Chapter 40

Combinatorics in Computer Science

L. Lovász
Department of Computer Science,
Eötvös University, Budapest, H-1088, Hungary, and
Princeton University, Princeton, NJ 08544, USA

D.B. Shmoys and É. Tardos
School of Operations Research and Industrial Engineering,
Cornell University, Ithaca, NY 14853, USA

(December 1990)

It is not surprising that computer science is perhaps the most important field of applications of combinatorial ideas. Modern computers operate in a discrete fashion both in time and space, and much of classical mathematics must be "discretized" before it can be implemented on computers as, for example, in the case of numerical analysis. The connection between combinatorics and computer science might be even stronger than suggested by this observation; each field has profited from the other. Combinatorics was the first field of mathematics where the ideas and concepts of computer science, in particular complexity theory, had a profound impact. This framework for much of combinatorics has been surveyed in Chapter 29. In this chapter, we illustrate what computer science profits from combinatorics: we have collected a number of examples, all of them rather important in computer science, where methods and results from classical discrete mathematics play a crucial role. Since many of these examples rely on concepts from theoretical computer science that have been discussed in Chapter 29, the reader is encouraged to refer to that chapter for background material.

1. Communication complexity

There are many situations where the amount of communication between two processors jointly solving a certain task is the real bottleneck; examples range from communication between the earth and a rocket approaching Jupiter to communication between different parts of a computer chip. We shall see that communication complexity also plays an important role in theoretical studies, and in particular, in the complexity theory of circuits. Other examples of such indirect applications of communication complexity include bounds on the depth of decision trees (Hajnal, Maass and Turán 1988) and pseudorandom number generation (Babai, Nisan and Szegedy 1989). Communication complexity is a much simpler and cleaner model than computational or circuit complexity, and it illustrates notions from complexity theory like non-determinism and randomization in a particularly simple way. Our interest in this field also stems from the many ways in which it relates to combinatorial problems and methods. This section gives just a glimpse of this theory; see Lovász (1990) for a broader survey.

Suppose that there are two players, Alice and Bob, who want to evaluate a function $f(x,y)$ in two variables; for simplicity, we assume that the value of f is a single bit. Alice knows the value of the variable x, Bob knows the value of the variable y, and they can communicate with each other. Local computation is free, but communication is costly. What is the minimum number of bits that they have to communicate?

We can describe the problem by a matrix as follows. Let $a_1, \ldots, a_N$ be the possible inputs of Alice and $b_1, \ldots, b_M$, the possible inputs of Bob. Note that since local computation is free, we need not worry about how these are encoded. Let $c_{ij} = f(a_i, b_j)$. The 0-1 matrix $C = (c_{ij})_{i=1}^{N}{}_{j=1}^{M}$ determines the communication problem. Both players know the matrix C; Alice also knows the index i of a row, Bob knows the index j of a column, and they want to determine the entry c_{ij}.

To solve their task for a particular matrix C, Alice and Bob, before learning their inputs i and j, agree in advance on a **protocol**, which is the communication analogue to the fundamental notion of an algorithm in computational complexity theory. Informally, a **communication protocol** is a set of rules specifying the order and "meaning" of the messages sent. The protocol prescribes each action for Alice and Bob: who is to send the first bit; depending on the input of that processor, what this bit should be; depending on this bit, who is to send the second bit, and depending on the input of that processor and on the first bit sent, what this second bit should be, and so on. The protocol terminates when one processor knows the output bit and the other one knows this about the first one. The **complexity** of a protocol is the number of bits communicated in the worst case.

The **trivial protocol** is that Alice tells her input to Bob. We shall see that sometimes there is no better protocol than this trivial one. This protocol takes $\lceil \log_2 N \rceil$ bits; if $M < N$ then the reverse trivial protocol is clearly better. (For the remainder of this chapter, we shall use log to denote $\log_2$.)

A protocol has the following combinatorial description in terms of the matrix. First, it determines who sends the first bit; say, Alice does. This bit is determined by the input of Alice; in other words, the protocol partitions the rows of the matrix C into two classes, and the first bit of Alice tells Bob which of the two classes contains her row. From now on, the game is limited to the submatrix C_1 formed by the rows in this class. Next, the protocol describes a partition of either the rows or columns of C_1 into two classes, depending on who is to send the second bit, and the second bit itself specifies which of these two classes contains the line (row or column) of the sender. This limits the game to a submatrix C_2, and so it continues.

If the game ends after k bits then the remaining submatrix C_k is the union of an all-1 submatrix and an all-0 submatrix. We shall call this an **almost-homogeneous matrix**. If, for example, Alice knows the answer, then her row in C_k must be all-0 or all-1, and since Bob knows for sure that Alice knows the answer, this must be true for every row of C_k.

We can therefore characterize the communication complexity by the following combinatorial problem: given a 0-1 matrix C, in how many rounds can we partition it into almost-homogeneous submatrices, if in each round we can split each of the current submatrices into two (either horizontally or vertically). We shall denote this number by $\kappa(C)$.

If $\mathrm{rk}(C)$ denotes the rank of C, then the following inequality, observed by Mehlhorn and Schmidt (1982), relates the communication complexity to the rank of the matrix:

1.1 Lemma. *Over any field,* $\log \mathrm{rk}(C) \leq \kappa(C) \leq \mathrm{rk}(C)$. ∎

(The lower bound is tightest if we use the real field, whereas the upper bound might be tightened by considering a finite field.)

In particular, we obtain that if C has full row rank then the trivial protocol is optimal. This corollary applies directly to a number of communication problems, of which we mention three. Suppose that Alice knows a subset X of an n-element set S and Bob knows a subset Y of S. The **equality problem** is to decide if the two subsets are equal; the **disjointness problem** is to decide if the two subsets are disjoint; the **binary inner product problem** is to decide if the intersection of the two sets has odd cardinality. In the first two cases, it is trivial to see that the corresponding $2^n \times 2^n$ matrices have full rank; in the third case, the rank of the matrix is $2^n - 1$. Hence the trivial protocols are optimal. (It is interesting to remark that in the third case, the rank over GF(2), which might seem more natural to use in this problem, gives a very poor bound here: the GF(2) rank of C is only n.)

The lower bound in the lemma is often sharp; on the other hand, no communication problem is known for which $\kappa(C)$ is anywhere near the bound $\mathrm{rk}(C)$. In particular, it is open whether $\kappa(C)$ can be bounded by any polynomial of $\log \mathrm{rk}(C)$.

To point out that the situation is not always trivial, consider again the equality problem. Recall that if N denotes the number of inputs, then the trivial protocol takes $\lceil \log N \rceil$ bits, and this is optimal.

In contrast, Freivalds (1979) obtained the very nice result that if Alice and Bob tolerate errors occurring with probability 2^{-100}, then the situation changes drastically. Consider the following protocol, which can be viewed as an extension of a "parity check". Treat the inputs as two natural numbers x and y, $0 \leq x, y \leq N-1$. Alice selects a random prime $p \leq (\log N)^2$, computes the remainder x' of x modulo p, and then sends the pair (x', p) to Bob. Bob then computes the remainder y' of y modulo p, and compares it with x'. If they are distinct, he concludes that $x \neq y$; if they are the same, he concludes that $x = y$.

If the two numbers x and y are equal then, of course, so are x' and y' and so the protocol reaches the right conclusion. If x and y are different, then it could happen that $x' = y'$ and so the protocol reaches the wrong conclusion. This happens if and only if p is a divisor of $x-y$. Since $|x-y| < N$, it follows that $x-y$ has fewer than $\log N$ different prime divisors. On the other hand, Alice had about $(\log N)^2/2\log\log N$ primes from which to choose, and so the probability that she chose one of the divisors of $x-y$ tends to 0. For N sufficiently large, the error will be smaller than 2^{-100}. Clearly, this protocol uses at most $4\log\log N$ bits.

Randomization need not lead to such dramatic improvements for all problems. We have seen that the binary inner product problem and the disjointness problem behave quite similarly to the equality problem from the point of view of deterministic communication complexity: the corresponding matrices have essentially full rank and hence the trivial protocols are optimal. But, unlike for the equality problem, randomization is of little help here: Chor and Goldreich (1985) proved that the randomized communication complexity of computing the binary inner product problem is $\Omega(n)$. Improving results of Babai, Frankl and Simon (1986), XXX (1989) showed that the randomized communication complexity of the disjointness problem is $\Omega(n)$. The proofs of these facts combine the work of Yao (1983) on randomized communication complexity with rather involved combinatorial considerations.

Just as in computational complexity theory, non-determinism plays a crucial role in communication complexity theory. Non-deterministic protocols were introduced by Lipton and Sedgewick (1981). Perhaps the best way to view a non-deterministic protocol is as a scheme by which a third party, knowing both inputs, can convince Alice and Bob what the value of f is. Again, we want to minimize the number of bits such a **certificate** contains for the worst inputs. For example, if in the disjointness problem the two subsets are not disjoint, announcing a common element convinces both players that the answer is "no". This certificate takes only $\lceil\log n\rceil$ bits, which is much smaller than the number of bits Alice and Bob would need to find the answer, which is n, as we have seen. On the other hand, if the sets are disjoint, then no such simple non-deterministic scheme exists. We shall distinguish between the non-deterministic protocol for the "0" and for the "1". In both cases, there is always the trivial protocol that announces the input of Alice.

To give a formal and combinatorial description of non-deterministic protocols, consider a non-deterministic protocol for "1", a particular certificate p, and those entries of C (i.e., inputs for Alice and Bob) for which this certificate "works". If the proof scheme is correct, these must be all 1's; from the fact that Alice and Bob have to verify the certificate independently, we also see that these 1's must form a submatrix. Thus, a non-deterministic protocol corresponds to a covering of C by all-1 submatrices. Conversely, every such covering gives a non-deterministic protocol: one can simply use the name of the submatrix containing the given entry as a certificate. The number of bits needed is the logarithm of the number of different certificates used: the non-deterministic communication complexity $\kappa_1(C)$ of a matrix C is the least natural number t such that the 1's in C can be covered by at most 2^t all-1 submatrices. One can analogously define $\kappa_0(C)$, the non-deterministic communication complexity of certifying a 0.

Note that the all-1 submatrices in the covering need not be disjoint. Therefore, there is no immediate relation between $\mathrm{rk}(C)$ and $\kappa_1(C)$. It is easy to formulate the non-deterministic communication complexity of a matrix as a set-covering problem: consider the hypergraph whose vertices are the 1's in C and whose edges are the all-1 submatrices; $\kappa_1(C)$ is the logarithm of the minimum number of edges covering all vertices. An immediate lower bound on $\kappa_1(C)$ follows from a simple counting argument: if C has a 1's but each all-1 submatrix of C has at most b entries, then trivially, $\kappa_1(C) \geq \log a - \log b$. We can also consider the natural dual problem: if α is the maximum number of 1's in C such that no two occur in the same all-1 submatrix, then $\kappa_1(C) \geq \log \alpha$.

Returning to the three problems on two sets mentioned above, we see that the 1's in the $2^n \times 2^n$ identity matrix cannot be covered by fewer that 2^n all-1 submatrices; hence the non-deterministic communication complexity of equality is n. Similarly, the non-deterministic communication complexity of set disjointness is n, since $\alpha = 2^n$. For the binary inner product problem, it is easy to see by elementary linear algebra that an all-1 submatrix has at most 2^{n-1} entries, and that exactly $2^{n-1}(2^n - 1)$ entries are 1's. This gives that at least $2^n - 1$ all-1 submatrices are needed to cover the 1's, and so $\kappa_1(C) = n$. For this matrix, a similar argument also shows that $\kappa_0(C) = n$.

We have derived the following lower bound on the communication complexity of a matrix:

$$\max\bigl\{\log \mathrm{rk}(C), \kappa_0(C), \kappa_1(C)\bigr\} \leq \kappa(C).$$

The identity matrix shows that κ_1 might be a very weak bound: $\kappa(C)$ can be exponentially large compared with $\kappa_1(C)$. Interchanging the roles of 1 and 0, we obtain that $\kappa_0(C)$ might also be very far from $\kappa(C)$. No such example is known for $\log \mathrm{rk}(C)$, but it is likely that the situation is similar.

