LECTURES 4,5 & 6

1 Complexity Theory

This part of the course will teach us why some problems are more difficult than others, but on the way we will learn a lot of different classes of problems like scheduling problems, routing problems and facility location problems.

The problems that we have seen in this course were all solvable with algorithms that were efficient, allowing to solve large instances exactly. E.g. every instance of MST with n vertices can be solved in $O(n \log n)$ time. We will be very strict in this part of the course in distinguishing between a *problem* and its *instances*.

The situation is much worse for the Travelling Salesman problem:

TRAVELLING SALESMAN. (TSP)
Instance: Given n points and a distance between each pair of points.
Question: Find a tour that passes each point exactly once, ending in the point where it started, and with minimum total length.

Despite 50 years of intensive research on the TSP no efficient algorithm has been found. Complexity theory aims to answer the question if the research community has been too stupid or unlucky, or that TSP is in fact intrinsically more difficult than other problems.

Complexity theory works with a mathematically rigorous definition of *efficient algorithm*:

Definition. An efficient algorithm for problem Π is an algorithm that solves *every* instance of Π in a number of elementary computer operations that is bounded by a *polynomial function* of the *size of the instance*.

Some phrases in this definition are emphasized. The algorithm is to solve *every* instance, even the hardest instance. Thus, efficiency is a *worst-case* notion.

I hope everyone knows the definition of polynomial function: f is polynomial if there exist a finite k and constants a_i, b_i, $i = 1, \ldots, k$ such that

$$f(x) = \sum_{i=1}^{k} a_i x^{b_i}.$$

Non-polynomial functions are exponential functions, like

$$\begin{aligned} f(x) &= 2^x \\ f(x) &= (x \log x)^x \\ f(x) &= 10^{-28} 3^{\sqrt{x}} \\ f(x) &= x! \end{aligned}$$

We call $f(x) = x^{\log x}$ pseudo-polynomial.

The *size of an instance* is defined as the number of bits that are needed in a digital computer to store all relevant data of the instance. Most of the time we can rely on an intuitive definition of the size, like in the TSP the number of points, or in Max Flow the number of vertices and the number of arcs of the graph. But let me once show you what the size of a MAX FLOW instance is according to the official definition.

For each of n vertices we need $\log n$ bits. For each of the m arcs (i, j) we need $2\log n + 3$ bits (the 3 extra to separate the pair and show they belong together). Finally we have to specify the capacities on the edges' Each capacities u_{ij} is specified in $\log u_{ij}$ bits (where I assume integer capacities). Thus the total length of the input is:

$$N = n\log n + m(2\log n + 3) + \sum_{(i,j)\in A} \log u_{ij}.$$

If we forget about the input size for representing the weights, then $N = O(n^2 \log n)$. Clearly if we have an algorithm with running time bounded by a polynomial function of n then the running time will also be bounded by a polynomial function of N. And the other way round. Thus, in fact for determining efficiency we could as well take n as size of the input. Sometimes the input size due to representation of numerical values like weights plays a crucial role.

Why do we draw the line of efficiency between polynomial time and exponential (non-polynomial) time. We may illustrate this with a table that shows what technological improvement in computer speed would give us in terms of problem size that we can handle in one second.

	Now	100× faster	1000× faster
n^5	N_1	$2.5N_1$	$4N_1$
2^n	N_2	$N_2 + 7$	$N_2 + 10$

Of course, some polynomial functions like $f(n) = n^{19}$ is for moderate values of n greater than $g(n) = 2^n$. But already for $n = 200$ $g(n) > f(n)$. For most problems that are solvable by efficient algorithms the running time is usually not a high power of n; mostly 4 or less than and exceptionally 6. Similarly, exponential running times of $O(3^n)$ hardly occur. Mostly it is $O(2^n)$.

We will most often speak of efficient algorithms as *polynomial time algorithms*. Based on the existence or non-existence of such algorithms we make a distinction between *easy* or *well-solved* problems and *hard* problems.

For formal reasons the complexity theory is based on *decision problems* i.o. optimisation problems. Decision problems are problems for which the answer is "yes" or "no". Examples: PRIME: Given a natural number n is it prime? Or HAMILTON CIRCUIT: Given a graph, does it have a Hamilton Circuit?

