

	\	0	1	
->	q0	q1	q0	current string has - no zero at the end
	q1	q2	q0	- exactly 1 zero at the end
	q2	q2	q0	- at least 2 zeros at the end

Regular Expressions

- (1) The expression (0) produces the string 0.
The expression (1) produces the string 1.
- (2) Given expressions (a) and (b), then (a+b) is the expression producing all strings belonging to (a) or to (b) (that is, which are produced by (a) or by (b)).
- (3) Given expressions (a) and (b), then (ab) is an expression producing all strings uv, where u belongs to (a) and v belongs to (b) .
- (4) Given an expression (a), then (a)* is an expression producing all finite sequences (including the empty sequence) of strings belonging to (a).

The *language produced* by an expression is the set of all strings we get by applying (1) to (4) recursively to the given expression.

Example A regular expression for the above L is $(0+1)^*00$

Regular Grammars

A *regular grammar* consists of

- *terminal symbols* (we will use 0 and 1 mostly)
- *nonterminal symbols* (we will use capital letters S,T,...)
- an *initial symbol* S (one of the nonterminal symbols)
- *productions* of the form

$$T \rightarrow \text{str } U, \quad T \rightarrow \varepsilon \quad (\varepsilon : \text{the empty string})$$

where T and U are nonterminals and str is a terminal string

Productions can be used for *generating* strings: Every T found in the current string, may be substituted by str U.

The set of all strings we get by these rules and containing only terminal symbols is called the *language generated* by the grammar.

Example A possible regular grammar for the above L is

$$\begin{aligned} S &\rightarrow 0S \mid 1S \mid 0T \\ T &\rightarrow 0 \end{aligned}$$

A basic theorem in automata theory tells that all of the above models of computation are equivalent:

Theorem Given a formal language L, then it is equivalent to say:

- L is accepted by a finite automaton or
- L is produced by a regular expression or
- L is generated by a regular grammar.

For scanning purposes (within a compiler program for example) **nondeterministic automata** are commonly used. In a *nondeterministic automaton* the transition function

$$\rho: Q \times \Sigma \rightarrow P(Q), \text{ } P(Q) \text{ powerset of } Q$$

is a set-valued function meaning that the next state is not uniquely determined by ρ but is chosen at random among a set of possible states.

This implies that for a given input-string the sequence of computations is not unique but depends on the randomly chosen state-transitions. We say that a string is *accepted*, if the automaton *may* end up in an accepting state (by appropriately selecting the follow-up states).

For parsing purposes it is common to add a **stack-container** to the automaton, but this line of analysis will not be followed in this part. Instead we focus on complexity-theory.

In **complexity-theory** attention is put on the (worst case) execution time of algorithms. All computer models mentioned above turn out *not* to have enough potential for describing algorithms.

A more powerful computer model (that at the same time is simple and elegant) was proposed by Alan Turing many years ago:

Deterministic Turingmachine (Alan Turing, 1936)

A *deterministic turingmachine* $M=(Q,\Sigma,\Gamma,b,q_0,\rho,F)$ works like a finite automaton except that

- the tape is unbounded in both directions
- all tape space not occupied by the inputstring is filled by a special blanc symbol b , $b \in \Gamma - \Sigma$.

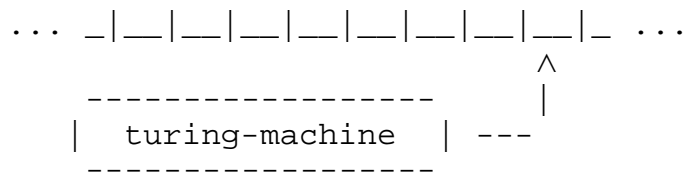
Γ is called the *tape alphabet*. In many cases we have $\Sigma = \{0,1\}$ and $\Gamma = \{0,1,b\}$.

The turingmachine is run by a *transition function* (a *program*)

$$\rho: Q \times \Gamma \rightarrow Q \times \Gamma \times \{R,L,N\}$$

which, depending on the current state and the current tape letter

- calculates the new state
- puts some letter on the tape (that is: replaces the current tape letter by a possibly different one) and
- moves the read&write head of the machine one position to the left or to the right or doesn't move the head at all.



A blanc symbol on the tape doesn't stop a turingmachine from working, so a turingmachine never seems to "stop". But in a weaker sense we can define a stop situation, when there is "no change" any more:

A state is a *final state*, if (no matter what the current tape letter is)

- the current state remains the same
- the current tape letter doesn't change
- the head doesn't move.

