Contents

4. Reliable Computation

Any planned computation will be subject to different kinds of unpredictable influences during execution. Here are some examples:

(1) Loss or change of stored data during execution.

(2) Random, physical errors in the computer.

(3) Unexpected interactions between different parts of the system working simultaneously, or loss of connections in a network.

(4) Bugs in the program.

(5) Malicious attacks.

Up to now, it does not seem that the problem of bugs can be solved just with the help of appropriate algorithms. The discipline of software engineering addresses this problem by studying and improving the structure of programs and the process of their creation.

Malicious attacks are addressed by the discipline of computer security. A large part of the recommended solutions involves cryptography.

Problems of kind (4) are very important and a whole discipline, distributed computing has been created to deal with them.

The problem of storage errors is similar to the problems of reliable communication, studied in information theory: it can be viewed as communication from the present to the future. In both cases, we can protect against noise with the help of ***error-correcting codes*** (you will see some examples below).

In this chapter, we will discuss some sample problems, mainly from category (4). In this category, distinction should also be made between permanent and transient errors. An error is ***permanent*** when a part of the computing device is damaged physically and remains faulty for a long time, until some outside intervention by repairmen to it. It is ***transient*** if it happens only in a single step: the part of the device in which it happened is not damaged, in the next step it operates correctly again. For example, if a position in memory turns from 0 to 1 by accident, but a subsequent write operation can write a 0 again then a transient error happened. If the bit turned to 1 and the computer cannot change it to 0 again, this is a permanent error.

Some of these problems, especially the ones for transient errors, are as old as computing. The details of any physical errors depend on the kind of computer it is implemented on (and, of course, on the kind of computation we want to carry out). But after abstracting away from a lot of distracting details, we are left with some clean but challenging theoretical formulations, and some rather pleasing solutions. There are also interesting connections to other disciplines, like statistical physics and biology.

The computer industry has been amazingly successful over the last five decades in making the computer components smaller, faster, and at the same time more reliable. Among the daily computer horror stories seen in the press, the one conspicuously missing is where the processor wrote a 1 in place of a 0, just out of caprice. (It undisputably happens, but too rarely to become the identifiable source of some visible malfunction.) On the other hand, the generality of some of the results on the correction of transient errors makes them applicable in several settings. Though individual physical processors are very reliable (error rate is maybe once in every 10^{20} executions), when considering a whole network as performing a computation, the problems caused by unreliable network connections or possibly malicious network participants is not unlike the problems caused by unreliable processors.

The key idea for making a computation reliable is *redundancy*, which might be formulated as the following two procedures:

(i) Store information in such a form that losing any small part of it is not fatal: it can be restored using the rest of the data. For example, store it in multiple copies.

(ii) Perform the needed computations repeatedly, to make sure that the faulty results can be outvoted.

Our chapter will only use these methods, but there are other remarkable ideas which we cannot follow up here. For example, method (4) seems especially costly; it is desireable to avoid a lot of repeated computation. The following ideas target this dilemma.

(A) Perform the computation directly on the information in its redundant form: then maybe recomputations can be avoided.

(B) Arrange the computation into "segments" such a way that those partial results that are to be used later, can be cheaply checked at each "milestone" between segments. If the checking finds error, repeat the last segment.

4.1. Probability theory

The present chapter does not require great sophistication in probability theory but there are some facts coming up repeatedly which I will review here. If you need additional information, you will find it in any graduate probability theory text.

4.1.1. Terminology

A ***probability space*** is a triple $(\Omega, \mathcal{A}, \mathbf{P})$ where Ω is the set of ***elementary events***, $\mathcal{A}$ is a set of subsets of Ω called the set of ***events*** and $\mathbf{P} : \mathcal{A} \to [0, 1]$ is a function.

For $E \in \mathcal{A}$, the value $\mathbf{P}(E)$ is called the ***probability*** of event E. It is required that $\Omega \in \mathcal{A}$ and that $E \in \mathcal{A}$ implies $\Omega \setminus E \in \mathcal{A}$. Further, if a (possibly infinite) sequence of sets is in $\mathcal{A}$ then so is their union. Also, it is assumed that $\mathbf{P}(\Omega) = 1$ and that if $E_1, E_2, \dots \in \mathcal{A}$ are disjoint then

$$\mathbf{P}\Big(\bigcup_i E_i\Big) = \sum_i \mathbf{P}(E_i) \ .$$

For $\mathbf{P}(F) > 0$, the ***conditional probability*** of E given F is defined as

$$\mathbf{P}(E \mid F) = \mathbf{P}(E \cap F)/\mathbf{P}(F) \ .$$

Events $E_1, \dots, E_n$ are ***independent*** if for any sequence $1 \le i_1 < \dots < i_k \le n$ we have

$$\mathbf{P}(E_{i_1} \cap \dots \cap E_{i_k}) = \mathbf{P}(E_{i_1}) \cdots \mathbf{P}(E_{i_k}) \ .$$

Example 4.1 Let $\Omega = \{1, \dots, n\}$ where $\mathcal{A}$ is the set of all subsets of Ω and $\mathbf{P}(E) = |E|/n$. This is an example of a ***discrete*** probability space: one that has a countable number of elements.

More generally, a discrete probability space is given by a countable set $\Omega = \{\omega_1, \omega_2, \dots\}$, and a sequence $p_1, p_2, \dots$ with $p_i \ge 0$, $\sum_i p_i = 1$. The set $\mathcal{A}$ of events is the set of all subsets of Ω, and for an event $E \subset \Omega$ we define $\mathbf{P}(E) = \sum_{\omega_i \in E} p_i$.

A ***random variable*** over a probability space Ω is a function f from Ω to the real numbers, with the property that every set of the form $\{\omega : f(\omega) < c\}$ is an event: it is in $\mathcal{A}$. Frequently, random variables are denoted by capital letters X, Y, Z, possibly with indices, and the argument ω is omitted from $X(\omega)$. The event $\{\omega : X(\omega) < c\}$ is then also written as $[\,X < c\,]$. This notation is freely and informally extended to more complicated events. The ***distribution*** of a random variable X is the function $F(c) = \mathbf{P}[\,X < c\,]$. We will frequently only specify the distribution of our variables, and not mention the underlying probability space, when it is clear from the context that it can be specified in one way or another. We can speak about the ***joint distribution*** of two or more random variables, but only if it is assumed that they can be defined as functions on a common probability space. Random variables $X_1, \dots, X_n$ with a joint distribution are ***independent*** if every n-tuple of events of the form $[\,X_1 < c_1\,], \dots, [\,X_n < c_n\,]$ is independent.

The ***expected value*** of a random variable X taking values $x_1, x_2, \dots$ with probabilities $p_1, p_2, \dots$ is defined as

$$\mathbf{E}X = p_1 x_1 + p_2 x_2 + \cdots \ .$$

It is easy to see that the expected value is a linear function of the random variable:

$$\mathbf{E}(\alpha X + \beta Y) = \alpha \mathbf{E}X + \beta \mathbf{E}Y \ ,$$

even if X, Y are not independent. On the other hand, if variables X, Y are independent then the expected values can also be multiplied:

$$\mathbf{E}XY = \mathbf{E}X \cdot \mathbf{E}Y \ . \tag{4.1}$$

There is an important simple inequality called the ***Markov inequality***, which says that for an arbitrary nonnegative random variable X and any value $\lambda > 0$ we have

$$\mathbf{P}[\, X \geq \lambda \,] \leq \mathbf{E}X/\lambda \, . \tag{4.2}$$

4.1.2. The law of large numbers (with "large deviations")

In what follows the bounds

$$\frac{x}{1+x} \leq \ln(1+x) \leq x \quad \text{ for } x > -1 \tag{4.3}$$

will be useful. Of these, the well-known upper bound $\ln(1+x) \leq x$ holds since the graph of the function $\ln(1+x)$ is below its tangent line drawn at the point $x = 0$. The lower bound is obtained from the identity $\frac{1}{1+x} = 1 - \frac{x}{1+x}$ and

$$-\ln(1+x) = \ln \frac{1}{1+x} = \ln \left(1 - \frac{x}{1+x} \right) \leq -\frac{x}{1+x} \, .$$

Consider n independent random variables $X_1, \dots, X_n$ that are identically distributed, with

$$\mathbf{P}[\, X_i = 1 \,] = p, \quad \mathbf{P}[\, X_i = 0 \,] = 1 - p \, .$$

Let

$$S_n = X_1 + \cdots + X_n \, .$$

We want to estimate the probability $\mathbf{P}[\, S_n \geq fn \,]$ for any constant $0 < f < 1$. The "law of large numbers" says that if $f > p$ then this probability converges fast to 0 as $n \to \infty$ while if $f < p$ then it converges fast to 1. Let

$$D(f,p) = f \ln \frac{f}{p} + (1-f) \ln \frac{1-f}{1-p} \tag{4.4}$$

$$> f \ln \frac{f}{p} - f = f \ln \frac{f}{ep} \, , \tag{4.5}$$

where the inequality (useful for small f and $ep < f$) comes via $1 > 1 - p > 1 - f$ and $\ln(1-f) \geq -\frac{f}{1-f}$ from (4.3). Using the concavity of logarithm, it can be shown that $D(f,p)$ is always nonnegative, and is 0 only if $f = p$ (see Exercise 4.1-1).

Theorem 4.1 (Large deviations for coin-toss). *If $f > p$ then*

$$\mathbf{P}[\, S_n \geq fn \,] \leq e^{-nD(f,p)} \, .$$

This theorem shows that if $f > p$ then $\mathbf{P}[\, S_n > fn \,]$ converges to 0 exponentially fast. Inequality (4.5) will allow the following simplification:

$$\mathbf{P}[\, S_n \geq fn \,] \leq e^{-nf \ln \frac{f}{ep}} = \left(\frac{ep}{f} \right)^{nf} , \tag{4.6}$$

useful for small f and $ep < f$.

Proof For a certain real number $\alpha > 1$ (to be chosen later), let Y_n be the random variable that is α if $X_n = 1$ and 1 if $X_n = 0$, and let $P_n = Y_1 \cdots Y_n = \alpha^{S_n}$: then

$$\mathbf{P}[\, S_n \geq fn \,] = \mathbf{P}[\, P_n \geq \alpha^{fn} \,] \;.$$

Applying the Markov inequality (4.2) and (4.1), we get

$$\mathbf{P}[\, P_n \geq \alpha^{fn} \,] \leq \mathbf{E} P_n/\alpha^{fn} = (\mathbf{E} Y_1/\alpha^f)^n,$$

where $\mathbf{E} Y_1 = p\alpha + (1-p)$. Let us choose $\alpha = \frac{f(1-p)}{p(1-f)}$, this is > 1 if $p < f$. Then we get $\mathbf{E} Y_1 = \frac{1-p}{1-f}$, and hence

$$\mathbf{E} Y_1/\alpha^f = \frac{p^f(1-p)^{1-f}}{f^f(1-f)^{1-f}} = e^{-D(f,p)} \;.$$

■

This theorem also yields some convenient estimates for binomial coefficients. Let

$$h(f) = -f \ln f - (1-f) \ln(1-f) \;.$$

This is sometimes called the ***entropy*** of the probability distribution $(f, 1-f)$ (measured in logarithms over base e instead of base 2). From inequality (4.3) we obtain the estimate

$$-f \ln f \leq h(f) \leq f \ln \frac{e}{f} \tag{4.7}$$

which is useful for small f.

Corollary 4.2 *We have, for $f \leq 1/2$:*

$$\sum_{i \leq fn}^{n} \binom{n}{i} \leq e^{nh(f)} \leq \left(\frac{e}{f}\right)^{fn}. \tag{4.8}$$

In particular, taking $f = k/n$ with $k \leq n/2$ gives

$$\binom{n}{k} = \binom{n}{fn} \leq \left(\frac{e}{f}\right)^{fn} = \left(\frac{ne}{k}\right)^{k}. \tag{4.9}$$

Proof Theorem 4.1 says for the case $f > p = 1/2$:

$$2^{-n} \sum_{i \geq fn}^{n} \binom{n}{i} = \mathbf{P}[\, S_n \geq fn \,] \leq e^{-nD(f,p)} = 2^{-n} e^{nh(f)},$$

$$\sum_{i \geq fn}^{n} \binom{n}{i} \leq e^{nh(f)} \;.$$

Substituting $g = 1 - f$, and noting the symmetries $\binom{n}{f} = \binom{n}{g}$, $h(f) = h(g)$ and (4.7) gives (4.8).

■

Remark 4.3 *Inequality (4.6) also follows from the trivial estimate* $\mathbf{P}[\,S_n \ge fn\,] \le \binom{n}{fn} p^{fn}$ *combined with (4.9).*

Exercises

4.1-1 Prove that the statement made in the main text that $D(f,p)$ is always non-negative, and is 0 only if $f = p$.

4.1-2 For $f = p + \delta$, derive from Theorem 4.1 the useful bound

$$\mathbf{P}[\,S_n \ge fn\,] \le e^{-2\delta^2 n}\ .$$

Hint. Let $F(x) = D(x,p)$, and use the Taylor formula: $F(p+\delta) = F(p) + F'(p)\delta + F''(p+\delta')\delta^2/2$, where $0 \le \delta' \le \delta$.

4.1-3 Prove that in Theorem 4.1, the assumption that X_i are independent and identically distributed can be weakened: replaced by the single inequality

$$\mathbf{P}[\,X_i = 1 \mid X_1, \dots, X_{i-1}\,] \le p\ .$$

4.2. Logic circuits

In a model of computation taking errors into account, the natural assumption is that errors occur *everywhere*. The most familiar kind of computer, which is separated into a single processor and memory, seems extremely vulnerable under such conditions: while the processor is not “looking”, noise may cause irreparable damage in the memory. Let us therefore rather consider computation models that are *parallel*: information is being processed everywhere in the system, not only in some distinguished places. Then error correction can be built into the work of every part of the system. We will concentrate on the best known parallel computation model: Boolean circuits.

4.2.1. Boolean functions and expressions

Let us look inside a computer, (actually inside an integrated circuit, with a microscope). Discouraged by a lot of physical detail irrelevant to abstract notions of computation, we will decide to look at the blueprints of the circuit designer, at the stage when it shows the smallest elements of the circuit still according to their computational functions. We will see a network of lines that can be in two states (of electric potential), “high” or “low”, or in other words “true” or “false”, or, as we will write, 1 or 0. The points connected by these lines are the familiar ***logic components***: at the lowest level of computation, a typical computer processes ***bits***. Integers, floating-point numbers, characters are all represented as strings of bits, and the usual arithmetical operations can be composed of bit operations.

Definition 4.4 *A* ***Boolean vector function*** *is a mapping* $f : \{0,1\}^n \to \{0,1\}^m$. *Most of the time, we will take* $m = 1$ *and speak of a* ***Boolean function****.*

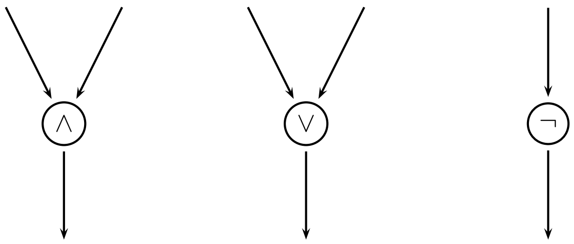


Figure 4.1 AND, OR and NOT gate.

The variables in $f(x_1, \dots, x_n)$ are sometimes called ***Boolean variables***, ***Boolean variables*** or ***bits***.

Example 4.2 Given an undirected graph G with N nodes, suppose we want to study the question whether it has a Hamiltonian cycle (a sequence $(u_1, \dots, u_n)$ listing all vertices of G such that (u_i, u_{i+1}) is an edge for each $i < n$ and also (u_n, u_1) is an edge). This question is described by a Boolean function f as follows. The graph can be described with $\binom{N}{2}$ Boolean variables x_{ij} $(1 \le i < j \le N)$: x_{ij} is 1 if and only if there is an edge between nodes i and j. We define $f(x_{12}, x_{13}, \dots, x_{N-1,N}) = 1$ if there is a Hamiltonian cycle in G and 0 otherwise.

Example 4.3[Boolean vector function] Let $n = m = 2k$, let the input be two integers u, v, written as k-bit strings: $x = (u_1, \dots, u_k, v_1, \dots, v_k)$. The output of the function is their product $y = u \cdot v$ (written in binary): if $u = 5 = (101)_2$, $v = 6 = (110)_2$ then $y = u \cdot v = 30 = (11110)_2$.

