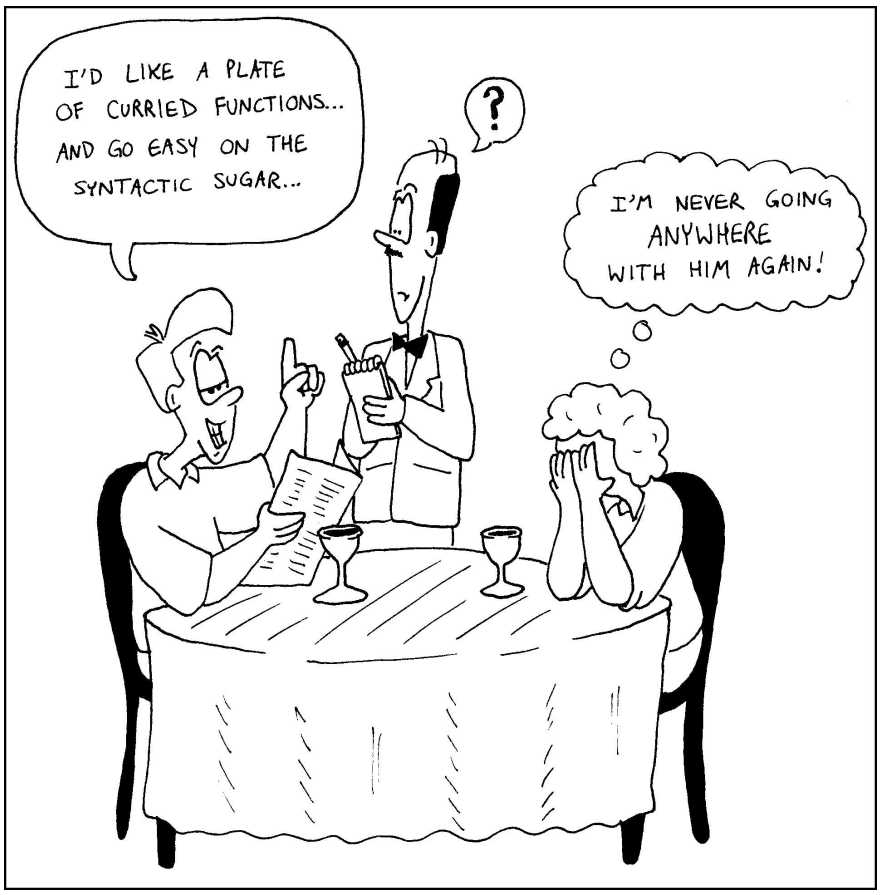
2.38. Repeat Exercise 2.37 for partial functions with signature $Bool^k \rightarrow Bool$. View each partial function as a table with the output field being either a Boolean or the special symbol $\perp$, standing for *undefined.*



3

Cardinalities and Diagonalization

In this chapter, we discuss the important idea of measuring sizes of infinite sets. In addition to helping reinforce many mathematical concepts, we obtain a better appreciation of the work of pioneers, notably George Cantor, who originated many of the fundamental ideas in this area. We will employ many of the ideas found in this chapter in later chapters to argue the existence of *non-computable* functions and certain languages called *non Turing-recognizable*—languages for which the membership test (testing whether an arbitrary string is a member of the language)—*cannot* be performed by any machine.

3.1 Cardinality Basics

The *cardinality* of a set is its size. The cardinality of a finite set is measured using natural numbers; for example, the size of $\{1, 4\}$ is 2. How do we "measure" the size of infinite sets? The answer is that we use "funny numbers," called *cardinal numbers*. The smallest cardinal number is $\aleph_0$, the next larger cardinal number is $\aleph_1$, and so on. If one infinite set has size $\aleph_0$, while a second has size $\aleph_1$, we will say that the second is larger than the first, even though both sets are infinite. Moreover, $\aleph_0$ is the number of elements of Nat, while $\aleph_1$ is the number of elements of *Real*. All these ideas will be made clear in this section.

To understand that there could be "smaller" infinities and "bigger" infinities, think of two infinitely sized dogs, Fifi and Howard. While Fifi is infinitely sized, every finite patch of her skin has a finite amount of hair. This means that if one tries to push apart the hair on Fifi's back, they will eventually find two adjacent hairs between which there is no other hair. Howard is not only huge - every finite patch of his skin has *an infinite* amount of hair! This means that if one tries to push apart

the hair on Howard's back, they will never find two hairs that are truly adjacent. In other words, *there will be a hair between every pair of hairs*! This can happen if Fifi has $\aleph_0$ amount of hair on her entire body while Howard has $\aleph_1$ amount of hair on his body.[1] Real numbers are akin to hair on Howard's body; there is a real number that lies properly between any two given real numbers. Natural numbers are akin to hair on Fifi's body; there is no natural number between adjacent natural numbers.

We begin with the question of how one "counts" the number of elements in an infinite set. For example, are there the same "number" of natural numbers as there are real numbers? Since we cannot count infinite sets, let us adopt a method that our ancestors sometimes used when they could not count certain *finite* sets.[2] Our ancestors used to conduct trade successfully through the *barter* system without actually counting the number of objects; say, a cabbage for an elephant, and so on.[3] The real idea behind barter is to establish a *bijection* or *correspondence* between two sets of elements without actually counting them. The same technique works quite well when we have to count the contents of infinite sets; in fact, that is the *only technique* that works! But what does *'counting'*, *'countable'*, etc., mean?

3.1.1 Countable sets

A set S is said to be *countable* if there is a 1-1 total mapping from it to natural numbers. (This mapping *need not* be onto). Clearly, finite sets are countable. Consider the infinite set Odd, the set of odd numbers. Since there is a total 1-1 mapping $\lambda x.(x-1)/2$ from Odd numbers to Nat, Odd is countable. The set $Real$ is not countable, as we shall show in this chapter.

3.1.2 Cardinal numbers

We now discuss the use of cardinal numbers more precisely. The cardinality of Nat is *defined* to be $\aleph_0$, written $|Nat| = \aleph_0$. Two sets A and B *have the same cardinality* if *there is a bijection from A to B.* Function $\lambda x.(x-1)/2$ actually serves as a bijection from Odd numbers to Nat. To sum up,

[1] Hope this wouldn't be viewed as splitting hairs...

[2] It is good that Romans didn't discover the concept of Avogadro's number - how could they have carved it out on stone tablets?

[3] Cabbages with magical powers, perhaps.

- Odd is countable,
- $|Odd| = |Even| = |Nat| = \aleph_0$,
- But, note that $Odd \subset Nat$ and $Even \subset Nat$;
- Therefore, it is entirely possible that for two sets A and B, $A \subset B$, and yet $|A| = |B|$.

The above example demonstrates that one cannot determine the cardinality of sets purely based on subset relationships. One correct (and handy) method for using subset relationships to determine the cardinality of sets is using *cardinality trap*s.

3.1.3 Cardinality "trap"

To motivate the notion of cardinality trap, consider the question, "how many points are there in the map of mainland USA?" Let us treat this map as a region of $Real \times Real$. The theorem which we call *cardinality trap* says:

> If, for three sets A, B, and C, we have $|A| = |C|$ and $A \subset B \subset C$, then $|A| = |B| = |C|$.

Specifically, cardinality trap allows one to "trap" the cardinality of a set B to be between those of two sets A and C. Exercise 3.12 asks you to prove that cardinality trap is a simple corollary of the famous *Schröder-Bernstein theorem.* For the question at hand,

- Any given map of the USA (set B of points) can be properly inscribed within a (larger) square (set C of points).
- Within the given map of the USA, one can properly inscribe a (smaller) square (set A of points).
- All squares in $Real \times Real$ have the same number of points, $\aleph_1$ (this is a result we shall prove later ($|A| = |C|$).
- Therefore, $|A| = |B| = |C|$, or the map of the USA, however drawn, has the same number of points as in a square, namely $\aleph_1$.

Now, we present one of George Cantor's central results, which allows us to prove that two sets have different cardinalities. Known as *the diagonalization method*, it is basically a specific application of the principle of *proof by contradiction*.

3.2 The Diagonalization Method

Let us return to our original question, "is there a bijection from Nat to $Real$?" The answer is *no* and we proceed to show how. We follow

the *powerful* approach, developed by Cantor, called *diagonalization.* Diagonalization is a particular application of the principle of *proof by contradiction* or *reductio ad absurdum* in which the solution-space is portrayed as a square matrix, and the contradiction is observed along the diagonal of this matrix. We now walk you through the proof, providing section headings to the specific steps to be performed along the way.

Most textbooks prove this result using numbers represented in decimal, which is much easier than what we are going to present in this section - namely, prove it in binary. We leave the proof in decimal as an exercise for you. In addition to being a 'fresh,' as well as illuminating proof, a proof for the binary case also allows us to easily relate cardinality of *Real*s to that of languages over some alphabet. Here, then, are the steps in this proof.

3.2.1 Simplify the set in question

We first simplify our problem as follows. Note that $(\lambda x.1/(1+x))$ is a bijection from $[0,\infty] \subset \mathit{Real}$ to $[0,1] \subset \mathit{Real}$. Given this, it suffices to show that there is *no* bijection from Nat to $[0,1] \subset \mathit{Real}$, since bijections are closed under composition. We do this because the interval $[0,1]$ is "easier to work with." We can use *binary fractions* to capture each number in this range, and this will make our proof convenient to present.

3.2.2 Avoid dual representations for numbers

The next difficulty we face is that certain numbers have *two* fractional representations. As a simple example, if the manufacturer of Ivory soap claims that their soap is 99.99% pure, it is not the same as saying it is 99.999% pure.[4] However, if they claim it is $99.9\overline{9}$% pure (meaning an infinite number of 9s following the fractional point), then it is equivalent to saying it is 100% pure. Therefore, in the decimal system, infinitely repeating 9s can be represented without infinitely repeating 9s. As another example, $5.123\overline{9} = 5.124$. The same 'dual representations' exist in the binary system also. For example, in the binary system, the fraction $0.010\overline{0}$ (meaning, 0.010 followed by an infinite number of 0s) represents 0.25 in decimal. However, the fraction $0.010\overline{1}$ (0.010 followed by an infinite number of 1s) represents $0.011\overline{0}$ in binary, or 0.375 in decimal. Since we would like to avoid dual representations, we will avoid dealing

[4] Such Ivory soap may still float.

with number 1.0 (which has the dual representation of $0.\overline{1}$). Hence, we will perform our proof by showing that there is no bijection from Nat to $[0,1) \subset Real$. This would be an even stronger result.

