6. Problems, algorithms, and running time

6.1. Introduction

Probably most of the readers will have some intuitive idea about what is a problem and what is an algorithm, and what is meant by the running time of an algorithm. Although for the greater part of this course this intuition will be sufficient to understand the substance of the matter, in some cases it is important to formalize this intuition. This is particularly the case when we deal with concepts like NP and NP-complete.

The class of problems solvable in polynomial time is usually denoted by P. The class NP, that will be described more precisely below, is a class of problems that might be larger (and many people believe it *is* larger). It includes most combinatorial optimization problems, including all problems that are in P. That is: P⊆NP. In particular, NP does **not** mean: "non-polynomial time". The letters NP stand for "nondeterministic polynomial-time". The class NP consists, roughly speaking, of all those questions with the property that for any input that has a positive answer, there is a 'certificate' from which the correctness of this answer can be derived in polynomial time.

For instance, the question:

(1) 'Given a graph G, is G Hamiltonian?'

belongs to NP. If the answer is 'yes', we can convince anyone that this answer is correct by just giving a Hamiltonian circuit in G as a certificate. With this certificate, the answer 'yes' can be checked in polynomial time — in fact: trivially. Here it is not required that we are able to *find* the certificate in polynomial time. The only requirement is that there *exists* a certificate which can be checked in polynomial time.

Checking the certificate in polynomial time means: checking it in time bounded by a polynomial in the original input. In particular, it implies that the certificate itself has size bounded by a polynomial in the original input.

To elucidate the meaning of NP, it is not known if for any graph G for which question (1) has a *negative* answer, there is a certificate from which the correctness of this answer can be derived in polynomial time. So there is an easy way of convincing 'your boss' that a certain graph is Hamiltonian (just by exhibiting a Hamiltonian circuit), but no easy way is known for convincing this person that a certain graph is non-Hamiltonian.

Within the class NP there are the "NP-complete" problems. These are by definition the hardest problems in the class NP: a problem Π in NP is NP-*complete* if every problem in NP can be reduced to Π, in polynomial time. It implies that if one NP-complete problem can be proved to be solvable in polynomial time, then *each* problem in NP can be solved in polynomial time. In other words: then P=NP would follow.

Surprisingly, there are several prominent combinatorial optimization problems that are NP-complete, like the traveling salesman problem and the problem of finding a maximum clique in a graph. This pioneering eye-opener was given by Cook [1971] and Karp [1972].

Since that time one generally sets the polynomially solvable problems against the NP-complete problems, although there is no proof that these two concepts really are distinct. For almost every combinatorial optimization problem one has been able either to prove that it is solvable in polynomial time, or that it is NP-complete. But theoretically it is still a possibility that these two concepts are just the same! Thus it is unknown which of the two diagrams in Figure 6.1 applies.

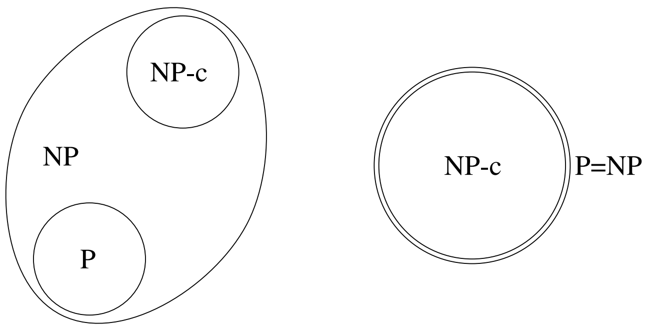

Figure 6.1

Below we make some of the notions more precise. We will not elaborate all technical details fully, but hope that the reader will be able to see the details with not too much effort. For precise discussions we refer to the books by Aho, Hopcroft, and Ullman [1974], Garey and Johnson [1979], and Papadimitriou [1994].

6.2. Words

If we use the computer to solve a certain graph problem, we usually do not put a picture of the graph in the computer. (We are not working with analog computers, but with digital computers.) Rather we put some appropriate encoding of the problem in the computer, by describing it by a sequence of symbols taken from some fixed finite 'alphabet' Σ. We can take for Σ for instance the ASCII set of symbols or the set $\{0, 1\}$. It is convenient to have symbols like (,) , { , } and the comma in Σ, and moreover some symbol like ␣ meaning: 'blank'. Let us fix one alphabet Σ.

We call any ordered finite sequence of elements from Σ a *word.* The set of all words is denoted by Σ^*.

