &\rightarrow x \\ E &\rightarrow y \end{aligned}$$

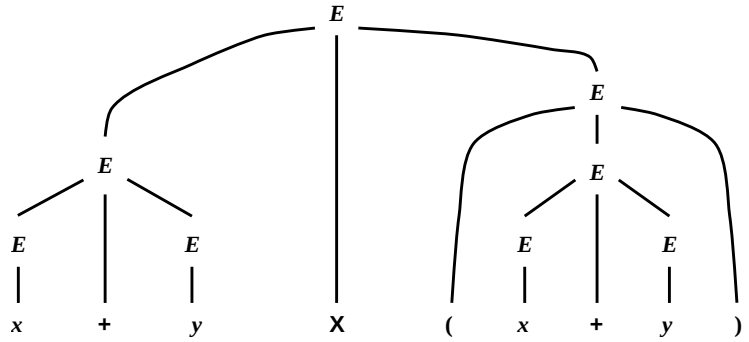
The way this grammar — call it G_1 — is used is to start with the symbol on left of the arrow in the first line, E. This is the "start symbol" — the string of length 1 we start out with. We then grow that string by repeatedly replacing what's on the left-hand side of an arrow with what's on the right-hand side. The best way to keep track is a tree:



There are choices of which replacements to make. The choices made above resulted in a tree whose leaves, read from left to right, read $x+y\times(x+y)$. The grammar G_1 is said to "derive" that string: $x + y \times (x + y) \in L(G_1)$. The tree is a "derivation tree" for the string.

Derivation trees are also called "parse trees." They are used in compilers to know how to translate a given expression, or a whole program, into machine language, and in interpreters to know how to evaluate an expression, or interpret a whole program.

Our sample grammar does not serve that purpose well, though. This is because it allows a second derivation tree for the same string:

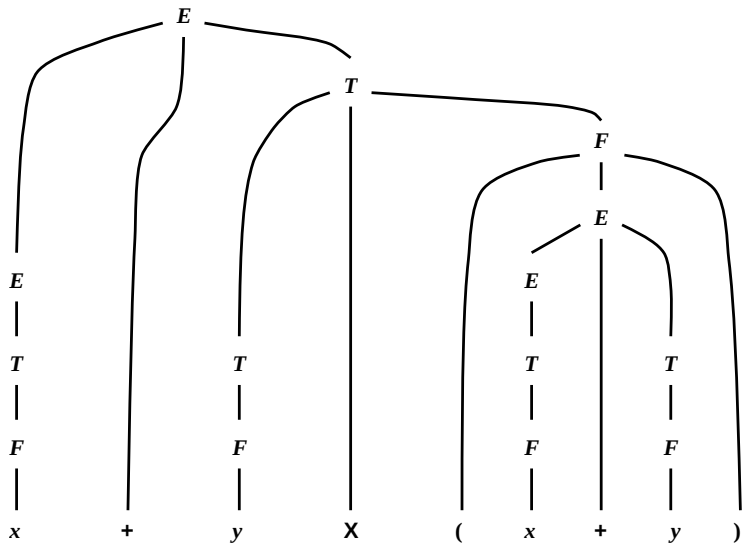


Such a grammar is said to be "ambiguous." The two parse trees for the string $x+y\times(x+y)$ lead to two different evaluations of the expression. If we plug in $x = 1$ and $y = 5$, the first tree leads to a value of 36 for the expression, whereas the second one leads to a value of 31. Of the two trees, only the first one matches our conventions about the precedence of arithmetic operations. The first one is "right," the second one is "wrong." The problem is, this grammar G_1, while it does perfectly well describe the language in question, does not help us define the one and only "correct" way to parse each string in the language. The following

grammar G_2 does do that:

$$\begin{aligned} E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow T \times F \\ T &\rightarrow F \\ F &\rightarrow (E) \\ F &\rightarrow x \\ F &\rightarrow y \end{aligned}$$

This grammar only allows one derivation tree, the "right" one, for any arithmetic expressions involving $+$, $\times$, parentheses, and the variables x and y. The tree for the expression $x + y \times (x + y)$ is:



3.2 Parsing

Recursive-Descent Parsing

A grammar like G_2 above is easily turned into a parser, i.e., a program that determines whether or not a given input string is in the language and does so (necessarily) by building a parse tree, implicitly or explicitly. See Figure 3.1. The program is nondeterministic, though — certainly a flaw we do not want to tolerate for very long.

In what sense does this program "parse" expressions?

```
main ( )
{
    E();
    if (getchar() == EOF)
        ACCEPT;
    else
        REJECT;
}

E ( )
{
    nondet-branch:
        {E(); terminal('+'); T();}
        {T();}
}

T ( )
{
    nondet-branch:
        {T(); terminal('*'); F();}
        {F();}
}

F ( )
{
    nondet-branch:
        {terminal('('); E(); terminal(')');}
        {terminal('x');}
        {terminal('y');}
}

terminal( char c )
{
    if (getchar() != c)
        REJECT;
}
```

Figure 3.1: A nondeterministic recursive-descent parser

Being a nondeterministic program, the program as a whole is said to accept an input string if there is at least one path of execution that accepts it. Consider again the string $x+y\times(x+y)$. The one path of execution that results in acceptance is the one where the first execution of `E` in turn calls `E`, `terminal('+')`, and `T`. These calls correspond to the children of the root of the derivation tree. If we chart the calling pattern of the accepting execution we end up drawing the parse tree. *The accepting computation looks like a preorder traversal of the parse tree of the input string.*

The above construction shows how to turn any context-free grammar into a program that is like the finite programs of Chapter 2 except that it uses two additional features: recursion and nondeterminism. Such programs are often called "pushdown machines" or "pushdown automata (pda's)." They are equivalent to finite-state machines augmented by a stack to implement recursion. Thus the above example generalizes to the following theorem.

Theorem 17 *For every context-free grammar G there is a pushdown machine M with $L(M) = L(G)$.*

The converse is also true, providing a complete characterization of context-free languages in terms of a type of program.

Theorem 18 *For every pushdown machine M there is a context-free grammar G with $L(M) = L(G)$.*

(We don't prove this here.)

Deterministic Parsing

Intrinsically Nondeterministic Context-Free Languages

With regular languages and finite-state machines it was always possible to remove the nondeterminism from the machines. The trick was to replace states by sets of states. This worked because the power set (set of subsets) of a finite set was again a finite set. This approach does not work for pushdown machines because a set of stacks, or even just two stacks, cannot be implemented with a single stack. Without giving any proof of this, here is an example of a grammar that derives an intrinsically nondeterministic language. The language is very simple, all bitstrings of odd length whose middle character is a 1.

$$\begin{aligned} S &\rightarrow B\ S\ B \mid 1 \\ B &\rightarrow 0 \mid 1 \end{aligned}$$

To parse this language with a pushdown machine, nondeterminism is needed. We do not prove this here, but the intuition is that the parser must keep building up a stack of recursive calls for the first half of the input string and climb out of the recursion during the second half. The problem is, with information limited by a constant amount, it is not possible to know when the second half starts.

A "Predictive" Recursive-Descent Parser

(The example is after Aho and Ullman, *Principles of Compiler Design*, 1978, p.176ff.)
While a deterministic recursive-descent parser does not exist for all context-free languages, some languages are simple enough to yield to such an approach. For example, the grammar for arithmetic expressions used earlier can be transformed, without changing the language, into the following:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \lambda \\ T &\rightarrow FT' \\ T' &\rightarrow \times FT' \mid \lambda \\ F &\rightarrow (\ E \ ) \\ F &\rightarrow x \\ F &\rightarrow y \end{aligned}$$

The work done by the first two productions version, which was to generate intermediate strings of the form

$$T + T + \cdots + T$$

is done by the first two productions of the new version. The new grammar leads to a parser in which the decision which of the nondeterministic paths has a chance of leading to acceptance of the input can be made on the basis of looking at the next character of the input. This is so because whenever the grammar has several right-hand sides for the same left-hand symbol, the right-hand sides differ in their first character. The resulting deterministic parser in shown in Figure 3.2.

A General Parsing Algorithm

Instead of using nondeterminism, which is impractical even when implemented by a deterministic tree-traversal, one can use a "dynamic programming" approach to parse any context-free language. Dynamic programming is a type of algorithm somewhat resembling "divide-and-conquer," which inductively (or recursively, depending on the implementation) builds up information about subproblems. It typically has $O(n^3)$ time complexity.

There are two steps to using this technique. First, we change the grammar into "Chomsky Normal Form," then we apply the actual dynamic programming algorithm.

Chomsky Normal Form Every context-free grammar can be converted into an equivalent one in which each production is of one of the two forms

```
main ( )
{
    E();
    if (getchar() == EOF)
        ACCEPT;
    else
        REJECT;
}

E ( ) { T(); E'(); }

E' ( )
{
    if (NEXTCHAR == '+')
        {terminal('+'); T(); E'();}
    else
        // do nothing, just return
}

T ( ) { F(); T'(); }

T' ( )
{
    if (NEXTCHAR == '*')
        {terminal('*'); F(); T'();}
    else
        // do nothing, just return
}

F ( )
{
    switch (NEXTCHAR)
    {
        case '(': {terminal('('); E(); terminal(')');}
                  break;
        case 'x': terminal('x');
                  break;
        case 'y': terminal('y');
                  break;
        default:  REJECT;
    }
}

...
```

Figure 3.2: A deterministic recursive-descent parser

$$\begin{aligned} A &\rightarrow BC \\ A &\rightarrow a \end{aligned}$$

where A, B, C are syntactic variables, not necessarily different ones, and a is a terminal symbol. This form of a context-free grammar is called "Chomsky Normal Form." To see how this conversion can be done consider this production:

$$A \rightarrow BCdEfG \tag{3.1}$$

The production 3.1 can equivalently be replaced by the set of productions

$$\begin{aligned} A &\rightarrow BCDEFG \\ D &\rightarrow d \\ F &\rightarrow f \end{aligned} \tag{3.2}$$

and the production 3.2 can then be replaced by the set of productions

$$\begin{aligned} A &\rightarrow BC' \\ C' &\rightarrow CD' \\ D' &\rightarrow DE' \\ E' &\rightarrow EF' \\ F' &\rightarrow FG \end{aligned}$$

(The example below is from Hopcroft and Ullman, Introduction to Automata Theory, Languages, and Computation, 1979, pp. 140-1.)

