Chapter 6. NP-Completeness

The **Clique Problem** is as follows: The input is an undirected graph $G = (V, E)$ with n nodes and an integer k. The question is to decide whether the complete graph on k nodes is a subgraph of G, i.e., whether there is $V' \subseteq V$, $|V'| = k$ such that $(u, v) \in E$ for all $u, v \in V'$. There is a trivial but inefficient algorithm for solving the Clique Problem. Run through all subsets $V' \subseteq V$ of cardinality k and check whether V' induces a complete graph. There are $\binom{n}{k}$ sets V' with k elements. Thus our simple algorithm checks $\binom{n}{n/2} \geq 2^n/(n+1)$ subsets in the case $k = n/2$. It is simple to check a subset V' for the clique property; time $O(n^2)$ will certainly suffice, and one time unit is certainly required. We conclude that our naive algorithm has running time at least $2^n/(n+1)$. Before we can state that this is inefficient, we have to define the size of a problem.

Throughout this chapter, we assume that combinatorial objects, i.e., graphs, integers, sets, etc., are coded over a finite alphabet in some "natural way". More precisely, we assume that graphs are coded by their adjacency matrix, i.e., a graph G with n nodes is coded by a bitstring of length n^2, the entries of the matrix in row major order. Integers are always written in binary representation and sets are specified by listing their elements in some order.

In this way a problem instance of the Clique Problem is a bitstring of length $n^2 + \log(n/2) + 1$, where we again assume $k = n/2$ for simplicity. Using this convention our naive algorithm accepts the language

$$\begin{aligned} \textit{clique} = \{w\#v;\ w, v \in \{0,1\}^*, |w| = n^2 &\text{ for some } n \text{ and the graph } G \\ &\text{with adjacency matrix } w \text{ has a clique of size } k, \\ &\text{where } v \text{ is the binary representation of } k\} \end{aligned}$$

in time $\Omega(2^{\sqrt{m}}/\sqrt{m})$, where m is the length of the input. When we wrote down this chapter, no algorithm existed, which is considerably more efficient than the naive algorithm described above, and we will see in this chapter, that it is very unlikely that there will ever be one. We will also see that Clique shares this property with many other combinatorial problems, e.g., the Traveling Salesman Problem and the Satisfiability Problem of propositional logic.

Let us take a closer look at the Clique Problem. There are $\binom{n}{k}$ subsets $V' \subseteq V$ of cardinality k, i.e., for a clique of size k there are very many candidates. It is easy to test whether a subset $V' \subseteq V$ defines a clique, i.e., whether a candidate is indeed a solution. However, the only known way of finding a solution is to exhaustively search through all candidates.

This concept is best described by the notion of a **nondeterministic algorithm**. The instruction set of nondeterministic RAM's contains one more instruction, the nondeterministic choice instruction:

choice
    label$_1$, *label*$_2$.

The execution of a choice instruction transfers control to either $label_1$ or $label_2$. There are no probabilities associated with the alternatives as in randomized algorithms, rather the choice is made by a demon. In so far there is a large number of possible computations on any fixed input depending on the sequence of nondeterministic choices made by the demon. A nondeterministic algorithm **accepts** an input, if there is *at least one* accepting computation on that input (see Section 6.1 for an exact definition). The **time complexity** of a nondeterministic algorithm is the length of the shortest accepting computation.

We illustrate the new concept of a nondeterministic algorithm by describing a nondeterministic algorithm for Clique which runs in polynomial time. We use nondeterministic choices to select a candidate set V' and then check deterministically whether it is indeed a solution. More precisely, the algorithm works in three stages. In the first stage the input string $w\#v$ is parsed and $n = |w|^{1/2}$ and k, the number represented by bitstring v, are computed. If the input is not of the form $w\#v$ or if $|w|$ is not a square then the input is rejected. Stage 1 can certainly be done in polynomial time. In stage 2, Program 1 nondeterministically selects a subset $V' \subseteq V$ of size k by nondeterministically generating a bitvector $A[1..n]$ which contains exactly k ones, i.e., $A[i] = 1$ iff $i \in V'$.

```
for $i$ from 1 to $n$
do choice $m_1$, $m_2$;
    $m_1$: $A[i] \leftarrow 0$;
          goto $m$;
    $m_2$: $A[i] \leftarrow 1$;
          $k \leftarrow k - 1$;
    $m$:
od;
if $k \neq 0$ then stop and reject fi.
```

Program 1

The (successful) execution of Program 1 generates one of the $\binom{n}{k}$ subsets $V' \subseteq V$, $|V'| = k$. Stage 2 takes time $O(n)$. Stage 3 checks V' for the clique property. This can certainly be done in polynomial time also. If V' is a clique then the input is accepted, otherwise it is rejected. The algorithm described has time complexity polynomial in n and hence polynomial in m, the length of the input. Also, it accepts the language *clique*, as we now show. If the graph $G = (V, E)$ does not have a clique of size k, then the subset $V' \subseteq V$ generated in stage 2 does not form a clique and hence there is no accepting computation on input G. Conversely, if G has a clique of size k, say V', then there is a computation which generates that very V' in stage 2. This computation is accepting.

We have thus shown that the Clique Problem can be solved in polynomial time by a nondeterministic algorithm. Let *NP* be the class of problems which can be solved in polynomial time on a nondeterministic machine and let P be the class of problems which can be solved in polynomial time on a deterministic machine. One of the major results of this chapter is:

$$P = NP \qquad \text{iff} \qquad \text{Clique} \in P.$$

The class *NP* contains a large number of combinatorial problems (cf. Section 6.5) for which many researchers have intensively been trying to find efficient algorithms but none has been discovered so far. If Clique were in P then there would be efficient deterministic algorithms for all these problems which is unlikely. Thus Clique $\notin P$ is a safe conjecture.

In the sequel we will frequently come across the class of all algorithms of a certain complexity. In principle, we could base the discussion on the RAM model. However, the discussion will be easier if we use a simpler machine model: Turing Machines (TM). The relation between TM's and real computers is less direct than between RAM's and real computers and therefore complexity bounds derived for TM's are not directly applicable to real computers. However, the loss in efficiency is bounded by a polynomial (Theorem 1 of Section 6.1) and therefore the classes P and NP will be the same for both models.

6.1. Turing Machines and Random Access Machines

The TM is a very simple model of a universal computer. There are only two parts: a control unit and a storage unit. The storage unit is a single semi-infinite (infinite to the right) tape. The tape is divided into squares which can store a single character of a finite alphabet each. The control unit has a finite number of states, it scans the tape by a single read/write-head which is one square in size.

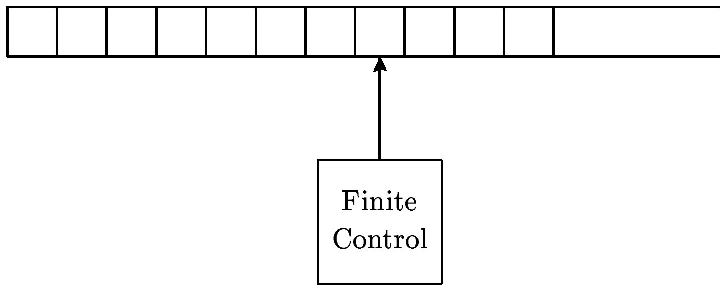

Figure 1. Visualization of a Turing Machine

We program TM's by Turing tables. For each state of the finite control and each letter of the tape alphabet the Turing table specifies a set of possible actions (in the case of a nondeterministic machine) or a single action (in the case of a deterministic machine). An action consists of three parts: changing the state of the finite control, printing a new symbol on the square under the read/write-head and moving the head by at most one square to the left or to the right. A TM is started by writing the input string of length n onto the first n squares of the tape, placing the head over the first (leftmost) symbol of the input string and putting the machine into a special state, the initial state. All other squares are initially empty, i.e., contain the symbol $\not{b}$ of the tape alphabet. The TM proceeds as specified by the Turing table until it reaches a pair of state and character scanned by the read/write-head for which no action is specified in the Turing table. At this point the machine stops. The output of the computation is the non-empty part of the tape. In this way we can use TM's to compute functions.

Often we use TMs to recognize languages. In this connection two definitions are possible: a TM recognizes a language if it computes the characteristic function, or we designate a subset of the states as accepting and call a computation accepting if it ends in an accepting state. Both definitions are equivalent; the second one allows for a more natural treatment of nondeterminism. The details are as follows.

Definition: A **nondeterministic TM** is a 4-tuple $M = (Z, \Gamma, \delta, F)$. Here $Z = \{Z_1, \ldots, Z_s\}$ is a finite set of states, $\Gamma = \{A_1, \ldots, A_v\}$ is a finite tape alphabet, Z_1 the initial state, $F \subseteq Z$ a set of accepting states and $\delta : Z \times \Gamma \to 2^{Z\times\Gamma\times\{-1,0,+1\}}$ a transition function. We assume $\delta(z, a) = \emptyset$ for all $z \in F$ and $a \in \Gamma$. The TM M is **deterministic** if $|\delta(z, a)| \leq 1$ for every state z and alphabet symbol a. Then δ is equivalent to a partial function $\delta : Z \times \Gamma \to Z \times \Gamma \times \{-1, 0, +1\}$ in a natural way.

We use A_1 to denote the empty tape square and write alternately A_1 or $\not{b}$. A **configuration** C is a string in $\Gamma^*(Z \times \Gamma)\Gamma^*$ and describes a snapshot of a computation. If $w(z, a)v$ is a configuration then wav is the tape content, z is the state of the finite control and the head scans symbol a. Let $C_1 = w_1(z_1, a_1)v_1$ and $C_2 = w_2(z_2, a_2)v_2$ be configurations. C_2 is a **successor configuration** of C_1, in symbols $C_1 \vdash C_2$, if:

a) $(z_2, b, -1) \in \delta(z_1, a_1)$ and $w_1 = w_2a_2$ and $v_2 = bv_1$ or

b) $(z_2, a_2, 0) \in \delta(z_1, a_1)$ and $w_1 = w_2$ and $v_1 = v_2$ or

c) $(z_2, b, +1) \in \delta(z_1, a_1)$ and $w_2 = w_1b$ and ($v_1 = a_2v_2$ or $v_1 = v_2 = \epsilon$ and $a_2 = \not{b}$).

$\vdash^*$ denotes the reflexive, transitive closure of relation $\vdash$. A configuration $C = w(z, a)v$ is a **halting configuration**, if $\delta(z, a) = \emptyset$, it is an **accepting configuration**, if $z \in F$, and it is an **initial configuration** with respect to string $x = a_1a_2 \ldots a_n$, if $z = Z_1$, $w = \epsilon$, $a = a_1$ and $v = a_2 \ldots a_n$.

A **computation** on input $x \in \Gamma^*$ is a sequence $C_0, C_1, \ldots C_k$ of configurations with $C_i \vdash C_{i+1}$ for $0 \leq i < k$ and C_0 an initial configuration with respect to x. A computation is accepting (halting), if C_k is an accepting (halting) configuration. The length of the computation is k. ∎

We are now ready to define the language accepted by a TM and the time complexity of the machine.

Definition: Let $M = (Z, \Gamma, \delta, F)$ be a TM.

a) $L(M) = \{x \in \Gamma^*;$ there is an accepting computation of M on input $x\}$ is the **language accepted** by M.

b) Let M be deterministic and total, i.e., there is a halting computation of M on every $x \in \Gamma^*$. Then M **computes function** $f_M : \Gamma^* \to \Gamma^*$ if for every $x \in \Gamma^*$ the tape content of the halting configuration of the computation of M on input x is $f_M(x)$.

c) The **time complexity** T_M of M is

$$T_M(n) = \max_{\substack{x \in L(M) \\ |x|=n}} \min\{k;\ k \text{ is the length of an accepting computation on input } x\}$$

if M is nondeterministic, and

$$T_M(n) = \max_{\substack{x \in \Gamma^* \\ |x|=n}} \{k;\ k \text{ is the length of the halting computation on input } x\}$$

if M is deterministic. Furthermore, $T_M(n) = \infty$ if there is $x \in \Gamma^*$, $|x| = n$, such that M does not halt on input x. ∎

One word of care is needed at this point. Note that in the definition of running time of nondeterministic machines the maximum is only taken over the strings in the accepted language. For each such string the shortest accepting computation is considered. In the case of deterministic machines the maximum is taken over all inputs of a certain length.

Example: A deterministic TM of time complexity $T(n) = O(n^2)$ which accepts $L = \{w\not{c}w;\ w \in \{0,1\}^*\}$. Let $Z = \{Z_1, Z_2, \ldots, Z_9\}$ be the set of states, let $\Gamma = \{0, 1, \bar{0}, \bar{1}, \not{c}, \not{b}\}$ be the tape alphabet and let $F = \{Z_9\}$ be the set of final states. The transition function δ is defined by the table of Figure 2.

The accepting computation on input $01\not{c}01$ is

$$\begin{aligned}(Z_1,0)1\not{c}01 &\vdash \bar{0}(Z_2,1)\not{c}01 \vdash \bar{0}1(Z_2,\not{c})01 \vdash \bar{0}1\not{c}(Z_4,0)1 \vdash \bar{0}1\not{c}(Z_6,\bar{0})1 \\ &\vdash \bar{0}1(Z_6,\not{c})\bar{0}1 \vdash \bar{0}(Z_7,1)\not{c}\bar{0}1 \vdash (Z_7,\bar{0})1\not{c}\bar{0}1 \vdash 0(Z_1,1)\not{c}\bar{0}1 \\ &\vdash 0\bar{1}(Z_3,\not{c})\bar{0}1 \vdash^* 01\not{c}\bar{0}(Z_8,\bar{1}) \vdash 01\not{c}\bar{0}\bar{1}(Z_8,\not{b}) \vdash 01\not{c}\bar{0}\bar{1}(Z_9,\not{b}).\end{aligned}$$

This machine compares the words before and after the $\not{c}$-sign character by character. It reads a character of the first word in state Z_1, stores the character in the finite control by changing to either state Z_2 or Z_3 and marks the character by baring it. It then moves to the right until it hits the $\not{c}$, records that fact in the finite control by changing its state either to Z_4 (from Z_2) or to Z_5 (from Z_3). It continues moving to the right until if finds an unbarred symbol. At this point the symbol under the reading head is compared with the symbol stored in the finite control. In the case of inequality, i.e., if a 1 is read in state Z_4 or a 0 in state Z_5 the machine halts. In the case of equality the machine moves the head back until an unbarred symbol in the first word is met and enters a new cycle. ∎

δ	0	1	$\bar{0}$	$\bar{1}$	$¢$	$\not b$
Z_1	$Z_2, \bar{0}, +1$	$Z_3, \bar{1}, +1$			$Z_8, ¢, +1$	
Z_2	$Z_2, 0, +1$	$Z_2, 1, +1$			$Z_4, ¢, +1$	
Z_3	$Z_3, 0, +1$	$Z_3, 1, +1$			$Z_5, ¢, +1$	
Z_4	$Z_6, \bar{0}, 0$		$Z_4, \bar{0}, +1$	$Z_4, \bar{1}, +1$		
Z_5		$Z_6, \bar{1}, 0$	$Z_5, \bar{0}, +1$	$Z_5, \bar{1}, +1$		
Z_6			$Z_6, \bar{0}, -1$	$Z_6, \bar{1}, -1$	$Z_7, ¢, -1$	
Z_7	$Z_7, 0, -1$	$Z_7, 1, -1$	$Z_1, 0, +1$	$Z_1, 1, +1$		
Z_8			$Z_8, \bar{0}, +1$	$Z_8, \bar{1}, +1$		$Z_9, \not b, 0$
Z_9						

Figure 2. Transition table for $L = \{w¢w;\ w \in \{0,1\}^*\}$

We now come to the central definition of this chapter. We now come to the central definition of this chapter.

Definition: (Classes P and NP)

$P = \{L;\ L \subseteq \Gamma^*$ for some finite Γ and there is adeterministic TM M and a polynomial p such that $L = L(M)$ and $T_M(n) \leq p(n)$ for all $n\}$

$NP = \{L;\ L \subseteq \Gamma^*$ for some finite Γ and there is a nondeterministic TM M and a polynomial p such that $L = L(M)$ and $T_M(n) \leq p(n)$ for all $n\}$ ∎

Since every deterministic TM is a nondeterministic TM and since the definition of running time is more strict in the deterministic case, we have the inclusion

$$P \subseteq NP.$$

The famous open problem is: "Is P equal to NP?" The answer to this problem is open; however, there are many reasons for believing that $P \neq NP$:

a) There are many problems, e.g., the CliqueProblem, such that

$$P = NP \quad \text{iff} \quad \text{Clique} \in P.$$

These problems arise from many different areas, e.g., graph theory, operations research, game theory and number theory. In all these areas a lot of work went into the development of efficient algorithms for these problems. No provable good algorithm was found.

b) Even worse, many algorithms which were suggested for these problems are known to have non-polynomial running time.

We used Turing machines to define classes P and NP. Theorem 1 below shows that we might have used RAM's under the logarithmic cost measure just as well. We will formally prove this only for languages over the binary alphabet $\{0,1\}$; a generalization to non-binary alphabets is trivial and left to the reader. RAM's were defined in Section 1.1; in particular, the I/O-behaviour of RAM's was defined at the end of Section 1.1. Let R be a RAM and let $x = a_1 a_2 \dots a_n \in \{0,1\}^*$, $a_i \in \{0,1\}$. Then the i-th execution of the statement "$\alpha \leftarrow$ input" places $a_i \in \{0,1\} \subseteq \mathbb{N}$ into the accumulator. We say that RAM R **accepts** $L \subseteq \{0,1\}^*$ iff it computes the characteristic function of L.

Theorem 1. *Let R be a RAM with time complexity $T_R(n)$ under the logarithmic cost measure and let L be the language accepted by R. Then there is a TM M which accepts L in time $T_M(n) = O\big((T_R(n))^5\big)$.*

Proof: (Sketch) We start with a description of the tape of TM M. The tape of M is divided into 8 tracks, i.e., the tape alphabet is Γ^8 for some alphabet Γ. Each single square can contain an element of Γ on each track.

input	$a_1\ a_2\ a_3\ a_4\ \dots$
memory	address # content # # address # content # # ...
accumulator	
index reg 1	
index reg 2	
index reg 3	
address reg	
scratch reg	

The first track contains the input $a_1 a_2 a_3 \dots$. Inputs which were read by the RAM are marked. The second track contains the memory of the RAM, more precisely, the second track contains a sequence of pairs $(ad_i, cont_i)$, $1 \le i \le m$; ad_i and $cont_i$ are integers written in binary. We maintain the following invariant. Let $ad \in \mathbb{N}$ be an address and let $cont$ be the content of the memory cell of the RAM with address ad. Then there is either no i with $ad = ad_i$ and then $cont = 0$ or there is an i with $ad = ad_i$ and then $cont = cont_j$ where $j = \max\{i;\ ad = ad_i\}$. Tracks 3 to 6 are used to store the accumulator and the index registers in binary. Finally tracks 7 and 8 are used for intermediate calculations. The RAM-program is stored in the finite control of TM M. We sketch the simulation in the case of instruction "$\alpha \leftarrow \alpha + \rho(3 + \gamma_2)$".

(1) Add 3 to the content of γ_2 (as stored on track 5) and store the result on track 7, the address register.

(2) Run through the pairs on track 2 and find the largest i with $ad_i = ad$, the content of track 7. The search is performed by an algorithm similar to the one described in the example above.

(3) If no i is found then place 0 onto the scratch register (track 8), otherwise copy $cont_i$ onto the scratch register. The details are very similar to step 2).

(4) Add the content of the scratch register to the accumulator.

All other instructions are simulated analogously. In particular, whenever a store instruction is executed, say $cont$ is stored in cell ad, then a new pair $(ad, cont)$ is added to the list on track 2.

It is easy to program the arithmetic operations in steps (1) and (4) on a TM. The running time is proportional to the length of the binary representation of the operands. Since we use the logarithmic cost measure this is in turn bounded by the cost of the simulated RAM-instruction.

Steps (2) and (3) are slightly more difficult to handle. The example above shows that the cost of steps 2) and 3) is bounded by $O(($length of the inscription of track $2)^2) = O(((\text{number of store instructions executed}) \cdot (\log(\text{maximal content of any cell}) + \log(\text{maximal address used})))^2)$.

The number of store instruction is certainly bounded by $T_R(n)$. The content of any cell and hence the maximal address is also bounded by $2^{T_R(n)}$ since the logarithmic cost measure charges one time unit for writing a bit. Hence the cost of steps (2) and (3) is bounded by $O\big((T_R(n))^4\big)$.