Let us represent each real number in the set $[0,1) \subset Real$ in binary. For example, 0.5 would be $0.100\ldots$, 0.375 would be $0.01100\ldots$. We shall continue to adhere to our convention that *we shall never use any bit-representation involving* $\overline{1}$. Fortunately, every number in $[0,1)$ can be represented without ever using $\overline{1}$. (This, again, is the reason for leaving out 1.0, as we don't wish to represent it as $0.\overline{1}$, or 1.0).

3.2.3 Claiming a bijection, and refuting it

For the simplicity of exposition, we first present a proof that is "nearly right," and much simpler than the actual proof. In the next section, we repair this proof, giving us the actual proof. Suppose there is a bijection f that puts Nat and $[0,1)$ in correspondence C1 as follows:

$0 \rightarrow .b_{00}b_{01}b_{02}b_{03}\ldots$
$1 \rightarrow .b_{10}b_{11}b_{12}b_{13}\ldots$
$\ldots$
$n \rightarrow .b_{n0}b_{n1}b_{n2}b_{n3}\ldots$
$\ldots$

where each b_{ij} is 0 or 1.

Now, consider the real number

$$D = 0.\neg b_{00}\ \neg b_{11}\ \neg b_{22}\ \neg b_{33}\ldots.$$

This number is *not* in the above listing, because it differs from the i-th number in bit-position b_{ii} **for every** i. Since this number D is not represented, f cannot be a bijection as claimed. Hence such an f does not exist.

3.2.4 'Fixing' the proof a little bit

Actually the above proof needs a small "fix"; *what if the complement of the diagonal happens to involve a* $\overline{1}$*?* The danger then is that we *cannot* claim that a number equal to the complemented diagonal does not appear in our listing. It might then end up existing in our listing of Reals in a "non $\overline{1}$ form."

We overcome this problem through a simple correction.[5] This correction ensures that the complemented diagonal will never contain a

[5] Exercise 3.6 asks you to propose an alternative correction.

$\overline{1}$. In fact, we arrange things so that the complemented diagonal will contain zeros infinitely often. This is achieved by placing a 1 in the uncomplemented diagonal every so often; we choose to do so for all *even* positions, by listing the *Real* number $.1^{2n+1}\overline{0}\ldots$ ($2n+1$ 1s followed by $\overline{0}$) at position $2n$, for all n. Consider the following correspondence, for example:

$0 \rightarrow .1\overline{0}$
$1 \rightarrow .c_{00}c_{01}c_{02}c_{03}\ldots$
$2 \rightarrow .111\overline{0}$
$3 \rightarrow .c_{10}c_{11}c_{12}c_{13}\ldots$
$4 \rightarrow .11111\overline{0}$
$5 \rightarrow .c_{20}c_{21}c_{22}c_{23}\ldots$
$6 \rightarrow .1111111\overline{0}$
$\ldots$
$2n \rightarrow .1^{2n+1}\overline{0}\ldots$
$2n+1 \rightarrow .c_{n0}c_{n1}c_{n2}c_{n3}\ldots$
$\ldots$

Call this correspondence C2. We obtain C2 as follows. We know that the numbers $.1\overline{0}$, $.111\overline{0}$, $.11111\overline{0}$, etc., exist in the original correspondence C1. C2 is obtained from C1 by first permuting it so that the above elements are moved to the *even positions* within C2 (they may exist arbitrarily scattered or grouped, within C1). We then go through C1, strike out the above-listed elements, and list its remaining elements in the odd positions within C2. We represent C2 using rows of $.c_{ij}$, as above.

We can now finish our argument as follows. The complemented diagonal doesn't contain a $\overline{1}$, because it contains 0 occurring in it infinitely often. Now, this complemented diagonal cannot exist anywhere in our $.c_{ij}$ listing. The complemented diagonal is certainly a Real number missed by the original correspondence C1 (and hence, also missed by C2). Hence, we arrive at a contradiction that we have a *correspondence*, and therefore, we cannot assign the same cardinal number to the set $[0,1) \subseteq Real$. It is therefore of higher cardinality.

The conclusion we draw from the above proof is that *Real* and Nat have different cardinalities. Are there any cardinalities "in between" that of *Real* and Nat? Loosely speaking, "is there a $\aleph_{0.5}$?!" The hypothesis that states "no there isn't a cardinality between $\aleph_0$ and $\aleph_1$," or in other words, "there isn't a $\aleph_{0.5}$," is known as the *Continuum Hypothesis*. It has been a problem of intense study over the last 120 years, and in fact is the *first* of Hilbert's 23 challenges to computer science [54].

These challenges helped spur considerable amounts of research in Computer Science, and contributed to much of the foundational knowledge of the subject area (e.g., as covered in this book). For further details, please see [26]. We shall use cardinality arguments when comparing the set of all functions and the set of all *computable* functions.

3.2.5 Cardinality of 2^{Nat} and $Nat \rightarrow Bool$

In this section, we argue that the sets 2^{Nat} (the powerset of Nat) and $Nat \rightarrow Bool$ (the set of functions from Nat to $Bool$) have the same cardinality as $Real$. Notice that each set within 2^{Nat} can be represented by an infinitely long characteristic sequence. For instance, the sequence $10010100\overline{0}$ represents the set $\{0, 3, 5\}$; the sequence $101010\ldots$ represents the set $Even$; the sequence $010101\ldots$ represents the set Odd; and so on. Notice that the very same characteristic sequences also represent functions from Nat to $Bool$. For instance, the sequence $10010100\overline{0}$ represents the function that maps 0, 3, and 5 to $true$, and the rest of Nat to $false$; the sequence $101010\ldots$ represents the function $\lambda x.even(x)$; and the sequence $010101\ldots$ represents the function $\lambda x.odd(x)$. Hence, the above two sets have the same cardinality as the set of all infinitely long bit-sequences. How many such sequences are there? By putting a "0." before each such sequence, it appears that we can define the $Real$s in the range $[0, 1]$. However, we face the difficulty caused by infinite 1s, i.e., we will end up having $\overline{1}$ occurring within an infinite number of infinite sequences. Therefore, we cannot directly use the arguments in Section 3.2.2, which rely on such numbers being absent from the listing under consideration.

We now present the Schröder-Bernstein Theorem which allows us to handle this, and other "hard-to-count" sets, very cleanly. We present the theorem and its applications in the next section.

3.3 The Schröder-Bernstein Theorem

Theorem 3.1. (Schröder-Bernstein Theorem): *For any two sets A and B, if there is a 1-1, total, and* into *map f going from A to B, and another 1-1, total, and* into *map g going from B to A, then these sets have the same cardinality.*

Section 3.3.3 discusses a proof of this theorem.

3.3.1 Application: cardinality of all C Programs

As our first application of the Schröder-Bernstein Theorem, let us arrive at the cardinality of the set of all C programs, CP. We show that this is $\aleph_0$ by finding 1-1, total, and *into* maps from Nat to CP and vice versa. The real beauty of this theorem is that we can find such maps *completely arbitrary*. For instance, we consider the class of C programs beginning with `main(){}`. This is, believe it or not, a *legal C program*! The next longer, such "weird but legal" C program, is `main(){;}`. The next ones are `main(){;;}`, `main(){;;;}`, `main(){;;;;}`, and so on! Now,

- A function $f : Nat \rightarrow CP$ that is 1-1, total, and into is the following:
  - Map 0 into the legal C program, `main(){}`
  - Map 1 into another legal C program `main(){;}`
  - Map 2 into another legal C program `main(){;;}`
  - ..., map i into the C program `main(){;`^{i}`}`—*i.e.*, one that contains i occurrences of `;`.
- A function $g : CP \rightarrow Nat$ that is 1-1, total, and into is the following: view each C program as a string of bits, and obtain the value of this bit-stream viewed as an unsigned binary number.

By virtue of the existence of the above functions f and g, from the Schröder-Bernstein Theorem, it follows that $|CP| = |Nat|$.

3.3.2 Application: functions in $Nat \rightarrow Bool$

We have already shown that such functions can be viewed as infinite bit-sequences, IBS. As already pointed out, we cannot interpret such sequences straightforwardly as Real numbers in $[0, 1)$ because of the presence of $\overline{1}$ that gives rise to multiple representations. We use the following alternative approach:

- We map every member of IBS that *does not contain* an occurrence of $\overline{1}$ into the range $[0, 1)$ by putting a "0." before it, and interpreting it as a Real number.
- We map every member of IBS that *contains* an occurrence of $\overline{1}$ into the range $[1, 2]$ by putting a "1." before it, interpreting it as a Real number in $[1, 2]$, and converting it into an equivalent form without $\overline{1}$. For example, $0.01010\overline{1}$ is mapped to $1.01011\overline{0}$.
- This is a 1-1, total, and *into* map from IBS to $[0, 2]$. Composing this map with the scale-factor $\lambda x.(x/2)$, we obtain the desired function f that goes from IBS into $[0, 1]$.

- Now, we obtain function g that is 1-1, total, and into, going from $[0, 1]$ to IBS as follows:
  - map every number except 1 in this range to the corresponding number in IBS (with a "0." before it) that does not contain $\overline{1}$.
  - map 1 to $0.\overline{1}$.

  This mapping hits every member of IBS except those containing $\overline{1}$ (the only exception being for 1). This is 1-1, total, and into.

From the Schröder-Bernstein Theorem, it follows that $|IBS| = |Real|$.

Illustration 3.3.1 *Using the Schröder-Bernstein Theorem, define a bijection between $Nat \times Int$ and Int.*
Solution: Using the SB theorem, we just need to find a 1-1 total into maps going from $Nat \times Int$ to Int and one going from Int to $Nat \times Int$. Here is the first map:

$$\lambda\langle x, y\rangle.sign(y) \times (2^x \times 3^{|y|}).$$

That this map is total is obvious because $\times$ is defined everywhere. Why is this 1-1? That's because of the unique prime decomposition of any number (note that we are not ignoring the sign). It is into because we can't generate numbers that are a multiple of 5, 7, etc.

The reverse map is much easier: just pair the Int with some arbitrary Nat:

$$\lambda x.\langle 0, x\rangle.$$

How about finding a bijection directly? It can be done as follows (but it will become apparent *how much harder* this is, compared to using the Schröder-Bernstein Theorem):

> List all $\langle Nat, Int\rangle$ pairs systematically, then list all Ints systematically against it.