However, it is a surprising fact that the product of any two of these three lower bounds is an upper bound on the communication complexity. The first part of the following theorem is due to Aho, Ullman and Yannakakis (1983), and the other two, to Lovász and Saks (1990):

1.2 Theorem. *For every matrix C,*

(a) $\kappa(C) \leq (\kappa_0(C) + 1)(\kappa_1(C) + 1)$;

(b) $\kappa(C) \leq 1 + \lfloor \log \mathrm{rk}(C) \rfloor (\kappa_0(C) + 2)$;

(c) $\kappa(C) \leq 1 + \lfloor \log(\mathrm{rk}(C) + 1) \rfloor (\kappa_1(C) + 2)$.

We shall sketch a result that is stronger than each of (a), (b) and (c). Let $\rho_1(C)$ denote the size of the largest square submatrix of C such that, after a suitable reordering of the rows and columns, each diagonal entry is 1 but each entry above the diagonal is 0. It is clear that $\rho_1(C) \leq \mathrm{rk}(C)$ and $\rho_1(C) \leq 2^{\kappa_1(C)}$. If we define $\rho_0(C)$ analogously, then $\rho_0(C) \leq \mathrm{rk}(C) + 1$. By using these inequalities, we can obtain all three parts of Theorem 1.2 from the following result.

1.3 Theorem. $\kappa(C) \leq 1 + \lfloor \log \rho_1(C) \rfloor \bigl(\kappa_0(C) + 1\bigr)$.

Proof. We use induction on $\rho_1(C)$. If $\rho_1(C) = 1$ then trivially $\kappa(C) \leq 1$. Assume that $\rho_1(C) > 1$ and let $k = \kappa_0(C)$; then the 0 elements of C can be covered by all-0 submatrices $C_1, \ldots, C_l$ where $l \leq 2^k$. Let A_i denote the submatrix formed by those rows of C that meet C_i, and let B_i denote the analogous submatrix formed from columns of C. Observe that

$$\rho_1(A_i) + \rho_1(B_i) \leq \rho_1(C), \ i = 1, \ldots, l; \tag{1.4}$$

this will play a crucial role in the following protocol.

First, Alice looks at her row to see if it intersects any submatrix C_i with $\rho_1(A_i) \leq \rho_1(C)/2$. If so, she sends a "1" and the name i of such a submatrix. They have thereby reduced the problem to the matrix A_i. If not, she sends a "0".

When Bob receives this "0", he looks at his column to see if it intersects any submatrix C_i with $\rho_1(B_i) \leq \rho_1(C)/2$. If so, he sends a "1" and the name i of such a submatrix. They have thereby reduced the problem to the matrix B_i. If not, he sends a "0".

If both Alice and Bob failed to find an appropriate submatrix, then by (1.4), the intersection of their lines cannot belong to any C_i, and so it must be a "1". They have found the answer. ∎

Theorem 1.2(a) has an interesting interpretation. Define a **communication problem** as a class $\mathcal{H}$ of 0-1 matrices; for simplicity, assume that they are square matrices. The communication complexity of any $N \times N$ matrix is at most $\log N$. We say that $\mathcal{H}$ is in $\mathcal{P}_{\text{comm}}$ if it can be solved substantially better: if there exists a constant $c > 0$ such that $\kappa(C) \leq (\log\log N)^c$ for each $N \times N$ matrix $C \in \mathcal{H}$. Similarly, we say that $\mathcal{H}$ is in $\mathcal{NP}_{\text{comm}}$ if there exists a constant $c > 0$ such that for each $C \in \mathcal{H}$, $\kappa_1(C) \leq (\log\log N)^c$, and define co-$\mathcal{NP}_{\text{comm}}$ analogously based on $\kappa_0(C)$. Just as for the analogous computational complexity classes, we have the trivial containment

$$\mathcal{P}_{\text{comm}} \subseteq \mathcal{NP}_{\text{comm}} \cap \text{co–}\mathcal{NP}_{\text{comm}}.$$

However, for the communication complexity classes we also have the following, rather interesting facts:

$$\mathcal{P}_{\text{comm}} \neq \mathcal{NP}_{\text{comm}},$$

$$\mathcal{NP}_{\text{comm}} \neq \text{co–}\mathcal{NP}_{\text{comm}},$$

which follow from the equality problem, but

$$\mathcal{P}_{\text{comm}} = \mathcal{NP}_{\text{comm}} \cap \text{co–}\mathcal{NP}_{\text{comm}},$$

by Theorem 1.2(a). This idea was developed by Babai, Frankl and Simon (1986), who defined and studied communication analogues of many other well-known complexity classes such as $\#\mathcal{P}$, $\mathcal{PSPACE}$ and $\mathcal{BPP}$.

Our previous examples were trivial from the point of view of communication: the trivial protocols are optimal. This is quite atypical. Let us define a **round** in the protocol as a maximal period during which one player sends bits to the other. The trivial protocol consists of one round. Let $\kappa^k(C)$ denote the minimum number of bits needed by a communication protocol with at most k rounds. Halstenberg and Reischuk (1988) proved that for every $k \geq 1$, there exist arbitrarily large matrices C such that $\kappa^k(C)$ is exponentially larger than $\kappa^{k+1}(C)$.

We consider two combinatorial examples to illustrate that protocols can be significantly more efficient than the trivial ones. Yannakakis (1988) considered the following problem: Alice and Bob are given a graph G on n nodes, Alice is given a stable set A and

Bob is given a clique B; they must decide whether these sets intersect. The corresponding matrix C has rank n but size exponential in n, and so the trivial protocol takes $\Omega(n)$ bits. (Recall that we always focus on the worst-case complexity.) On the other hand, the non-deterministic communication complexity of non-disjointness is $\log n$, since the name of a common node of A and B is a certificate; by Theorem 1.2, $\kappa(C) \leq (2+\log n)^2$. (The number of rounds in the protocol is $O(\log n)$.) It is not known whether the deterministic complexity, or even the non-deterministic complexity of disjointness, is smaller, such as $O(\log n)$. It is interesting to remark that the latter question is equivalent to the following purely graph-theoretic question: is there a constant $c > 0$ so that in each graph G on n nodes, there exist n^c cuts such that each pair of disjoint sets U, V, where U is a stable set and V is a clique, is separated by one of these cuts.

In the **subtree disjointness problem**, there is a tree T known to both players. Alice gets a subtree T_A and Bob gets a subtree T_B, and their task is to decide whether T_A and T_B are node-disjoint. It can be shown that the corresponding matrix C has rank $2n$ but has exponential size. The non-deterministic complexity of non-disjointness is $\log n$, and hence by Theorem 1.2, $\kappa(C) \leq (2+\log n)^2$. In this case, one can do better by using the following simple protocol: Alice sends any node x of her tree to Bob. If y is the node in Bob's tree that is closest to x, Bob responds by sending y to Alice. Then Alice checks if $y \in T_A$: if so, then clearly the subtrees are not disjoint; if not, the subtrees are disjoint. This protocol uses $2\log n$ bits. Lovász and Saks (1988) showed that it can be modified to use only $\log n + \log\log n$ bits.

An interesting and rather general class of communication problems, for which good bounds on the complexity can be obtained by non-trivial combinatorial means, was formulated by Hajnal, Maass and Turán (1988). Let $(\mathcal{L}, \wedge, \vee)$ be a finite lattice. Assume that Alice is given an element $a \in \mathcal{L}$ and Bob is given an element $b \in \mathcal{L}$, and their task is to decide whether $a \wedge b = 0$, the minimal element of the lattice.

This problem generalizes both the disjointness problem (where $\mathcal{L}$ is a Boolean algebra), and the subtree disjointness problem (where $\mathcal{L}$ is the lattice of subtrees of a tree). A third special case worth mentioning is the following **spanning subgraph problem**: Alice is given a graph G_A and Bob is given a graph G_B on the same set of nodes V; they wish to decide whether $G_A \cup G_B$ is connected. This case relies on the lattice of partitions of V, but "upside down" so that the indiscrete partition is 0. Alice can compute the partition a of V into the connected components of G_A, Bob can compute the partition b of V into the connected components of G_B, and then they decide whether $a \wedge b = 0$.

For a given lattice $\mathcal{L}$, let C be the matrix associated with the corresponding problem: its rows and columns are indexed by the elements of $\mathcal{L}$, and $c_{ij} = 1$ if and only if $i \wedge j = 0$. To find the rank of this matrix, we give the following factorization of it, using the Möbius function μ of $\mathcal{L}$ (see Chapter 21 for the definition and some basic properties). Let $Z = (z_{ij})$ be the zeta-matrix of the lattice, i.e., let $z_{ij} = 1$ if and only if $i \leq j$ $(i, j \in \mathcal{L})$. Let $D = (d_{ij})$ denote the diagonal matrix defined by $d_{ii} = \mu(0, i)$. Then it is easy to verify the following identity, found by Wilf (1968) and Lindström (1969) (see Chapter 31):

$$C = Z^T D Z. \tag{1.5}$$

Since Z is trivially non-singular, this implies that

$$\mathrm{rk}(C) = \mathrm{rk}(D) = |\{i : \ \mu(0,i) \neq 0\}|. \tag{1.6}$$

This gives a lower bound on the communication complexity of our problem, but how good is this bound? One case when this bound is tight occurs when $\mu(0,i) \neq 0$ for all i. We obtain a lower bound of $\lceil \log |\mathcal{L}| \rceil$, which is also the upper bound achieved by the trivial protocol; that is, the trivial protocol is optimal. By Corollary 3.10 of Chapter 21, this case occurs when $\mathcal{L}$ is a geometric lattice or a geometric lattice "upside down". In particular, we see that for the spanning subgraph problem, the trivial protocol is optimal.

It turns out that the lower bound given by $\log \mathrm{rk}(C)$ is not too far from the truth for any lattice.

1.7 Theorem. *For every lattice $\mathcal{L}$,*

$$\log \mathrm{rk}(C) \leq \kappa(C) \leq (1 + \log \mathrm{rk}(C))^2.$$

Proof (upper bound). Observe that a non-deterministic certificate of non-disjointness of two elements $a, b \in \mathcal{L}$ can be provided by exhibiting an atom of the lattice below both a and b. Hence the logarithm of the number of atoms is an upper bound on $\kappa_0(C)$. Since for every atom i, $\mu(0,i) = -1$, it follows from the identity (1.6) that the number of atoms is at most $\mathrm{rk}(C)$. Hence, the upper bound follows directly from Theorem 1.2. ∎

2. Circuit complexity

One promising approach to proving lower bounds on the computational complexity of a problem focuses on the Boolean circuit model of computation, and recent results in this area are possibly the deepest applications of combinatorial methods to computer science thus far. The best way to view a circuit is not as an abstract electronic device; instead, view it as the bit-operational skeleton of any computational procedure ˚ MR7 this way, it is not hard to see that this model is equivalent to other models such as the Turing machine or RAM, and that the number of functional elements, or gates, in a circuit is equivalent to the time taken by an algorithm in those models. (More precisely, a RAM algorithm, for example, is equivalent to a family of Boolean circuits, one for each input length.) As a result, the extremely difficult task of proving lower bounds on the computational complexity of a given problem can be posed in a way much more suited to combinatorial methods. Many people believe that this is the direction of research that may eventually lead to the solution of famous problems, such as $\mathcal{P}$ vs. $\mathcal{NP}$. Unfortunately, the handful of results obtained at this point are rather difficult, and yet quite far from this objective.

Let us recall some definitions from Chapter 29. A **Boolean circuit** is an acyclic directed graph; nodes of indegree 0 are called **input gates**, and are labelled with the input variables $x_1, \ldots, x_n$; nodes with outdegree 0 are called **output gates**; every node with indegree $r > 0$ is called a **functional gate**, and is labelled with a Boolean function in r variables, corresponding to the predecessors of the node. For our purposes, it suffices to allow only the logical negation, conjunction, and disjunction as functions. The number of gates in a circuit is called its **size**; the **circuit complexity** of a problem is the size of the smallest circuit that computes this Boolean function. The outdegree and indegree of a node are referred to as its **fan-out** and **fan-in**, respectively.