There are only four one-variable Boolean functions: the identically 0, identically 1, the identity and the ***negation***: $x \to \neg x = 1 - x$. We mention only the following two-variable Boolean functions: the operation of ***conjunction*** (logical AND):

$$x \wedge y = \begin{cases} 1 & \text{if } x = y = 1 \ , \\ 0 & \text{otherwise} \ , \end{cases}$$

this is the same as multiplication. The operation of ***disjunction***, or logical OR:

$$x \vee y = \begin{cases} 0 & \text{if } x = y = 0 \ , \\ 1 & \text{otherwise} \ . \end{cases}$$

It is easy to see that $x \vee y = \neg(\neg x \wedge \neg y)$: in other words, disjunction $x \vee y$ can be expressed using the functions $\neg, \wedge$ and the operation of ***composition***. The following two-argument Boolean functions are also frequently used:

$$\begin{aligned} x \to y &= \neg x \vee y && \text{(implication)} \ , \\ x \leftrightarrow y &= (x \to y) \wedge (y \to x) && \text{(equivalence)} \ , \\ x \oplus y &= x + y \bmod 2 = \neg(x \leftrightarrow y) && \text{(binary addition)} \ . \end{aligned}$$

A finite number of Boolean functions is sufficent to express all others: thus, arbitrarily complex Boolean functions can be "computed" by "elementary" operations. In some sense, this is what happens inside computers.

Definition 4.5 *A set of Boolean functions is a* ***complete basis*** *if every other Boolean function can be obtained by repeated composition from its elements.*

Claim 4.6 *The set* $\{\wedge, \vee, \neg\}$ *forms a complete basis; in other words, every Boolean function can be represented by a Boolean expression using only these connectives.*

The proof can be found in all elementary introductions to propositional logic. Note that since $\vee$ can be expressed using $\{\wedge, \neg\}$, this latter set is also a complete basis (and so is $\{\vee, \neg\}$).

From now on, under a ***Boolean expression*** (formula), we mean an expression built up from elements of some given complete basis. If we do not mention the basis then the complete basis $\{\wedge, \neg\}$ will be meant.

In general, one and the same Boolean function can be expressed in many ways as a Boolean expression. Given such an expression, it is easy to compute the value of the function. However, most Boolean functions can still be expressed only by very large Boolean expression (see Exercise 4.2-4).

4.2.2. Circuits

A Boolean expression is sometimes large since when writing it, there is no possibility for *reusing partial results.* (For example, in the expression

$$((x \vee y \vee z) \wedge u) \vee (\neg(x \vee y \vee z) \wedge v) ,$$

the part $x \vee y \vee z$ occurs twice.) This deficiency is corrected by the following more general formalism.

A Boolean circuit is essentially an acyclic directed graph, each of whose nodes computes a Boolean function (from some complete basis) of the bits coming into it on its input edges, and sends out the result on its output edges (see Figure 4.2). Let us give a formal definition.

Definition 4.7 *Let* Q *be a complete basis of Boolean functions. For an integer* N *let* $V = \{1, \dots, N\}$ *be a set of* ***nodes****. A* ***Boolean circuit*** *over* Q *is given by the following tuple:*

$$\mathcal{N} = (V, \{\, k_v : v \in V \,\}, \{\, \mathrm{arg}_j(v) : v \in V ; j = 1, \dots, k_v \,\}, \{\, b_v : v \in V \,\}) . \qquad (4.10)$$

For every node v *there is a natural number* k_v *showing its number of* ***inputs.*** *The* ***sources,*** *nodes* v *with* $k_v = 0$, *are called* ***input nodes****: we will denote them, in increasing order, as*

$$\mathrm{inp}_i \quad (i = 1, \dots, n) .$$

To each non-input node v *a Boolean function*

$$b_v(y_1, \dots, y_{k_v})$$

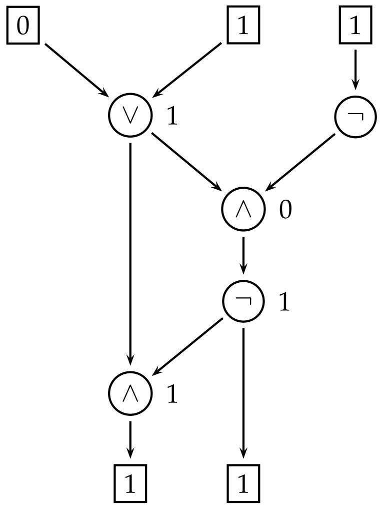


Figure 4.2 The assignment (values on nodes, configuration) gets propagated through all the gates. This is the "computation".

from the complete basis Q is assigned: it is called the ***gate*** *of node v. It has as many arguments as the number of entering edges. The* ***sinks*** *of the graph, nodes without outgoing edges, will be called* ***output nodes****: they can be denoted by*

$$\mathrm{out}_i \quad (i = 1, \dots, m) \ .$$

(Our Boolean circuits will mostly have just a single output node.) To every non-input node v and every $j = 1, \dots, k_v$ belongs a node $\arg_j(v) \in V$ (the node sending the value of input variable y_j of the gate of v). The circuit defines a graph $G = (V, E)$ whose set of edges is

$$E = \{\, (\arg_j(v), v) : v \in V,\ j = 1, \dots, k_v \,\} \ .$$

We require $\arg_j(v) < v$ for each j, v (we identified the with the natural numbers $1, \dots, N$): this implies that the graph G is acyclic. The ***size***

$$|\mathcal{N}|$$

of the circuit $\mathcal{N}$ is the number of nodes. The ***depth*** *of a node v is the maximal length of directed paths leading from an input node to v. The* ***depth*** *of a circuit is the maximum depth of its output nodes.*

Definition 4.8 *An* ***input assignment,*** *or* ***input configuration*** *to our circuit $\mathcal{N}$ is a vector $\boldsymbol{x} = (x_1, \dots, x_n)$ with $x_i \in \{0, 1\}$ giving value x_i to node inp_i:*

$$\mathrm{val}_{\boldsymbol{x}}(v) = y_v(\boldsymbol{x}) = x_i$$

for $v = \mathrm{inp}_i$, $i = 1, \dots, n$. The function $y_v(\boldsymbol{x})$ can be extended to a unique ***configuration*** *$v \mapsto y_v(\boldsymbol{x})$ on all other nodes of the circuit as follows. If gate b_v has k*

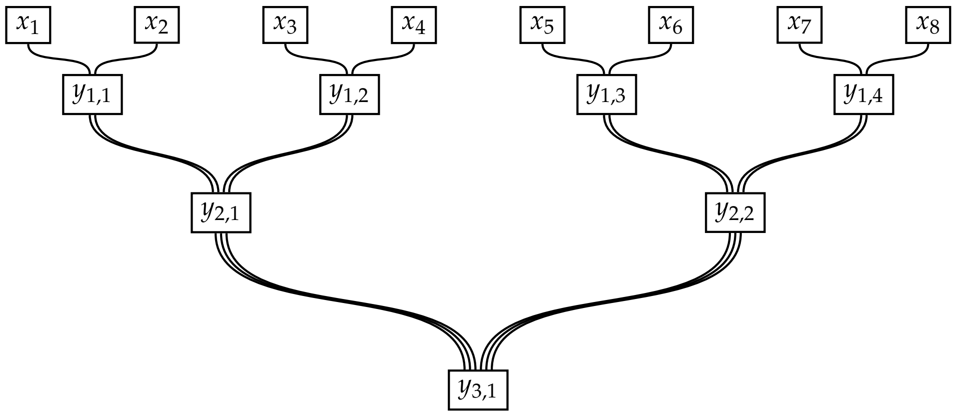


Figure 4.3 Naive parallel addition.

arguments then

$$y_v = b_v(y_{\arg_1(v)}, \dots, y_{\arg_k(v)}) . \tag{4.11}$$

For example, if $b_v(x, y) = x \wedge y$*, and* $u_j = \arg_j(v)$ *(*$j = 1, 2$*) are the input nodes to* v *then* $y_v = y_{u_1} \wedge y_{u_2}$*. The process of extending the configuration by the above equation is also called the* ***computation*** *of the circuit. The vector of the values* $y_{\mathrm{out}_i}(\boldsymbol{x})$ *for* $i = 1, \dots, m$ *is the* ***result*** *of the computation. We say that the Boolean circuit* ***computes*** *the vector function*

$$\boldsymbol{x} \mapsto (y_{\mathrm{out}_1}(\boldsymbol{x}), \dots, y_{\mathrm{out}_m}(\boldsymbol{x})) .$$

The assignment procedure can be performed in ***stages****: in stage* t*, all nodes of depth* t *receive their values.*

We assign values to the edges as well: the value assigned to an edge is the one assigned to its start node.

4.2.3. Fast addition by a Boolean circuit

The depth of a Boolean circuit can be viewed as the shortest time it takes to compute the output vector from the input vector by this circuit. Az an example application of Boolean circuits, let us develop a circuit that computes the sum of its input bits very fast. We will need this result later in the present chapter for error-correcting purposes.

Definition 4.9 *We will say that a Boolean circuit computes a* ***near-majority*** *if it outputs a bit* y *with the following property: if* $3/4$ *of all input bits is equal to* b *then* $y = b$*.*

The depth of our circuit is clearly $\Omega(\log n)$, since the output must have a path to the majority of inputs. In order to compute the majority, we will also solve the task of summing the input bits.

Theorem 4.10

(a) *Over the complete basis consisting of the set of all 3-argument Boolean functions, for each n there is a Boolean circuit of input size n and depth $\le 3\log(n+1)$ whose output vector represents the sum of the input bits as a binary number.*

b *Over this same complete basis, for each n there is a Boolean circuit of input size n and depth $\le 2\log(n+1)$ computing a near-majority.*

Proof. First we prove (4.10). For simplicity, assume $n = 2^k - 1$: if n is not of this form, we may add some fake inputs. The naive approach would be proceed according to Figure 4.3: to first compute $y_{1,1} = x_1 + x_2$, $y_{1,2} = x_3 + x_4$, ..., $y_{1,2^{k-1}} = x_{2^k-1} + x_{2^k}$. Then, to compute $y_{2,1} = y_{1,1} + y_{1,2}$, $y_{2,2} = y_{1,3} + y_{1,4}$, and so on. Then $y_{k,1} = x_1 + \cdots + x_{2^k}$ will indeed be computed in k stages.

It is somewhat troublesome that $y_{i,j}$ here is a number, not a bit, and therefore must be represented by a *bit vector*, that is by group of nodes in the circuit, not just by a single node. However, the general addition operation

$$y_{i+1,j} = y_{i,2j-1} + y_{i,2j} \ ,$$

when performed in the naive way, will typically take more than a constant number of steps: the numbers $y_{i,j}$ have length up to $i+1$ and therefore the addition may add i to the depth, bringing the total depth to $1 + 2 + \cdots + k = \Omega(k^2)$.

The following observation helps to decrease the depth. Let a, b, c be three numbers in binary notation: for example, $a = \sum_{i=0}^{k} a_i 2^i$. There are simple parallel formulas to represent the sum of these three numbers as the sum of two others, that is to compute $a + b + c = d + e$ where d, e are numbers also in binary notation:

$$\begin{aligned} d_i &= a_i + b_i + c_i \bmod 2 \ , \\ e_{i+1} &= \lfloor (a_i + b_i + c_i)/2 \rfloor \ . \end{aligned} \tag{4.12}$$

Since both formulas are computed by a single 3-argument gate, 3 numbers can be reduced to 2 (while preserving the sum) in a single parallel computation step. Two such steps reduce 4 numbers to 2. In $2(k-1)$ steps therefore they reduce a sum of 2^k terms to a sum of 2 numbers of length $\le k$. Adding these two numbers in the regular way increases the depth by k: we found that 2^k bits can be be added in $3k-2$ steps.

To prove (4.10), construct the circuit as in the proof of (4.10), but without the last addition: the output is two k-bit numbers whose sum we are interested in. The highest-order nonzero bit of these numbers is at some position $< k$. If the sum is more than 2^{k-1} then one these numbers has a nonzero bit at position $(k-1)$ or $(k-2)$. We can determine this in two applications of 3-input gates. ■

Exercises

4.2-1 Show that $\{1, \oplus, \wedge\}$ is a complete basis.

4.2-2 Show that the function $x \,\mathrm{NOR}\, y = \neg(x \vee y)$ forms a complete basis by itself.

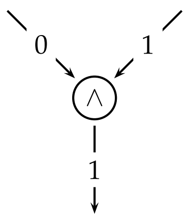


Figure 4.4 Failure at a gate.

4.2-3 Let us fix the complete basis $\{\wedge, \neg\}$. Prove Proposition 4.6 (or look up its proof in a textbook). Use it to give an upper bound for an arbitrary Boolean function f of n variables, on:

(a) the smallest size of a Boolean expression for f;

(b) the smallest size of a Boolean circuit for f;

(c) the smallest depth of a Boolean circuit for f;

4.2-4 Show that for every n there is a Boolean function f of n variables such that every Boolean circuit in the complete basis $\{\wedge, \neg\}$ computing f contains $\Omega(2^n/n)$ nodes. *Hint.* For a constant $c > 0$, upperbound the number of Boolean circuits with at most $c2^n/n$ nodes and compare it with the number of Boolean functions over n variables.]

4.2-5 Consider a circuit $\mathcal{M}_3^r$ with 3^r inputs, whose single output bit is computed from the inputs by r levels of 3-input majority gates. Show that there is an input vector $\boldsymbol{x}$ which is 1 in only $n^{1/\log 3}$ positions but with which $\mathcal{M}_3^r$ outputs 1. Thus a small minority of the inputs, when cleverly arranged, can command the result of this circuit.

4.3. Expensive fault-tolerance in Boolean circuits

Let $\mathcal{N}$ be a Boolean circuit as given in Definition 4.7. When noise is allowed then the values

$$y_v = \mathrm{val}_{\boldsymbol{x}}(v)$$

will not be determined by the formula (4.11) anymore. Instead, they will be random variables Y_v. The random assignment $(Y_v : v \in V)$ will be called a ***random configuration***.

Definition 4.11 *At vertex v, let*

$$Z_v = b_v(Y_{\mathrm{arg}_1(v)}, \dots, Y_{\mathrm{arg}_k(v)}) \oplus Y_v \ . \tag{4.13}$$

*In other words, $Z_v = 1$ if gate Y_v is not equal to the value computed by the noise-free gate b_v from its inputs $Y_{\mathrm{arg}_j(v)}$. (See Figure 4.4.) The set of vertices where Z_v is non-zero is the set of **faults**.*

*Let us call the difference $\mathrm{val}_{\boldsymbol{x}}(v) \oplus Y_v$ the **deviation** at node v.*

Let us impose conditions on the kind of noise that will be allowed. Each fault should occur only with probability at most ϵ, two specific faults should only occur with probability at most ϵ^2, and so on.

Definition 4.12 *For an $\epsilon > 0$, let us say that the random configuration $(Y_v : v \in V)$ is **ϵ-admissible** if*

(a) $Y_{\mathrm{inp}(i)} = x_i$ *for* $i = 1, \dots, n$.

(b) *For every set C of non-input nodes, we have*

$$\mathbf{P}[\, Z_v = 1 \text{ for all } v \in C \,] \le \epsilon^{|C|} \; . \tag{4.14}$$

In other words, in an ϵ-admissible random configuration, the probability of having faults at k different specific gates is at most ϵ^k. This is how we require that not only is the fault probability low but also, faults do not "conspire". The admissibility condition is satisfied if faults occur independently with probability $\le \epsilon$.

Our goal is to build a circuit that will work correctly, with high probability, despite the ever-present noise: in other words, in which errors *do not accumulate*. This concept is formalized below.

Definition 4.13 *We say that the circuit $\mathcal{N}$ with output node w is (ϵ, δ)-**resilient** if for all inputs $\boldsymbol{x}$, all ϵ-admissible configurations $\mathbf{Y}$, we have $\mathbf{P}[\, Y_w \ne \mathrm{val}_{\boldsymbol{x}}(w) \,] \le \delta$.*

Let us explore this concept. There is no (ϵ, δ)-resilient circuit with $\delta < \epsilon$, since even the last gate can fail with probability ϵ. So, let us, a little more generously, allow $\delta > 2\epsilon$. Clearly, for each circuit $\mathcal{N}$ and for each $\delta > 0$ we can choose ϵ small enough so that $\mathcal{N}$ is (ϵ, δ)-resilient. But this is not what we are after: hopefully, one does not need more reliable gates every time one builds a larger circuit. So, we hope to find a function

$$F(N, \delta)$$

and an $\epsilon_0 > 0$ with the property that for all $\epsilon < \epsilon_0$, $\delta \ge 2\epsilon$, every Boolean circuit $\mathcal{N}$ of size N there is some (ϵ, δ)-resilient circuit $\mathcal{N}'$ of size $F(N, \delta)$ computing the same function as $\mathcal{N}$. If we achieve this then we can say that we prevented the accumulation of errors. Of course, we want to make $F(N, \delta)$ relatively small, and ϵ_0 large (allowing more noise). The function $F(N, \delta)/N$ can be called the ***redundancy***: the factor by which we need to increase the size of the circuit to make it resilient. Note that the problem is nontrivial even with, say, $\delta = 1/3$. Unless the accumulation of errors is prevented we will lose gradually all information about the desired output, and no $\delta < 1/2$ could be guaranteed.

How can we correct errors? A simple idea is this: do "everything" 3 times and then continue with the result obtained by majority vote.

Definition 4.14 *For odd natural number d, a d-input majority gate is a Boolean function that outputs the value equal to the majority of its inputs.*

Note that a d-input majority can be computed using $O(d)$ gates of type AND and NOT.