Listing all of the former proceeds systematically as follows:

- All that "add up to 0 ignoring signs," with signs later attached in every possible way. $\langle 0, 0\rangle$ is the only pair that adds up to 0. Additionally, -0 is 0.
- All that "add up to 1 ignoring signs," with signs later attached in every possible way. $\langle 0, 1\rangle$ and $\langle 1, 0\rangle$ add up to 1. List them as follows:

  $\langle 0, 1\rangle$, $\langle 0, -1\rangle$, $\langle 1, 0\rangle$.
- All that "add up to 2 ignoring signs," with signs later attached in every possible way. $\langle 0, 2\rangle, \langle 1, 1\rangle$, and $\langle 2, 0\rangle$ add up to 2. List them as follows:

10.3.3 Minimizing DFA

The most important result with regard to DFA minimization is the *Myhill-Nerode Theorem.*

Theorem 10.1. *The result of minimizing DFAs is unique, up to isomorphism.*[1]

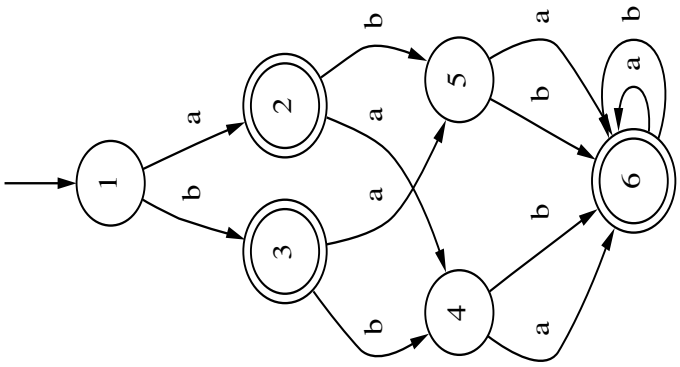
The theorem says that given two DFAs over the same alphabet that are language-equivalent, they will result in identical DFAs when minimized, up to the renaming of states. One very dramatic illustration of the Myhill-Nerode Theorem will be in Chapter 13, where it will be shown that *Binary Decision Diagrams* (BDDs)—an efficient data structure for Boolean functions—are minimized DFAs for certain finite languages of binary strings. These finite strings, in fact, encode the truth assignment for Boolean formulas according to certain conventions that will be explained in Chapter 13. Because of this uniqueness, equality testing between two Boolean functions can be reduced to pointer-equality in a representation of BDDs using hash tables. Another illustration is provided in Section 10.4. We now discuss the minimization algorithm itself.

The basic idea behind DFA state minimization is to consider all pairs of states systematically by constructing a table. For each pair of states, we consider all strings of length zero and up, and see if they can distinguish any pair of states. Initially, we distinguish all pairs of states $\langle p, q\rangle$ such that p is a final state and q is nonfinal. We enter an `x` in the table to record that these states are ε-distinguishable. In essence, at the beginning of the algorithm we are treating all final states as belonging to one equivalence class, and all non-final states as belonging to another.

Thereafter, in the ith iteration of table filling, we see if any of the state pairs $\langle p, q\rangle$ that are not yet distinguished have a move on some $a \in \Sigma$ such that they go to states $\langle p', q'\rangle$ that are distinguishable (at the end of the $i-1$st iteration); if so, distinguish $\langle p, q\rangle$. The algorithm stops when two iterations k and $k+1$ result in the same table.
Illustration: Consider the DFA in Figure 10.11 (from [71]) where the final states are 2, 3, 6, and $\Sigma = \{a, b\}$.

1. The initial blank table that permits all pairs of states to be compared is in Figure 10.11.

[1] The concept of isomorphism comes from graph theory. Two directed graphs G_1 and G_2 are isomorphic if there is a bijection b between their nodes that preserves the graph connectivity structure. In other words, if n_1 and n_2 are nodes of G_1 and G_2, respectively, and if the bijection relates n_1 and n_2, then the list of successors of n_1 in G_1 are also bijective with the list of successors of n_2 in G_2.




```
2 .
3 .  .
4 .  .  .
5 .  .  .  .
6 .  .  .  .  .
  1  2  3  4  5
```

```
2 x                2 x                2 x                2 x
3 x  .             3 x  .             3 x  .             3 x  .
4 .  x  x          4 .  x  x          4 x  x  x          4 x  x  x
5 .  x  x  .       5 .  x  x  .       5 x  x  x  .       5 x  x  x  .
6 x  .  .  x  x    6 x  x  x  x  x    6 x  x  x  x  x    6 x  x  x  x  x
  1  2  3  4  5      1  2  3  4  5      1  2  3  4  5      1  2  3  4  5

   0-dist             1-dist             2-dist          3-dist? No change!
                                                         So, done.
```

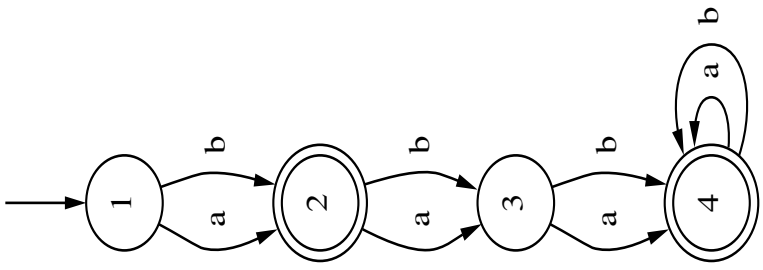


Fig. 10.11. (i) Example for DFA minimization, (ii) initial table for DFA minimization, (iii) steps in DFA minimization, and (iv) the final minimized DFA

2. All 0-length string distinguishable states are all pairs of states that consist of *exactly* one accept state (see below). All subsequent steps identifying i-distinguishable states for all i are also in Figure 10.11.
3. Let us understand how one `x` was added. In Figure 10.11, we put an `x` (to distinguish between) states `2` and `6`. Why is this so? This is because `2` has a move upon input `a` to state `4`, while state `6` moves upon `a` to state `6` itself. From the `0-dist` table, we know that states `4` and `6` are distinguishable: one being a final and the other being a nonfinal state.
4. At the `3-dist` step, there was no change from the previous table (in other words, a *fixed-point has been reached*, as described in Chapter 6). At this point, state pairs `3,2` and `5,4` still have a '.' connecting them (they could not be distinguished). Therefore, we merge these states, resulting in the minimized DFA of Figure 10.11.

10.4 Error-correcting DFAs

Consider another experiment that shows the value of tool-assisted debugging of machines. In this experiment, a DFA has to be designed to recognize all strings that are a Hamming distance of 2 away from the set of strings denoted by the regular expression `(0101)+`. For instance, `010101` and `010110` are a Hamming distance of 2 apart, as are `010111` and `000110`.

Definition 10.2. *(Hamming Distance)* Given two strings V_1 and V_2, of equal length and over $\{0,1\}^*$, they are a Hamming distance of d apart if they differ in exactly d positions.

The DFA we are to design can be regarded as an *error-correcting* DFA which corrects two-bit errors. We shall derive this DFA using two distinct approaches, each time by following two different lines of logic:

1. The first approach will be to develop a cyclic DFA that performs transitions between states `I`, `A`, `B`, `C`, `F`, and back to `A` upon seeing `01010`. However, upon seeing any erroneous symbol—for instance, seeing a `1` in state `I`, it goes to a cycle at a lower "stratum" labeled with states `A1`, `B1`, etc. This is presented in Figure 10.12.
2. Another approach will be to write a regular expression that captures all possible zero-bit errors, all possible one-bit errors, and all possible two-bit errors.

We shall find the minimal DFAs corresponding to these constructions. If correctly performed, we must obtain *isomorphic* minimal DFAs, again

serving to verify our construction methods. We discuss these constructions in the following sections.

10.4.1 DFA constructed using error strata

```
I - 0 -> A
I - 1 -> A1

A - 1 -> B     A1 - 1 -> B1     A2 - 0 -> BH
A - 0 -> B1    A1 - 0 -> B2     A2 - 1 -> B2
B - 0 -> C     B1 - 0 -> C1     B2 - 0 -> C2
B - 1 -> C1    B1 - 1 -> C2     B2 - 1 -> BH
C - 1 -> F     C1 - 1 -> F1     C2 - 1 -> F2
C - 0 -> F1    C1 - 0 -> F2     C2 - 0 -> BH
F - 0 -> A     F1 - 0 -> A1     F2 - 0 -> A2
F - 1 -> A1    F1 - 1 -> A2     F2 - 1 -> BH     BH - 0,1 -> BH
```

Fig. 10.12. A DFA that has two error strata implementing all strings that are a Hamming distance of 2 away from the language `(0101)+`

The DFA corresponding to the use of error-correcting strata is captured in Figure 10.12. By running the command `perl fa2grail.perl h2 > h2fa`,[2] we convert this ASCII input into a `grail` representation. Following that, we apply the command

```
cat h2fa | fmdeterm | fmmin | perl grail2ps.perl - > h2fa.ps
```

The result is shown in Figure 10.13 on the left-hand side.

10.4.2 DFA constructed through regular expressions

A regular expression that captures all possible zero, one, and two-bit errors is in Figure 10.14. We have added spaces and newlines, as well as comments beginning with "`--`" to enhance the readability of the above RE. We then perform the command sequence:

```
cat h2re | retofm | fmdeterm | fmmin | perl grail2ps.perl - > h2fa1.ps
```

The result is shown in Figure 10.13 on the right-hand side. Contrasting it with the other DFA in this figure, we can see that barring node numberings as well as the layout (under the control of the '`dot`' drawing package), these DFAs are isomorphic.

[2] We present this file in an intuitively layered manner; before running the script, one must present all the entries to occupy one column.