Let us summarize: TM M can simulate a proper RAM-instruction (arithmetic and test) in time $O\big((T_R(n))^2\big)$; the square is required for simulating divisions and multiplications. Furthermore, $O\big((T_R(n))^4\big)$ steps are needed to simulate storage access of the RAM. Thus $O\big((T_R(n))^5\big)$ steps suffice for the simulation of a $T_R(n)$ time bounded RAM. ∎

The proof of the converse of Theorem 1 is much simpler and is left to the reader. If L is accepted in time $T_M(n)$ by a TM then L is accepted in time $T_M(n) \log T_M(n)$ by a RAM under the logarithmic cost measure. Thus the complexities of a problem on a RAM and on a TM are polynomially related. A problem is solvable on a RAM under the logarithmic cost measure in polynomial time iff it is solvable on a TM in polynomial time. In other words, the definition of P is fairly robust with respect to the particular definition of machine chosen.

We end this section by relating deterministic and nondeterministic complexity. A fairly direct simulation yields an exponential loss in efficiency; no more efficient simulation is known.

Definition: A function $T : \mathbb{N} \to \mathbb{N}$ is a **step function**, if there is a deterministic TM M which stops after exactly $T(n)$ steps on every input of length n. ∎

Most common functions are step functions, e.g., $T(n) = n$, $T(n) = n\lfloor \log n \rfloor$, $T(n) = n^2$, $T(n) = 2^n$.

Theorem 2. *(Deterministic Simulation of Nondeterministic Turing Machines) Let $T(n)$ be a step function, let $L \subseteq \Gamma^*$ be a language and let N be a nondeterministic TM which accepts L in time $T_N(n) = O(T(n))$. Then there is a deterministic TM M with $L(M) = L$ and $T_M(n) = O(c^{T(n)})$ for some constant c.*

Proof: For every $x \in L$ there is an accepting computation of length $\leq T_N(|x|)$. Let $k = \max\{|\delta(z,a)|;\ z \text{ is a state of } N \text{ and } a \in \Gamma\}$. Then machine N has the choice between at most k different actions in every step. Thus there are at most $k^{T_N(|x|)}$ different computations of length $\leq T_N(|x|)$ on input x. Also $x \in L$ iff one of these computations is accepting. This suggests the following deterministic simulation of N. Let m be such that $T_N(n) \leq m \cdot T(n)$ for every n. Count from 0 to $k^{m \cdot T(n)} - 1$ in base k. Every number l between 0 and $k^{m \cdot T(n)} - 1$ represents a possible computation. More precisely, the i-th digit in the k-nary representation of l determines the action in the i-th move of N. It is easy to see that every fixed computation of N may be simulated in time $p(m \cdot T(|x|))$ where p is some polynomial. Thus the entire simulation takes time $p(m \cdot T(|x|)) \cdot k^{m \cdot T(|x|)} = O(c^{T(|x|)})$ for some constant c. ∎

6.2. Problems, Languages and Optimization Problems

Classes P and NP are sets of languages. However, problems are usually not defined as language recognition problems. Let us for example consider the **Traveling Salesman Problem** (TSP).

Name: Traveling Salesman Optimization Problem (TSOP).
Input: A distance matrix $dist : [0\,..\,n-1]^2 \to \mathbb{N}$.
Output: A permutation Π of $[0\,..\,n-1]$ which minimizes

$$\sum_{i=0}^{n-1} dist(\Pi(i), \Pi(i+1 \bmod n)),$$

i.e., a tour through n cities $0, 1, \ldots, n-1$ of minimal total length. ∎

The Traveling Salesman Optimization Problem is definitely not a language. It rather requires the computation of a function from n by n matrices *dist* to permutations of n elements. However, we can associate a language recognition problem with the optimization problem in a somewhat artificial way.

Name: Traveling Salesman Recognition Problem (TSRP).
Input: A distance matrix $dist : [0\,..\,n-1]^2 \to \mathbb{N}$ and an integer D.
Question: Is there a tour of length at most D, i.e., is there a permutation Π of $[0\,..\,n-1]$ such that $\sum_{i=0}^{n-1} dist(\Pi(i), \Pi(i+1 \bmod n)) \leq D$? ∎

The Traveling Salesman Recognition Problem gives rise to a language in a natural way; namely the language (set) of all problem instances $I = (\mathit{dist}, D)$ of TSRP with positive answer, i.e.,

$$L_{TSP} = \{(\mathit{dist}, D);\ \mathit{dist} \text{ has a tour of length } \leq D\}.$$

Actually, we have to be more precise. A language is a subset of Γ^* for some finite alphabet. Thus instead of talking about problem instances (dist, D) we should rather talk about the encodings of distance matrices *dist* and integers D over some finite alphabet Σ. There are many possible encodings to choose from. We use the following encodings:

1) Integers are written in binary representation.
2) Sequences (sets, matrices) are specified by listing the (encodings of their) elements separated by some special symbol.
3) Graphs are specified by their adjacency matrix; labelled graphs are specified by the matrix of labellings.

What do we have achieved now? We associated a language with the Traveling Salesman Recognition Problem and we can now ask the question whether this language is in P. Even if it were, what would that mean for our original problem, the Traveling Salesman Optimization Problem? In this section we show, that the function which maps distance matrices to optimal solutions would be computable in polynomial time also. So we have not really lost anything by moving from the optimization problem to the recognition problem, at least as far as polynomial time computability is concerned. This observation is not only true for the TSP but also in a more general sense. We start with an alternative characterization of NP.

Theorem 1.
$NP = \{L;\ L \subseteq \Gamma^*$ *for some finite* Γ *and there is a polynomial* p *and a polynomial time computable predicate* $Q \subseteq \Gamma^* \times \Gamma^*$ *such that for all* $x \in \Gamma^* : x \in L$ *iff* $\exists y \in \Gamma^* : |y| \leq p(|x|) \wedge Q(x, y)\}$

Proof: "$\supseteq$": Let p be a polynomial and let Q be a polynomial time computable predicate, i.e., there is a deterministic TM M which computes $Q(x, y)$ in time $q(|x|, |y|)$ for some polynomial q. The following nondeterministic algorithm accepts L.

(1) On input x generate nondeterministically a string $y \in \Gamma^*$, $|y| \leq p(|x|)$.
(2) Accept x if $Q(x, y)$. $Q(x, y)$ is computed using machine M.

The details of step (1) are very similar to the example given in the introduction to this chapter and therefore step (1) takes time at most $c \cdot p(|x|)$ for some constant c on a RAM and hence by Theorem 1 of Section 6.1 time polynomial in $|x|$ on a TM. The cost of step (2) is also bounded by a polynomial in $|x|$ and $|y|$, which is in turn bounded by a polynomial in $|x|$. Thus $L \in NP$.

"$\subseteq$": Let $L \in NP$, $L \subseteq \Gamma^*$. Let N be a nondeterministic TM which accepts L in time bounded by a polynomial p. As in the proof of Theorem 2 in Section 6.1 let k be the maximal number of actions N can choose from at any step. Then the proof of that theorem shows that the strings of length $\leq p(|x|)$ over a k-ary alphabet encode all possible computations of N of length $\leq p(|x|)$ on input x. Assume w.l.o.g. that $|\Gamma| \geq k$. Take

$$Q(x,y) = \text{“}y \text{ encodes an accepting computation of } N \text{ on input } x\text{”}.$$

Then Q is certainly computable in polynomial time and $L = \{x;\ \exists y : |y| \leq p(|x|) \wedge Q(x,y)\}$. ▮

The characterization of *NP* given in Theorem 1 is interesting as such. No explicit mention is made of nondeterminism. Rather, nondeterminism is concealed in the existential quantifier. More mathematically oriented readers may find it helpful to use Theorem 1 as the definition of *NP*.

Definition: A **minimization problem** is given by a polynomial time computable predicate $Q \subseteq \Gamma^* \times \Gamma^*$ and a polynomial time computable cost function $c : \Gamma^* \times \Gamma^* \to \mathbb{N}$. If $Q(x,y)$ then y is a **feasible solution** for problem instance x with cost $c(x,y)$. If $Q(x,y)$ and $c(x,y) \leq c(x,y')$ for all y' with $Q(x,y')$ then y is an **optimal solution** for x. The minimization problem is **polynomially bounded** if there is a polynomial p such that $Q(x,y)$ implies $|y| \leq p(|x|)$. ▮

We only deal with polynomially bounded optimization problems in this book. A similar definition is possible for maximization problems. In the TSP example we have: $Q(x,y)$ if x is the encoding of a distance matrix $dist : [0\,..\,n-1]^2 \to \mathbb{N}$ and y is the encoding of a permutation Π of $[0\,..\,n-1]$, and $c(x,y) = \sum dist(\Pi(i), \Pi(i+1 \bmod n))$.

Definition: Let (Q,c) be a polynomially bounded minimization problem. We define four versions of that problem.

a) Name: (Q,c)-recognition problem.
   Input: Instance $x \in \Gamma^*$, integer C.
   Question: Is there a y with $Q(x,y)$ and $c(x,y) \leq C$?

b) Name: (Q,c)-optimization problem.
   Input: Instance $x \in \Gamma^*$.
   Output: An optimal solution $optsol(x) \in \Gamma^*$ for x.

c) Name: (Q,c)-optimal value problem.
   Input: Instance $x \in \Gamma^*$.
   Output: $optval(x) = c(x, optsol(x))$.

d) Name: (Q,c)-witness problem.
   Input: Instance $x \in \Gamma^*$ and integer C.

Output: $\quad$ *witness*$(x, C) = y$ with $Q(x, y)$ and $c(x, y) \le C$, if there is any such y.▮

We have already seen the first two versions in the case of TSP. The recognition problem poses the question whether a tour of some given length exists and the optimization problem requires that we produce an optimal tour. The optimal value problem requires that we compute the length of the optimal tour and the witness problem requires that we compute a tour of length at most C. We can now start to relate the complexities of the four versions of a problem. It is obvious that the recognition problem is not more difficult than either the optimal value or the witness problem which in turn are both simpler than the optimization problem. We can capture this fact in the following diagram:

$$\begin{array}{ccc} & \le \text{witness} & \\ \text{recognition} & & \le \text{optimization.} \\ & \le \text{optimal value} & \end{array}$$

A precise formulation is

Lemma 1. *Let (Q, c) be a polynomially bounded optimization problem. Then*

a) *If function optsol : $\Gamma^* \to \Gamma^*$ is computable in polynomial time, then so are functions optval : $\Gamma^* \to \mathbb{N}^*$ and witness : $\Gamma^* \times \mathbb{N} \to \Gamma^*$.*

b) *If either function witness or optval is computable in polynomial time then $L_{(Q,c)} = \{(x, C);\ x \in \Sigma^*, C \in \mathbb{N}$ and optval$(x) \le C\} \in P$.*

Proof: Obvious. ▮

A partial converse of Lemma 1 is provided by

Lemma 2. *Let (Q, c) be a polynomially bounded optimization problem.*

a) *If functions witness and optval are computable in polynomial time, then so is optsol.*

b) *If the recognition problem is in P then optval is computable in polynomial time.*

Proof: a) We only have to observe that *optsol*$(x) =$ *witness*$(x,$ *optval*$(x))$ for all x.

b) We observe first that there is a polynomial q such that *optval*$(x) \le 2^{q(|x|)}$ for all x. We can then use binary search on the interval $[1\,..\,2^{q(|x|)}]$ in order to determine the exact value of *optval*(x). The details are given in Program 1.

It is easy to see that *low* $\le$ *optval*$(x) \le$ *high* is an invariant of the loop. Hence Program 2 correctly computes the value of *optval*(x) in $\log(2^{q(|x|)}) = q(|x|)$ iterations of the loop. In line (4) the polynomial time algorithm for the (Q, c)-recognition problem is used. Thus each iteration of the loop has polynomial cost. ▮

```
$low \leftarrow 1$; $high \leftarrow 2^{q(|x|)}$;
while $high - low \geq 1$
do $middle \leftarrow \lfloor (high + low)/2 \rfloor$;
    if $x$ has a solution of cost $\leq middle$
    then $high \leftarrow middle$
    else  $low \leftarrow middle + 1$
    fi
od;
$optval(x) \leftarrow low$.
```

Program 2

We still have to relate the complexity of the (Q, c)-witness problem and the (Q, c)-recognition problem. We treat TSP first and then comment on a general reduction.

Lemma 3. *If the Traveling Salesman Recognition Problem is in P then the TSP witness function can be computed in polynomial time.*

Proof: Program 3 computes $witness_{TSP}(dist, C)$, i.e., computes a tour of length at most C if there is any.

```
(1)  define $dist' : [0 \mathinner{..} n-1]^2 \to \mathbb{N}$ by $dist' = dist$;
(2)  if $dist$ does not have a tour of length at most $C$
     then halt "there is no tour of length $\leq C$" fi;
(3)  for all pairs $(i, j)$, $i \neq j$,
(4)  do change $dist'[i, j]$ to $\infty$;
(5)      if $dist'$ still has a tour of length at most $C$
(6)      then do nothing
(7)      else  reverse the change made in line (4)
                and include edge $(i, j)$ into the tour
(8)      fi
(9)  od.
```

Program 3

The set of edges selected in line (7) forms a tour of length at most C. The polynomial time algorithm for the recognition problem of TSP is used in lines (2) and (5). Note that all problem instances $(dist'C)$ in line (5) are not more difficult than the original problem $(dist, C)$; in particular, the length of the encoding of these instances does not (much) exceed the encoding of $(dist, C)$. The number of iterations of the loop is clearly bounded by a polynomial in input length, the cost of each iteration is also polynomially bounded. Thus $witness_{TSP}$ can be computed in polynomial time if TSP Recognition is in P. ∎

The main component in the proof of Lemma 3 is **selfreduction**. In order to find a witness (tour) for an instance of TSP we reduce the problem to a *simpler* (one less edge with cost $< \infty$) instance of TSP. We solve the simpler problem by the recognition algorithm and thus construct a piece of the witness. A similar approach works for all other problems treated in this book. We illustrate the technique by one more example, the Satisfiability Problem.

Let $V = \{x_1, x_2, \ldots\}$ be an infinite supply of propositional variables. If x_i is a variable then x_i and $\bar{x}_i$ are literals. We will use $\bar{x}_i$ and $\neg x_i$ interchangeably. If $y_1, \ldots, y_k$ are literals then $(y_1 \vee y_2 \vee \cdots \vee y_k)$ is a clause of degree k. Finally if $c_1, \cdots, c_m$ are clauses of degree at most k then $c_1 \wedge c_2 \wedge \ldots \wedge c_m$ is a formula in conjunctive normal form with at most k literals per clause. A truth assignment is a mapping $\psi : V \to \{0,1\}$. We extend ψ to literals, clauses and formulas by

$$\psi(y) = \begin{cases} \psi(x_i) & \text{if } y = x_i; \\ 1 - \psi(x_i) & \text{if } y = \neg x_i \end{cases}$$

and

$$\psi(y_1 \vee y_2 \vee \cdots \vee y_k) = \max\{\psi(y_i); 1 \le i \le k\},$$
$$\psi(c_1 \wedge c_2 \wedge \cdots \wedge c_m) = \min\{\psi(c_j); 1 \le j \le m\}.$$

A formula α is satisfiable if there is an assignment ψ with $\psi(\alpha) = 1$. The Satisfiability Recognition Problem is given by:

Name:	**Satisfiability Problem** (SAT).
Input:	A formula α in conjunctive normal form.
Question:	Is α satisfiable?

The SAT Witness Problem is to compute a function *witness* which maps to truth assignments such that $witness_{SAT}(\alpha)$ satisfies α if α is satisfiable.

Lemma 4. *If* SAT *is in P then* $witness_{SAT}$ *can be computed in polynomial time.*

Proof: Program 4 computes $\psi = witness_{SAT}(\alpha)$. It is clearly correct and has polynomial running time. ∎

Again the main component of the proof of Lemma 4 is selfreduction. We construct the witness (assignment) piecewise by reducing formula α to a simpler formula (one variable less) and testing the simpler formula for satisfiability.

```
(1) $\alpha' \leftarrow \alpha$;
(2) for all $x_i$ occurring in $\alpha$
(3) do let $\alpha''$ be obtained from $\alpha'$ by substituting the constant 0
        for all occurrences of $x_i$ in $\alpha'$;
(4)     if $\alpha''$ is satisfiable      co use the algorithm for SAT here oc
(5)     then $\psi(x_i) \leftarrow 0$; $\alpha' \leftarrow \alpha''$
(6)     else  $\psi(x_i) \leftarrow 1$; $\alpha' \leftarrow$ the formula obtained from $\alpha$
                                  by substituting 1 for all occurrences of $x_i$
(7)     fi
(8) od
```

Program 4

6.3. Reductions and -complete Problems

Reductions are useful tools for classifying problems. We have seen the technique already in the previous section and we will use it extensively throughout the chapter. For example, we showed how to convert an algorithm for TSP Recognition into an algorithm for TSP Witness, in this way reducing the witness problem to the recognition problem. More generally, reductions allow us to transform solutions to one problem into solutions to other problems.

Definition:

a) Let Σ and Γ be finite alphabets. A mapping $f : \Sigma^* \to \Gamma^*$ is a (polynomial time computable) **transformation**, if f can be computed on a TM in polynomial time.

b) Let $L_1 \subseteq \Sigma^*$ and $L_2 \subseteq \Gamma^*$ be languages. L_1 is (polynomially, many-one) **reducible** to L_2 if there is a polynomial time computable transformation f such that $x \in L_1$ iff $f(x) \in L_2$ for all $x \in \Sigma^*$. In this case we write $L1 \preceq L_2$.

c) Language L is ***NP*-complete**, if

  i) $L \in NP$,

  ii) $L' \preceq L$ for all $L' \in NP$. ∎

Theorem 1 shows the importance of this definition.

Theorem 1. *Let L_0 be NP-complete. Then*

a) *$L_0 \in P$ iff $P = NP$.*

b) *If $L_0 \preceq L_1$ and $L_1 \in NP$ then L_1 is NP-complete.*

Proof: a) If $P = NP$ then certainly $L_0 \in P$. The converse remains to be proved. Assume that $L_0 \in P$ and let $L \in NP$ be arbitrary. Since $L_0 \in P$ there is a deterministic TM M which accepts L_0 in time bounded by p for some polynomial p. Since L_0 is NP-complete and $L \in NP$ we have $L \preceq L_0$. Thus a mapping f exists with $L = f^{-1}(L_0)$. Let N be a deterministic TM which computes f in time bounded by polynomial q. We construct a deterministic acceptor A for L from M and N. A behaves as follows on input x.

(1) Compute $f(x)$ using N in time $q(|x|)$.

(2) Reset the read/write-head onto the first symbol of $f(x)$ in time $|f(x)|$.

(3) Decide if $f(x) \in L_0$ using M in time $p(|f(x)|)$. If $f(x) \in L_0$ then accept x, otherwise reject x.

The algorithm above clearly accepts L. It has running time $q(|x|) + |f(x)| + p(|f(x)|) \le r(|x|)$ for some polynomial r. The last inequality follows from the fact that $|f(x)| \le |x| + q(|x|)$ since a TM can write at most one square in one time unit.

b) We have to show $L \preceq L_1$ for every $L \in NP$. Let $L \in NP$ be arbitrary. Since L_0 is NP-complete there is a transformation f such that $L = f^{-1}(L_0)$. Since $L_0 \preceq L_1$ there is a transformation g such that $L_0 = g^{-1}(L_1)$. Let $h = g \circ f$. Then $L = f^{-1}(L_0) = f^{-1}(g^{-1}(L_1)) = (g \circ f)^{-1}(L_1) = h^{-1}(L_1)$. It remains to be shown that h is computable in polynomial time. This is easily done by an argument similar to the one used in part a). ∎

Part a) of Theorem 1 states that NP-complete problems are the most difficult problems in NP. If one of them is solvable in polynomial time, then all problems in NP are solvable in polynomial time. Conversely, if $P \ne NP$ (and this is generally believed) then no NP-complete problem can be solved in polynomial time.

Part b) introduces a simple, but extremely useful technique for showing NP-completeness. Show first that a particular language L_0 is NP-complete. For language L_0 this must be done the complicated way, namely by demonstrating $L \preceq L_0$ for all $L \in NP$. Once we have established NP-completeness for L_0, there is a simpler way of establishing NP-completeness of L_1. Show $L_1 \in NP$ and $L_0 \preceq L_1$.

6.4. The Satisfiability Problem is -complete

In this section we establish the *NP*-completeness of the Satisfiability problem. SAT was defined in Section VI.2.

Name:	SAT.
Input:	A formula α in conjunctive normal form (CNF).
Question:	Is α satisfiable?

Theorem 1. SAT *is NP-complete.*

Proof: We show SAT $\in$ *NP* first. Let α be a formula in CNF, let V_α be the set of variables occurring in α, and let $|\alpha|$ be the length of the encoding of α. Then certainly $|V_\alpha| \leq |\alpha|$. We describe a nondeterministic machine N which accepts SAT. On input α, N nondeterministically chooses an assignment $\psi : V_\alpha \to \{0,1\}$ by writing down a bitstring of length $|V_\alpha|$. Then it evaluates α with respect to assignment ψ by one of the well known methods, e.g., by the stack principle. Machine N accepts α iff $\psi(\alpha) = 1$. N clearly operates in polynomial time and accepts SAT.