CYK Parsing: An Example of "Dynamic Programming"

Figure 3.3 shows an example. The "*length = 2, starting position = 3*" field contains all those variables of the grammar from which one can derive the substring of length 2 that starts at position 3. (The string is *ab*. The variables are S and C.) The rows of the table get filled in from top (length=1) to bottom (length=5). The whole string is in the language if the start symbol of the grammar shows up in the bottom field. Filling in a field makes use of the contents of fields filled in earlier, in a pattern indicated by the two arrows in the Figure.

The algorithm is easily modified to yield a parse tree, or even sets of parse trees.

3.3 The Pumping Lemma for Context-Free Grammars

The grammar

$$
\begin{aligned}
S &\rightarrow AB \mid BC \\
A &\rightarrow BA \mid a \\
B &\rightarrow CC \mid b \\
C &\rightarrow AB \mid a
\end{aligned}
$$

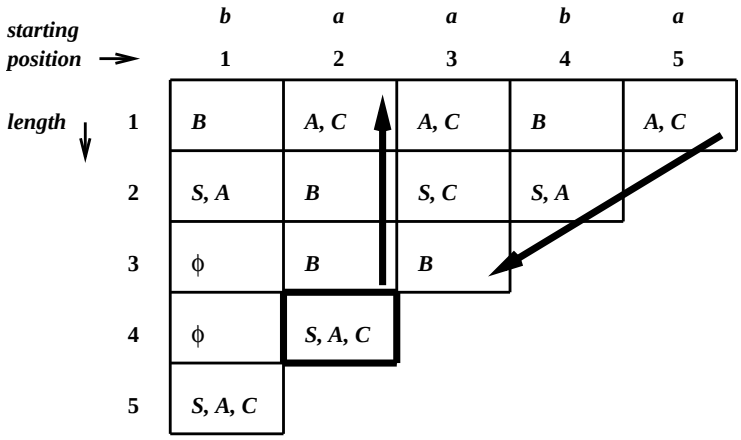


Figure 3.3: An example of "CYK" parsing

$$S \rightarrow 0\ S\ 1 \mid 0\ 1$$

generates the language

$$L_4 = \{0^n 1^n | n \geq 1\}.$$

(0^n means a string of n 0s.)

Similar grammars generate

$$L_5 = \{0^n 1^n 0^m 1^m | n, m \geq 1\},$$

$$L_6 = \{a^n b^m c^n | n, m \geq 1\},$$

$$L_7 = \{a^n b^m c^m d^n | n, m \geq 1\}.$$

If we think of 0, a, and b as opening parentheses and 1, c, and d as closing parentheses, then these language all are examples of balanced parentheses.[1] More complicated languages arise if we try to have equal numbers of not just two but three or more different characters, as for example in

$$L_8 = \{a^n b^n c^n : n \geq 1\},$$

and if we match numbers of characters beyond a nesting pattern, as for example in

$$L_9 = \{a^n b^m c^n d^m : n, m \geq 1\}.$$

If you try to write context-free grammars for L_8 and L_9, you sooner or later get frustrated trying to achieve the specified equalities between the exponents (i.e., the numbers of repetitions of characters) in the definitions of these languages. In fact,

Theorem 19 *L_8 and L_9 are not context-free.*

The languages L_8 and L_9 are of no practical interest. But after we develop a proof technique for showing that these languages are not context-free we will be able to use that technique to show that some aspects of programming languages are not context-free either.

The proof of Theorem 19 uses a "cutting-and-pasting" argument on the parse trees of context-free grammars. It is the most common argument for proving languages not to be context-free. The heart of the argument is that every context-free language has a certain "closure" property: The fact that certain strings are in the language forces other strings to be in the language as well. A

[1] L_7 is an example of nesting of different types of parentheses. Replace a and d by opening and closing parentheses and b and c by opening and closing square brackets. Then L_7 becomes $\{(^n [^m]^m)^n | n, m \geq 1\}$.

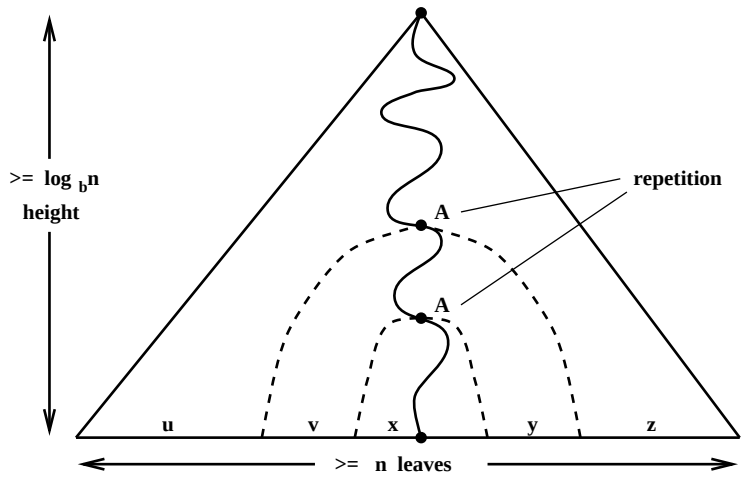


Figure 3.4: Illustrating the proof of Theorem 19

language that does not have the closure property cannot be context-free. The details of that closure property are developed next.

If a language L is context-free there is, by definition, a context-free grammar G that generates L, $L = L(G)$. Let b be the maximum length of right-hand sides of productions of G. In other words, b is a bound on the maximum number of children for nodes in parse trees of G. Then a parse tree with n leaves must be of height at least $\log_b n$, where height is defined to be 0 for a tree consisting of a single node. This means that a tree with n leaves must have a path from the root to a leaf with at least $\log_b n - 1$ interior nodes. What if n is so large that $\log_b n - 1$ exceeds the number of different left-hand sides of productions in G? (These are also called "syntactic variables.") Then some syntactic variable must show up at least twice along that path. Figure 3.4 illustrates this.

Pick such a variable, call it A, and pick two occurrences of A along one path. These two occurrences of A split the string w that the tree derives into five parts, labeled u, v, x, y, z in Figure 3.4. (The substring vxy is derived from the first of the two occurrences of the variable A, the substring x from the second occurrence.)

Knowing that we can derive x from A, the tree shown in the top half of Figure 3.5 is also a parse tree in this grammar. It derives the string uxz. There are other trees we can construct. For example, as illustrated in the bottom half of Figure 3.5, the second occurrence of A can be made to derive vxy, resulting in a tree that derives the string $uvvxyyz$. No need to stop there. We can build trees that derive $uvvvxyyyz$, $uvvvvxyyyyz$, in fact any string uv^ixy^iz for any $i \geq 2$.

For $i = 0, 1$, the string uv^ixy^iz becomes uxz and $uvxyz$, respectively. This lets us summarize the effect of all this cutting-and-pasting of trees as the fol-

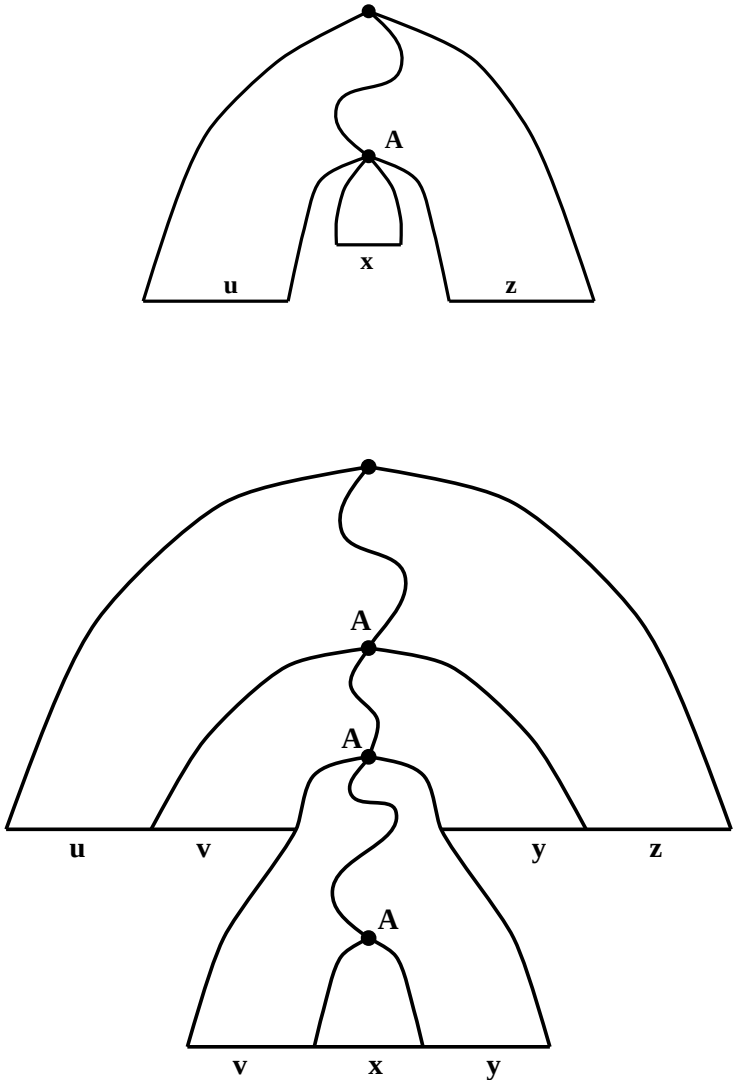


Figure 3.5: Deriving uxz and $uvvxyyz$

lowing

Lemma 4 (Pumping Lemma for Context-Free Languages, Version 1) *If L is an context-free language then there is a constant k such that for every string w in L of length k or more the following is true.*

There are five strings, u, v, x, y, z, such that

1. *$w = uvxyz$, and*
2. *for all $i \geq 0$, $uv^ixy^iz \in L$, and*
3. *$|vy| > 0$.*[2]

This lemma is strong enough to let us prove that the language $L_8 = \{a^nb^nc^n : n \geq 1\}$ from Theorem 19 is not context-free. (We will need to add one more refinement before we can handle L_9.)

Proof that L_8 is not context-free. Assuming it was. Then Lemma 4 applies. Let k be the constant said to exist in Lemma 4. Consider the string $w = a^kb^kc^k$. The length of w is k or more (in fact, $3k$), hence statements (1) — (3) from the Lemma apply, implying that it is possible to split w into five parts, $w = uvxyz$, such that $|vy| > 0$ and $uv^2xy^2z \in L_8$. But this is not true. No matter what five-way split of w one tries, uv^2xy^2z is never of the form of equal numbers of a's, b's, and c's following each other.[3] This contradiction shows that L_8 is not context-free. □

The Pumping Lemma can be stated as follows. **Why is this so messy?**

$$L \textit{ context-free}$$

$$\Longrightarrow$$

There is a constant k such that
for all strings $w \in L$ with $|w| \geq k$
there are strings u, v, x, y, z with $w = uvxyz$ and $|vy| > 0$ such that
for all $i \geq 0$, $uv^ixy^iz \in L$.