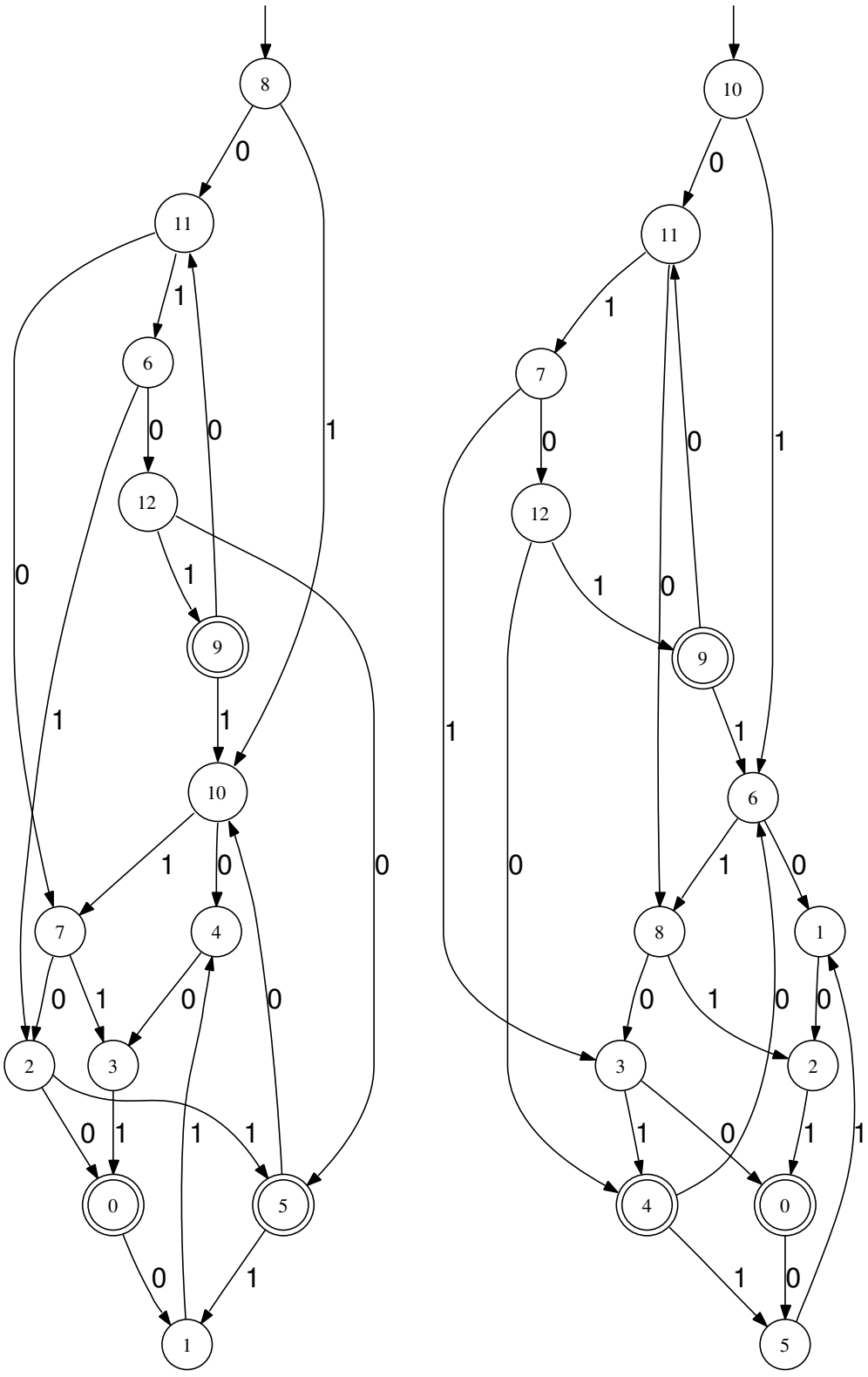


Fig. 10.13. Minimal DFAs for the same language are isomorphic

```
((0101)*)                                  -- Embedding head sequence

          ( 0101                           -- 0-bit error option

            + 1101+0001+0111+0100          -- 1-bit  error option

            +
              (1101+0001+0111+0100)        -- One possibility for a
                  ((0101)*)                -- 2-bit error as two one-
              (1101+0001+0111+0100)        -- bit errors with a
                                           -- correct mid-sequence
            +

            1001+1111+1100+0011+0000+0110 -- Another possibility for
                                           -- a 2-bit error as two
          )                                -- erroneous  bits within
                                           -- a block of four bits

((0101)*)                                  -- Embedding tail sequence
```

Fig. 10.14. A regular expression for all 0-, 1-, and 2-bit errors

10.5 Ultimate Periodicity and DFAs

Ultimate periodicity is a property that captures a central property of regular sets (recall the definition of ultimate periodicity given in Definition 5.1, page 76).

Theorem 10.3. *If L is a regular language over an alphabet Σ, then the set $Len = \{length(w) \mid w \in L\}$ is ultimately periodic.*

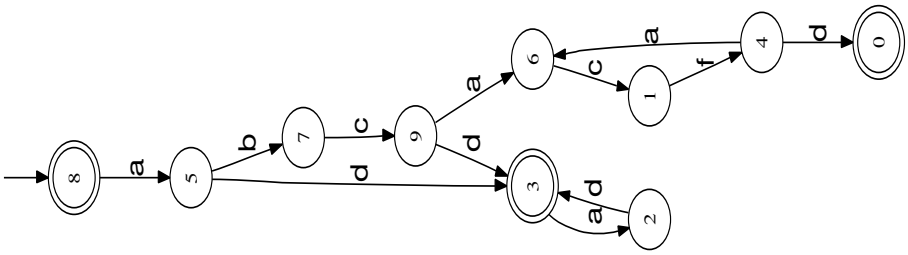
Note: The converse is *not* true. For example, the language $\{0^n 1^n \mid n \geq 0\}$ has strings whose lengths are ultimately periodic, and yet this language is not regular.

A good way to see that Theorem 10.3 is true is as follows. Given any DFA D over some alphabet Σ, consider the NFA N obtained from D by first replacing every transition labeled by a symbol in $\Sigma \setminus \{a\}$ by a. In other words, we are applying a homomorphism that replaces every symbol of the DFA by a. Hence, the resulting machine is bound to be an NFA, and the *lengths* of strings in its language will be the same as those of the strings in the language of the original DFA (in other words, what we have described is a *length-preserving* homomorphism). Now, if we convert this NFA to a DFA, we will get a machine that starts out in a start state and proceeds for some number (≥ 0) of steps before it "coils" into itself. In other words, it attains a *lasso shape.* It *cannot*

have any other shape than the 'coil' (why?). This coiled DFA shows that the length of strings in the language of any DFA is *ultimately periodic* with the period defined by the size of the coil.

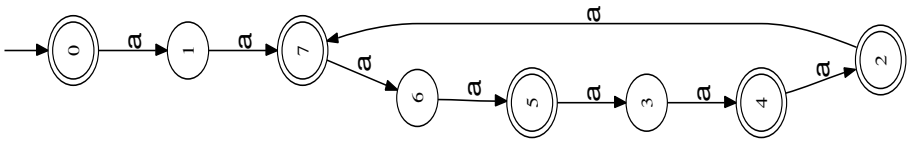
We now take an extended example to illustrate these ideas. Consider the DFA built over $\Sigma = \{a, b, c, d, f\}$ using the following command pipeline:

```
echo '(ad)*+((abc)((acf)*+(da)*)d)' | retofm | fmdeterm
    | fmmin | grail2ps.perl - | gv -
```



The DFA generated by converting every symbol to a and determinizing the result is as follows.

```
echo '(aa)*+((aaa)((aaa)*+(aa)*)a)' | retofm | fmdeterm
    | fmmin | grail2ps.perl - | gv -
```



The length of strings in the language of this DFA is ultimately periodic, with the values of the constants $n = 2$ and $p = 6$ as per the definition of UP appearing in Section 5.2.1. Based on all these observations, we can state another theorem:

Theorem 10.4. *A language over a singleton alphabet is regular if and only if the length of strings in this language is ultimately periodic.*

Chapter Summary

This chapter covered quite a bit of important ground in terms of conversions between machine types. It also illustrated two very fascinating topics assisted by the `grail` tools. The first is that minimal DFAs for the same language are isomorphic. The second is that DFAs with infinite languages over a singleton alphabet always have a "lasso" shape to them, and accepting states are sprinkled along the lasso. This has the

effect of making the *string lengths* of strings in this language *ultimately periodic*.

Exercises

10.1. Suppose an alternate definition of δ_D is offered:

$$\delta_D(S, a) = \{y \mid \exists s \in S : y \in Eclosure(\delta(s, a))\}.$$

Does it change the behavior of the resulting DFA? Justify your answer. Why do we perform *Eclosure* before and after δ in case of the NFA, on page 161?

10.2. Convert the NFA in Exercise 9.4 into an equivalent DFA, showing all the steps.

10.3. Convert the NFA of Figure 9.2 to a DFA for $k = 2$ and $k = 5$. Repeat for the modified NFA in Figure 9.3.

10.4. The token game of an NFA can be succinctly stated as follows: an NFA accepts a string x if there *exists* a path labeled by z from the start state to some final state, and x is z projected onto the alphabet Σ. Consider a variant of an NFA called *all-paths NFA*. In an all-paths NFA, a string x is accepted if and only if *all* such z paths that are in the machine actually lead to some final state. Formally define the all-paths NFA as a five-tuple mathematical structure, and prove that its language is regular, by converting it to an equivalent DFA.

10.5. What are the languages of the machines in Figure 10.2 and Figure 10.3?

10.6. Argue that L_{add} is regular, where

$$L_{add} = \{a_0 b_0 c_0 a_1 b_1 c_1 \ldots a_{k-1} b_{k-1} c_{k-1} \mid k > 0 \wedge \text{AddOK}\}$$

where AddOK $= (a_{k-1} \ldots a_0) + (b_{k-1} \ldots b_0) = (c_{k-1} \ldots c_0)$. In other words, the addition of the unsigned binary words $(a_{k-1} \ldots a_0)$ and $(b_{k-1} \ldots b_0)$ yields $(c_{k-1} \ldots c_0)$, where a_{k-1} is the MSB and a_0 the LSB (and likewise for b and c).

10.7.
Let $\Sigma = \{0, 1\}$ and let D be a DFA over Σ. Obtain an NFA N for the language of strings that are the first third of all the strings accepted by D (the "first-third" language). Formally,

$$L_{x--} = \{x \mid x \in \Sigma^* \wedge \exists y, z \in \Sigma^* : |x| = |y| = |z| \wedge xyz \in L(D)\}$$

10.8. Repeat Exercise 10.7 for the middle-third language.

10.9. Call the DFA in Figure 8.5 D. Obtain the complement DFA $\overline{D}$ by the complementation algorithm. Then obtain a DFA corresponding to the union of D and $\overline{D}$. Repeat for the intersection of D and $\overline{D}$. Check that you are indeed obtaining the right answers.

10.10. Give an example of an NFA on which performing complementation, as with DFAs, (exchanging final and nonfinal states) is correct, and another example where it is incorrect. *This shows that exchanging final and non-final states does* **not** *complement NFAs!*

10.11. Section 10.2.5 describes a construction for star. Describe an alternate construction for star that results in an NFA with exactly one final state.