However, for each optimisation problem we can formulate a decision problem. For example:
MST-DECISION. Given G and $w : E \to \mathbb{Z}$ and a constant K, does there exist a spanning tree of G with total weight at most K?

Clearly the decision version of an optimisation problem is easier than the optimisation problem itself. But in most cases if there exists an efficient algorithm for solving the decision version then there exists an efficient algorithm for the optimisation problem (e.g., in most cases bisection search on the possible optimal value will work). Make Exercise 2 from Chapter 15 in [PS].

Exercise 5.1. Formulate TSP-DECISION.

Exercise 5.2. Formulate LP-DECISION.

Exercise 5.3. Formulate MAX FLOW-DECISION.

1.1 The classes P and NP

Definition. P is the class of decision problems for which an efficient algorithm exists.

All problems that we have treated so far in this course belong to this class. (Check for yourself that indeed the running times are polynomial functions of the input sizes.) But also LP belongs to this class, even though simplex is not a polynomial time algorithm for LP. Read for yourself Section 8.6 in [PS]. Simplex works almost always very fast in practice for any LP of whatever size, but as mentioned before the running time of an algorithm is determined in the worst-case. For LP and any pivoting rule devised so far for Simplex, there have been constructed instances on which Simplex has to visit an exponential number of basic feasible solutions before arriving at the optimal one. The ellipsoid method is a polynomial time algorithm for LP. Although explained in the book, it is not part of this course, but it will be studied in the LNMB course ALP in the spring 2010. It is an example of where the time bound is polynomial in the logarithm of the largest coefficient in the instance next to the number of variables and the number of restrictions. It is one of the most interesting open research questions in OR if there exist an algorithm in LP whose running time is a polynomial function of the number of variables and the number of restrictions only.

Next to this class P there is the class NP. The N in NP is not an abbreviation for "non-", but for "non-deterministic". The computer model used to define this class is a non-deterministic Turing machine. Fortunately there is an equivalent definition that is perfectly intuitive:
Definition. NP is the class of decision problems for a "yes"-answer of which a certificate can be given with which correctness of the answer can be verified in polynomial time.

HAMILTON CIRCUIT$\in$NP. A "yes"-answer can be certified by a set of n edges that are claimed to form the circuit. Just following the edges it is easy to verify that they form a

single cycle on which no vertex appears twice in $O(n)$ time.

LP-DECISION$\in$NP. A "yes"-answer is given by a feasible solution with the claimed objective value. It is just a matter of verifying feasibility ($O(mn)$ time) and if the objective value is less than or equal to K ($O(n)$ time). Thus in total $O(mn)$ time.

Exercise 5.4. Show that TSP-DECISION$\in$NP.

Exercise 5.5. Show that MST-DECISION$\in$NP.

Exercise 5.6. Show that the decision version of the problem of **Exercise 2.3.**, STEINER TREE-DECISION, is in NP. (*Does there exist a tree with total weight at most K, spanning a given subset of the vertices of a graph.*)

Exercise 5.7. Show that the decision version of the problem of **Exercise 2.4.** is in NP. (*Does there exist a spanning tree of total weight at most K with the property that its leaves are contained in a subset of the vertices.*)

Exercise 5.8. Is PRIME in NP?

If you think a little bit it will become clear that P$\subset$NP. For problems like TSP, HAMILTON CIRCUIT, KNAPSACK, STEINER TREE and many other problems there have not been found efficient (polynomial time) algorithms. This suggests that some problems in the class NP are intrinsically more difficult than others. Complexity Theory gives a theoretical evidence that this is true. It defines within the class NP a subclass of most difficult problems, the so-called *NP-Complete* problems. Most difficult means here that if for any of the problems of this subclass there will ever be found a polynomial time algorithm then there exists a polynomial time algorithm for each problem in NP, which would imply the P=NP. One of the 7 millennium problems in mathematics is to prove the conjecture that P$\neq$NP. So we do believe that in fact there are problems in NP that are not solvable efficiently.