Again we say: An input-string w is *accepted*, if processing w the machine stops (in the above sense) in an accepting state.

For a turingmachine it is possible, that (even in the weaker sense discussed above) she never stops (for example she might alternate between two nonfinal states). So we have two possible understandings of the "accepted language":

Decidable Languages

A language L is *decidable (in the strong sense)*, if there is a turingmachine, that always stops and that accepts exactly those strings belonging to L .

A language L is *decidable (in the weak sense)*, if there is a turingmachine, that accepts exactly all the strings of L (but might never stop on other strings).

Example: Hilbert's 10-th problem is not decidable in the strong sense.

At present it is commonly believed, that every computer program (whatever complicated) can be simulated by an appropriately designed turing program. As a consequence, in complexity-theory an algorithm is viewed as a program for some turing-machine that decides a formal language (accepts/rejects input-strings).

Given a turing-program and an input-string for that program, the *execution-time* of the turing-program is measured by counting the number of steps until the program stops. Here one *step* of the program is understood as one evaluation of the transition-function ρ .

Now, given a turing-program and feeding to it a series of input-strings of increasing length, it is obvious that the execution-time of the program also will tend to get larger. Informally speaking, we understand a turing-program as "well behaved", if the execution-time of the program only increases moderately as the input-strings get larger. What we accept for the growth-rate to be "moderate" is a matter of taste or experience or computing power of available hardware. In general opinion polynomial growth-rate is viewed as moderate growth-rate:

Complexity Class /P

The *complexity class /P* contains all *decision problems* (formal languages) that can be *solved* (decided) by some deterministic turing-machine in polynomial time.

To be more specific: A problem P belongs to $/P$, if there is a deterministic turing-machine and a polynomial $p(*)$ such that

- the turing-machine decides P (gives the correct answer for all instances of P)
- for all instances of P with input length n the execution-time is bounded by $p(n)$.

Example

In the above example (natural numbers dividible by 4) a turing programm (in comparison to an automaton) is

conceptually simple to design. We just go to the right end of the input-string and check the last two bits:

	0	1	b	
--> q0	q0,0,R	q0,1,R	q1,b,L	step to the right
q1	q2,0,L	qN,1,N	qN,b,N	check last bit
q2	qY,0,N	qN,1,N	qN,b,N	check second last bit
qY	-	-	-	qY accepting state
qN	-	-	-	error state

Execution-time of this turing programm obviously is $n+2$, and this is polynomial (even linear) in time.

Examples of problems belonging to Complexity Class /P

In Graph Theory: **SHORTEST_PATH**, **MINIMUM_TREE**, **MATCHING**, **MAXIMUM_FLOW**

In Operations Research: **LINEAR_EQUATIONS**, **LINEAR_PROGRAMMING**, **QUADRATIC_PROGRAMMING**

In Logic: **2_SAT** (see below for **3_SAT**)

In Computer Science: **SCANNING**, **SEARCHING**, **PATTERN_MATCHING**

For all these (and many more) problems polynomial algorithms have been designed.

For other seemingly simple problems like **KNAPSACK**, **TRAVEL-LING_SALESMAN** ... no polynomial algorithms are known in literatur. This implies that the above complexity class /P may not be large enough to describe important problems. We need a larger problem-class than /P and to this end we need a more powerful computer-model than deterministic turing-machines.

Many important problems in Operations Research, Computer Science ... can be described/solved using some kind of branch& bound procedure. The idea is to add branching capability to our turing-machine.

Branching means subdividing the given problem instance into mutually disjoint subinstances and at each step in the

execution chose one sub-instance at random (or using some strategy).

The latter is what can be caught by nondeterminism, and this is what we do (as was proposed by Cook, 1971): We add nondeterminism to our turing-machine:

Non-deterministic Turingmachine

In a *nondeterministic turing-machine* the transition function

$$\rho: Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{R, L, N\}) \text{ , } P(*) \text{ powerset}$$

is a set-valued function meaning that action is not uniquely determined by ρ but is chosen at random among a set of possible actions.

We again say that a string is *accepted*, if the turing-machine may end up in an accepting state.

For accepted strings the *execution time* is measured by counting the minimum number of steps that lead to an accepting state.

Complexity Class NP

The *complexity class* **NP** contains all decision problems that can be solved by some non-deterministic turing-machine in polynomial time.

