Case 3: NP-completeness

Antti Valmari
Tampere University of Technology

File \LaTeX ed October 6, 2009
1 Solving a problem with an algorithm for another problem

1.1 Reducing a problem to another problem

Sometimes a problem can be immediately reduced to a special case of another problem

- that is, a solution method for the other problem can be easily harnessed to produce a solution to the first problem

```
double square( double x ){ return multiply( x, x ); }
```

\[ x^2 = x \cdot x \]

We say that the first problem (squaring in the example) is \textit{reduced to} the second problem (multiplication in the example)

There is, however, no point in solving a problem by reducing it to another, if a better direct solution method is known
bool is_sorted( int A[], int n ){
    int B[n];
    for( int i1 = 0; i1 < n; ++i1 ) { B[i1] = A[i1]; }  
    sort( B[i1] );
    for( int i1 = 0; i1 < n; ++i1 ){
        if( B[i1] != A[i1] ) { return false; }  
    }
    return true;
}

Even if the target problem of the reduction is easy, reduction does not bring good performance, if the target problem has to be solved very many times

bool sort_sloooowly( int A[], int n ){
    while( !is_sorted( A, n ) ){
        int i1 = random( 0, n-1 ), i2 = random( 0, n-1 );
    }
}
1.2 Can problems be ordered according to difficulty?

Often we have a strong feeling that a certain problem is easier than another

However, it is surprisingly often possible to reduce the seemingly more difficult problem to the seemingly easier

```
double multiply( double x, double y ){
    return ( square( x+y ) - square( x-y ) ) >> 2;
}
```

\[
\frac{1}{4}((x+y)^2 - (x-y)^2) = \frac{1}{4}((x^2 + 2xy + y^2) - (x^2 - 2xy + y^2)) = xy
\]

It is admitted that this reduction involves more work than the reduction from squaring to multiplication

- two squarings
- three additions or subtractions
- one shifting by two bits

Asymptotically, that is, at the level of the \( O \)-notation, they have the same complexity

⇒ The ordering of problems according to difficulty is not as obvious as it might seem
A somewhat big example of problem reduction

Let us assume that we have an algorithm $A$ that answers the question

*Does a given map has a closed path which visits every city at least once, and whose length is at most $x$?*

- we assume that roads can be travelled in both directions
- we call the cities *vertices* and direct roads between them *edges*
- we think of a direct road between two cities as a separate edge, even if it goes partially along the same road as a direct road to a third city

A closed path is called *cycle*
If the sum of the lengths of all direct roads (= edges) is the positive integer \( p \), then the length of the shortest cycle is obtained by calling \( A \) at most \( \log_2 p + 3 \) times as follows:

\[
\begin{align*}
\text{low} & := 0; \quad \text{high} := 2p + 1 \\
\text{while} \; \text{low} < \text{high} \; \text{do} \\
\quad \text{mid} & := \lfloor (\text{low} + \text{high})/2 \rfloor \\
\quad \text{if} \; A(\text{mid}) \; \text{then} \; \text{high} := \text{mid} \; \text{else} \; \text{low} := \text{mid} + 1 \\
\text{length} & := \text{low} \\
\text{if} \; \text{length} > 2p \; \text{then} \; \text{length} := \infty
\end{align*}
\]

Why \( 2p + 1 \)?

- \( 2p + 1 \) is more than the length of a closed path can be
  \[ \Rightarrow \text{there is no closed cycle, there is no connection from everywhere to everywhere} \]
- necessary:
- suffices: if edge \((u, v)\) is travelled twice in the direction \( u \to v \), then both uses of the edge can be removed by reversing the sub-path \( v_1, \ldots, u_2 \)
Why does it work?

- all the time it holds \( low \leq \text{precise} \leq high \)
- terminates, when \( low \geq high \)
- during each iteration, \( low \) increases or \( high \) decreases

\[ \Rightarrow \text{the computation terminates, and then} \quad low = high \]

After that, if \( length < \infty \), (some) shortest cycle may be found like this:

\[
\begin{align*}
\text{for} \quad (u,v) & \in Edges \quad \text{do} \\
& \quad \text{remove} \quad (u,v) \quad \text{from the map} \\
& \quad \text{if} \quad \neg A(length) \quad \text{then} \quad \text{return} \quad (u,v) \quad \text{to the map}
\end{align*}
\]

Why does it work?

- always in the beginning of the body of the loop there is a cycle of the length \( length \)

\[ \Rightarrow \text{at least one such a cycle remains} \]

- \( (u,v) \) is removed permanently, except if it is in every remaining cycle of the right length

- if an edge is in e.r.c.o.t.r.l. now, it is in e.r.c.o.t.r.l. also always in the future
⇒ in the end every edge is in e.r.c.o.t.r.l.
⇒ only one cycle remains
  • more than one edge could be removed at the same time
    – how?
⇒ If the numbers of vertices and edges are $V$ and $E$, and the running time of the algorithm $A$ is $A(V, E, p)$, then a shortest cycle can be found in time

$$(E + \log_2 p + 3)A(V, E, p)$$

If $A(V, E, p)$ is a polynomial of $V$, $E$, and $\log_2 p$, then also the time consumption as a whole is

• $\log_2 p$
  
  = the number of bits needed to represent $p$
  
  = a reasonable measure of the size of $p$

The same formula used in the reverse direction: if finding a shortest cycle requires exponential time, then so does also the finding of the answer to the question “is there a cycle, whose length is at most $p$”
1.4 Taking advantage of reduction

An obvious benefit: reduction may yield a practical algorithm for the reduced problem

- we will not discuss this now

Another advantage: reduction makes it often possible to reason that it would not be wise to try to find an algorithm for the target problem that is more efficient than a certain limit

- we will soon discuss an example where this way of thinking reveals that there cannot be an algorithm with certain features

Another use: if the literature discusses problem \( P \) a lot but only gives rather bad algorithms for it, and if it is easy to find an efficient reduction from \( P \) to a problem \( M \) that is of interest to me, then it is very likely that I will not invent or find from the literature an efficient algorithm for \( M \)

- such an algorithm would give a good algorithm for \( P \)

\[ \Rightarrow \text{if such an algorithm had been found or were easy to find, then the literature would present a good algorithm for } P \text{ based on it} \]
An example: how efficient a data structure it is reasonable to try to find for the problem “does a given rectangle intersect any rectangle in a given set?”