1.2 NP-completeness

To specify a most difficult problem in NP, we introduce the notion of *polynomial transformation* from one decision problem to another decision problem. But first we notice that a polynomial function of a polynomial function is again a polynomial function:

Lemma 1. *If $f(n)$ and $g(n)$ are polynomial functions of n then $f(g(n)$ is a polynomial function of n.*

Definition. A *polynomial transformation* from decision problem Π_1 to decision problem Π_2 is a mapping $F : \Pi_1 \to \Pi_2$ that satisfies

- For every instance $\pi_1 \in \Pi_1$ there is an instance $\pi_2 = F(\pi_1) \in \Pi_2$;
- The transformation F of $\pi_1 \in \Pi_1$ can be executed in a number of elementary computer operations bounded by a polynomial function of the input size of π_1;

– π_1 has a "yes"-answer if and only if $\pi_2 = F(\pi_1)$ has a "yes"-answer.

If there exists a polynomial transformation from Π_1 to Π_2 then from the point of view of polynomial time solvability Π_2 is more difficult to solve than problem Π_1. Because, suppose that we could solve Π_2 in polynomial time then to solve Π_1, for each instance we construct in polynomial time the transformation $\pi_2 = F(\pi_1)$ and solve π_2 with the polynomial time algorithm for Π_2. From the definition of polynomial transformation we know that if the answer to π_2 is "yes" then the answer to π_1 is also "yes", and the other way round. So by Lemma 1 we have a polynomial time algorithm for Π_1.

The existence of a polynomial time algorithm for Π_1 does not say anything about the existence of a polynomial time algorithm for Π_2. Because, the polynomial transformation F is not required to be a 1-to-1 mapping. In general the mapping F of Π_1 will have as domain only a subset of the instances of Π_2. It could very well be that this subset is a set of easy instances of Π_2. Since complexity is a worst-case notion, this means that there could exist very well another subset of instances of Π_2 that are hard to solve.

If there exist a polynomial transformation from Π_1 to Π_2 then we say that Π_1 can be *reduced to* Π_2 or that there is a *reduction from* Π_1 *to* Π_2, and we write $\Pi_1 \propto \Pi_2$.

Reducability is a transitive property:

Lemma 2. *If* $\Pi_1 \propto \Pi_2$ *and* $\Pi_2 \propto \Pi_3$ *then* $\Pi_1 \propto \Pi_3$

Given this notion of reduction we can now define a most difficult problem in NP.

Definition. A problem Π in NP is *NP-Complete* if every problem $\Pi' \in$NP can be reduced to Π.

This is not a very helpful definition for proving that TSP is NP-Complete. However, the class NP has a very precise definition in terms of executions of non-deterministic Turning machines, which we skipped and persist in skipping, but which enabled Steven Cook in 1974, to prove that any such execution can be reduced to an instance of a famous problem in Boolean logic called the SATISFIABILITY Problem. I will of course also skip this reduction but I will give you the definition of the logic problem in a minute.

Thus Cook provided us with a most difficult problem in the class NP. But this may make life a lot easier to prove that TSP is a most difficult problem in the class NP. To prove that TSP is NP-Complete we "just" need to prove that TSP is in NP and to reduce SATISFIABILITY to TSP. Given that we find such a reduction and given that SATISFIABILITY is NP-complete, then TSP, that we prove then to be more difficult than SATISFIABILITY, must be NP-Complete as well.

Nobody ever made a direct reduction from SATISFIABILITY to TSP, but I will show NP-Completeness of TSP in steps, on the way showing NP-Completeness of other problems.

In SATISFIABILITY the basic ingredients are *variables*. A variable reflects an expression which can be TRUE or FALSE. As an example one could have $x_1 :=$ *Koen is longer than Michael* and $x_2 :=$ *Soup is always eaten with a fork*. The variables can appear in

a composed expression as they are or in negated form, which I will indicate as $\neg x_1$ (saying that *Koen is not longer than Michael*). But let us completely forget about the interpretation of the variables and their negations and instead concentrate on the role they play in the composed expressions that we call *clauses*. A clause consists of *literals*, with each literal a variable or its negation. A clause is written as:

$$C_1 = (x_1 \vee \neg x_2 \vee x_3 \vee x_4)$$

C_1 is TRUE if x_1 is TRUE OR ($\vee$) not-x_2 is TRUE OR x_3 is TRUE OR x_4 is TRUE. An instance of SATISFIABILITY is a *Boolean expression in Conjunctive Normal Form* (CNF), i.e., an expression in clauses of the form

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

which is TRUE if C_1 is TRUE AND ($\wedge$) C_2 is TRUE AND ... AND C_m is TRUE; i.e., the expression is TRUE if *all* clauses are TRUE and a clause is TRUE if at least one of its literals is TRUE.