Another important parameter of a circuit is its **depth**, the maximum length of a path from an input gate to an output gate. A circuit can also be viewed as parallel algorithm, and then the depth corresponds to the (parallel) time that the algorithm takes. Note that every Boolean function can be computed by a Boolean circuit of depth 2: this is easily derived from its conjunctive normal form. Of course, the number of gates, which is essentially the number of terms in this normal form, is typically exponential.

A circuit is **monotone** if only conjunctions and disjunctions are allowed as functional gates. Note that every monotone increasing Boolean function can be computed by a monotone Boolean circuit.

The predominant approach to proving circuit complexity lower bounds is to restrict the class of allowed circuits. Two kinds of restrictions have proved sufficiently strong, and yet reasonably interesting, to allow the derivation of superpolynomial lower bounds on the number of gates: monotonicity and bounding the depth. Two main methods seem to emerge: the random restriction method and the approximation method. Both methods have applications for both kinds of restricted problems.

The first superpolynomial lower bound in a restricted model of computation concerned constant-depth circuits. Note that for this class of circuits to make sense, we must allow that the gates have arbitrarily large fan-out and fan-in. Furst, Saxe and Sipser (1981) and Ajtai (1983) proved independently that every constant-depth circuit computing the parity function has superpolynomial size; the **parity** function maps $(x_1, \ldots, x_n) \mapsto x_1 + \ldots + x_n$, where here, and throughout this section, addition is the mod 2 sum. Yao (1985) established a truly exponential lower bound by extending the techniques of Furst, Saxe and Sipser. Hastad (1986) has further strengthened the bound and greatly simplified the proof. All of these proofs are based on probabilistic combinatorial arguments; the proof of the following theorem can be found in Chapter 33.

2.1 Theorem. *If C is a circuit with n input bits and depth d that computes the parity function, then C has at least $2^{(1/10)n^{1/(d-1)}}$ gates.* ∎

Razborov (1987) gave an exponential lower bound on the size of constant-depth circuits that compute another simple Boolean function, the so-called **majority function**, i.e., the function

$$f(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if at least } n/2 \text{ of the } x_i \text{ are } 1, \\ 0, & \text{otherwise.} \end{cases}$$

In fact, he proved a stronger result by allowing circuits that may have **parity gates**, in addition to the usual AND, OR and NOT gates, where a parity gate computes the parity of the number of 1's in its input. The proof uses the **approximation method**, which was first used by Razborov (1985a) in his pathbreaking paper on monotone circuits. The later application of the method is perhaps the cleanest, and we are able to reproduce the full proof.

2.2 Theorem. *If C is a circuit of depth d, with* AND, OR, NOT *and parity gates that computes the majority function of n input bits, then C has at least $2^{n^{(1/2d)}}/10\sqrt{n}$ gates.*

Proof. Consider a circuit that computes the majority function. We can assume without loss of generality that the circuit uses only parity and OR gates, since these can be used to simulate both AND and NOT gates within constant depth. The idea of the proof is to introduce "approximations" of the gates used during the computation. Using the approximate gates, instead of the real gates, one computes an approximation of the majority function. The quality of the approximation will be measured in terms of the number of inputs on which the modified circuit differs from the original. The main point of the approximation is to keep the computed function "simple" in some sense. We will show that every "simple" function, and in particular the approximation we compute, differs from the majority function on a significant fraction of the inputs. Since the approximation of each gate has a limited effect on the function computed, we can conclude that many approximations had to occur.

Each Boolean function can be expressed as a polynomial over the two-element field $GF(2)$. The measure of simplicity of a Boolean function f for this proof is the degree of the polynomial representing the function or for short, the **degree of the function**.

In fact, the approximation technique is applied not to the majority function, but to a closely related function, the **k-threshold function** f_k. This function is 1 when at least k of the inputs are 1. It is easy to see that if there is a circuit of size s that computes the majority function of $2n-1$ elements in depth d, then, for each k, $1 \leq k \leq n$, there is a circuit of depth d and size at most s that computes the k-threshold function on n elements. Therefore, any exponential lower bound for f_k implies a similar bound for the majority function. We shall consider $k = \lceil (n+h+1)/2 \rceil$ for an appropriate h.

First we show that any function of low degree has to differ from the k-threshold function on a significant fraction of the inputs.

2.3 Lemma. *Let $n/2 \leq k \leq n$. Every polynomial with n variables of degree $h = 2k-n-1$ differs from the k-threshold function on at least $\binom{n}{k}$ inputs.*

Proof. Let g be a polynomial of degree h and let $\mathcal{B}$ denote the set of vectors where it differs from f_k. Let $\mathcal{A}$ denote the set of all 0-1 vectors of length n containing exactly k 1's.

For each Boolean function f, consider the summation function $\hat{f}(x) = \sum_{y \leq x} f(y)$. It is trivial to see that the summation function of the monomial $x_{i_1} \dots x_{i_r}$ is 1 for the incidence vector of the set $\{i_1, \dots, i_r\}$ and 0 on all other vectors. Hence it follows that f has degree at most h if and only if $\hat{f}$ vanishes on all vectors with more than h 1's. In contrast to this, the summation function of the k-threshold f_k is 0 on all vectors with fewer than k 1's, but 1 on all vectors with exactly k 1's.

Consider the matrix $M = (m_{ab})$ whose rows are indexed by the members of $\mathcal{A}$, whose columns are indexed by the members of $\mathcal{B}$, and

$$m_{ab} = \begin{cases} 1, & \text{if } a \geq b, \\ 0, & \text{otherwise.} \end{cases}$$

We want to show that the columns of this matrix generate the whole linear space. This will imply that $|\mathcal{B}| \geq |\mathcal{A}| = \binom{n}{k}$.

Let $a_1, a_2 \in \mathcal{A}$ and let $a_1 \wedge a_2$ denote their coordinatewise minimum. Then we have, by the definition of $\mathcal{B}$,

$$\sum_{\substack{b \leq a_1 \\ b \in \mathcal{B}}} m_{a_2 b} = \sum_{\substack{b \leq a_1 \wedge a_2 \\ b \in \mathcal{B}}} 1 = \sum_{u \leq a_1 \wedge a_2} \big(f_k(u) + g(u)\big) = \sum_{u \leq a_1 \wedge a_2} f_k(u) + \sum_{u \leq a_1 \wedge a_2} g(u).$$

The second term of this last expression is 0, since $a_1 \wedge a_2$ contains at least $h + 1$ 1's. The first term is also 0 except if $a_1 = a_2$. The columns of M therefore generate the unit vector corresponding to the coordinate a_1, and so they generate the whole space. ∎

If p_1 and p_2 are polynomials representing two functions, then $p_1 + p_2$ is the polynomial corresponding to the parity of the two functions. The polynomial $p_1 p_2$ corresponds to their AND, which makes it easy to see that $(p_1+1)(p_2+1)+1$ corresponds to their OR. Note that the inputs have degree 1, i.e., they are very simple. Since the degree is not increased by computing the sum, the parity gates do not have to be approximated. On the other hand, unbounded fan-in OR gates can greatly increase the degree of the computed functions. We will approximate the OR gates so that the approximated function will have fairly low degree. The following lemma will serve as the basis for the approximation.

2.4 Lemma. *Let $g_1, \ldots, g_m$ be Boolean functions of degree at most h. If $r \geq 1$ and $f = \vee_{i=1}^m g_i$, then there is a function f' of degree at most rh that differs from f on at most 2^{n-r} inputs.*

Proof. Randomly select r subsets $I_j \subseteq \{1, \ldots, m\}$ $(1 \leq j \leq r)$, where each i is independently included in I_j with probability $1/2$. Let f_j be the sum of the g_i with $i \in I_j$, and consider $f' = \vee_{j=1}^r f_j$. We claim that the probability that f' satisfies the requirements of the lemma is non-zero. It is clear that the degree of the polynomial for f' is at most rh. Furthermore, consider an input α; we claim that the probability that $f'(\alpha) \neq f(\alpha)$ is at most 2^{-r}. To see this, consider two cases. If $g_i(\alpha) = 0$ for every i, then both $f(\alpha) = 0$ and $f'(\alpha) = 0$. On the other hand, if there exists an index i for which $g_i(\alpha) = 1$, then $f(\alpha) = 1$ and for each j, $f_j(\alpha) = 0$ independently with probability at most $1/2$. Therefore, $f'(\alpha) = 0$ with probability at most 2^{-r}, and the expected number of inputs on which $f' \neq f$ is at most 2^{n-r}. Hence for at least one particular choice of the sets I_j, the polynomial f' differs from f on at most 2^{n-r} inputs. ∎

To finish the proof, assume that there is a circuit of size s and depth d to compute the k-threshold function for inputs of size n. Now apply Lemma 2.4 with $r = \lfloor n^{1/(2d)} \rfloor$ to approximate the OR gates in this circuit. The functions computed by the gates at the ith level will be approximated by polynomials of degree at most r^i. Therefore, each resulting approximation p_k of the k-threshold function will have degree at most r^d. Lemma 2.3 implies that for $k = \lceil (n+r^d+1)/2 \rceil$, the polynomial p_k differs from the k-threshold function on at least $\binom{n}{k}$ inputs. This shows that $s2^{n-r} \geq \binom{n}{k}$. ¿From this, routine calculations yield that

$$s \geq \binom{n}{k} 2^{r-n} > \frac{2^r}{\sqrt{\pi n}},$$

which establishes the desired exponential lower bound. ∎

Smolensky (1987) generalized this result to prove that every constant-depth circuit that decides whether the sum of the inputs is 0 modulo p using AND, OR and modulo-q gates has exponential size, where p and q are powers of different primes.

How far can one relax the assumption on bounded depth and still obtain superpolynomial lower bounds? The methods of Yao, Hastad and Razborov can be extended to depth near $\log n / \log\log n$. One cannot hope for much more, since the parity function can in fact be computed by a linear-size circuit of depth $O(\log n / \log\log n)$.

Perhaps the deepest result on circuit complexity is contained in the groundbreaking paper of Razborov (1985a). He gave a superpolynomial lower bound on the monotone circuit complexity of the clique problem, without any restriction on the depth. Shortly afterwards, Andreev (1985) used similar techniques to obtain an exponential lower bound on a less natural $\mathcal{NP}$-complete problem. Alon and Boppana (1987), by strengthening the combinatorial arguments of Razborov, proved an exponential lower bound on the monotone circuit complexity of the k-clique function.

2.5 Theorem. *If C is a monotone circuit with $\binom{n}{2}$ input bits that decides whether a given graph on n nodes contains a clique with at least s nodes, then the number of gates in C is at least*

$$\frac{1}{8}\left(\frac{n}{4s^{3/2}\log n}\right)^{(\sqrt{s}+1)/2}.$$

∎

The proof uses a much more elaborate application of the approximation technique. The main combinatorial tool used is the "sunflower theorem" of Erdös and Rado (see Chapter 24).

How different can the monotone and non-monotone circuit complexity of a monotone function be? Pratt (1975) proved that the monotone circuit complexity of Boolean matrix multiplication is $\Theta(n^3)$. This, together with the $O(n^{\log 7})$ matrix multiplication technique of Strassen (1969), proves that these two notions are distinct. Razborov (1985b), using techniques similar to those used for the clique lower bound, showed that the perfect matching problem, which is in $\mathcal{P}$, has superpolynomial monotone circuit complexity, thereby establishing a superpolynomial gap. Tardos (1988) showed that this could be increased to an exponential separation, by combining the arguments of Razborov (1985a), Alon and Boppana (1987) and results of Grötschel, Lovász and Schrijver (1981) on the polynomial computability of a graph function ϑ that is closely related to the clique function (see Chapter 31).

As remarked above, no methods are known to handle general circuits with depth greater than $\log n$. In the case of monotone circuits with fan-in 2, however, a version of the random restriction method has been successfully applied by Karchmer and Wigderson (1990) to prove a lower bound on the depth proportional to $\log^2 n$. It is clear that a circuit with fan-in 2 and size N must have depth at least $\log N$; hence Theorem 2.5 implies that every monotone circuit with fan-in 2 computing the k-clique function must have depth $n^{1/3-\varepsilon}$. However, the function that Karchmer and Wigderson consider is computable by polynomial-size monotone circuit, and so no non-trivial bound on the depth is implied by only considering its size.