- let the size of the set be $n$
- solved in $\Theta(n)$ time by comparing the new rectangle to each rectangle in the set one at a time
- the literature discusses a lot the problem “does a given interval of the real line intersect any interval from a given set”
- the interval tree is suggested as the “standard solution”

$\Rightarrow$ the running time of searching, inserting, and deleting an interval is $O(\log n)$

- the interval problem is reduced to the rectangle problem by replacing the intervals by rectangles that have the same $y$-coordinates

$\Rightarrow$ it would not be reasonable to try to find a better solution to the rectangle problem than $O(\log n)$

- this reasoning does not promise that even that good an algorithm is found—it may be that the rectangle problem is strictly more difficult than the interval problem
1.5 An example: the sliding median

Signal processing people in particular are interested in the following problem:

- $2w + 1$ is a constant known as window size
  - for simplicity we assume that the window size is an odd integer
- in comes the sequence $a_1, a_2, \ldots$ of numbers
- each time when $a_{i+2w}$ has arrived, the algorithm must print the median of $a_i, a_{i+1}, \ldots, a_{i+2w-1}, a_{i+2w}$

It is easy to store $a_i, \ldots, a_{i+2w}$ in a ring buffer:

```c
double buffer[ 2*w + 1 ]; int place = 0;
void insert( double x ){
    buffer[ place ] = x; ++place; place %= 2*w + 1;
}
```

Would it be a good idea to find the median by taking a copy of the buffer, sorting it, and printing the middlemost element?

- $O(w \log w)$ work for each median
A linear ($O(w)$) time algorithm for finding the median is known

- a theoretically interesting algorithm guarantees linear worst-case time
- “lazy quicksort” is very practical, and its average time consumption is linear

The “hourglass algorithm” gives a result that is still much better

- the data values are still kept in a ring buffer
- there are two heaps, each of size $w + 1$, that share their root
- one of them holds indices to those places in the ring buffer that contain the $w + 1$ smallest elements, and it is ordered with the biggest at root
  - indices are compared by comparing the elements that they point to
- similarly, the other holds indices to the $w + 1$ biggest elements, with the smallest at root

$\Rightarrow$ the median is at the joint root of both heaps

- we represent the heaps with one array `heaps[]` whose indices are $-w \ldots w$
  - the index 0 is shared by the heaps, and the roots are there
• furthermore, the ring buffer contains the index `to_heap[ place ]` to the place in the pair of heaps that corresponds to the element

• the new element is written into its place in the ring buffer, after which the heap where it belongs to is fixed

• if the fixing moves the element to the root, then fixing is continued at the other heap

```c
int to_heap[ 2*w + 1 ], heaps[ -w ... w ];

void insert( double x ){
    buffer[ place ] = x;
    int i1 = to_heap[ place ], i2 = (i1-1) / 2;
    while( i1 > 0 && x < buffer[ heaps[ i2 ] ] ){
        heaps[ i1 ] = heaps[ i2 ]; to_heap[ heaps[ i1 ] ] = i1;
        i1 = i2; i2 = (i1-1) / 2;
    }
    ... 
    heaps[ i1 ] = place; to_heap[ place ] = i1;
    ++place; place %= 2*w + 1;
}
```
• \( \forall i; 0 \leq i < 2w+1: \text{heaps[ to_heap[i] ] } = i \)

• \( \forall i; -w \leq i \leq w: \text{to_heap[ heaps[i] ] } = i \)

• running time \( O(\log w) \)!

We have improved from \( O(w \log w) \) running time to \( O(\log w) \)

Can we get a still better result?

Any algorithm for the sliding median can be used to sort the array \( A[1], \ldots, A[w+1] \) as follows:

• the smallest and biggest elements \( \text{min} \) and \( \text{max} \) of \( A \) are found in \( O(w) \) time

• the sliding median algorithm is given \( w \) copies of \( \text{min} \), then \( A[1], \ldots, A[w+1] \), and finally \( w \) copies of \( \text{max} \)

• after giving \( i - 1 \) copies of \( \text{max} \), the buffer contains \( w - i + 1 \) copies of \( \text{min} \); \( A[1], \ldots, A[w+1] \); and \( i - 1 \) copies of \( \text{max} \)

• their median is the \( i \)th smallest element of \( A \)

\( \Rightarrow \) the algorithm returns the elements of \( A \) in increasing order
If computing the sliding median takes \( O(f(w)) \) time per median on the average, then the algorithm as a whole takes \( O(w) + O(w f(w)) \) time.

It is known that every *comparison-based* sorting algorithm takes \( \Omega(w \log w) \) time.

\[ \Rightarrow \text{Every comparison-based algorithm for the sliding median takes } \Omega(\log w) \text{ time.} \]

If you need a sliding median algorithm that uses constant time per median, do not waste your time on comparison-based attempts.
1.6 One time consumption in a reduction

The time consumption of an algorithm that is based on reduction depends on

- the running time of the reduction (without the calls of the target algorithm)
- the running time of the target algorithm
- the number of times the target algorithm is called
- how the size of the problem changes in the reduction

The last factor is easily forgotten!

If

- the size of the problem is squared in the reduction
- the running time of the reduction is $\Theta(n^3)$
- the running time of the target algorithm is $\Theta(n^4)$
- the target algorithm is called once

then the running time of the algorithm that is based on reduction is

$\Theta(n^3 + (n^2)^4) = \Theta(n^8)$!
The running time of the reduction (without the calls of the target algorithm) sets an upper bound to both the number of calls and the size of the input of the target algorithm.

⇒ If the running time of the target algorithm is \( O(g(n)) \) and the running time of the reduction is \( O(f(n)) \), then a pessimistic estimate of the running time of the algorithm as a whole is \( O(f(n)g(f(n))) \)

⇒ If \( f \) and \( g \) are polynomials, then also the total running time is \( O(\text{polynomial}) \)
1.7 The significance of polynomial time

The most important distinction in the theory of computational complexity is polynomial time vs. others.

*Polynomial time* means that the running time is bounded from above by some polynomial of the size of the input.