SATISFIABILITY.
Instance: Given a Boolean expression in CNF.
Question: Is there a TRUE-FALSE assignment to the variables such that the expression is TRUE?

Exercise 5.9. Given that SATISFIABILITY is NP-Complete show that 0-1-ILP-DECISION is NP-Complete, simply by formulating SATISFIABILITY as a 0-1-ILP problem.

Exercise 5.10. Given that 0-1-ILP-DECISION is NP-Complete show that ILP-DECISION is NP-Complete. *Hint: This is trivial!*

Let us start by showing that a restriction of SATISFIABILITY called EXACT-3-SAT is NP-Complete.

EXACT-3-SAT.
Instance: Given a Boolean expression in CNF with each clause containing 3 literals.
Question: Is there a TRUE-FALSE assignment to the variables such that the expression is TRUE?

Theorem 1. EXACT-3-SAT *is NP-complete.*

Proof. . Notice very well the structure of the proof, which is the same for all NP-completeness proofs.

1.) It is easy to see that EXACT-3-SAT) is in NP. A certificate is a TRUE-FALSE assignment to the variables and it is a matter to verify for each clause that at least one of its three literals is TRUE. Thus, this takes $O(m)$ time if m is the number of clauses.
2.) Reduce a problem known to be NP-Complete (in this case SATISFIABILITY) to the problem to be proven to be NP-Complete (in this case EXACT-3-SAT).

2.) SATISFIABILITY $\propto$ EXACT-3-SAT.
For the reduction we notice that a clause of SATISFIABILITY does not necessarily consist of 3 literals. If it does we do not need to do anything and transfer the clause of SATISFIABILITY to a clause of EXACT-3-SAT. Now let us make the transformation for clauses that do not consist of exactly 3 literals, and let us for each of the three cases we distinguish immediately verify that YES-answers correspond.

In the case we distinguish, a clause C contains just one literal, x say, then we introduce 2 extra artificial variables a and b and replace the clause C by the sequence of clauses $(x \vee a \vee b) \wedge (x \vee \neg a \vee b) \wedge (x \vee a \vee \neg b) \wedge (x \vee \neg a \vee \neg b)$. Clearly, if C is TRUE then all four clauses are TRUE. Reversely, notice that the only way in which all these four clauses can be become TRUE is by making x TRUE.

If a clause C has two literals, $(x \vee y)$ say, then we introduce an artificial variable c and replace the clause C by the duo of clauses $(x \vee y \vee c) \wedge (x \vee y \vee \neg c)$. Again, it is easy to see that if C is TRUE then the two new clauses will be TRUE. Also here, the only way in which both these clauses can be made TRUE is by making x or y TRUE, i.e. the original clause C TRUE.
If a clause C has $k > 3$ literals, $(x_1 \vee x_2 \vee \ldots \vee x_k)$ say. Then we introduce artificial variables and replace C by the sequence of clauses $(x_1 \vee x_2 \vee z_1) \wedge (\neg z_1 \vee x_3 \vee z_2) \wedge (\neg z_2 \vee x_4 \vee z_3) \wedge \ldots \wedge (\neg z_{k-4} \vee x_{k-2} \vee z_{k-3}) \wedge (\neg z_{k-3} \vee x_{k-1} \vee x_k)$. Here it is a bit more tricky to see that YES-answers correspond. Suppose that x_i is TRUE. Then, the clause $(\neg z_{i-2} \vee x_i \vee z_{i-1})$ is immediately TRUE. But then it is enough to set z_{i-2} to TRUE to make $(\neg z_{i-3} \vee x_{i-1} \vee z_{i-2})$ TRUE, and then set z_{i-3} to TRUE to make $(\neg z_{i-4} \vee x_{i-2} \vee z_{i-3})$ TRUE, etc. until setting z_1 to TRUE. In the other direction we set z_{i-1} to FALSE to make clause $(\neg z_{i-1} \vee x_{i+1} \vee z_i)$ TRUE, etc, till setting z_{k-3} to FALSE to make clause $(\neg z_{k-3} \vee x_{k-1} \vee x_k)$ TRUE.