Karchmer and Wigderson considered the **undirected reachability problem**: given a graph G and two nodes s and t, is there an $s-t$ path in G? This problem is clearly in $\mathcal{P}$, and in fact, it can be decided by a polynomial-size monotone circuit that has depth $O(\log^2 n)$. Karchmer and Wigderson proved the following result:

2.6 Theorem. *There exists a constant $c > 0$ such that if C is a monotone circuit with fan-in 2 that solves the undirected reachability problem for a graph on n nodes, its depth is least $c\log^2 n$.*

The proof, which uses a version of the random restriction method, is quite involved and is not given here. We describe, however, the starting point, which is a new characterization of the depth of circuits with fan-in at most 2 in terms of communication complexity, thereby establishing a surprising link with the material of the previous section.

Consider the following game between two players. The game is given by a Boolean function f in n variables. The first player gets $x \in \{0,1\}^n$ such that $f(x) = 0$ and the second player gets $y \in \{0,1\}^n$ such that $f(y) = 1$. The goal of the game is that the two players should agree on a coordinate i such that $x_i \neq y_i$. Let $\kappa(f)$ denote the minimum number of bits that the two players must communicate to agree on such a coordinate. (For example, the first player could tell x to the second player, and then the second player can find an appropriate coordinate to tell to the first player, so $\kappa(f) \leq n + \log n$.) Karchmer and Wigderson proved that the minimum depth in which a Boolean function f can be computed with a circuit with fan-in 2 is equal to $\kappa(f)$. A similar characterization can be given for the monotone circuit complexity of monotone Boolean functions.

In the case of the undirected reachability problem, the corresponding game can be phrased as follows: the first player is given an $[s,t]$-path and the second player is given an $[s,t]$-cut, and the goal of the game is to find an edge in the intersection of the path and the cut. Consider the following protocol for this connectivity game: the first player sends the name of the midpoint on the path, and the second player responds by telling on which side of the cut this node lies. This protocol requires $O(\log n)$ rounds, and in each round the first player sends $\log n$ bits and the second player sends 1.

Karchmer and Wigderson prove that even if the second player were allowed to send $O(n^\epsilon)$ bits in each round (instead of just 1), the players would still need at least $\Omega(\log n)$ rounds. The claimed lower bound on the monotone circuit depth is a consequence of this fact.

3. Data structures

Imagine a huge science library. It contains a wealth of information, but to make this information useful, catalogues, reference and review volumes, indices (and librarians) are needed. Similarly, information in the memory of a computer is useful only if it is accessible, i.e., it is provided with extra structures that make the storage, retrieval, and modification of this information possible, and in fact easy. This is particularly important when implementing complicated algorithms: the fast storage and retrieval of certain partial results of the computation is often the main bottleneck in speeding up such algorithms. Such auxiliary structures, called **data structures**, are becoming increasingly important as the amount of information stored increases.

The theory of data structures is very broad and we shall restrict ourselves to two examples that illustrate the depth of combinatorial ideas used in this field. For a more thorough treatment, see Aho, Hopcroft and Ullman (1983), Tarjan (1983) and Gonnet (1984).

a. Shortest paths and Fibonacci heaps. Let $G = (V(G), E(G))$ be a graph with n nodes and m edges, and with a specified node s. Let every edge e have a non-negative length $c(e)$. We want to find a shortest path from s to every other node. We have seen in Chapter 2 that using Dijkstra's algorithm, this is quite easily done in polynomial ($O(n^2)$) steps. (To be precise, we use the RAM machine model of computation, and a step means an arithmetic operation — addition, multiplication, comparison, storage or retrieval — of numbers whose length is at most a constant multiple of the maximum of $\log n$ and the length of the input parameters.) Let us review this procedure.

We construct a subtree T of G, one node at a time. We shall only be concerned with the nodes of this tree, so we consider T as a set of nodes. Initially, we let $T = \{s\}$. At any given step, for each $v \in T$ we know the length of the shortest path from s to v, i.e., the distance $d(s, v)$. It would be easy to also obtain the edges of the tree; then the unique $[s, v]$-path in this tree realizes this distance.

The essence of Dijkstra's algorithm is to find an edge $uv \in E(G)$ with $u \in T$ and $v \in V(G) \setminus T$ for which $d(s, u) + c(uv)$ is minimal, and then add v to T. As shown in Chapter 2, we then have that $d(s, v) = d(s, u) + c(uv)$. The issue is to find this edge economically. At first glance, we have to select the smallest member from a set of size $O(m)$, and we have to repeat this n times, so this rough implementation of Dijkstra's algorithm takes $O(mn)$ steps.

We can easily do better, however, by keeping track of some of the partial results that we have obtained. Let S denote the set of nodes not in T but adjacent to T. For each node $v \in S$, we maintain the value $\ell(v) = \min\{d(s, u) + c(uv)\}$, where the minimum is taken over all edges uv with $u \in T$. For $v \notin S$, we define $\ell(v) = \infty$ for notational convenience. Clearly, $\ell(v)$ is an upper bound on $d(s, v)$. At the beginning, $\ell(v) = c(sv)$ for each neighbor v of s and $\ell(u) = \infty$ for each other node u. A step then consists of (a) selecting the node $v \in S$ for which $\ell(v)$ is minimum and setting $d(s, v) = \ell(v)$, (b) deleting v from S and adding it to T, (c) adding each neighbor w of v that is in $V(G) \setminus (T \cup S)$ to S, and (d) updating the value $\ell(w)$ for each neighbor w of v that is in S by

$$\ell(w) := \min\big\{\ell(w), d(s, v) + c(vw)\big\}.$$

This way, it takes $O(n)$ steps to select the node $v \in S$ for which $\ell(v)$ is minimum, and so the total number of steps spent on selecting these nodes is $O(n^2)$. There is also the time needed to update the values $\ell(w)$: this is a constant number of steps per node, and we have at most $d(v)$ nodes to consider, where $d(v)$ is the degree of v. Updating therefore takes $O\big(\sum_v d(v)\big) = O(m)$ steps, which is dominated by $O(n^2)$; by maintaining the current best path lengths, Dijkstra's algorithm takes $O(n^2)$ steps.

Can we do better? It is natural to assume that we have to take at least m steps, in order to read the data. If m is proportional to n^2, the algorithm just described is best possible. But for most real-life problems, the graph is sparse, i.e., m is much smaller than n^2, and then there is space for improvement. To obtain this improvement, we shall store the set S, together with the values $\ell(v)$, in a clever way.

A first idea is to keep the set S sorted in order of the value of $\ell(v)$. This makes it trivial to select the next node v and delete it from S, but to achieve (c) and (d), we insert new items in the sorted list, which can be done in $O(\log n)$ steps per insertion. However, even this is non-trivial. If we simply store the sorted elements of S in an array, i.e., in consecutive fields, then to insert a new element, we expect to move half of the old elements for each insertion. Another possibility is to store these elements in a **list**: each element is stored along with a **pointer**, which specifies the memory location of the next element in the list. In this data structure, insertion is trivial, but to find the point of insertion, we must traverse the list, which takes a linear number of steps. Advantages of both methods can be combined using a data structure called a **binary search tree**. We shall not discuss these in detail, since we will show how to do better with another kind of data structure, called a heap. Nonetheless, Dijkstra's algorithm with the current best path lengths stored in a binary search tree takes $O(m \log n)$ steps. This may be much better than $O(n^2)$, but there is still room for improvement.

At this point, it is worth while to formulate the requirements of the desired data structure in an axiomatic way. We have some objects, the elements of S, which are to be stored. Each object has a value $\ell(v)$ associated with it, which is called its **key**. Reviewing the algorithm, we see that we must perform the following operations on this collection of data:

(3.1) DELETEMIN. We might want to find the element of S with smallest key and delete it from S (steps (a) and (b)).

(3.2) INSERT. We might want to add new elements to S (step (c)).

(3.3) DECREASEKEY. We might change the key of elements of S; in fact, we only need to decrease it (step (d)).

Observe that (3.1) and (3.2) are performed $O(n)$ times, since every node is added at most once and deleted once, whereas (3.3) is performed $O(m)$ times.

As mentioned above, a heap is a data structure that can handle these operations in logarithmic time. Since DECREASEKEY is performed more often than the other two operations, we can improve the overall running time by decreasing the cost of performing just this operation. Fredman and Tarjan (1987) showed how to do this by using a more sophisticated data structure, called a Fibonacci heap.

A **heap** is a rooted tree defined on the elements of S, with the property that the key of any node is no more than the key of any of its children. In particular, the root is an element with the smallest key. If the tree is a single path, then the heap is a sorted list, but it will be worth while to consider more compact trees.

Before deciding about the shape of the heap, let us discuss how to perform the tasks (3.1)–(3.3). The heap itself can be realized by maintaining a pointer from each non-root node to its parent. Moreover, the children of each node are ordered in an arbitrary way, and each child contains a pointer to the previous child as well as to the next child. Each parent maintains pointers to its first and last child. Each node has five pointers, some of which may be undefined: parent, first child, last child, previous sibling, next sibling. Changes in the heap are made by manipulating these pointers.

The most common way to implement operations (3.1)–(3.3) in a heap is as follows. DECREASEKEY is perhaps the easiest. Assume that we decrease the key of an element p_0. Let $p_0 p_1 \dots p_t$ be the path in the tree connecting p_0 to the root. If $\ell(p_0)$ is still at most $\ell(p_1)$ then we still have a heap; otherwise, we interchange the elements p_0 and p_1. Next we compare the key of p_0 with the key of p_2; if $\ell(p_0) \le \ell(p_2)$, we have a heap; otherwise, interchange p_0 and p_2, and so on. After at most t interchanges we end up with a heap. Note that the tree has not changed, only the vertices have been permuted.

INSERT can be reduced to DECREASEKEY: if we want to add a new element w then we can assign to it a temporary key $+\infty$, and make it the child of any preexisting element. Trivially, this produces a heap. We can then decrease the key of w to the right value, and reorder the elements as before.

Finally, DELETEMIN can be performed as follows. Let r be the root element of the heap. Select any leaf p of the tree and replace r by p. This interchange deletes the root, but we do not necessarily have a heap, since the key of p may be larger than the key of one or more of its children. Find the child with smallest key, and interchange that child with p. It is easy to see that the resulting tree again can violate the heap condition only at the node p. If p has larger key than some of its children, then find its child with the smallest key and interchange them; and so on. Eventually, we obtain a heap.

The **height** of a node in a rooted tree is the maximum distance to a leaf; the height of the tree is the height of its root. If the tree has height h then the operations INSERT and DECREASEKEY take $O(h)$ steps; to make these operations efficient, we want the tree as short as possible. But DELETEMIN puts limits on this: it also involves $O(h)$ interchanges, but before each interchange, we must also find the child with smallest key, and this takes roughly d steps for a node with d children. If we use **balanced k-ary trees** in which, with at most one exception, all internal nodes have k children and each leaf is at distance h or $h-1$ from the root, then $h \le \log_k n$ and so the total number of steps is $O(n \log_k n + m \log_k n + nk \log_k n)$. The best choice is $k = 2 + (m/n)$, and this shows that Dijkstra's algorithm with the current best path lengths stored in a k-ary heap takes $O(m \log n / \log(2 + (m/n)))$ steps, which is a slight improvement.

One can use more sophisticated trees with a more sophisticated implementation of the basic operations and with a more sophisticated way to count steps. A rooted tree is called a **Fibonacci tree** if

(3.4) for every node v and natural number k, the number of children of v with degree at most k is at most $k+1$.

(We use the term degree in the graph-theoretic sense: the degree of a non-root node is one larger than the number of its children.)

The **degree** of the tree is the degree of the root. We want to build heaps on such trees. For technical reasons, it will be convenient to add an artificial element r with key $-\infty$; hence r is always the root. Moreover, for the root we shall impose the following condition, stronger than (3.4):

(3.5) The degrees of the children of r are distinct.

A **Fibonacci heap** is a heap whose underlying tree is a Fibonacci tree that satisfies condition (3.5). We are going to show that by using Fibonacci heaps, we can implement Dijkstra's algorithm to run in $O(m + n \log n)$ steps, which is best possible for every $m \geq n \log n$. For very sparse graphs, this is, in some sense, an optimal implementation of Dijkstra's algorithm, but other algorithms may be better.