In other words,

- let the size of the input be $n$, and denote the running time with $t(n)$
- there is a polynomial $P(n)$ so that $\forall n : t(n) \leq P(n)$
- the same can be defined in another, equivalent way: $\exists k : t(n) = O(n^k)$
- and also like this: $\exists c : \exists d : \forall n : t(n) \leq n^c + d$
  - choose $d = \max\{t(0), t(1)\}$
  - when $n \geq 2$, then for big enough $c$ it holds $a_i n^i \leq \frac{1}{k} n^c$ for $1 \leq i \leq k$
The distinction is not the same as the fast against the slow

- $n^{100}$ grows quickly unmanageably big
- even $O(n^2)$ may be annoyingly slow with big $n$
- the worst-case running time of some exponential algorithms realizes so seldom that it has next to no practical significance
  - e.g., simplex is almost always fast in practice
- in many cases, an exponential algorithm can be made to run sufficiently quickly for sufficiently big inputs
  - e.g., solving sudokus

The distinction is, however, a useful **approximation** to the distinction between the fast and the slow
The distinction is preserved when algorithms are combined

- this does not apply, for instance, to the distinction $O(n^2)$ vs. $\omega(n^2)$
  - in the example above, $\Theta(n^3)$ reduction and $\Theta(n^4)$ target algorithm yielded $\Theta(n^8)$ total time
  - if an algorithm calls a $\Theta(n^2)$ algorithm $n^2$ times with an input of the original size, and is otherwise $O(n^2)$, the total running time is $\Theta(n^4)$

- even so, a polynomial time reduction that calls the target algorithm a polynomial number of times yields a polynomial total time

⇒ Reduction from one problem to another can be used when investigating the distinction between polynomial time and others

- for this to work, the running time of the reduction must be polynomial

⇒ The distinction has made it possible to get useful results
2 Nondeterministic running time

2.1 Why discuss such an odd thing?

Often checking a solution is—or at least seems—much easier than finding a solution

- the solution of the queens’ problem
- the solution of the “Pulmakulma” pentomino puzzle
- checking a given cycle on a map and counting its length

This idea is made mathematically precise with the notions of *nondeterministic running time* and *nondeterministic memory usage*

The notions are a bit hard to grasp because of at least two reasons

1. in addition to proper solutions, there is often also the possibility that there are no solutions, which cannot be checked in the same way as solutions

2. the question is often of the “yes” / “no” form, so more than just the answer is needed for the checking
   - the answer is then just “yes”
There often seems to be an asymmetry between the possibilities “yes” and “no”, or “the solution is this” and “there is no solution”

- it is easy to check the solution to “Pulmakulma”, but how to verify that there is no solution?

If our theory is to be any good, it must be able to discuss yes/no-problems, because that is the simplest form of output

⇒ We will first develop the theory for yes/no-problems, and then extend it to more general problems where a proper solution is sought

- yes ∼ solution

- no ∼ there is no solution
2.2 Problems, decision problems, and instances

A problem usually (but not necessarily) has an input that may get an infinite number of different values

- “Is this number a prime?”: the input is a natural number
- “What is the sum of these two integers?”: the input consists of two integers
- “Does a given map contain a cycle that visits each city at least once, and whose length is at most a given natural number?”: the map and natural number

When necessary, we may emphasize what is the input and what is its type

- **input**: map $G$ and natural number $n$
- **question**: does $G$ contain a cycle that visits each city at least once, and whose length is at most $n$?

For brevity, we will most of the time not do so, but denote the input with phrases like “in a given map” or “this number”

- however, when there is risk of misunderstanding, it may be a good idea to emphasize the input
The *instance* of a problem is what was just called the input

An algorithm for the problem reads the instance and produces the answer

- the instance is thus the input to the solution algorithm

A *decision problem* is a problem whose answer is “yes” or “no”

- “What is the shortest cycle” is not a decision problem
- “Is there any cycle, whose length is at most 23 456 km” is a decision problem
2.3 Uninteresting unless infinitely many different instances

If a problem only has a finite number of different instances, then a fast solution algorithm certainly exists

- just write all the answers as such in an array or the program code

```python
if number = 0 then return "no"
else if number = 1 then return "no"
else if number = 2 then return "yes"
else if number = 3 then return "yes"
else if number = 4 then return "no"
else if number = 5 then return "yes"
else if number = 6 then return "no"
else if number = 7 then return "yes"
else if number = 8 then return "no"
else if number = 9 then return "no"
```

⇒ the situation is trivial from the point of view of the theory of algorithms and computation
However, this algorithm is usually not practical

- far too big a program that may be difficult to write

It is not “philosophically” satisfactory either

- the algorithm does not “find” the answer “all by itself”, but only prints an answer that has been given to it as ready

- compare:

  - **problem:** are there positive integers \( x, y, z, \) and \( n \) such that \( n \geq 3 \) and \( x^n + y^n = z^n \)

  - **algorithm:**

    ```cpp
    int main(){ std::cout << "No.\n"; }
    ```

\[ \Rightarrow \] To get a useful theory, this algorithm must somehow be ruled out
“Finding the answer all by itself” is, however, a vague notion

- if a program that computes \( \sin 0.873 \) picks \( \sin 0.87 \) and \( \sin 0.88 \) from a table and then interpolates, does it find the answer “all by itself”?

- if a primality testing program checks small numbers against a ready-made table of small primes, and big numbers by dividing them with the primes in the table one by one, does it find the answer “all by itself”?

⇒ It is not easy to make the requirement “must find the answer all by itself” precise

If, however, the number of different instances is infinite, then the program cannot contain a ready-made answer for each of them

- the concept of “program” is subject to the requirement that its size is finite

⇒ The algorithm must then find the answer “all by itself” at least in some sense, for infinitely many instances
That there are infinitely many instances is thus a way of ruling out the above-mentioned trivial uninteresting algorithm

- does it rule any interesting algorithms out or leave other uninteresting algorithms in is another question
- in practice, it focuses the attention to interesting algorithms pretty well, although not perfectly
- it is also the case that there are usually very many possible instances—more than we are willing to list—and infinitely many is a good approximation of very many

(Perhaps one could start to develop an alternative theory by defining that a program produces something “all by itself”, if it answers more yes/no-questions than its code has bits)
This approach (and its above-mentioned alternative) has also the problem that it pays no attention to the question how difficult it is to verify that the algorithm produces correct results

• it took more than 300 years to prove correct the algorithm that was given above for the “$x^n + y^n = z^n$” problem

• it is known that the problem “Can every even integer that is greater than two be represented as a sum of two primes” is solved by one of the following two algorithms, but it is still not known which one
  – **algorithm 1:** int main(){ std::cout << "Yes.\n"; }
  – **algorithm 2:** int main(){ std::cout << "No.\n"; }