Let $L \in NP$ be arbitrary. We have to show $L \preceq$ SAT. Since $L \in NP$ there is a nondeterministic TM M which accepts L in time bounded by some polynomial p. For every input x of M we will now construct a formula $\alpha(x)$ which describes the behaviour of M on x. In particular, $\alpha(x)$ is satisfiable iff M accepts x. The mapping $x \to \alpha(x)$ is the desired transformation.

Machine M has states $Z_1, \ldots, Z_s$ and tape alphabet $\Gamma = \{A_1, \ldots, A_v\}$. Symbol A_1 denotes the empty tape square, we use A_1 and $\not{b}$ interchangeably. Z_1 is the initial state of M and $Z_r, Z_{r+1}, \ldots, Z_s$ are the accepting states. We change the transition function δ of M by defining $\delta(z,a) = \{(z,a,0)\}$ whenever $\delta(z,a) = \emptyset$. Then M never stops and halting configurations (of the original machine) are endlessly repeated. Thus: M accepts x iff there is a sequence $C_0, C_1, \ldots, C_{p(n)}$ of configurations such that C_1 is the initial configuration for x, $n = |x|$, $C_i \vdash C_{i+1}$ for $0 \leq i < p(n)$, and the state in $C_{p(n)}$ is accepting. The transition function δ of M defines a relation on $Z \times \Gamma \times Z \times \Gamma \times \{-1, 0, +1\}$. We order the tuples of this relation in some way into lines; let m be the number of tuples in relation δ.

Let $x \in \Gamma^*$, $|x| = n$, be arbitrary. Formula $\alpha(x)$ is built from the following variables. We also give the intended meaning of each variable.

		Intended meaning:
$z_{t,k}$	$0 \leq t \leq p(n)$ $1 \leq k \leq s$	$z_{t,k} = 1$, if M is in state Z_k at time t.
$a_{t,i,j}$	$0 \leq t \leq p(n)$ $1 \leq i \leq p(n)$ $1 \leq j \leq v$	$a_{t,i,j} = 1$, if A_j is the content of the i-th tape square at time t.
$s_{t,i}$	$0 \leq t \leq p(n)$	$s_{t,i} = 1$, if M scans the i-th tape square at time t.

	$1 \le i \le p(n)$	
$b_{t,l}$	$0 \le t < p(n)$ $1 \le l \le m$	$b_{t,l} = 1$, if line l of δ is used for the transition from time t to time $t+1$.

Variables $z_{t,k}$ represent the state of the finite control, the $a_{t,i,j}$'s encode the tape content or more exactly, the content of the first $p(n)$ tape squares, i.e., of the relevant part of the tape, the $s_{t,i}$'s give the position of the tape head and the $b_{t,l}$'s encode the transition behaviour. Since M is $p(n)$ time bounded it can never use more than the first $p(n)$ tape squares. Thus the variables $a_{t,i,j}$ are only needed for $i \le p(n)$ and the variables $s_{t,i}$ are only needed for $i \le p(n)+1$.

We will next set up formulae which fix the intended meaning of the variables. We have to ensure the

initial condition (M starts in state Z_1, its head is on the first square, and $x\not{b}^{p(n)-n}$ is the tape content), the

boundary condition (at any time unit M is in exactly one state, each tape square contains exactly one symbol, the tape head is at exactly one square, and exactly one line of the Turing table is applied), and the

transition condition (the configurations at subsequent time units are compatible with the Turing table).

Let I be a formula for the initial condition, B a formula for the boundary condition, and T a formula for the transition condition. Then

$$\alpha(x) = I \wedge B \wedge T \wedge (z_{p(n),r} \vee z_{p(n),r+1} \vee \cdots \vee z_{p(n),s}).$$

We still have to construct I, B and T in CNF. Formula I is easy to set up. Let $x = A_{j_1}A_{j_2}\ldots A_{j_n}$. Then

$$I = (z_{0,1} \wedge s_{0,1} \wedge a_{0,1,j_1} \wedge a_{0,2,j_2} \wedge a_{0,3,j_3} \wedge \cdots \wedge a_{0,n,j_n} \wedge a_{0,n+1,1} \wedge \cdots \wedge a_{0,p(n),1}).$$

I is clearly in CNF; there are $p(n)+2$ occurrences of variables in I. Formula B is more difficult to set up. In B we want to express facts like: machine M is in exactly one state at time t. Thus we need a short formula which expresses the fact that exactly one out of a set of variables is true. Let $x_1, \ldots, x_h$ be variables. Then

$$\begin{aligned} \mathit{Exactly\text{-}One}(x_1,\ldots,x_h) = {} & \mathit{At\text{-}Least\text{-}One}(x_1,\ldots,x_h) \wedge \\ & \mathit{At\text{-}Most\text{-}One}(x_1,\ldots,x_h), \end{aligned}$$

and

$$\mathit{At\text{-}Least\text{-}One}(x_1,\ldots,x_h) = (x_1 \vee x_2 \vee \cdots \vee x_h),$$

and

$$\begin{aligned}\textit{At-Most-One}(x_1,\ldots,x_h) &= \neg \textit{At-Least-Two}(x_1,\ldots,x_h)\\ &= \neg \bigvee_{1\le i<j\le h} (x_i \wedge x_j)\\ &= \bigwedge_{1\le i<j\le h} (\bar{x}_i \vee \bar{x}_j).\end{aligned}$$

Formula *Exactly-One*$(x_1,\ldots,x_h)$ is in CNF; there are exactly $h + 2h(h-1)/2 = h^2$ occurrences of variables in it. A truth assignment ψ of $x_1,\ldots,x_h$ satisfies *Exactly-One*$(x_1,\ldots,x_h)$ iff $\psi(x_i) = 1$ for exactly one i, $1 \le i \le h$.

We define

$$B = \bigwedge_{0\le t\le p(n)} (B_{state}(t) \wedge B_{position}(t) \wedge B_{tape\ content}(t) \wedge B_{transition}(t))$$

where

$$\begin{aligned}B_{state}(t) &= \textit{Exactly-One}(z_{t,1},\ldots,z_{t,s})\\ B_{position}(t) &= \textit{Exactly-One}(s_{t,1},\ldots,s_{t,p(n)+1})\\ B_{tape\ content}(t) &= \bigwedge_{1\le i\le p(n)} \textit{Exactly-One}(a_{t,i,1},\ldots,a_{t,i,v}) \qquad \text{and}\\ B_{transition}(t) &= \textit{Exactly-One}(b_{t,1},\ldots,b_{t,m}).\end{aligned}$$

B is clearly in CNF; there are $(p(n)+1)\cdot(s^2+(p(n)+1)^2+p(n)\cdot v^2+m^2)$ occurrences of variables in B. An assignment ψ to the variables of $\alpha(x)$ satisfies B iff for every t there is exactly one j with $\psi(z_{t,j}) = 1$, exactly one h with $\psi(s_{t,h}) = 1$, ... Next we define

$$T = \bigwedge_{0\le t<p(n)} T(t),$$

where $T(t)$ expresses the fact that the transition from time t to $t+1$ is compatible with the line of the Turing table selected by variables $b_{t,1},\ldots,b_{t,m}$. Let

$$Z_{k_l}, A_{j_l}, Z_{\tilde{k}_l}, A_{\tilde{j}_l}, R_l$$

be the l-th line of the Turing table, $1 \le l \le m$, i.e., if M is in state Z_{k_l} and reads symbol A_{j_l}, then it *can* (M is nondeterministic) change its state to $Z_{\tilde{k}_l}$, print $A_{\tilde{j}_l}$ and move the head by $R_l \in \{-1,0,+1\}$ squares. Using this notation we define

$$\begin{aligned}T(t) = \bigwedge_{1\le i\le p(n)} \Bigg\{ \quad & \bigwedge_{1\le j\le v} (s_{t,i} \vee \bar{a}_{t,i,j} \vee a_{t+1,i,j}) \wedge \\ & \bigwedge_{1\le l\le m} \Big[\qquad (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee z_{t,k_l}) \wedge (s_{t,i} \vee \bar{b}_{t,l} \vee a\\ & (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee z_{t+1,\tilde{k}_l}) \wedge (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee a_{t+1,i,\tilde{j}_l}) \wedge \\ & (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee s_{t+1,i+R_l})\Big] \Bigg\}.\end{aligned}$$

Formula $T(t)$ needs some explanation. When is $(s_{t,i} \vee \bar{a}_{t,i,j} \vee a_{t+1,i,j})$ true? It is true if either $s_{t,i}$ is true, i.e., M scans cell i at time t, or $\bar{a}_{t,i,j}$ is false, i.e., the content of that cell at t is not A_j, or $a_{t+1,i,j}$ is true, i.e., the content of that cell at $t+1$ is A_j. In other words, if $s_{t,i}$ is false and $a_{t,i,j}$ is true then $a_{t+1,i,j}$ must be true, i.e., if M does not scan the i-th tape square at time t then the content of that cell does not change. Similarly, when is $(\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee a_{t,i,j_l})$ true? Well, if $s_{t,i}$ is true and $b_{t,l}$ is true then a_{t,i,j_l} must also be true, i.e., if M scans cell i at time t and we want to apply the l-th transition, then cell i must contain A_{j_l}, etc.

Formula $T(t)$ is clearly in CNF, there are $p(n) \cdot (3v + 15m)$ occurrences of variables in $T(t)$. This completes the construction of formula $\alpha(x)$.

Lemma 1. *Formula $\alpha(x)$ can be constructed in polynomial time given x, i.e., the mapping $x \to \alpha(x)$ is a polynomial time computable transformation.*

Proof: Formula $\alpha(x)$ is clearly in CNF. There are $p(n)+2+(p(n)+1)\cdot(s^2+(p(n)+1)^2+p(n)\cdot v^2+m^2)+p(n)^2\cdot(3v+15m)+s-r+1 = O(p(n)^3)$ occurrences of variables in $\alpha(x)$. If we write indices of variables in binary representation, i.e., a single variable consumes space $O(\log p(n))$, then the natural encoding of $\alpha(x)$ over the alphabet $\{(,),\wedge,\vee,\neg,s,z,a,b,0,1\}$ has length $O(p(n)^3\cdot\log p(n))$. The structure of formula $\alpha(x)$ is very simple; expression I depends on x in a very simple way, the other parts B and T only depend on $n = |x|$. Thus $\alpha(x)$ can be constructed by a TM in polynomial time. ∎

Next we have to formally relate $x \in L$ and the satisfiability of $\alpha(x)$.

Lemma 2. *If $x \in L$ then $\alpha(x)$ is satisfiable.*

Proof: If $x \in L$ then there is a sequence $C_0, C_1, \ldots, C_{p(n)}$ of configurations such that C_0 is the initial configuration of M for x, $C_i \vdash C_{i+1}$ for $0 \le i < p(n)$, and the state in $C_{p(n)}$ is accepting. Define truth assignment ψ by

$$\psi(z_{t,k}) = \begin{cases} 1 & \text{if } z_k \text{ is the state in } C_t; \\ 0 & \text{otherwise,} \end{cases}$$

$$\psi(a_{t,i,j}) = \begin{cases} 1 & \text{if } A_j \text{ is the symbol on the } i\text{-th tape square of } C_t; \\ 0 & \text{otherwise,} \end{cases}$$

$$\psi(s_{t,i}) = \begin{cases} 1 & \text{if the } i\text{-th square is scanned in } C_t; \\ 0 & \text{otherwise,} \end{cases}$$

$$\psi(b_{t,l}) = \begin{cases} 1 & \text{if the } l\text{-th line of the Turing table of } M \text{ is used} \\ & \text{in the transition from } C_t \text{ to } C_{t+1}; \\ 0 & \text{otherwise.} \end{cases}$$

It is a simple exercise to check that ψ indeed satisfies α. ∎

Lemma 3. *If $\alpha(x)$ is satisfiable then $x \in L$.*

Proof: Let ψ be a truth assignment which satisfies $\alpha(x)$. Then $\psi(B) = 1$, also, and hence for example $\psi(B_{state}(t)) = \psi(\mathit{Exactly\text{-}One}(z_{t,1}, \ldots, z_{t,s})) = 1$ for all t. Thus for every t there is exactly one k, say $k(t)$, such that $\psi(z_{t,k}) = 1$. Similarly, for every t there is exactly one i, say $i(t)$, such that $\psi(s_{t,i}) = 1$ and one l, say $l(t)$, such that $\psi(b_{t,l}) = 1$. Finally, for every t and i there is exactly one j, say $j(t,i)$, such that $\psi(a_{t,i,j}) = 1$. We conclude that the true variables define a configuration C_t for every t : $z_{k(t)}$ is the state, $A_{j(t,1)}A_{j(t,2)} \ldots A_{j(t,p(n))}$ is the tape content, and $i(t)$ is the position of the read/write-head. Assignment ψ also satisfies subformula I. Hence $k(0) = 1$, $i(0) = 1$, $j(0,1) = j_1$, $j(0,2) = j_2$, etc., where $x = A_{j_1}A_{j_2} \ldots$. Thus C_0 is the initial configuration of M on input x. Next observe that $\psi(z_{p(n),r} \vee \cdots \vee z_{p(n),s}) = 1$ and hence $r \leq k(p(n)) \leq s$. Thus the state in $C_{p(n)}$ is accepting. We still have to show $C_t \vdash C_{t+1}$ for $0 \leq t < p(n)$.

We also have $\psi(T(t)) = 1$. Thus $\psi(s_{t,i} \vee \bar{a}_{t,i,j} \vee a_{t+1,i,j}) = 1$ for all i and j. In particular for $i \neq i(t)$ and hence $\psi(s_{t,i}) = 0$ and $j = j(t,i)$ and hence $\psi(a_{t,i,j}) = 1$, we infer $\psi(a_{t+1,i,j}) = 1$. Thus the content of the i-th tape square is the same in C_t and C_{t+1} for $i \neq i(t)$, i.e., C_t and C_{t+1} differ at most in the vicinity of the tape head. We also have for all i and l

$$\psi((\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee z_{t,k_l}) \wedge (\bar{s}_{t,i} \vee \cdots) \wedge \cdots \wedge (\cdots \vee s_{t+1,i+R_l})) = 1.$$

Thus $k(t) = k_{l(t)}$, $j(t,i(t)) = j_{l(t)}$, $k(t+1) = \tilde{k}_{l(t)}$, $j(t+1, i(t+1)) = \tilde{j}_{l(t)}$, and $i(t+1) = i(t) + R_{l(t)}$. In other words: line $l(t)$ of M's Turing table is applicable to C_t and yields C_{t+1} when applied to C_t. Thus C_{t+1} is a successor configuration of C_t.

In summary, we have shown that there is an accepting computation of M on input x, i.e., $x \in L$. ▮

Lemmata 1, 2 and 3 establish that $L \preceq$ SAT. Since L is arbitrary, we conclude $L \preceq$ SAT for every $L \in \mathit{NP}$. ▮

SAT is our first *NP*-complete problem. We will use it to show *NP*-completeness of many other problems in the next section. Before doing so, we show that SAT remains *NP*-complete if we confine ourselves to formulae having at most three literals per clause.

Name:	SAT(3).
Input:	A formula α in CNF with at most three literals per clause.
Question:	Is α satisfiable?

Theorem 2. SAT(3) *is NP-complete.*

Proof: SAT(3) $\in \mathit{NP}$ is trivial; we only have to show SAT $\preceq$ SAT(3); In order to reduce SAT to SAT(3) we have to replace clauses of arbitrary degree by clauses having degree 3. Let $x_1 \vee x_2 \vee \cdots \vee x_n$ be a clause. Let $y_1, \ldots, y_n$ be new variables. Consider

$$\alpha = (x_1 \vee \bar{y}_1) \wedge (y_1 \vee x_2 \vee \bar{y}_2) \wedge \cdots \wedge (y_{n-1} \vee x_n \vee \bar{y}_n) \wedge y_n.$$

Claim: *Let $\psi : \{x_1 \ldots x_n\} \to \{0,1\}$ be a truth assignment. There is an extension $\phi : X \cup Y \to \{0,1\}$ of ψ with $\phi(\alpha) = 1$ iff $\psi(x_1 \vee \cdots \vee x_n) = 1$.*

Proof: "$\Leftarrow$": Assume $\psi(x_1 \vee \cdots \vee x_n) = 1$. Let i_0 be the least i such that $\psi(x_i) = 1$. Define

$$\phi(z) = \begin{cases} \psi(x_j) & \text{if } z = x_j \text{ for some } j; \\ 0 & \text{if } z = y_j \text{ for some } j < i_0; \\ 1 & \text{otherwise.} \end{cases}$$

Then $\phi(\alpha) = 1$ as the reader can check easily.

"$\Rightarrow$": Let $\phi : X \cup Y \to \{0,1\}$ be a truth assignment with $\phi(\alpha) = 1$. We have to show $\phi(x_1 \vee x_2 \vee \cdots \vee x_n) = 1$. Assume otherwise. Then $\phi(x_i) = 0$ for all i. We show $\phi(y_i) = 0$ for all i by induction on i. Since $\phi(x_1 \vee \bar{y}_1) = 1$ we must have $\phi(y_1) = 0$. Next, we infer from $\phi(y_1 \vee x_2 \vee \bar{y}_2) = 1$ that $\phi(y_2) = 0$. Similarly, we conclude $\phi(y_3) = \cdots = \phi(y_n) = 0$ and hence $\phi(\alpha) = 0$, a contradiction. ▮

The reduction of SAT to SAT(3) is easy now. Replace any clause with more than three literals by a set of clauses as described above. ▮

How about SAT(2)? Is it still *NP*-complete? No, SAT(2) is in *P* (Exercise 6).

6.5. More -complete Problems

We extend our list of *NP*-complete problems and show *NP*-completeness of Clique, (0,1)-Integer Programming, Vertex Cover, Hamiltonian Cycle, Traveling Salesman, 3-dimensional Matching, Knapsack, Scheduling Independent Tasks and Precedence Constrained Scheduling.

Name:	Clique.
Input:	Undirected graph $G = (V, E)$ and integer k.
Question:	Is there a clique of size k in G, i.e., is there $V' \subseteq V$ with $\lvert V' \rvert = k$ and $(v, w) \in E$ for all $v, w \in V'$?

Theorem 1. *Clique is NP-complete.*

Proof: Clique $\in$ *NP* was shown in the introduction. We complete the proof by showing SAT(3) $\preceq$ Clique. Let $\alpha = c_1 \wedge \cdots \wedge c_k$ be a formula in CNF with at most three literals per clause. Let $c_i = x_{i,1}^{\beta_{i,1}} \vee x_{i,2}^{\beta_{i,2}} \vee x_{i,3}^{\beta_{i,3}}$ where $\beta_{i,h} \in \{0,1\}$ and x^1 denotes x and x^0 denotes $\bar{x}$. We construct an undirected graph $G = (V, E)$ with

$$V = \{v_{i,h}; 1 \le i \le k, 1 \le h \le 3\}$$

and $(v_{i,h}, v_{j,m}) \in E$ iff $i \neq j$ and ($x_{i,h} \neq x_{j,m}$ or $\beta_{i,h} = \beta_{j,m}$), i.e., if $v_{i,h}$ and $v_{j,m}$ are not complements of each other.

Claim: *α is satisfiable iff G has a clique of size k.*

Proof: "$\Rightarrow$": Let ψ be truth assignment with $\psi(\alpha) = 1$ and hence $\psi(c_i) = 1$ for all i, $1 \leq i \leq k$. For every i there must be h, say $h(i)$, such that $\psi(x_{i,h(i)}^{\beta_{i,h(i)}}) = 1$. Let $V' = \{v_{i,h(i)};\ 1 \leq i \leq k\}$. Then V' is a clique of size k.

"$\Leftarrow$": Let $V' \subseteq V$ be a clique of size k. Since $(v_{i,h}, v_{j,m}) \in E$ implies $i \neq j$ we conclude that $V' = \{v_{i,h(i)};\ 1 \leq i \leq k\}$ for some function h. Define truth assignment ψ by

$$\psi(x) = \begin{cases} 1 & \text{if } x = x_{i,h(i)} \text{ for some } i \text{ and } \beta_{i,h(i)} = 1, \\ 0 & \text{if } x = x_{i,h(i)} \text{ for some } i \text{ and } \beta_{i,h(i)} = 0, \\ \text{arbitrary} & \text{otherwise.} \end{cases}$$

ψ is well-defined. If $\psi(x)$ were not well-defined then there would have to be i and j, $i \neq j$, such that $x = x_{i,h(i)} = x_{j,h(j)}$ and $\beta_{i,h(i)} \neq \beta_{j,h(j)}$. However, this implies $(v_{i,h(i)}, v_{j,h(j)}) \notin E$, contradiction. Thus ψ is well defined. Also $\psi(x_{i,h(i)}^{\beta_{i,h(i)}}) = 1$ for all i and hence $\psi(\alpha) = 1$. ∎

It is easy to construct G from α in polynomial time. Thus SAT(3) $\preceq$ Clique. ∎

Name:	$(0,1)$-Integer Programming (IP).
Input:	Integer matrix C and integer vector d.
Question:	Is there a $(0,1)$-vector c such that $C \cdot c \geq d$?