[2]This last part of the statement we did not prove. The proof is left as an exercise. The statement is saying that not both of v and y are empty.

[3]One can come up with a more detailed argument for this. The challenge is to cover all ways in which the strings could get divided up into five pieces. An added challenge is to come up with a *concise* argument. Here is an attempt:

The substring v cannot contain more than one type of character because otherwise v^2 contains characters in an order not permitted in L_8. The same is true for y. Hence at most two types of characters take part in the "pumping" from $uvxyz$ to uv^2xy^2z, which therefore does not contain equal numbers of all three characters anymore.

What makes such a statement hard to digest are the alternations of "*for all*" with "*there is/are*," akin to the nesting of loops in a program.[4]

The above statement is of the form "L *context-free* $\Rightarrow B$," where B is that complicated statement with four alternating quantifiers. Since we are trying to prove that some languages are not context-free, a more useful version of the statement is "L *not context-free* $\Leftarrow \neg B$." Spelled out in detail, this version reads

$$L \text{ not context-free}$$

$$\Longleftarrow$$

$$\begin{array}{c} \textit{For all constants } k \\ \textit{there is a string } w \in L \textit{ with } |w| \geq k \textit{ such that} \\ \textit{for all strings } u, v, x, y, z \textit{ with } w = uvxyz \textit{ and } |vy| > 0 \\ \textit{there is an } i \geq 0 \textit{ with } uv^i xy^i z \notin L. \end{array}$$

Thus, we can prove that some language L is not context-free by proving the above four-line statement $\neg B$: *"For all ... $\notin L$"*. This is what we did in the previous example. Here is another one.

Another Example Let's take the language that consists of strings like

```
1234+111=1345
10000+10000=20000
1020304+505=1020809
11111+22222=33333
```

The language consists of strings of the form $\alpha + \beta = \gamma$ that represent a correct equation between decimal integers.

This language L is not context-free, a fact we prove by proving the statement "$\neg B$" above.

How do we prove something *"for all constants k"*? By starting out the proof by saying "Let k be an arbitrary constant" and carrying the k along as a parameter of the argument.

What's next? We need to show that *there exists* a string w with certain properties. To show this, all we need to do is exhibit *one* such string. In this example, let's pick w to be the string

$$\underbrace{1\cdots1}_{k} + \underbrace{2\cdots2}_{k} = \underbrace{3\cdots3}_{k}$$

[4]This this called an "alternation of quantifiers." There are formal results that say that adding a level of alternation makes statements harder in the sense that even if we had a program that could determine the truth of statements with q alternations of quantifiers, this would not enable us to build a program that determines truth of statements with $q + 1$ quantifiers. (The concept of transformations is rearing its ugly head.) Look for the subject of "the arithmetic hierarchy" in textbooks on recursive function theory if you want to read about this.

(Note how the k plays its role.) What's the next step? We have to show that *for all* strings u, v, x, y, z which represent a five-way breakup of w ("$w = uvxyz$") and which don't have the second and fourth piece both empty ("$|vy| > 0$"), there is a constant i with $uv^ixy^iz \notin L$. This is the hard part of the proof because we somehow have to provide an argument that covers all possible such five-way breakups of w. It is tedious, but it can be done, by dealing with an exhaustive set of cases. Here is such an exhaustive set of cases for this example.

Case 1. Either v or y or both contain the + and/or the =. Then $uv^0xy^0z \notin L$ if for no other reason then because it does not have a + and an = in it.

Case 2. Neither v nor y contain the + or the =.

Case 2.1. Both v and y lie on the same side of the =. Then pumping (with any $i \neq 1$) changes that side of the equation but not the other, thus creating a string outside of L.

Case 2.2. v lies entirely on the left side of the equation and y lies entirely on the right side.

Case 2.2.1. One of v and y is empty. Then pumping changes one side of the equation but not the other, creating a string outside L.

Case 2.2.2. Neither v nor y is empty.

Case 2.2.2.1. v lies within the first number of the equation. Then uv^2xy^2z is of the form

$$\underbrace{1\cdots1}_{>k}+\underbrace{2\cdots2}_{k}=\underbrace{3\cdots3}_{>k}$$

which is not a correct equation.

Case 2.2.2.2. v lies within the second number of the equation. This case is analog to Case 2.2.2.1. uv^2xy^2z is of the form

$$\underbrace{1\cdots1}_{k}+\underbrace{2\cdots2}_{>k}=\underbrace{3\cdots3}_{>k}$$

which is not a correct equation.

An example that does not work

Going back to the language $L_9 = \{a^nb^mc^nd^m : n, m > 0\}$, if we tried to construct a proof that it is not context-free by constructing an exhaustive set of cases just like in the previous example, we would sooner or later realize that there is a case (or two) that does not work.

This is the case where v consists of a number of a's and y consists of the same number of c's. (The other case is the analogous situation with b's and d's.)

The proof attempt fails.[5] It takes a stronger version of the Pumping Lemma to prove that this language is not context-free.

[5]This is nothing but bad news. Failing to prove that L is not context-free does not prove that it is context-free either. It proves nothing.

A Stronger Version of the Pumping Lemma

In the proof of the Pumping Lemma, instead of merely arguing that there is a repetition of some syntactic variable along a path from the root to a leaf, we could instead argue that such a repetition can always be found "close to" a leaf. This then implies a limit on how long the string vxy can be. Here are the details.

If a grammar has s different syntactic variables and no right-side of a production is longer than b, then a repetition of a syntactic variable occurs within $s+1$ steps along the path from a leaf to the root and the subtree under the second occurrence of the variable (second meaning the one further away from the leaf) has at most $k' = b^{s+1}$ leaves. This number is a constant that depends only on the grammar. (Draw a picture.) This improvement in the proof of the Pumping Lemma makes it easier to prove that a language L is context-free. We only have to show that

For all constants k
there is a string $w \in L$ with $|w| \geq k$ such that
for all strings u, v, x, y, z with $w = uvxyz$ and $|vy| > 0$ and $|vxy| \leq k$
there is an $i \geq 0$ with $uv^ixy^iz \notin L$.

(There is a little subtlety in this statement in that the old constant k and the new constant k' were combined into a single constant, which is the maximum of the two and called again k.)

Now it works

The only difference to the old version is the additional condition that "$|vxy| \leq k$." This is very helpful, though, because it means that cases in which "$|vxy| > k$" need not be considered. This takes care of the failed cases in the unsuccessful proof attempt above, because for v to consist of a's only and y of c's only (or b's and d's, respectively) the string vxy would have had to stretch over all the b's (or the c's) and hence be longer than k.

3.4 A Characterization of Context-Free Languages

As alluded to in the title of this current chapter, context-free grammars "place parentheses." When writing grammars for languages like $\{a^ib^ic^jd^j : i, j \geq 0\}$, it helps to recognize the pattern of "parentheses" in the language: Every "opening" a is matched by a "closing" b, and all of this is then followed by opening cs matched by closing ds. The type of production that is at the core of a grammar for such matching characters is

$$T \to a\ T\ b$$

which results in a path of Ts in a derivation tree with matching as and bs being generated on the left and right of the path. A complete grammar for the above language is

$$\begin{array}{rcl} S & \rightarrow & T\ T' \\ T & \rightarrow & a\ T\ b\ \mid\ \lambda \\ T' & \rightarrow & c\ T'\ d\ \mid\ \lambda \end{array}$$

On the other hand, a non-nested pattern of matching characters as in $\{a^i b^j c^i d^j : i, j \geq 0\}$ is beyond the power of context-free grammars (as can be shown with a pumping argument).

It is easy to see that any depth of nesting with any set of parentheses-like characters is within the reach of context-free grammars. The grammar

$$\begin{array}{rcl} S & \rightarrow & S\ S\ \mid\ \lambda \\ S & \rightarrow & (\ S\ ) \\ S & \rightarrow & [\ S\ ] \\ S & \rightarrow & \{\ S\ \} \\ S & \rightarrow & a\ \mid\ b \end{array}$$

generates all properly nested strings of three types of parentheses with letters a and b interspersed. Three types of parentheses is not the limit, nor are two characters, of course. If we use $(_i$ and $)_i$ to denote parentheses of type i, $1 \leq i \leq n$, and let $T = \{a_1, \ldots, a_m\}$ be any finite set of characters, then the grammar

$$\begin{array}{rcl} S & \rightarrow & S\ S\ \mid\ \lambda \\ S & \rightarrow & (_1\ S\ )_1 \\ & \vdots & \\ S & \rightarrow & (_n\ S\ )_n \\ S & \rightarrow & a_1\ \mid\ \cdots\ \mid\ a_m \end{array}$$

generates all properly nested strings of these n types of parentheses with letters $a_1\ \ldots a_m$ interspersed. Let's call this language $PAR_n(T)$. Furthermore, for every set A of characters and every language L, let $Erase_A(L)$ be the set of all strings in L from with all occurrences of characters in A have been deleted.

Theorem 20 *Every context-free language is equal to*

$$Erase_A(R \cap PAR_n(T))$$

for some sets A and T of characters, some regular language R, and some integer $n \geq 0$.

What this theorem says is that every context-free language can be built by starting with a context-free grammar that produces essentially nothing but

nested parentheses, and intersecting this language with a regular language and finally throwing away all occurrences of certain characters. The only context-free step in this construction is the initial generation of $PAR_n(T)$. This supports the contention that the only aspect of a context-free language that could possibly require a context-free grammar is some form of placement of parentheses and conversely, any placement of matching characters that does not follow the nesting pattern of parentheses is bound to be beyond the power of context-free grammars. [A proof of the above theorem can be found in *Davis and Weyuker*.]

Exercises

Ex. 3.1. Which of the following languages are context-free?

1. $L = \{a^n b^m : n, m \geq 1$ and both even $\}$
2. $L = \{a^n b^m c^q : n, m, q \geq 1$ and at least two of the three numbers are equal $\}$
3. $L = \{a^n b^m c^q : n, m, q \geq 1$ and at least two of the three numbers are different $\}$

Ex. 3.2. In the dynamic programming algorithm for parsing, if all the entries in the top row are $\emptyset$, does this imply that the string is not in the language? What if one of them is $\emptyset$? What if all the entries in the second row are $\emptyset$? The third row? Fourth?