⇒ it is difficult also for a program that is based on ready-made answers!
2.4 Formalization 1: certificates and verification algorithms

The goal of certificates is to apply the idea of easy checking of a solution to decision problems

- nontrivial, because mere “yes” is usually not easy to check as such

Every “yes”-instance has an associated certificate, with which the answer can be verified in a manner agreed beforehand

- the certificate may be, but it need not be, the solution to the corresponding search or similar problem

- e.g., the certificate for the decision problem “is there a cycle whose length is at most this” could be a cycle that meets the requirements

- the word “certificate” also applies to the cases where the verification fails

⇒ a certificate is valid or invalid

Of course, there must be no valid certificates for “no”-instances
How the certificate is verified has been chosen beforehand and represented as an algorithm that inputs an instance $\sigma$ and a certificate $\xi$, and answers correctly either “pass” or “fail” as follows:

- **pass:** The certificate $\xi$ is valid. Therefore, the answer to the problem with the input $\sigma$ is “yes”.

- **fail:** The certificate $\xi$ is not valid. Therefore, we did not learn whether the answer to the problem with the input $\sigma$ is “yes” or “no”. It may be that some other certificate is valid for this input, or it may be that the correct answer is “no”.

We call the above algorithm a *verification algorithm*

Let us write

- $\varphi = \text{the decision problem in question}$
- $\Sigma^* = \text{the set of all possible inputs to } \varphi$
  - that is, the set of all entities whose form is suitable as input to $\varphi$
- $\varphi(\sigma)$ means that the answer to $\varphi$ with the input $\sigma$ is “yes”
- $\neg\varphi(\sigma)$ means that the answer to $\varphi$ with the input $\sigma$ is “no”
The notation $\Sigma^*$ comes from the fact that in theoretical computer science, the input is thought of as a finite sequence of symbols from some finite set $\Sigma$.

- $\Sigma$ is known as the alphabet.
- It may be easier for an algorithm person to think of the input as a structured entity, such as the pair (map, number).
- In a real computer, the input is usually ultimately represented as a sequence of 8-bit bytes.

$\Rightarrow$ thinking of the input as a sequence of symbols is faithful to how it actually is.

Let further:

- $\Xi^* = \text{some set}$
- $A : \Sigma^* \times \Xi^* \rightarrow \{ \text{“fail”}, \text{“pass”} \} = \text{some algorithm}$
- $A(\sigma, \xi)$ means that $A$ replies “pass” with the input $\sigma$ and $\xi$.
- $\neg A(\sigma, \xi)$ means that $A$ replies “fail” with the input $\sigma$ and $\xi$. 
With this notation, $A$ is a verification algorithm for $\vartheta$, if and only if $\forall \sigma \in \Sigma^*$:

- $\vartheta(\sigma) \rightarrow \exists \xi \in \Xi^* : A(\sigma, \xi)$
- $\neg \vartheta(\sigma) \rightarrow \forall \xi \in \Xi^* : \neg A(\sigma, \xi)$

Here $\Xi^*$ is, of course, the set of all certificates (= entities that are of suitable form as a certificate).

A **certificate system** of a decision problem $\vartheta$ consists of the set $\Xi^*$ and algorithm $A$ described above.

An algorithm $B(\sigma)$ that solves $\vartheta$ can be thought of as a verification algorithm that always uses the empty (or just any) sequence as the certificate.

- “yes” is interpreted as “pass”
- “no” is interpreted as “fail”

$\Rightarrow$ An algorithm that solves a problem is a special case of a verification algorithm.

$\Rightarrow$ A verification algorithm is only interesting if it is more efficient than known solution algorithms.
2.5 Polynomial time certificate systems

The certificate system described above is polynomial time, if and only if $A$ runs in polynomial time with respect to $|\sigma|$

- that is, there are constants $c$ and $d$ such that the running time of $A$ with the input $\sigma$ and $\xi$ is at most $|\sigma|^c + d$

The allowed running time does not depend on $|\xi|!$

- the goal is to measure how demanding $\vartheta$ is
- if the time consumption were allowed to depend on $\xi$, we could “buy” as much computation time as we wish by giving a very long $\xi$ that consists mostly of meaningless information as a certificate

It follows from the requirement that every “yes”-instance has a shortish valid certificate

- otherwise the time allowed for $A$ does not suffice for reading the valid certificate
- to be more precise: there are numbers $c$ and $d$ such that every “yes”-instance has a valid certificate whose length is at most $|\sigma|^c + d$

$$\exists c : \exists d : \forall \sigma : \vartheta(\sigma) \rightarrow \exists \xi : A(\sigma, \xi) \land |\xi| \leq |\sigma|^c + d$$
Also the answer “fail” must come in the allowed time

- that is, \(A\) notices in the allowed time that the certificate is invalid, instead of letting it fool itself to compute for too long

Even so, the certificate system is asymmetric with respect to “yes” and “no”

- running \(A\) once may yield \(\vartheta(\sigma) = \text{yes}\) or \(\vartheta(\sigma) \text{ was left open}\) as the result, but cannot yield \(\vartheta(\sigma) = \text{no}\)

- the answer “no” may be obtained by running \(A(\sigma, \xi)\) with every short enough certificate \(\xi\), of which there are usually very many

**Definition**

\(\text{The set } \textbf{NP} \text{ is the set of those decision problems that have a polynomial time certificate system.}\)
Comments on the definition

- a “yes”-instance can always be checked in polynomial time (if a valid certificate is known)
- a “no”-instance does not necessarily have an efficient checking method
- there is not necessarily a polynomial time method to find the answer without knowing a valid certificate
- there is not necessarily a polynomial time method to find a valid certificate, even if it is known that the answer is “yes”

“NP” comes from the words “nondeterministic polynomial time”

NP is most likely asymmetric with respect to “yes” and “no”

- co-NP is the set of those decision problems that are obtained from the problems in NP by swapping the answers “yes” and “no”
- e.g., “is every cycle that visits every city at least once in the given map longer than a given number”
- it looks like NP \neq co-NP, but this is not known for certain
it also looks obvious that $\text{NL} \neq \text{co-NL}$, but it has been proven that $\text{NL} = \text{co-NL}$!