10.12. Notice that the NFA of Illustration 10.3.1 had a very direct correspondence with the corresponding RE. In fact, if we apply the GNFA method to convert it to an RE, we will obtain an RE that is very close to that in Figure 10.14. However, the NFA of Section 10.3.2 when converted to an RE resulted in an extremely complex RE. Now, if we were to convert this RE back to an NFA using the procedure described in Illustration 10.3.1, we would obtain something quite different from the NFA of Figure 10.6. Intuitively describe the kinds of NFAs and REs for which close correspondence will be maintained during conversions, and those NFAs and REs where such correspondence will not be obtained.

10.13. Instantiate the NFA in Figure 9.3 for $n = 2$ and $n = 3$, calling these machines N_2 and N_3, respectively. Perform union and concatenation. With respect to both these results, list eight strings in lexicographic order.

10.14. For the N_3 machine in Exercise 10.13, perform the star operation. How does the result differ from the machine in Figure 9.3 for $k = 3$?

10.15. Reverse the NFAs in Figure 9.2 and Figure 9.3 for $n = 3$. Convert each resulting NFA to a DFA and compare their languages.

10.16. Modify the regular expression in Figure 10.14 to account for the constraint that no two consecutive bits may be in error. Perform this modification in two ways:

1. By directly editing the RE of Figure 10.14
2. By constraining the RE of Figure 10.14 suitably

Compare the results using `grail` by obtaining isomorphic minimal DFAs.

10.17. Complete the derivation in Page 173.

10.18. Express the NFA to RE conversion algorithm through recursive pseudocode. Assume that you are given a preprocessed NFA. Check whether this NFA is "done" by seeing that it has a direct path from B to E, and if so, output the RE that labels this path. Else, express the choice of a state s at random using $\exists$ or *choose*. Then, eliminate s, updating the REs of all the states that are directly reachable from s or directly reach s. Recurse on the resulting automaton.

10.19. Convert the DFA of Figure 9.5 into a regular expression using the conversion procedure that you just now pseudocoded in Exercise 10.18. Now convert the RE you obtain to an NFA. Determinize this NFA, and compare the resulting DFA to the one you started from.

10.20. Another way to convert NFAs to RE uses Arden's Lemma [68], which is:

> A language equation of the form $X = AX \cup B$, where $\varepsilon \notin A$ has a unique solution $X = A^*B$.

1. Write a system of recursive equations corresponding to the example DFA in Figure 9.5. Some of the equations are the following, where L_1, L_2, etc, denote the languages of states 1, 2, etc. (meaning, if the start state were to be set to these states, these would be the languages of the DFA):
   $L_1 = 0\ L_2$
   $L_2 = 1\ L_4$
   $L_2 = 1\ L_4$
   $L_4 = 0\ L_5 \cup \{\varepsilon\}$
2. Convert this mutually recursive system of equations into a self-recursive equation in terms of one variable, and solve it using Arden's lemma. You may refer to a method such as Gaussean elimination which is used to solve simultaneous equations over Reals.
3. Solve the self-recursion to a closed form solution using Arden's lemma, and back substitute the result to obtain a closed form solution to all languages.

10.21. Use the least fixed-point approach introduced in Chapter 6 to derive Arden's Lemma.

10.22. Apply the DFA minimization algorithm to the DFA of Figure 9.5.

10.23. Perform the `grail` command sequence

```
> echo '(00*+"")(100*)*(11+1+"")(00*1)*(00*+"")' | retofm
  | fmdeterm >! a1fixed-unmin.txt
```

Then, hand-minimize the result (meaning, construct the DFA from the regular expression by inspection and then hand-minimize it), comparing it with the minimized version `a1fixed.txt` discussed on page 156.

10.24. Express the DFA minimization algorithm neatly using pseudocode. Analyze its time complexity.

10.25. Describe a language over a singleton alphabet such that the length of strings in this language is *not* ultimately periodic.

10.26. Apply the homomorphism $1 \to 0$, $0 \to 0$, and $\varepsilon \to \varepsilon$ to the DFA of Figure 9.5. Convert the resulting NFA into a DFA. Show that the length of strings in this DFA is ultimately periodic by finding n and p parameters.

11

The Automaton/Logic Connection, Symbolic Techniques

Most believe that computer science is a very young subject. In a sense, that is true - there was the theory of relativity, vacuum tubes, radio, and Tupperware well before there were computers. However, from another perspective, computer science is at least 150 years old! Charles Babbage[1] started building his Analytic Engine in 1834 which remained unfinished till his death in 1871. His less programmable Difference Engine No. 2 was designed between 1847 and 1849, and built to his specifications in 1991 by a team at London's Science Museum. As for the 'theory' or 'science' behind computer science, George Boole published his book on Boolean Algebra[2] in 1853.

Throughout the entire 150 years (or so) history of computer science, one can see an attempt on part of researchers to understand *reasoning* as well as *computation* in a unified setting. This direction of thinking is best captured by Hilbert in one of his famous speeches made in the early 1900s in which he challenged the mathematical community with 23 open problems. Many of these problems are still open, and some were solved only decades after Hilbert's speech. One of the conjectures of Hilbert was that the entire body of mathematics could perhaps be "formalized." What this meant is basically that mathematicians had no more creative work to carry out; if they wanted to discover a new result in mathematics, all they had to do was to program a computer to systematically crank out all possible proofs, and check to see whether the theorem whose proof they are interested in appears in one of these proofs!

[1] Apparently, Babbage is also credited with the invention of the 'cow-catcher' that you see in front of locomotive engines!

[2] Laws of thought. (You might add: to prevent loss of thought through loose thought).

In 1931, Kurt Gödel dropped his 'bomb-shell.[3] He formally stated and proved the result, "Such a device as Hilbert proposed is impossible!" By this time, Turing, Church, and others demonstrated the true limits of computing through concrete computational devices such as Turing machines and the Lambda calculus. The rest "is history!"

11.1 The Automaton/Logic Connection

Scientists now have a firm understanding of how computation and logic are inexorably linked together. The work in the mid 1960s, notably that of J.R. Büchi, furthered these connections by relating branches of mathematics known as *Presburger arithmetic* and branches of logic known as WS1S[4] with deterministic finite automata. Work in the late 1970s, notably by Pnueli, resulted in the adoption of *temporal logic* as a formal logic to reason about concurrency. Temporal logic was popularized by Manna and Pnueli through several textbooks and papers. Work in the 1980s, notably by Emerson, Clarke, Kurshan, Sistla, Sifakis, Vardi, and Wolper established deep connections between temporal logic and automata on infinite words (in particular Büchi automata). Work in the late 1980s, notably by Bryant, brought back yet another thread of connection between logic and automata by the proposal of using binary decision diagrams, essentially minimized deterministic finite automata for the finite language of satisfying instances of a Boolean formula, as a data structure for Boolean functions. The *symbolic model checking algorithm* proposed by McMillan in the late 1980s hastened the adoption of BDDs in verification, thus providing means to tackle the correctness problem in computer science. Also, spanning several decades, several scientists, including McCarthy, Wos, Constable, Boyer, Moore, Gordon, and Rushby, led efforts on the development of mechanical theorem-proving tools that provide another means to tackle the correctness problem in computer science.

11.1.1 DFA can 'scan' and also 'do logic'

In terms of practical applications, the most touted application domain for the theory of finite automata is in string processing – pattern matching, recognizing tokens in input streams, scanner construction, etc. However, the theory of finite automata is much more fundamental

[3] Some mathematicians view the result as their salvation.
[4] WS1S stands for *the weak monadic second-order logic of one successor.*

to computing. Most in-depth studies about computing in areas such as concurrency theory, trace theory, process algebras, Petri nets, and temporal logics rest on the student having a solid foundation on *classical automata*, such as we have studied so far. This chapter introduces some of the less touted, but nevertheless equally important, ramifications of the theory of finite automata in computing. It shows how the theory of DFA helps arrive at an important method for representing *Boolean functions* known as *binary decision diagrams*. The efficient representation as well as manipulation of Boolean functions is central to automated reasoning in several areas of computing, including computer-aided design and formal verification. In Chapter 21, we demonstrate how exploiting the "full power" of DFAs, one can represent logics with more power than propositional logic. In Chapter 22, we demonstrate how automata on infinite words can help reason about finite as well as infinite computations generated by finite-state devices. In this context, we briefly sketch the connections between automata on infinite words as well as temporal logics. We now turn to binary decision diagrams, the subject of this chapter.

Note: We use $\vee$ and $+$ interchangeably, depending on what looks more readable in a given context; they both mean the same (the *or* function).

11.2 Binary Decision Diagrams (BDDs)

Binary Decision Diagrams (BDDs) are bit-serial DFA for satisfying instances of Boolean formulas.[5] To better understand this characterization, consider the *finite* language

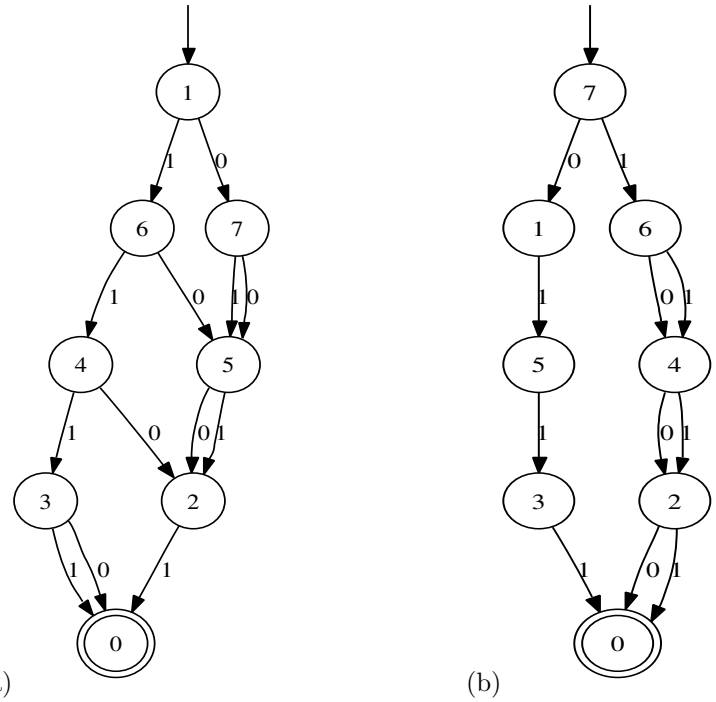
$$L_1 = \{abcd \mid d \vee (a \wedge b \wedge c)\}.$$

Since all finite languages (a finite number of finite strings in this case) are regular, a regular expression describing this language can be obtained by spelling out all the satisfying instances of $d \vee (a \wedge b \wedge c)$. This finite regular language is denoted by the following regular expression:

```
(1110+1111+0001+0011+0101+0111+1001+1011+1101)
```

By putting this regular expression in a file called `a.b.c+d`, we can use the following `grail` command sequence to obtain a minimal DFA for it, as shown in Figure 11.1(a):

[5] BDDs may also be viewed as optimized decision trees. We view BDDs as DFA following the emphasis of this book. Also note that strictly speaking, we must say *Reduced Ordered Binary Decision Diagrams* or ROBDDs. We use "BDD" as a shorthand for ROBDD.