Note that the subtree of a Fibonacci tree formed by a node and its descendants is also a Fibonacci tree. If we delete a node and its descendants then the only node where condition (3.4) could be violated is the grandparent of the deleted node. In particular, if we delete a child of the root and its descendants, we are left with a Fibonacci tree. By applying induction to this observation, we obtain the following lemma, which explains the name Fibonacci tree.

3.6 Lemma. *Let $F_0 = 0$ and $F_1 = 1$. Then the number of nodes in a Fibonacci tree of degree k is at least F_{k+2}, the $(k+2)$nd Fibonacci number.* ∎

It follows that a Fibonacci tree with n nodes has degree $O(\log n)$. More generally, each node has degree $O(\log n)$, which follows by considering the Fibonacci tree formed by this node and its descendants.

Assume now that we have a Fibonacci heap, and we want to perform INSERT, DELETEMIN and DECREASEKEY operations. In each case, we will first produce a heap that satisfies the Fibonacci property at all non-root nodes; we will then give a procedure that restores the stronger property (3.5) at the root.

(a) INSERT. Add the new node x as a child of r.

(b) DELETEMIN. Of course, we do not want to delete the root, but the minimal "real" element. Find the child of the root with the smallest key, delete it, and make its children have the root as their new parent.

(c) DECREASEKEY. Suppose that we want to decrease the key of a node v_1. If v_1 is a child of the root, simply decrease the key. Otherwise, let $v_1 v_2 \ldots v_t r$ be the path connecting v_1 to the root. Delete the edge connecting v_1 to its parent v_2, and let v_1 become a child of the root. The resulting tree satisfies the heap condition even with the decreased key. Moreover, condition (3.4) is satisfied by all non-root nodes except possibly by v_3, the grandparent of v_1. Until condition (3.4) is satisfied at all non-root nodes, fix the violation at v_j by making v_{j-1} a child of the root. After at most t steps, we obtain a tree that satisfies (3.4) for all non-root nodes.

(d) For each of the three operations, we finish by restoring property (3.5). To do this, we look at the degrees of the root's children, and assume that two children u and v have the same degree t. Suppose that the key of u is no more than the key of v; then we delete the edge vr and make v a child of u. We will show that property (3.4) remains valid at every non-root. This is trivial for all nodes except u. To see that it holds for u, consider any

$s \geq 1$. If $s < t$ then u has the same children with degree at most s as before, and so their number is at most $s+1$. For $s \geq t$, u now has at most s children altogether. (It will be important that a nearly identical argument applies even if a child of v were deleted!(*))

If we find two children of the root with the same degree then we can transform the heap into one where (3.4) is still valid at all non-roots, and the root has lower degree. We repeat this until all children of the root have different degrees; then (3.4) is trivially satisfied at the root.

There are two points to clarify.

— How do we know in (c) how many edges $v_i v_{i+1}$ must be deleted? To check condition (3.4) for each v_i would take too much time. Instead, we will classify each node as either **safe** or **unsafe**. If a node is safe, then deleting one of its children will not violate (3.4) at a non-root node. In contrast, classifying a node as unsafe implies only that we are not sure if such a violation would occur. Thus, we can always classify the root and its children as safe. In particular, each newly inserted node is safe. We shall reclassify a node in only two cases:

(i) If an unsafe node becomes a child of the root (in a DELETEMIN or DECREASEKEY operation), then it is reclassified as safe.

(ii) If a safe node different from the root loses a child (in a DECREASEKEY operation), then it is reclassified as unsafe.

It follows that in the DECREASEKEY operation, we delete all edges of the path $v_1 v_2 \ldots v_t r$ up to the first safe node v_j, and make $v_1, \ldots, v_{j-1}$ children of the root. We reclassify v_j as unsafe, and $v_1, \ldots, v_{j-1}$ as safe. Note that in the parenthetical remark (*), we have already indicated that when r gets a new grandchild v in step (d), it is correct to still classify v as safe.

— In performing (d), how do we find those children of the root with the same degree? To sort the degrees would be too time-consuming. We wish to perform (d) in $O(d)$ steps, where d is the degree of the root after performing steps (a)-(c). For each node, we can always store its current degree in an array. We will also maintain an array a, where $a[k]$, $k = 1, \ldots, d$, indicates the name of a child of the root of degree k, if one exists. This array is not changed during steps (a)-(c), but is updated during step (d) instead. It is trivial to update a to reflect the deletion of a child of the root. To update a for the new children of the root added during steps (a)-(c), we consider them one at a time. To update a to reflect the next child u, if u has degree k then we check if $a[k]$ contains the name of a node. If not, we let $a[k] := u$. If it already contains the name of a child v, then we make one of u and v the child of the other, and update $a[k]$ accordingly. This creates a child of degree $k+1$, which we must then be checked with $a[k+1]$, and so forth. Since each "collision" of two children with the same degree causes the degree of r to decrease, there are fewer than d collisions overall, and so the new children are added in $O(d)$ time. In fact, if adding t children causes c collisions, the total work of step (d) is proportional to $t + c$.

We must still bound the time needed to perform these operations. It is easy to see from Lemma 3.6 that INSERT and DELETEMIN operations take $O(\log n)$ steps. But a DECREASEKEY operation may take enormous time! For example, if the (Fibonacci) tree is a single path from the root, with all nodes but the root and its child unsafe, then it takes about n steps to decrease the key of the bottom node. Furthermore, if the root has roughly $\log n$ children, then adding just a single child of the root could take roughly $\log n$ steps.

The remedy is a book-keeping trick called **amortization of costs**. Imagine that we maintain the Fibonacci heap as a service. The customer may order any of the INSERT, DELETEMIN and DECREASEKEY operations. For each operation, we ought to charge him the actual cost, say a cent for each step. But suppose that we also require that he pay a deposit of one dollar for each unsafe node that is created, and a deposit of 25 cents for each child of the root that is created. If either the number of unsafe nodes or the number of children of the root decreases, the appropriate deposit is refunded. With this billing system, an INSERT or DELETEMIN operation still costs only $O(\log n)$ cents, but now we can bound the cost (to him) of a DECREASEKEY operation. Let t be the number of nodes to be made children of the root; this is at most one larger than the number of unsafe nodes becoming safe. Since in step (c), at most one node becomes unsafe, the customer then gets a net refund of at least $75t-200$ cents. The actual cost of step (c) is proportional to t, certainly at most $50t$. The actual cost of step (d) is proportional to $t+c$, certainly at most $25(t+c)$. However, in step (d), the customer also gets a refund of $25c$ cents. The total cost to the customer of steps (c) and (d) is at most 2 dollars: with this billing system, a DECREASEKEY operation costs only a constant amount.

Summarizing, we have shown the following theorem.

3.7 Theorem. *In a Fibonacci heap, performing n INSERT, n DELETEMIN and m DECREASEKEY operations takes $O(n\log n+m)$ steps.* ∎

For our original problem we get the following result.

3.8 Theorem. *Dijkstra's algorithm can be implemented, using Fibonacci heaps, in $O(n\log n+m)$ steps.* ∎

b. Shortest spanning trees and the UNION–FIND problem. Many of the data structures discussed in the previous subsection can also be used in computing a shortest spanning subtree of a graph. In particular, Fibonacci heaps can be used to implement Prim's algorithm to run on a connected graph G with n nodes and m edges in $O(m+n\log n)$ time. However, in some cases, we may already know the sorted order of the lengths of the edges, or can find this sorted order extremely quickly, such as when the lengths are known to be small integers. In these cases, we can obtain an even more efficient algorithm by using Kruskal's algorithm implemented with a data structure with surprising combinatorial complications. For the following discussion, assume that the sorted order of the edge lengths is known in advance.

Kruskal's algorithm is very simple: it takes the edges one-by-one in the given sorted order, and it adds the next edge to a set T if it does not form a circuit with the edges already in T; otherwise, it disposes of the edge. This seems to take m steps, except that we must check whether the new edge forms a circuit with T. Let uv be the edge considered.

To search T for a $[u, v]$-path would be too time-consuming; it would lead to an $O(mn)$ implementation of Kruskal's algorithm.

We can do better by maintaining the partition $\{V_1, \ldots, V_k\}$ of $V(G)$ induced by the connected components of the graph $(V(G), T)$. Each iteration amounts to checking whether u and v belong to the same class; if not, we add uv to T. Furthermore, updating the partition is simple: adding uv to T will cause the two classes containing u and v to merge.

To implement Kruskal's algorithm efficiently, we must therefore find a good way to store a partition of $V(G)$ so that the following two operations can be performed efficiently:

(3.9) FIND. Given an element u, return the partition class containing u.

(3.10) UNION. Given two partition classes, replace them by their union.

We assume that each partition class has a name, and for our purposes, it will be convenient to use an arbitrary element of the class to name the partition class. We shall call this element the **boss**. In a UNION operation, we can keep either one of the old bosses as the new boss. Kruskal's algorithm uses $2m$ FIND operations and $n - 1$ UNION operations.

A trivial way to implement a UNION-FIND structure is to maintain a pointer for each element, pointing to the boss of its class. A FIND operation is then trivial; it takes only one step. On the other hand, to do a UNION operation we may have to redirect almost n pointers, which yields an $O(n^2)$ implementation of Kruskal's algorithm. This is unsatisfactory for sparse graphs, and so we must do the UNION operation more economically.

An almost trivial observation already yields a lot: when merging two classes, we redirect the pointers in the smaller class. We call this rule **selection by size.** To estimate the number of steps needed when using this rule, observe that whenever a pointer is redirected, the size of the class containing it gets at least doubled. Hence each pointer is redirected at most $\log n$ times. The total number of steps spent on UNION operations is therefore $O(n \log n)$, and we get an $O(m + n \log n)$ implementation of Kruskal's algorithm. For $m = \Omega(n \log n)$, this is optimal.

To be able to do better for really sparse graphs (e.g., with a linear number of edges), we use a more sophisticated way to keep track of the boss. We shall maintain a rooted tree on each class, with the boss as its root. Each UNION operations then takes only constant time: we make one boss the child of the other. But this makes the FIND operation more expensive: we have to walk up in the tree to the root, so it may take as much time as the height of the tree. This suggests that we should keep the trees short. We can use an analogue to the selection by size rule, called **selection by height**: when merging the two trees, if r_1 is the root of greater height, then it is made the parent of r_2, the root of the other tree. This increases the height only if the two trees had the same height.

Of course, it is time-consuming to compute the height of the trees at each UNION operation, but we can maintain the height $h[v]$ for each node v. This is easily updated: it changes only if the union of two trees is performed and the roots have $h[r_1] = h[r_2]$; then 1 is added at the new root. It is easy to verify by induction that the number of elements in a tree with root r is at least $2^{h[r]}$. Hence $h[r] \leq \log n$ for every r, and the cost of a FIND operation is $O(\log n)$. This does not yet yield any improvement in the implementation of Kruskal's algorithm.

But we can use another idea, called **path compression**. Suppose that we perform a FIND operation which involves traversing a fairly long path $v_1 \ldots v_k r$. Then we can traverse this path again, and make each v_i the child of the root. This doubles the number of steps, but the tree becomes shorter.

We combine this idea with a variant of selection by height, called **selection by rank.** For each element v, we maintain a number $\rho[v]$ called its **rank**. The rank of each node is initially 0; if two trees with roots r_1 and r_2 are merged, where $\rho[r_1] \geq \rho[r_2]$, we make r_2 a child of r_1, and update ρ by setting

$$\rho[v] := \begin{cases} \rho[r_1]+1, & \text{if } v = r_1 \text{ and } \rho[r_2] = \rho[r_1], \\ \rho[v], & \text{otherwise.} \end{cases}$$

A path compression does not change ρ. The number $\rho[v]$ is no longer the height of v, but it will be an upper bound on the height. Moreover, the number of elements in a tree with root v is still at least $2^{\rho[v]}$. We shall need the following generalization of this fact, which also follows by induction.

3.11 Lemma. *The number of elements v with $\rho[v] = t$ is at most $n/2^t$.* ∎

For each leaf v, $\rho[v] = 0$, ρ is strictly increasing along any path to the root. Note that this guarantees that the height of the tree is at most $\log n$.