- $\text{NL}$ = decision problems that have a certificate system that runs in logarithmic memory

Let us also define the set $P$

**Definition**

*The set $P$ is the set of those decision problems that have a polynomial time solution algorithm.*

⇒ “$P$” is a rather simple and natural concept

It is immediately obvious that $P = \text{co-P}$

- it suffices to swap `std::cout << "yes";` and `std::cout << "no";` in the solution algorithm

- $P$ is not asymmetric with respect to “yes” and “no”

Because a solution algorithm can be thought of as a verification algorithm, we get immediately the results $P \subseteq \text{NP}$ and $P \subseteq \text{co-NP}$
This result is not a great achievement!

- it is an immediate consequence from the basic definitions
- if it did not hold, we would not have changed our idea of how the classes of problems are related to each other, but we would have fixed the basic definitions

⇒ the result did not bring new knowledge but acted as one test on whether building the theory has started in an appropriate way

Immediately—and also after almost 40 years of research—it seems natural to guess that $P \neq NP$, but it is not known for sure

To avoid confusion, please be warned that we will later introduce two new concepts $NP$-hard and $NP$-complete that are not the same as $NP$, although are related
2.6 Formalization 2: nondeterministic running time

The set \( \text{NP} \) can be defined equivalently also in terms of nondeterministic computation.

A *nondeterministic algorithm* is otherwise like an ordinary algorithm, but it may also contain *nondeterministic selection statements*:

- a nondeterministic selection statement has a finite number of alternative branches
- it is executed by executing one of its branches—it may be just any one
- if the same nondeterministic selection statement is executed many times, the branch is always chosen independently of earlier choices
- the choice of the branch is later called briefly as a “nondeterministic choice”
A nondeterministic algorithm $C(\sigma)$ “processes” a decision problem $\vartheta(\sigma)$ in time $f(|\sigma|)$, if and only if

- independently of what nondeterministic choices $C$ makes, it terminates in time $f(|\sigma|)$
- when terminating, $C$ answers either “yes” or “left open”
- if the correct answer is “yes”, then there is at least one sequence of nondeterministic choices that makes $C$ answer “yes”
- if the correct answer is “no”, then every sequence of nondeterministic choices makes $C$ answer “left open”

The commonly used technical word is that the algorithm accepts the language corresponding to $\vartheta$

It is often said that $C$ guesses how it must make the nondeterministic choices

- this way of speaking does not emphasize sufficiently that in the end $C$ tells whether it guessed right often enough or not
- a wrong guess does not mean that the correct answer is “no” but that the correct answer was left open
**NP** is now defined as the set of those decision problems that have a polynomial time nondeterministic algorithm.

It is easy to see that this definition is equivalent to the definition based on certificates:

- Let the certificate correspond to the sequence of nondeterministic choices.

  ⇒ $A$ is otherwise the same as $C$ except that where $C$ makes a nondeterministic choice, $A$ reads from the certificate which branch it must choose.

  ⇒ the same running time.

- "yes" is a possible answer ⇔ there is at least one sequence of nondeterministic choices that makes $C$ answer "yes" ⇔ there is at least one certificate that makes $A$ answer "yes".
3 The “hardest side” of NP

3.1 An open question

There are many search, optimization, and other problems, for which it is easy to find a polynomial time certificate system, but extremely difficult to find a polynomial time solution algorithm.

⇒ The question arises, do they have polynomial time solution algorithms?

The doubt is supported by the fact that it has been known for a long time that:

- rather small increase to the allowed running time makes the set of decision problems that can be solved in that time strictly grow
- there are decision problems that require exponentially, doubly exponentially or even more time

The question whether they have polynomial time algorithms is still open.
It is literally a million dollar question!

- because of known reductions we know that it is the same question as “does $P = NP$ hold”
- one of the Millennium problems of Clay Mathematics Institute
3.2 Cook’s theorem

The central processing unit of a computer can be thought of as working according to the simplified picture

Let many copies of the central processing unit be taken such that the results of no computation step are stored in the copy but, instead, are given as input to the next copy. This kind of a circuit makes the same computation as the computer, although it does not have registers or memory.

- in practice this kind of “unfolding the hardware” would cause a lot of costs and no benefit
- now, however, we are seeking for a theoretical result
Unfolding is just copying of ready-made units and information

⇒ its cost is roughly proportional to the size of the result

To execute an algorithm that uses $O(n^m)$ memory and $O(n^t)$ time in this way, $O(n^t)$ copies suffice, each of size $O(n^m)$

⇒ $O(n^{m+t})$ circuit elements suffice

• a polynomial number
Let the unfolding be done so that

- the memory contains the $A$ and $\sigma$ of a certificate system for a decision problem $\vartheta$ to start with
- the memory has input wires for $\xi$
- a wire is taken out of the memory bit that represents the answer by $A$

$\Rightarrow$ We got a memoryless digital circuit (a combinatorial circuit), whose
- input wires receive $\xi$ encoded in binary
- the output wire has value “1” if and only if the input is a valid certificate for $\sigma$

$\Rightarrow$ We obtained a combinatorial circuit whose output can be made “1” by choosing the input in a suitable way if and only if the answer to the problem $\vartheta$ with the instance $\sigma$ is “yes”

If the starting point is a polynomial time certificate system, then the time spent on the reduction and the size of the result are polynomial with respect to $|\sigma|$
The problem “does the given combinatorial circuit with one output wire have such an input that the output value is 1” is known as *circuit satisfiability*.

⇒

*If circuit satisfiability has a polynomial time algorithm, then every problem in NP has a polynomial time algorithm.*

As a matter of fact, we showed how an arbitrary polynomial time verification algorithm can be reduced to circuit satisfiability in polynomial time.

A polynomial time algorithm for circuit satisfiability would prove that $P = NP$.

⇒ would imply polynomial time algorithms for numerous problems, for which polynomial time algorithms have been sought in vain.

Circuit satisfiability is itself in NP.

- the output of a combinatorial circuit is easy to compute, when the input is given.

⇒ the input acts as a certificate of a polynomial time certificate system.

- if circuit satisfiability were not in NP, then it could not have a polynomial time algorithm, because $P \subseteq NP$. 

46
Circuit satisfiability is in the “hardest side” of $\textbf{NP}$

Stephen Cook published a result of this kind in 1971

- instead of circuit satisfiability he had the satisfiability of formulae in propositional logic
- instead of an ordinary computer he had the Turing machine
3.3 A simpler problem in the “hardest side”: 3-CNF

The previous result does not necessarily seem earth-shaking, because the problem that was shown hard is of the form “what can be computed with a certain very expressive method”

Now we will show a couple of results, according to which also problems of the form “what can be obtained with a method that seems weak at start” contain difficult ones.