Theorem 2. *$(0,1)$-Integer Programming is NP-complete.*

Proof: IP $\in$ *NP* is obvious; guess vector c nondeterministically and check $C \cdot c \geq d$. We show SAT $\preceq$ IP. Let $\alpha = z_1 \wedge \cdots \wedge z_k$ be a formula in CNF and let $x_1, \ldots, x_n$ be the variables occurring in α. Define C and d as follows: $C = (C_{ij})_{1 \leq i \leq k, 1 \leq j \leq n}$ and $d = (d_i)_{1 \leq i \leq k}$ where

$$C_{rj} = \begin{cases} 1 & \text{if } x_j \text{ occurs in } z_i; \\ -1 & \text{if } \bar{x}_j \text{ occurs in } z_i; \\ 0 & \text{otherwise,} \end{cases}$$

and

$$d_i = 1 - \#\text{ of variables } x_j \text{ with } \bar{x}_j \text{ occurs in } z_i.$$

Claim: *α is satisfiable iff there is a $(0,1)$-vector c such that $C \cdot c \geq d$.*

Proof: "$\Rightarrow$": Let ψ be an assignment with $\psi(\alpha) = 1$. Define $c_j = \psi(x_j)$ for all j. Then

$$(C \cdot c)_i = \sum_{j=1}^{n} C_{ij} \cdot c_j = \sum_{x_j \in z_i} \psi(x_j) - \sum_{\bar{x}_j \in z_i} \psi(x_j)$$
$$\geq 1 - \sum_{\bar{x}_j \in z_i} 1 = d_i,$$

since there is either $x_j \in z_i$ with $\psi(x_j) = 1$ or $\bar{x}_j \in z_i$ with $\psi(x_j) = 0$.

"$\Leftarrow$": Let c be a $(0,1)$-vector with $C \cdot c \geq d$. Define the truth assignment ψ by $\psi(x_j) = c_j$. We claim $\psi(\alpha) = 1$. Assume otherwise. Then there must be an i such that $\psi(z_i) = 0$; in particular $\psi(x_j) = c_j = 0$ if $x_j \in z_i$ and $\psi(x_j) = c_j = 1$ if $\bar{x}_j \in z_i$. Hence

$$d_i \leq (C \cdot c)_i = \sum_{x_j \in z_i} c_j - \sum_{\bar{x}_j \in z_i} c_j = - \sum_{\bar{x}_j \in z_i} 1 < d_i,$$

contradiction. Thus $\psi(\alpha) = 1$. ∎

The observation that C and d can be computed in polynomial time finishes the proof that SAT $\preceq$ IP. ∎

Name:	Vertex Cover (VC).
Input:	Undirected graph $G = (V, E)$ and integer k.
Question:	Is there $V' \subseteq V$, $\lvert V' \rvert = k$ such that for every edge $(v, w) \in E$ at least one fo v and w belongs to V'?

Theorem 3. *Vertex Cover is NP-complete.*

Proof: VC $\in$ *NP* is obvious. We show Clique $\preceq$ VC. Let $G = (V, E)$ be an undirected graph and let k be an integer. Let $\hat{G} = (V, V \times V - E)$ be the complement of G and let $\hat{k} = |V| - k$. Then V' is a clique in G iff $V - V'$ is a vertex cover of $\hat{G}$. Thus G has a clique of size k iff $\hat{G}$ has a vertex cover of size $\hat{k}$. The mapping $G \to \hat{G}$ is clearly computable in polynomial time. ∎

Name:	Directed Hamiltonian Cycle (DHC).
Input:	Directed graph $G = (V, E)$.
Question:	Is there a simple cycle in G which goes through all vertices, i.e., is there a sequence $v_0, \ldots, v_{n-1}$ with $v_i \neq v_j$ for $i \neq j$ and $(v_i, v_{(i+1) \bmod n}) \in E$ for $0 \leq i < n$, $n = \lvert V \rvert$?

Theorem 4. *Directed Hamiltonian Cycle is NP-complete.*

Proof: Throughout this proof we will write undirected edges as sets of their endpoints in order to distinguish them from directed edges. DHC $\in$ *NP* is obvious. We show VC $\preceq$ DHC. Let $G = (V, E)$ be an undirected graph and let k be an integer. For every node v_i let $e_{i1}, e_{i2}, \ldots, e_{ih_i}$ be the edges incident to v_i. We construct a directed graph $G' = (V', E')$ by

$$V' = \{a_1, \ldots, a_k\} \cup \{(i, j, \alpha);\ 1 \leq i \leq n, 1 \leq j \leq h_i, \alpha \in \{0, 1\}\}$$

and

$$\begin{aligned} E' = \{&(a_r, (i,1,0));\ 1 \le r \le k, 1 \le i \le n\} \cup \\ &\{((i,j,0),(i,j,1));\ 1 \le i \le n, 1 \le j \le h_i\} \cup \\ &\{((i,j,1),(i,j+1,0));\ 1 \le i \le n, 1 \le j < h_i\} \cup \\ &\{((i,h_i,1),a_r);\ 1 \le i \le n, 1 \le r \le k\} \cup \\ &\{((i,j,0),(i',j',0));\ e_{ij} = \{v_i, v_{i'}\} \text{ and } e_{i'j'} = \{v_{i'}, v_i\} cup \\ &\{((i',j',1),(i,j,1));\ e_{ij} = \{v_i, v_{i'}\} \text{ and } e_{i'j'} = \{v_{i'}, v_i\} cr \end{aligned}$$

$$\begin{tikzcd}
(110) \arrow[r] \arrow[d, shift left] & (111) \arrow[d, shift left] & & & & \\
(210) \arrow[r] \arrow[u, shift left] & (211) \arrow[r] \arrow[u, shift left] & (220) \arrow[r] \arrow[d, shift left] & (221) \arrow[d, shift left] & & \\
& & (310) \arrow[r] \arrow[u, shift left] & (311) \arrow[r] \arrow[u, shift left] & (320) \arrow[r] \arrow[d, shift left] & (321) \arrow[d, shift left] \\
& & & & (410) \arrow[r] \arrow[u, shift left] & (411) \arrow[u, shift left]
\end{tikzcd}$$

Figure 3. Example for proof of Theorem 4

Figure 3 illustrates the construction for $v = \{v_1, v_2, v_3, v_4\}$, $E = \{\{v_1, v_2\}v_2, v_2\}v_3, v_4\}$ and $k = 2$. We have $e_{21} = \{v_2, v_1\}$, $e_{22} = \{v_2, v_3\}$, $e_{31} = \{v_3, v_2\}$ and $e_{32} = \{v_3, v_4\}$. Nodes a_1 and a_2 are not drawn. Vertex cover $\{v_1, v_3\}$ of G corresponds to the Hamiltonian path a_1, (110), (210), (211), (111), a_2, (310), (220), (221), (311), (320), (410), (411), (321), a_1 in G'.

Claim: *G has a vertex cover $\tilde{V} \subseteq V$ of size k iff G' has a Hamiltonian cycle.*

Proof: "$\Rightarrow$": Let $\tilde{V} = \{v_{i_1}, v_{i_2}, \ldots, v_{i_k}\}$ be a vertex cover of G of size k. We construct a Hamiltonian cycle in G' inductively. We start in vertex a_1 and go to $(i_1, 1, 0)$ first. Suppose now, that we reached vertex $(i_1, j, 0)$.

Case 1: $e_{i_1 j} = (v_{i_1}, v_{i'})$ and $v_{i'} \notin \tilde{V}$.
Then we proceed $(i_1, j, 0) \to (i', j', 0) \to (i', j', 1) \to (i_1, j, 1)$. Here $e_{i'j'} = e_{i_1 j}$.

Case 2: $e_{i_1 j} = (v_i, v_{i'})$ and $v_{i'} \in \tilde{V}$.
Then we proceed directly to $(i_1, j, 1)$.

We have reached vertex $(i_1, j, 1)$ by now. If $j + 1 \le h_{i_1}$ then we proceed to $(i_1, j + 1, 0)$, otherwise we proceed to a_2. From a_2 we go to $(i_2, 1, 0)$, $\ldots$. In this way we construct a Hamiltonian cycle in G'.

"$\Leftarrow$": Suppose that G' has a Hamiltonian cycle C. We note some properties of C first.

$$\begin{tikzcd}
{} \arrow[r] & (i,j,0) \arrow[r] \arrow[d, shift left] & (i,j,1) \arrow[r] \arrow[d, shift left] & {} \\
{} \arrow[r] & (i',j',0) \arrow[r] \arrow[u, shift left] & (i',j',1) \arrow[r] \arrow[u, shift left] & {}
\end{tikzcd}$$

Figure 4. Fundamental square of G'

Consider one of the squares as depicted in Figure 4and assume that C enters the square through the left upper corner, i.e., through vertex $(i, j, 0)$. Then it leaves the square through one of the right corners. If C leaves the square through the right lower corner, vertex $(i', j', 1)$, then either $(i, j, 1)$ or $(i', j', 0)$ is not on C, a contradiction. Hence C leaves the square through vertex $(i, j, 1)$. Thus the four corners of the square are traversed in one of the two following ways.

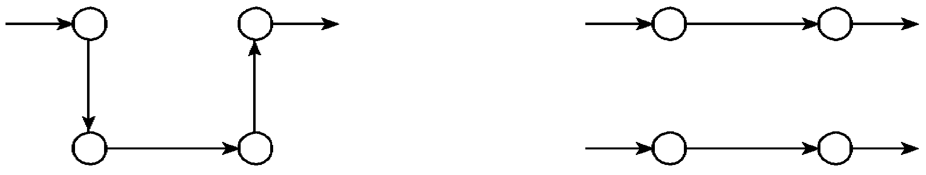

Figure 5. The 2 possibilities to traverse a square

C goes through vertices $a_1, \ldots, a_k$. The edges emanating from a_r, $1 \leq r \leq k$, end in vertices $(i, 1, 0)$, $1 \leq i \leq n$. Let $(i_r, 1, 0)$ be the node following a_r in cycle C. We show that $\widetilde{V} = \{v_{i_r};\ 1 \leq r \leq k\}$ is a vertex cover of G.

C goes from a_r to $(i_r, 1, 0)$. The argument above shows that the subsequent vertices on C are essentially the vertices $(i_r, j, 0)$, $(i_r, j, 1)$, $1 \leq j \leq h_{i_r}$. However, between $(i_r, j, 0)$ and $(i_r, j, 1)$ there is possibly a detour to nodes $(i', j', 0)$ and $(i', j', 1)$ for some i', j'.

It is now easy to show that $\tilde{V}$ is a vertex cover. Let $e \in E$ be arbitrary, say $e = (v_i, v_{i'})$. Consider the square corresponding to e. It is traversed in one of the two possible ways described above. In the first case vertex v_i is in $\widetilde{V}$, in the second case v_i and $v_{i'}$ are in $\widetilde{V}$. ∎

Of course, it is easy to construct from a given undirected graph $G = (V, E)$ the directed graph $G' = (V', E')$ in polynomial time, and hence VC $\preceq$ DHC. ∎

Name:	Undirected Hamiltonian Cycle (UHC).
Input:	Undirected graph $G = (V, E)$.
Question:	Is there a simple cycle going through every node, i.e., is there a sequence $v_0, \ldots, v_{n-1}$ such that $v_i \neq v_j$ for $i \neq j$ and $(v_i, v_{(i+1) \bmod n}) \in E$ for $0 \leq i < n$, $n = \lvert V \rvert$?

Theorem 5. *UHC is NP-complete.*

Proof: We show DHC $\preceq$ UHC. Let $G = (V, E)$ be a directed graph. Construct undirected graph G' from G by replacing every vertex v as shown in Figure 6.

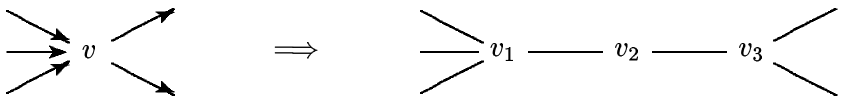

Figure 6. Transformation of DHC into UHC

There is a one-to-one correspondence between Hamiltonian cycles of G and G' because in G' a cycle must enter through v_1 (or v_3) then go to v_2 and then continue to v_3 (or v_1). this transformation is computable in polynomial time. ∎

Name: Symmetric Traveling Salesman Problem with Triangle Inequality ($\triangle$TSP)▌

Input: A matrix *dist* : $[0 \ldots n-1]^2 \to \mathbb{N}$ and integer D. Matrix *dist* is symmetric and satisfies the triangle inequality, i.e., $dist(i,j) + dist(j,k) \geq dist(i,k)$ for all i, j and k.

Question: Is there a tour of length at most D, i.e., a permutation Π of $[0\,..\,n-1]$ such that

$$\sum_{i=0}^{n-1} dist(\Pi(i), \Pi((i+1) \bmod n)) \leq D \ ?$$

Theorem 6. *$\triangle$TSP is NP-complete.*

Proof: We show UHC $\preceq$ $\triangle$TSP. Let $G = (V, E)$ be an undirected graph with $V = [0\,..\,n-1]$. We define

$$dist(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E; \\ 2 & \text{otherwise} \end{cases}$$

and $D = n$. Then G has a Hamiltonian cycle iff *dist* has a tour which uses only edges of length 1 iff *dist* has a tour of length n. Also matrix *dist* can be computed in polynomial time. ∎

Name: 3-dimensional Matching (3DM).
Input: Sets X, Y, Z of equal cardinality and a relation $U \subseteq X \times Y \times Z$.
Question: Is there $U' \subseteq U$, $|U'| = |X|$, which covers all elements of $X \cup Y \cup Z$, i.e., $\forall w \in X \cup Y \cup Z\ \exists u \in U'$ such that $u = (w, \ , \ )$ or $u = (\ , w, \ )$ or $u = (\ , \ , w)$?

One can think of X as boys, Y as girls, Z as houses and U as the compatibility relation between boys, girls and houses. The question is to find a complete (or perfect) matching. The corresponding 2-dimensional problem is equivalent to finding complete matchings in bipartite graphs. It can be solved in polynomial time, see Chapter 4.

Theorem 7. *3DM is NP-complete.*

Proof: 3DM is obviously in *NP*. We show SAT(3) $\preceq$ 3DM to prove completeness of 3DM. Let $\alpha = C_0 \wedge C_1 \wedge \cdots \wedge C_{k-1}$ be an instance of SAT(3) in variables $x_1, \ldots, x_n$. We will construct an instance I of 3DM such that α is satisfiable iff I has a solution. The set U of triples is logically divided into three groups which we will now specify. The tiles in the first group are used to select a truth assignment, the tiles in the second group are used to check for satisfaction and the tiles in the third group perform garbage collection.

Group 1: Selecting a truth assignment.
For every variable x_i, $1 \leq i \leq n$, we have $2k$ triples

$$\{(a_{ij}, x_{ij}, b_{ij}), (a_{i,(j+1) \bmod k}, \bar{x}_{ij}, b_{ij});\ 0 \leq j < k\}.$$

There will be no other triples containing points a_{ij} and b_{ij}, $1 \leq i \leq n$, $0 \leq j < k$. The triples in group 1 can be visualized as in Figure 7.

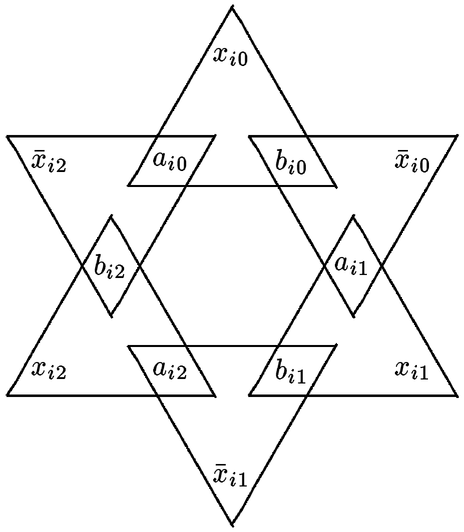

Figure 7. Triples or group 1 for x_i, where $k = 3$

There are exactly two ways of covering points a_{ij}, b_{ij}, $0 \leq j < k$, using the triples in group 1. One leaves all points x_{ij} exposed and covers all points $\bar{x}_{ij}$, $0 \leq j < k$, i.e., x_i is assigned true, and the other one leaves all points $\bar{x}_{ij}$ exposed and covers all points x_{ij}, $0 \leq j < k$, i.e., x_i is assigned false. In this way, the triples in group 1 fix a truth assignment.

Group 2: Checking for satisfaction.
For every clause C_j, $0 \leq j < k$, we have triples (C_j^1, l_j, C_j^2) for every literal l appearing in clause C_j, i.e., we have either one, two or three triples for clause C_j. There are no other triples containing points C_j^1 and C_j^2. Suppose now that the tiles

in group 1 have already been placed. Then points C_j^1 and C_j^2 can be covered iff there is some i such that x_{ij} (or $\bar{x}_{ij}$) appears in C_j and is left exposed by the tiles in group 1. Thus points C_j^1, C_j^2, $0 \leq j < k$ can be covered iff the truth assignment specified by the triples in group 1 satisfies α.

Group 3: Garbage collection.
At this point we construct an instance I of 3DM from α having the following property. If α is unsatisfiable then there can be no complete matching in I. If α is satisfiable then there is a way to select the triples in groups 1 and 2 so that points a_{ij}, b_{ij}, C_j^1 and C_j^2 are all covered and that exactly $n \cdot k - k = (n-1) \cdot k$ of the points x_{ij}, $\bar{x}_{ij}$ are uncovered. Thus in group 3 we add triples

$$\{(h_r, x_{ij}, q_r), (h_r, \bar{x}_{ij}, q_r);\ 1 \leq r \leq (n-1) \cdot k, 1 \leq i \leq n, 0 \leq j < k\}$$

which allow us to complete the covering.

It is now easy to see that the instances $I(\alpha)$ of 3DM defined above can be indeed constructed in polynomial time given α and that α is satisfiable iff $I(\alpha)$ allows for a complete matching. This shows SAT(3) $\preceq$ 3DM. ∎

Name:	Knapsack.
Input:	A set $a_1, \ldots, a_n, b$ of integers.
Question:	Is there a $J \subseteq \{1, \ldots, n\}$ such that $\sum_{j \in J} a_j = b$?

Theorem 8. *Knapsack is NP-complete.*

Proof: Knapsack is apparently in *NP*. We show 3DM $\preceq$ Knapsack to prove completeness. Let $X = \{x_1, \ldots, x_q\}$, $Y = \{y_1, \ldots, y_q\}$, $Z = \{z_1, \ldots, z_q\}$ and $U \subseteq X \times Y \times Z$, $|U| = m$, be an instance of 3DM. We construct an instance of Knapsack from it. For every triple $u_l = (x_i, y_j, z_k) \in U$ define integer a_l by

$$a_l = 2^{s(2q+i-1)} + 2^{s(q+j-1)} + 2^{s(k-1)}$$

where $s = \lceil 1 + \log m \rceil$, i.e., the binary representation of a_l consists of $3q$ blocks, the first q blocks representing points in X, the second q blocks representing points in Y and the last q blocks representing points in Z. A block is $s = 1 + \lceil \log m \rceil$ bits long and either contains (the binary representation of) integer 0 or 1 depending on whether the corresponding point belongs to the triple or not. Finally let

$$b = \sum_{j=1}^{3q} 2^{s(j-1)},$$

i.e., every block of b contains integer 1. Next note that if $U' \subseteq U$ is a solution of 3DM then $b = \sum_{u_l \in U'} a_l$. However, if $U' \subseteq U$ is not a solution of 3DM then $b \neq \sum_{u_l \in U'} a$. This follows from the fact that $|U'| \leq |U| = m$ and hence the one's in any block can add up to at most $m < 2^s$. Thus there can be no overflow from one block to the next. Also, the transformation is computable in polynomial time.
∎

It is worthwhile to take a closer look at the problem instances of Knapsack constructed in the proof of Theorem 8. We showed NP-completeness of Knapsack by a two-step reduction:

$$\mathrm{SAT}(3) \preceq \mathrm{3DM} \preceq \mathrm{Knapsack}.$$

Suppose that we start out with a formula including k clauses in n variables. From this we construct instances of 3DM with $q = |X| = |Y| = |Z| = 2n \cdot k$ and $m = |U| \leq 2n \cdot k + 3k + 2 \cdot (n \cdot k)^2 \leq q^2$. The reduction of 3DM to Knapsack then yields $m + 1 = O(q^2)$ numbers of length at most $3q \cdot (1 + \lceil \log m \rceil) = O(q \log q)$. Thus a problem instance obtained in this way has length $L = O(q^3 \log q)$. However, $b = (2^{s \cdot 3q} - 1)/(2^s - 1) = \Omega(2^{q \log q}) = \Omega(2^{L^{1/3}})$, i.e., the numerical value of b is exponential in the size of the problem instance. The very large value of b is intrinsic to *NP*-completeness as we will see in the next section on dynamic programming. We will show there that Knapsack can be solved in time $O(n \cdot b)$.