Ex. 3.3. Is it decidable if a context-free language, given by a context-free grammar

1. ... contains a string of length less than 100?
2. ... contains a string of length more than 100?
3. ... is finite?

Ex. 3.4. Is it decidable if the set of all outputs of a program forms a context-free language? Explain briefly.

Pumping Lemma

Ex. 3.5. Consider the language $L = \{\alpha\#\beta : \alpha, \beta \in \{0,1\}^*$, α and β have the same length, and the bitwise AND between α and β is not all zeroes $\}$. Examples of strings in L are 01#11, 010000#111111, 010101#101011.

Prove that L is *not* context-free.

Ex. 3.6. Which of the following languages are context-free?

1. $L_{10} = \{a^i b^j a^j b^i : i, j \geq 0\}$
2. $L_{11} = \{a^i b^j a^i b^j : i, j \geq 0\}$
3. $L_{12} = \{\alpha_1 \# \alpha_2 \# \cdots \# \alpha_p \$ \beta_1 \# \beta_2 \# \cdots \# \beta_q : p, q \geq 0,$
   $\alpha_i, \beta_i \in \{0,1\}^\star,$ and $\{\alpha_1, \alpha_2, \cdots, \alpha_p\} \supseteq \{\beta_1, \beta_2, \cdots, \beta_q\}\}$

   (L_{12} captures the properties of some programming languages that variables need to be declared before they are used.) Prove your answer either by giving a grammar or by providing an argument that the language is not context-free.

Ex. 3.7. Even if A and B are both context-free, $A \cap B$ may not be. Prove this by giving an example of two context-free languages whose intersection is not context-free.

Ex. 3.8. The proof of the Pumping Lemma did not show that the pumped strings v and y could not both be empty. This might happen when the grammar has what is known as "unit productions" — productions that merely replace one variable by another. (An example of a unit production is $E \to T$ in one of our grammars for arithmetic expressions.)

(a) The mere presence of unit productions does not yet make it possible for both v and y to be empty. For example, our grammar for arithmetic expressions (the version with variables E, F, and T) has unit productions but does not have derivation trees with v and y both empty. Something else must be true about the unit productions. What is that?

(b) There are two ways to correct this deficiency in the proof. One is to show that unit productions can be removed can be removed from any grammar without changing the language — this involves suitable changes to the other productions. The other is to rephrase the proof of the Pumping Lemma to work around the fact that some nodes of a parse tree might only have one child. The latter approach entails a redefined notion of "height" of a parse tree. Usually, the height of a tree is defined to be 1 plus the maximum height of its subtrees. What would be a suitable modification of this definition?

Ex. 3.9. Any finite-state machine for a language L can be turned into a context-free grammar for L. If the machine has a transition labeled c from state A to state B, then the grammar has a production $A \to cB$.

(a) Carry out this translation from machine to grammar for the machine that accepted binary numbers divisible by 3.

(b) What other productions need to be added to the grammar to make sure it derives the same strings that the machine accepts? (Give an answer for the specific example and also for the general case.)

Ex. 3.10. Following up on the previous exercise, if every regular language is context-free (as proven by that exercise), then the Pumping Lemma applies to every regular language. The special form of the grammars obtained from finite-state machines suggest a simplification to the statement of such a Pumping Lemma.

(a) Spell out such a simplified version.

(b) Use it to prove that $L_{13} = \{a^n b^n | n > 0\}$ is not regular.

Chapter 4

Time Complexity

4.1 The State of Knowledge Before NP-Completeness

Up to and through the 1960s, algorithm designs such as greedy algorithms, local optimization, divide-and-conquer, and dynamic programming were applied successfully to many problems. But there were also plenty of problems which defied all attempts at solving them with reasonably fast algorithms[1]. Here are four examples of such "intractable" problems.

SATISFIABILITY

SATISFIABILITY, introduced in Chapter 1, can be phrased in terms of Boolean expressions or in terms of circuits. In terms of Boolean expressions, the problem is to find out, given an expression E, whether or not there is an assignment of Boolean values to the variables of E that makes E true. In terms of circuits, the problem is to find out, given a circuit C, whether or not there is an assignment of Boolean values to the wires in the circuit that "satisfies" all the gates. "Satisfying a gate" means assigning consistent values to its input and output wires. For example, assigning *true* to both inputs of an AND-gate and *false* to its output would not be consistent, nor would be assigning *true* to one input wire, *false* to the other, and *true* to the output wire.

Boolean expressions or circuits — one can easily translate circuits into expressions and vice versa — turn out to be a rather flexible "language" for expressing constraints on solutions in many situations.

GRAPH 3-COLORABILITY (3-COLOR)

GRAPH 3-COLORABILITY (3-COLOR) problem also was introduced in Chapter 1. It is really a scheduling problem, trying to fit a set of activities, some of which cannot take place at the same time, into three time slots. This situation is captured by a graph with the activities being the nodes and with conflicting activities being connected by an edge. Then the problem becomes, given a

[1] On the question of what is "reasonably fast," there is general agreement that the exponential running of exhaustive search is not "reasonably fast." Below that, let's say that any algorithm whose running time is bounded by a polynomial in the input length is "reasonably fast."

graph, to find out if it is possible[2] to color each node with one of the three colors without ever giving the same color to the two endpoints of an edge.

VERTEX COVER VERTEX COVER is another problem defined on graphs. The challenge is to "cover" the graph in the following sense. You can choose a node and thereby "cover" all the edges connected to that node. The problem is to cover all the edges of the graph choosing the smallest number of nodes. For example, the graph in Figure 4.1 can be covered with 4 nodes, but not with 3. In the decision version of this problem you are given a graph G and a number n, and the Yes/No question to decide is whether or not the graph G can be covered with n nodes.

HAMILTONIAN PATH Think of a graph as a map of cities that are connected by roads. In the HAMILTONIAN PATH problem the challenge is to find a tour from a given start node to a given end node that visits each city exactly once. This is a special case of the so-called TRAVELING SALESMAN PROBLEM (TSP) where each edge has a length and the challenge is to find a tour of minimal length that visits each city.

How Algorithm Designs Can Fail: One Example It is instructive to try to apply some of our algorithm design ideas to some of these intractable problems. As one example, let us try to solve SAT with a divide-and-conquer approach.

Dividing a Boolean expression is easy. Consider the parse tree of the expression and divide it into the left and right subtree. Figure 4.3 provides an illustration.

A simple-minded notion of "conquering" a subtree would be to find out whether or not it is satisfiable. But this information about a subtrees is not enough to find out whether or not the whole expression is satisfiable. For example, if both subtrees are satisfiable and are connected with an AND operation, the whole expression may still not be satisfiable. It could be that in order to satisfy the left subtree it is necessary to make a variable, say x, *true*, but in order to satisfy the right subtree it is necessary to make the same variable x *false*.

This suggests a more sophisticated divide-and-conquer. Instead of just finding out whether or not a subtree is satisfiable, let's find the set of assignments of *true* and *false* to its variables that satisfy it. In fact, we only need to keep track of the assignments to those variables that also occur in the other "half" of the expression. Then to compute the set of assignments that satisfy the whole expression, we can intersect (in the case of an AND) the sets of assignments that satisfy the two parts.

Have we succeeded in devising a divide-and-conquer solution to Boolean satisfiability? We need to fill in some details, such as how to handle operations other than AND, but, yes, it would work. It would not work very fast, though, because the sets of assignments can get large, in fact exponentially large in terms of the number of variables. So we have a exponential-time divide-and-conquer

[2] With all these problems, there is the "decision version" — *"Is it possible ... ?"* — and the full version — *"Is it possible and if so, how ... ?"*.

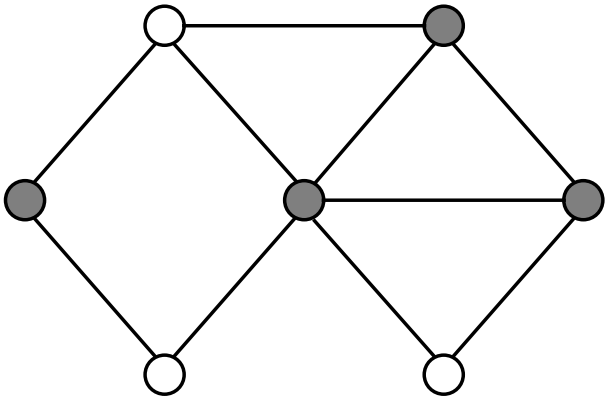


Figure 4.1: Illustrating the VERTEX COVER problem

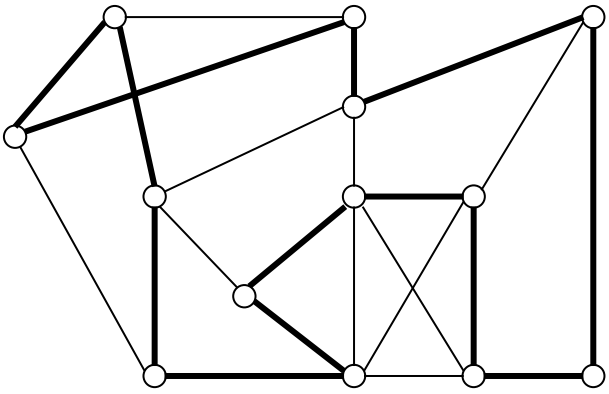


Figure 4.2: Illustrating the HAMILTONIAN CIRCUIT problem

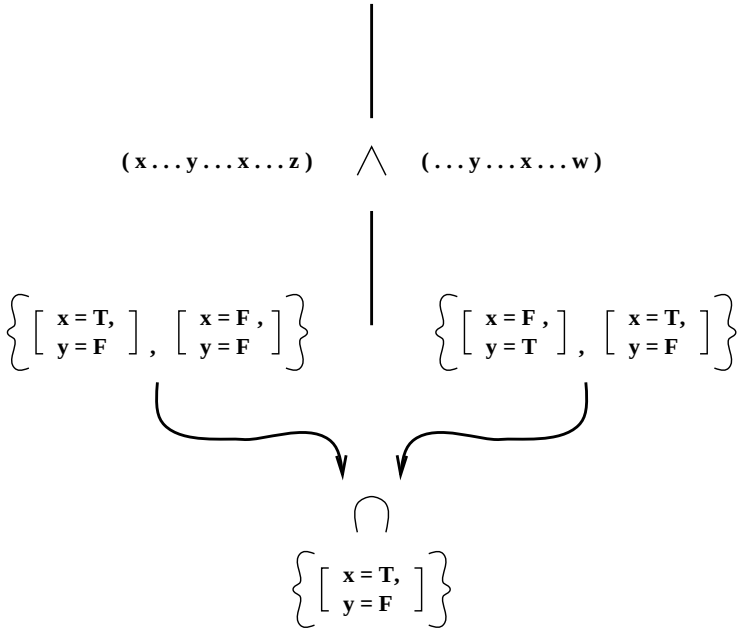


Figure 4.3: Trying to solve SAT with a divide-and-conquer approach

algorithm for SAT. But if we were willing to tolerate exponential running times (which we are not), we could have done a simple exhaustive search in the first place.