A propositional logic formula is in conjunctive normal form, if and only if it is made of variables and operators “¬”, “∨”, and “∧” as follows:

- the formula as a whole is of the form \( \text{clause} \land \cdots \land \text{clause} \)
- each \( \text{clause} \) is of the form \( \text{literal} \lor \cdots \lor \text{literal} \)
- each \( \text{literal} \) is of the form \( P \) or \( \neg P \), where \( P \) is a variable
- e.g., the following formula is in conjunctive normal form:
  \[(P \lor \neg R) \land (Q \lor \neg R) \land (\neg P \lor \neg Q \lor R)\]

A formula in conjunctive normal form yields true if and only if its every clause yields true.
Next we show that circuit satisfiability can be reduced in polynomial time to the problem “does a given formula in conjunctive normal form have a combination of variable truth values such that the truth value of the formula is true”

One variable is introduced for every wire in the circuit

For every “not”-gate the subformula \((P \lor Q) \land (\neg P \lor \neg Q)\), is written, where

- \(P\) is the variable that corresponds to the input wire of the gate
- \(Q\) is the variable that corresponds to the output wire of the gate

For every “and”-gate the following subformula is written

\[(P \lor \neg R) \land (Q \lor \neg R) \land (\neg P \lor Q \lor R)\]

Subformulae for other types of logic gates are constructed according to the same principle

⇒ The subformula that corresponds to a logic gate has the value true if and only if the value of the output variable is what the gate yields with the values of the input variables

- “0” ~ false and “1” ~ true
Also $P_{out}$ is taken as a subformula (as such), where $P_{out}$ is the variable that represents the output wire of the circuit as a whole

The subformulae are combined with the operator “$\land$”

⇒ If there is an input which makes the output of the circuit “1”, then the truth value of the formula can be made true by choosing the values of the variables as follows

• the variables that correspond to the input wires of the circuit are set according to that input

• the output variable of each gate is set according to the input variables and the operation of the gate

In the opposite case, the value of the formula is false independently of the values of the variables

• if $P_{out}$ is false, then the clause $P_{out}$ yields false

• otherwise the output variable of at least one gate is in contradiction with the input variables and operation of the gate

⇒ one of the clauses that corresponds to the gate yields false
We have shown: there is a combination of the values of the variables which makes the formula true ⇔ there is a combination of the values of the input wires of the circuit which makes the value of the output wire 1.

This reduction is a simple transformation that only affects the size of the instance by a constant factor.

⇒ clearly polynomial time

Therefore, if this problem has a polynomial time algorithm, then there is a polynomial time algorithm for circuit satisfiability and, furthermore, for every problem in NP, and P = NP.

We will fine-tune the target problem of the reduction a bit further: we will ensure that every clause consists of precisely three literals, and they use different variables:

- also until now the variables have been different, but the number of literals has varied from one to three
- because a literal is a variable or its negation, also \( P \lor \neg P \lor Q \) is forbidden
- well, it would not be useful, because it is always true.
Why to make this more complicated by requiring \textit{precisely} three instead of at most three?

Because “precisely three” makes the problem potentially less general

- every algorithm that solves the “at most three” problem also solves the “precisely three” problem
- the opposite is not obvious (but will become obvious due to the reasoning that will be presented soon)
- the more restricted problem is proven to belong to the “hardest side”, the stronger is the result and the easier it is to continue reasoning from it
“Precisely three” can easily be made hold without changing the meaning of the formula as follows

- choose any two variables $P$ and $Q$ that are not $P_{out}$ and are not the same variable
  - if necessary, new variables are introduced
- the clause $P_{out}$ is replaced by the subformula
  \[(P \lor Q \lor P_{out}) \land (P \lor \neg Q \lor P_{out}) \land (\neg P \lor Q \lor P_{out}) \land (\neg P \lor \neg Q \lor P_{out})\]
  - the new subformula $\Leftrightarrow P_{out}$
- the subformulae $\ell_1 \lor \ell_2$ that only have two literals are replaced by the subformula
  \[(\ell_1 \lor \ell_2 \lor P) \land (\ell_1 \lor \ell_2 \lor \neg P)\]
  where $P$ is a variable that does not occur in the literals $\ell_1$ and $\ell_2$

We have reduced circuit satisfiability, and thus every problem in $\textbf{NP}$, to the problem

"Does a given formula in conjunctive normal form whose each clause contains precisely three literals and they have different variables, have such a combination of truth values of the variables that the truth value of the formula as a whole is true?"
This decision problem is known as **3-CNFSAT**

- CNF $\sim$ conjunctive normal form
- SAT $\sim$ satisfiability

Therefore, if 3-CNFSAT has a polynomial time algorithm, then every problem in **NP** has a polynomial time algorithm, and $P = NP$

Also this result is from Cook’s publication of 1971

To ponder

- can the following problem be solved in polynomial time: “does a given formula in disjunctive normal form whose each part that has been composed with $\land$ contains precisely three literals and they have different variables, have such a combination of truth values of the variables that the truth value of the formula as a whole is **true**”
  - that is, otherwise like 3-CNFSAT, but “$\land$” and “$\lor$” have been swapped

- can 2-CNFSAT be solved in polynomial time?
  - that is, otherwise like 3-CNFSAT, but each clause has precisely two literals
### 3.4 From logic to graphs . . .

Next we show that 3-CNF-SAT can be easily and quickly reduced to the problem “does a given undirected graph have a set of $k$ vertices such that from each vertex of the set there is an edge to each other vertex of the set”

- such a set of $k$ vertices is known as a $k$-clique

An entertaining problem: construct two graphs from a set of six people as follows

- it is assumed that the relation “$A$ and $B$ know each other” is symmetric
- an edge is drawn between two people in graph 1, if and only if those people know each other
- an edge is drawn between two people in graph 2, if and only if those people do not know each other
- show that at least one of these two graphs has a 3-clique
3-CNF-SAT is reduced to the clique problem by making the graph, where

- there is a vertex for each literal in each clause
- an edge is drawn between two vertices if they are from different clauses and they either correspond to different variables or are identical
- that is, the edge is not drawn if one literal is $P$ and the other is $\neg P$

The value of $k$ is the number of clauses

$\Rightarrow$ the graph has $3k$ vertices

An example:

$$(P \lor Q \lor R) \land (\neg P \lor Q \lor S) \land (\neg Q \lor \neg R \lor \neg S) \land (P \lor R \lor S)$$