Why should majority voting help? The following *informal discussion* helps understanding the benefits and pitfalls. Suppose for a moment that the output is a single bit. If the probability of each of the three independently computed results failing is δ then the probability that at least 2 of them fails is bounded by $3\delta^2$. Since the majority vote itself can fail with some probability ϵ the total probability of failure is bounded by $3\delta^2 + \epsilon$. We decrease the probability δ of failure, provided the condition $3\delta^2 + \epsilon < \delta$ holds.

We found that if δ is small, then repetition and majority vote can "make it" smaller. Of course, in order to keep the error probability from accumulating, we would have to perform this majority operation repeatedly. Suppose, for example, that our computation has t stages. Our bound on the probability of faulty output after stage i is δ_i. We plan to perform the majority operation after each stage i. Let us perform stage i three times. The probability of failure is now bounded by

$$\delta_{i+1} = \delta_i + 3\delta^2 + \epsilon \; . \tag{4.15}$$

Here, the error probabilities of the different stages accumulate, and even if $3\delta^2+\epsilon < \delta$ we only get a bound $\delta_t < (t-1)\delta$. So, this strategy will not work for arbitrarily large computations.

Here is a somewhat mad idea to avoid accumulation: repeat *everything* before the end of stage i three times, not only stage i itself. In this case, the growing bound (4.15) would be replaced with

$$\delta_{i+1} = 3(\delta_i + \delta)^2 + \epsilon \; .$$

Now if $\delta_i < \delta$ and $12\delta^2 + \epsilon < \delta$ then also $\delta_{i+1} < \delta$, so errors do not accumulate. But we paid an enormous price: the fault-tolerant version of the computation reaching stage $(i+1)$ is 3 times larger than the one reaching stage i. To make t stages fault-tolerant this way will cost a factor of 3^t in size. This way, the function $F(N, \delta)$ introduced above may become exponential in N.

The theorem below formalizes the above discussion.

Theorem 4.15 *Let R be a finite and complete basis for Boolean functions. If $2\epsilon \le \delta \le 0.01$ then every function can be computed by an (ϵ, δ)-resilient circuit over R.*

Proof. For simplicity, we will prove the result for a complete basis that contains the three-argument majority function and contains not functions with more than three arguments. We also assume that faults occur independently.

Let $\mathcal{N}$ be a noise-free circuit of depth t computing function f. We will prove that there is an (ϵ, δ)-resilient circuit $\mathcal{N}'$ of depth $2t$ computing f. The proof is by induction on t. The sufficient conditions on ϵ and δ will emerge from the proof.

The statement is certainly true for $t = 1$, so suppose $t > 1$. Let g be the output gate of the circuit $\mathcal{N}$, then $f(\boldsymbol{x}) = g(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), f_3(\boldsymbol{x}))$. The subcircuits $\mathcal{N}_i$ computing the functions f_i have depth $\le t - 1$. By the inductive assumption, there exist (ϵ, δ)-resilient circuits $\mathcal{N}_i'$ of depth $\le 2t - 2$ that compute f_i. Let $\mathcal{M}$ be a new circuit containing copies of the circuits $\mathcal{N}'_i$ (with the corresponding input nodes merged), with a new node in which $f(\boldsymbol{x})$ is computed as g is applied to the outputs of $\mathcal{N}'_i$. Then the probability of error of $\mathcal{M}$ is at most $3\delta + \epsilon < 4\delta$ if $\epsilon < \delta$ since each circuit $\mathcal{N}'_i$ can err with probability δ and the node with gate g can fail with probability ϵ.

Let us now form $\mathcal{N}'$ by taking three copies of $\mathcal{M}$ (with the inputs merged) and adding a new node computing the majority of the outputs of these three copies. The error probability of $\mathcal{N}'$ is at most $3(4\delta)^2 + \epsilon = 48\delta^2 + \epsilon$. Indeed, error will be due to either a fault at the majority gate or an error in at least two of the three independent copies of $\mathcal{M}$. So under condition

$$48\delta^2 + \epsilon \le \delta \; , \tag{4.16}$$

the circuit $\mathcal{N}'$ is (ϵ, δ)-resilient. This condition will be satisfied by $2\epsilon \le \delta \le 0.01$. ■

The circuit $\mathcal{N}'$ constructed in the proof above is at least 3^t times larger than $\mathcal{N}$. So, the redundancy is enormous. Fortunately, we will see a much more economical solution. But there are interesting circuits with small depth, for which the 3^t factor is not extravagant.

Theorem 4.16 *Over the complete basis consisting of all 3-argument Boolean functions, for all sufficiently small $\epsilon > 0$, if $2\epsilon \le \delta \le 0.01$ then for each n there is an (ϵ, δ)-resilient Boolean circuit of input size n, depth $\le 4\log(n+1)$ and size $(n+1)^7$ outputting a near-majority (as given in Definition 4.9).*

Proof. Apply Theorem 4.15 to the circuit from part (4.10) of Theorem 4.10: it gives a new, $4\log(n+1)$-deep (ϵ, δ)-resilient circuit computing a near-majority. The size of any such circuit with 3-input gates is at most $3^{4\log(n+1)} = (n+1)^{4\log 3} < (n+1)^7$. ■

Exercises

4.3-1 Exercise 4.2-5 suggests that the iterated majority vote $\mathcal{M}_3^r$ is not safe against manipulation. However, it works very well under some circumstances. Let the input to $\mathcal{M}_3^r$ be a vector $\boldsymbol{X} = (X_1, \ldots, X_n)$ of independent Boolean random variables with $\mathbf{P}[\, X_i = 1 \,] = p < 1/6$. Denote the (random) output bit of the circuit by Z. Assuming that our majority gates can fail with probability $\le \epsilon \le p/2$ independently, prove

$$\mathbf{P}[\, Z = 1 \,] \le \max\{10\epsilon,\ 0.3(p/0.3)^{2^k}\} \; .$$

Hint. Define $g(p) = \epsilon + 3p^2$, $g_0(p) = p$, $g_{i+1}(p) = g(g_i(p))$, and prove $\mathbf{P}[\, Z = 1 \,] \le g_r(p)$.]

4.3-2 We say that a circuit $\mathcal{N}$ computes the function $f(x_1, \ldots, x_n)$ in an (ϵ, δ)-***input-robust*** way, if the following holds: For any input vector $\boldsymbol{x} = (x_1, \ldots, x_n)$, for any vector $\boldsymbol{X} = (X_1, \ldots, X_n)$ of independent Boolean random variables "perturbing it" in the sense $\mathbf{P}[\, X_i \ne x_i \,] \le \epsilon$, for the output Y of circuit $\mathcal{N}$ on input $\boldsymbol{X}$ we have $\mathbf{P}[\, Y = f(\boldsymbol{x}) \,] \ge 1 - \delta$. Show that if the function $x_1 \oplus \cdots \oplus x_n$ is computable on an $(\epsilon, 1/4)$-input-robust circuit then $\epsilon \le 1/n$.

4.4. Safeguarding intermediate results

In this section, we will see ways to introduce fault-tolerance that scale up better. Namely, we will show:

Theorem 4.17 *There are constants R_0, ϵ_0 such that for*

$$F(n, \delta) = N \log(n/\delta) ,$$

for all $\epsilon < \epsilon_0$, $\delta \geq 3\epsilon$, for every deterministic computation of size N there is an (ϵ, δ)-resilient computation of size $R_0 F(N, \delta)$ with the same result.

Let us introduce a concept that will simplify the error analysis of our circuits, making it independent of the input vector x.

Definition 4.18 *In a Boolean circuit $\mathcal{N}$, let us call a majority gate at a node v a* ***correcting majority gate*** *if for every input vector $\boldsymbol{x}$ of $\mathcal{N}$, all input wires of node v have the same value. Consider a computation of such a circuit $\mathcal{N}$. This computation will make some nodes and wires of $\mathcal{N}$* ***tainted****. We define taintedness by the following rules:*

- *The input nodes are untainted.*
- *If a node is tainted then all of its output wires are tainted.*
- *A correcting majority gate is tainted if either it fails or a majority of its inputs are tainted.*
- *Any other gate is tainted if either it fails or one of its inputs is tainted.*

Clearly, if for all ϵ-admissible random configurations the output is tainted with probability $\leq \delta$ then the circuit is (ϵ, δ)-resilient.

4.4.1. Cables

So far, we have only made use of redundancy idea (4) of the introduction to the present chapter: repeating computation steps. Let us now try to use idea (4) (keeping information in redundant form) in Boolean circuits. To protect information traveling from gate to gate, we replace each wire of the noiseless circuit by a “cable” of k wires (where k will be chosen appropriately). Each wire within the cable is supposed to carry the same bit of information, and we hope that a majority will carry this bit even if some of the wires fail.

Definition 4.19 *In a Boolean circuit $\mathcal{N}'$, a certain set of edges is allowed to be called a* ***cable*** *if in a noise-free computation of this circuit, each edge carries the same Boolean value. The* ***width*** *of the cable is its number of elements. Let us fix an appropriate constant threshold ϑ. Consider any possible computation of the noisy version of the circuit $\mathcal{N}'$, and a cable of width k in $\mathcal{N}'$. This cable will be called* ***ϑ-safe*** *if at most ϑk of its wires are tainted.*

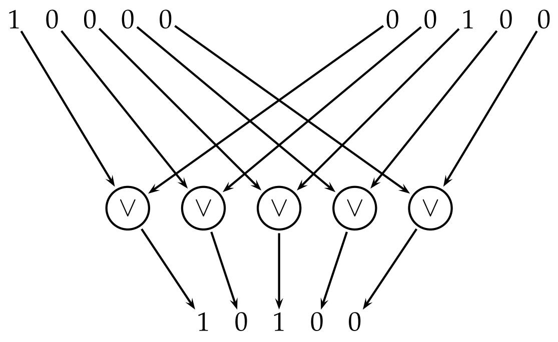


Figure 4.5 An executive organ.

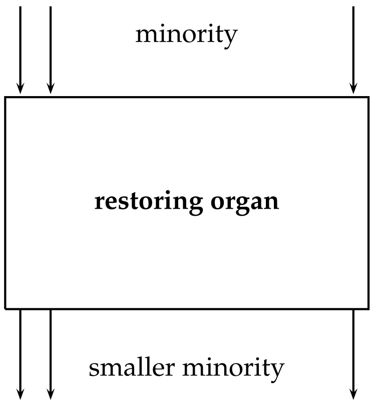


Figure 4.6 A restoring organ.

Let us take a Boolean circuit $\mathcal{N}$ that we want to make resilient. As we replace wires of $\mathcal{N}$ with cables of $\mathcal{N}'$ containing k wires each, we will replace each noiseless 2-argument gate at a node v by a module called the ***executive organ*** of k gates, which for each $i = 1, \dots, k$, passes the ith wire both incoming cables into the ith node of the organ. Each of these nodes contains a gate of one and the same type b_v. The wires emerging from these nodes form the ***output cable*** of the executive organ.

The number of tainted wires in this output cable may become too high: indeed, if there were ϑk tainted wires in the x cable and also in the y cable then there could be as many as $2\vartheta k$ such wires in the $g(x, y)$ cable (not even counting the possible new taints added by faults in the executive organ). The crucial part of the construction is to attach to the executive organ a so-called ***restoring organ***: a module intended to decrease the taint in a cable.

4.4.2. Compressors

How to build a restoring organ? Keeping in mind that this organ itself must also work in noise, one solution is to build (for an approriate δ') a special (ϵ, δ')-resilient circuit that computes the near-majority of its k inputs in k independent copies. Theorem 4.16 provides a circuit of size $k(k+1)^7$ to do this.

It turns out that, at least asymptotically, there is a better solution. We will look for a *very simple* restoring organ: one whose own noise we can analyse easily. What could be simpler than a circuit having only *one level* of gates? We fix an odd positive integer constant d (for example, $d = 3$). Each gate of our organ will be a d-input majority gate.

Definition 4.20 *A* ***multigraph*** *is a graph in which between any two vertices there may be several edges, not just 0 or 1. Let us call a bipartite multigraph with k inputs and k outputs, d-**half-regular.** if each output node has degree d. Such a graph is a (d, α, γ, k)**-compressor** if it has the following property: for every set E of at most $\leq \alpha k$ inputs, the number of those output points connected to at least $d/2$ elements of E (with multiplicity) is at most $\gamma\alpha k$.*

The compressor property is interesting generally when $\gamma < 1$. For example, in an $(5, 0.1, 0.5, k)$-compressor the outputs have degree 5, and the majority operation in these nodes decreases every error set confined to 10% of all input to just 5% of all outputs. A compressor with the right parameters could serve as our restoring organ: it decreases a minority to a smaller minority and may in this way restore the safety of a cable. But, are there compressors?

Theorem 4.21 *For all $\gamma < 1$, all integers d with*

$$1 < \gamma(d-1)/2\ , \tag{4.17}$$

there is an α such that for all integer $k > 0$ there is a (d, α, γ, k)-compressor.

Note that for $d = 3$, the theorem does not guarantee a compressor with $\gamma < 1$.
Proof We will not give an explicit construction for the multigraph, we will just show that it exists. We will select a d-half-regular multigraph randomly (each such multigraph with the same probability), and show that it will be a (d, α, γ, k)-compressor with positive probability. This proof method is called the ***probabilistic method***. Let

$$s = \lfloor d/2 \rfloor\ .$$

Our construction will be somewhat more general, allowing $k' \neq k$ outputs. Let us generate a random bipartite d-half-regular multigraph with k inputs and k' outputs in the following way. To each output, we draw edges from d random input nodes chosen independently and with uniform distribution over all inputs. Let A be an input set of size αk, let v be an output node and let E_v be the event that v has $s+1$ or more edges from A. Then we have

$$\mathbf{P}(E_v) \leq \binom{d}{s+1}\alpha^{s+1} = \binom{d}{s}\alpha^{s+1} =: p\ .$$

On the average (in expected value), the event E_v will occur for pk' different output nodes v. For an input set A, let F_A be the event that the set of nodes v for which E_v holds has size $> \gamma\alpha k'$. By inequality (4.6) we have

$$\mathbf{P}(F_A) \le \left(\frac{ep}{\gamma\alpha}\right)^{k'\gamma\alpha} .$$

The number M of sets A of inputs with $\le \alpha k$ elements is, using inequality (4.7),

$$M \le \sum_{i \le \alpha k} \binom{k}{i} \le \left(\frac{e}{\alpha}\right)^{\alpha k} .$$

The probability that our random graph is not a compressor is at most as large as the probability that there is at least one input set A for which event F_A holds. This can be bounded by

$$M \cdot \mathbf{P}(F_A) \le e^{-\alpha D k'}$$

where

$$D = -(\gamma s - k/k') \ln \alpha - \gamma\bigl(\ln \tbinom{d}{s} - \ln \gamma + 1\bigr) - k/k' .$$

As we decrease α the first term of this expression dominates. Its coefficient is positive according to the assumption (4.17). We will have $D > 0$ if

$$\alpha < \exp\left(-\frac{\gamma\bigl(\ln \binom{d}{s} - \ln \gamma + 1\bigr) + k/k'}{\gamma s - k/k'}\right) .$$

■

Example 4.4 Choosing $\gamma = 0.4$, $d = 7$, the value $\alpha = 10^{-7}$ will work.

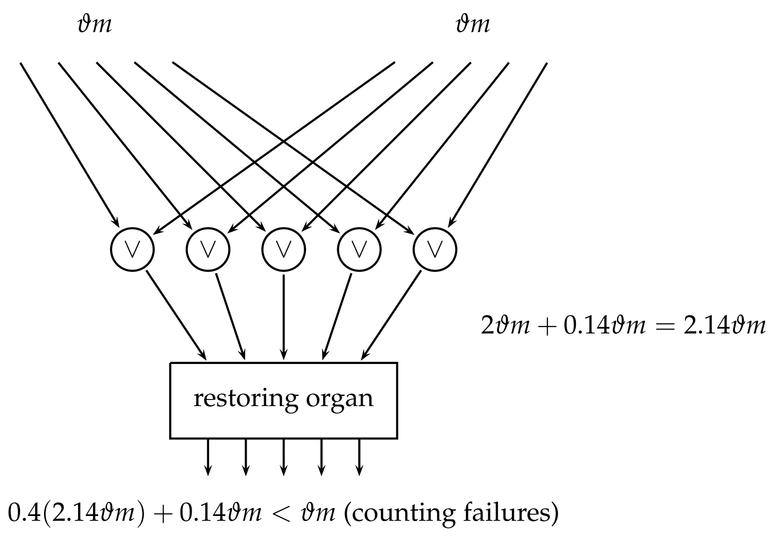
We turn a (d, α, γ, k)-compressor into a restoring organ $\mathcal{R}$, by placing d-input majority gates into its outputs. If the majority elements sometimes fail then the output of $\mathcal{R}$ is random. Assume that at most αk inputs of $\mathcal{R}$ are tainted. Then $(\gamma + \rho)\alpha k$ outputs can only be tainted if $\alpha\rho k$ majority gates fail. Let

$$p_{\mathcal{R}}$$

be the probability of this event. Assuming that the gates of $\mathcal{R}$ fail independently with probability $\le \epsilon$, inequality (4.6) gives

$$p_{\mathcal{R}} \le \left(\frac{e\epsilon}{\alpha\rho}\right)^{\alpha\rho k} . \tag{4.18}$$

Example 4.5 Choose $\gamma = 0.4$, $d = 7$, $\alpha = 10^{-7}$ as in Example 4.4, further $\rho = 0.14$ (this



ϑm
ϑm
$\vee$
restoring organ
$2\vartheta m + 0.14\vartheta m = 2.14\vartheta m$
$0.4(2.14\vartheta m) + 0.14\vartheta m < \vartheta m$ (counting failures)

Figure 4.7 An executive organ followed by a restoring organ.

will satisfy the inequality (4.19) needed later). With $\epsilon = 10^{-9}$, we get $p_{\mathcal{R}} \le e^{-10^{-8}k}$.