What to do? We could try to compute and store those sets more efficiently. To make a long story short, nobody has succeeded in trying to do that. Up to this date, there is no polynomial-time algorithm for SAT, and in fact there is evidence that none might exist. The same is true for the other three problems (3-COLOR, VERTEX COVER, HAMILTONIAN CIRCUIT) and dozens or hundreds[3] of similar "combinatorial optimization" problems. This leads us to developments of the early 1970s.

4.2 NP

Nondeterminism

SAT, 3-COLOR and all the other problems are in fact easy to solve, in polynomial time — if we "cheat." The technical term for this form of cheating is "nondeterministic time."

For an example, consider 3-COLOR. A nondeterministic program, as described earlier, may contain a statement like

```
nondet {x[i] = BLUE; x[i] = RED; x[i] = GREEN;}
```

Execution of such a statement causes a three-way branch in the execution sequence, resulting in an execution tree. If this statement is inside a loop that runs `i` from 1 to n, there is successive branching that builds up a tree with 3^n nodes at the end of this loop. Each node at that level in the execution tree corresponds to one assignment of colors to the nodes of the graph. Figure 4.4 shows the shape of such a tree.

Following this loop, we can have a deterministic function that tests if the assignment of colors is a correct 3-coloring. If so, it prints "Yes," if not, it prints "No."

Now is the time to "cheat," in two ways. First, we define the output of the whole tree to be "Yes" if there exists a path at the end of which the program prints "Yes" (and "No" if the program prints "No" at the end of *all* paths). Second, we define the time spent on this computation to be the *height* of the tree. This is the notion of *nondeterministic time*.

With this notion of programs and time, not only 3-COLOR becomes easy to solve in polynomial time, but so do SAT, VERTEX COVER, HAMILTONIAN CIRCUIT and in fact just about any combinatorial optimization problem that you can think of. After all, this nondeterministic time allows us to conduct what amounts to an exhaustive search without paying for it.

Definition 5 *The class of problems that can be solved in nondeterministic polynomial time is called NP.*

[3] depending on whether one counts related problems as separate ones

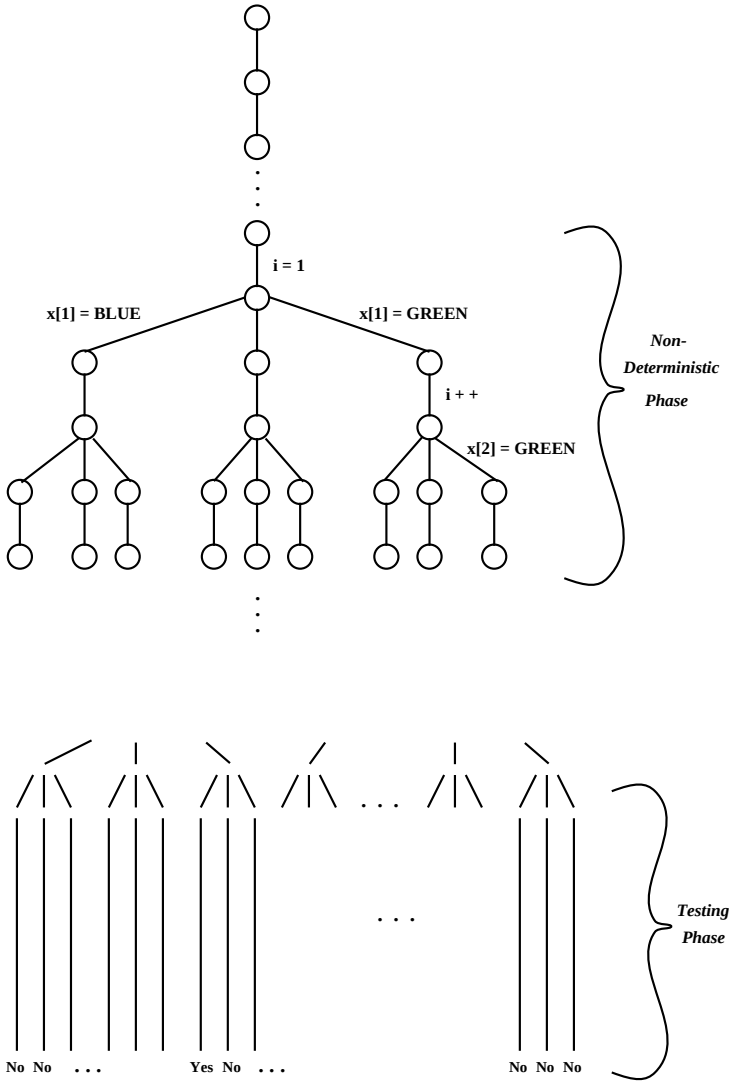


Figure 4.4: The shape of an execution tree for 3-Color

Observation 6 SAT, 3-COLOR, VERTEX COVER, HAMILTONIAN CIRCUIT $\in$ *NP.*[4]

4.3 The NP-completeness of SATISFIABILITY

The big step forward in the early 1970s was to realize that many of these intractable problems were related to each other. For example, as we have seen in Chapter 1, 3-COLOR can be transformed into SAT. Since we are now concerned about running time, it matters how fast that transformation can be carried out, but it is not difficult to verify that

Observation 7 *There is a polynomial-time*[5] *transformation of* 3-COLOR *into* SAT.

As a consequence,

Observation 8 *If we could find a polynomial-time algorithm for* 3-SAT, *we would also have a polynomial-time algorithm for* 3-COLOR.

This last Observation makes 3-SAT the more ambitious of the two problems to solve (or, more precisely, no less ambitious than 3-COLOR). Moreover, what the Observation says about 3-COLOR is also true for VERTEX COVER and HAMILTONIAN CIRCUIT and in fact for all problems in NP:

Theorem 21 (Cook, [Coo71]) *For every decision problem L* $\in$*NP, there is a polynomial-time transformation of L into* SAT.

The proof of Cook's Theorem needs to be an argument that for any problem in NP there is a polynomial-time transformation into SAT.

In Chapter 1, we did transformations of two different problems into SAT. One was GRAPH 3-COLORING, the other a problem of addition of binary integers. Transforming specific problems into SAT is usually not that hard. The challenge with Cook's Theorem is that we need to argue that there exists a transformation of a problem into SAT without even knowing exactly what that problem is. All we know is that the problem is in NP. As it turns out, there aren't a lot of messy technical details to this proof, just the right use of familiar concepts like compilation and circuits.

CIRCUIT SATISFIABILITY and SATISFIABILITY of Boolean Expressions

What happens when a Boolean circuit executes can be captured by a boolean expression. The value of a wire w of a circuit at time t becomes the value of a variable w_t in the expression, and the operation of a gate in the circuit at time t becomes a boolean operation that defines the correct operation of the gate.

[4]Here we are talking about the "decision versions" of the problems, the versions that ask for a "Yes/No" answer.

[5]honestly, i.e., deterministically, polynomial-time

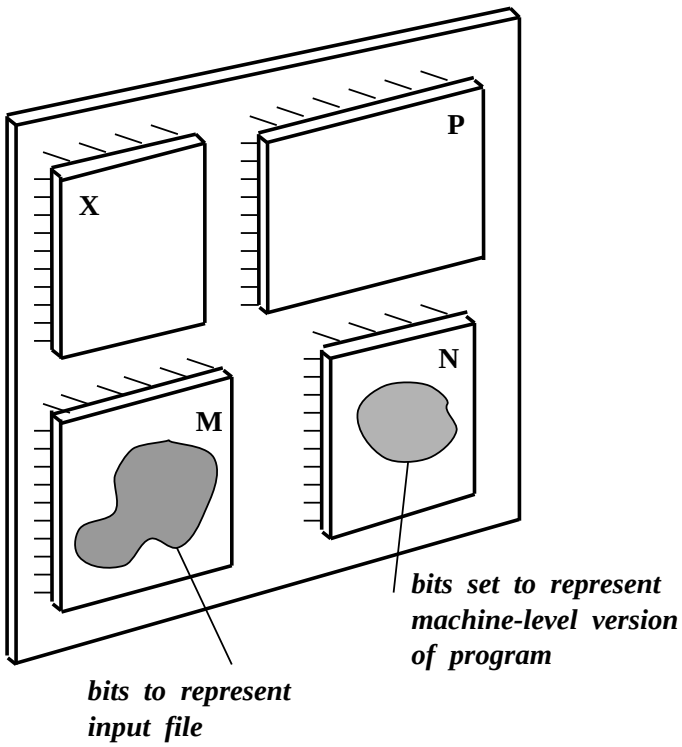


Figure 4.5: A simple computer, just before executing a program with an input file

Thus, at the risk of being repetitive, SATISFIABILITY of Boolean expressions is the problem, given a Boolean expression which may contain constants, of deciding whether or not it is possible to assign values to the variables such that the value of the whole expression becomes *true*.

CIRCUIT SATISFIABILITY is the problem, given a circuit with some constant values given to some wires at some points in time, of deciding whether or not it is possible to assign values to all the wires at all times such that all gates operate correctly at all times.

If you issue a sequence of two commands like

```
cc my.c
a.out < infile
```

then the situation just before the execution of the compiled program is illustrated in Figure 4.5. The program is stored as some pattern of 0s and 1s on

some of the wires, and we might as well assume that the input file has been copied from disk and is also stored on some set of wires.[6]

"Compilation" into a circuit

What happens when you issue the statement

```
cc my.c
a.out < infile
```

to this circuit? The "compiling" process creates the machine-level version of the program `my.c` and stores it by setting some of the wires of the circuit to specific values.