To be more specific: A problem **P** belongs to **NP**, if there is a non-deterministic turing-machine and a polynomial $p(*)$ such that

- exactly the instances of **L** are accepted by the turing-machine
- for all accepted instances of **P** with input length n the execution time is bounded by $p(n)$.

Remark: In connection with non-deterministic turing-machines one shouldn't be misled by the term "polynomial execution-time". A real implementation of a polynomial-time nondeterministic algorithm generally leads to exponential running time of the computer program.

What needs to be polynomial in a polynomial-time nondeterministic algorithm is only one single accepting branch of the entire branching-tree. - Obviously we have:

Theorem $P \subset NP$

In a subsequent part we need the capability of turing-machines to evaluate functions:

Turing-machines evaluating functions

Assume we are given an input-string for some turing-program. The return-value of that string is understood to be the content of the tape at stop-time between head position (including) and the first blanc symbol (excluding).

Note: For evaluating functions accepting states are not needed.

Example

This following turing-program adds 1 to a number (that is to the binary representation of that number).

The general idea of the program is: First go to the far end of the string, then (while going back) change any 1 to 0. But on finding the first 0 change it to 1 and go back to the string beginning.

	0	1	b
q0	q0,0,R	q0,1,R	q1,b,L
q1	q2,1,L	q1,0,L	s,1,N
q2	q2,0,L	q2,1,L	s,b,R
s	-	-	-

The class **NP** isn't just an unstructured set of problems. Some problems within **NP** seem to be simpler/harder than others.

Example The problem dicussed above of dividing a number by 4 is simpler than the satisfiability-problem (SAT for short):

$$\text{DIVISIBILITY_BY_4} \propto \text{SATISFIABILITY}$$

Here **SATISFIABILITY** is the logical problem of deciding if some given boolean expression can be evaluated to TRUE by setting the boolean variables to appropriate values.

To justify our opinion that the DIVISIBILITY-problem is simpler compared to the SATISFIABILITY-problem we will design a function f that transforms any DIVISIBILITY-instance (a binary string) w into a closely related SATISFIABILITY-instance (a boolean expression) $A=f(w)$.

Having done this we can view the **DIVISIBILITY_BY_4**-problem as a special (=simpler) case of the **SATISFIABILITY**-problem. The transformation is as follows. Any binary string (having at least 2 bits) can be subdivided to $w = v, b, c$, where v denotes the initial part of w and b and c pinpoint the last two bits. The function f is:

$$v, b, c \mid \text{--->} u_b \wedge \neg u_1 \wedge v_c \wedge \neg v_1$$

or, stating all 4 cases explicitly:

$$v, b, c \mid \text{--->} \begin{array}{ll} u_0 \wedge \neg u_1 \wedge v_0 \wedge \neg v_1 & \text{if } (b, c) = (0, 0) \\ u_0 \wedge \neg u_1 \wedge v_1 \wedge \neg v_1 & \text{if } (b, c) = (0, 1) \\ u_1 \wedge \neg u_1 \wedge v_0 \wedge \neg v_1 & \text{if } (b, c) = (1, 0) \\ u_1 \wedge \neg u_1 \wedge v_1 \wedge \neg v_1 & \text{if } (b, c) = (1, 1) \end{array}$$

Obviously only the first boolean expression (corresponding to $v, 0, 0$) can possibly be evaluated to TRUE.

As a consequence, checking if the two trailing bits both are 0 can be done by checking if the corresponding boolean expression can be evaluated to TRUE.

Or using other words (and giving more insight to our notion of problems being "simpler" than others): Any algorithm that might exist for SATISFIABILITY can also (by making use of the above transformation) be used for solving DIVISIBILITY.

Or, putting it into still other words: We have "reduced" the DIVISIBILITY-problem to the SATISFIABILITY-problem:

Note: Although we will not try to describe the details there is no doubt that we could (if requested) design a turing-machine for evaluating the above transformation f .

Polynomial Transformation

A *polynomial transformation* $L1 \propto L2$ of one problem (formal language) $L1(\subset \Sigma^*)$ to another problem $L2(\subset \Sigma^*)$ is a function

$$f : \Sigma^* \text{ ---> } \Sigma^* \\ w \mid \text{-->} f(w)$$

having the properties that

- f can be evaluated by a deterministic turingmaschine in polynomial time
- for all strings $w \in \Sigma^*$ we have: $w \in L1$ if and only if $f(w) \in L2$

In our example: The given binary string has 2 (or more) trailing zeros if and only if the corresponding boolean expression can be evaluated to TRUE

For $L1 \propto L2$ we say that $L1$ has been *reduced* to $L2$

In case we can reduce in both directions $L1 \propto L2$ and $L2 \propto L1$ we write $L1 \propto L2$ and say that $L1$ and $L2$ are *polynomial equivalent*.