The attractively small degree $d = 7$ led to an extremely unattractive probability bound on the failure of the whole compressor. This bound does decrease exponentially with cable width k, but only an extremely large k would make it small.

Example 4.6 Choosing again $\gamma = 0.4$, but $d = 41$ (voting in each gate of the compressor over 41 wires instead of 7), leads to somewhat more realistic results. This choice allows $\alpha = 0.15$. With $\rho = 0.14$, $\epsilon = 10^{-9}$ again, we get $p_{\mathcal{R}} \le e^{-0.32k}$.

These numbers look less frightening, but we will still need many scores of wires in the cable to drive down the probability of compression failure. And although in practice our computing components fail with frequency much less than 10^{-9}, we may want to look at the largest ϵ that still can be tolerated.

4.4.3. Propagating safety

Compressors allow us to construct a reliable Boolean circuit all of whose cables are safe.

Definition 4.22 *Given a Boolean circuit $\mathcal{N}$ with a single bit of output (for simplicity), a cable width k and a Boolean circuit $\mathcal{R}$ with k inputs and k outputs, let*

$$\mathcal{N}' = \mathrm{Cab}(\mathcal{N}, \mathcal{R})$$

be the Boolean circuit that we obtain as follows. The input nodes of $\mathcal{N}'$ are the same as those of $\mathcal{N}$. We replace each wire of $\mathcal{N}$ with a cable of width k, and each gate of

$\mathcal{N}$ with an executive organ followed by a restoring organ that is a copy of the circuit $\mathcal{R}$. The new circuit has k outputs: the outputs of the restoring organ of $\mathcal{N}'$ belonging to the last gate of $\mathcal{N}$.

In noise-free computations, on every input, the output of $\mathcal{N}'$ is the same as the output of $\mathcal{N}$, but in k identical copies.

Lemma 4.23 *There are constants $d, \epsilon_0, \vartheta, \rho > 0$ and for every cable width k a circuit $\mathcal{R}$ of size $2k$ and gate size $\le d$ with the following property. For every Boolean circuit $\mathcal{N}$ of gate size ≤ 2 and number of nodes N, for every $\epsilon < \epsilon_0$, for every ϵ-admissible configuration of $\mathcal{N}' = \mathrm{Cab}(\mathcal{N}, \mathcal{R})$, the probability that not every cable of $\mathcal{N}'$ is ϑ-safe is $< 2N(\frac{e\epsilon}{\vartheta\rho})^{\vartheta\rho k}$.*

Proof We know that there are d, α and $\gamma < 1/2$ with the property that for all k a (d, α, γ, k)-compressor exists. Let ρ be chosen to satisfy

$$\gamma(2+\rho) + \rho \le 1 \ , \tag{4.19}$$

and define

$$\vartheta = \alpha/(2+\rho) \ . \tag{4.20}$$

Let $\mathcal{R}$ be a restoring organ built from a (d, α, γ, k)-compressor. Consider a gate v of circuit $\mathcal{N}$, and the corresponding executive organ and restoring organ in $\mathcal{N}'$. Let us estimate the probability of the event E_v that the input cables of this combined organ are ϑ-safe but its output cable is not. Assume that the two incoming cables are safe: then at most $2\vartheta k$ of the outputs of the executive organ are tainted due to the incoming cables: new taint can still occur due to failures. Let E_{v1} be the event that the executive organ taints at least $\rho\vartheta k$ more of these outputs. Then $\mathbf{P}(E_{v1}) \le (\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$, using the estimate (4.18). The outputs of the executive organ are the inputs of the restoring organ. If no more than $(2+\rho)\vartheta k = \alpha k$ of these are tainted then, in case the organ operates perfectly, it would decrease the number of tainted wires to $\gamma(2+\rho)\vartheta k$. Let E_{v2} be the event that the restoring organ taints an additional $\rho\vartheta k$ of these wires. Then again, $\mathbf{P}(E_{v2}) \le (\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$. If neither E_{v1} nor E_{v2} occur then at most $\gamma(2+\rho)\vartheta k + \rho\vartheta k \le \vartheta k$ (see (4.19)) tainted wires emerge from the restoring organ, so the outgoing cable is safe. Therefore $E_v \subset E_{v1} \cup E_{v2}$ and hence $\mathbf{P}(E_v) \le 2(\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$.

Let $V = \{1, \ldots, N\}$ be the nodes of the circuit $\mathcal{N}$. Since the incoming cables of the whole circuit $\mathcal{N}'$ are safe, the event that there is some cable that is not safe is contained in $E_1 \cup E_2 \cup \cdots \cup E_N$; hence the probability is bounded by $2N(\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$. ■

4.4.4. Endgame

ProofProof of Theorem 4.17 We will prove the theorem only for the case when our computation is a Boolean circuit with a single bit of output. The generalization with more bits of output is straightforward. The proof of Lemma 4.23 gives us a circuit $\mathcal{N}'$ whose output cable is safe except for an event of probability $< 2N(\frac{e\epsilon}{\rho\vartheta})^{\rho\vartheta k}$.

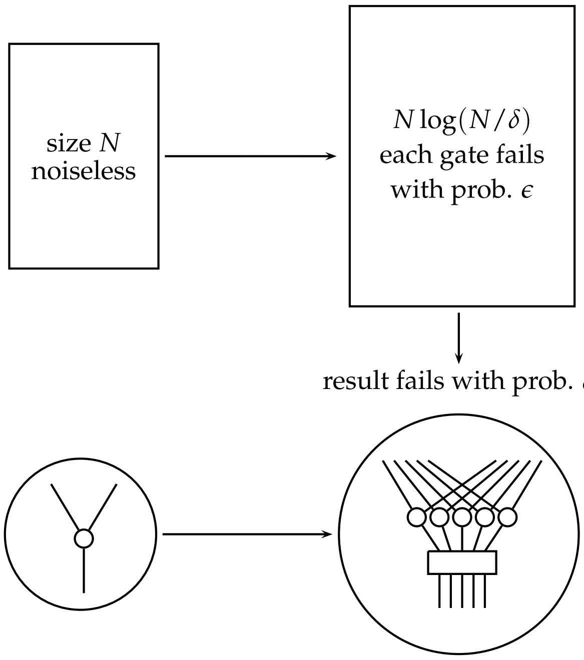


Figure 4.8 Reliable circuit from a fault-free circuit.

Let us choose k in such a way that this becomes $\leq \delta/3$:

$$k \geq \frac{\log(6N/\delta)}{\rho\vartheta \log \frac{\rho\vartheta}{e\epsilon_0}} . \tag{4.21}$$

It remains to add a little circuit to this output cable to extract from it the majority reliably. This can be done using Theorem 4.16, adding a small extra circuit of size $(k+1)^7$ that can be called the ***coda*** to $\mathcal{N}'$. Let us call the resulting circuit $\mathcal{N}''$.

The probability that the output cable is unsafe is $< \delta/3$. The probability that the output cable is safe but the "coda" circuit fails is bounded by 2ϵ. So, the probability that $\mathcal{N}''$ fails is $\leq 2\epsilon + \delta/3 \leq \delta$, by the assumption $\delta \geq 3\epsilon$.

Let us estimate the size of $\mathcal{N}''$. By (4.21), we can choose cable width $k = O(\log(N/\delta))$. We have $|\mathcal{N}'| \leq 2kN$, hence

$$|\mathcal{N}''| \leq 2kN + (k+1)^7 = O(N\log(N/\delta)).$$

■

Example 4.7 Take the constants of Example 4.6, with ϑ defined in equation (4.20): then $\epsilon_0 = 10^{-9}$, $d = 41$, $\gamma = 0.4$, $\rho = 0.14$, $\alpha = 0.15$, $\vartheta = 0.07$, giving

$$\frac{1}{\rho\vartheta \ln \frac{\rho\vartheta}{e\epsilon_0}} \approx 6.75 ,$$

so making k as small as possible (ignoring that it must be integer), we get $k \approx 6.75 \ln(N/\delta)$. With $\delta = 10^{-8}$, $N = 10^{12}$ this allows $k = 323$. In addition to this truly unpleasant cable size, the size of the "coda" circuit is $(k+1)^7 \approx 4 \cdot 10^{17}$, which dominates the size of the rest of $\mathcal{N}''$ (though as $N \to \infty$ it becomes asymptotically negligible).

As Example 4.7 shows, the actual price in redundancy computable from the proof is unacceptable in practice. The redundancy $O(\lg(N/\delta))$ sounds good, since it is only logarithmic in the size of the computation, and by choosing a rather large majority gate (41 inputs), the factor 6.75 in the $O(\cdot)$ here also does not look bad; still, we do not expect the final price of reliability to be this high. How much can this redundancy improved by optimization or other methods? Problem 4-6 shows that in a slightly more restricted error model (all faults are independent and have the same probability), with more randomization, better constants can be achieved. Exercises 4.4-1, 4.4-2 and 4.4-5 are concerned with an improved construction for the "coda" circuit. Exercise 4.5-2 shows that the coda circuit can be omitted completely. But none of these improvements bring redundancy to acceptable level. Even aside from the discomfort caused by their random choice (this can be helped), concentrators themselves are rather large and unwieldy. The problem is probably with using circuits as a model for computation. There is no natural way to break up a general circuit into subunits of non-constant size in order to deal with the reliability problem in modular style.

4.4.5. The construction of compressors

This subsection is sketchier than the preceding ones, and assumes some knowledge of linear algebra.

We have shown that compressors exist. How expensive is it to find a (d, α, γ, k)-compressor, say, with $d = 41$, $\alpha = 0.15$, $\gamma = 0.4$, as in Example 4.6? In a deterministic algorithm, we could search through all the approximately d^k d-half-regular bipartite graphs. For each of these, we could check all possible input sets of size $\leq \alpha k$: as we know, their number is $\leq (e/\alpha)^{\alpha k} < 2^k$. The cost of checking each subset is $O(k)$, so the total number of operations is $O(k(2d)^k)$. Though this number is exponential in k, recall that in our error-correcting construction, $k = O(\log(N/\delta))$ for the size N of the noiseless circuit: therefore the total number of operations needed to find a compressor is polynomial in N.

The proof of Theorem 4.21 shows that a randomly chosen d-half-regular bipartite graph is a compressor with large probability. Therefore there is a faster, randomized algorithm for finding a compressor. Pick a random d-half-regular bipartite graph, check if it is a compressor: if it is not, repeat. We will be done in a constant expected number of repetititons. This is a faster algorithm, but is still exponential in k, since each checking takes $\Omega(k(e/\alpha)^{\alpha k})$ operations.

Is it possible to construct a compressor explicitly, avoiding any search that takes exponential time in k? The answer is yes. We will show here only, however, that the compressor property is implied by a certain property involving linear algebra, which can be checked in polynomial time. Certain explicitly constructed graphs are known that possess this property. These are generally sought after not so much for their compressor property as for their *expander* property (see the section on reliable storage).

For vectors $\boldsymbol{v}, \boldsymbol{w}$, let $(\boldsymbol{v}, \boldsymbol{w})$ denote their inner product. A d-half-regular bipartite multigraph with $2k$ nodes can be defined by an ***incidence matrix*** $\boldsymbol{M} = (m_{ij})$, where m_{ij} is the number of edges connecting input j to output i. Let $\boldsymbol{e}$ be the vector $(1, 1, \dots, 1)^T$. Then $\boldsymbol{M}\boldsymbol{e} = d\boldsymbol{e}$, so $\boldsymbol{e}$ is an ***eigenvector*** of $\boldsymbol{M}$ with ***eigenvalue*** d. Moreover, d is the largest eigenvalue of $\boldsymbol{M}$. Indeed, denoting by $|\boldsymbol{x}|_1 = \sum_i |x_i|$ for any row vector $\boldsymbol{x} = (x_1, \dots, x_k)$, we have $|\boldsymbol{x}\boldsymbol{M}|_1 \le |\boldsymbol{x}|_1$.

Theorem 4.24 *Let G be a multigraph defined by the matrix $\boldsymbol{M}$. For all $\gamma > 0$, and*

$$\mu < d\sqrt{\gamma}/2, \tag{4.22}$$

there is an $\alpha > 0$ such that if the second largest eigenvalue of the matrix $\boldsymbol{M}^T\boldsymbol{M}$ is μ^2 then G is a (d, α, γ, k)-compressor.

Proof The matrix $\boldsymbol{M}^T\boldsymbol{M}$ has largest eigenvalue d^2. Since it is symmetric, it has a basis of orthogonal eigenvectors $\boldsymbol{e}_1, \dots, \boldsymbol{e}_k$ of unit length with corresponding non-negative eigenvalues

$$\lambda_1^2 \ge \dots \ge \lambda_k^2$$

where $\lambda_1 = d$ and $\boldsymbol{e}_1 = \boldsymbol{e}/\sqrt{k}$. Recall that in the orthonormal basis $\{\boldsymbol{e}_i\}$, any vector $\boldsymbol{f}$ can be written as $\boldsymbol{f} = \sum_i (\boldsymbol{f}, \boldsymbol{e}_i)\boldsymbol{e}_i$. For an arbitrary vector $\boldsymbol{f}$, we can estimate $|\boldsymbol{M}\boldsymbol{f}|^2$ as follows.

$$\begin{aligned}
|\boldsymbol{M}\boldsymbol{f}|^2 = (\boldsymbol{M}\boldsymbol{f}, \boldsymbol{M}\boldsymbol{f}) = (\boldsymbol{f}, \boldsymbol{M}^T\boldsymbol{M}\boldsymbol{f}) = \sum_i \lambda_i^2 (\boldsymbol{f}, \boldsymbol{e}_i)^2 \\
\le d^2(\boldsymbol{f}, \boldsymbol{e}_1)^2 + \mu^2 \sum_{i>1} (\boldsymbol{f}, \boldsymbol{e}_i)^2 \le d^2(\boldsymbol{f}, \boldsymbol{e}_1)^2 + \mu^2(\boldsymbol{f}, \boldsymbol{f}) \\
= d^2(\boldsymbol{f}, \boldsymbol{e})^2/k + \mu^2(\boldsymbol{f}, \boldsymbol{f}) \ .
\end{aligned}$$

Let now $A \subset \{1, \dots, k\}$ be a set of size αk and $\boldsymbol{f} = (f_1, \dots, f_k)^T$ where $f_j = 1$ for $j \in A$ and 0 otherwise. Then, coordinate i of $\boldsymbol{M}\boldsymbol{f}$ counts the number d_i of edges coming from the set A to the node i. Also, $(\boldsymbol{f}, \boldsymbol{e}) = (\boldsymbol{f}, \boldsymbol{f}) = |A|$, the number of elements of A. We get

$$\begin{aligned}
\sum_i d_i^2 = |\boldsymbol{M}\boldsymbol{f}|^2 \le d^2(\boldsymbol{f}, \boldsymbol{e})^2/k + \mu^2(\boldsymbol{f}, \boldsymbol{f}) = d^2\alpha^2 k + \mu^2 \alpha k, \\
k^{-1} \sum_i (d_i/d)^2 \le \alpha^2 + (\mu/d)^2 \alpha \ .
\end{aligned}$$

Suppose that there are $c\alpha k$ nodes i with $d_i > d/2$, then this says

$$c\alpha \le 4(\mu/d)^2 \alpha + 4\alpha^2 \ .$$

Since (4.22) implies $4(\mu/d)^2 < \gamma$, it follows that $\boldsymbol{M}$ is a $(d, \alpha, \gamma, k, k)$-compressor for small enough α. ■

It is actually sufficient to look for graphs with large k and $\mu/d < c < 1$ where d, c are constants. To see this, let us define the product of two bipartite multigraphs with $2k$ vertices by the multigraph belonging to the product of the corresponding matrices.

Suppose that $\boldsymbol{M}$ is symmetric: then its second largest eigenvalue is μ and the ratio of the two largest eigenvalues of $\boldsymbol{M}^r$ is $(\mu/d)^r$. Therefore using $\boldsymbol{M}^r$ for a sufficiently large r as our matrix, the condition (4.22) can be satisfied. Unfortunately, taking the power will increase the degree d, taking us probably even farther away from practical realizability.