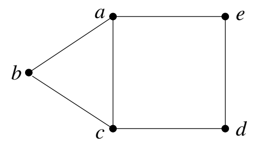

Figure 6.2

It is not difficult to encode objects like rational numbers, vectors, matrices, graphs, and so on, as words. For instance, the graph given in Figure 6.2 can be encoded, as usual, by the word:

(2) $$(\{a,b,c,d,e\},\{\{a,b\},\{a,c\},\{b,c\},\{c,d\},\{d,e\},\{e,a\}\}).$$

A function f defined on a finite set X can be encoded by giving the set of pairs $(x, f(x))$ with $x \in X$. For instance, the following describes a function defined on the edges of the graph above:

(3)
$$\{(\{a,b\},32),(\{a,c\},-17),(\{b,c\},5/7),(\{c,d\},6),(\{d,e\},-1),(\{e,a\},-9)\}.$$

A pair of a graph and a function can be described by the word (w,v), where w is the encoding of the graph and v is the encoding of the function.

The *size* of a word w is the number of symbols used in w, counting multiplicities. (So the word $abaa32bc$ has size 8.) The size is important when we make estimates on the running time of algorithms.

Note that in encoding numbers (integers or rational numbers), the size depends on the number of symbols necessary to encode these numbers. Thus if we encounter a problem on a graph with numbers defined on the edges, then the size of the input is the total number of bits necessary to represent this structure. It might be much larger than just the number of nodes and edges of the graph, and much smaller than the sum of all numbers occurring in the input.

Although there are several ways of choosing an alphabet and encoding objects by words over this alphabet, any way chosen is quite arbitrary. We will be dealing with solvability in polynomial time in this chapter, and for that purpose most encodings are equivalent. Below we will sometimes exploit this flexibility.

6.3. Problems

What is a problem? Informally, it is a question or a task, for instance, “Does this given graph have a perfect matching?” or “Find a shortest traveling salesman tour in this graph!”. In fact there are two types of problems: problems that can be answered by ‘yes’ or ‘no’ and those that ask you to find an object with certain prescribed properties. We here restrict ourselves to the first type of problems. From a complexity point of view this is not that much of a restriction. For instance, the problem of finding a shortest traveling salesman tour in a graph can be studied by the related problem: Given a graph, a length function on the edges, and a rational number r, does there exist a traveling salesman tour of length at most r? If we can answer this question in polynomial time, we can find the length of a shortest tour in polynomial time, for instance, by binary search.

So we study problems of the form: Given a certain object (or sequence of objects), does it have a certain property? For instance, given a graph G, does it have a perfect matching?

As we encode objects by words, a problem is nothing but: given a word w, does it have a certain property? Thus the problem is fully described by describing the “certain property”. This, in turn, is fully described by just the set of all words that have the property. Therefore we have the following mathematical definition: a *problem* is any subset Π of Σ^*.

If we consider any problem $\Pi \subseteq \Sigma^*$, the corresponding ‘informal’ problem is:

(4) Given word w, does w belong to Π?

In this context, the word w is called an *instance* or the *input*.

6.4. Algorithms and running time

An algorithm is a list of instructions to solve a problem. The classical mathematical formalization of an algorithm is the *Turing machine*. In this section we will describe a slightly different concept of an algorithm (the ‘Thue system’) that is useful for our purposes (explaining NP-completeness). In Section 6.10 below we will show that it is equivalent to the notion of a Turing machine.

A basic step in an algorithm is: replace subword u by u'. It means that if word w is equal to tuv, where t and v are words, we replace w by the word $tu'v$. Now by definition, an *algorithm* is a finite list of instructions of this type. It thus is fully described by a sequence

(5) $((u_1, u_1'), \ldots, (u_n, u_n'))$,

where $u_1, u_1', \ldots, u_n, u_n'$ are words. We say that word w' *follows from* word w if there exists a $j \in \{1, \dots, n\}$ such that $w = tu_jv$ and $w' = tu'_jv$ for certain words t and v, in such a way that j is the smallest index for which this is possible and the size of t is as small as possible. The algorithm *stops at* word w if w has no subword equal to one of $u_1, \dots, u_n$. So for any word w, either there is a unique word w' that follows from w, or the algorithm stops at w. A (finite or infinite) sequence of words $w_0, w_1, w_2, \dots$ is called *allowed* if each w_{i+1} follows from w_i and, if the sequence is finite, the algorithm stops at the last word of the sequence. So for each word w there is a unique allowed sequence starting with w. We say that A *accepts* w if this sequence is finite.