- please do not confuse the $P$, $Q$, $R$, and $S$ in this example with the use of the same symbols as variables in general
\[(P \lor Q \lor R) \land (\neg P \lor Q \lor S) \land (\neg Q \lor \neg R \lor \neg S) \land (P \lor R \lor S)\]

If the graph has a \(k\)-clique, then the formula yields \textbf{true} if the values of the variables in the nodes of the clique are chosen so that vertex \(P\) sets \(P\) to \textbf{true} and vertex \(\neg P\) sets \(P\) to \textbf{false}

- the same variable cannot be set to both \textbf{true} and \textbf{false}, because no edge was drawn between \(P\) and \(\neg P\)
- if the value of some variable is left unspecified, it may be chosen freely

\[\Rightarrow\] the truth values were chosen such that every clause yields \textbf{true}

\[\Rightarrow\] the formula has a combination of the values of the variables that makes the formula \textbf{true}
If the formula has a combination of the values of the variables that makes the formula \textbf{true}, then a $k$-clique is found by choosing from every clause a literal whose value is \textbf{true} and then choosing the corresponding vertex:

- the chosen vertices cannot contain the pair $P$ and $\neg P$

$\Rightarrow$ they constitute a $k$-clique

$\Rightarrow$ The graph has a $k$-clique if and only if the answer to 3-CNF-SAT is “yes”

That is, we have reduced 3-CNF-SAT to the clique problem.

Therefore, if the clique problem has a polynomial time algorithm, then every problem in \textbf{NP} has a polynomial time algorithm, and $\textbf{P} = \textbf{NP}$

Although the clique problem seems restricted and narrow, it is in some sense capable of solving a great number of problems of many kinds!
3.5  ... and further

The clique problem can be further reduced in polynomial time in a small number of steps to the following problem, among others:

“Can a given set of natural numbers be divided to two parts such that the sum of the numbers in part 1 = the sum of the numbers in part 2?”

3-CNF-SAT can be reduced in polynomial time to the problem

“Does a given undirected graph have a hamiltonian cycle?”

and further

“Does a given map have a cycle that visits every city at least once, and whose length is at most a given number?”

- a hamiltonian cycle is a closed path that visits every vertex precisely once
By reducing problems to more and more problems in this way, a tree is constructed, whose root is circuit satisfiability

- if any problem in the tree has a polynomial time algorithm, then every problem in \( \texttt{NP} \) has, and \( \texttt{P} = \texttt{NP} \)

Already in 1979 the tree contained hundreds of problems

4 NP-hardness and NP-completeness

4.1 Making the notion of reduction more strict

Polynomial time reducibility of a problem in the tree discussed above to a given problem is an important property

• tells that if the given problem has a polynomial time algorithm, then every problem in \( \text{NP} \) has a polynomial time algorithm, and \( \text{P} = \text{NP} \)

• for many problems in \( \text{NP} \), a polynomial time algorithm has been sought for long and intensively, but in vain

\( \Rightarrow \) it is likely that they, and thus also the given problem, do not have polynomial time algorithms
The reductions used in Section 3—like all reductions in the theory of \textbf{NP}-completeness—are more restricted than the reductions introduced in Section 1:

- the target algorithm is only called once
- the result of the target algorithm is given as such as the answer
- logarithmic memory suffices for executing the reduction

In principle, with the less resources the reduction succeeds, the more the existence of the reduction tells about the complexity of the target problem:

- reducing a problem in \textbf{NP} in exponential time does not tell much about the complexity of the target problem, because that time suffices for finding the answer and thus reducing to the low-complexity problem “is the string given as the input the same as ‘yes’”
- because of this it is surprising that the theory of \textbf{NP}-completeness can be built with rather weak reductions and still covers so many decision problems
Running in logarithmic memory guarantees running in polynomial time

- polynomial time suffices for going through all bit combinations that fit in logarithmic memory

⇒ the algorithm terminates then at the latest, because otherwise it would re-enter a state that it has already visited, and would thus enter an eternal loop

- originally the definitions were written for polynomial time reduction algorithms, but later researchers started to use logarithmic memory, when it was realized that it, too, suffices

- other definitions than that of the reduction (in particular, the definition of \( \mathbf{NP} \)) still use polynomial time

- if logarithmic memory is replaced for polynomial time in the definition of \( \mathbf{NP} \), the set of decision problems called “\( \mathbf{NL} \)” is obtained

⇒ If logarithmic memory feels hard to understand, you may all the time think of polynomial time instead, as long as you remember that the current practice is slightly different
The notion of the reduction algorithm can thus be made more restricted by requiring that the reduction algorithm does not call the target algorithm, but

- takes as its input the input to the original decision problem
- produces a string that is suitable as input to the target problem, and to which the answer of the target problem is the same as the answer of the original problem to the original input
- runs in logarithmic memory

**Definition**

*Algorithm A is a logarithmic memory reduction of a decision problem ϑ to the decision problem χ, if and only if*

- A reads an input that is appropriate to ϑ and produces an appropriate input to χ,
- $\chi(A(\sigma)) \Leftrightarrow \vartheta(\sigma)$ and
- A runs in logarithmic memory.
It is worth noticing that

• \( \chi \) and \( \vartheta \) are in a symmetric relation in that “if and only if” is required between their answers

• however, the definition as a whole is asymmetric, because the input to \( \vartheta \) is converted to an input to \( \chi \), but not the other way round

\( \Rightarrow \) the reduction algorithm processes all possible inputs to \( \vartheta \), but does not necessarily produce all possible inputs to \( \chi \)

• as a matter of fact, in principle it would suffice that \( A \) has only two different outputs: one to which \( \chi \) answers “yes” and another one to which \( \chi \) answers “no”

• in practice, unless a polynomial time algorithm for \( \vartheta \) is known, we only know how to make an \( A \) that has infinitely many different outputs
  – to ponder: why do we know that?
4.2 The definition of NP-hardness

Definition

*Decision problem \( \varnothing \) is NP-hard, if and only if every problem in NP can be reduced in logarithmic memory to \( \varnothing \).*

NP-hardness has thus not been defined by saying “if \( \varnothing \) has a polynomial time algorithm, then every problem in NP has”

This would be a bad definition, because the connection it requires between \( \varnothing \) and NP is too weak:

- as a matter of fact, it only means that “either \( \varnothing \) does not have a polynomial time algorithm, or every problem in NP has”

\[ \Rightarrow \text{it means either } \varnothing \notin P \text{ or } \text{true} \text{ depending on whether } P = NP \]

This property, however, follows from the definition of NP-hardness
One may test the understanding of the definition of \( \text{NP} \)-hardness by answering the following questions:

- If \( P \neq \text{NP} \), is any problem in \( P \) \( \text{NP} \)-hard?