Name:	Scheduling Independent Tasks (SIT).
Input:	A sequence $(t_1, \ldots, t_n)$ of time requirements for n jobs, $t_i \in \mathbb{N}$, a number $m \in \mathbb{N}$ of machines and a deadline T.
Question:	Is there a schedule $S : [1\,..\,n] \to [1\,..\,m]$ such that for every $j \in [1\,..\,m]$ $\sum_{i \in S^{-1}(j)} t_i \leq T$, i.e., is it possible to distribute the jobs onto the machines such that all jobs are finished before time T?

Theorem 9. *Scheduling Independent Tasks (SIT) is NP-complete.*

Proof: SIT $\in$ *NP* is obvious. We show Knapsack $\preceq$ SIT. Let $a_1, \ldots, a_n, b$ be an instance of Knapsack and let $c = a_1 + a_2 + \cdots + a_n$. We may assume w.l.o.g. that $c \geq 2b$. Consider the following instance of SIT: $a_1, a_2, \ldots, a_n, c - 2b$ are the time requirements on $n + 1$ jobs, $m = 2$ and $T = c - b$. If the instance of Knapsack has a solution J, then the assignment of the jobs in J and of the jobs with cost $c - 2b$ to machine 1 and of all other jobs to machine 2 is a solution to the instance of SIT. Conversely, let $S : [1\,..\,n+1] \to \{1, 2\}$ be a solution of SIT. Assume w.l.o.g. that $S(n + 1) = 1$. Let $J = S^{-1}(1) \cap [1\,..\,n]$. Then $c - b = T = c - 2b + \sum_{i \in J} s_i$ and hence J solves the instance of Knapsack. ∎

The proof of Theorem 9 actually shows a stronger result than claimed. Scheduling Independent Tasks remains *NP*-complete for any fixed number $m \geq 2$ machines.

Name:	Precedence Constrained Scheduling (PCS).
Input:	A number $n \in \mathbb{N}$ of unit-cost jobs, a number $m \in \mathbb{N}$ of machines, a deadline $T \in \mathbb{N}$ and a precedence relation R on the jobs, i.e., $([1\,..\,n], R)$ is an acyclic digraph.
Question:	Is there a schedule $S : [1\,..\,n] \to [1\,..\,T]$ such that $\lvert S^{-1}(t) \rvert \leq m$ for all t and $(i, j) \in R \Rightarrow S(i) < S(j)$ for all i, j?

Theorem 10. *Precedence Constrained Scheduling (PCS) is NP-complete.*

Proof: PCS $\in NP$ is obvious. We show Clique $\preceq$ PCS. Let $G = (V, E)$ be an undirected graph without isolated vertices and let $k \in \mathbb{N}$ be an instance of Clique. We construct an instance I of PCS from it with deadline $T = 3$ such that there is a feasible schedule iff G has a clique of size k. The jobs in I consist of 5 groups: the vertices V of G, the edges E of G, and three non-empty sets of fill-in jobs F_1, F_2, F_3, i.e., $n = |V| + |E| + |F_1| + |F_2| + |F_3|$. Furthermore

$$R = F_1 \times F_2 \cup F_2 \times F_3 \cup V \times F_3 \cup \{(v, e);\ v \in V, e \in E \text{ and } v \text{ is incident to } e\}.$$

In a feasible schedule all jobs in F_i are executed at time i, $1 \le i \le 3$. Also only vertices and no edges are executed at time one and all vertices must be executed before time three. We complete the construction by choosing $|F_i|$, $1 \le i \le 3$, and m appropriately:

$$|F_1| \ge 1, \quad |F_2| \ge 1, \quad |F_3| \ge 1, \text{ and}$$
$$m = k + |F_1| = n - k + k \cdot (k-1)/2 + |F_2| = |E| - k \cdot (k-1)/2 + |F_3|.$$

It is easy to see that $|F_1|$, $|F_2|$ and $|F_3|$ exist. We still have to relate the existence of a clique of size k in G with the existence of a schedule of length 3. Note first that there are exactly $3m$ jobs, i.e., in a schedule of length 3 all machines must be busy at all times. Next note that a feasible schedule has to schedule exactly k vertices, say $V' \subseteq V$, at time one and the remaining $n - k$ vertices at time two. Hence a schedule of length three exists iff exactly $k \cdot (k-1)/2$ edges can be scheduled at time two. However, only edges between nodes of V' can be scheduled at time two and there are at most $|V'| \cdot (|V'| - 1)/2 = k \cdot (k-1)/2$ of them. The equality holds iff V' is a clique of size k. ▮

Again the proof of Theorem 10 actually shows a somewhat stronger result. PCS remains NP-complete if only instances with deadline $T = 3$ are considered.

6.6. Solving NP-complete Problems

We have seen that NP-complete problems are probably very difficult to solve. Nevertheless, they occur frequently and have to be solved in practice. What can we do? There are several useful approachs.

a) Special cases: Reexamine the problems at hand. Do you really want to solve the NP-complete problem in its full generality, or is it (good) enough to solve a special case? The special case might have a polynomial time solution. Precedence Constrained Scheduling is a good example. At least the following special cases of PCS are in P: The case of only two processors and the case of the precedence relation being a forest.

b) Dynamic Programming and Branch-and-Bound are two problem solving techniques which can be applied to most NP-complete problems. In this section we treat these techniques in detail. Both techniques are essentially clever variants of exhaustive search. Dynamic Programming yields surprisingly efficient algorithms for some problems, e.g., Knapsack; Branch-and-Bound uses lower bounds on the cost of optimal solutions to guide the search.

c) Probabilistic analysis can sometimes show that the complicated instances of an NP-complete problem are quite rare. It is therefore possible to design algorithms with good expected running time. Of course, the problem of justifying the probability distribution postulated on the problem instances always remains.

d) Approximation algorithms can sometimes yield very good solutions in little time. Section 6.7 is devoted to approximation algorithms.

e) Heuristics: Finally there is still room for heuristics, i.e., for algorithms which seem to work well in practice but for a reason nobody understands.

6.6.1. Dynamic Programming

Dynamic Programming is a clever form of exhaustive search. We illustrate the method on two examples: TSP and Knapsack.

Consider an instance $dist : [0 \ldots n-1]^2 \to \mathbb{N}$ of TSP. Naive exhaustive search has to test $n!$ possibilities. Dynamic programming allows us to cut down that number considerably, although it is still necessary to test an exponential number of candidates. We construct iteratively optimal tours through k cities, $k = 1, 2, 3, \ldots$. Since every tour has to go through city 0 we start our tours w.l.o.g. in city 0. For $S \subseteq [1 \,..\, n-1]$ and $i \in S$ let $C(S, i)$ be the minimal length of any tour which starts in city 0, goes through all cities in S and ends in city i. Then

$$C(\{i\}, i) = dist(0, i) \quad \text{for } 1 \le i \le n-1$$

and

$$C(S,i) = \min_{k \in S-\{i\}} [C(S-\{i\},k) + dist(k,i)]$$

for $|S| \geq 2$. The length of the optimal tour is given by $\min\{C([1..n-1],i) + dist(i,0),\ 1 \leq i \leq n-1\}$. The optimal tour itself is also easily constructed. One only has to store the value of k which defines $C(S,i)$ together with $C(S,i)$. Then the optimal tour can be constructed in a second pass over the matrix of $C(S,i)$'s. The total cost of this algorithm is

$$O(n-1+\sum_{l=2}^{n-1} \underbrace{\binom{n-1}{l} \cdot l}_{\substack{\text{Number of } C(S,i)\\ \text{with } |S|=l}} \cdot \underbrace{(l-1)}_{\text{Number of } k\text{'s}})$$

$$= O((n-1)(n-2) \cdot 2^{n-3} + (n-1)).$$

Thus dynamic programming reduces the number of candidates to be tested from $N!$ to $n^2 \cdot 2^n$, a drastic improvement.

Another illustrative example is Knapsack. Let $a_1, \ldots, a_n, b$ be an instance of Knapsack. Define bitvector $B[0..b]$ by:

$$B[s] = \begin{cases} 1 & \text{if } s = \sum_{i=1}^{b} a_i \cdot x_i \text{ for some } (x_1, \ldots, x_n) \in \{0,1\}^n; \\ 0 & \text{otherwise.} \end{cases}$$

Vector B can be computed in time $O(n \cdot b)$ by Program 5.

```
$B[0] \leftarrow$ true; $B[s] \leftarrow$ false for $1 \leq s \leq b$;
for all $i \in [1..n]$
do co $B[s] = 1$ iff $s = \sum_{j=1}^{i-1} a_j \cdot x_j$ for some $(x_1, \ldots, x_{i-1}) \in \{0,1\}^{i-1}$ oc
    for $s$ from $b$ step -1 to $x_i$
    do if $B[s - x_i]$ then $B[s] \leftarrow$ true od;
od.
```

Program 5

Theorem 1. *Knapsack can be solved in time $O(n \cdot b)$.*

Proof: The algorithm given above solves Knapsack in time $O(n \cdot b)$. ∎

What happened? The simple algorithm above solves Knapsack in polynomial time hence establishes $P = NP$. Is the entire chapter a fraud? No! Running time is polynomial in the value of b but not in the length of the binary representation of b. At this point the reader should go back to the remark following Theorem 6 of Section 6.5. There we argued that the reduction of 3DM to Knapsack generates problem instances of Knapsack where b is exponential in the size of the instance. Hence the dynamic programming algorithm for Knapsack has running time exponential in the size of the input. Nevertheless, Theorem 1 is interesting from a practical point of view, because it seems to be the case that in "realistic" instances of Knapsack the a_i's and b are bounded by a polynomial in n. The special case of such instances can be solved in polynomial time by dynamic programming. This phenomenon is so interesting that it deserves its own name.

Definition:

a) Let I be an instance of some algorithmic problem, typically I is a set of graphs, integers, sets, $\ldots$. Then $number(I)$ is the largest integer occurring in I.

b) An algorithm for a combinatorial problem is **pseudo-polynomial** if its running time on instance I is polynomial in $size(I)$ and $number(I)$. ∎

We can now rephrase Theorem 1. Knapsack has a pseudo-polynomial algorithm. So have weighted Knapsack and Scheduling Independent Tasks for every fixed number of machines (Exercises 15 and 16). A pseudo-polynomial algorithm is very useful. Whenever the problem instances contain only small numbers, i.e., $number(I) \leq p(size(I))$ for some polynomial p, then a pseudo-polynomial algorithm is indeed an algorithm with polynomial running time. Does every *NP*-complete problem have a pseudo-polynomial algorithm? The answer is "No" provided that $P \neq NP$.

Definition: An *NP*-complete problem is **strongly *NP*-complete** iff it remains *NP*-complete when integers are coded in unary form. ∎

When we introduced *NP*-complete problems we were not very specific about encodings. However we put forth one principle: Integers are coded in binary (or decimal) notation, i.e., the representation of integer n has length $\log n$. If we code integer n in unary representation, i.e., as a sequence of n ones, then the representation of n has length n. A strongly *NP*-complete problem is *NP*-complete even when we use this very redundant encoding of integers. Most of the problems in Section 6.5 are strongly *NP*-complete because these problems do not involve numbers at all or only in an inessential way. Examples are SAT, 3DM, Clique (note that we may assume $k \leq |V|$ w.l.o.g.), VC, DHC, UHC, PCS and $\triangle$TSP (note that all edges have length one or two in the problem instances constructed in the *NP*-completeness of $\triangle$TSP). Strongly *NP*-complete problems do not have pseudo-polynomial algorithms provided that $P \neq NP$.

Theorem 2. *If a strongly NP-complete problem has a pseudo-polynomial algorithm, then $P = NP$.*

Proof: If integers are coded in unary form then $number(I) \leq size(I)$ for all instances I. Hence a pseudo-polynomial algorithm is a polynomial algorithm in the usual sense. ∎

SAT, 3DM and Clique are not very interesting strongly NP-complete problems because numbers do not play a crucial role in these problems. A more interesting example is provided by:

Name:	3-Partition.
Input:	Integers $c_1, \dots, c_{3n}$ such that $B/4 < c_i < B/2$ for all i, where $B = (c_1 + \cdots + c_{3n})/n$.
Question:	Is there a partition $T_1, \dots, T_n$ of $\{1, \dots, 3n\}$ such that $\sum_{j \in T_i} c_j = B$ for all i? ▮

Theorem 3. *3-Partition is strongly NP-complete.*

Proof: The proof is by a lengthy reduction from 3DM and can be found in M.R. Garey/D.S. Johnson: Complexity Results for Multiprocessor Scheduling under Resource Constraints, SICOMP 4 (1975), 397–341. ▮

Note that in any solution of the 3-Partition problem all sets T_i must have cardinality exactly three. Hence finding a solution to Scheduling Independent Tasks with $3n$ jobs of time requirements $c_1, \dots, c_{3n}$, deadline $T = 3$ and $m = n$ machines is equivalent to 3-Partition. Thus SIT is strongly NP-complete.

6.6.2. Branch and Bound

Branch and Bound is another variant of exhaustive search. The branch step corresponds to exhaustive search. However, the feasible solutions generated in the branch step are not searched in arbitrary order. Rather, easily computable bounds on the cost of an optimal solution are used to direct the search for an optimal solution. More concretely, let I_0 be an instance of a minimization problem. In a branch step we generate from I_0 simpler instances $I_1, \dots, I_k$ of the same problem such that:

1) Every feasible solution L of I_i, $1 \le i \le k$, corresponds to a feasible solution $g_i(L)$ of I_0 and $\{g_i(L);\ 1 \le i \le k$ and L is a feasible solution of $I_i\}$ is the set of all feasible solutions of I_0. The g_i's are very often the identity function. The branch step splits problem I_0 into subproblems $I_1, \dots, I_k$.

2) For every I_i, $1 \le i \le k$, one computes a lower bound C_i on the cost of solutions $g_i(L)$ of I_0 where L is a feasible solution of I_i (bound). Then the cost of an optimal solution of I_0 is at least $\max\{C_i;\ 1 \le i \le k\}$. The next branch step is applied to the node with the least C-value.

Branch and Bound steps are iterated until an instance I is obtained such that the feasible solutions for I can be computed directly in little time and such that I has a feasible solution L for which the cost of $g(L)$ is not larger than the C-values of all unexpanded subproblems. Then L is an optimal solution. Branch and Bound is very similar to finding least cost paths in directed graphs. This relation is formulated in Exercise 17.

We take the Traveling Salesman Problem as a concrete example. We need a lower bound on the cost of an optimal tour first. Since every city has to be entered and left on an optimal tour

$$\sum_{i=0}^{n-1}(\min_{j\neq 0} \mathit{dist}(i,j) + \min_{j\neq 0} \mathit{dist}(j,i))/2$$

is a lower bound on the cost of an optimal tour. Consider the following instance on four cities A, B, C, D. Function *dist* is given by matrix

$$I_0 = \begin{array}{c} \\ A \\ B \\ C \\ D \end{array} \begin{array}{c} \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{pmatrix} \infty & 3 & 2 & 7 \\ 4 & \infty & 3 & 6 \\ 1 & 1 & \infty & 3 \\ 1 & 6 & 6 & \infty \end{pmatrix} \end{array}$$

City A has to be left somehow. It is either left on the road to C (the city closer to A) or it is not left on the road to C. We can thus generate the following subproblems I_1 and I_2 from I_0. In I_1 we make sure that road AC is taken by setting $AB \leftarrow AD \leftarrow \infty$. Since C cannot be entered twice on an optimal tour we can also change BC and BD to infinity. In I_2 we make sure that road AC is not taken by changing its length to ∞. We obtain

$$I_1 = \begin{pmatrix} \infty & \infty & 2 & \infty \\ 4 & \infty & \infty & 6 \\ \infty & 1 & \infty & 3 \\ 1 & 6 & \infty & \infty \end{pmatrix}$$

$\mathit{bound} = (2+4+1+1+1+1+2+3)/2 = 7.5$

and

$$I_2 = \begin{pmatrix} \infty & 3 & \infty & 7 \\ 4 & \infty & 3 & 6 \\ 1 & 1 & \infty & 3 \\ 1 & 6 & 6 & \infty \end{pmatrix}$$

$\mathit{bound} = (3+3+1+1+1+1+3+3)/2 = 8.$

Subproblem I_1 has the smaller bound. Thus I_1 is branched in the next step. We can continue from city C to either B or to another city. This step generates the subproblems I_3, where we make sure that CB is taken (and hence CD, AB and DB cannot be taken) and I_4, where we make sure that CB is not taken.

$$I_1 = \begin{pmatrix} \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 6 \\ \infty & 1 & \infty & \infty \\ 1 & \infty & \infty & \infty \end{pmatrix}$$

$bound = (2 + 6 + 1 + 1 + 1 + 1 + 12 + 6)/2 = 10$

and

$$I_2 = \begin{pmatrix} \infty & \infty & 2 & \infty \\ 4 & \infty & 3 & 6 \\ \infty & \infty & \infty & 3 \\ 1 & 6 & 6 & \infty \end{pmatrix}$$

$bound = (2 + 3 + 3 + 1 + 1 + 6 + 2 + 3)/2 = 10.5.$

At this point we know that all solutions of I_3 (I_4) have length at least 10 (10.5) and hence all solutions of I_1 have length at least 10. Thus I_2 is branched in the next step. If we continue in this way we generate the following tree of subproblems. The edges labels in this tree indicate the edge which was either chosen or excluded and the node labels indicate the bound (cf. Figure 8).

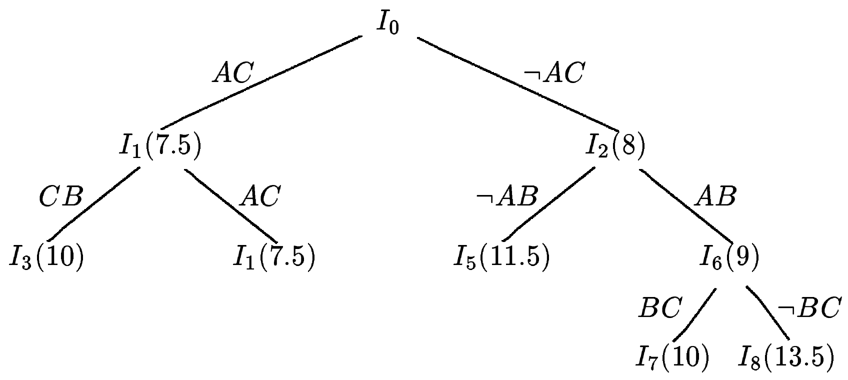

Figure 8. Branch-and-Bound tree

Subproblems I_7 and I_3 allow only one tour each, namely $A \to C \to B \to D \to A$ and $A \to B \to C \to D \to A$ of cost 10 each. Any feasible solution to the other subproblems has cost larger than 10. Thus the two tours given above are the only optimal tours. Our simple branch and bound algorithm can be improved in at least two ways:

a) Better lower bounds direct the search more directly towards optimal solutions. So far we only made use of the fact, that every city has to be entered and left at least once. The additional requirement that every city is entered and left exactly once considerably improves the lower bound in many cases. A solution to that problem (called weighted matching) can be found in polynomial time (cf. 6.7.3); it consists of a set of cycles in the graph.

b) Improving the branch step. So far we included or excluded an arbitrary edge in the branch step. The improved lower bound suggests an improved strategy. Take a cycle of minimum length given in the solution to the matching problem and generate subproblems by excluding one of the edges of the cycle.

Branch and bound techniques are very often a drastic improvement on pure exhaustive search. Nevertheless, it is easy to show that our simple approach still has exponential running time (Exercise 18). Even branching by itself can sometimes yield reasonable algorithms.

Theorem 4. *Let $G = (V, E)$ be an undirected graph and let $sopt$ be the size of an optimal vertex cover. Then a vertex cover of size $sopt$ can be found in time $O(2^{sopt}|E|)$.*

Proof: Note first that at least one of the two endpoints of any edge is in any optimal cover. This suggests the following simple algorithm. Take any edge (v, w) of G and generate two subproblems. In the first subproblem node v is added to the cover and all edges incident to v are deleted from the graph; in the second subproblem node w is added to the cover and all edges incident to w are deleted from the graph. Generate the tree of subproblems breadth-first. Then a tree of depth $sopt$ with 2^{sopt} nodes is generated. In every node we spend time at most $O(|E|)$. ∎

Theorem 4 describes an algorithm whose running time is polynomial in problem size and exponential only in the size of the solution. Hence such an algorithm is very useful if we know in advance that the problem instance at hand has a small solution. A similar phenomenon holds true for cycle cover (Exercise 19).