For reasons of consistency it is important to have the 'empty space' at both sides of a word as part of the word. Thus instead of starting with a word w, we start with $\text{\_}w\text{\_}$, where _ is a symbol indicating space.

Let A be an algorithm and let $\Pi \subseteq \Sigma^*$ be a problem. We say that A *solves* Π if Π equals the set of words accepted by A. Moreover, A solves Π *in polynomial-time* if there exists a polynomial $p(x)$ such that for any word $w \in \Sigma^*$: if A accepts w, then the allowed sequence starting with w contains at most $p(\text{size}(w))$ words.

This definition enables us indeed to decide in polynomial time if a given word w belongs to Π. We just take $w_0 := w$, and next, for $i = 0, 1, 2, \dots$, we choose 'the first' subword u_j in w_i and replace it by u'_j (for some $j \in \{1, \dots, n\}$) thus obtaining w_{i+1}. If within $p(\text{size}(w))$ iterations we stop, we know that w belongs to Π, and otherwise we know that w does not belong to Π.

Then P denotes the set of all problems that can be solved by a polynomial-time algorithm.

6.5. The class NP

We mentioned above that NP denotes the class of problems for which a positive answer has a 'certificate' from which the correctness of the positive answer can be derived in polynomial time. We will now make this more precise.

The class NP consists of those problems $\Pi \subseteq \Sigma^*$ for which there exist a problem $\Pi' \in$P and a polynomial $p(x)$ such that for any $w \in \Sigma^*$:

(6) $w \in \Pi$ if and only if there exists a word v such that $(w, v) \in \Pi'$ and such that $\text{size}(v) \leq p(\text{size}(w))$.

So the word v acts as a certificate showing that w belongs to Π. With the polynomial-time algorithm solving Π', the certificate proves in polynomial time that w belongs to Π.

As examples, the problems

(7) $\Pi_1 := \{G \mid G \text{ is a graph having a perfect matching}\}$ and
$\Pi_2 := \{G \mid G \text{ is a Hamiltonian graph}\}$

(encoding G as above) belong to NP, since the problems

(8) $\Pi'_1 := \{(G,M) \mid G$ is a graph and M is a perfect matching in $G\}$
and
$\Pi'_2 := \{(G,H) \mid G$ is a graph and H is a Hamiltonian circuit in $G\}$

belong to P.

Similarly, the problem

(9) TSP $:= \{(G,l,r) \mid G$ is a graph, l is a 'length' function on the edges of G and r is a rational number such that G has a Hamiltonian tour of length at most $r\}$

('the traveling salesman problem') belongs to NP, since the problem

(10) TSP$'$ $:= \{(G,l,r,H) \mid G$ is a graph, l is a 'length' function on the edges of G, r is a rational number, and H is a Hamiltonian tour in G of length at most $r\}$

belongs to P.

Clearly, P$\subseteq$NP, since if Π belongs to P, then we can just take the empty string as certificate for any word w to show that it belongs to Π. That is, we can take $\Pi' := \{(w,) \mid w \in \Pi\}$. As $\Pi \in$P, also $\Pi' \in$P.

The class NP is apparently much larger than the class P, and there might be not much reason to believe that the two classes are the same. But, as yet, nobody has been able to show that they really are different! This is an intriguing mathematical question, but besides, answering the question might also have practical significance. If P=NP can be shown, the proof might contain a revolutionary new algorithm, or alternatively, it might imply that the concept of 'polynomial-time' is completely useless. If P$\neq$NP can be shown, the proof might give us more insight in the reasons why certain problems are more difficult than other, and might guide us to detect and attack the kernel of the difficulties.

6.6. The class co-NP

By definition, a problem $\Pi \subseteq \Sigma^*$ belongs to the class co-NP if the 'complementary' problem $\overline{\Pi} := \Sigma^* \setminus \Pi$ belongs to NP.