Tarjan (1975) showed that using selection by rank with path compression reduces the cost substantially: the average cost of a FIND operation grows only very, very slowly.

Let us recall Ackerman's function from Chapter 25. First, we define a series of functions $f_i : \mathbb{N} \to \mathbb{N}$ by the double recurrence

$$f_1(n) = 2n, \qquad f_i(0) = 2 \quad (i \geq 2),$$

$$f_{i+1}(n) = f_i(f_{i+1}(n-1)).$$

Thus, $f_2(n) = 2^{n+1}$, $f_3(n)$ is roughly a tower of $n+1$ 2's, and so on. Ackerman's function is the diagonal $A(n) = f_n(n)$. This grows faster than any f_i, and in fact, faster than any primitive recursive function. The **inverse Ackerman function** $\alpha(n)$ is defined by

$$\alpha(n) = \min\{k : \ A(k) \geq n\}.$$

This function grows extremely slowly (e.g., slower than $\log\log n$ or $\log^* n$). We shall also need to introduce inverse functions ϕ_i of the f_i, defined by

$$\phi_i(n) = \max\{k : \ f_i(k) \leq n+2\}.$$

The definition is made so that for fixed $n \geq 3$, $\phi_i(n)$ is strictly decreasing as a function of i until it reaches 0. Tarjan's result, in a slightly weakened form, is as follows:

3.12 Theorem. *In a rooted forest with n elements, using selection by rank and path compression, m FIND operations and at most $n-1$ UNION operations take $O((n+m)\alpha(n))$ steps.*

Proof. A UNION operation takes only a constant number of steps. To analyze the FIND operation, we first develop a few tools. For each non-root node v, let v' denote its parent. We define the **level** of a node v as the least i for which $\phi_i(\rho[v]) = \phi_i(\rho[v'])$; a root is on level 0. (Note that if $\phi_i(\rho[v]) = \phi_i(\rho[v'])$ then $\phi_{i+1}(\rho[v]) = \phi_{i+1}(\rho[v'])$.) Initially, each node is therefore on level 0. When a node becomes a non-root, it then reaches a positive level. Note that from this step on, $\rho[v]$ does not change. The value $\rho[v']$ can change in only two ways: v may get a new parent (from a path compression) or v has the same parent and yet $\rho[v']$ changes (from a UNION operation). In either case, $\rho[v']$ increases. Hence the level of the node v can only increase with time. On the other hand, the level of a node remains very small: it is bounded by the least i for which $\phi_i(n) = \phi_i(0)$, which is easily seen to be at most $2\alpha(n+3)$, which is $O(\alpha(n))$.

Let $v_1 \dots v_k r$ be a path that is being compressed, where r is a root. The cost of this compression is proportional to k; using the charging analogy again, let us say at most k dollars. Consider a node v_j $(1 \le j \le k)$ on the path, and let i be its level. If $\phi_{i-1}(\rho[r]) > \phi_{i-1}(\rho[v_{j+1}])$, then charge one dollar to the node. Otherwise, charge one dollar to the customer.

How much do we charge to the customer per path compression? If he is charged for a node v_j at level i, then by the monotonicity of ϕ_{i-1} and ρ, we see that $\phi_{i-1}(\rho[v_{j+1}]) = \phi_{i-1}(\rho[v_{j+2}]) = \cdots = \phi_{i-1}(\rho[r])$. This implies that $v_{j+1}, \dots, v_k, r$ are at level $i-1$ or lower, i.e., v_j is the last node at level i on the path. So the customer gets charged for at most one node at each level, which is a total charge of $O(\alpha(n))$. For m path compressions, this is a total of $O(m\alpha(n))$.

How much charge is left on the nodes? While a node v stays on level i, the value $\phi_{i-1}(\rho[v'])$ increases whenever v is charged a dollar, and so the total charge it accumulates is bounded by the maximum value $\phi_{i-1}(\rho[v'])$ can attain. Note that $\phi_i(\rho[v']) = \phi_i(\rho[v])$ by the definition of level i, and so from the definition of ϕ_i we have

$$\begin{aligned}\rho[v'] + 3 &\le f_i\big(\phi_i(\rho[v']) + 1\big) = f_{i-1}\Big(f_i\big(\phi_i(\rho[v'])\big)\Big) = f_{i-1}\Big(f_i\big(\phi_i(\rho[v])\big)\Big) = \\ &\le f_{i-1}(\rho[v] + 2),\end{aligned}$$

and hence,

$$\phi_{i-1}(\rho[v']) \le \rho[v] + 1.$$

So the charge accumulated by a node v while at level i is at most $\rho[v] + 1$, and since $\rho[v]$ never decreases, we can use the final value in this bound. Adding this over all nodes, we can use Lemma 3.11 to show that the total contribution of level i to the charges to the nodes is at most

$$\sum_v 1 + \rho[v] \le n + \sum_t t\frac{n}{2^t} < 3n.$$

Since there are $O\big(\alpha(n)\big)$ levels, the total charge to the nodes is $O\big(n\alpha(n)\big)$. ∎

3.13 Corollary. *Kruskal's algorithm, with pre-sorted edges, can be implemented in $O\big(m\alpha(n)\big)$ steps.* ∎

For a discussion of other implementations of Kruskal's algorithm and its relatives, see Tarjan (1983).

4. Cryptography and pseudorandom numbers

Let $f : \{0,1\}^* \to \{0,1\}^*$ be a one-to-one function, and assume that this function can be evaluated in polynomial time. Can the inverse function be evaluated in polynomial time? We don't know the answer, but it is quite likely that the answer is in the negative; there are some suspected examples (and some constructions that have been disproved). It turns out that such **one-way functions**, and their extensions, could be used in various important applications of complexity theory. We examine some of these applications and constructions.

a. Cryptography. The basic goal of cryptography is to provide a means for private communication in the presence of adversaries. Until fairly recently, cryptography has been more of an art than a science. No guarantees were given for the security level of the codes, and in fact, all the classical codes were eventually broken. Modern cryptography has suggested ways to use complexity theory for designing schemes with provable levels of security. We will not be able to discuss in detail, or even mention, most of the results here. We intend rather to give a flavor of the problems considered and the results proved. The interested reader is referred to the surveys by Rivest (1990), Goldreich (1988), Goldwasser (1989), or to the special issue of SIAM Journal on Computing (April, 1988) on cryptography.

A traditional solution uses secret-key crypto-systems. Suppose that two parties wish to communicate a message M, a string of 0's and 1's of length n, and they agree in advance on a secret key K, a string of 0's and 1's of the same length as M. They send the encrypted message $C = M \oplus K$ instead of the real message M, where $\oplus$ denotes the componentwise addition over $GF(2)$. This scheme is provably secure, in the sense that, assuming the key K is kept secret, an adversary can learn nothing about the message by intercepting it: every string of length n is equally likely to be the message encoded by the string C. Unfortunately, it is very inconvenient to use this system since before each transmission the two parties must *agree* on a new secret key.

In the paper that founded the area of modern cryptography, Diffie and Hellman (1976) suggested the idea of public-key crypto-systems. They proposed that a transmission might be sufficiently secure if the task of decryption is computationally infeasible, rather than impossible in an information theoretic sense. Such a weaker assumption might make it possible to securely communicate without *agreeing* on a new secret key every time, and also to achieve a variety of tasks previously considered impossible.

We illustrate the idea of security through intractability by a simple example. Assume that you have a bank account that can be accessed electronically by a password. If the password is long enough, and you keep it safely, then this is secure enough. Except that the programmer of the computer may inspect the memory, learn your password, and then access your account. It would seem that there is no protection against this: the computer has to remember the password, and a good hacker can learn anything stored in the computer.

But the computer does not have to know your password! When you open your account you first generate a Hamiltonian circuit H on, say 10,000 nodes, and then add edges arbitrarily to obtain a graph G. The computer of the bank will only store the graph G while your password is the code of a Hamiltonian circuit H. When you punch in your password the computer checks whether it is the code of a Hamiltonian circuit in G, and lets you access the account if it is. For you, the Hamiltonian circuit functions like a password in the usual sense.

But what about the programmer? He can learn the graph G; but to access your account, he should specify a Hamiltonian circuit of G, which is a computationally intractable task. This intractability protects your account!

The crucial point here is that given a Hamiltonian circuit, it is easy to construct a graph containing it, but given a graph, it can be quite hard to find a Hamiltonian circuit in it. Unfortunately, it is easy to find a Hamiltonian circuit in most graphs, and there is no way known to produce graphs for which finding the Hamiltonian circuit is expected to be difficult.

These considerations motivate the following definition. A **one-way function** is a one-to-one function $f : \{0,1\}^* \to \{0,1\}^*$ such that f can be computed in polynomial time, but no polynomial-time algorithm can invert f on even a polynomial fraction of the instances of length n. Given a one-way function f, we can choose an $x \in \{0,1\}^*$ as our password and store the value $f(x)$ in the computer. The length of the word chosen is the **security parameter** of the scheme. This is indeed done in practice. (The above scheme with Hamiltonian circuits leads to a somewhat weaker notion: a "one-way relation", but for most applications, we need a one-way function.)

The password scheme above is just a simple example of what can be achieved by a **public-key crypto-system**. This can have any number of participants. The participants agree on an encryption function Φ, a decryption function Ψ, and a security parameter n. Messages to be sent are divided into pieces of length n. The system functions as follows:

— Each participant should randomly choose a public encryption key E and a corresponding secret decryption key D depending on the security parameter n. A directory of the public keys is published.

— There must be a deterministic polynomial-time algorithm that, given a message M of length n and an encryption key E, produces the encrypted message $\Phi(E, M)$.

— Similarly, there must exist a deterministic polynomial-time algorithm that, given a message M of length n and a decryption key D, produces the decrypted message $\Psi(D, M)$.

— It is required that for every message M, $\Psi(D, \Phi(E, M)) = M$.

and the crucial security requirement:

— One cannot efficiently compute $\Psi(D, M)$, without knowing the secret key D. More precisely, for every constant c and sufficiently large n, the probability that a (randomized) polynomial-time algorithm using the public key E but not the private key D can decrypt a randomly chosen message of length n is less than n^{-c}.

When a user named Bob wants to send a message M to another user named Alice, he looks up Alice's public-key E_A in this directory, and sends her the encrypted message $\Phi(E_A, M)$. By the assumptions, only Alice can decrypt this message using her private key D_A.

The question is: how do we find such encryption and decryption functions? The basic ingredient of such a system is a "trapdoor" function. The encryption function $\Phi(E, .)$ for a fixed participant must be easy to compute but hard to invert, i.e., a one-way function; but in order for a one-way function to be useful in the above scheme, we need a further feature: it has to be a **trapdoor function**, which is a 2-variable function like Φ in the scheme above: $\Phi : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ such that for every $E \in \{0,1\}^n$, $f(E, .)$ is one-to-one and easily computable, but its inverse is difficult to compute, unless one knows the "key" D belonging to E.

Given the present state of complexity theory, there is little hope to prove that any particular function is one-way or trapdoor (or even that one-way functions exist), or that a public-key crypto-system satisfies the last requirement above. Note that the existence of a one-way function would imply that $\mathcal{P} \neq \mathcal{NP}$. Therefore, a more realistic hope would be to prove the security of a crypto-system based on the assumption that $\mathcal{P} \neq \mathcal{NP}$. However, this seems to be quite difficult for two reasons. Complexity theory is mainly concerned with worst-case analysis. For the purpose of cryptography, average-case analysis, and a corresponding notion of completeness (such as the one suggested by Levin (1986)) would be more appropriate. Furthermore, one-way functions lead to languages in $\mathcal{NP}$ which have a unique "witness", whereas the nondeterministic algorithms for $\mathcal{NP}$-complete problems have several accepting paths for most instances.

Over the last ten years, several public-key crypto-systems have been suggested and analyzed. Some have been proven secure, based on assumptions about the computational difficulty of a particular problem (e.g., factoring integers), or on the more general assumption that one-way functions exist.