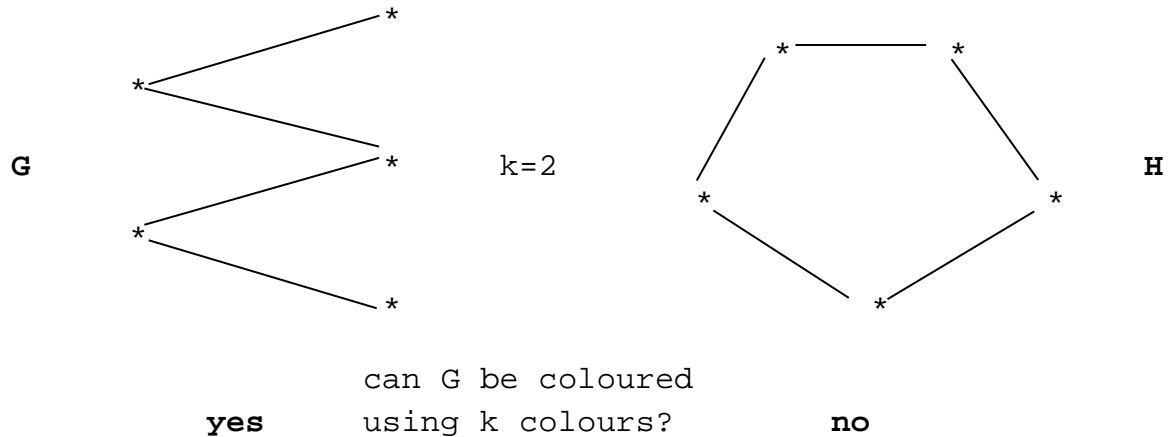
The SATISFIABILITY-Problem is particularly fruitful for polynomial reductions. We will describe one more reduction taking an example from Graph-Theory.

CHROMATIC_NUMBER

Given an undirected graph G and a parameter $k \in \mathbb{N}$

We ask: Is the chromatic number $\chi(G)$ equal or less k ?
(Can the vertices of G be coloured in such a way, that neighboring vertices are coloured differently?)

Example



We will show:

$$\text{CHROMATIC_NUMBER} \propto \text{SATISFIABILITY}$$

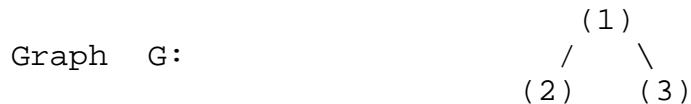
We will reduce the colouring-problem to SAT by using a specific boolean variable for any possible colouring of any possible vertex. That is we use variables

c_{ij} ; vertex i , colour j

and we think c_{ij} to be TRUE if vertex i has colour j (otherwise c_{ij} is FALSE).

Now we describe how a coloring-instance (G,k) can be transformed to a related logical expression $A=f(G,k)$:

condition	expressed using boolean variables
i) neighboring vertices need different colours:	$\neg c_{ik} \vee \neg c_{jk}$ for neighboring vertices i and j and all colours k
ii) every vertex i needs at least 1 colour:	$c_{i1} \vee c_{i2} \vee \dots \vee c_{ik}$ for every vertex i
iii) no vertex has two colours:	$\neg c_{ik} \vee \neg c_{il}$ for every vertex i and all colours $k, l, k \neq l$

Example

Using $k=2$, the colouring-instance (G,k) is transformed to the following boolean expression:

$A = f(G,k) =$

$$(\neg c_{11} \vee \neg c_{21}) \wedge (\neg c_{12} \vee \neg c_{22}) \wedge (\neg c_{11} \vee \neg c_{31}) \wedge (\neg c_{12} \vee \neg c_{32}) \quad (i)$$

$$\wedge (c_{11} \vee c_{12}) \wedge (c_{21} \vee c_{22}) \wedge (c_{31} \vee c_{32}) \quad (ii)$$

$$\wedge (\neg c_{11} \vee \neg c_{12}) \wedge (\neg c_{21} \vee \neg c_{22}) \wedge (\neg c_{31} \vee \neg c_{32}) \quad (iii)$$

The relation between colouring-instance and boolean expression is straightforward: The given graph G can legitimately be coloured using at most k colours if and only if the conjunction $A=f(G,k)$ of all the boolean expressions of types (i),(ii),(iii) can be evaluated to TRUE.