(a) (b)

Fig. 11.1. Minimal DFAs for $d \vee (a \wedge b \wedge c)$ for (a) variable ordering *abcd*, and (b) *dabc*. The edges show transitions for inputs arriving according to this order.

```
cat a.b.c+d | retofm | fmdeterm | fmmin | perl grail2ps.perl -
   > a.b.c+d.ps
```

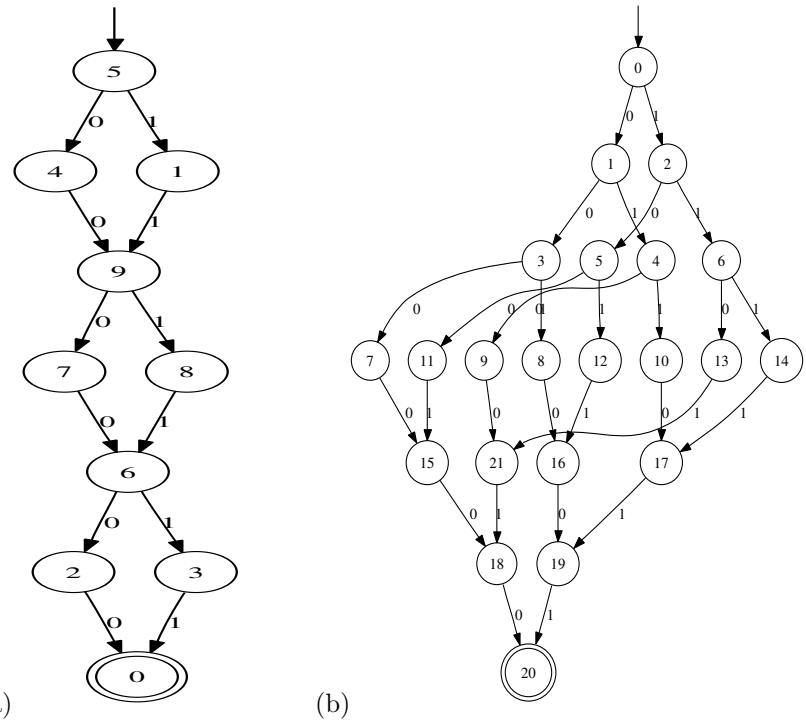
Now consider the language that merely changes the bit-serial order in which the variables are examined from *abcd* to *dabc*:

$$L_2 = \{dabc \mid d \vee (a \wedge b \wedge c)\}.$$

Using the regular expression

```
(0111+1000+1001+1010+1011+1100+1101+1110+1111)
```

as before, we obtain the minimal DFA shown in Figure 11.1(b). The two minimal DFAs seem to be of the same size. Should we expect this in general? The minimal DFAs in Figure 11.1 and Figure 11.2, are suboptimal as far as their role in decoding the binary decision goes, as they contain redundant decodings. For instance, in Figure 11.1(a),



(a) (b)

Fig. 11.2. Minimal DFAs where the variable ordering matters

after $abc = 111$ has been seen, there is no need to decode d; however, this diagram redundantly considers 0 and 1 both going to the accepting state 0. In Figure 11.1(b), we can make node 6 point directly to node 0. Eliminating such redundant decodings, Figures 11.1(a) and (b) will, essentially, become BDDs; the only difference from a BDD at that point would be that BDDs explicitly include a 0 node to which all falsifying assignments lead to.

Let us now experiment with the following two languages where we shall discuss these issues even more, and actually present the drawing of a BDD.

$$L_{interleaved} = \{abcdef \mid a = b \wedge c = d \wedge e = f\}$$

has a regular expression of satisfying assignments

```
(000000+001100+000011+110000+001111+110011+111100+111111)
```

and

$$L_{noninterleaved} = \{acebdf \mid a = b \wedge c = d \wedge e = f\}$$

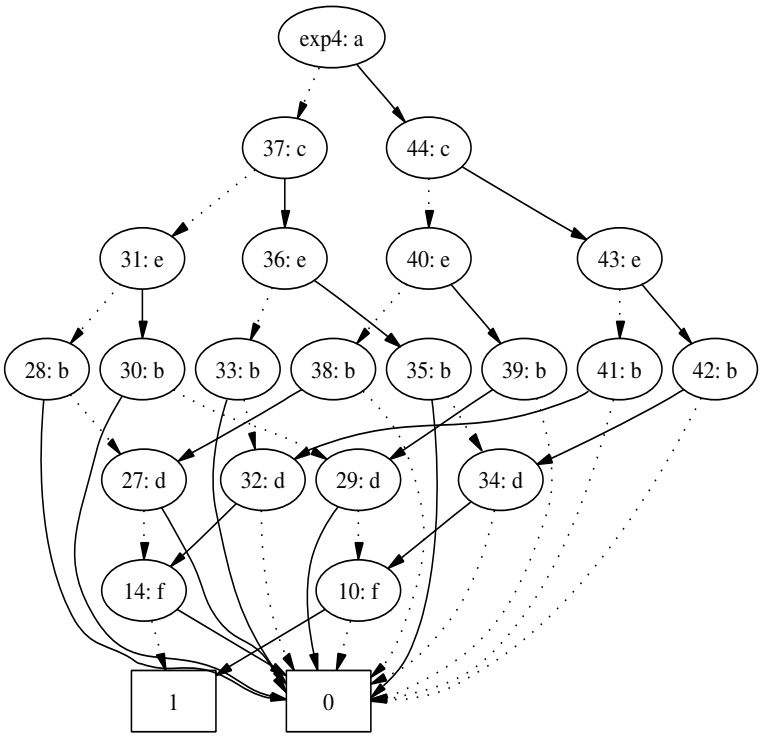


Fig. 11.3. BDD for $a = b \wedge c = d \wedge e = f$ for variable order *acebdf*

has

```
(000000+010010+001001+100100+011011+101101+110110+111111).
```

When converted to minimized DFAs, these regular expressions yield Figures 11.2(a) and (b), where the size difference due to the *variable orderings* is very apparent. The BDD for Figure 11.2(b) created using the BED tool appears in Figure 11.3. The commands used to create this BDD were:

```
bed> var a c e b d f                        % declares six variables
bed> let exp4 = (a=b) and (c=d) and (e=f) % defines the desired expn.
bed> upall exp4                             % builds the BDD -
bed> view exp4                              % displays the BDD
```

By comparing Figure 11.3 against Figure 11.2(b), one can see how, in general, BDDs eliminate redundant decodings.[6]

[6] The numbers inside the BDD nodes—such as the "14:" and "10:" in the nodes for variable f—may be ignored. They represent internal numberings chosen by the BED tool.

BDDs are efficient data structures for representing Boolean functions and computing the reachable states of state transition systems. In these applications, they are very often 'robust,' *i.e.*, their sizes remain modest as the computation advances. As many of these state transition systems have well over 2^{150} states (just to pick a large number!), this task cannot be accomplished in practice by explicitly enumerating the states. However, BDDs can often very easily represent such large state-spaces by capitalizing on an implicit representation of states as described in Section 11.3. However, BDDs can deliver this 'magic' only if a "good" variable ordering is chosen.

One also has to be aware of the following realities when it comes to using BDDs:

- The problem of determining an optimal variable ordering is NP-complete (see Chapter 20 for a definition of NP-completeness). [42]; this means that the best known algorithms for this task run in exponential worst-case time.
- In many problems, as the computation proceeds and new BDDs are built, variable orderings must be recomputed through *dynamic variable re ordering* algorithms, which are never ideal and add to the overhead.
- For certain functions (e.g., the middle bits of the result of multiplying two N-bit numbers), the BDD is provably exponentially sized, no matter which variable ordering is chosen.

Even so, BDDs find extensive application in representing as well as manipulating state transition systems realized in hardware and software. We now proceed to discuss how BDDs can be used to represent state transition relations and also how to perform reachability analysis.

11.3 Basic Operations on BDDs

BDDs are capable of efficiently representing transition relations of finite-state machines. In some cases, transition relations of finite-state machines that have of the order of 2^{100} states have been represented using BDDs. For example, a BDD that represents the transition relation for a 100-bit digital ripple-counter can be built using about 200 BDD nodes.[7] Such compression is, of course, achieved by *implicitly* representing the state space; an explicit representation (*e.g.*, using pointer based

[7] Basically, each bit of such a counter toggles when all the lower order bits are a 1, and thus all the BDD basically represents is an *and* function involving all the bits.

data structures) of a state-space of this magnitude is practically impossible. Given a transition relation, one can perform *forward* or *backward reachability* analysis. 'Forward reachability analysis' is the term used to describe the act of computing reachable states by computing the forward image ("image") of the current set of states (starting from the initial states). Backward reachability analysis computes the *pre-image* of the current set of states. One typically starts from the current set of states *violating* the desired property, and attempts to find a path back to the initial state. If such a path exists, it indicates the possibility of a computation that violates the desired property.

Each step in reachability analysis takes the current set of states represented by a BDD and computes the next set of states, also represented by a BDD. It essentially performs a *breadth-first* traversal, generating each breadth-first frontier in one step from the currently reached set of states. The commonly used formulation of traversal is in terms of computing the least fixed-point as explained in Section 11.3.2. When the least fixed-point is reached, one can query it to determine the overall properties of the system. One can also check whether desired system invariants hold in an incremental fashion (without waiting for the fixed-point to be attained) by testing the invariant after each new breadth-first frontier has been generated. Here, an *invariant* refers to a property that is true at every reachable state.

We will now take up these three topics in turn, first illustrating how we are going to represent state transition systems.