What are we accomplishing with all of this? A transformation. Let's say the program `my.c` is such that it always produces a 0 or a 1 as an answer (or a "Yes" or "No"). Then the question

Does `my.c` on input `infile` produce a "Yes"?

is equivalent to a satisfiability question about our circuit. What question? With certain wires initially set to 0s and 1s to represent the program and the input file, and with the "output wire" set to 1 when the computation terminates, is that whole circuit satisfiable?

This transformation of a question about a program and input file into a satisfiability question — a transformation of software and data into hardware — is most of the proof of Cook's Theorem. In fact, what we have proven so far is that every problem in "P," i.e., every problem that can really (i.e., deterministically) be solved in polynomial time, can be transformed (by a polynomial-time transformation) into Circuit Satisfiability. That result is worthless, but our proof works equally well if we now allow nondeterminism in the program `my.c`.

Enter Nondeterminism

What happens to the preceding "compilation" process if the program `my.c` is nondeterministic? What if `my.c` had a statement

```
nondet {x = TRUE; x = FALSE;}
```

How would we compile this into our circuit? Much like an `if`-statement

```
if ...
    x = TRUE;
else
    x = FALSE;
```

except that we let the "choice" between the two options depend on a wire that is not the output of any gate. This is a wire which in the circuit satisfiability problem that is being constructed is simply not assigned a value. All we need is

[6] Otherwise we would need to draw a disk and turn our neat "circuit" satisfiability problem into a "circuit–and–disk" satisfiability problem.

one such wire. Then compiling a two-way nondeterministic branch is just like an `if`-statement with the choice being determined by the value of that wire at the time of the execution of the statement. □

4.4 More "NP-Complete" Problems

Definition 6 *A problem in NP which has the property that every problem in NP can be transformed into it by a polynomial-time transformation, is called* NP-complete.

Cook's Theorem states that SAT is NP-complete. In 1972, Richard Karp ([Kar72]) showed that other problems were NP-complete as well. VERTEX COVER was one of them.

Theorem 22 VERTEX COVER *is NP-complete.*

Proof It is easy to see that VERTEX COVER can be solved by the exhaustive search approach that costs only "nondeterministic polynomial time" — VERTEX COVER is in NP.

To show that VERTEX COVER is NP-complete we describe a polynomial transformation from SATISFIABILITY into VERTEX COVER. Since there is a polynomial-time transformation from any problem in NP to SATISFIABILITY this then implies that there is (a two-step) polynomial-time transformation from any problem in NP to VERTEX COVER.

This transformation is made easier if we transform only a special form of SATISFIABILITY known as "SATISFIABILITY of boolean expressions in 3-conjunctive normal form," or 3-SAT for short. Expressions of this form look like this:

$$(x \vee \neg y \vee z) \wedge (w \vee z \vee w) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (z \vee \neg w \vee \neg z)$$

They are conjunctions (ANDs) of disjunctions (ORs) of three "literals" — variables or negated variables. (There is no requirement that literals in a clause have to be different.) Each 3-literal disjunction is called a "clause" of the expression.

For such a transformation of 3-SAT to VERTEX COVER to prove that VERTEX COVER is NP-complete we have to show that 3-SAT is NP-complete. We do this by slightly modifying our original proof showing that SATISFIABILITY was NP-complete.

That proof showed that SATISFIABILITY of circuits was NP-complete. Turning circuits into boolean expressions results in one "clause" per gate, with all those clauses ANDed together. If we could turn each clause into 3-conjunctive normal form, we would have shown 3-SAT to be NP-complete.

To simplify matters we could build those circuits out of just one type of gate. Let's choose NAND-gates. (NOR-gates would have been the other possibility.) Each gate then turns into a clause of the form

$$(x \vee \neg y \vee z) \wedge (x \vee z \vee w) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee w \vee \neg z)$$

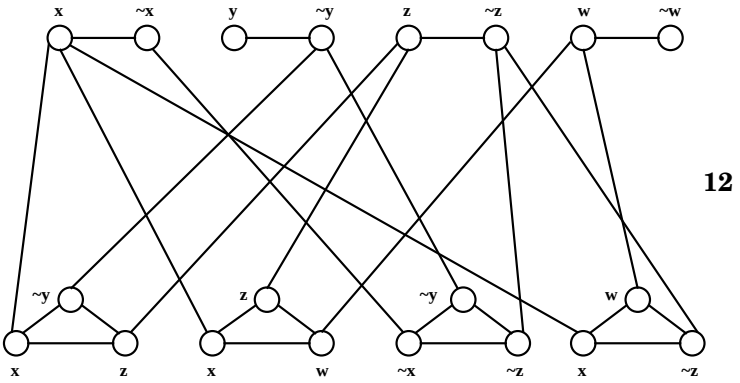


Figure 4.6: Translating 3-SAT into VERTEX COVER

$$(\neg(x_t \wedge y_t) \Leftrightarrow z_{t+1})$$

which we could split into

$$(\neg(x_t \wedge y_t) \Rightarrow z_{t+1}) \wedge (z_{t+1} \Rightarrow \neg(x_t \wedge y_t))$$

and further rewrite into

$$((x_t \wedge y_t) \vee z_{t+1}) \wedge (\neg z_{t+1} \vee \neg(x_t \wedge y_t))$$

One more rewriting get us close to the right form:

$$(x_t \vee z_{t+1}) \wedge (y_t \vee z_{t+1}) \wedge (\neg z_{t+1} \vee \neg x_t \vee \neg y_t)$$

The rewriting into 3-SAT is complete if we repeat a literal in those clauses that are short of having three literals:

$$(x_t \vee z_{t+1} \vee z_{t+1}) \wedge (y_t \vee z_{t+1} \vee z_{t+1}) \wedge (\neg z_{t+1} \vee \neg x_t \vee \neg y_t)$$

This proves

Lemma 5 3-SAT *is NP-complete.*

What's left to do in order to show that VERTEX COVER is NP-complete is to transform 3-SAT into VERTEX COVER. As an example, consider the expression

$$(x \vee \neg y \vee z) \wedge (x \vee z \vee w) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee w \vee \neg z)$$

The translation algorithm is as follows. (See Figure 4.6 for an illustration.) For each variable in the expression there is an edge in the graph with one of its endpoints labeled by that variable and the other by the negation of the variable. (These "variable edges" are at the top of the graph in Figure 4.6.) For each clause in the expression there is a "triangle" in the graph with its edges labeled by the three literals in the clause. (These "clause triangles" are at the bottom of the graph in Figure 4.6.) Now edges get drawn between the variable edges and the clause triangles, but only between nodes that have the same label. Finally, for this to be a "Yes/No" VERTEX COVER problem, there needs to be a number — the number of nodes about which we ask whether or not there is a vertex cover of that size. This number is equal to the number of different variables in the expression plus twice the number of clauses in the expression. In the example, this number is 12.

To explore the connection between a cover of this size for the graph and a satisfying assignment of truth values for the expression, let's build up some intuition about the matter by trying first to find a vertex cover for the graph. How would we go about that?

One might be tempted to think about some algorithm for this. That's not a good idea, since no sub-exponential algorithms are known — and it is conjectured that none exist. So just "eyeball" the given example. It is probably small enough to do that with success.

Heuristics may help. For example, to cover a triangle, any two of its three corners suffice, but a single corner does not. So we might as well set aside two of our 12 choices of nodes for each of the triangles. And to cover a variable edge, either one of its endpoints suffices. So we set aside one of our 12 choices for each variable edge. No matter how we choose nodes within these set-aside allocations, the variable edges and the clause triangles will all be covered. The challenge is to make the remaining choices such that the edges than run *between* variable edges and clause triangles get covered as well.

Consider the $(w, \neg w)$ variable edge. w covers the only edge that $\neg w$ covers and then some. Thus there is no disadvantage in choosing w. This has further ramifications. The (w, w) edge down to the rightmost triangle is now covered and there is no disadvantage in choosing the other two corners of that triangle for covering it. The same applies to the second triangle from the left.

We are doing well so far. Two triangles are covered (which as said before is automatic) and, importantly, all the edges between those two triangles and variable edges are covered as well. Here is the secret for success: We have to choose endpoints of the variable edges such that for each triangle at least one of three edges from the triangle corners up to the variable edges is covered *from above*. (One can already smell the connection to the 3-SAT problem: To satisfy the expression, we have to choose values for the variables such that for each clause at least one of its literals is true.)

Choosing the $\neg y$ endpoint of the $(y, \neg y)$ edge then lets us take care of the remaining two triangles.

What happened to the $(x, \neg x)$ and the $(z, \neg z)$ edges? They are not covered yet, and they are the only edges that are not covered yet. But we have spent

only 10 of our 12 node choices so far and can choose either endpoint of those two edges.

Back to the question of how such a solution to the vertex cover problem implies that the expression is satisfiable. Choosing w over $\neg w$ initially in the graph corresponds to making the variable w in the expression *true*. (The reason why this was a no-loss choice in the graph translates, on the expression side, into the observation that $\neg w$ does not occur in any clause.) Choosing the $\neg y$ over the y node in the graph corresponds to making the variable y in the expression *false* and satisfies the remaining clauses. Being free to choose either endpoint of the $(x, \neg x)$ and the $(z, \neg z)$ edges to complete the cover of the graph corresponds to the fact that the expression is already satisfied no matter what we assign to x and z.

A similar discussion illustrates how a satisfying assignment to the variables of the expression translates into a cover of the right size for the graph. Making a variable x true (false) in the expression translates into choosing the x ($\neg x$) endpoint of the $(x, \neg x)$ variable edge of the graph. The rest of the cover then falls into place without any difficult choices. □

4.5 The "Holy Grail": P = NP?

In order not to lose sight of the big picture over all these technical details, let's summarize the results of this chapter and contrast them with the results of Chapter 1. In both chapters we deal with proving problems to be difficult. In Chapter 1, we ignored the importance of time and considered as "difficult" problems that could not be solved by any program running in a finite amount of time. In the current chapter, the notion of difficulty of a problem is that there is no program that solves the problem *in polynomial time*. Figures 4.7 and 4.8 illustrate the results in both of these two settings.

In both settings, there are problems that are known to be "easy" — solvable (ODDLENGTH, BINARY ADDITION, SATISFIABILITY) or solvable in polynomial time (ODDLENGTH, BINARY ADDITION).

In both settings there are transformations that turn one problem into another. In both settings these transformations "propagate difficulty" — if A can easily be transformed into B and A is difficult, then B is difficult as well. In Chapter 1, for "difficulty," i.e., unsolvability, to propagate, the transformations merely had to be computable[7]. In the current chapter, for "difficulty," i.e., the lack of a polynomial-time solution, to propagate, the transformations had to be computable in polynomial time.