6.7. Approximation Algorithms

Many NP-complete problems are naturally formulated as optimization problems. In fact, the Traveling Salesman Problem had to be artificially formulated as a language recognition problem. Similarly, Scheduling Independent Tasks can be formulated as an optimization problem, and this even in two ways. Given a set $t_1, \ldots, t_n$ of time requirements of n jobs we can either fix the number m of machines and ask for the minimal deadline or we can fix the deadline and ask for the minimal number of machines. The latter problem is usually called bin packing. In the preceding section we studied dynamic programming and branch-and-bound algorithms for finding optimal solutions. Although these algorithms were more efficient than pure exhaustive search their running time was still exponential. It is then natural to search for approximation algorithms which yield nearly optimal solutions in little time. In Section III.4 we described linear time algorithms for finding nearly optimal binary search trees; in contrast, the best known algorithm for optimum trees had quadratic running time. The savings are even more substantial in the case of NP-complete problems. We describe approximation algorithms for various NP-complete problems. The first example is the Traveling Salesman problem with triangle inequality ($\triangle$TSP). A very simple algorithm always produces a tour of length at most twice the length of the optimum tour. It has running time $O(n^2)$. An improved algorithm with running time $O(n^4)$ always finds a tour of length at most 3/2 the length of the optimum tour. No better approximation algorithm for $\triangle$TSP is known. It is useful to introduce some additional terminology at this point.

Definition: Let (Q, C) be a polynomial bounded minimization problem, in particular $Q \subseteq \Sigma^* \times \Sigma^*$ and $c : \Sigma^* \times \Sigma^* \to \mathbb{N}$ (cf. 6.2). An algorithm A, which computes a mapping $f_A : \Sigma^* \to \Sigma^*$ from problem instances to feasible solutions, is a ***g*-approximate algorithm** if

$$\frac{c(I, f_A(I)) - c(I, f_{opt}(I))}{c(I, f_{opt}(I))} \leq g(c(I, f_{opt}(I)))$$

for every problem instance I. Here $f_{opt} : \Sigma^* \to \Sigma^*$ maps instances to optimal solutions and $g : \mathbb{N} \to \mathbb{R}$ is some function. ▮

In this terminology, we described a $g(x) = (2/x)$-approximate algorithm for optimum search trees in Section III.4 and will describe a $g(x) = 0.5$-approximate algorithm for $\triangle$TSP in 6.7.1.

The situation is even better for Scheduling Independent Tasks when we want to minimize the deadline. We describe an (1/3)-approximate algorithm with running time $O(n \log n)$ first and then improve it to an ϵ-approximate algorithm for any $\epsilon > 0$. The running time of the ϵ-approximate algorithm is $O(n \log n + m^{(m-1)/\epsilon})$, a polynomial in n for any fixed ϵ.

Definition: A **polynomial time approximation scheme** for a minimization problem (Q, c) takes problem instances I and performance guarantees $\epsilon > 0$ and returns ϵ-approximate solutions. For any fixed $\epsilon > 0$ the running time is bounded by a polynomial in instance size. ▮

The algorithm referred to above is a polynomial time approximation scheme for SIT(m), Scheduling Independent Tasks on m machines. Unfortunately, the complexity of the algorithm is not so good as a function of $1/\epsilon$. In 6.7.3 we describe an algorithm for the weighted Knapsack problem whose running time is polynomial in instance size and $1/\epsilon$, a full polynomial approximation scheme.

Definition: A polynomial time approximation scheme is **full** if its running time is bounded by $p(n, 1/\epsilon)$, a polynomial in input size n and performance guarantee ϵ. ▮

6.7.1. Approximation Algorithms for the TSP

We start with a simple and efficient 1-approximate algorithm for the Traveling Salesman Problem with triangle inequality, the once-around-a-least-cost-spanning-tree algorithm.

Let $dist : [0 \dots n-1]^2 \to \mathbb{R}$ be an instance of $\triangle$TSP, i.e., $dist(i,j) = dist(j,i)$ and $dist(i,j) + dist(j,k) \geq dist(i,k)$ for all i, j and k. We use C_{opt} to denote the length of an optimal Traveling Salesman tour. We can define a network $N = (V, E, c)$ from $dist$ in a natural way: $V = \{0, \dots, n-1\}$, $E = V \times V$ and $c(i,j) = dist(i,j)$ for all i, j. Let (V, T) be a least cost spanning tree of N; it can be found by the methods of Section 4.8. Tree (V, T) gives rise to a tour (not necessarily a Traveling Salesman tour) which uses every edge of T twice. We only have to run around the tree once. This tour can be shortened to a Traveling Salesman Tour by introducing shortcuts. The shortcuts do not increase the cost because of the triangle inequality.

Example: Network N of Figure 9 has an optimal Traveling Salesman tour of cost $C_{opt} = 6$. It is shown wiggled. The minimum cost spanning tree right of it has cost 4.

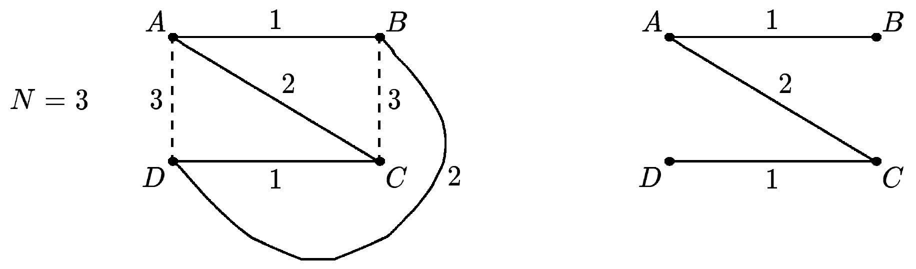

Figure 9. Network and minimum spanning tree

It gives rise to a tour (once around the tree) A, B, A, C, D, C, A of length 8 which can be shortened to a Traveling Salesman Tour A, B, C, D, A of length 7 as shown in Figure 10.

Now we describe the details of the approximation algorithm.

Lemma 1. *Let dist be an instance of $\triangle$TSP, let $N = (V, E, c)$ be the associated network, and let (V, T) be a least cost spanning tree of N. Then*

$$C(T) = \sum_{e \in T} c(e) \le C_{opt},$$

where C_{opt} is the length of an optimal Traveling Salesman Tour.

Proof: An optimal Traveling Salesman tour minus any edge is a spanning tree and has therefore cost at least $C(T)$. ▮

A sequence $S = v_0, v_1, \dots, v_{m-1}, v_0$ is a tour of N if $(v_i, v_{i+1}) \in E$ for all i and $V = \{v_0, v_1, \dots, v_{m-1}\}$, i.e., every node is visited. Its cost is $C(S) = \sum_{i=0}^{m} c(v_i, v_{(i+1) \bmod m})$. It is a Traveling Salesman tour if $m = |V|$.

Let (V, T) be a spanning tree of N. The once-around-the-tree tour S is defined as follows. Let $r \in V$ be arbitrary. If $V = \{r\}$ then S consists of node r only. If $|V| > 1$ then let $r_1, \dots, r_k$ be the neighbour of r in (V, T) and let S_i be the once-around-the-tree tours of the subtrees rooted at r_i. Then $S = rS_1rS_2r \dots rS_kr$. It is easy to see that S can be constructed in time $O(n)$ from (V, T) by depth first search.

Lemma 2. *Let (V, T) be a least cost spanning tree of N and let S be the once-around-the-tree tour of T. Then $C(S) \le 2 \cdot C_{opt}$.*

Proof: $C(S) \le 2 \cdot C(T)$ since every edge of T is used twice in S and $C(T) \le C_{opt}$ by Lemma 1. ▮

Lemma 3. *Let S be a tour of N. Then there is a Traveling Salesman tour of N of cost at most $C(S)$.*

Proof: Let $S = v_0, v_1, \dots, v_{m-1}, v_0$. If $m = |V|$ then there is nothing to be shown. Otherwise there must be a least j, say j_0, such that there is an $i < j$ with $v_i = v_j$. Consider $S' = v_0, v_1, \dots, v_{j-1}, v_{j+1}, \dots, v_{m-1}, v_0$. We have

$$C(S') = C(S) - c(v_{j-1}, v_j) - c(v_j, v_{j+1}) + c(v_{j-1}, v_{j+1}) \le C(S),$$

since *dist* and hence c satisfies the triangle inequality. Thus S' has cost at most $C(S)$ and one node less than S. Repeated application off the construction produces a Traveling Salesman tour. ▮

We summarize:

Theorem 1. *There is an 1-approximate $O(n^2)$ algorithm for $\triangle$TSP; here n is the number of cities.*

Proof: A least cost spanning tree can be found in time $O(n^2)$ by the results of Section 4.8. The rest of the construction takes time $O(n)$. ▮

In the Euclidian case we can do even better. The Euclidian Traveling Salesman problem is as follows. n cities in the plane $\mathbb{R}^2$ are given. The distance between two cities is the Euclidian distance. In Chapter VII we will see that Euclidian least cost spanning trees can be found in time $O(n \log n)$.

Theorem 2. *There is an 1-approximate $O(n \log n)$ algorithm for the Euclidian TSP.*

Proof: Obvious from the discussion above. ▮

Can we improve Theorem 1? Lemma 2 is at the heart of the above construction. It also suggests an improvement. There is another way of visualizing the once-around-the-tree tour. Let (V, T) be a least cost spanning tree. If one draws every edge of T *twice* then we obtain a Eulerian graph, i.e., a graph where every node has even degree. Such a graph has a Eulerian tour, i.e., a tour which uses every edge (of expanded graph) exactly once; cf. Exercise 14. Eulerian tours in Eulerian graphs can be constructed in linear time. We turned (V, T) into a Eulerian graph by doubling every edge and thus doubling the cost. Hence we will obtain a better approximation algorithm for $\triangle$TSP if we find a cheaper way of turning (V, T) into a Eulerian graph. Let V_{odd} be the set of nodes of odd degree in (V, T). One solution is to give exactly the nodes in V_{odd} an additional edge. This is possible since the cardinality of V_{odd} is even. Therefore a least cost matching of V_{odd} will turn (V, T) into a Eulerian graph.

Example: In our previous example, $V_{odd} = \{B, D\}$. Adding edge (B, D) turns the spanning tree into a Eulerian graph. It gives rise to a tour A, B, D, C, A of cost 6, the optimum. ▮

Definition: Let $N = (V, E, c)$ be an undirected network. A **complete matching** is a set $M \subseteq E$ of edges such that $|M| = |V|/2$ and no two edges in M share a common endpoint. The cost of M is $C(m) = \sum_{e \in M} c(e)$. ▮

Lemma 4. *Let dist be an instance of $\triangle$TSP and let $N = (V, E, c)$ be the associated network. Let (V, T) be a least cost spanning tree of N and let V_{odd} be the set of nodes of odd degree in (V, T).*

a) $|N_{odd}|$ *is even.*

b) *The subnetwork of N induced by V_{odd} has a complete matching M of cost $C_{opt}/2$.*

Proof: a) Let $\deg_T(v)$ be the degree of v in (V, T). Then

$$2 \cdot |T| = \sum_{v \in V} \deg_T(v) = \sum_{v \in V_{odd}} \deg_T(v) + \sum_{v \in V - V_{odd}} \deg_T(v).$$

Thus $\sum\{\deg_T(v);\ v \in V_{odd}\}$ is even and hence $|V_{odd}|$ must be even.

b) Let $v_0, v_1, \ldots, v_{n-1}, v_0$ be a Traveling Salesman tour of cost C_{opt}. Furthermore, let $v_{i_1}, v_{i_2}, \ldots, v_{i_{2k}}$ be the nodes in V_{odd}; $|V_{odd}| = 2k$ and $i_1 < i_2 < \ldots < i_{2k}$. Then $M_1 = \{(i_{2j-1}, i_{2j});\ 1 \le j \le k\}$ and $M_2 = \{(i_{2j}, i_{2j+1});\ 1 \le j < k\} \cup \{(i_{2k}, i_1)\}$ are two complete matchings of V_{odd}. Also

$$\begin{aligned} C(M_1) + C(M_2) &= \sum_{j=1}^{2k-1} c(v_{i_j}, v_{i_{j+1}}) + c(v_{i_{2k}}, v_{i_1}) \\ &\le \sum_{l=1}^{n-1} c(v_l, v_{(l+1) \bmod n}) = C_{opt}. \end{aligned}$$

The inequality follows from the triangle inequality. Note that

$$c(v_{i_j}, v_{i_{j+1}}) \le c(v_{i_j}, v_{i_j+1}) + \cdots + c(v_{i_{j+1}-1}, v_{i_{j+1}})$$

by the triangle inequality. Hence $\min(C(M_1), C(M_2)) \le C_{opt}/2$. ∎

Theorem 3. *There is a 0.5-approximate $O(n^3)$ algorithm for $\triangle$TSP; n is the number of cities.*

Proof: A least cost spanning tree (V, T) can be found in time $O(n^2)$. Clearly V_{odd} can then be extracted in time $O(n)$. A least cost matching M of V_{odd} can be found in time $O(n^3)$ (cf. E. Lawler: "Combinatorial Optimization: Networks and Matroids"). Then $(V, T \cup M)$ is a Eulerian graph with $C(T) + C(M) \le 3C_{opt}/2$. Next we construct a Eulerian tour of $(V, T \cup M)$ and shorten it to a Traveling Salesman tour as described in Lemma 3. Altogether, we obtain a tour of length at most $3 \cdot C_{opt}/2$. ∎

The performance bounds given for the once-around-the-tree algorithm and the 0.5-approximate algorithm are best possible in the following sense. One can find a class of problem instances (Exercise 21) where the approximation algorithms produce tours tours with length almost $(1+\epsilon)\cdot C_{opt}$; here $\epsilon = 1$ or $\epsilon = 0.5$, respectively.

Can we find an ϵ-approximate algorithm (with polynomial running time and some $\epsilon > 0$) for every NP-complete problem or are there NP-complete problems which resist even approximate solutions, provided that $P \neq NP$? Unfortunately, the second possibility is true and TSP without triangle inequality provides us with an example.

Theorem 4. *Let $\epsilon > 0$ be arbitrary. If there is an algorithm A having the following properties*

a) *A's running time is polynomially bounded,*

b) *for every symmetric distance matrix $dist : [0\ldots n-1]^2 \to \mathbb{N}$, A constructs a Traveling Salesman tour of length at most $(1+\epsilon)\cdot C_{opt}$, where C_{opt} is the length of the optimal Traveling Salesman tour,*

then $P = NP$.

Proof: Assume that A exists. Let p be a polynomial which bounds the running time of A. We will show that UHC (Undirected Hamiltonian Cycle) is in P. Since UHC is NP-complete this implies $P = NP$. Let $G = (V, E)$ be an instance of UHC with $V = \{v_0, \ldots, v_{n-1}\}$. Define $dist : [0\ldots n-1]^2 \to \mathbb{N}$ by

$$dist(i,j) = \begin{cases} 1 & \text{if } (v_i, v_j) \in E; \\ \lceil \epsilon \cdot n \rceil + 2 & \text{otherwise.} \end{cases}$$

Matrix $dist$ is clearly symmetric. Furthermore, if G has a Hamiltonian cycle then $dist$ has a Traveling Salesman tour of length n and if G does not have a Hamiltonian cycle then every Traveling Salesman tour has length at least $(n-1) + \lceil \epsilon \cdot n \rceil + 2 > (1+\epsilon)\cdot n$ with respect to $dist$. What does algorithm A do on instance $dist$? It constructs a tour of length C. We distinguish two cases.

Case 1: $C > (1+\epsilon)\cdot n$.
Since $C \leq (1+\epsilon)\cdot C_{opt}$ by assumption b) on A, we conclude $C_{opt} > n$. Thus G does not have a Hamiltonian cycle.

Case 2: $C \leq (1+\epsilon)\cdot n$.
Then $C = n$ by the discussion above. Thus G has a Hamiltonian cycle.

This shows that we can use algorithm A for solving UHC. Matrix $dist$ requires $O(n^2 \log(n+2)) = O(n^2 \log n)$ bits to be written down. Since A's running time is bounded by polynomial p, we infer that UHC can be solved in time $O(p(n^2 \log n))$. Hence UHC $\in P$ and therefore $P = NP$. ∎

Theorem 3 shows that NP-complete problems can behave dramatically different with respect to approximation. We will see more of this in sections to come. It is not known whether the existence of such an algorithm is excluded by the assumption $P \neq NP$.

6.7.2. Approximation Schemes

With respect to the Scheduling Independent Tasks (SIT) optimization problem the situation is better. We are given the time requirements $t_1, \ldots, t_n$ of a set of n jobs and a number m of machines and are asked to find a schedule $S : [1 \ldots n] \to [1 \ldots m]$ which minimizes the finishing time $T = \max_j \sum_{S(i)=j} t_i$. Throughout this section we use T_{opt} to denote the finishing time of an optimal schedule. We describe an (1/3)-approximate algorithm first and then refine it to an approximation scheme for SIT(m). In SIT(m) the number of machines is not an input to the algorithm but fixed in advance.

The (1/3)-approximate algorithm is based on a very simple idea: Schedule long jobs first and always schedule a job on a machine which was used least so far. The details are as follows.

Assume w.l.o.g. that $t_1 \geq t_2 \geq \cdots \geq t_n$. In fact, reordering the jobs takes time $O(n \log n)$ and is the most time consuming part of the algorithm *Construct* $S : [1 \ldots n] \to [1 \ldots m]$ formulated in Program 6.

```
$(T_1, \ldots, T_m) \leftarrow (0, \ldots, 0)$;
**co** $T_j$ time units are used up on machine $j$ so far **oc**
**for** $i$ **from** 1 **to** $n$
**do** let $j$ be such that $T_j = \min(T_1, \ldots, T_m)$;
    $S[i] \leftarrow j$;
    $T_j \leftarrow T_j + t_i$
**od**.
```

Program 6

We refer to the schedule constructed by this algorithm as the Longest Processing Time (LPT) schedule S_{LPT}. Its finishing time is denoted by T_{LPT}. If the T_i's are kept in a heap then one iteration of the loop takes time $O(\log m)$, for a total running time of $O(n \log m)$.

Theorem 5. *The LPT algorithm is a* $1/3 - 1/(3m)$ *approximate algorithm for Scheduling Independent Tasks.*

Proof: (Indirect.) Assume otherwise. Let I be an instance of SIT with a minimal number of jobs such that $(T_{LPT}(I) - T_{opt}(I))/T_{opt}(I) > 1/3 - 1/(3m)$. Here $T_{opt}(I)$ is the finishing time of an optimal schedule $S_{opt} : [1 \ldots n] \to [1 \ldots m]$ and $T_{LPT}(I)$ is the finishing time of the LPT schedule S_{LPT}.

Lemma 5. *If $S_{LPT}(n) = j$ then $\sum_{S_{LPT}(i)=j} t_i = T_{LPT}(I)$, i.e., job n finishes at time $T_{LPT}(I)$.*

Proof: Assume otherwise. Then the LPT algorithm also constructs a schedule of length $T_{LPT}(I)$ for jobs $t_1, \ldots, t_{n-1}$. Denote the problem instance with time requirements $t_1, \ldots, t_{n-1}$. Denote the problem instance with requirements $t_1, \ldots, t_{n-1}$ by I'. Then $T_{LPT}(I') = L_{LPT}(I)$. Also $T_{opt}(I') \leq T_{opt}(I)$ since I' is a "subinstance" of I. Hence

$$\begin{aligned} T_{LPT}(I') - T_{opt}(I') &\geq T_{LPT}(I) - T_{opt}(I) \\ &\geq [1/3 - 1/(3m)] \cdot T_{opt}(I) \\ &\geq [1/3 - 1/(3m)] \cdot T_{opt}(I'). \end{aligned}$$

Thus I' is a counterexample containing one fewer job than I, a contradiction to the choice of I. ∎

We will next show that the LPT algorithm constructs a bad schedule only if $T_{opt}(I)$ is small.

Lemma 6. *$T_{opt}(I) < 3 \cdot t_n$.*

Proof: Consider the state $(T_1, \ldots, T_m)$ of the LPT algorithm just prior to scheduling job t_n. Let $T_k = \min(T_1, \ldots, T_m)$. Then t_n is scheduled on machine k and $T_k + t_n = T_{LPT}(I)$ by Lemma 1. Hence $t_1 + \cdots + t_{n-1} \geq m(T_{LPT}(I) - t_n)$ or

$$t_1 + \cdots + t_n \geq m \cdot T_{LPT}(I) - (m-1) \cdot t_n.$$

Next note that $T_{opt}(I) \geq (t_1 + \cdots + t_n)/m$, since every job has to be executed on some machine and hence

$$((m-1)/m) \cdot t_n \geq T_{LPT}(I) - T_{opt}(I) > [1/3 - 1/(3m)] \cdot T_{opt}(I).$$

The last inequality follows from the fact that I is a counterexample. Thus $3 \cdot t_n > T_{opt}(I)$. ∎

In Lemma 3 we complete the contradiction and show that the LPT algorithm indeed constructs optimal schedules if $T_{opt}(I)$ is small.