We found that there is a construction of a compressor with the desired parameters as soon as we find multigraphs with arbitrarily large sizes $2k$, with symmetric matrices $\boldsymbol{M}_k$ and with a ratio of the two largest eigenvalues of $\boldsymbol{M}_k$ bounded by a constant $c < 1$ independent of k. There are various constructions of such multigraphs (see the references in the historical overview). The estimation of the desired eigenvalue quotient is never very simple.

Exercises

4.4-1 The proof of Theorem 4.17 uses a "coda" circuit of size $(k+1)^7$. Once we proved this theorem we could, of course, apply it to the computation of the final majority itself: this would reduce the size of the coda circuit to $O(k \log k)$. Try out this approach on the numerical examples considered above to see whether it results in a significant improvement.

4.4-2 The proof of Theorem 4.21 provided also bipartite graphs with the compressor property, with k inputs and $k' < 0.8k$ outputs. An idea to build a smaller "coda" circuit in the proof of Theorem 4.17 is to concatenate several such compressors, decreasing the number of cables in a geometric series. Explore this idea, keeping in mind, however, that as k decreases, the "exponential" error estimate in inequality (4.18) becomes weaker.

4.4-3 In a noisy Boolean circuit, let $F_v = 1$ if the gate at vertex v fails and 0 otherwise. Further, let $T_v = 1$ if v is tainted, and 0 otherwise. Suppose that the distribution of the random variables F_v does not depend on the Boolean input vector. Show that then the joint distribution of the random variables T_v is also independent of the input vector.

4.4-4 This exercise extends the result of Exercise 4.3-1 to random input vectors: it shows that if a random input vector has only a small number of errors, then the iterated majority vote $\mathcal{M}_3^r$ of Exercise 4.2-5 may still work for it, if we rearrange the input wires randomly. Let $k = 3^r$, and let $\boldsymbol{j} = (j_1, \dots, j_k)$ be a vector of integers $j_i \in \{1, \dots, k\}$. We define a Boolean circuit $C(\boldsymbol{j})$ as follows. This circuit takes input vector $\boldsymbol{x} = (x_1, \dots, x_k)$, computes the vector $\boldsymbol{y} = (y_1, \dots, y_k)$ where $y_i = x_{j_i}$ (in other words, just leads a wire from input node j_i to an "intermediate node" i) and then inputs $\boldsymbol{y}$ into the circuit $\mathcal{M}_3^r$.

Denote the (possibly random) output bit of $C(\boldsymbol{j})$ by Z. For any fixed input vector $\boldsymbol{x}$, assuming that our majority gates can fail with probability $\le \epsilon \le \alpha/2$ independently, denote $q(\boldsymbol{j}, \boldsymbol{x}) := \mathbf{P}[\, Z = 1 \,]$. Assume that the input is a vector $\boldsymbol{X} = (X_1, \dots, X_k)$ of (not necessarily independent) Boolean random variables, with

$p(\boldsymbol{x}) := \mathbf{P}[\,\boldsymbol{X} = \boldsymbol{x}\,]$. Denoting $|\boldsymbol{X}| = \sum_i X_i$, assume $\mathbf{P}[\,|\boldsymbol{X}| > \alpha k\,] \le \rho < 1$. Prove that there is a choice of the vector $\boldsymbol{j}$ for which

$$\sum_{\boldsymbol{x}} p(\boldsymbol{x}) q(\boldsymbol{j}, \boldsymbol{x}) \le \rho + \max\{10\epsilon,\ 0.3(\alpha/0.3)^{2^k}\}\ .$$

The choice may depend on the distribution of the random vector $\boldsymbol{X}$. *Hint.* Choose the vector $\boldsymbol{j}$ (and hence the circuit $C(\boldsymbol{j})$) randomly, as a random vector $\boldsymbol{J} = (J_1, \dots, J_k)$ where the variables J_i are independent and uniformly distributed over $\{1, \dots, k\}$, and denote $s(\boldsymbol{j}) := \mathbf{P}[\,\boldsymbol{J} = \boldsymbol{j}\,]$. Then prove

$$\sum_{j} s(\boldsymbol{j}) \sum_{\boldsymbol{x}} p(\boldsymbol{x}) q(\boldsymbol{j}, \boldsymbol{x}) \le \rho + \max\{10\epsilon,\ 0.3(\alpha/0.3)^{2^k}\}.$$

For this, interchange the averaging over $\boldsymbol{x}$ and $\boldsymbol{j}$. Then note that $\sum_j s(\boldsymbol{j}) q(\boldsymbol{j}, \boldsymbol{x})$ is the probability of $Z = 1$ when the "wires" J_i are chosen randomly "on the fly" during the computation of the circuit.]

4.4-5 Taking the notation of Exercise 4.4-3 suppose, like there, that the random variables F_v are independent of each other, and their distribution does not depend on the Boolean input vector. Take the Boolean circuit $\mathrm{Cab}(\mathcal{N}, \mathcal{R})$ introduced in Definition 4.22, and define the random Boolean vector $\boldsymbol{T} = (T_1, \dots, T_k)$ where $T_i = 1$ if and only if the ith output node is tainted. Apply Exercise 4.4-4 to show that there is a circuit $C(\boldsymbol{j})$ that can be attached to the output nodes to play the role of the "coda" circuit in the proof of Theorem 4.17. The size of $C(\boldsymbol{j})$ is only linear in k, not $(k+1)^7$ as for the coda circuit in the proof there. But, we assumed a little more about the fault distribution, and also the choice of the "wiring" $\boldsymbol{j}$ depends on the circuit $\mathrm{Cab}(\mathcal{N}, \mathcal{R})$.

4.5. The reliable storage problem

There is hardly any simpler computation than not doing anything, just keeping the input unchanged. This task does not fit well, however, into the simple model of Boolean circuits as introduced above.

4.5.1. Clocked circuits

An obvious element of ordinary computations is missing from the above described Boolean circuit model: *repetition*. If we want to repeat some computation steps, then we need to introduce *timing* into the work of computing elements and to *store* the partial results between consecutive steps. Let us look at the drawings of the circuit designer again. We will see components like in Figure 4.9, with one ingoing edge and no operation associated with them; these will be called ***shift registers***. The shift registers are controlled by one central ***clock*** (invisible on the drawing). At each clock pulse, the assignment value on the incoming edge jumps onto the outgoing edges and "stays in" the register. Figure 4.10 shows how shift registers may be used inside a circuit.

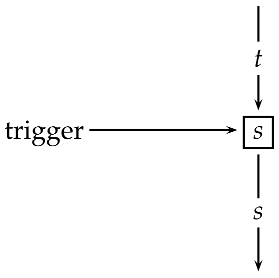


Figure 4.9 A shift register.

Definition 4.25 *A **clocked circuit** over a complete basis Q is given by a tuple just like a Boolean circuit in (4.10). Also, the circuit defines a graph $G = (V, E)$ similarly. Recall that we identified nodes with the natural numbers $1, \dots, N$. To each non-input node v either a gate b_v is assigned as before, or a **shift register:** in this case $k_v = 1$ (there is only one argument). We do not require the graph to be acyclic, but we do require every directed cycle (if there is any) to pass through at least one shift register.*

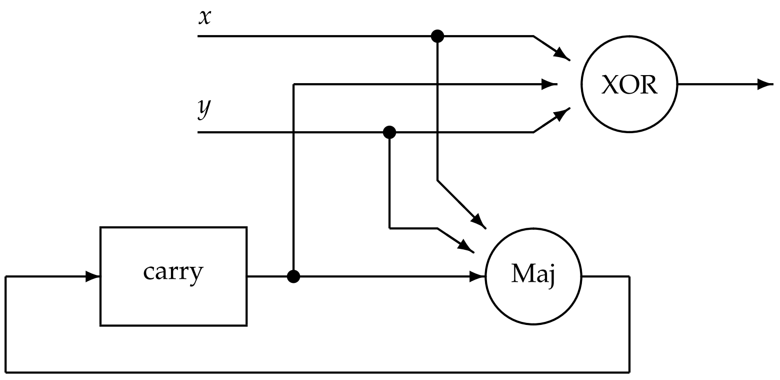


Figure 4.10 Part of a circuit which computes the sum of two binary numbers x, y. We feed the digits of x and y beginning with the lowest-order ones, at the input nodes. The digits of the sum come out on the output edge. A shift register holds the carry.

*The circuit works in a sequence $t = 0, 1, 2, \dots$ of **clock cycles**. Let us denote the input vector at clock cycle t by $\boldsymbol{x}^t = (x_1^t, \dots, x_n^t)$, the shift register states by $\boldsymbol{s}^t = (s_1^t, \dots, s_k^t)$, and the output vector by $\boldsymbol{y}^t = (y_1^t, \dots, y_m^t)$. The part of the circuit*

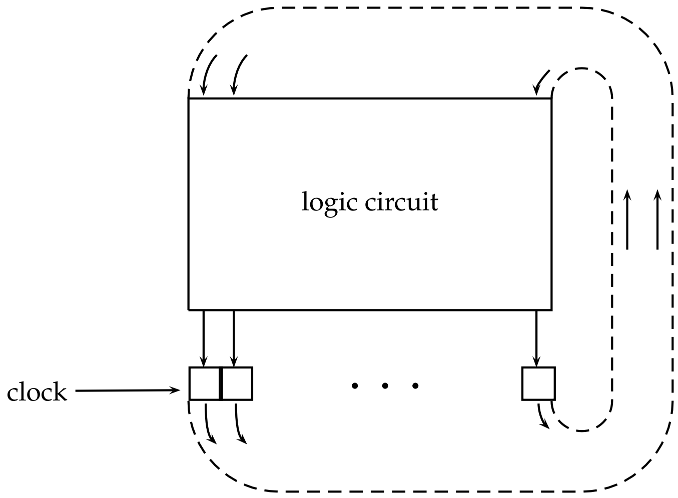


Figure 4.11 A "computer" consists of some memory (shift registers) and a Boolean circuit operating on it. We can define the ***size of computation*** as the size of the computer times the number of steps.

going from the inputs and the shift registers to the outputs and the shift registers defines two Boolean vector functions $\lambda : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^m$ *and* $\tau : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^k$. *The operation of the clocked circuit is described by the following equations (see Figure 4.11, which does not show any inputs and outputs).*

$$\boldsymbol{y}^t = \lambda(\boldsymbol{s}^t, \boldsymbol{x}^t), \quad \boldsymbol{s}^{t+1} = \tau(\boldsymbol{s}^t, \boldsymbol{x}^t) \ . \tag{4.23}$$

Frequently, we have no inputs or outputs during the work of the circuit, so the equations (4.23) can be simplified to

$$\boldsymbol{s}^{t+1} = \tau(\boldsymbol{s}^t) \ . \tag{4.24}$$

How to use a clocked circuit described by this equation for computation? We write some initial values into the shift registers, and propagate the assignment using the gates, for the given clock cycle. Now we send a clock pulse to the register, causing it to write new values to their output edges (which are identical to the input edges of the circuit). After this, the new assignment is computed, and so on.

How to compute a *function* $f(x)$ with the help of such a circuit? Here is a possible convention. We enter the input x (only in the first step), and then run the circuit, until it signals at an extra output edge when desired result $f(x)$ can be received from the other output nodes.

Example 4.8 This example uses a convention different from the above described one: new input bits are supplied in every step, and the output is also delivered continuously. For the binary adder of Figure 4.10, let u^t and v^t be the two input bits in cycle t, let c^t be the content of the carry, and w^t be the output in the same cycle. Then the equations (4.23) now have the form

$$w^t = u^t \oplus v^t \oplus c^t, \quad c^{t+1} = \mathrm{Maj}(u^t, v^t, c^t) ,$$

where Maj is the majority operation.

4.5.2. Storage

A clocked circuit is an interesting parallel computer but let us pose now a task for it that is trivial in the absence of failures: information storage. We would like to store a certain amount of information in such a way that it can be recovered after some time, despite failures in the circuit. For this, the transition function τ introduced in (4.24) cannot be just the identity: it will have to perform some *error-correcting* operations. The restoring organs discussed earlier are natural candidates. Indeed, suppose that we use k memory cells to store a bit of information. We can call the content of this k-tuple ***safe*** when the number of memory cells that dissent from the correct value is under some treshold ϑk. Let the rest of the circuit be a restoring organ built on a (d, α, γ, k)-compressor with $\alpha = 0.9\vartheta$. Suppose that the input cable is safe. Then the probability that after the transition, the new output cable (and therefore the new state) is not safe is $O(e^{-ck})$ for some constant c. Suppose we keep the circuit running for t steps. Then the probability that the state is not safe in some of these steps is $O(te^{-ck})$ which is small as long as t is significantly smaller than e^{ck}. When storing m bits of information, the probability that any of the bits loses its safety in some step is $O(mte^{-cm})$.

To make this discussion rigorous, an error model must be introduced for clocked circuits. Since we will only consider simple transition functions τ like the majority vote above, with a single computation step between times t and $t + 1$, we will make the model also very simple.

Definition 4.26 *Consider a clocked circuit described by equation* (4.24)*, where at each time instant* $t = 0, 1, 2, \ldots$*, the configuration is described by the bit vector* $\boldsymbol{s}^t = (s^t_1, \ldots, s^t_n)$*. Consider a sequence of random bit vectors* $\boldsymbol{Y}^t = (Y^t_1, \ldots, Y^t_n)$ *for* $t = 0, 1, 2, \ldots$*. Similarly to* (4.13) *we define*

$$Z_{i,t} = \tau(\boldsymbol{Y}^{t-1}) \oplus Y^t_i \; . \tag{4.25}$$

Thus, $Z_{i,t} = 1$ *says that a* ***failure*** *occurs at the space-time point* (i, t)*. The sequence* $\{\boldsymbol{Y}^t\}$ *will be called* ϵ-***admissible*** *if* (4.14) *holds for every set* C *of space-time points with* $t > 0$.

By the just described construction, it is possible to keep m bits of information for T steps in

$$O(m \lg(mT)) \tag{4.26}$$

memory cells. More precisely, the cable $\boldsymbol{Y}^T$ will be safe with large probability in any admissible evolution $\boldsymbol{Y}^t$ $(t = 0, \ldots, T)$.

Cannot we do better? The reliable information storage problem is related to the problem of ***information transmission:*** given a ***message*** $\boldsymbol{x}$, a ***sender*** wants to transmit it to a ***receiver*** throught a ***noisy channel.*** Only now sender and receiver are the same person, and the noisy channel is just the passing of time. Below, we develop some basic concepts of reliable information transmission, and then we will apply them to the construction of a reliable data storage scheme that is more economical than the above seen naive, repetition-based solution.

4.5.3. Error-correcting codes

Error detection To protect information, we can use redundancy in a way more efficient than repetition. We might even add only a single redundant bit to our message. Let $\boldsymbol{x} = (x_1, \dots, x_6)$, $(x_i \in \{0, 1\})$ be the word we want to protect. Let us create the ***error check bit***

$$x_7 = x_1 \oplus \cdots \oplus x_6 \ .$$

For example, $\boldsymbol{x} = 110010$, $\boldsymbol{x}' = 1100101$. Our ***codeword*** $\boldsymbol{x}' = (x_1, \dots, x_7)$ will be subject to noise and it turns into a new word, $\boldsymbol{y}$. If $\boldsymbol{y}$ differs from $\boldsymbol{x}'$ in a single changed (not deleted or added) bit then we will ***detect*** this, since then $\boldsymbol{y}$ violates the ***error check relation***

$$y_1 \oplus \cdots \oplus y_7 = 0 \ .$$

We will not be able to correct the error, since we do not know which bit was corrupted.

Correcting a single error. To also ***correct*** corrupted bits, we need to add more error check bits. We may try to add two more bits:

$$\begin{aligned} x_8 &= x_1 \oplus x_3 \oplus x_5 \ , \\ x_9 &= x_1 \oplus x_2 \oplus x_5 \oplus x_6 \ . \end{aligned}$$

Then an uncorrupted word $\boldsymbol{y}$ must satisfy the error check relations

$$\begin{aligned} y_1 \oplus \cdots \oplus y_7 &= 0 \ , \\ y_1 \oplus y_3 \oplus y_5 \oplus y_8 &= 0 \ , \\ y_1 \oplus y_2 \oplus y_5 \oplus y_6 \oplus y_9 &= 0 \ , \end{aligned}$$

or, in matrix notation $\boldsymbol{H}\boldsymbol{y} \bmod 2 = 0$, where

$$\boldsymbol{H} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} = (\boldsymbol{h}_1, \dots, \boldsymbol{h}_9) \ .$$

Note $\boldsymbol{h}_1 = \boldsymbol{h}_5$. The matrix $\boldsymbol{H}$ is called the ***error check matrix***, or ***parity check matrix***.

Another way to write the error check relations is

$$y_1\boldsymbol{h}_1 \oplus \cdots \oplus y_5\boldsymbol{h}_5 \oplus \cdots \oplus y_9\boldsymbol{h}_9 = 0.$$

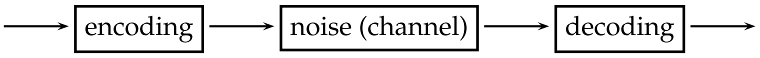


Figure 4.12 Transmission through a noisy channel.