11.3.1 Representing state transition systems

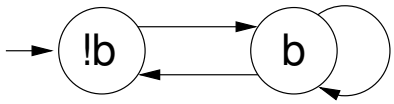


Fig. 11.4. Simple state transition system (example SimpleTR)

We capture transition systems by specifying a binary state transition relation between the *current* and *next* states, and also specifying a predicate capturing the initial states. If inputs and outputs are to be modeled, they are made part of the state vector. Depending on the problem being modeled, we may not care to highlight which parts of the state vector are inputs and which are outputs. In some cases, the entire state of the system will be captured by the states of inputs and

outputs. Figure 11.4 presents an extremely simple state transition system, called SimpleTR. Initially, the state is `0`. Whenever the state is `0`, it can become `1`. When it is `1`, it can either stay `1` or become `0`. These requirements can be captured using a single Boolean variable b representing the current state, another Boolean variable b' representing the next state,[8] and an initial state predicate and a state transition relation involving these variables, as follows:

- The initial state predicate for SimpleTR is $\lambda b.\neg b$, since the initial state is `0`. Often, instead of using the lambda syntax, initial state predicates are introduced by explicitly introducing a named initial state predicate I and defining it by an equation such as $I(b) = \neg b$. For brevity,[9] we shall often say "input state represented by $\neg b$."
- The state transition relation for SimpleTR is $\lambda(b, b').\neg b b' + b b' + b \neg b'$, where each product term represents one of the transitions. The values of b and b' for which this relation is satisfied represent the present and next states in our example. In other words,
  - a move where b is false now and true in the next state is represented by $\neg b b'$.
  - a move where b is true in the present and next states is represented by $b b'$.
  - finally, a move where b is true in the present state and false in the next state is represented by $b \neg b'$.

  This expression can be simplified to $\lambda(b, b').(b + b')$. The above relation can also written in terms of a transition relation T defined as $T(b, b') = b + b'$. We shall hereafter say "transition relation $b + b'$." Notice that this transition relation is false for $b = 0$ and $b' = 0$, meaning there is no move from state 0 to itself (all other moves are present).

11.3.2 Forward reachability

The set of reachable states in SimpleTR starting from the initial state $\neg b$ can be determined as follows:

- Compute the set of states in the initial set of states.
- Compute the set of states reachable from the initial states in n steps, for $n = 1, 2, \ldots$.

[8] The 'primed variable' notation was first used by Alan Turing in one of the very first program proofs published by him in [89].

[9] Syntactic sugar can cause cancer of the semi-colon – Perlis

In other words, we can introduce a predicate P such that a state x is in P if and only if it is reachable from the initial state I through a finite number of steps, as dictated by the transition relation T. The above recursive recipe is encoded as

$$P(s) = (I(s) \vee \exists x.(P(x) \wedge T(x, s))).$$

This formula says that s is in P if it is in I, or there exists a state x such that x is in P, and the transition relation takes x to s.
Rewriting the above definition, we have

$$P = \lambda s.(I(s) \vee \exists x.(P(x) \wedge T(x, s)))).$$

Rewriting again, we have

$$P = (\lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x, s))))) \; P.$$

In other words, P is a fixed-point of

$$\lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x, s)))).$$

Let us call this Lambda expression H:

$$H = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x, s)))).$$

In general, H can have multiple fixed-points. Of these, the *least fixed-point* represents exactly the reachable set of states, as next explained in Section 11.3.3.

11.3.3 Fixed-point iteration to compute the least fixed-point

As shown in Section 6.1, the least fixed-point can be obtained by "bottom refinement" using the functional obtained from the recursive definition. In the same manner, we will determine P, the least fixed-point of H, by computing its approximants that, in the limit, become P. Let us denote the approximants P_0, P_1, P_2, ... We have $P_0 = \lambda x.false$, the "everywhere false" predicate. The next approximation to P is obtained by feeding P_0 to the "bottom refiner" (as illustrated in Section 6.1):

$$P_1 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x, s))))P_0$$

which becomes $\lambda s.I(s)$. This approximant says that P is a predicate true of s whenever $I(s)$. While this is not true (P must represent the reachable state set and not the initial state alone), it is certainly a better answer than what P_0 denotes, which is that there are *no* states in the reachable state set! We now illustrate all the steps of this computation, taking SimpleTR for illustration. We use the abbreviation of not showing the lambda abstracted variables in each step.

- $I = \lambda b.\neg b.$
- $T = \lambda(b, b^{'}).\ (b + b^{'}).$
- $P_0 = \lambda s.false$, which encodes the fact that "we've reached nowhere yet!"
- $P_1 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x, s))))P_0.$
  This simplifies to $P_1 = I$, which is, in effect, an assertion that we've "just reached" the initial state, starting from P_0.
- Let's see the derivation of P_1 in detail. Expanding T and P_0, we have
  $P_1 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge\ (x + s)\ )))\ \ (\lambda x.false).$
- The above simplifies to $\neg b$.
- By this token, we are expecting P_2 to be all states that are zero or one step away from the start state. Let's see whether we obtain this result.
- $P_2 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x, s))))P_1.$
  $= \lambda s.(\neg s \vee \exists x.(\neg x \wedge (x + s))).$
  $= \lambda s.1.$
- This shows that the set of states reached by all the breadth-first frontiers (combined) that are zero and one step away from the start state, includes every state. Another iteration would not change things; the[10] least fixed-point has been reached.

BED Commands for SimpleTR:

The BED commands given in Figure 11.5 compute the reachable set of states using forward reachability in our example. We can see that `P2`, the least fixed-point, is indeed *true* — namely, the characteristic predicate for the set of all states. (*Note: In BED, the primed variables must be declared immediately after the unprimed counterparts*). In addition to the explicit commands to calculate the least fixed-point, BED also provides a single command called `reach`. Using that, one can calculate the least fixed-point in one step. In our present example, `RS` and `P2` end up denoting the BDD for *true*.

```
let RS = reach(I,T)
upall RS
view RS
```

Section 11.3.4 discusses another example where the details of the fixed-point iteration using BED are discussed.

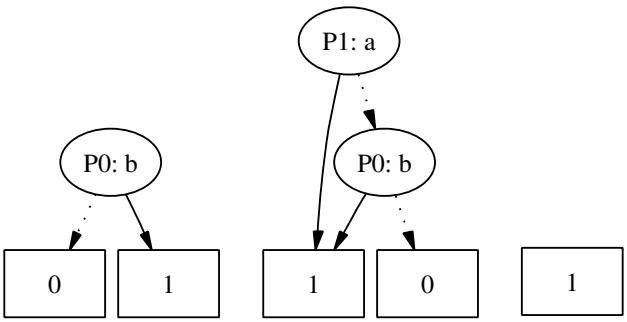
[10] We do not discuss many of the theoretical topics associated with computing fixed-points in the domain of state transition systems — such as why least fixed-points are unique, etc. For details, please see [20].

```
var b bp                % Declare b and b'
let I = !b              % Declare init state
let t1 = !b and bp      % 0 --> 1
upall t1                % Build BDD for it
view t1                 % View it
let t2 = b and bp       % 1 --> 1
let t3 = b and !bp      % 1 --> 0
let T = t1 or t2 or t3  % All three edges
upall T                 % Build and view the BDD
view T                  %

let   P0 = false
upall P0
view  P0

let   P1 = I or ((exists b. (P0 and T))[bp:=b])
upall P1
view  P1

let   P2 = I or ((exists b. (P1 and T))[bp:=b])
upall P2
view  P2
```



P0 P1 P2, the least fixed-point

Fig. 11.5. BED commands for reachability analysis on SimpleTR, and the fixed-point iteration leading up to the least fixed-point that denotes the set of reachable states starting from I

Why Stabilization at a Fixed-Point is Guaranteed

In every finite-state system modeled using a finite-state Boolean transition system, the least fixed-point is always reached in a finite number of steps. Let us try to argue this fact first using only a simple observation. The observation is that all the Boolean expressions generated during the course of fixed-point computation are over the same set of vari-

ables. Since there are exactly 2^{2^N} Boolean functions over N Boolean variables (see Illustration 4.5.2), eventually two of the approximants in the fixed-point computation process will have to be the same Boolean function. However, this argument does not address whether it is possible to have "evasive" or "oscillatory" approximants $P_i, P_{i+1}, \ldots, P_j$ such that $i \neq j$ and $P_j = P_i$. If this were possible, it would be possible to cycle through $P_i, \ldots, P_j$ without ever stabilizing on a fixed-point. Fortunately, this is not possible! Each approximant P_{i+1} is *more defined* than the previous approximant P_i, in the sense defined by the implication lattice defined in Illustration 4.5.3. With this requirement, the number of these ascending approximants is finite, and one of these would be the least fixed-point. See Andersson's paper [7] for additional examples of forward reachability. The book by Clarke et.al. [20] gives further theoretical insights.

11.3.4 An example with multiple fixed-points

Consider the state transition system in Figure 11.6 with initial state s0 (called MultiFP). The set of its reachable states is simply {s0} (and is characterized by the formula $a \wedge b$), as there is no reachable node from s0. Now, a fixed-point iteration beginning with the initial approximant for the reachable states set to $P0 = false$ will converge to the fixed-point $a \wedge b$. What are the other fixed-points one can attain in this system? Here they are:

- With the initial approximant set to {s0,s1}, which is characterized by b, the iteration would reach the fixed-point of $a \vee b$, which characterizes {s0,s1,s2}.
- Finally, we may iterate starting from the initial approximant being 1, corresponding to {s0,s1,s2,s3}. The fixed-point attained in this case is 1, which happens to be the greatest fixed-point of the recursive equation characterizing reachable states.