The other way round, suppose none of the x_i's is TRUE, then we show that not the whole sequence of clauses can be made TRUE. To make the first clause TRUE, z_1 must be set to TRUE, implying that to make the second clause TRUE, z_2 must be set to TRUE. etc., to make the last but one clause TRUE, z_{k-3} must be set to TRUE. But then all three literals of the last clause are FALSE.

Thus on the way in describing the transformation we have shown that YES-answers correspond. It leaves ut to prove that the transformation can be executed in polynomial time. For each clause of k literals in SATISFIABILITY we create at most $4k$ clauses of three literals of EXACT-3-SAT. Thus, in fact the transformation takes a time linear in the size of the SATISFIABILITY instance. □

In [PS] it is then shown that CLIQUE-DECISION is NP-Complete. Let us skip that one (takes too much time) and do some easy ones given that CLIQUE-DECISION is NP-Complete.

Given a graph $G = (V.E)$ a *clique* in G is a subset of the vertices that induces a complete subgraph of G. The opposite, a subset of the vertices such that no two of them are incident to the same edge is called an *independent set* or *stable set*. A subset of the vertices such that for each edge at least one of its two incident vertices is in the subset is called a

vertex cover.

CLIQUE-DECISION.
Instance: Given a graph $G = (V,E)$ and a number K. *Question*: Does G contain a clique of size at least K?

INDEPENDENT SET-DECISION.
Instance: Given a graph $G = (V,E)$ and a number K. *Question*: Does G contain an independent set of size at least K?

VERTEX COVER-DECISION.
Instance: Given a graph $G = (V,E)$ and a number K. *Question*: Does G contain a vertex cover of size at most K?

Theorem 2. INDEPENDENT SET-DECISION *is NP-Complete.*

Proof. A certificate for a "yes"-answer is a subset of the vertices that for an independent set. To verify this, we just need to check if between each pair of vertices in the subset there does not exist an edge. This takes at most $O(n^2)$ time.

For the reduction CLIQUE$\propto$INDEPENDENT SET, we use the notion of *complement graph*. The complement of a graph $G = (V,E)$ is the graph $G^C = (V,E^C)$ with $\{i.j\} \in E^C \Leftrightarrow \{i.j\} \notin E$.

Given a graph G as instance of CLIQUE we define the complement graph G^C as an instance of INDEPENDENT SET and we keep the same constant K. To show that this is a reduction we have to show that it can be executed in polynomial time and that "yes"-answers correspond.

The mapping is clearly polynomial time. For every pair of vertices in V if there is no edge in E then create the edge in E^C, and leave it out otherwise. This takes at most $O(|V|^2)$ time.

To prove that "yes"-answers correspond, we claim that $V' \subset V$ is a clique in G if and only if V' is an independent set in G^C. That V' is a clique implies that between each pair of its vertices there exist an edge in E, hence there does not exist an arc in E^C, implying that each pair of vertices in V' is independent, and thus V' is an independent set in G^C, and vice versa. Therefore G has a clique of size at least K if and only if G^C has an independent set of size at least K. □

Theorem 3. VERTEX COVER-DECISION *is NP-Complete.*

Exercise 5.11. Prove this theorem. *Write it out in the form I just did. Use a reduction from* INDEPENDENT SET *and prove for corresponding "yes"-answers that V' is an independent set in a graph if and only if $V \setminus V'$ is a vertex cover in the graph.*

As a difficult one I explain you the proof in [PS] that HAMILTON CIRCUIT is NP-Complete by a reduction from EXACT 3-SAT.

In fact what we proved is that HAMILTON CIRCUIT on a graph with vertices of degree at most four is NP-Complete. See Theorem 16.7 in [PS] that HAMILTON CIRCUIT remains NP-Complete if the graph has vertices all of degree at most three. Since these special cases of HAMILTON CIRCUIT on a general graph are NP-Complete and since HAMILTON CIRCUIT on a general graph is in NP, NP-Completeness of HAMILTON CIRCUIT follows immediately.

NP-Completeness of TSP-DECISION follows rather straightforwardly. For all of this I just follow [PS], Theorem 15.6 and its Corollary 2. Study for yourself Corollary 1, saying that HAMILTON PATH is NP-Complete proved by a reduction from HAMILTON CIRCUIT.