Now if $\boldsymbol{y}$ is corrupted, even if only in a single position, unfortunately we still cannot correct it: since $\boldsymbol{h}_1 = \boldsymbol{h}_5$, the error could be in position 1 or 5 and we could not tell the difference. If we choose our error-check matrix $\boldsymbol{H}$ in such a way that the colum vectors $\boldsymbol{h}_1, \boldsymbol{h}_2, \dots$ are *all different* (of course also from 0), then we can always correct an error, provided there is only one. Indeed, if the error was in position 3 then

$$\boldsymbol{H}\boldsymbol{y} \bmod 2 = \boldsymbol{h}_3.$$

Since all vectors $\boldsymbol{h}_1, \boldsymbol{h}_2, \dots$ are different, if we see the vector $\boldsymbol{h}_3$ we can imply that the bit y_3 is corrupted. This code is called the ***Hamming code.*** For example, the following error check matrix defines the Hamming code of size 7:

$$\boldsymbol{H} = \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} = (\boldsymbol{h}_1, \dots, \boldsymbol{h}_7) \ . \tag{4.27}$$

In general, if we have s error check bits then our code can have size $2^s - 1$, hence the number of bits left to store information, the ***information bits*** is $k = 2^s - s - 1$. So, to protect m bits of information from a single error, the Hamming code adds $\approx \log m$ error check bits. This is much better than repeating every bit 3 times.

Codes. Let us summarize the error-correction scenario in general terms. In order to fight noise, the sender ***encodes*** the ***message*** $\boldsymbol{x}$ by an ***encoding function*** ϕ_* into a longer string $\phi_*(\boldsymbol{x})$ which, for simplicity, we also assume to be binary. This ***codeword*** will be changed by noise into a string $\boldsymbol{y}$. The receiver gets $\boldsymbol{y}$ and applies to it a ***decoding function*** ϕ^*.

Definition 4.27 *The pair of functions $\phi_* : \{0,1\}^m \to \{0,1\}^n$ and $\phi^* : \{0,1\}^n \to \{0,1\}^m$ is called a* ***code*** *if $\phi^*(\phi_*(\boldsymbol{x})) = \boldsymbol{x}$ holds for all $\boldsymbol{x} \in \{0,1\}^m$. The strings $\boldsymbol{x} \in \{0,1\}^m$ are called* ***messages****, words of the form $\boldsymbol{y} = \phi_*(\boldsymbol{x}) \in \{0,1\}^n$ are called* ***codewords.*** *(Sometimes the set of all codewords by itself is also called a code.) For every message $\boldsymbol{x}$, the set of words $C_{\boldsymbol{x}} = \{\, \boldsymbol{y} : \phi^*(\boldsymbol{y}) = \boldsymbol{x} \,\}$ is called the* ***decoding set*** *of $\boldsymbol{x}$. (Of course, different decoding sets are disjoint.) The number*

$$R = m/n$$

is called the ***rate*** *of the code.*

We say that our code that ***corrects t errors*** *if for all possible messages $\boldsymbol{x} \in \{0,1\}^m$, if the received word $\boldsymbol{y} \in \{0,1\}^n$ differs from the codeword $\phi_*(\boldsymbol{x})$ in at most t positions, then $\phi^*(\boldsymbol{y}) = \boldsymbol{x}$.*

If the rate is R then the n-bit codewords carry Rn bits of useful information. In terms of decoding sets, a code corrects t errors if each decoding set $C_{\boldsymbol{x}}$ contains all words that differ from $\phi_*(\boldsymbol{x})$ in at most t symbols (the set of these words is a kind of "ball" of radius t).

The Hamming code corrects a single error, and its rate is close to 1. One of the important questions connected with error-correcting codes is how much do we have to lower the rate in order to correct more errors.

Having a notion of codes, we can formulate the main result of this section about information storage.

Theorem 4.28 (Network information storage). *There are constants $\epsilon, c_1, c_2, R > 0$ with the following property. For all sufficiently large m, there is a code (ϕ_*, ϕ^*) with message length m and codeword length $n \le m/R$, and a Boolean clocked circuit $\mathcal{N}$ of size $O(n)$ with n inputs and n outputs, such that the following holds. Suppose that at time 0, the memory cells of the circuit contain string $\boldsymbol{Y}_0 = \phi_*(\boldsymbol{x})$. Suppose further that the evolution $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \dots, \boldsymbol{Y}_t$ of the circuit has ϵ-admissible failures. Then we have*

$$\mathbf{P}[\,\phi^*(\boldsymbol{Y}_t) \ne \boldsymbol{x}\,] < t(c_1\epsilon)^{-c_2 n} .$$

This theorem shows that it is possible to store m bits information for time t, in a clocked circuit of size

$$O(\max(\log t, m)) .$$

As long as the storage time t is below the exponential bound e^{cm} for a certain constant c, this circuit size is only a constant times larger than the amount m of information it stores. (In contrast, in (4.26) we needed an extra factor $\log m$ when we used a separate restoring organ for each bit.)

The theorem says nothing about how difficult it is to compute the codeword $\phi_*(\boldsymbol{x})$ at the beginning and how difficult it is to carry out the decoding $\phi^*(\boldsymbol{Y}_t)$ at the end. Moreover, it is desireable to perform these two operations also in a noise-tolerant fashion. We will return to the problem of decoding later.

Linear algebra. Since we will be dealing more with bit matrices, it is convenient to introduce the algebraic structure

$$\mathbb{F}_2 = (\{0, 1\}, +, \cdot) ,$$

which is a two-element ***field.*** Addition and multiplication in $\mathbb{F}_2$ are defined modulo 2 (of course, for multiplication this is no change). It is also convenient to vest the set $\{0, 1\}^n$ of binary strings with the structure $\mathbb{F}_2^n$ of an n-dimensional vector space over the field $\mathbb{F}_2$. Most theorems and algorithms of basic linear algebra apply to arbitrary fields: in particular, one can define the row rank of a matrix as the maximum number of linearly independent rows, and similarly the column rank. Then it is a theorem that the row rank is equal to the colum rank. From now on, in algebraic operations over bits or bit vectors, we will write $+$ in place of $\oplus$ unless this leads to confusion.

To save space, we will frequently write column vectors horizontally: we write

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = (x_1, \dots, x_n)^T ,$$

where $\boldsymbol{A}^T$ denotes the transpose of matrix $\boldsymbol{A}$. We will write

$$\boldsymbol{I}_r$$

for the identity matrix over the vector space $\mathbb{F}_2^r$.

Linear codes. Let us generalize the idea of the Hamming code.

Definition 4.29 *A code (ϕ_*, ϕ^*) with message length m and codeword length n is* ***linear*** *if, when viewing the message and code vectors as vectors over the field $\mathbb{F}_2$, the encoding function can be computed according to the formula*

$$\phi_*(\boldsymbol{x}) = \boldsymbol{G}\boldsymbol{x} ,$$

with an $m \times n$ matrix $\boldsymbol{G}$ called the ***generator matrix*** *of the code. The number m is called the the number of* ***information bits*** *in the code, the number*

$$k = n - m$$

the number of ***error-check bits.***

Example 4.9 The matrix $\boldsymbol{H}$ in (4.27) can be written as $\boldsymbol{H} = (\boldsymbol{K}, \boldsymbol{I}_3)$, with

$$\boldsymbol{K} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} .$$

Then the error check relation can be written as

$$\boldsymbol{y} = \begin{pmatrix} \boldsymbol{I}_4 \\ -\boldsymbol{K} \end{pmatrix} \begin{pmatrix} y_1 \\ \vdots \\ y_4 \, . \end{pmatrix} .$$

This shows that the bits $y_1, \dots, y_4$ can be taken to be the message bits, or "information bits", of the code, making the Hamming code a linear code with the generator matrix $(\boldsymbol{I}_4, -\boldsymbol{K})^T$. (Of course, $-\boldsymbol{K} = \boldsymbol{K}$ over the field $\mathbb{F}_2$.)

The following statement is proved using standard linear algebra, and it generalizes the relation between error check matrix and generator matrix seen in Example 4.9.

Claim 4.30 *Let $k, m > 0$ be given with $n = m + k$.*

(a) *For every $n \times m$ matrix $\boldsymbol{G}$ of rank m over $\mathbb{F}_2$ there is a $k \times n$ matrix $\boldsymbol{H}$ of rank k with the property*

$$\{\, \boldsymbol{G}\boldsymbol{x} : \boldsymbol{x} \in \mathbb{F}_2^m \,\} = \{\, \boldsymbol{y} \in \mathbb{F}_2^n : \boldsymbol{H}\boldsymbol{y} = 0 \,\}. \tag{4.28}$$

(b) *For every $k \times n$ matrix $\boldsymbol{H}$ of rank k over $\mathbb{F}_2$ there is an $n \times m$ matrix $\boldsymbol{G}$ of rank m with property* (4.28).

Definition 4.31 *For a vector $\boldsymbol{x}$, let $|\boldsymbol{x}|$ denote the number of its nonzero elements: we will also call it the **weight** of $\boldsymbol{x}$.*

In what follows it will be convenient to define a code starting from an error-check matrix $\boldsymbol{H}$. If the matrix has rank k then the code has rate

$$R = 1 - k/n.$$

We can fix any subset S of k linearly independent columns, and call the indices $i \in S$ ***error check bits*** and the indices $i \notin S$ the ***information bits***. (In Example 4.9, we chose $S = \{5, 6, 7\}$.) Important operations can performed over a code, however, without fixing any separation into error-check bits and information bits.

4.5.4. Refreshers

Correcting a single error was not too difficult; finding a similar scheme to correct 2 errors is much harder. However, in storing n bits, typically ϵn (much more than 2) of those bits will be corrupted in every step. There are ingenious and quite efficient codes of positive rate (independent of n) correcting even this many errors. When applied to information storage, however, the error-correction mechanism itself must also work in noise, so we are looking for a particularly simple one. It works in our favor, however, that not all errors need to be corrected: it is sufficient to cut down their number, similarly to the restoring organ in reliable Boolean circuits above.

For simplicity, as gates of our circuit we will allow certain Boolean functions with a large, but constant, number of arguments. On the other hand, our Boolean circuit will have just depth 1, similarly to a restoring organ of Section 4.4. The output of each gate is the input of a memory cell (shift register). For simplicity, we identify the gate and the memory cell and call it a ***cell.*** At each clock tick, a cell reads its inputs from other cells, computes a Boolean function on them, and stores the result (till the next clock tick). But now, instead of majority vote among the input values cells, the Boolean function computed by each cell will be slightly more complicated.

Our particular restoring operations will be defined, with the help of a certain $k \times n$ parity check matrix $\boldsymbol{H} = (h_{ij})$. Let $\boldsymbol{x} = (x_1, \dots, x_n)^T$ be a vector of bits. For some $j = 1, \dots, n$, let V_j (from "vertical") be the set of those indices i with $h_{ij} = 1$. For integer $i = 1, \dots, k$, let H_i (from "horizontal") be the set of those indices j with $h_{ij} = 1$. Then the condition $\boldsymbol{H}\boldsymbol{x} = 0$ can also be expressed by saying that for all i, we have $\sum_{j \in H_i} x_j \equiv 0 \pmod 2$. The sets H_i are called the ***parity check sets*** belonging to the matrix $\boldsymbol{H}$. From now on, the indices i will be called ***checks***, and the indices j ***locations.***

Definition 4.32 *A linear code $\boldsymbol{H}$ is a **low-density parity-check code** with bounds $K, N > 0$ if the following conditions are satisfied:*

(a) *For each j we have $|V_j| \le K$;*

(b) *For each i we have $|H_i| \le N$.*

In other words, the weight of each row is at most N and the weight of each column is at most K.

In our constructions, we will keep the bounds K, N constant while the length n of codewords grows. Consider a situation when $\boldsymbol{x}$ is a codeword corrupted by some errors. To check whether bit x_j is incorrect we may check all the sums

$$s_i = \sum_{j \in H_i} x_j$$

for all $i \in V_j$. If all these sums are 0 then we would not suspect x_j to be in error. If only one of these is nonzero, we will know that $\boldsymbol{x}$ has some errors but we may still think that the error is not in bit x_j. But if a significant number of these sums is nonzero then we may suspect that x_j is a culprit and may want to change it. This idea suggests the following definition.

Definition 4.33 *For a low-density parity-check code $\boldsymbol{H}$ with bounds K, N, the **refreshing operation** associated with the code is the following, to be performed simultaneously for all locations j:*

> *Find out whether more than $\lfloor K/2 \rfloor$ of the sums s_i are nonzero among the ones for $i \in V_j$. If this is the case, flip x_j.*

*Let $\boldsymbol{x}^{\boldsymbol{H}}$ denote the vector obtained from $\boldsymbol{x}$ by this operation. For parameters $0 < \vartheta, \gamma < 1$, let us call $\boldsymbol{H}$ a $(\vartheta, \gamma, K, N, k, n)$-**refresher** if for each vector $\boldsymbol{x}$ of length n with weight $|\boldsymbol{x}| \le \vartheta n$ the weight of the resulting vector decreases thus: $|\boldsymbol{x}^{\boldsymbol{H}}| \le \gamma\vartheta n$.*

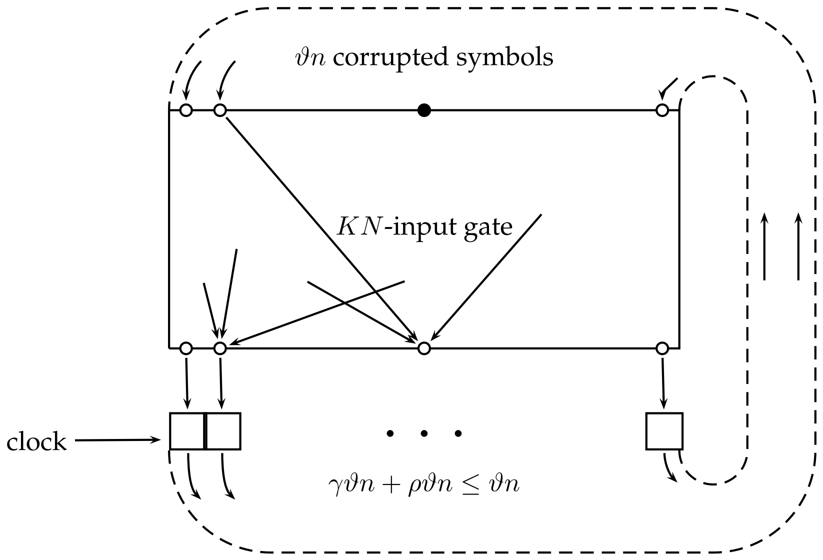
Notice the similarity of refreshers to compressors. The following lemma shows the use of refreshers, and is an example of the advantages of linear codes.

Lemma 4.34 *For an $(\vartheta, \gamma, K, N, k, n)$-refresher $\boldsymbol{H}$, let $\boldsymbol{x}$ be an n-vector and $\boldsymbol{y}$ a codeword of length n with $|\boldsymbol{x} - \boldsymbol{y}| \le \vartheta n$. Then $|\boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{y}| \le \gamma\vartheta n$.*

Proof Since $\boldsymbol{y}$ is a codeword, $\boldsymbol{H}\boldsymbol{y} = 0$, implying $\boldsymbol{H}(\boldsymbol{x} - \boldsymbol{y}) = \boldsymbol{H}\boldsymbol{x}$. Therefore the error correction flips the same bits in $\boldsymbol{x} - \boldsymbol{y}$ as in $\boldsymbol{x}$: $(\boldsymbol{x} - \boldsymbol{y})^{\boldsymbol{H}} - (\boldsymbol{x} - \boldsymbol{y}) = \boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{x}$, giving $\boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{y} = (\boldsymbol{x} - \boldsymbol{y})^{\boldsymbol{H}}$. So, if $|\boldsymbol{x} - \boldsymbol{y}| \le \vartheta n$, then $|\boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{y}| = |(\boldsymbol{x} - \boldsymbol{y})^{\boldsymbol{H}}| \le \gamma\vartheta n$. ■

Theorem 4.35 *There is a parameter $\vartheta > 0$ and integers $K > N > 0$ such that for all sufficiently large codelength n and $k = Nn/K$ there is a $(\vartheta, 1/2, K, N, k, n)$-refresher with at least $n - k = 1 - N/K$ information bits.*

In particular, we can choose $N = 100$, $K = 120$, $\vartheta = 1.31 \cdot 10^{-4}$.



ϑn corrupted symbols
KN-input gate
clock
$\gamma\vartheta n + \rho\vartheta n \le \vartheta n$

Figure 4.13 Using a refresher.