So far, so similar. But there is one huge difference. In Chapter 1, we did have a method to prove that one of the problems (SELFLOOPING) was indeed difficult. That method was diagonalization. With this starting point of one problem provably difficult, the transformations take over and prove all sorts of other problems to be difficult as well.

[7] All the examples were actually computable in polynomial time.

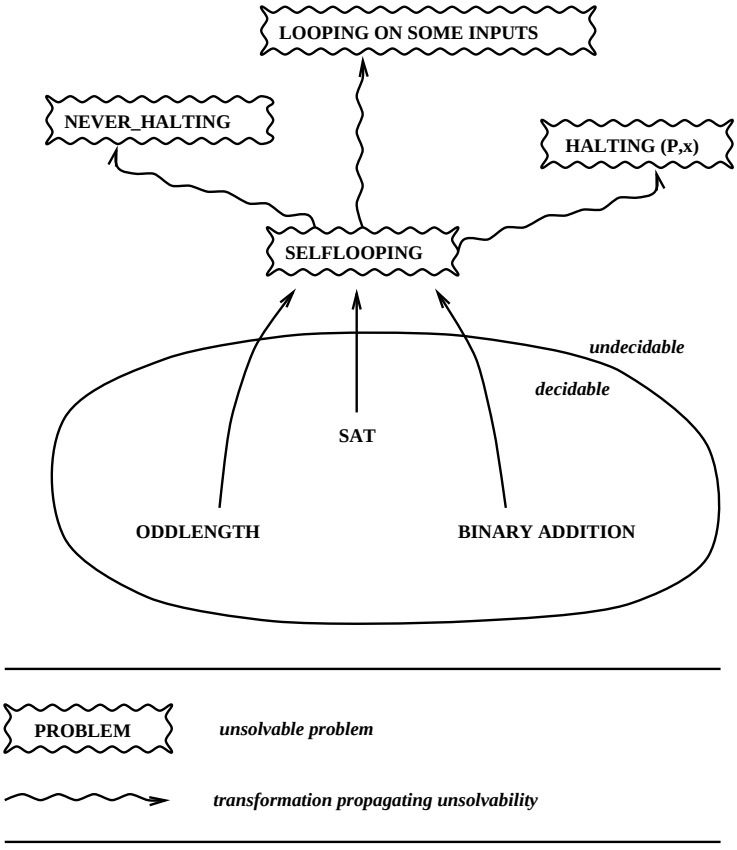


Figure 4.7: Results from Chapter 1

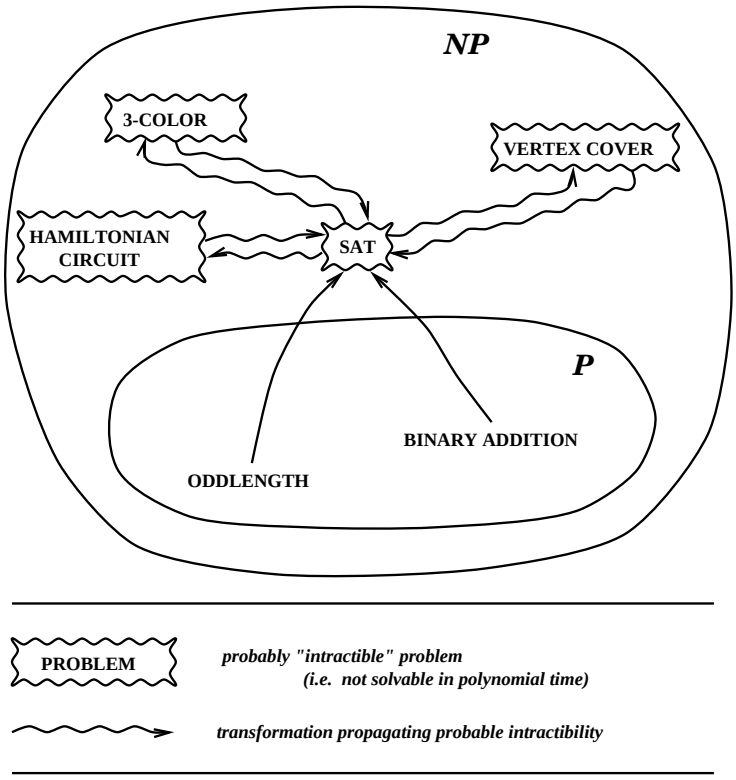


Figure 4.8: Results from the current chapter

In the current chapter, there is no such starting point. We do not have a proof that SAT*, or* VERTEX COVER*, or* 3-COLOR*, or* HAMILTONIAN CIRCUIT*, do indeed require more than polynomial running time to be solved.*

So what's missing in the current chapter is a proof technique — some way to show that, say, SAT cannot be solved in polynomial time. Such a result would show that P $\neq$ NP. Conversely, finding a polynomial-time algorithm for SAT because of the NP-completeness of SAT would imply that P = NP.[8]

Why is P $\stackrel{?}{=}$ NP So Hard to Resolve?

The question of whether or not there are polynomial-time algorithms for SAT, VERTEX COVER, or — equivalently — any other NP-complete problem has been the main unresolved problem in theory of computation for twenty years. While most people conjecture that there are no such polynomial-time algorithms, in other words P $\neq$ NP, there is no proof. What then makes this question so hard?

The sensitivity of the question

One reason for the difficulty is that there are models of computation that are very close to the standard model — very close to the kind of programs you write all the time — in which P = NP and, in a different model, P $\neq$ NP. In other words, a slight change in our concept of what a computer is (or, because the distinction between hardware and software is of no consequence here, in the definition of what a program is) changes the answer to the P $\stackrel{?}{=}$ NP question.

Why is this a reason that resolving the P $\stackrel{?}{=}$ NP question might be hard? Because it implies that any proof technique that can resolve the P $\stackrel{?}{=}$ NP question must be sensitive to those slight changes. It must work in the standard model of computation but must not work in a closely related model. Worse yet, of the two obvious approaches for trying to resolve P $\stackrel{?}{=}$ NP question — one approach for showing equality, the other for showing inequality — neither is sensitive to those slight changes in the model of computation.

Co-processors

What are these "slight changes" in the model? In terms of hardware, the change is to plug in a "co-processor" — a circuit, or chip, which takes some input and produces some output. The most common co-processor in practice may be a "numerical co-processor" — a chip that takes as input two floating-point numbers and produces, say, their product. The co-processors we are interested in here take as input one bitstring and produce one bit as output. Each such co-processor can be thought of as implementing a set of strings. For each string it responds with a "Yes" or a "No." Figure 4.9 illustrates this. In terms of software, this amounts to stipulating that there is a new boolean function built into our programming language which takes a bitstring as argument and that we do not count the execution time of that function as part of our programs' execution time.

What happens to our computer (or programming language) once we add this new feature? Programs that do not access that co-processor (do not call that new function) run like they always did. What about programs that do

[8] Current betting is overwhelmingly against this resolution of the question.

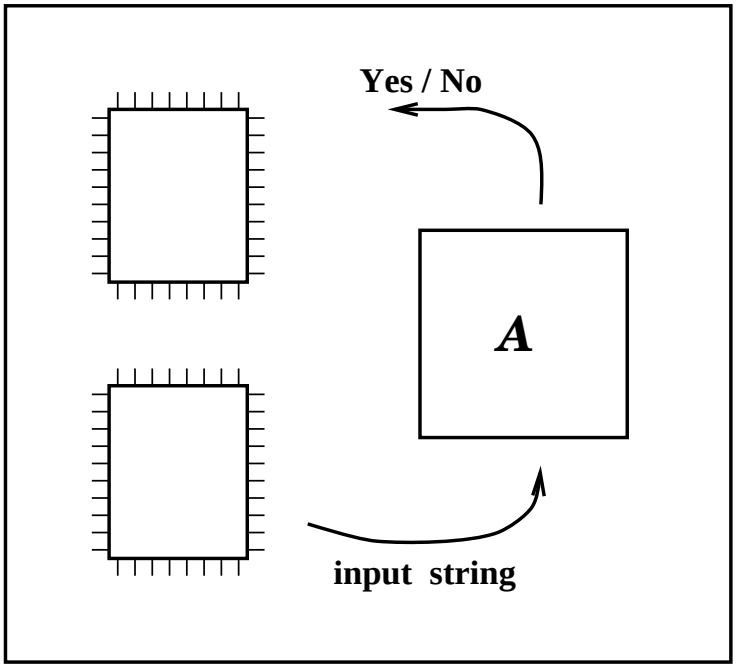


Figure 4.9: A co-processor

access the new feature? What they do depends on exactly which model of a co-processor we plugged in, i.e., which set it implements.

As long as that new co-processor only does things we could have done by programming our own boolean function it is not very interesting. But what if somehow, mysteriously, we had come across a co-processor that does in one single step something that we might otherwise not be able to do at all with a program or at least not as fast?

Co-processors that are "out of this world"

We can consider co-processors of any kind of power, e.g. even a co-processor that solves HALTING in a single leaping bound: a co-processor which, once you have fed it a bitstring that describes a program and input, responds with a "Yes," if that program on that input would terminate, and with a "No" otherwise.

Does such a co-processor exist? Definitely not on this planet, but that need not keep us from considering what such a new — very powerful — machine could do.[9]

Another co-processor of interest might be one that solves SATISFIABILITY in a single step.

P_A and NP_A

For any co-processor A, let P_A be the class of problems that a machine with co-processor A can solve in polynomial time. For any co-processor A that is worth talking about, P_A is bigger than P. This is certainly true if $A = \text{HALTING}$ and probably true if $A = \text{SAT}$.

Similarly, let NP_A be the class of problems that a machine with co-processor A can solve in nondeterministic polynomial time. NP_A can be bigger than NP. This is certainly true if $A = \text{HALTING}$ and probably true if $A = \text{SAT}$.

Theorem 23 (Baker, Gill, Solovay 1976) *1. There is a co-processor*[10] *A such that*

$$P_A = NP_A.$$

2. There is a co-processor B such that

$$P_B \neq NP_B.$$

(A proof can be found in *Hopcroft and Ullman*. A co-processor A for which $\mathrm{P}_A = \mathrm{NP}_A$ is the one that answers questions of the form, "Is it true that

there is a u such that for all v ... there is a z such that $E(u, v, \ldots y, z)$?"

where E is a boolean expression. This kind of problem — "quantified Boolean expressions" — is a generalization of SAT, which has only one "quantifier." ("There is a set of values that makes the expression true."))