Hence, in this example, there are three distinct fixed-points for the recursive formula defining reachable states. Of these, the *least* fixed-point is $a \wedge b$, and truly characterizes the set of reachable states; $a \vee b$ is the intermediate fixed-point, and 1 is the greatest fixed-point. It is clear that $(a \wedge b) \Rightarrow (a \vee b)$ and $(a \vee b) \Rightarrow 1$, which justifies these fixed-point orderings. Figure 11.6 also describes the BED commands to produce this intermediate fixed-point.

```
var a ap b bp

let T = (a  and b  and ap  and bp)  or /* S0 -> S0 */
        (!a and b  and !ap and bp)  or /* S1 -> S1 */
        (a  and !b and ap  and !bp) or /* S2 -> S2 */
        (!a and !b and !ap and !bp) or /* S3 -> S3 */
        (!a and b  and ap  and !bp) or /* S1 -> S2 */
        (a and !b  and !ap and bp)  or /* S2 -> S1 */
        (!a and b  and ap  and bp)  or /* S1 -> S0 */
        (a and !b  and ap  and bp)     /* S2 -> S0 */

upall T
view T              /* Produces BDD for TREL 'T' */

let I = a and b
let P0 = b
let P1 = I or ((exists a. (exists b. (P0 and T)))[ap:=a][bp:=b])

upall P1
view P1
```

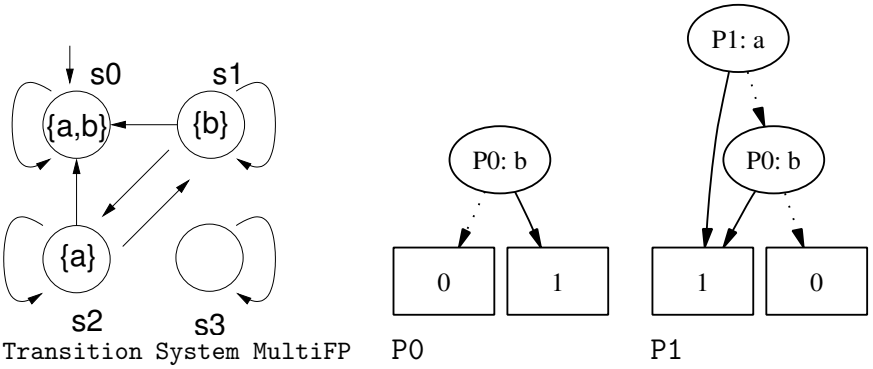


Fig. 11.6. Example where multiple fixed-points exist. This figure shows attainment of a fixed-point $a \vee b$ which is between the least fixed-point of $a \wedge b$ and the greatest fixed-point of 1. The figure shows the initial approximant P0 and the next approximant P1

11.3.5 Playing tic-tac-toe using BDDs

What good are state-space traversal techniques using BDDs? How does one obtain various interesting answers from real-world problems? While we cannot answer these questions in detail, we hope to leave this chapter with a discussion of how one may model a game such as tic-tac-toe and, say, compute the set of all draws in one fell swoop. Following through this example, the reader would obtain a good idea of how to employ

mathematical logic to specify a transition system through constraints, and reason about it. We assume the reader knows the game of tic-tac-toe (briefly explained in the passing).

Modeling the players and the board:

We model two players, A and B. The state of the game board is modeled using a pair of variables $a_{i,j}, b_{i,j}$ (we omit the pairing symbols $\langle\rangle$ for brevity) for each square i, j where $i \in 3$ and $j \in 3$. We assume that player A marks square i, j with an o, by setting $a_{i,j}$ and resetting $b_{i,j}$, while player B marks square i, j with an x, by resetting $a_{i,j}$ and setting $b_{i,j}$. We use variable *turn* to model whose turn it is to play (with $turn = 0$ meaning it is A's turn). The state transition relation for each square will be specified using the four variables $a_{i,j}, a_{i,j}p, b_{i,j}$, and $b_{i,j}p$. We model the conditions for a row or column remaining the same, using predicates $samerow_i$ and $samecol_i$. We define nine possible moves for both A and for B. For example, `M00` model's A's move into cell $0, 0$; Similarly, we employ `N00` to model B's move into cell $0, 0$, and so on for the remaining cells. The transition relation is now defined as a disjunction of the $M_{i,j}$ and $N_{i,j}$ moves. We now capture the constraint *atmostone* that says that, at most one player can play into any square. We then enumerate the gameboard for all possible wins and draws. In the world of BDDs, these computations are achieved through "symbolic breadth first" traversals. We compute the reachable set of states, first querying it to make sure that only the correct states are generated. Then we compute the set of states defining draw configurations. The complete BED definitions are given in Appendix B.

Chapter Summary

This chapter briefly reviewed the history of mathematical logic and pointed out the fact that in the early days of automata theory, mathematical logic and automata were discussed in a unified setting. This approach has immense pedagogical value which this book tries to restore to some extent. A practitioner who works on advanced hardware/software debugging method needs to know *both* of these topics well. For instance, automata theory has, traditionally, been considered an essential prerequisite for an advanced class on compilation. However, recent publications in systems/compilers (e.g., [121]) indicate the central role played by BDDs (see below) and related notions in mathematical logic.

We then discuss how Boolean formulas can be represented in a canonical fashion using the so-called 'reduced ordered binary decision

diagrams,' or "BDDs" for short. We then present how finite-state machines can be represented and manipulated using BDDs. We show how reachable states starting from a set of start states can be computed using forward reachability, by using the notion of fixed-points introduced in Chapter 6. We finish the chapter with an illustration of how the game of tic-tac-toe may be modeled using BDDs, and how a tool called `BED` may be used to compute interesting configurations, such as all the *draw* positions, all possible *win* positions, etc.

BDDs are far richer in scope and application than we have room to elaborate here. The reader is referred to [14, 13] for an exposition of how BDDs are used in hardware and software design, how BDDs may be combined using Boolean operations through the *apply* operator, etc. An alternate proof of canonicity of BDDs appears in [14]. Our presentation of BDDs as automata draws from [22], and to some extent from [111].

Exercises

11.1. Similar to Figure 11.3, draw a BDD for all 16 Boolean functions over variables x and y. (Some of these functions are $\lambda(x, y).true$, $\lambda(x, y).false$, $\lambda(x, y).x$, $\lambda(x, y).y$, etc. Down this list, you have more "familiar" functions such as $\lambda(x, y).nand(x, y)$, and so on. Express these functions without the "lambda" part in BED syntax, and generate the BDDs using BED.)

11.2.

1. Obtain an un-minimized DFA (in the form of a binary tree) for the language
$$L = \{abc \mid a \Rightarrow b \wedge c\}$$
picking the best variable ordering (in case two variable orderings are equal, pick the one that is in lexicographic order). Show the black-hole state also.
2. Minimize this DFA, and then show the additional steps that cast the minimized DFA into a BDD.

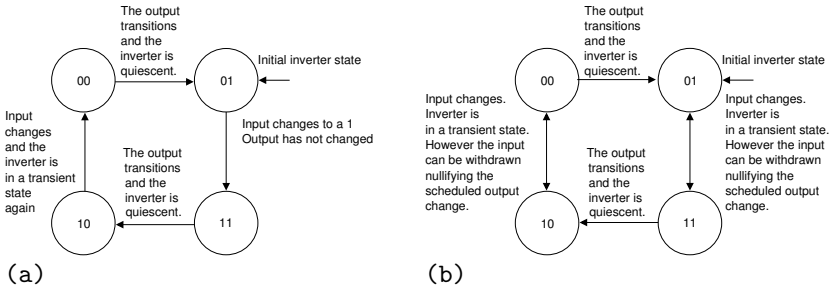
11.3. Consider the examples given in Figure 11.2. Construct similar examples for the *addition* operation. More specifically, consider the binary addition of two unsigned two-bit numbers a_1a_0 and b_1b_0, resulting in answer $c_2c_1c_0$. Generate a BDD for the carry output bit, c_2. Choose a variable ordering that minimizes the size of the resulting BDD and experimentally confirm using BED.

11.4. Repeat Exercise 11.3 to find out the variable ordering that maximizes the BDD size.

11.5. Represent the behavior of a nand gate, under the inertial delay model, as a state transition system. Encode this transition system using a BDD. Here are some general details on how to approach this problem.

The behavior of an inverter can be modeled using a pair of bits representing its input and output. (For a nand gate, we will need to employ three bits.) In the *transport delay* model, every input change, however short, is faithfully copied to the output, but after a small delay. There is another delay model called the *inertial* delay model in which "short" pulses may not make it to the output.

The behavior of an inverter under these delay models are shown in figures (a) and (b) below.

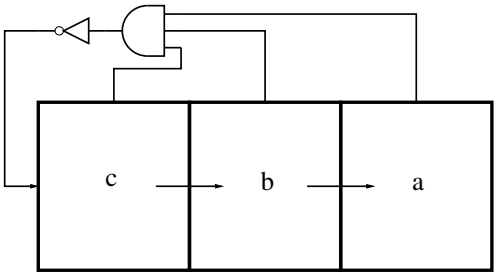


11.6. Draw a BDD for the transition relation of a two-bit binary counter with output bits $a_1 a_0$ for initial state 00, counting in the usual $0, 1, 2, 3$ order. Repeat for a two-bit gray-code counter that counts 00, 01, 11, 10, and back to 00.

11.7.

1. With respect to the state transition relation of Figure 11.6(a), identify all the fixed-points of the recursive equation for reachability.
2. Given a state transition system (say, as a graph, as in Figure 11.6(a)), what is a general algorithm to determine the number of fixed-points of its recursive equation for reachability?

11.8. Consider a three-bit shift register based counter with the indicated next-state relation for its three bits:



1. Represent the next-state relation of this counter using a single ROBDD. Choose a variable ordering that minimizes the size of your ROBDD and justify your choice.
2. Compute the set of all reachable states using forward reachability analysis, using the `reach` command, starting at state `000`.
3. Justify the correctness of the answer you obtain. The answer you obtain must be a Boolean formula over a, b, c. Show that this formula is satisfied exactly for those states reachable by the system.

11.9. A three-bit Johnson counter[11] consists of a three-bit shift register where the final $\overline{Q}$ output is connected to the first D input. Starting from a reset state of 000, this counter will go through the sequence 100, 110, 111, 011, 001, and back to 000. For this counter, repeat what Exercise 11.8 asks.

11.10. Using BED, determine the shortest number of steps to win in Tic-Tac-Toe. Appendix B has a full description of the problem encoding.

11.11. Check two conjectures concerning Tic-Tac-Toe, using BED: (i) if a player starts by marking the top-left corner, he/she may lose; (ii) if a player starts by marking the middle square, he/she may win.

11.12. Construct an example with four distinct fixed-points under forward reachability, and verify your construction similar to that explained in Figure 11.6.

11.13. Encode the Man-Wolf-Goat-Cabbage problem using BDDs. In this problem, a man has to carry a wolf, goat, and cabbage across a river. The man has to navigate the boat in each direction. He may carry no more than one animal/object on the boat (besides him) at a time. He must not leave the wolf and goat unattended on either bank,

[11] Named after Emeritus Prof. Bob Johnson, University of Utah.

nor must he leave the goat and cabbage unattended on either bank. The number of moves taken is to be minimal. Use appropriate Boolean variables to model the problem.