For instance, the problem Π_1 defined in (7) belongs to co-NP, since the problem

(11) $\Pi''_1 := \{(G,W) \mid G$ is a graph and W is a subset of the vertex set of G such that the graph $G - W$ has more than $|W|$ odd components$\}$

belongs to P. This follows from Tutte's '1-factor theorem' (Corollary 5.1a): a graph G has no perfect matching if and only if there is a subset W of the vertex set of G with the properties described in (11). (Here, strictly speaking, the complementary problem $\overline{\Pi_1}$ of Π_1 consists of all words w that either do not represent a graph, or represent a graph having no perfect matching. We assume however that there is an easy way of deciding if a given word represents a graph. Therefore, we might assume that the complementary problem is just $\{G \mid G$ is a graph having no perfect matching$\}$.)

It is not known if the problems Π_2 and TSP belong to co-NP.

Since for any problem Π in P also the complementary problem $\overline{\Pi}$ belongs to P, we know that P⊆co-NP. So P⊆NP∩co-NP. The problems in NP∩co-NP are those for which there exist certificates both in case the answer is positive and in case the answer is negative. As we saw above, the perfect matching problem Π_1 is such a problem. Tutte's theorem gives us the certificates. Therefore, Tutte's theorem is called a *good characterization.*

In fact, there are very few problems known that are proved to belong to NP∩co-NP, but that are not known to belong to P. Most problems having a good characterization, have been proved to be solvable in polynomial time. The notable exception for which this is not yet proved is *primality testing* (testing if a given natural number is a prime number).

6.7. NP-completeness

The NP-complete problems are by definition the hardest problems in NP. To be more precise, we first define the concept of a polynomial-time reduction. Let Π and Π' be two problems and let A be an algorithm. We say that A is a *polynomial-time reduction* of Π' to Π if A is a polynomial-time algorithm ('solving' Σ^*), so that for any allowed sequence starting with w and ending with v one has: $w \in \Pi'$ if and only if $v \in \Pi$. A problem Π is called NP-*complete*, if $\Pi \in$NP and for each problem Π' in NP there exists a polynomial-time reduction of Π' to Π.

It is not difficult to see that if Π belongs to P and there exists a polynomial-time reduction of Π' to Π, then also Π' belongs to P. It implies that if one NP-complete problem can be solved in polynomial time, then each problem in NP can be solved in polynomial time. Moreover, if Π belongs to NP, Π' is NP-complete and there exists a polynomial-time reduction of Π' to Π, then also Π is NP-complete.

6.8. NP-completeness of the satisfiability problem

We now first show that in fact there exist NP-complete problems. In fact we show that the so-called *satisfiability problem*, denoted by SAT, is NP-complete.

To define SAT, we need the notion of a *boolean expression.* Examples are:

(12) $((x_2 \wedge x_3) \vee \neg(x_3 \vee x_5) \wedge x_2),\ ((\neg x_{47} \wedge x_2) \wedge x_{47}),\ \neg(x_7 \wedge \neg x_7).$

Boolean expressions can be defined inductively. First, for each natural number n, the 'word' x_n is a boolean expression (using some appropriate encoding of natural numbers and of subscripts). Next, if v and w are boolean expressions, then also $(v \wedge w)$, $(v \vee w)$ and $\neg v$ are boolean expressions. These rules give us all boolean expressions. (If necessary, we may use other subscripts than the natural numbers.)

Now SAT is a subcollection of all boolean expressions, namely it consists of those boolean expressions that are satisfiable. A boolean expression $f(x_1, x_2, x_3, \ldots)$ is called *satisfiable* if there exist $\alpha_1, \alpha_2, \alpha_3, \ldots \in \{0,1\}$ such that $f(\alpha_1, \alpha_2, \alpha_3, \ldots) = 1$, using the well-known identities

(13)
$$\begin{aligned} &0 \wedge 0 = 0 \wedge 1 = 1 \wedge 0 = 0, 1 \wedge 1 = 1,\\ &0 \vee 0 = 0, 0 \vee 1 = 1 \vee 0 = 1 \vee 1 = 1,\\ &\neg 0 = 1, \neg 1 = 0, (0) = 0, (1) = 1. \end{aligned}$$

Exercise. Let $n \geq 1$ be a natural number and let W be a collection of words in $\{0,1\}^*$ all of length n. Prove that there exists a boolean expression $f(x_1, \ldots, x_n)$ in the variables $x_1, \ldots, x_n$ such that for each word $w = \alpha_1 \ldots \alpha_n$ in the symbols 0 and 1 one has: $w \in W$ if and only if $f(\alpha_1, \ldots, \alpha_n) = 1$. ▮

The satisfiability problem SAT trivially belongs to NP: we can take as certificate for a certain $f(x_1, x_2, x_3, \ldots)$ to belong to SAT, the equations $x_i = \alpha_i$ that give f the value 1. (We only give those equations for which x_i occurs in f.)