Proof techniques can be "too powerful"

Think of the result of Baker et al as saying that in one set of circumstances

[9] This is a form of transformation. We are asking what problems we could solve if we had a way to solve HALTING.

[10] A set of bitstrings, really. The commonly used term is actually "oracle."

("on planet A") P = NP and in a different set of circumstances ("on planet B") P $\neq$ NP. Why do we on planet Earth care? Because if we want to prove that P $\neq$ NP, we have no chance unless we come up with a proof technique that does not work on planet A. Or, if space travel does not seem the right subject to drag into this discussion, we have no chance unless we come up with a proof technique that will collapse – will not be applicable, will not work — once we plug in co-processor A. Any technique that is so robust that it even works in the presence of any co-processor is "too powerful" to prove P $\neq$ NP. Similarly, any technique that is so robust that it even works in the presence of any co-processor, especially co-processor B from the theorem, is "too powerful" to prove P = NP.

Are we talking about some far-out or nonexisting proof techniques? Not at all. If we want to prove that SAT cannot be solved in polynomial time, what would be a natural first technique to try? What technique do we have to show that something cannot be done? The one and only method we have is diagonalization. How could we try to use it for proving that something takes, say, exponential time to solve?

Time-limited diagonalization

Consider the following "exponential-time" version of HALTING. Given a program P and an input x, decide whether or not P halts within $2^{|x|}$ steps. This is clearly a decidable problem — just run the program for that long and see. Is there a faster way to decide it? No, by a diagonalization proof that is practically identical to the one we used to prove that the (unlimited) halting problem was undecidable. The central argument in such a proof is just like our argument in Chapter 1 that SELFLOOPING was not decidable. There the argument was that if SELFLOOPING was decidable then we could write a program that took itself as input, decided what its input (that is, itself) will do (loop or not) and based on that decision do the opposite. Now, with time playing a role, the argument is that if it was possible to decide in substantially less than 2^n steps that a program was going to halt within that many steps then we could write a program that took itself as input, decided what its input (that is, itself) will do (run for more than 2^n steps or not) and based on that decision do the opposite.

This is of course very exciting. We have a way to prove that something takes exponential time! All that's left to do in order to prove that P $\neq$ NP is to transform this EXPONENTIAL-TIME HALTING problem to SAT.[11] Are we likely to succeed? Not if this approach would also work "on other planets," i.e., in the presence of co-processors. Well, does it? Does an argument like

> "if it was possible to decide in substantially less than 2^n steps that a program was going to halt within that many steps then we could write a program that took itself as input, decided what its input (that is, itself) will do (run for more than 2^n steps or not) and based on that decision do the opposite"

[11] with a polynomial-time transformation, but that should not worry us as we have yet to come up with any transformation that was not polynomial-time

still work when there is a co-processor present? Unfortunately, the answer is yes. There is nothing wrong with the same argument on machines that have a co-processor:

> "if it was possible *on a machine with co-processor A* to decide in substantially less than 2^n steps that a program was going to halt within that many steps then we could write a program *to run on a machine with co-processor A* that took itself as input, decided what its input (that is, itself) will do (run for more than 2^n steps or not) and based on that decision do the opposite"

Thus this diagonalization technique is doomed. It is "too good."

Well, then maybe, just maybe, P = NP ? One reason why a conjecture can be hard, actually impossible, to prove is that the conjecture might not be true. We have used this insight before. If we had a hard time coming up with a context-free grammar for some language, a reasonable alternative was to consider the possibility that the language was not context-free and to try to prove that instead.

Whatever the reason for entertaining the conjecture that P = NP, we should be aware that "extra-terrestrial" arguments similar to the ones presented above show that P = NP is not likely to be provable with "the obvious" kind of approach.

What is the "obvious" kind of approach? What is the problem?

The problem here would be to find a way to solve SAT and its equivalent problems with a deterministic polynomial-time algorithm. How could we even approach this problem? By trying to devise some very clever way of searching nondeterministic execution trees of polynomial height but exponential size in (deterministic) polynomial time.

But if we succeeded in such an efficient deterministic "simulation" of nondeterminism, why should such a simulation not work just because a machine might have a co-processor? Why should this not work "on planet B." If it does, of course, the technique is again "too good."

Now what? That's where the matter stands. We are not able to settle the question of whether or not P = NP and variations of existing proof techniques are not likely to help. We are able to build and program computers but we cannot claim to understand the nature of computation.

Exercises

Ex. 4.1. Is it true that if a circuit is satisfiable then there is more than one way to satisfy it? Explain briefly.

Ex. 4.2. Is it true that if a graph is 3-Colorable then there is more than one way to 3-Color it? Explain briefly.

Ex. 4.3. Consider the following attempt at transforming 3-SAT to 2-SAT. Replace each clause

$$(A \vee B \vee C)$$

where A, B, C are, as usual, variables or negated variables, by

$$(A \vee B) \wedge (B \vee C) \wedge (A \vee C)$$

What is wrong with this "transformation"?

Ex. 4.4. For each of the five statements below, state whether it is `TRUE` or `FALSE`. Don't explain your answer.

1. If A and B are two problems in NP, A is NP-complete, and there is a polynomial-time transformation from A to B then B is NP-complete as well.
2. If A and B are two problems in NP, A is NP-complete, and there is a polynomial-time transformation from B to A then B is NP-complete as well.
3. If A and B are two NP-complete problems then there is a polynomial-time transformation from A to B and also a polynomial-time transformation from B to A.
4. If A is a problem that cannot be solved in polynomial time and there is a polynomial-time transformation from A to B then B cannot be solved in polynomial time either.
5. Any problem A that can be transformed into CIRCUIT SATISFIABILITY by a polynomial-time transformation is NP-complete.

Ex. 4.5. Which of the following statements are true?
("Polynomial-time" is understood to mean "deterministic polynomial-time.")

1. No polynomial-time algorithm for VERTEX COVER is known.
2. No polynomial-time algorithm for GRAPH 3-COLORING is known.
3. No polynomial-time algorithm for GRAPH 2-COLORING is known.
4. If a polynomial-time algorithm exists for SAT then polynomial-time algorithms exist for all problems in NP.
5. There is a polynomial-time transformation from 3-SAT to VERTEX COVER.
6. There is a polynomial-time transformation from VERTEX COVER to 3-SAT.

7. If a polynomial-time algorithm exists for VERTEX COVER then polynomial-time algorithms exist for all problems in NP.
8. If A and B are two problems in NP, if we know that B is NP-complete, and we know that there is a polynomial-time transformation of A into B, then we know that A is NP-complete as well.
9. VERTEX COVER is known to be NP-complete.

Ex. 4.6. Describe a transformation from HAMILTONIAN PATH to HAMILTONIAN CIRCUIT.

For reference and/or clarification note that ...

1. HAMILTONIAN PATH is the problem of deciding, given a graph G and two nodes a and b in G, whether or not there exists a path from a to b that touches each node in the graph exactly once.
2. HAMILTONIAN CIRCUIT is the problem of deciding, given a graph G, whether or not there exists a loop that touches each node in the graph exactly once.
3. To solve this problem, you need to show how to construct, given a graph G and two nodes a and b in G, a new graph G' such that there is a Hamiltonian Path in G from a to b if and only if G' has a Hamiltonian Circuit.

Ex. 4.7. Given a polynomial-time solution to the *decision* version of SATISFIABILITY, i.e. given a Boolean function that tells you whether or not a circuit you pass to it is satisfiable, how could you program a polynomial-time solution to the full version of the problem? (The full version of the problem means that if a satisfying assignment exists you have to find one, not just say that one exists.)

Ex. 4.8. *(a)* Apply the transformation shown in class from 3SAT to VERTEX COVER to this expression:

$$(x \vee \neg y \vee z) \wedge (x \vee y \vee w)$$

(b) If in a Boolean expression in 3-CNF the same variable shows up unnegated in all clauses, what property of the graph does that translate into (again, in the transformation discussed in class)?

(c) Still about the same transformation, if there are 10 solutions to the vertex cover problem, does this imply that there are 10 ways to satisfy the expression? Explain briefly.

Ex. 4.9. Describe a transformation of 4-SAT into VERTEX COVER. (You can do this by an example if you like.)

Bibliography

[BB88] G. Brassard and P. Bratley. *Algorithmics: Theory and Practice*. Prentice–Hall, 1988.
`QAA9.6.B73.`
Includes chapters on greedy algorithms, divide and conquer, dynamic programming, graph search.

[Bro89] J. G. Brookshear. *Theory of Computation*. Benjamin/Cummings, 1989.
`QAA 267.B76.`

[CLR90] T. H. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press / McGraw–Hill, 1990.
`QAA 76.6.C662.`
Comprehensive and readable. A standard reference.

[Coo71] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd Ann. ACM Symposium on Theory of Computing*, pages 151–158, New York, 1971. Assoc. for Computing Machinery.

[Dea96] N. Dean. *The Essence of Discrete Mathematics*. Addision–Wesley, 1996.

[GKP89] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison–Wesley, 1989.
`QAA 39.2.G733.`
Well written.

[Gur89] E. M. Gurari. *An Introduction to the Theory of Computation*. Computer Science Press, 1989.
`QAA 76.G828.`
Presents the standard hierarchy of languages and machines ("Chomsky's hierarchy") in terms of programs, restricted in various ways. Quite formal.

[Kar72] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations* (R. E. Miller and J. W. Thatcher, Eds.), pages 85–104, New York, 1972. Plenum Press.

[LP98] H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 2 edition, 1998.
QAA 267.L49.

[Man89] U. Manber. *Introduction to Algorithms. A Creative Approach*. Addison–Wesley, 1989.
QAA.9.D35M36.

[Mar91] J. C. Martin. *Introduction to Languages and the Theory of Computation*. McGraw–Hill, 1991.
QA 267.5.S4M29.

[MR91] S. B. Maurer and A. Ralston. *Discrete Algorithmic Mathematics*. Addison–Wesley, 1991.
QAA 39.2.M394.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, 1982.

[Raw92] G. J. E. Rawlins. *Compared to What? An Introduction to the Analysis of Algorithms*. Computer Science Press, 1992.
QAA 76.9.A43R39.

[Sch96] C. Schumacher. *Chapter Zero: Fundamental Notions of Abstract Mathematics*. Addison–Wesley, 1996.

[Woo87] D. Wood. *Theory of computation*. Harper & Row, 1987.
QAA 267.W66.
Standard textbook, comprehensive, formal.