Lemma 7. *If $T_{opt}(I) < 3 \cdot t_n$ then $T_{LPT}(I) = T_{opt}(I)$.*

Proof: If $S_{opt} = S_{LPT}$ then there is nothing to show. Assume otherwise. Since $t_1 \geq t_2 \geq \cdots \geq t_n$ no more than two jobs can be scheduled on any machine. Hence $n \leq 2m$. We may assume $n = 2m$ by adding jobs $n+1, \ldots, 2m$ with time requirements $t_{n+1} = \ldots = t_{2m} = 0$. The LPT algorithm schedules jobs J and $2m+1-j$ on machine j, $1 \leq j \leq m$. Let j be maximal such that $T_{LPT}(I) =$

$t_j + t_{2m+1-j}$. Construct a graph G with nodes $v = [1\,..\,n]$ as follows. Draw a red edge (i,k) if $S_{opt}(i) = S_{opt}(k)$ and draw a green edge (i,k) if $S_{LPT}(i) = S_{LPT}(k)$, i.e., if $i+k = 2m+1$. Since every node has degree exactly two in graph G (recall that exactly two jobs are scheduled on any machine by S_{opt} and S_{LPT}), the connected components of G are simple cycles. Consider the component containing node j. It contains nodes $j_1, \ldots, j_l, 2m+1-j_1, \ldots, 2m+1-j_l$ for some $j_1, \ldots, j_l \in [1\,..\,m]$. Since the red edges form a matching on these nodes there must be a red edge (i,k) such that $i \le j$ and $k \le 2m+1-j$. Hence $T_{opt}(I) \ge t_i + t_k \ge t_j + t_{2m+1-j} = L_{LPT}(I)$. ∎

Lemma 1, 2 and 3 imply that I does not exist. ∎

In Exercise 25 it is shown that the worst case performance of the LPT algorithm is indeed $1/3 - 1/(3m)$. The LPT algorithm processes long jobs first. This principle works quite well for a number of problems, cf. Exercise 22 on Bin Packing and Exercise 26 on the Knapsack problem. This suggests that we can do even better if we are very careful with the longest jobs, say if we schedule the longest k jobs optimally. This leads to the $LPT(k)$-algorithm: Schedule the k longest jobs optimally (in time m^k by branch and bound), then continue with LPT.

Theorem 6. *The $LPT(k)$ algorithm always produces an $(m-1)/(k+1)$-approximate schedule.*

Proof: Let $t_1 \ge t_2 \ge \cdots \ge t_n$ be the time requirements of a set of n jobs and let $T_{LPT(k)}$ resp. T_{opt} be the finishing time of the $LPT(k)$ resp. optimal schedule. We have to show that

$$T_{LPT(k)} - T_{opt} \le ((m-1)/(k+1)) \cdot T_{opt}.$$

Let t be the length of an optimal schedule for jobs $1, \ldots, k$. If $T_{LPT(K)} = t$ then the claim is obviously true. So assume otherwise. Let $j > k$ be a job with finishing time $T_{LPT(k)}$. Then all processors are busy up to time $T_{LPT(k)} - t_j$ and hence $(t_1 + \cdots + t_{j-1})/m \ge T_{LPT(k)} - t_j$. Also $T_{opt} \ge (t_1 + \cdots + t_j)/m$ and hence

$$\begin{aligned} T_{LPT(k)} - T_{opt} &\le ((m-1)/m) \cdot t_j \le ((m-1)/m) \cdot t_{k+1} \\ &\le (1-1/m)(m/(k+1)) \cdot T_{opt} \le ((m-1)/(k+1)) \cdot T_{opt}. \end{aligned}$$

The third inequality follows from $T_{opt} \ge (t_1 + \ldots t_{k+1})/m \ge (k+1) \cdot t_{k+1}/m$. ∎

At this point we almost have an approximation scheme for SIT(m), Scheduling Independent Tasks on m machines. Let $\epsilon > 0$ be arbitrary. Choose k such that $(m-1)/(k+1) \le \epsilon$; $k = \lfloor (m-1)/\epsilon \rfloor$ will certainly do. Then use the $LPT(k)$ algorithm to construct an ϵ-approximate schedule. The running time of $LPT(k)$ is $O(m^k)$ for finding an optimal schedule for the k longest jobs plus $O(n \log n)$ for sorting and scheduling the $n-k$ remaining jobs. Thus total running time is $O(n \log n + m^k) = O(n \log n + m^{m/\epsilon})$. This is polynomial in n for every fixed ϵ.

Theorem 7. *There is a polynomial approximation scheme for SIT(m) which constructs ϵ-approximate solutions in time $O(n \log n + m^{m/\epsilon})$ for any $\epsilon > 0$.*

Proof: By the preceding discussion. ∎

Can we always turn an ϵ-approximate algorithm for some ϵ into a polynomial approximation scheme as we did for SIT(m)? Note that $\triangle$TSP is not a counterexample; there we just do not know how to do better than 0.5-approximate solutions. There is no reason to believe that 0.5 is a boundary which will exist forever. However, our second *NP*-complete scheduling problem can serve as an example.

Theorem 8.

a) *There is a 1-approximate linear time algorithm for Precedence Constrained Scheduling (PCS).*
b) *If there is an $(1/4)$-approximate polynomial time algorithm for PCS then $P = NP$.*

Proof: a) Consider any instance of PCS. Let n be the number of jobs, m the number of machines and R be the precedence relation on jobs. Let T_{opt} be the finishing time of an optimal schedule. For any job $i \in [1..n]$ define its depth by $depth(i) = 1 + \max\{depth(j);\ (j,i) \in R\}$. As always, the maximum of the empty set is defined to be zero. Then $T_{opt} \geq LB := \max(\lceil n/m \rceil, maxdepth)$ where $maxdepth = \max\{depth(i);\ 1 \leq i \leq n\}$. It remains to be shown that we can always schedule all jobs in $2 \cdot LB$ time units. For d, $1 \leq d \leq maxdepth$, let $L_d = \{i;\ depth(i) = d\}$ be the jobs of depth d. We schedule the jobs in L_d for time units $\sum_{j<d} \lceil |L_j|/m \rceil + 1, \ldots, \sum_{j \leq d} \lceil |L_j|/m \rceil$. Then the maximal time unit used is

$$\sum_{j \leq maxdepth} \lceil |L_j|/m \rceil \leq \sum_j (|L_j|/m + 1) = n/m + maxdepth \leq 2 \cdot LB.$$

b) Recall the *NP*-completeness proof of PCS (Theorem 6.5.7). The instance of PCS constructed in the reduction Clique $\leq$ PCS had either finishing time three or four. More precisely, they had finishing time three iff we start with a graph which has a clique and finishing time four otherwise. Suppose now that we have a $(1/4)$-approximate algorithm for PCS. Then we can use it to solve Clique because the approximation algorithm must be an exact algorithm on the instances constructed by the reduction. ∎

6.7.3. Full Approximation Schemes

We will now turn to the ultimate in the field of approximation algorithms: full approximation schemes. We will also get to know a very important technique for approximation algorithms: scaling. Scaling is applicable to many problems; it is particularly appropriate when we have a pseudo-polynomial algorithm available.

In Section 6.6.1 we described a pseudo-polynomial algorithm for the Knapsack Problem. This was extended to the Weighted Knapsack Problem in Exercise 15. Let $c_1, \ldots, c_n, w_1, \ldots, w_n, K$ be instances I of the weighted Knapsack problem. Exercise 15 shows how to compute $C_{opt}(I) = \max\{\sum_i c_i \cdot x_i;\ x_i \in \{0,1\}, \sum x_i \cdot w_i \le K\}$ in time $O(n \cdot C_{opt}) = O(n^2 \max c_i)$.

Let S be any integer. We scale the costs by S and obtain a scaled instance $I_S = (c_1/S, \ldots, c_n/S, w_1, \ldots, w_n, K)$. What does scaling do for us? First of all, the scaled instance can be solved by our pseudo-polynomial algorithm in time $O(n \cdot C_{opt}(I_S)) = O(n \cdot C_{opt}(I)/S)$, which can be made arbitrarily small by choosing S large enough. Secondly the optimal solution for the scaled instance is a very good solution to the original problem. More precisely, let $(x_1, \ldots, x_n) \in \{0,1\}^n$ be such that $C_{opt}(I_S) = \sum_i x_i \cdot \lfloor c_i/S \rfloor$ and $\sum_i x_i \cdot w_i \le K$ and let $(y_1, \ldots, y_n) \in \{0,1\}^n$ be such that $C_{opt}(I) = \sum y_i \cdot c_i$ and $\sum y_i \cdot w_i \le K$. Then

$$\begin{aligned}
C_{opt}(I) &= \sum y_i \cdot c_i \\
&\ge \sum x_i \cdot c_i && \text{, since } (y_1, \ldots, y_n) \text{ is optimal for } I \\
&= S \sum x_i \cdot c_i/S \\
&\ge S \sum x_i \cdot \lfloor c_i/S \rfloor \\
&\ge S \sum y_i \cdot \lfloor c_i/S \rfloor && \text{, since } (x_1, \ldots, x_n) \text{ is optimal for } I_S \\
&\ge S \sum u_i \cdot (c_i/S - 1) \\
&\ge C_{opt}(I) - n \cdot S.
\end{aligned}$$

Thus $(C_{opt}(I) - \sum x_i \cdot c_i)/C_{opt}(I) \le n \cdot S/C_{opt}(I)$. We summarize in

Lemma 8. *Let S be any integer. Then in time $O(n \cdot C_{opt}(I)/S)$ we can compute an $n \cdot S/C_{opt}(I)$-approximate solution namely $\sum x_i \cdot c_i$ where $(x_1, \ldots, x_n)$ is the optimal solution vector for the scaled problem.*

We still have to choose S appropriately. Let $\epsilon > 0$ arbitrary. Setting $S = \epsilon \cdot C_{opt}(I)/n$ we obtain an ϵ-approximate algorithm with running time $O(n^2/\epsilon)$. There is a catch, however; we do not know $C_{opt}(I)$. After all the purpose of the algorithm is to approximate $C_{opt}(I)$. The way out is to use a reasonable approximation for $C_{opt}(I)$ in the definition of S.

In a first attempt we use $S = \lfloor \epsilon \max c_i/n \rfloor \le \epsilon \cdot C_{opt}(I)/n$. Then we obtain an ϵ-approximate algorithm with running time $O(n \cdot C_{opt}(I)/S) = O(n^2 \cdot (\max c_i)/S) = O(n^3/\epsilon)$ which is definitely worse than what we would get if we chose S optimally. The reason is of course that we used only very weak bounds on $C_{opt}(I)$, namely $\max c_i \le C_{opt}(I) \le n \max c_i$. A much better bound is provided by Exercise 26. There it is shown that C with $C_{opt}(I) \ge C \ge C_{opt}(I)/2$ can be computed in time $O(n \log n)$. Setting $S = \lfloor \epsilon \cdot C/n \rfloor$ we obtain an ϵ-approximate algorithm with running time $O(n \cdot C_{opt}/S) = O(n^2/\epsilon)$.

Theorem 9. *The Weighted Knapsack Problem has a full polynomial approximation scheme; more precisely, an ϵ-approximate solution can be computed in time $O(n^2/\epsilon)$ for any $\epsilon > 0$.*

Proof: By the discussion above. ∎

The technique described above is not only applicable to the weighted Knapsack problem. We can rather state quite generally:

Scaling + Pseudo-polynomial algorithm $\Rightarrow$ good approximation algorithm.

See Exercises 27 and 28. In fact, the connection between pseudo-polynomial algorithms and full approximation schemes is even stronger. The existence of a full approximation scheme implies the existence of a pseudo-polynomial algorithm.

Theorem 10. *Let (Q, c) be a polynomially bounded minimization problem. For instance x, let $optval(x)$ be the cost of an optimal solution for x. If (Q, c) has a full polynomial approximation scheme and $optval(x)$ is polynomially bounded in the $size(x)$ of x and the largest integer $number(x)$ appearing in x then (Q, c) has a pseudo-polynomial algorithm.*

Proof: Let A be a full approximation scheme for (Q, c). Choose $\epsilon = 1/(p(size(x), number(x))+$ ∎ 1) where polynomial p is such that $optval(x) \leq p(size(x), number(x))$ for all instances x. Let y be the solution produced by A on inputs x and ϵ. Then

$$\begin{aligned} 0 &\leq c(x,y) - optval(x) && \text{(by definition of } optval(x)) \\ &\leq \epsilon \cdot optval(x) && \text{(since } A \text{ is an approximation scheme)} \\ &< 1 && \text{(by definition of } \epsilon). \end{aligned}$$

Thus $c(x, y) = optval(x)$, y is an optimal solution for x. Furthermore, the running time of A on inputs x and ϵ is bounded by a polynomial in $size(x)$ and $1/\epsilon$ which in turn is a polynomial in $size(x)$ and $number(x)$. So (Q, c) has a pseudo-polynomial algorithm. ∎

Theorem 9 has strong implications for strongly *NP*-complete problems. They cannot have a full polynomial approximation scheme.

Theorem 11. *SIT does not have a full polynomial approximation scheme (provided that $P \neq NP$).*

Proof: SIT satisfies the assumption of Theorem 9. Note that the total time requirement of all jobs is certainly an upper bound for the finishing time. Also SIT is strongly *NP*-complete (Theorem 6.6.3) and therefore does not have a pseudo-polynomial algorithm (provided that $P \neq NP$, Theorem 6.6.2). ∎

6.8. The Landscape of Complexity Classes

The main concern of this chapter were NP-complete problems and how to cope with them. In this section we go beyond NP-completeness and relate the complexity classes P and NP to various other classes. Moreover, we list a number of problems and give their status with respect to these classes.

The following diagram shows a relevant part of the landscape of complexity classes where "$A \to B$" means "$A \subseteq B$".

$$\begin{tikzcd}
 & & RP \arrow[rr] \arrow[dr] & & NP = \Sigma_1 \arrow[r] & \Sigma_2 \arrow[r] & \Sigma_3 \arrow[r] & \Sigma_4 \arrow[r] & \cdots \arrow[dr] & \\
P \arrow[r] & LVP = RP \cap co-RP \arrow[ur] \arrow[dr] & & BPP \arrow[ur] \arrow[dr] & & & & & & PSPACE \\
 & & co-RP \arrow[rr] \arrow[ur] & & co-NP = \Pi_1 \arrow[r] & \Pi_2 \arrow[r] & \Pi_3 \arrow[r] & \Pi_4 \arrow[r] & \cdots \arrow[ur] &
\end{tikzcd}$$

The classes P and NP were defined above. The class $PSPACE$ is the set of languages which can be recognized by polynomially space bounded Turing machines. For class $PSPACE$ it is irrelevant whether deterministic or nondeterministic machines are considered (Savitch (70)). The classes $\Sigma_1, \Sigma_2, \Sigma_3, \ldots$ and $\Pi_1, \Pi_2, \Pi_3, \ldots$ form the polynomial hierarchy (Stockmeyer/Meyer (73)). These classes are defined as follows:

$$\begin{aligned}
\Sigma_i = \{L;\ & L \subseteq \Gamma^* \text{ for some finite alphabet } \Gamma \text{ and there is some polynomial} \\
& \text{time computable predicate } p(x, y_1, \ldots, y_i) \text{ and a polynomial } q \\
& \text{such that for all } x \in \Gamma^*\text{:} \\
& x \in L \text{ iff } \exists y_1 \forall y_2 \exists y_3 \ \ldots\ : \\
& \qquad (|y_1| \le q(|x|) \text{ and } \ldots (|y_i| \le q(|x|) \text{ and } p(x, y_1, \ldots, y_i))\} \\
\Pi_i = co&- \Sigma_i
\end{aligned}$$

where $co - C = \{\Gamma^* - L;\ L \in C \text{ and } \Gamma \text{ a finite alphabet}\}$ for any class C of languages. The languages in Σ_i are defined by formulae with i alternations of quantifiers starting with an existential quantifier. The quantifiers range over all strings whose length is polynomially bounded in the length of x. Similarly, the languages in Π_i are defined by formulae with i alternations of quantifiers starting with an universal quantifier. In Theorem 1 of Section 6.2 we proved $NP = \Sigma_1$. The inclusions $\Sigma_i \subseteq \Sigma_{i+1}$, $\Sigma_i \subseteq \Pi_{i+1}$, $\Pi_i \subseteq \Sigma_{i+1}$, and $\Sigma_i \subseteq PSPACE$ for all i are obvious. It is not known whether the polynomial hierarchy is infinite or whether it collapses at some finite level.

The remaining classes are defined by probabilistic machines (cf. Section I.2). We have

$$\begin{aligned}
BPP = \{L;\ & L \subseteq \Gamma^* \text{ for some finite alphabet } \Gamma \text{ and there is a polynomial} \\
& \text{computable predicate } p(x, y) \text{ and a polynomial } q \text{ such that} \\
& x \in L \Rightarrow |\{y;\ |y| = q(|x|) \text{ and } p(x, y)\}| \ge \tfrac{3}{4}|\Gamma|^{q(|x|)} \\
& x \notin L \Rightarrow |\{y;\ |y| = q(|x|) \text{ and } \neg p(x, y)\}| \ge \tfrac{3}{4}|\Gamma|^{q(|x|)}\}.
\end{aligned}$$

Thus a language is in BPP (bounded probability of error) if it can be recognized by a probabilistic machine whose worst-case running time is polynomially bounded and whose answers are reliable with probability 3/4. Algorithms in this class are also referred to as polynomially bounded Monte Carlo algorithms. In the definition of RP (random P) we allow only one-sided error.

$$\begin{aligned} RP = \{L;\ & L \subseteq \Gamma^* \text{ for some finite alphabet } \Gamma \text{ and there is a polynomial time} \\ & \text{computable predicate } p(x,y) \text{ and a polynomial } q \text{ such that} \\ & x \in L \Rightarrow |\{y;\ |y| = q(|x|) \text{ and } p(x,y)\}| \geq \tfrac{3}{4}|\Gamma|^{q(|x|)} \\ & x \notin L \Rightarrow |\{y;\ |y| = q(|x|) \text{ and } \neg p(x,y)\}| = |\Gamma|^{q(|x|)}\}. \end{aligned}$$

Thus a language L is in RP if a string $x \in L$ is accepted with probability at least 3/4 and a string outside L is never accepted. In other words, there is a probabilistic algorithm whose worst-case running time is bounded by a polynomial. Moreover, its "yes-answers" are completely reliable but its "no-answers" are not, i.e., there is a possibility that $x \in L$ and the algorithm outputs "no". However, the probability that en element $x \in L$ is declared to be outside of L is at most 1/4. The inclusions $RP \subseteq BPP$, $co{-}RP \subseteq BPP$, $RP \subseteq NP$ and $co{-}RP \subseteq co{-}NP$ are obvious. Again it is not known whether any of these inclusions are proper. Incidentally, the quantity 3/4 in the definition of RP and BPP is not the only choice possible. Any real a with $1/2 < a < 1$ could be taken instead of 3/4. This follows immediately from the results of Section I.2. Also, we might require that only the expected running time is bounded by a polynomial without any change in the language classes. Finally, in the definition of class LVP (Las Vegas P) we allow no probability of error and we just require that the expected running time is polynomially bounded.

$$\begin{aligned} LVP = \{L;\ & L \subseteq \Gamma^* \text{ for some finite alphabet } \Gamma \text{ and the characteristic function} \\ & \text{of } L \text{ is computed by a Las Vegas algorithm (cf. Section I.2)} \\ & \text{whose running time is bounded by a polynomial}\} \end{aligned}$$

The following theorem states the two non-trivial inclusions in the diagram above.

Theorem 1.
a) $LVP = RP \cap co - RP$.
b) $BPP \subseteq \Sigma_2$.

Proof: a) We show $LVP \subseteq RP$, $LVP \subseteq co - RP$ and $RP \cap co - RP \subseteq LVP$. Consider $LVP \subseteq RP$ first. If $L \subseteq LVP$ then we can recognize L by an algorithm A whose expected running time is bounded by a polynomial, say q, and whose outputs are completely reliable. Consider the following algorithm A': On input x it runs algorithm A for at most $10 \cdot q(|x|)$ steps. If A has terminated before that time (and this occurs with probability exceeding 9/10) then A' outputs whatever A outputs.

If A has not terminated within $10 \cdot q(|x|)$ steps then A' outputs no. In this way the yes-answers of A' are completely reliable but no-answers are not. However, the probability of a no-answer for $x \in L$ is at most $1/10$. Thus $L \in RP$ and hence $LVP \subseteq RP$.

A similar argument shows $LVP \subseteq co - RP$. (A' outputs yes if the clock is exhausted).

Next we turn to the inclusion $RP \cap co - RP \subseteq LVP$. Let $L \in RP \cap co - RP$. Then we have probabilistic algorithms A_1 and A_2 such that

1) the worst-case running time of A_1 and A_2 is bounded by a polynomial, say q,
2) the yes-answers of A_1 and the no-answers of A_2 are reliable;
3) the probability that A_1 gives a wrong answer is at most $1/4$ and the probability that A_2 gives a wrong answer is at most $1/4$.

We show how to recognize L with zero probability of error and polynomial average running time. Let x be arbitrary. Consider the following experiment.