Exercise 5.12. Indicate for each of the four statements below if it is TRUE or FALSE, given two problems Π_1 and Π_2 that are both member of the class NP:
i) If $\Pi_1 \in P$ and $\Pi_1 \propto \Pi_2$ then $\Pi_2 \in P$;
ii) If Π_1 is NP-Complete and $\Pi_1 \propto \Pi_2$ then Π_2 is NP-Complete;
iii) If $\Pi_1 \in P$ and $\Pi_2 \propto \Pi_1$ then $\Pi_2 \in P$;
iv) If Π_1 is NP-Complete and $\Pi_2 \propto \Pi_1$ then Π_2 is NP-Complete.

In the proof of NP-Completeness of TSP-DECISION it is actually shown that the restriction of TSP-DECISION, in which all distances are 1 or 2 is NP-Complete. Since TSP-DECISION is in NP and it is a generalisation of this problem, NP-Completeness of TSP-DECISION follows directly.

Same with SATISFIABILITY. If we would not have known that it is NP-Complete but we would know NP-Completeness of EXACT-3-SAT then NP-Completeness of SATISFIABILITY follows automatically from the fact that EXACT-3-SAT is a special case of SATISFIABILITY.

Restrictions can create an easier subclass of problem instances. But generalisations always create a more difficult problem. This gives sometimes easy ways to prove NP-Completeness of problems.

1.3 Co-NP

As we argued, for many decision problems, their optimisation version is the more natural problem. Like TSP is the problem of finding a shortest travelling salesman tour. VERTEX COVER is the problem of finding a minimum cardinality vertex cover. Etc. It is clear that these problems are at least as hard as their decision versions. But it is not so clear that they are in NP. First of all they are no decision problems, which is a formal reason to exclude them from the class NP. But even if polynomial verifiability of a certificate would permit membership to the class NP, it is not so clear that they are in NP.

A natural certificate for an optimal solution would be the solution itself, but how would one verify that this is indeed the optimal solution. In LP it is easy, we could also give (or compute) the corresponding dual solution, and verification of feasibility of both solutions and equality of their objective values can be done in polynomial time. In Max Flow, an optimal flow and a corresponding minimum cut have the same easy verifiability. But for TSP of VERTEX COVER we do not have such characterisations of optimal solutions and verifications of claimed optimality of a given solution is much less obvious.

Usually we do not bother about this and any problem that is not in NP, but for which polynomial solvability would imply that all problems in the class NP can be solved in polynomial time is called an *NP-Hard* problem. We usually restrict the use of the term NP-hard for optimisation problems, the decision version of which is NP-Complete. In the same way the term *NP-Easy* is used for problems that can be solved in polynomial time using the solution of a problem in NP.

Officially, NP-hard problems may be in a higher complexity class.

Another class related to NP is the class *co-NP*. The complement of a decision problem is defined as the same set of instances, but with the complementary question. As an example:

HAMILTON CIRCUIT-COMPLEMENT.
Instance: $G = (V, E)$
Question: Does G NOT have a Hamilton Circuit?

It is not known if a certificate for a "yes"-answer to this question can be given that can be verified in polynomial time.

A decision problem belongs to the class co-NP if and only if its complement belongs to the class NP. Or alternatively, if a certificate for a "no"-answer can be verified in polynomial time.

It is clear that P$\subset$NP$\cap$co-NP. Before it was known that LP$\in$P, there was strong evidence that it was, since, by duality theory, it was known that LP$\in$NP$\cap$co-NP.

Lemma 3. *If the complement of any NP-Complete problem is in NP, then NP=co-NP.*

Proof. Notice that any polynomial transformation F from any problem Π_1 to problem Π is also a polynomial transformation F from problem Π_1-COMPLEMENT to problem Π-COMPLEMENT. Suppose now that Π-COMPLEMENT, the complement of NP-complete problem Π, is in NP. Then we can give a certificate for a "yes"-answer for an instance π_1 of Π_1-COMPLEMENT by giving the polynomial transformation and the certificate for a "yes"-answer for $F(\pi_1) \in \Pi$-COMPLEMENT. Since the latter is in NP this certificate can be verified in polynomial time. Hence Π_1-COMPLEMENT is in NP. Since this holds for any problem Π_1-COMPLEMENT in co-NP it implies that NP=co-NP.