To show that SAT is NP-complete, it is convenient to assume that $\Sigma = \{0,1\}$. This is not that much a restriction: we can fix some order of the symbols in Σ, and encode the first symbol by 10, the second one by 100, the third one by 1000, and so on. There is an easy (certainly polynomial-time) way of obtaining one encoding from the other.

The following result is basic for the further proofs:

Theorem 6.1. *Let $\Pi \subseteq \{0,1\}^*$ be in* P. *Then there exist a polynomial $p(x)$ and an algorithm that finds for each natural number n in time $p(n)$ a boolean expression $f(x_1, x_2, x_3, \ldots)$ with the property:*

(14) *any word $\alpha_1\alpha_2 \ldots \alpha_n$ in $\{0,1\}^*$ belongs to Π if and only if the boolean expression $f(\alpha_1, \ldots, \alpha_n, x_{n+1}, x_{n+2}, \ldots)$ is satisfiable.*

Proof. Since Π belongs to P, there exists a polynomial-time algorithm A solving Π. So there exists a polynomial $p(x)$ such that a word w belongs to Π if and only if the allowed sequence for w contains at most $p(\text{size}(w))$ words. It implies that there exists a polynomial $q(x)$ such that any word in the allowed sequence for w has size less than $q(\text{size}(w))$.

We describe the algorithm meant in the theorem. Choose a natural number n. Introduce variables $x_{i,j}$ and $y_{i,j}$ for $i = 0, 1, \ldots, p(n)$, $j = 1, \ldots, q(n)$. Now there exists (cf. the Exercise above) a boolean expression f in these variables with the following properties. Any assignment $x_{i,j} := \alpha_{i,j} \in \{0,1\}$ and $y_{i,j} := \beta_{i,j} \in \{0,1\}$ makes f equal to 1 if and only if the allowed sequence starting with the word $w_0 := \alpha_{0,1}\alpha_{0,2} \ldots \alpha_{0,n}$ is a finite sequence $w_0, \ldots, w_k$, so that:

(15) (i) $\alpha_{i,j}$ is equal to the jth symbol in the word w_i, for each $i \leq k$ and each $j \leq \text{size}(w_i)$;
(ii) $\beta_{i,j} = 1$ if and only if $i > k$ or $j \leq \text{size}(w_i)$.

The important point is that f can be found in time bounded by a polynomial in n. To see this, we can encode the fact that word w_{i+1} should follow from word w_i by a boolean expression in the 'variables' $x_{i,j}$ and $x_{i+1,j}$, representing the different positions in w_i and w_{i+1}. (The extra variables $y_{i,j}$ and $y_{i+1,j}$ are introduced to indicate the sizes of w_i and w_{i+1}.) Moreover, the fact that the algorithm stops at a word w also can be encoded by a boolean expression. Taking the 'conjunction' of all these boolean expressions, will give us the boolean expression f. ▮

As a direct consequence we have:

Corollary 6.1a. *Theorem* 6.1 *also holds if we replace* P *by* NP *in the first sentence.*

Proof. Let $\Pi \subseteq \{0,1\}^*$ belong to NP. Then, by definition of NP, there exists a problem Π' in P and a polynomial $r(x)$ such that any word w belongs to Π if and only if (w, v) belongs to Π' for some word v with $\text{size}(v) \leq r(\text{size}(w))$. By properly re-encoding, we may assume that for each $n \in \mathbb{N}$, any word $w \in \{0,1\}^*$ belongs to Π if and only if wv belongs to Π' for some word v of size $r(\text{size}(w))$. Applying Theorem 6.1 to Π' gives the corollary. ▮

Now the main result of Cook [1971] follows:

Corollary 6.1b (Cook's theorem)**.** *The satisfiability problem* SAT *is* NP*-complete.*

Proof. Let Π belong to NP. We describe a polynomial-time reduction of Π to SAT. Let $w = \alpha_1 \ldots \alpha_n \in \{0,1\}^*$. By Corollary 6.1a we can find in time bounded by a polynomial in n a boolean expression f such that w belongs to Π if and only if $f(\alpha_1, \ldots, \alpha_n, x_{n+1}, \ldots)$ is satisfiable. This is the required reduction to SAT. ▮