Choose a random y with $|y| = q(|x|)$ and compute $A_1(x, y)$ and $A_2(x, y)$. This takes time $O(q(|x|))$. If $A_1(x, y) =$ "yes" then x belongs to L and if $A_2(x, y) =$ "no" then x does not belong to L. (Note that it is impossible that $A_1(x, y) =$ "yes" and $A_2(x, y) =$ "no" occur simultaneously.) In either case we have correctly decided the membership of x with respect to L. The final case that is to be considered is $A_1(x, y) =$ "no" and $A_2(x, y) =$ "yes". Then $x \in L$ and $x \notin L$ are both conceivable. If $x \in L$ then the answer $A_1(x, y) =$ "no" is wrong and this event has probability at most $1/4$ and if $x \notin L$ then the answer $A_2(x, y) =$ "yes" is wrong and this event has probability at most $1/4$. In either case, we see that the probability that we cannot decide the membership of x in L is at most $1/4$. Hence the probability that i experiments are needed to decide the membership is at most $1/4^{i-1}$ and hence the expected number of experiments needed is $\sum_{i \geq 0} i/4^{i-1} = O(1)$. This proves $L \in LVP$.

b) A proof can be found in Sipser (83) or Lautemann (84). ∎

None of the inclusions in the diagram above are known to be proper. However, the inclusions are not independent. For example, it is known that $RP = NP$ implies $\Sigma_2 = \Sigma_3 = \Sigma_4 \ldots$ (Karp/Lipton (80)). We will now turn to problems and give a list of problems and their status with respect to these classes.

Name:	Quantified Boolean Formulae (QBF).
Input:	A quantified boolean formula of the form $Q_1 x_1 Q_2 x_2 \ldots Q_m x_m E(x_1, \ldots, x_m, y_1, y_n)$ where E is a boolean expression over operators **and**, **or**, **not**, $x_1, \ldots, x_m, y_1, \ldots, y_n$ are boolean variables, and $Q_i \in \{\exists, \forall\}$ is a quantifier.
Question:	"Yes" if the formula is satisfiable, and "no", otherwise.

Theorem 2. *(Stockmeyer/Meyer (73)): QBF is PSPACE-complete.*

For the classes of the polynomial hierarchy, no complete problems are known except for classes Σ_1 (*NP*-complete) and Σ_2.

Name:	Uniquely Optimal Traveling Salesman Tour (UOTS).
Input:	An instance of the traveling salesman problem.
Question:	"Yes", if there is a unique optimal tour, and "no", otherwise

Theorem 3. *(Papadimitriou (82)): UOTS is Σ_2-complete.* ∎

Another example of a Σ_2-complete problem is given in Huynh (82): the inequivalence problem for context-free grammars with one letter terminal alphabet.

For the probabilistic classes *LVP*, *RP*, $co - RP$, *BPP* there are no known complete problems. In fact, Adleman (78) gives strong reasons that these classes cannot contain problems.

Name:	PRIMES.
Input:	An integer n in binary notation.
Question:	Yes, if n is a prime, and no, otherwise.

Theorem 4.

a) *PRIMES* $\in co - NP$;

b) *PRIMES* $\in NP$;

c) *PRIMES* $\in co - RP$.

Proof: a) is trivial, b) can be found in Pratt (75) and c) can be found in Solovay/Strassen (77). We give a very brief sketch of part c). For integers p and q which are relatively prime the Legendre symbol (q/p) is defined by

$$(q/p) = \begin{cases} 1 & \text{if } q \text{ is a quadratic residue mod } p\text{, i.e., } q = x^2 \bmod p \text{ for some } x; \\ -1 & \text{otherwise.} \end{cases}$$

There is a well-known efficient algorithm for computing the Legendre symbol, it is based on the law of reciprocity, namely $(q/p) = -(p/q)$ if $p = q = 3 \bmod 4$ and $(q/p) = (p/q)$ otherwise. Furthermore, when $q > p$ and hence $q = m \cdot p + r$ for some $r < p$ then $(q/p) = (r/p)$. These three relations immediately suggest a polynomial time algorithm for computing the Legendre symbol which is similar to the Euclidian algorithm for greatest common divisor.

The important observation is now that if p is prime then $(a/p) = a^{(p-1)/2} \bmod p$ for all a, $1 \le a \le p-1$. However, if p is not prime then the above relation holds for at most half of the a's which are relatively prime to p. This follows from the fact that the set of a's for which the above relation holds is a proper subgroup of the multiplicative group of integers which are relatively prime to p.

This suggests the following algorithm. In order to check the primality of p select a random integer a, $1 \le a < p$. If a and p are not relatively prime (a gcd calculation) then p is composite. Otherwise we check the relation $(a/p) = a^{(p-1)/2} \bmod p$; note that $a^{(p-1)/2} \bmod p$ can be computed by repeated squaring (cf. Section I.1). If the inequality does not hold then p is composite. If the equality holds then p may be prime or composite. However, if p is composite then the equality holds with probability at most 1/2. Repeating the experiment for several a's reduces the probability of error below 1/4. Hence COMPOSITE $\in RP$ or PRIMES $\in co - RP$. ∎

Another interesting example of a problem in $co - RP$ is given by

Name: Checking Polynomial Identities (CPI).
Input: An identity of the form $Q = P$ where Q and P are expressions formed from real variables $x_1, x_2, \ldots$ using operators $+$, $-$, and $\cdot$.
Question: "Yes", if the identity is true, and "no", otherwise.

Theorem 5. *(Schwartz (80)): CPI $\in co - RP$.* ∎

We end this section by listing problems which were recently shown to be in P.

Name: Linear Programming (LP).
Input: An integer matrix A and an integer vector b.
Question: "Yes", if there is a *real* vector x with $A \cdot x \leq b$.

Theorem 6. *(Khachiyan (79)): LP $\in P$.* ∎

If we require that the solution vector is to be integral then we face the integer programming problem. It is NP-complete as was shown in Section 5. However, for every fixed dimension there is a polynomial algorithm.

Name: Integer Programming in d-dimensional space (IPd).
Input: An n by d integer matrix A and an integer vector b.
Question: "Yes", if there is a d-dimensional integer vector x with $A \cdot x \leq b$, and "no", otherwise.

Theorem 7. *(Lentstra (84)): IP$d \in P$ for every fixed d.* ∎

An improved algorithm for IPd can be found in Kannan (83). Finally, we want to mention the graph isomorphism problem.

Name: Graph Isomorphism (GI).
Input: Undirected Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.
Question: "Yes", if G_1 and G_2 are isomorphic, and "no", otherwise. G_1 and G_2 are isomorphic if there is a bijection $a : V_1 \to V_2$ with $(v, w) \in E$, iff $(a(v), a(w)) \in E_2$.

Clearly, GI $\in NP$. It is not known whether GI is NP-complete. Various special cases of GI have been shown to be in P, e.g., graphs of bounded valence (Luks (80)), and k-contractible graphs (Miller (83)).

6.9. Exercises

1) Let $L = \{w¢v;\ w, v \in \{0,1\}^* \text{ and } w \neq v\}$. Describe deterministic and nondeterministic TM's which accept L. Running time?

2) Show that $T(n) = n$, $T(n) = n^2$, $T(n) = n \cdot \lfloor \log n \rfloor$, $T(n) = 2^n$ are step functions.

3) Consider RAM's where the only arithmetic operations are addition and subtraction. Show that Theorem 1 of Section 6.1 is true for unit-cost RAM's.

4) The Clique optimization problem is to find a larges clique in an undirected graph. Prove Lemmas 2 and 3 of 6.2 for Clique.

5) Same as exercise 4) but for Knapsack problem.

6) Show that SAT(2) is in P. [Hint: Let α be a formula with at most 2 literals per clause. Let $x_1, \ldots, x_n$ be the variables occurring in α. Construct a directed graph with nodes $x_1, \ldots, x_n, \bar{x}_1, \ldots, \bar{x}_n$ as follows. If $y_1 \vee y_2$ is a clause of α then add directed edges $\bar{y}_1 \to y_2$ (Interpretation: If $\bar{y}_1$ is true then y_2 must be true) and $\bar{y}_2 \to y_1$. Then α is satisfiable iff there is no cycle of the form $x_i \to \cdots \to \bar{x}_i \to \cdots \to x_i$. Cycles of this form can be detected by determining the strongly connected components of the graph.]

7) Show that the weighted Knapsack problem is *NP*-complete.

Name:	Weighted Knapsack.
Input:	Integers $w_1, \ldots, w_n$ (the weights), $c_1, \ldots, c_n$ (the costs), K, C.
Question:	Are there $x_i \in \{0,1\}$, $1 \leq i \leq n$ such that $\sum w_i \cdot x_i \leq K$ and $\sum c_i \cdot x_i \geq C$?

8) Show that partition is *NP*-complete.

Name:	Partition.
Input:	Integers $a_1, \ldots, a_n$.
Question:	Is there $I \subseteq [1\,..\,n]$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$?

9) Show that Chromatic Number is *NP*-complete.

Name:	Chromatic Number.
Input:	Undirected graph $G = (V, E)$, integer k.
Question:	Is there a node coloring with at most k colors, i.e., is there a mapping $c : V \to [1\,..\,k]$ such that $c(v) \neq c(w)$ for all $(v, w) \in E$?

[Hint: Show SAT(3) $\leq$ Chromatic Number.]

10) Show that Planar 3-SAT is *NP*-complete.

Name:	Planar 3-SAT.
Input:	A formula α in CNF such that the following bipartite graph (V, E) is planar. V is the set of variables and clauses of α and $(v, c) \in E$ for variables v and clause c if either v or $\bar{v}$ occur in C.
Question:	Is α satisfiable?

11) Show that Bandwidth is *NP*-complete.

Name:	Bandwidth.
Input:	Undirected graph $G = (V, E)$, integer K.
Question:	Is there a bijection $f : V \to [1 \dots \lvert V \rvert]$ such that $\lvert f(u) = f(v) \rvert \leq K$ for all $(u, v) \in E$?

12) Show that Cycle Cover is *NP*-complete.

Name:	Cycle Cover.
Input:	Undirected graph $G = (V, E)$, integer K.
Question:	Is there a set $V' \subseteq V$, $\lvert V' \rvert = K$, such that every cycle in G contains at least one node in V'?

13) Show that SIT(2) (Scheduling on *two* machines) is in P. [Hint: Schedule by depth in the graph $G = (V, E)$.]

14) Eulerian Cycle is in P.

Name:	Eulerian Cycle.
Input:	Undirected graph $G = (V, E)$.
Question:	Is there a cycle which uses every edge exactly once, i.e., is it possible to order $E = \{e_1, \dots, e_m\}$ such that e_i and e_{i+1} have a common endpoint, $1 \leq i < m$?

[Hint: Show that a Eulerian Cycle exists iff every node has even degree.] Derive a linear time algorithm to construct a Eulerian Cycle.

15) Design a pseudo-polynomial time algorithm for the weighted Knapsack optimization problem, i.e., given $w_1, \dots, w_n, c_1, \dots, c_n, K$ compute

$$C_{opt} = \max\{\sum_{i=1}^{n} c_i \cdot x_i;\ x_i \in \{0,1\} \quad \text{and} \quad \sum w_i x_i \leq K\}.$$

[Hint: For $0 \leq c < \infty$ and $0 \leq j \leq n$ let $F(c, j) = \min(\{\infty\} \cup \{w;$ there is $(x_1, \dots, x_j) \in \{0,1\}^j$ such that $\sum_{i=1}^{j} c_i \cdot x_i = c$ and $\sum_{i=1}^{j} w_i \cdot x_i = w\})$. Then $F(0, 0) = 0$ and $F(c, 0) = \infty$ for $c > 0$ and $F(c, j+1) = \min\{F(c, j), F(c - c_{j+1}) + w_{j+1}\}$. Show that the relevant part of table F, i.e., $F(c, j) \leq K$ can be computed in time $O(n \cdot C_{opt}) = O(n^2 \max c_i)$. Also, vector $(x_1, \dots, x_n)$ can be found in that time bound.]

16) Design a pseudo-polynomial time algorithm for Scheduling Independent Tasks for fixed number m of machines. [Hint: Let $t_1, \dots, t_n$ be the time requirements of n jobs and let T be the deadline. For $i \in [0 \dots n]$ let $f(i) = \{(T_1, \dots, T_m);\ T_j \leq T$ and there is a partial schedule $PS : [1 \,..\, i] \to [1 \,..\, m]$ such that $T_j = \sum\{t_k;\ k \leq i$ and $PS(k) = j\}$ for $1 \leq j \leq m\}$. Then $f(0) = (0, 0, \dots, 0)$ and $f(i+1) = \{T_1', \dots, T_m');$ there is $(T_1, \dots, T_m) \in f(i)$ and $k \in [1 \dots m]$ such that $T_k' = T_k + t_{i+1}$ and $T_j' = T_j$ for $j \neq k\}$. Derive an $O(n \cdot m \cdot (T+1)^m)$ algorithm for these observations.]

17) Let $G = (V, E)$ be a directed graph, let s and t be distinguished nodes and let $c : E \to \mathbb{R}^+$ be a non-negative cost function on the edges. For a node $v \in V$ let $\mu(s, v)$ be the cost of the least cost path from s to v. A function $g : V \to \mathbb{R}_0^+$ is called an estimator if $g(v) \leq \{\min c(p);\ p$ is path from v to $t\}$. In Section 4.7.2 we treated algorithms for computing $\mu(s, t)$ using an estimator g.

a) Formulate Branch-and-Bound in terminology of finding least cost paths. What are the values of c and g in the case of the Branch-and-Bound algorithm for the traveling salesman problem discussed in Section 6.2.

b) Are the estimators which arise from the Branch-and-Bound algorithms consistent in the sense of Section 4.7.2, i.e., $g(v) + c(v, w) \geq g(w)$ for all edges $(v, w) \in E$?

18) Use the Branch and Bound algorithm of Section 6.5 to solve the following problem of $n + 6$ cities $v_0, v_1, v_2, w_0, w_1, w_2, x_0, \dots, x_{n-1}$.

$$\begin{aligned}
dist(v_i, v_{i+1 \bmod 3}) &= 2 && i = 0, 1, 2 \\
dist(v_i, w_i) &= 1 && i = 0, 1 \\
dist(v_2, w_2) &= 0 && \\
dist(w_i, v_i) &= 2 && i = 0, 1, 2 \\
dist(w_i, w_{i+1 \bmod 3}) &= 1 && i = 0, 1, 2 \\
dist(w_i, x_j) &= 1 && \text{for } i = 0, 1, 2,\ j = 0, 1, \dots, n-1 \\
dist(x_i, w_j) &= 1 && \text{for } i = 0, 1, 2,\ j = 0, 1, \dots, n-1 \\
dist(x_j, w_j) &= 1 && j = 0, 1, \dots, n-1 \\
dist(x_j, x_k) &= 0 && \text{for } j, k = 0, \dots, n-1
\end{aligned}$$

The remaining distances are infinite. Run the algorithm with starting points v_1 and v_3 respectively and always proceed to the nearest city.

19) Design an $O((\log sopt)^{sopt}|E|)$ algorithm for finding an optimal cycle cover of an undirected graph (cf. Exercise 12). [Hint: Use the following graph-theoretic lemma: If G is a graph of minimum degree three and if G has p pairwise disjoint cycles then G has a cycle of length $O(\log p)$. Also, a shortest cycle containing a fixed node v can be found in time $O(|E|)$ by breadth-first-search.]

20) Repeat the Branch-and-Bound algorithm for TSP using the weighted matching problem as a lower bound on cost.

21) Design a class of examples where the once-around-the-tree algorithm for $\triangle$TSP actually constructs a tour which has almost twice the length of the optimum. Similarly for the 0.5-approximate algorithm.

22) Bin Packing.

Input: Integers $l_1, \ldots, l_n$ and bound L
Output: Minimal m such that there is a mapping $S : [1 \ldots n] \to [1 \ldots m]$ with $\sum\{l_i;\ S(i) = j\} \leq L$ for all j, $1 \leq j \leq m$.

a) Show that the recognition version (m is additional input) is NP-complete.

b) Design approximation algorithms for bin packing. [Hint: The following strategies are good: Start with empty bins $1, 2, 3, \ldots$. Add objects one by one. When a new object is added place it into the least numbered (first fit) or the fullest (best fit) bin which can still take the object. These two rules can either be applied to the objects in any order or to the objects in order of decreasing length. First fit and best fit use at most $(17/10) \cdot m_{opt} + 2$ bins and first fit and best fit decreasing use at most $(11/9) \cdot m_{opt} + 2$ bins.]

23) Design a 1-approximate algorithm for vertex cover. [Hint: start with graph $G = (V, E)$; take any edge and remove both endpoints and all edges adjacent to them from the graph. Repeat.]

24) Design a $g(x) = O((\log x)^2)$-approximate algorithm for cycle cover (Exercise 12 and 19). [Hint: Start with graph $G = (V, E)$; search for the shortest cycle and remove all nodes of the cycle. Repeat.]

25) (Worst case performance of LPT algorithm). Let $n = 2m + 1$ and $t_i = 2m - \lfloor (i + 1)/2 \rfloor$, $1 \leq i \leq 2m$, $t_{2m+1} = m$. Construct optimal and LPT schedule.

26) Let $w_1, \ldots, w_n, c_1, \ldots, c_n, K$ be an instance of the weighted Knapsack problem (cf. Exercise 7 and 15). Assume w.l.o.g. that $c_1/w_1 \geq c_2/w_2 \geq \cdots \geq c_n/w_n$ and $w_i \leq K$. Consider the following algorithm.

```
$c \leftarrow 0$; $W \leftarrow 0$;
**for** $i$ **from** 1 **to** $n$
**do if** $W + w_i \leq K$
    **then** $x_i \leftarrow 1$; $C \leftarrow C + c_i$
    **else** $x_i \leftarrow 0$
    **fi**
**od**;
$C \leftarrow \max(\{C\} \cup \{c_i;\ 1 \leq i \leq n\})$.
```

Let C_{alg} be the value of C after termination and let $C_{opt} = \max\{\sum_i c_i \cdot y_i;\ y_i \in \{0,1\}$ and $\sum w_i \cdot y_i \leq K\}$. Show $C_{opt} \geq C_{alg} \geq C_{opt}/2$, i.e., the algorithm above is $0,5$-approximate. [Hint: Let $(y_1, \dots, y_n)$ be an optimal solution, and let i be minimal such that $x_{i+1} = 0$. Then $i \geq 1$, $w_1 + \dots + w_i \leq K < w_1 + \dots + w_i + w_{i+1}$, and $C_{alg} \geq c_1 + \dots + c_i$. Also $C_{opt} = \sum c_j \cdot y_j = \sum_{j \leq i} c_j + \sum_{j \leq i} (c_j/w_j) \cdot w_j \cdot (y_j - 1) + \sum_{j>1} (c_j/w_j) \cdot w_j \cdot y_j \leq \sum_{j \leq i} c_j + (c_{i+1}/w_{i+1}) \cdot (\sum_{j \leq i} w_j (y_j - 1) + \sum_{j \geq i} w_j \cdot y_j) \leq C_{alg} + (c_{i+1}/w_{i+1})(\sum_j w_j \cdot y_j - \sum_{j \leq i} w_i) \leq C_{alg} + (c_{i+1}/w_{i+1}) \cdot (K - \sum_{j \leq i} w_j) \leq C_{alg} + (c_{i+1}/w_{i+1}) \cdot w_{i+1} \leq C_{alg} + c_{i+1} \leq 2 \cdot C_{alg}$.] ∎

27) Describe a full approximation scheme for the Knapsack problem. [Hint: Use scaling and pseudo-polynomial algorithm for Knapsack.]

28) Describe a full approximation scheme for SIT(m). [Hint: Use scaling, the pseudo-polynomial algorithm of Exercise 16 and the *LPT* algorithm to obtain a good initial value for T_{opt}.]

6.10. Bibliographic Notes

A detailed treatment of Turing machines can be found in the books by Hopcroft/Ullman (69) and Paul (78). The connection between recognition and optimization problems, in particular the concept fo self-reducibility, is treated by Schnorr (76). Ladner/Lynch/Selman (74), Ladner (75) and Mehlhorn (76) investigate the properties of polynomial transformations. Theorem 5 (NP-completeness of SAT) is by St. Cook (71). Most NP-complete problems of Section 5 are taken from the paper of R. Karp (72). NP-complete scheduling problems can be found in J. Ullman (75), Planar 3-SAT is from D. Lichtenstein (1982) and Bandwith is from Papadimitriou (76). The concept of strong NP-completeness was introduced by Garey/Johnson (78). Their book is also an extensive treatment of NP-completeness. Held/Karp describe the dynamic programming approach to TSP, a more detailed treatment of branch-and-bound methods for TSP is contained in Christofides' book. Christofides is also the author of the $0,5$-approximate algorithm for $\triangle$TSP. Theorem 4 of Section 6.6 and Exercises 19 and 24 are from B. Monien (82), Exercise 17 is from N. Nillson (71), and Exercise 22 is from Johnson et. al. (74). De la Vargee/Lueker (81) describe a full approximation scheme for bin packing, the approximate algorithms for SIT is by Graham (69), and the full approximation scheme for Knapsack is by Ibarra/Kim (75) and Lawler (77).