We postpone the proof of this theorem, and apply it first.
ProofProof of Theorem 4.28 Theorem 4.35 provides us with a device for information storage. Indeed, we can implement the operation $\boldsymbol{x} \to \boldsymbol{x}^{\boldsymbol{H}}$ using a single gate g_j of at most KN inputs for each bit j of $\boldsymbol{x}$. Now as long as the inequality $|\boldsymbol{x} - \boldsymbol{y}| \le \vartheta n$ holds for some codeword $\boldsymbol{y}$, the inequality $|\boldsymbol{x}^{\boldsymbol{H}} - \boldsymbol{y}| \le \gamma\vartheta n$ follows with $\gamma = 1/2$. Of course, some gates will fail and introduce new deviations resulting in some $\boldsymbol{x}'$ rather than $\boldsymbol{x}^{\boldsymbol{H}}$. Let $e\epsilon < \vartheta/2$ and $\rho = 1 - \gamma (= 1/2)$. Then just as earlier, the probability that there are more than $\rho\vartheta n$ failures is bounded by the exponentially decreasing expression $(e\epsilon/\rho\vartheta)^{\rho\vartheta n}$. With fewer than $\rho\vartheta n$ new deviations, we will still have $|\boldsymbol{x}' - \boldsymbol{y}| < (\gamma + \rho)\vartheta n < \vartheta n$. The probability that at any time $\le t$ the number of failures is more than $\rho\vartheta n$ is bounded by

$$t(e\epsilon/\rho\vartheta)^{\rho\vartheta n} < t(6\epsilon/\vartheta)^{(1/2)\vartheta n} .$$

■

Example 4.10 Let $\epsilon = 10^{-9}$. Using the sample values in Theorem 4.35 we can take $N = 100$, $K = 120$, so the information rate is $1 - N/K = 1/6$. With the corresponding values of ϑ, and $\gamma = \rho = 1/2$, we have $\rho\vartheta = 6.57 \cdot 10^{-5}$. The probability that there are more than $\rho\vartheta n$ failures is bounded by

$$(e\epsilon/\rho\vartheta)^{\rho\vartheta n} = (10^{-4}e/6.57)^{6.57\cdot 10^{-5}n} \approx e^{-6.63\cdot 10^{-4}n}.$$

This is exponentially decreasing with n, albeit initially very slowly: it is not really small until $n = 10^4$. Still, for $n = 10^6$, it gives $e^{-663} \approx 1.16 \cdot 10^{-288}$.

Decoding? In order to use a refresher for information storage, first we need to enter the encoded information into it, and at the end, we need to decode the information from it. How can this be done in a noisy environment? We have nothing particularly smart to say here about encoding besides the reference to the general reliable computation scheme discussed earlier. On the other hand, it turns out that if ϵ is sufficiently small then *decoding can be avoided* altogether.

Recall that in our codes, it is possible to designate certain symbols as information symbols. So, in principle it is sufficient to read out these symbols. The question is only how likely it is that any one of these symbols will be corrupted. The following theorem upperbounds the probability for any symbol to be corrupted, at any time.

Theorem 4.36 *For parameters $\vartheta, \gamma > 0$, integers $K > N > 0$, codelength n, with $k = Nn/K$, consider a $(\vartheta, 1/2, K, N, k, n)$-refresher. Build a Boolean clocked circuit $\mathcal{N}$ of size $O(n)$ with n inputs and n outputs based on this refresher, just as in the proof of Theorem 4.28. Suppose that at time 0, the memory cells of the circuit contain string $\boldsymbol{Y}_0 = \phi_*(\boldsymbol{x})$. Suppose further that the evolution $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \dots, \boldsymbol{Y}_t$ of the circuit has ϵ-admissible failures. Let $\boldsymbol{Y}_t = (Y_t(1), \dots, Y_t(n))$ be the bits stored at time t. Then $\epsilon < (2.1KN)^{-10}$ implies*

$$\mathbf{P}[\, Y_t(j) \neq Y_0(j)\,] \leq c\epsilon + t(6\epsilon/\vartheta)^{(1/2)\vartheta n}$$

for some c depending on N, K.

Remark 4.37 *What we are bounding is only the probability of a corrupt symbol in the particular position j. Some of the symbols will certainly be corrupt, but any one symbol one points to will be corrupt only with probability $\leq c\epsilon$.*

The upper bound on ϵ required in the condition of the theorem is very severe, underscoring the theoretical character of this result.

Proof As usual, it is sufficient to assume $\boldsymbol{Y}_0 = 0$. Let $D_t = \{\, j : Y_t(j) = 1\,\}$, and let E_t be the set of circuit elements j which fail at time t. Let us define the following sequence of integers:

$$b_0 = 1, \quad b_{u+1} = \lceil (4/3)b_u \rceil, \quad c_u = \lceil (1/3)b_u \rceil \,.$$

It is easy to see by induction

$$b_0 + \dots + b_{u-1} \leq 3b_u \leq 9c_u \,. \tag{4.29}$$

The first members of the sequence b_u are 1,2,3,4,6,8,11,15,18,24,32, and for c_u they are 1,1,1,2,2,3,4,5,6,8,11.

Lemma 4.38 *Suppose that $Y_t(j_0) \neq 0$. Then either there is a time $t' < t$ at which $\geq (1/2)\vartheta n$ circuit elements failed, or there is a sequence of sets $B_u \subseteq D_{t-u}$ for $0 \leq u < v$ and $C \subseteq E_{t-v}$ with the following properties.*

(a) *For $u > 0$, every element of B_u shares some error-check with some element of B_{u-1}. Also every element of C shares some error-check with some element of B_{v-1}.*

(b) *We have $|E_{t-u} \cap B_u| < |B_u|/3$ for $u < v$, on the other hand $C \subseteq E_{t-v}$.*

(c) *We have $B_0 = \{j_0\}$, $|B_u| = b_u$, for all $u < v$, and $|C| = c_v$.*

Proof We will define the sequence B_u recursively, and will say when to stop. If $j_0 \in E_t$ then we set $v = 0$, $C = \{0\}$, and stop. Suppose that B_u is already defined. Let us define B_{u+1} (or C if $v = u + 1$). Let B'_{u+1} be the set of those j which share some error-check with an element of B_u, and let $B''_{u+1} = B'_{u+1} \cap D_{t-u-1}$. The refresher property implies that either $|B''_{u+1}| > \vartheta n$ or

$$|B_u \setminus E_{t-u}| \le (1/2)|B''_{u+1}| .$$

In the former case, there must have been some time $t' < t - u$ with $|E_{t'}| > (1/2)\vartheta n$, otherwise D_{t-u-1} could never become larger than ϑn. In the latter case, the property $|E_{t-u} \cap B_u| < (1/3)|B_u|$ implies

$$\begin{aligned}(2/3)|B_u| &< |B_u \setminus E_{t-u}| \le (1/2)|B''_{u+1}| ,\\ (4/3)b_u &< |B''_{u+1}| .\end{aligned}$$

Now if $|E_{t-u-1} \cap B''_{u+1}| < (1/3)|B''_{u+1}|$ then let B_{u+1} be any subset of B''_{u+1} with size b_{u+1} (there is one), else let $v = u + 1$ and $C \subseteq E_{t-u-1} \cap B''_{u+1}$ a set of size c_v (there is one). This construction has the required properties. ■

For a given B_u, the number of different choices for B_{u+1} is bounded by

$$\binom{|B'_{u+1}|}{b_{u+1}} \le \binom{KNb_u}{b_{u+1}} \le \left(\frac{eKNb_u}{b_{u+1}}\right)^{b_{u+1}} \le ((3/4)eKN)^{b_{u+1}} \le (2.1KN)^{b_{u+1}} ,$$

where we used (4.9). Similarly, the number of different choices for C is bounded by

$$\binom{KNb_{v-1}}{c_v} \le \mu^{c_v} \text{ with } \mu = 2.1KN .$$

It follows that the number of choices for the whole sequence $B_1, \dots, B_{v-1}, C$ is bounded by

$$\mu^{b_1+\dots+b_{v-1}+c_v} .$$

On the other hand, the probability for a fixed C to have $C \subseteq E_v$ is $\le \epsilon^{c_v}$. This way, we can bound the probability that the sequence ends exactly at v by

$$p_v \le \epsilon^{c_v}\mu^{b_1+\dots+b_{v-1}+c_v} \le \epsilon^{c_v}\mu^{10c_v} ,$$

where we used (4.29). For small v this gives

$$p_0 \le \epsilon, \quad p_1 \le \epsilon\mu, \quad p_2 \le \epsilon\mu^3, \quad p_3 \le \epsilon^2\mu^6, \quad p_4 \le \epsilon^2\mu^{10}, \quad p_5 \le \epsilon^3\mu^{16} .$$

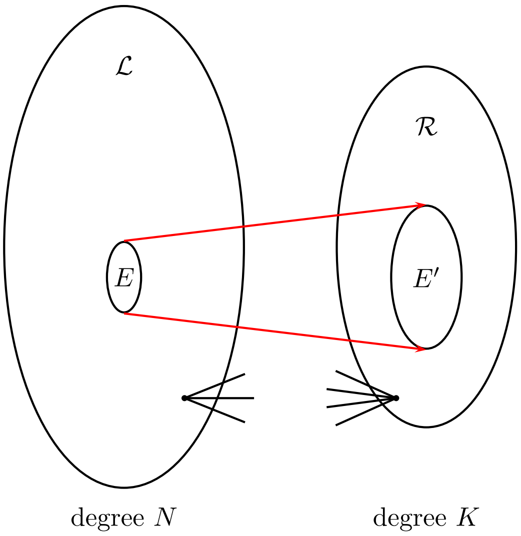


Figure 4.14 A regular expander.

Therefore

$$\sum_{v=0}^{\infty} p_v \le \sum_{v=0}^{5} p_v + \sum_{v=6}^{\infty} (\mu^{10}\epsilon)^{c_v} \le \epsilon(1+\mu+\mu^3) + \epsilon^2(\mu^6+\mu^{10}) + \frac{\epsilon^3\mu^{16}}{1-\epsilon\mu^{10}} \ ,$$

where we used $\epsilon\mu^{10} < 1$ and the property $c_{v+1} > c_v$ for $v \ge 5$. We can bound the last expression by $c\epsilon$ with an appropriate constantb c.

We found that the event $Y_t(j) \ne Y_0(j)$ happens either if there is a time $t' < t$ at which $\ge (1/2)\vartheta n$ circuit elements failed (this has probability bound $t(2e\epsilon/\vartheta)^{(1/2)\vartheta n}$) or an event of probability $\le c\epsilon$ occurs. ■

Expanders. We will construct our refreshers from bipartite multigraphs with a property similar to compressors: expanders.

Definition 4.39 *Here, we will distinguish the two parts of the bipartite (multi) graphs not as inputs and outputs but as* ***left nodes*** *and* ***right nodes.*** *A bipartite multigraph B is (N,K)-**regular** if the points of the left set have degree N and the points in the right set have degree K. Consider such a graph, with the left set having n nodes (then the right set has nN/K nodes). For a subset E of the left set of B, let* $\mathrm{Nb}(E)$ *consist of the points connected by some edge to some element of E. We say that the graph B* ***expands*** *E by a factor λ if we have $|\mathrm{Nb}(E)| \ge \lambda|E|$. For $\alpha, \lambda > 0$, our graph B is an (N,K,α,λ,n)-expander if B expands every subset E of size $\le \alpha n$ of the left set by a factor λ.*

Definition 4.40 *Given an (N, K)-regular bipartite multigraph B, with left set $\{u_1, \dots, u_n\}$ and right set $\{v_1, \dots, v_k\}$, we assign to it a parity-check code $\boldsymbol{H}(B)$ as follows: $h_{ij} = 1$ if v_i is connected to u_j, and 0 otherwise.*

Now for every possible error set E, the set $\mathrm{Nb}(E)$ describes the set of parity checks that the elements of E participate in. Under some conditions, the lower bound on the size of $\mathrm{Nb}(E)$ guarantees that a sufficient number of errors will be corrected.

Theorem 4.41 *Let B be an $(N, K, \alpha, (7/8)N, n)$-expander with integer αn. Let $k = Nn/K$. Then $\boldsymbol{H}(B)$ is a $((3/4)\alpha, 1/2, K, N, k, n)$-refresher.*

Proof More generally, for any $\epsilon > 0$, let B be an $(N, K, \alpha, (3/4 + \epsilon)N, n)$-expander with integer αn. We will prove that $\boldsymbol{H}(B)$ is a $(\alpha(1 + 4\epsilon)/2, (1 - 4\epsilon), K, N, k, n)$-refresher. For an n-dimensional bit vector $\boldsymbol{x}$ with $A = \{\, j : x_j = 1 \,\}$, $a = |A| = |\boldsymbol{x}|$, assume

$$a \le n\alpha(1 + 4\epsilon)/2 \; . \tag{4.30}$$

Our goal is to show $|\boldsymbol{x}^{\boldsymbol{H}}| \le a(1 - 4\epsilon)$: in other words, that in the corrected vector the number of errors decreases at least by a factor of $(1 - 4\epsilon)$.

Let F be the set of bits in A that the error correction operation fails to flip, with $f = |F|$, and G the set of of bits that were 0 but the operation turns them to 1, with $g = |G|$. Our goal is to bound $|F \cup G| = f + g$. The key observation is that each element of G shares at least half of its neighbors with elements of A, and similarly, each element of F shares at least half of its neighbors with other elements of A. Therefore both F and G contribute relatively weakly to the expansion of $A \cup G$. Since this expansion is assumed strong, the size of $|F \cup G|$ must be limited.

Let

$$\delta = |\mathrm{Nb}(A)|/(Na) \; .$$

By expansion, $\delta \ge 3/4 + \epsilon$.

First we show $|A \cup G| \le \alpha n$. Assume namely that, on the contrary, $|A \cup G| > \alpha n$, and let G' be a subset of G such that $|A \cup G'| = \alpha n =: p$ (an integer, according to the assumptions of the theorem). By expansion,

$$(3/4 + \epsilon)Np \le \mathrm{Nb}(A \cup G') \; .$$

Each bit in G has at most $N/2$ neighbors that are not neighbors of A; so,

$$|\mathrm{Nb}(A \cup G')| \le \delta Na + N(p - a)/2 \; .$$

Combining these:

$$\begin{aligned} \delta a + (p - a)/2 &\ge (3/4 + \epsilon)p, \\ a &\ge p(1 + 4\epsilon)/(4\delta - 2) \ge \alpha n(1 + 4\epsilon)/2, \end{aligned}$$

since $\delta \le 1$. This contradiction with (4.30) shows $|A \cup G| \le \alpha n$.

Now $|A \cup G| \le \alpha n$ implies (recalling that each element of G contributes at most $N/2$ new neighbors):

$$\begin{aligned}(3/4+\epsilon)N(a+g) &\le |\mathrm{Nb}(A\cup G)| \le \delta N a + (N/2)g,\\ (3/4+\epsilon)(a+g) &\le \delta a + g/2\ ,\\ (3/4+\epsilon)a + (1/4+\epsilon)g &\le \delta a\ . \end{aligned} \tag{4.31}$$

Each $j \in F$ must share at least half of its neighbors with others in A. Therefore j contributes at most $N/2$ neighbors on its own; the contribution of the other $N/2$ must be divided by 2, so the the total contribution of j to the neighbors of A is at most $(3/4)N$:

$$\begin{aligned}\delta N a = \mathrm{Nb}(A) &\le N(a-f) + (3/4)Nf = N(a-f/4)\ ,\\ \delta a &\le a - f/4\ .\end{aligned}$$

Combining with (4.31):

$$\begin{aligned}(3/4+\epsilon)a + (1/4+\epsilon)g &\le a - f/4\ ,\\ (1-4\epsilon)a &\ge f + (1+4\epsilon)g \ge f+g\ .\end{aligned}$$

■

Random expanders. Are there expanders good enough for Theorem 4.41? The maximum expansion factor is the degree N and we require a factor of $(7/8)N$. It turns out that random choice works here, too, similarly to the one used in the “construction” of compressors.

The choice has to be done in a way that the result is an (N, K)-regular bipartite multigraph of left size n. We will start with Nn left nodes $u_1, \dots, u_{Nn}$ and Nn right nodes $v_1, \dots, v_{Nn}$. Now we choose a random ***matching***, that is a set of Nn edges with the property that every left node is connected by an edge to exactly one right node. Let us call the resulting graph M. We obtain B now as follows: we collapse each group of N left nodes into a single node: $u_1, \dots, u_N$ into one node, $u_{N+1}, \dots, u_{2N}$ into another node, and so on. Similarly, we collapse each group of K right nodes into a single node: $v_1, \dots, v_K$ into one node, $v_{K+1}, \dots, v_{2K}$ into another node, and so on. The edges between any pair of nodes in B are inherited from the ancestors of these nodes in M. This results in a graph B with n left nodes of degree N and nN/K right nodes of degree K. The process may give multiple edges between nodes of B, this is why B is a multigraph. Two nodes of M will be called ***cluster neighbors*** if they are collapsed to the same node of B.

Theorem 4.42 *Suppose*

$$0 < \alpha \le e^{\frac{-1}{N/8-1}} \cdot (22K)^{\frac{-1}{1-8/N}}\ .$$

Then the above random choice gives an $(N, K, \alpha, (7/8)N, n)$-expander with positive probability.