6.9. NP-completeness of some other problems

We next derive from Cook's theorem some of the results of Karp [1972]. First we show that the 3-*satisfiability problem* 3-SAT is NP-complete. Let B_1 be the set of all words $x_1, \neg x_1, x_2, \neg x_2, \ldots$. Let B_2 be the set of all words $(w_1 \vee \cdots \vee w_k)$, where $w_1, \cdots, w_k$ are words in B_1 and $1 \leq k \leq 3$. Let B_3 be the set of all words $w_1 \wedge \ldots \wedge w_k$, where $w_1, \ldots, w_k$ are words in B_2. Again, we say that a word $f(x_1, x_2, \ldots) \in B_3$ is *satisfiable* if there exists an assignment $x_i := \alpha_i \in \{0, 1\}$ $(i = 1, 2, \ldots)$ such that $f(\alpha_1, \alpha_2, \ldots) = 1$ (using the identities (13)).

Now the 3-satisfiability problem 3-SAT is: Given a word $f \in B_3$, decide if it is satisfiable.

Corollary 6.1c. *The 3-satisfiability problem* 3-SAT *is* NP-*complete.*

Proof. We give a polynomial-time reduction of SAT to 3-SAT. Let $f(x_1, x_2, \ldots)$ be a boolean expression. Introduce a variable y_g for each subword g of f that is a boolean expression.

Now f is satisfiable if and only if the following system is satisfiable:

$$(16) \qquad \begin{array}{ll} y_g = y_{g'} \vee y_{g''} & (\text{if } g = g' \vee g''), \\ y_g = y_{g'} \wedge y_{g''} & (\text{if } g = g' \wedge g''), \\ y_g = \neg y_{g'} & (\text{if } g = \neg g'), \\ y_f = 1. & \end{array}$$

Now $y_g = y_{g'} \vee y_{g''}$ can be equivalently expressed by: $y_g \vee \neg y_{g'} = 1, y_g \vee \neg y_{g''} = 1, \neg y_g \vee y_{g'} \vee y_{g''} = 1$. Similarly, $y_g = y_{g'} \wedge y_{g''}$ can be equivalently expressed by: $\neg y_g \vee y_{g'} = 1, \neg y_g \vee y_{g''} = 1, y_g \vee \neg y_{g'} \vee \neg y_{g''} = 1$. The expression $y_g = \neg y_{g'}$ is equivalent to: $y_g \vee y_{g'} = 1, \neg y_g \vee \neg y_{g'} = 1$.

By renaming variables, we thus obtain words $w_1, \ldots, w_k$ in B_2, so that f is satisfiable if and only if the word $w_1 \wedge \ldots \wedge w_k$ is satisfiable. ∎

We next derive that the *partition problem* PARTITION is NP-complete. This is the problem: Given a collection $\mathcal{C}$ of subsets of a finite set X, is there a subcollection of $\mathcal{C}$ that forms a partition of X?

Corollary 6.1d. *The partition problem* PARTITION *is* NP-*complete.*

Proof. We give a polynomial-time reduction of 3-SAT to PARTITION. Let $f = w_1 \wedge \ldots \wedge w_k$ be a word in B_3, where $w_1, \ldots, w_k$ are words in B_2. Let $x_1, \ldots, x_m$ be the variables occurring in f. Make a bipartite graph G with colour classes $\{w_1, \ldots, w_k\}$ and $\{x_1, \ldots, x_m\}$, by joining w_i and x_j by an edge if and only if x_j or $\neg x_j$ occurs in w_i. Let X be the set of all vertices and edges of G.

Let $\mathcal{C}'$ be the collection of all sets $\{w_i\} \cup E'$, where E' is a nonempty subset of the edge set incident with w_i. Let $\mathcal{C}''$ be the collection of all sets $\{x_j\} \cup E'_j$ and $\{x_j\} \cup E''_j$, where E'_j is the set of all edges $\{w_i, x_j\}$ so that x_j occurs in w_i and where E''_j is the set of all edges $\{w_i, x_j\}$ so that $\neg x_j$ occurs in w_i.

Now f is satisfiable if and only if the collection $\mathcal{C}' \cup \mathcal{C}''$ contains a subcollection that partitions X. Thus we have a reduction of 3-SAT to PARTITION. ▮

We derive the NP-completeness of the *directed Hamiltonian cycle problem* DIRECTED HAMILTONIAN CYCLE: Given a directed graph, does it have a directed Hamiltonian cycle?