Rivest, Shamir and Adleman (1978) were the first to suggest such a scheme. Their scheme is based on the assumed difficulty of factoring integers. Each user of this scheme has to select a number n that is the product of two random primes. Random primes can be selected using a randomized primality testing algorithm, since every $(\log n)$th number is expected to be prime. The public key consists of a pair of integers (n, e) such that e is relative prime to $\phi(n)$, where $\phi(n)$ denotes the number of integers less than n relatively prime to n. A message M is encrypted by $C = M^e \pmod{n}$. The private key consists of integers (n, d), where $d \cdot e \equiv 1 \pmod{\phi(n)}$, and decryption is done by computing $C^d = M \pmod{n}$. Given the prime factorization of n, one can find the required private key d in polynomial time, but the task of finding the appropriate d given only n and e is equivalent to factoring.

Rabin (1979) suggested a variation on this scheme in which any algorithm for decryption, rather than any algorithm for finding the decryption key, can be used to factor an integer. The idea is to encrypt a message M by $C = M^2 \pmod{n}$. (A slight technical problem that has to be overcome before turning this into an encryption scheme is that squaring modulo a product of two primes is not a one-to-one function, but rather is four-to-one.) An algorithm that can extract square roots modulo n can be used to factor: choose a random integer x and let the algorithm find a square root y of the integer (x^2 mod n); with probability 1/2, the greatest common divisor of $x - y$ and n is one of the two primes used to create n.

Unfortunately, the problem of factoring an integer is not known to be $\mathcal{NP}$-hard; it would be conceptually appealing to suggest schemes that are based on $\mathcal{NP}$-complete problems. Merkle and Hellman (1978) and since then several others, have suggested schemes based on the subset sum problem. The public key of such systems is an integer vector $a = (a_1, \ldots, a_n)$. A message M, that is a 0-1 vector of length n, is encrypted by the inner product $C = (a \cdot M)$. One problem with this scheme is that there does not appear to be a private key that can make the task of decryption easier for the intended receiver. Crypto-systems based on this idea have built some additional trapdoor information into the structure of the vector a, and so the decryption problem is based on a restricted variant of the subset sum problem, which is not known to be $\mathcal{NP}$-hard. Furthermore, randomly chosen subset sum problems can be easy to solve. In an innovative paper, Shamir (1982) used Lenstra's integer programming algorithm to break the Merkle-Hellman scheme; that is, he gave a polynomial-time decryption algorithm that does not need the secret key. Since then, several other subset sum-based schemes have been broken by clever use of the LLL basis reduction algorithm (see Chapter 19).

The schemes mentioned so far have encrypted messages of length n, for some given security parameter n. An apparent problem with this approach is that even if we prove that no polynomial-time algorithm can decrypt the messages, it still might be possible to deduce some relevant information about the message in polynomial time. To formalize what it means to exclude this, Goldwasser and Micali (1984) suggested the following framework for a randomized encryption procedure and its security. Suppose that there is a function $B: \{0,1\}^n \rightarrow \{0,1\}^m$ and a randomized polynomial-time algorithm that, for any image $B(x)$, produces a value y such that $B(y) = B(x)$, and y is randomly distributed among all such values; an m-bit message can be encrypted by running this algorithm. Intuitively, this encoding is secure if, for any two messages M_1 and M_2 and some pair of codes for them, E_1 and E_2, (i.e., $B(E_i) = M_i$, $i = 1, 2$) it is "impossible" to gain any advantage in guessing which message corresponds to each encryption. More precisely, the randomized encryption based on B is **secure** if, for each pair (M_1, M_2) any randomized polynomial-time algorithm that is used to guess, given (M_1, M_2) and an unknown random ordering of E_1 and E_2, which value comes from which argument, the probability that the algorithm answers correctly is $1/2 + O(n^{-c})$ for all but an n^{-c} fraction of the encryptions, for every $c > 0$. Note that if the encryption algorithm is a deterministic polynomial-time algorithm, then it is not secure, since one could simply apply the encryption algorithm to M_1 and M_2 to determine the correspondence.

Goldwasser and Micali proposed a way to achieve this level of security by encrypting messages bit by bit. How does one encode a single bit? A natural solution is to append a number of random bits to the one bit of information, and encrypt the resulting longer string. While this was later shown to be an effective approach, Goldwasser and Micali proposed a different randomized bit-encryption scheme based on a function $B: \{0,1\}^n \rightarrow \{0,1\}$

and proved its security, based on a number-theoretic assumption, namely on the assumed difficulty of the quadratic residuosity problem.

An integer x is a **quadratic residue** modulo n if $x = y^2 \pmod{n}$ for some integer y. The **quadratic residuosity problem** is to decide for given integers n and x whether x is a quadratic residue modulo n. If n is a prime then it is easy to recognize if x is a quadratic residue modulo n, e.g., by computing $x^{(n-1)/2}$ modulo n: this is 0 or 1 if and only if x is a quadratic residue.

The quadratic residuosity problem for composite moduli is more difficult. One has to be careful, however, since there is a polynomially checkable necessary condition that could help in certain cases. To formulate this, define the **Legendre symbol** for any prime n by

$$\left(\frac{x}{n}\right) = \begin{cases} 1, & \text{if } n \not| x \text{ and} \\ & x \text{ is a quadratic residue mod } n, \\ -1, & \text{if } n \not| x \text{ and} \\ & x \text{ is not a quadratic residue mod } n, \\ 0, & \text{if } n | x. \end{cases}$$

We have seen above that it is easy to compute the Legendre symbol (where n is a prime). The **Jacobi symbol** is a generalization of the Legendre symbol to composite n, but not in the obvious way: if $n = p_1 \cdots p_k$ (where the p_i are not necessarily distinct primes), then

$$\left(\frac{x}{n}\right) = \prod_{i=1}^{k} \left(\frac{x}{p_i}\right).$$

It cannot be seen from this definition, but the Jacobi symbol can be computed in polynomial time for any n.

Now if n and x are coprime integers then $\left(\frac{x}{n}\right) = 1$ is a necessary condition for x to be a quadratic residue modulo n, but it is not sufficient: if n is a product of two primes, then exactly half of the residue classes x with $\left(\frac{x}{n}\right) = 1$ are quadratic residues. The Goldwasser-Micali encryption scheme is based on the assumption that there is no efficient way to obtain further information on the quadratic residuosity of $x \bmod n$. It works as follows: a public key consists of an integer n that is the product of two large primes, and a quadratic non-residue y with $\left(\frac{y}{n}\right) = 1$. The bit 0 is encrypted by a random quadratic residue $r^2 \pmod{n}$, the bit 1 is encrypted by a random quadratic non-residue of the form $r^2 y \pmod{n}$. The task of distinguishing encryptions of 0's from encryptions of 1's is exactly the quadratic residuosity problem. Decryption is easy for the intended receiver, who knows the prime factorization of n.

Yao (1982) has extended this result by proving that a secure randomized bit-encryption scheme exists if a one-way function exists; in fact, from every one-way function one can extract a "secure bit". The following simple construction was given by Goldreich and Levin (1989). Let $f : \{0,1\}^* \to \{0,1\}^*$ be a length-preserving one-way function, where a function is **length-preserving** if it maps n-bit strings into n-bit strings, for all n. Define a Boolean function by $B(x) = f^{-1}(x') \cdot x''$, where, if $n = |x|$, then x' and x'' are the first and last $\lfloor n/2 \rfloor$ bits of x, and "$\cdot$" denotes inner product. (If we also want to decode this bit, we take a trapdoor function for f.)

Diffie and Hellman noticed that under a further assumption, a public-key crypto-system can also be used to solve the **signature problem**, where each participant wants a way to electronically sign its messages so that no one else can forge it, in the sense that each recipient can verify that the message must have been signed by the claimed sender. The assumption, which is not very restrictive, is that $\Psi(D, \cdot)$ is one-to-one, i.e., $\Phi(E, \Psi(D, M)) = M$ for each message M. In this case, the system can be used for signatures in the following way. When Bob sends a message M to Alice he can use his private decryption key D_B to append the "signature" $\Psi(D_B, M)$ after the message M. Given such a signature, Alice can use Bob's public key E_B to convince herself that the message came from Bob, exactly as she received it. The Rivest, Shamir and Adleman scheme has this additional property, and therefore can be used for signatures as well.

b. Pseudo-random numbers. When random numbers are used in algorithms and crypto-systems, it is essential that the random bits used are unbiased and independent. The speed or the reliability of the algorithm, and the security of the crypto-system depend on the quality of the random numbers used. Natural sources of randomness, such as coins or noise diodes, are fairly slow in generating random bits. On the other hand, for both of these applications, truly random bits could be replaced by any sequence of bits that no polynomial-time algorithm can distinguish from truly random bits. A **pseudo-random number generator** takes a **seed** x, a truly random string of length n, and expands it to a **pseudo-random string** y of length n^k for some constant k. A pseudo-random number generator can be subjected to certain statistical tests. It passes the **next bit test** if after seeing the first i bits of its output y for some $i < n^k$ no polynomial-time algorithm can predict the next bit with probability more than $1/2 + n^{-c}$ for any constant c.

Most computers have built-in pseudo-random number generators; one of the simplest ones is the linear congruential generator (where the seed consists of integers a, b, m and x_0, and the pseudo-random numbers are generated by the recurrence $x_{i+1} = ax_i + b \pmod{m}$). This is easily shown to fail the next bit test. Other, more sophisticated pseudo-random number generators can also be shown to output inappropriate sequences, by clever use of the LLL basis reduction algorithm. An example is the binary expansion of algebraic numbers (where the seed is the polynomial defining the algebraic number).

The first provably secure pseudo-random bit generator was developed by Blum and Micali (1984). They proved that pseudo-random bit generators exist, based on the following paradigm: there exist a polynomial-time computable permutation F of the set $\{0,1\}^n$ and a function $B : \{0,1\}^n \to \{0,1\}$, such that $B(x)$ yields a secure bit (as discussed above), but given $F^{-1}(x)$, $B(x)$ can be computed in polynomial time. Such an F is necessarily a one-way function; B is called the "hard core" of F. The Goldreich–Levin construction of a secure bit can be used to show that such a pair of functions exists if a one-way function exists, by taking $F(x) = f(x') \cdot x''$ for some length-preserving one-way function f.

The Blum-Micali pseudo-random number generator produces the sequence $b_0, \ldots, b_k$, defined by $b_i = B(x_{k-i})$, and $x_{i+1} = F(x_i)$, where the random seed is used to select the functions used and the initial vector x_0. The defined pseudo-random number generator can be proved to pass the next bit test. (Informally, suppose that we have an algorithm that can predict $b_{i+1} = B(x_{k-i-1})$, given $b_0, \ldots, b_i$; we will use this to contradict the fact that B yields a secure bit. Note that given just x_{k-i-1}, we can compute $x_{k-i}, \ldots, x_{k-1}$, and use these values, $F^{-1}(x_{k-i}), \ldots, F^{-1}(x_k)$, to compute the bits $b_0, \ldots, b_i$ in polynomial time; hence we can use the assumed procedure to predict b_{i+1}, which is impossible.)

One might wonder whether certain pseudo-random number generators pass statistical tests other than the next bit test. However, Yao (1982) proved that if a pseudo-random number generator passes the next bit test then it passes any statistical test, i.e., no randomized polynomial-time algorithm can distinguish the generated pseudo-random numbers from truly random numbers.

c. Zero-knowledge proofs.

Let us return to our example with the bank account access. The programmer of the computer of the bank may be tricky and store your password after you have used it once. Can you avoid using it at all and only *prove* to your bank that you have a password (i.e., know a Hamiltonian circuit in the graph G) without giving any help to find it (or mimicking you in any other way)?

This question also comes up in some of the above cryptographic applications: it might be useful to be able to convince someone that a number is the product of two primes, without telling the two primes themselves. This is impossible in the classical sense of proofs, but interactive proof systems, discussed in Chapter 29, make it possible. In fact, this was one of the motivating examples for Goldwasser, Micali and Rackoff (1985) when developing their notion of interactive proof systems. Informally, an interactive proof of a statement is said to be a zero-knowledge proof if the verifier cannot learn anything from the proof except the validity of the statement.