Example 4.11 If $N = 48$, $K = 60$ then the inequality in the condition of the theorem becomes

$$\alpha \le 1/6785 .$$

Proof Let E be a set of size αn in the left set of B. We will estimate the probability that E has too few neighbors. In the above choice of the graph B we might as well start with assigning edges to the nodes of E, in some fixed order of the $N|E|$ nodes of the preimage of E in M. There are $N|E|$ edges to assign. Let us call a node of the right set of M ***occupied*** if it has a cluster neighbor already reached by an earlier edge. Let X_i be a random variable that is 1 if the ith edge goes to an occupied node and 0 otherwise. There are

$$Nn - i + 1 \ge Nn - N\alpha n = Nn(1-\alpha)$$

choices for the ith edge and at most $KN|E|$ of these are occupied. Therefore

$$\mathbf{P}[\, X_i = 1 \mid X_1, \dots, X_{i-1} \,] \le \frac{KN|E|}{Nn(1-\alpha)} = \frac{K\alpha}{1-\alpha} =: p .$$

Using the large deviations theorem in the generalization given in Exercise 4.1-3, we have, for $f > 0$:

$$\mathbf{P}[\sum_{i=1}^{N\alpha n} X_i \ge fN\alpha n \,] \le e^{-N\alpha n D(f,p)} \le \left(\frac{ep}{f}\right)^{fN\alpha n} .$$

Now, the number of different neighbors of E is $N\alpha n - \sum_i X_i$, hence

$$\mathbf{P}[\, N(E) \le N\alpha n(1-f) \,] \le \left(\frac{ep}{f}\right)^{fN\alpha n} = \left(\frac{eK\alpha}{f(1-\alpha)}\right)^{fN\alpha n} .$$

Let us now multiply this with the number

$$\sum_{i \le \alpha n} \binom{n}{\alpha n} \le (e/\alpha)^{\alpha n}$$

of sets E of size $\le \alpha n$:

$$\left(\frac{e}{\alpha}\right)^{\alpha n} \left(\frac{eK\alpha}{f(1-\alpha)}\right)^{fN\alpha n} = \left(\alpha^{fN-1} e \left(\frac{eK}{f(1-\alpha)}\right)^{fN}\right)^{\alpha n} \le \left(\alpha^{fN-1} e \left(\frac{eK}{0.99f}\right)^{fN}\right)^{\alpha n} ,$$

where in the last step we assumed $\alpha \le 0.01$. This is < 1 if

$$\alpha \le e^{\frac{-1}{fN-1}} \left(\frac{eK}{0.99f}\right)^{\frac{-1}{1-1/(fN)}} .$$

Substituting $f = 1/8$ gives the formula of the theorem. ■

ProofProof of Theorem 4.35 Theorem 4.41 shows how to get a refresher from an expander, and Theorem 4.42 shows the existence of expanders for certain parameters. Example 4.11 shows that the parameters can be chosen as needed for the refreshers. ■

Exercises

4.5-1 Prove Proposition 4.30.

4.5-2 Apply the ideas of the proof of Theorem 4.36 to the proof of Theorem 4.17, showing that the "coda" circuit is not needed: each wire of the output cable carries the correct value with high probability.

Problems

4-1 Critical value

Consider a circuit $\mathcal{M}_k$ like in Exercise 4.2-5, assuming that each gate fails with probability $\le \epsilon$ independently of all the others and of the input. Assume that the input vector is all 0, and let $p_k(\epsilon)$ be the probability that the circuit outputs a 1. Show that there is a value $\epsilon_0 < 1/2$ with the property that for all $\epsilon < \epsilon_0$ we have $\lim_{k\to\infty} p_k(\epsilon) = 0$, and for $\epsilon_0 < \epsilon \le 1/2$, we have have $\lim_{k\to\infty} p_k(\epsilon) = 1/2$. Estimate also the speed of convergence in both cases.

4-2 Regular compressor

We defined a compressor as a d-halfregular bipartite multigraph. Let us call a compressor ***regular*** if it is a d-regular multigraph (the input nodes also have degree d). Prove a theorem similar to Theorem 4.21: for each $\gamma < 1$ there is an integer $d > 1$ and an $\alpha > 0$ such that for all integer $k > 0$ there is a regular $(\d,\alpha,\gamma,k)$-compressor. *Hint.* Choose a random d-regular bipartite multigraph by the following process: (1. Replace each vertex by a group of d vertices. 2. Choose a random complete matching betwen the new input and output vertices. 3. Merge each group of d vertices into one vertex again.) Prove that the probability, over this choice, that a d-regular multigraph is a not a compressor is small. For this, express the probability with the help of factorials and estimate the factorials using Stirling's formula.

4-3 Two-way expander

Recall the definition of expanders. Call a $(d, \alpha, \lg, k)$-expander ***regular*** if it is a d-regular multigraph (the input nodes also have degree d). We will call this multigraph a ***two-way expander*** if it is an expander in both directions: from A to B and from B to A. Prove a theorem similar to the one in Problem 4-2: for all $\lg < d$ there is an $\alpha > 0$ such that for all integers $k > 0$ there is a two-way regular $(d, \alpha, \lg, k)$-expander.

4-4 Restoring organ from 3-way voting

The proof of Theorem 4.21 did not guarantee a $(\d,\alpha,\gamma,k)$-compressor with any $\gamma < 1/2$, $\d < 7$. If we only want to use 3-way majority gates, consider the following construction. First create a 3-halfregular bipartite graph G with inputs $u_1, \dots, u_k$ and outputs $v_1, \dots, v_{3k}$, with a 3-input majority gate in each v_i. Then create new nodes $w_1, \dots, w_k$, with a 3-input majority gate in each w_j. The gate of w_1 computes the majority of v_1, v_2, v_3, the gate of w_2 computes the majority of v_4, v_5, v_6, and so on. Calculate whether a random choice of the graph G will turn the circuit with inputs $(u_1, \dots, u_k)$ and outputs $(w_1, \dots, w_k)$ into a restoring organ. Then consider three stages instead of two, where G has $9k$ outputs and see what is gained.

4-5 Restoring organ from NOR gates

The majority gate is not the only gate capable of strengthening the majority. Recall the NOR gate introduced in Exercise 4.2-2, and form $\mathrm{NOR}_2(x_1, x_2, x_3, x_4) = (x_1 \mathrm{NOR} x_2)\mathrm{NOR}(x_3 \mathrm{NOR} x_4)$. Show that a construction similar to Problem 4-4 can be carried out with NOR_2 used in place of 3-way majority gates.

4-6 More randomness, smaller restoring organs

Taking the notation of Exercise 4.4-3, suppose like there, that the random variables F_v are independent of each other, and their distribution does not depend on the Boolean input vector. Apply the idea of Exercise 4.4-5 to the construction of each restoring organ. Namely, construct a different restoring organ for each position: the choice depends on the circuit preceding this position. Show that in this case, our error estimates can be significantly improved. The improvement comes, just as in Exercise 4.4-5, since now we do not have to multiply the error probability by the number of all possible sets of size $\le \alpha k$ of tainted wires. Since we know the distribution of this set, we can average over it.

4-7 Near-sorting with expanders

In this problem, we show that expanders can be used for "near-sorting". Let G be a regular two-way $(d, \alpha, \lg, k)$-expander, whose two parts of size k are A and B. According to a theorem of Kőnig, (the edge-set of) every d-regular bipartite multigraph is the disjoint union of (the edge-sets of) d complete matchings $M_1, \dots, M_d$. To such an expander, we assign a Boolean circuit of depth d as follows. The circuit's nodes are subdivide into levels $i = 0, 1, \dots, d$. On level i we have two disjoint sets A_i, B_i of size k of nodes a_{ij}, b_{ij} $(j = 1, \dots, k)$. The Boolean value on a_{ij}, b_{ij} will be x_{ij} and y_{ij} respectively. Denote the vector of $2k$ values at stage i by $\boldsymbol{z}_i = (x_{i1}, \dots, y_{ik})$. If (p, q) is an edge in the matching M_i, then we put an $\wedge$ gate into a_{ip}, and a $\vee$ gate into b_{iq}:

$$x_{ip} = x_{(i-1)p} \wedge y_{(i-1)q}, \quad y_{iq} = x_{(i-1)p} \vee y_{(i-1)q} .$$

This network is trying to "sort" the 0's to A_i and the 1's to B_i in d stages. More generally, the values in the vectors $\boldsymbol{z}_i$ could be arbitrary numbers. Then if $x \wedge y$ still means $\min(x, y)$ and $x \vee y$ means $\max(x, y)$ then each vector $\boldsymbol{z}_i$ is a permutation of the vector $\boldsymbol{z}_0$. Let $\mathbf{G} = (1 + \lambda)\alpha$. Prove that $\boldsymbol{z}_d$ is $\mathbf{G}$-***sorted*** in the sense that for all m, at least $\mathbf{G}m$ among the m smallest values of $\boldsymbol{z}_d$ is in its left half and at least $\mathbf{G}m$ among the m largest values are in its right half.

4-8 Restoring organ from near-sorters

Develop a new restoring organ using expanders, as follows. First, split each wire of the input cable A, to get two sets A'_0, B'_0. Attach the $\mathbf{G}$-sorter of Problem 4-7, getting outputs A'_d, B'_d. Now split the wires of B'_d into two sets A''_0, B''_0. Attach the $\mathbf{G}$-sorter again, getting outputs A''_d, B''_d. Keep only $B = A''_d$ for the output cable. Show that the Boolean vector circuit leading from A to B can be used as a restoring organ.

Chapter Notes

The large deviation theorem (Theorem 4.1), or theorems similar to it, are sometimes attributed to Chernoff or Bernstein. One of its frequently used variants is given in Exercise 4.1-2.

The problem of reliable computation with unreliable components was addressed by John von Neumann in [14] on the model of logic circuits. A complete proof of the result of that paper (with a different restoring organ) appeare first in the paper [5] of R. L. Dobrushin and S. I. Ortyukov. Our presentation relied on parts of the paper [18] of N. Pippenger.

The lower-bound result of Dobrushin and Ortyukov in the paper [4] (corrected in [16], [19] and [8]), shows that reduncancy of $\log n$ is unavoidable for a general reliable computation whose complexity is n. However, this lower bound only shows the necessity of putting the input into a redundantly encoded form (otherwise critical information may be lost in the first step). As shown in [18], for many important function classes, linear redundancy is achievable.

It seems natural to separate the cost of the initial encoding: it might be possible to perform the rest of the computation with much less redundancy. An important step in this direction has been made by D. Spielman in the paper [23] in (essentially) the clocked-circuit model. Spielman takes a parallel computation with time t running on w elementary components and makes it reliable using only $(\log w)^c$ times more processors and running it $(\log w)^c$ times longer. The failure probability will be $t\exp(-w^{1/4})$. This is small as long as t is not much larger than $\exp(w^{1/4})$. So, the redundancy is bounded by some power of the logarithm of the *space requirement*; the time requirement does not enter explictly. In Boolean circuits no time- and space-complexity is defined separately. The size of the circuit is analogous to the quantity obtained in other models by taking the product of space and time complexity.

Questions more complex than Problem 4-1 have been studied in [17]. The method of Problem 4-2, for generating random d-regular multigraphs is analyzed for example in [2]. It is much harder to generate simple regular graphs (not multigraphs) uniformly. See for example [10].

The result of Exercise 4.2-4 is due to C. Shannon, see [20]. The asymptotically best circuit size for the worst functions was found by Lupanov in [12]. Exercise 4.3-1 is based on [5], and Exercise 4.3-2 is based on [4] (and its corrections).

Problem 4-7 is based on the starting idea of the $\lg n$ depth sorting networks in [1].

For storage in Boolean circuits we partly relied on A. V. Kuznietsov's paper [11] (the main theorem, on the existence of refreshers is from M. Pinsker). Low density parity check codes were introduced by R. G. Gallager in the book [6], and their use in reliable storage was first suggested by M. G. Taylor in the paper [24]. New, constructive versions of these codes were developed by M. Sipser and D. Spielman in the paper [22], with superfast coding and decoding.

Expanders, invented by Pinsker in [15] have been used extensively in theoretical computer science: see for example [13] for some more detail. This book also gives references on the construction of graphs with large eigenvalue-gap. Exercise 4.4-4 and Problem 4-6 are based on [5].

The use of expanders in the role of refreshers was suggested by Pippenger (private communication): our exposition follows Sipser and Spielman in [21]. Random expanders were found for example by Pinsker. The needed expansion rate ($> 3/4$ times the left degree) is larger than what can be implied from the size of the eigenvalue gap. As shown in [15] (see the proof in Theorem 4.42) random expanders have the needed expansion rate. Lately, constructive expanders with nearly maximal expansion rate were announced by Capalbo, Reingold, Vadhan and Wigderson in [3].

Reliable computation is also possible in a model of parallel computation that is much more regular than logic circuits: in cellular automata. We cannot present those results here: see for example the papers [7, 9].

Bibliography

[1] M. Ajtai, J. Komlós, E. Szemerédi. Sorting in $c \log n$ parallel steps. *Combinatorica*, 3(1):1–19, 1983. 214

[2] E. Bender, R. Canfield. The asymptotic number of labeled graphs with given degree sequences. *Combinatorial Theory Series A*, 24:296–307, 1978. 214

[3] M. Capalbo, O. Reingold, S. Vadhan, A. Widgerson. Randomness conductors and constant-degree lossless expanders. In *Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 443–452, 2001. IEEE Computer Society. 215

[4] R. Dobrushin, S. Ortyukov. Lower bound for the redundancy of self-correcting arrangements of unreliable functional elements. *Problems of Information Transmission* (translated from Russian), 13(1):59–65, 1977. 214

[5] R. Dobrushin, S. Ortyukov. Upper bound for the redundancy of self-correcting arrangements of unreliable elements. *Problems of Information Transmission* (translated from Russian), 13(3):201–208, 1977. 214

[6] R. Gallager. *Low-density Parity-check Codes*. The MIT Press, 1963. 214

[7] P. Gács. Reliable cellular automata with self-organization. *Journal of Statistical Physics*, 103(1–2):45–267, 2001. See also www.arXiv.org/abs/math.PR/0003117 and *The Proceedings of the 1997 Symposium on the Theory of Computing*. 215

[8] P. Gács, A. Gál. Lower bounds for the complexity of reliable Boolean circuits with noisy gates. *IEEE Transactions on Information Theory*, 40(2):579–583, 1994. 214

[9] P. Gács, J. Reif. A simple three-dimensional real-time reliable cellular array. *Journal of Computer and System Sciences*, 36(2):125–147, 1988. 215

[10] J. Kim, V. Vu. Generating random regular graphs. In *Proceedings of the Thirty Fifth ACM Symposium on Theory of Computing*, pages 213–222, 2003. 214

[11] A. V. Kuznetsov. Information storage in a memory assembled from unreliablecomponents. *Problems of Information Transmission* (translated from Russian), 9(3):254–264, 1973. 214

[12] O. B. Lupanov. On a method of circuit synthesis. *Izvestia VUZ (Radiofizika)*, pages 120–140, 1958. 214

[13] R. Motwani, P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. 214

[14] J. Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. Shannon and P. J. *McCarthy* Eds. *Automata Studies*. Princeton University Press, 1956, pages 43–98. 214

[15] M. Pinsker. On the complexity of a concentrator. *International Teletraffic Congr.*, 7:318/1–318/4, 1973. 214, 215

[16] N. Pippenger, G. Staomulis, J. N. Tsitsiklis. On a lower bound for the redundancy of reliable networks with noisy gates. *IEEE Transactions on Information Theory*, 37(3):639–643, 1991. 214

[17] N. Pippenger. Analysis of error correction by majority voting. In S. Micali (Ed.) *Randomness in Computation*. JAI Press, 1989, 171–198. 214

[18] N. Pippenger. On networks of noisy gates. In *Proceeding of the 26th IEE FOCS Symposium*, pages 30–38, 1985. 214

[19] R. Reischuk, B. Schmelz, B.. Reliable computation with noisy circuits and decision trees—a general $n \log n$ lower bound. In *Proceedings of the 32-nd IEEE FOCS Symposium*, pages 602–611, 1991. 214

[20] C. Shannon. The synthesis of two-terminal switching circuits. *The Bell Systems Technical Journal*, 28:59–98, 1949. 214

[21] M. Sipser, D. A. Spielman. Expander codes. *IEEE Transactions on Information Theory*, 42(6):1710–1722, 1996. 215

[22] D. Spielman. Linear-time encodable and decodable error-correcting codes. In *Proceedings of the 27th ACM STOC Symposium*, 1995, 387–397 (further *IEEE Transactions on Information Theory* 42(6):1723–1732). 214

[23] D. Spielman. Highly fault-tolerant parallel computation. In *Proceedings of the 37th IEEE Foundations of Computer Science Symposium*, pages 154–163, 1996. 214

[24] M. G. Taylor. Reliable information storage in memories designed from unreliable components. *The Bell Systems Technical Journal*, 47(10):2299–2337, 1968. 214

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, “2-3-4-tree" is indexed as if were “two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *ex* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.
The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.