1.4 Psuedo-polynomiality and Strong NP-completeness

Consider the INTEGER KNAPSACK problem. A decision version of it is if, given n integer numbers $c_1, \ldots, c_n$ and K, there exist integer numbers $x_1, \ldots, x_n$ such that $\sum_{j=1}^{n} c_j x_j = K$.

This problem can be translated to the problem of existence of a path in a directed graph. For every integer point $i \in \{0, 1, \ldots, K\}$ create n arcs $\{\{i, j\} \mid j = i + c_k, k = 1, \ldots, n\}$. Then it is easy to verify that the instance has a "YES" answer if and only if there is a path in this graph from 0 to K. Such a path or the none-existence of such a path can be found in time $O(nK)$.

This is not polynomial time since $nK = n2^{\log K}$. But if K would happen to be bounded by a polynomial function of n then the algorithm would be a polynomial time algorithm. We call such an algorithm which is polynomial in the largest integer necessary for the description of the problem instance a pseudo-polynomial time algorithm. Clearly this largest integer is K for INTEGER KNAPSACK (assuming that $c_1, \ldots, c_n < K$). Similarly the largest integer is $K > distance(i, j),\ i, j = 1, \ldots, n$ for TRAVELLING SALESMAN.

Thus if we consider the subclass of INTEGER KNAPSACK consisting of instances in which $K = O(p(|I|))$, with p a polynomial function and $|I|$ instance size, then this is solvable in polynomial time.

Problems that remain NP-Complete even if the largest integer appearing in its description is bounded by a polynomial function of input size are called *strongly NP-Complete*.

But this largest integer is $n = |V|$ in graph problems like CLIQUE, HAMILTON CIRCUIT or INDEPENDENT SET, which is less than $|I|$. So clearly such problems are strongly NP-Complete.

In fact also TSP is strongly NP-Complete since we proved that it remains NP-Complete if we restrict to instances with distances only 1 or 2.

We cannot expect to find a pseudo-polynomial time algorithm for any strongly NP-Complete problem unless P=NP. This follows from the transitivity of polynomiality. Read yourself the proof in the book.

1.5 PSPACE

The classes P and NP (and co-NP) were based on running time. The class PSPACE is a class that is based on memory requirement.

Definition. PSAPCE is the class of problems for which an answer to each instance can be found using a number of memory bits which is bounded by a polynomial function of the size of the instance.

It is clear that P$\subset$PSPACE. But also NP$\subset$PSPACE since all possible certificates can be checked one after the other, every time deleting the previous one from memory.

1.6 Special NP-Complete problems

Here I give a list of some NP-Complete problems that are very often used in reductions. I just give them without their NP-Completeness proofs.

2-PARTITION.
Instance: Given integers $a_1, \ldots, a_n$.
Question: Is there a set $S \subset \{1, \ldots, n\}$ such that $\sum_{i \in S} a_i = \frac{1}{2} \sum_{i=1}^{n} a_i$?

3-PARTITION.
Instance: Given $3n$ rational numbers $\frac{1}{4} < a_1, \ldots, a_{3n} < \frac{1}{2}$.
Question: Is there a partition into n triplets of these numbers $T_1, \ldots, T_n$, such that $\sum_{a_j \in T_i} a_j = 1$ for all $i = 1, \ldots, n$?

SET COVER.
Instance: Given a universe of n elements U, a family of m subsets of this universe $S_1, \ldots, S_m$ and an integer K.
Question: Is there a selection of at most K subsets the union of which is U?

Exercise 5.13. Find out for each of the above 3 problems if it is strongly NP-Complete?

Material

[PS] Chapter 8: Sections 8.1-8.5, I advise you to (just) read Section 8.6;

[PS] Chapter 15: Except 15.5;

[PS] Chapter 16: Except 16.4.3 and 16.4.4, study yourself 16.2, I will later present a pseudo-polynomial time algorithm for 0-1 KNAPSACK.

Exercises

The Exercises 5.1-5.13 in the text above;

From Chapter 15: Exercises 2, 3, 6, 11;

From Chapter 16: Exercise 4, 12.