We still need check that the set of all those logical expressions (i),(ii) and (iii) really can be evaluated in polynomial time. To this end we check how many logical expressions we get.

Assume our graph has n vertices. Then we may assume our parameter k also to be n or smaller (otherwise the colouring-problem would be trivial). - We check:

Expressions of type (i) we have no more than n^3

Expressions of type (ii) we have no more than n

Expressions of type (iii) we have no more than n^3

And we have what we wanted: Transforming a graph using the above procedure leads to a logical expression whose size is polynomially bounded by the size of the graph. //

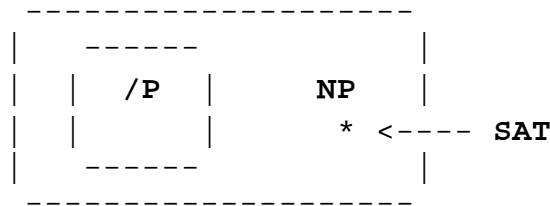
We presented all the details of the above reduction to give evidence that the technique used in that proof seems to be

quite general and that *many* problems in **NP** can be reduced to **SATISFIABILITY**. Indeed *all* problems in **NP** can be reduced to SAT as was shown by Cook in his fundamental paper:

NP-Completeness (Cook 1971)

The **SATISFIABILITY**-Problem is *NP-Complete*, that is: SAT belongs to the class **NP** and *any* other problem in **NP** can be reduced to SAT.

As a consequence, our view of the "world of NP" by now is the following:



Note: It seems obvious that the complexity-class **/P** is a *proper* subclass of **NP**, but until now this never has been formally proved!

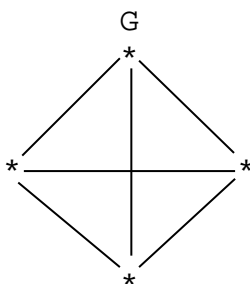
Using our above interpretation of difficult and simple problems we might come to the conclusion, that the SATISFIABILITY-Problem is *the* most difficult problem in **NP**. But this view is short sighted, because other problems in NP still might be polynomial equivalent to SAT. - For this possibility we will give an example. First we have to explain the CLIQUE-problem from Graph-Theory:

CLIQUE

Given : A graph G and a positive number k

We ask: Has G a complete subgraph having at least k vertices?

Note: The CLIQUE-problem belongs to the complexity-class NP, because obviously it can be solved by a branching-procedure.



does G have a 3-clique?

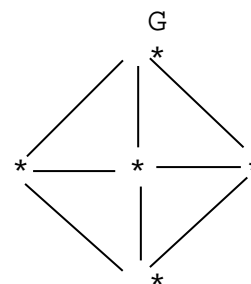
yes

yes

does G have a 4-clique?

yes

no



We might see the vertex set of the graph as representing a

group of persons. An edge indicates that two persons have a "relation", and we are asking for a subgroup of k persons where any two in the subgroup have a relation.

Now the following polynomial equivalence is true:

$$\text{-----}$$

$$| \quad \text{SATISFIABILITY} \propto \text{CLIQUE} \quad |$$

$$\text{-----}$$

To give an explanation of the proof, we first note that Cook's original proof only needs a special case of the SATISFIABILITY-problem, namely **3_SAT**, where boolean expressions in *conjunctive normal form* are considered and where only 3 literals per clause are allowed, for example:

$$A = (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (a \vee b \vee c)$$

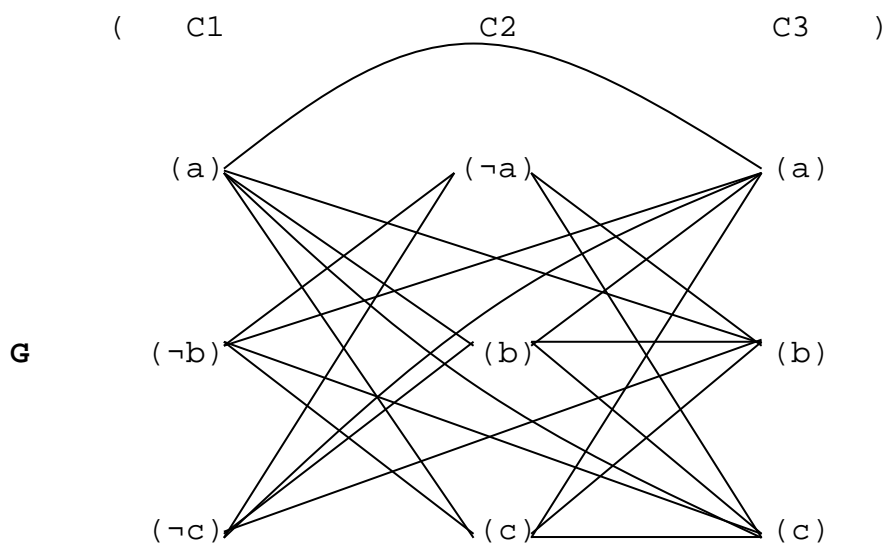
Hence **3_SAT** is NP-complete and what remains to be shown is:

$$\mathbf{3_SAT} \propto \mathbf{CLIQUE}$$

We proceed by looking at our example:

$$\mathbf{A} = \underbrace{(a \vee \neg b \vee \neg c)}_{C1} \wedge \underbrace{(\neg a \vee b \vee c)}_{C2} \wedge \underbrace{(a \vee b \vee c)}_{C3}$$

Given A , we construct a clique-instance as follows. The number k is set to be the number of clauses, $k=3$ in the example. - The graph is this:



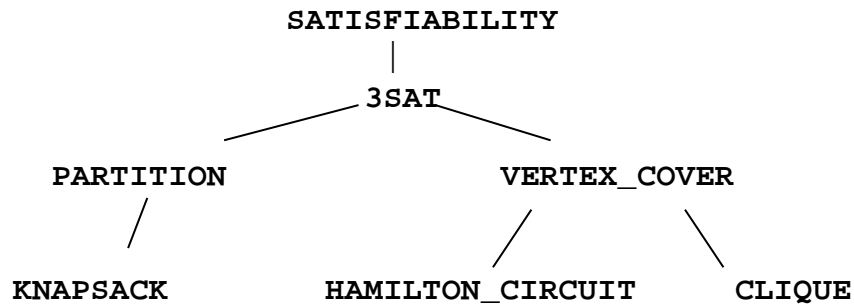
We give hints how this graph was constructed:

- the vertices are obvious.
- the edges only connect vertices in *different* columns.

- no two vertices like $(x), (\neg x)$ are connected.
- all other vertices are connected.

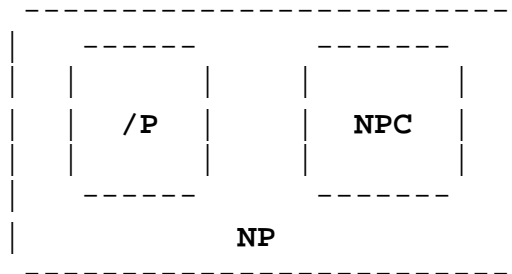
Now one has to convince oneself that the boolean expression A can be evaluated to TRUE if and only if the corresponding graph G has a k -clique. //

By now we identified three different NP-complete problems, SATISFIABILITY, 3SAT and CLIQUE. We reproduce a figure taken from Garey/Johnson's fundamental book on complexity-theory showing some more NP-complete problems:



The arcs (from top to down) in this figure indicate the direction in the original proof.

In current days a huge universe of NP-complete problems is known and our world of **NP** currently looks like this:



The class **/P** contains the "simple" problems, SHORTEST_PATH for example. **NPC** contains the NP-complete problems, SAT for example and many, many more.

Note: Discovering NP-completeness for a specific problem indicates, that this problem algorithmically is one of the most difficult to solve. This is basically a negative answer. - What should we do about that?

May be we would like to forget about the problem, but sometimes we can't. There are several answers on what we should do and what we should NOT do.

We should NOT waste our time trying to design an efficient (polynomial) algorithm for the general case of the problem because such an algorithm is unlikely to exist.

Instead we may try to concentrate on special cases of the problem that still might be solvable efficiently. Or we might look for an approximation algorithm that gives us (while not the optimal, but) reasonable good solutions.

In their book, Garey and Johnson put it this way:
 Discovering that a problem is NP-complete is usually just the beginning of work on that problem...

[**Cook**: The Complexity of Theorem Proving Procedures, Proc. 3rd ACM Symp. on the Theory of Computing, ACM(1971), 151-58]

[**Garey/Johnson**: Computers and Intractability, A Guide to the Theory of NP-Completeness, Freeman, 1979]