Corollary 6.1e. DIRECTED HAMILTONIAN CYCLE *is* NP-*complete.*

Proof. We give a polynomial-time reduction of PARTITION to DIRECTED HAMILTONIAN CYCLE. Let $\mathcal{C} = \{C_1, \ldots, C_m\}$ be a collection of subsets of the set $X = \{x_1, \ldots, x_k\}$. Introduce 'vertices' $r_0, r_1, \ldots, r_m, s_0, s_1, \ldots, s_k$.

For each $i = 1, \ldots, m$ we do the following. Let $C_i = \{x_{j_1}, \ldots, x_{j_t}\}$. We construct a directed graph on the vertices r_{i-1}, r_i, s_{j_h-1}, s_{j_h} (for $h = 1, \ldots, t$) and $3t$ new vertices, as in Figure 6.3. Moreover, we make arcs from r_m to s_0 and from s_k to r_0.

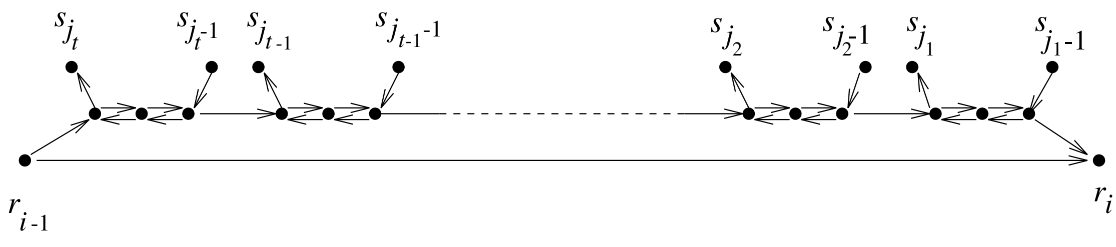

Figure 6.3

Let D be the directed graph arising. Then it is not difficult to check that there exists a subcollection $\mathcal{C}'$ of $\mathcal{C}$ that partitions X if and only if D has a directed Hamiltonian cycle C. (Take: $(r_{i-1}, r_i) \in C \Longleftrightarrow C_i \in \mathcal{C}'$.) ▮

From this we derive the NP-completeness of the *undirected Hamiltonian cycle problem* UNDIRECTED HAMILTONIAN CYCLE: Given a graph, does it have a Hamiltonian cycle?

Corollary 6.1f. UNDIRECTED HAMILTONIAN CYCLE *is* NP-*complete.*

Proof. We give a polynomial-time reduction of DIRECTED HAMILTONIAN CYCLE to UNDIRECTED HAMILTONIAN CYCLE. Let D be a directed graph. Replace each vertex v by three vertices v', v'', v''', and make edges $\{v', v''\}$ and $\{v'', v'''\}$. Moreover, for each arc (v_1, v_2) of D, make an edge $\{v'_1, v'''_2\}$. This makes the undirected graph G. One easily checks that D has a directed Hamiltonian cycle if and only if G has an (undirected) Hamiltonian cycle. ∎

This trivially implies the NP-completeness of the *traveling salesman problem* TSP: Given a complete graph $G = (V, E)$, a 'length' function l on E, and a rational r, does there exist a Hamiltonian cycle of length at most r?

Corollary 6.1g. *The traveling salesman problem* TSP *is* NP-*complete.*

Proof. We give a polynomial-time reduction of UNDIRECTED HAMILTONIAN CYCLE to TSP. Let G be a graph. Let G' be the complete graph on V. Let $l(e) := 0$ for each edge e of G and let $l(e) := 1$ for each edge of G' that is not an edge of G. Then G has a Hamiltonian cycle if and only if G' has a Hamiltonian cycle of length at most 0. ∎

6.10. Turing machines

In Section 6.4 we gave a definition of 'algorithm'. How adequate is this definition? Can any computer program be modelled after that definition?

To study this question, we need to know what we understand by a 'computer'. Turing [1937] gave the following computer model, now called a *Turing machine* or a *one-tape Turing machine*.

A Turing machine consists of a 'processor' that can be in a finite number of 'states' and of a 'tape', of infinite length (in two ways). Moreover, there is a 'read-write head', that can read symbols on the tape (one at a time). Depending on the state of the processor and the symbol read, the processor passes to another (or the same) state, the symbol on the tape is changed (or not) and the tape is moved one position 'to the right' or 'to the left'.