Before formalizing the notion of zero-knowledge proofs, let us describe a solution to the bank problem (due essentially to M. Blum). To make it more transparent, we imagine another setup: suppose that you are giving a talk on Hamiltonian graphs, and you show your audience a Hamiltonian graph G. For didactical purposes, you want to convince them that the graph G has a Hamiltonian circuit without showing them the circuit itself. This seemingly impossible task can be accomplished using an overhead projector. You prepare two transparencies: both show the same set $V(G)$ of nodes, in some random position; the first shows the edges of the Hamiltonian circuit C in G, the second, the remaining edges of G. On this second transparency, the labels of the nodes are also shown, but not on the first! You place both transparencies on the projector and cover them with a piece of paper, then switch on the projector and let the readers choose whether the top sheet or the top two sheets should be removed.

If only the top sheet is removed, the audience sees the graph G, politely labelled so that the audience can verify that it is indeed the graph G shown at the beginning. If both upper sheets are removed, the audience sees a Hamiltonian circuit on $|V(G)|$ randomly placed nodes in the plane. In either case, no information is given on how the Hamiltonian circuit lies in the graph G. (The only information the audience gets is that they see what they expect.)

On the other hand, if you want to cheat and show a graph G that is not Hamiltonian, then either your bottom transparency does not show a Hamiltonian circuit with the right number of nodes, or the two transparencies together do not show the right graph. So there is a chance of $1/2$ that you get caught! If you repeat this 100 times (a bit boring for a talk, but easily done on paper), and you don't get caught then the audience can be reasonably certain that G is Hamiltonian: your chance of getting away with a non-Hamiltonian graph is one in 2^{100}.

To make the above protocol precise, we have to get rid of the physical devices like projectors and transparencies; but this can be done using the methods of cryptography discussed above. The basic cryptographic tool needed for this is a secure bit-encryption scheme. You must be able to encrypt a bit, so that the audience has no chance to figure out on his own what the bit is, but later you can prove which bit was encrypted. To convince the audience that G has a Hamiltonian circuit, you choose a random permutation P, and use this permutation to obtain a random isomorphic copy G' of the graph G. You encode the permutation, and the $n(n-1)/2$ bits representing the adjacency matrix of the graph G'. The audience can choose either to ask for a proof that the encoded graph G' is isomorphic to the original, or to ask for a proof that G' has a Hamiltonian circuit. In the first case, you decrypt every encrypted bit, thereby providing the permutation P and the graph G'. In the second case, you decrypt only the bits corresponding to edges participating in the Hamiltonian circuit C.

There are several ways to formalize the notion of zero-knowledge proofs. The one we shall use is **computational** zero knowledge. We say that an interactive protocol is a **zero-knowledge protocol** if the verifier can generate, in randomized polynomial time, a sequence of communication whose distribution is indistinguishable in polynomial time from the distribution of the true transcript of the conversation.

In our example above, the audience could predict that if it chooses to see both transparencies together, it will see the given graph with nodes randomly drawn in the plane; while if it chooses to see the bottom transparency, then it will see a circuit with the right number of (unlabelled) nodes, again randomly drawn in the plane.

In contrast, consider the example of the interactive proof for the graph non-isomorphism problem (Chapter 29, Section 2). At first sight, this seems to be a zero-knowledge protocol, since, so long as the verifier does not deviate from the protocol, he always knows the prover's next move, and hence could generate the conversation himself. There are zero-knowledge protocols for the graph non-isomorphism problem, but this protocol is not, in fact, zero-knowledge, since the verifier can use it to test if a third graph is isomorphic to one of the two in the input (i.e., the verifier can gain extra information by deviating from the protocol). Goldreich, Micali and Wigderson (1986) and, subsequently but independently (under a somewhat stronger assumption), Brassard and Crépeau (1986) proved the following result.

Theorem 4.1. *If one-way functions exist, then every language in $\mathcal{NP}$ has a zero-knowledge interactive proof.*

Proof. To prove that all languages in $\mathcal{NP}$ have zero-knowledge proofs, one merely has to provide a zero-knowledge proof for a single $\mathcal{NP}$-complete problem. We have sketched such a protocol for the Hamiltonian circuit problem. ∎

Acknowledgments

We would like to thank Shafi Goldwasser and Avi Wigderson for their helpful suggestions and comments on an earlier draft. The research of the second author was supported in part by an NSF Presidential Young Investigator award CCR-89-96272 with matching support from IBM, UPS and Sun and by Air Force Contract AFOSR-86-0078. The research of the third author was supported in part by Air Force Contract AFOSR-86-0078 and by a Packard Research Fellowship. The second and third authors were both supported in part by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through NSF grant DMS-8920550.

References

M. Ajtai (1983), Σ_1^1-formulae on finite structures, *Ann. Pure Appl. Logic* **24**, 1–48.

A. V. Aho, J. E. Hopcroft and J. D. Ullman (1983), *Data Structures and Algorithms,* Addison-Wesley, Reading, MA.

A. V. Aho, J. D. Ullman and M. Yannakakis (1983), On notions of information transfer in VLSI circuits, *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 133–139.

N. Alon and R. B. Boppana (1987), The monotone circuit complexity of Boolean functions, *Combinatorica* **7**, 1–22.

A. E. Andreev (1985), On a method for obtaining lower bounds for the complexity of individual monotone functions (in Russian), *Dokl. Akad. Nauk. SSSR* **282**, 1033–1037. English transl. *Soviet Math. Dokl.* **31**, 530–534.

L. Babai, P. Frankl and J. Simon (1986), Complexity classes in communication complexity theory, *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Washington, D.C., 337–347.

L. Babai, N. Nisan and M. Szegedy (1989), Multiparty protocols and logspace-hard pseudorandom sequences, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 1–11.

M. Blum and S. Micali (1984), How to generate cryptographically strong sequences of pseudo-random bits, *SIAM J. Comput.* **13**, 850–864.

G. Brassard and C. Crépeau (1986), Non-transitive transfer of confidence: a perfect zero-knowledge protocol for SAT and beyond, *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Washington, D.C., 188–195.

B. Chor and O. Goldreich (1985), Unbiased bits from sources of weak randomness and probabilistic communication complexity, *Proceedings of the 26th Annual IEEE Sympo-*

sium on Foundations of Computer Science, IEEE Computer Society Press, Washington, D.C., 429–442.

W. Diffie and M.E. Hellman (1976), New directions in cryptography, *IEEE Trans. Inform. Theory* **22**, 644–654.

M. L. Fredman and R. E. Tarjan (1987), Fibonacci heaps and their uses in improved network optimization algorithms, *J. Assoc. Comput. Mach.* **34**, 596–615.

R. Freivalds (1979), Fast probabilistic algorithms, in: *Mathematical Foundations of Computer Science 1979* (ed. J. Bečvář), *Lecture Notes in Computer Science* **74**, Springer (Berlin), 57–69.

M. Furst, J.B. Saxe and M. Sipser (1981), Parity, circuits, and the polynomial time hierarchy, *Math. Systems Theory* **17**, 13–27.

O. Goldreich (1988), Randomness, interactive proofs, and zero-knowledge — a survey, in: *The Universal Turing Machine: A Half-Century Survey* (ed. R. Herken), Kammerer & Unverzagt, (Hamburg – Berlin), 377–405.

O. Goldreich and L.A. Levin (1989), A hard-core predicate for all one-way functions, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 25–32.

O. Goldreich, S. Micali and A. Wigderson (1986), Proofs that yield nothing but their validity and a methodology of cryptographic protocol design, *Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Washington, D.C., 174–187.

S. Goldwasser (1989), Interactive proof systems, in: *Computational Complexity Theory*, (ed. J. Hartmanis) AMS Symposia in Applied Mathematics **38**, AMS, (Providence), 108–128.

S. Goldwasser and S. Micali (1984), Probabilistic encryption, *J. Comput. System Sci.* **28**, 270–299.

S. Goldwasser, S. Micali and C. Rackoff (1989), The knowledge complexity of interactive proof systems, *SIAM J. Comput.* **18**, 186–208.

G. H. Gonnet (1984), *Handbook of Algorithms and Data Structures,* Addison-Wesley, London.

M. Grötschel, L. Lovász and A. Schrijver (1981), The ellipsoid method and its consequences in combinatorial optimization, *Combinatorica* **1**, 169–197.

A. Hajnal, W. Maass and G. Turán (1988), On the communication complexity of graph properties, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 186–191.

B. Halstenberg and R. Reischuk (1988), On different modes of communication, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 162–172.

J. Hastad (1986), Almost optimal lower bounds for small depth circuits, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 6–20.

M. Karchmer and A. Wigderson (1990), Monotone circuits for connectivity require super-logarithmic depth, *SIAM J. Discrete Math.* **3**, 255–265.

L. Levin (1986), Average case complete problems, *SIAM J. Comput.* **15**, 285–286.

B. Lindström (1969), Determinants on semilattices, *Proc. Amer. Math. Soc.* **20**, 207–208.

R. J. Lipton and R. Sedgewick (1981), Lower bounds for VLSI, *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 300–307.

L. Lovász (1990), Communication complexity: a survey, in: *Paths, Flows, and VLSI* (ed. B. Korte, L. Lovász, A. Schrijver), Algorithms and Combinatorics 8, Springer, ???–???.

L. Lovász and M. Saks (1988), Lattices, Möbius functions and communication complexity, *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Washington, D.C., 81–90.

L. Lovász and M. Saks (1990), Lattices, Möbius functions and communication complexity, ???

K. Mehlhorn and E. M. Schmidt (1982), Las Vegas is better than determinism in VLSI and distributed computing, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 330–337.

R. C. Merkle and M. Hellman (1978), Hiding information and signatures in trapdoor knapsacks, *IEEE Trans. Inform. Theory* **24**, 525–530.

V. R. Pratt (1975), The power of negative thinking in multiplying Boolean matrices, *SIAM J. Comput.* **4**, 326–330.

M. O. Rabin (1979), *Digitalized Signatures as Intractable as Factorization*, TR-212, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

A. A. Razborov (1985a), Lower bounds on the monotone circuit complexity of some Boolean functions (in Russian), *Dokl. Akad. Nauk SSSR* **281**, 798–801. English translation. *Soviet Math. Dokl.* **31**, 354–357.

A. A. Razborov (1985b), A lower bound on the monotone network complexity of the logical permanent (in Russian), *Math. Zametki 37* (1985), 887–900. English translation. *Mathematical Notes of the Academy of Sciences of the USSR* **37**, 485–493.

A. A. Razborov (1987), Lower bounds on the size of bounded depth circuits over a complete basis with logical addition (in Russian), *Math. Zametki* **41**, 598–607. English translation. *Mathematical Notes of the Academy of Sciences of the USSR* **41**, 333–338.

R. Rivest (1990), Cryptography, in: *The Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, edited by J. van Leeuwen, Elsevier, Amsterdam, 717–755.

R. Rivest, A. Shamir and Adleman (1978), A method for obtaining digital signatures and public-key cryptosystems, *Comm. ACM* **21**, 120–126.

A. Shamir (1982), A polynomial time algorithm for breaking the basic Merkle-Hellman cryptosystem, *Proceedings of the 23th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Washington, D.C., 145–152.

R. Smolensky (1987), Algebraic methods in the theory of lower bounds for Boolean circuit complexity, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 77–82.

V. Strassen (1969), Gaussian elimination is not optimal, *Numer. Math.* **13**, 354–356.

É. Tardos (1988), The gap between monotone and non-monotone circuit complexity is exponential, *Combinatorica* **8**, 141–142.

R. E. Tarjan (1975), Efficiency of a good but not linear set union algorithm, *J. Assoc. Comput. Mach.* **22**, 215–225.

R. E. Tarjan (1983), *Data Structures and Network Algorithms,* CBMS–NSF Regional Conf. Series in Applied Math. **44**, SIAM, Philadelphia.

H. S. Wilf (1968), Hadamard determinants, Möbius functions and the chromatic number of a graph, *Bull. Amer. Math. Soc.* **74**, 960–964.

M. Yannakakis (1988), Expressing combinatorial optimization problems by linear programs, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, ACM Press, New York, NY, 223–228.

A. C. Yao (1982), Theory and applications of trapdoor functions, *Proceedings of the 23th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Washington, D.C., 80–91.

A. C. Yao (1983), Lower bounds by probabilistic arguments, *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Washington, D.C., 420–428.

A. C.-C. Yao (1985), Separating the polynomial-time hierarchy by oracles, *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society Press, Washington, D.C., 1–10.