The whole system can be described by just giving the dependence mentioned in the previous sentence. So, mathematically, a *Turing machine* is just a function

(17) $$T : M \times \Sigma \to M \times \Sigma \times \{+1, -1\}.$$

Here M and Σ are finite sets: M is interpreted as the set of states of the processor, while Σ is the set of symbols that can be written on the tape. The function T describes an 'iteration': $T(m, \sigma) = (m', \sigma', +1)$ should mean that if the processor is in state m and the symbol read on the tape is σ, then the next state will be m', the symbol σ is changed to the symbol σ' and the tape is moved one position to the right. $T(m, \sigma) = (m', \sigma', -1)$ has a similar meaning — now the tape is moved one position to the left.

Thus if the processor is in state m and has the word $w'\alpha'\sigma\alpha''w''$ on the tape, where the symbol indicated by σ is read, and if $T(m, \sigma) = (m', \sigma', +1)$, then next the processor will be in state m' and has the word $w'\alpha'\sigma'\alpha''w''$ on the tape, where the symbol indicated by α'' is read. Similarly if $T(m,\sigma) = (m',\sigma',-1)$.

We assume that M contains a certain 'start state' 0 and a certain 'halting state' ∞. Moreover, Σ is assumed to contain a symbol _ meaning 'blank'. (This is necessary to identify the beginning and the end of a word on the tape.)

We say that the Turing machine T *accepts* a word $w \in (\Sigma \setminus \{\_\})^*$ if, when starting in state 0 and with word w on the tape (all other symbols being blank), so that the read-write head is reading the first symbol of w, then after a finite number of iterations, the processor is in the halting state ∞. (If w is the empty word, the symbol read initially is the blank symbol _.)

Let Π be the set of words accepted by T. So Π is a problem. We say that T *solves* Π. Moreover, we say that T *solves* Π *in polynomial time* if there exists a polynomial $p(x)$ such that if T accepts a word w, it accepts w in at most $p(\text{size}(w))$ iterations.

It is not difficult to see that the concept of algorithm defined in Section 6.4 above is at least as powerful as that of a Turing machine. We can encode any state of the computer model (processor+tape+read-write head) by a word (w', m, w''). Here m is the state of the processor and $w'w''$ is the word on the tape, while the first symbol of w'' is read. We define an algorithm A by:

(18) replace subword $,m,\sigma$ by σ',m', whenever $T(m,\sigma) = (m',\sigma',+1)$ and $m \neq \infty$;
replace subword α,m,σ by $m',\alpha\sigma'$, whenever $T(m,\sigma) = (m',\sigma',-1)$ and $m \neq \infty$.

To be precise, we should assume here that the symbols indicating the states in M do not belong to Σ. Moreover, we assume that the symbols (and) are not in Σ. Furthermore, to give the algorithm a start, it contains the tasks of replacing subword $\_\alpha$ by the word $(,0,\alpha$, and subword $\alpha\_$ by $\alpha)$ (for any α in $\Sigma \setminus \{\_\}$). Then, when starting with a word w, the first two iterations transform it to the word $(,0,w)$. After that, the rules (18) simulate the Turing machine iterations. The iterations stop as soon as we arrive at state ∞.

So T accepts a word w if and only if A accepts w — in (about) the same number of iterations. That is, T solves a problem Π (in polynomial time) if and only if A solves Π (in polynomial time).

This shows that the concept of 'algorithm' defined in Section 6.4 is at least as powerful as that of a Turing machine. Conversely, it is not hard (although technically somewhat complicated) to simulate an algorithm by a Turing machine. But how powerful is a Turing machine?

One could think of several objections against a Turing machine. It uses only one tape, that should serve both as an input tape, and as a memory, and as an output tape. We have only limited access to the information on the tape (we can shift only one position at a time). Moreover, the computer program seems to be implemented in the 'hardware' of the computer model; the Turing machine solves only one problem.

To counter these objections, several other computer models have been proposed that model a computer more realistically: multi-tape Turing machines, random access machines (RAM's), the universal Turing machine. However, from a polynomial-time algorithmic point of view, these models all turn out to be equivalent. Any problem that can be solved in polynomial time by any of these computer models, can also be solved in polynomial time by some one-tape Turing machine, and hence by an algorithm in the sense of Section 6.4. We refer to Aho, Hopcroft, and Ullman [1974] and Papadimitriou [1994] for an extensive discussion.