- Assume that we use polynomial time reductions (instead of logarithmic memory). Prove that if \( P = \text{NP} \), then almost every problem in \( P \) is \( \text{NP} \)-hard.

- List those problems in \( P \) that are not \( \text{NP} \)-hard, if \( P = \text{NP} \). (Still polynomial time reductions.)

- Is the problem “Does \( P = \text{NP} \) hold?” \( \text{NP} \)-hard?

One more note: we have implicitly assumed that two successive logarithmic memory reductions together make a logarithmic memory reduction

- if this did not hold, then proving a problem \( \text{NP} \)-hard via a chain of reductions would not be valid reasoning
  - via a chain of reductions \( = \) by reducing a known \( \text{NP} \)-hard problem to it, instead of reducing an arbitrary polynomial time certificate system to it
Fortunately, it is easy to see that transitivity holds

- let $A$ reduce $\vartheta$ to $\chi$ and $B$ reduce $\chi$ to $\zeta$

$\Rightarrow B(A(\cdot))$ is well-defined (the output of $A$ is appropriate input to $B$)

- $B(A(\cdot))$ reads an appropriate input to $\vartheta$ and produces an appropriate input to $\zeta$

- $\zeta(B(A(\sigma))) \iff \chi(A(\sigma)) \iff \vartheta(\sigma)$

- for some $k$, the memory usage of $B(A(\sigma))$ is

$$O(\log |\sigma|) + O(\log(|\sigma|^k)) = O(\log |\sigma|)$$

- (the running time is $O(|\sigma|^k) + O((|\sigma|^k)^h) = O(|\sigma|^{hk})$)
4.3 The definition of NP-completeness

**NP**-hardness only sets a lower bound to the complexity of a problem.

An example: the problem

"Can a given set \( \{x_1, x_2, \ldots, x_n\} \) of natural numbers be divided to two parts so that the sum of the numbers in part 1 \( = \) the sum of the numbers in part 2?"

can be easily reduced to the problem

"Does a given predicate logic formula that talks about natural numbers hold?":

\[
\exists a_1 : \exists a_2 : \ldots \exists a_n : \\
(a_1 = 0 \lor a_1 = 1) \land (a_2 = 0 \lor a_2 = 1) \land \cdots \land (a_n = 0 \lor a_n = 1) \land \\
2(a_1 x_1 + a_2 x_2 + \cdots + a_n x_n) = x_1 + x_2 + \cdots + x_n
\]

\( \Rightarrow \) the latter problem is **NP**-hard

- because of other reasons we know that it is much more complex: it has *no general algorithm whatsoever* (let alone polynomial time)

\( \Rightarrow \) it is certainly strictly more complex than the former one
On the other hand, we already have a suitable notion for an upper bound of complexity: presence in \textbf{NP}

- for instance, it is known that “does a given predicate logic formula that talks about natural numbers hold” \textit{does not} belong to \textbf{NP}

**Definition**

\[ \text{The decision problem } \varnothing \text{ is NP-complete, if and only if } \varnothing \text{ is NP-hard and } \varnothing \in \textbf{NP}. \]

We may thus roughly say

- “belongs to \textbf{NP}” \sim complexity \leq 3-CNF-SAT
- “\textbf{NP}-hard” \sim complexity \geq 3-CNF-SAT
- “\textbf{NP}-complete” \sim complexity = 3-CNF-SAT

If a problem belongs to \textbf{NP}, that is usually easy to prove

- at least one exception is known
It is easy to see that circuit satisfiability, 3-CNF-SAT, clique, hamiltonian cycle, etc., belong to $\text{NP}$

$\Rightarrow$ They are thus not only $\text{NP}$-hard, but also $\text{NP}$-complete

We can now make the notion of “hardest side of $\text{NP}$” that we have been using precise by letting it mean “$\text{NP}$-complete”
5 NP-completeness and optimization problems

5.1 The travelling salesman problem

The travelling salesman problem is the decision problem

“Does a given map contain a cycle that visits every city precisely once and whose length is at most a given number?”

- the map is represented as an undirected graph \((V, E)\)
- every edge \((u, v) \in E\) has an associated length \(\delta(u, v) \in \{0, 1, 2, \ldots\}\)
- the graph is complete, that is, \(\forall u : \forall v : u \neq v \rightarrow (u, v) \in E\)

This kind of a structure can also be represented as a pair \((V, \delta)\), where

\[
\delta : V \times V \rightarrow \{0, 1, 2, \ldots\} \quad \text{and} \quad \forall u \in V : \forall v \in V : \delta(u, u) = 0 \land \delta(u, v) = \delta(v, u)
\]
We have been talking about a problem with a slightly different formulation:

“Does a given map contain a cycle that visits every city at least once and whose length is at most a given number?”

- in it, the graph needs not be complete

⇒ the range of δ changes: $\delta : V \times V \rightarrow \{0, 1, 2, \ldots, \infty\}$

The latter is known as the travelling salesman problem with triangle inequality

The name comes from another equivalent formulation, where the ordinary travelling salesman problem is augmented with the requirement that the lengths of the edges satisfy the so-called triangle inequality, that is

$$\forall u : \forall v : \forall w : \delta(u, w) \leq \delta(u, v) + \delta(v, w)$$

We now prove that the formulations are equivalent by reducing the problems to each other
“Precisely once” with the triangle inequality $\iff$ “at least once”

- assume that the graph is complete and the triangle inequality holds
- every shortest cycle of the “precisely once” problem is, of course, also a cycle of the “at least once” problem for the same graph
- if an “at least once” cycle contains extra visits to vertices, thanks to the triangle inequality they can be replaced by single edges without making the cycle longer

$\Rightarrow$ some shortest cycle of the “at least once” problem is a cycle of the “precisely once” problem

$\Rightarrow$ every shortest cycle of the “precisely once” problem is some shortest cycle of the “at least once” problem

$\Rightarrow$ the “precisely once” problem with the triangle inequality was reduced to the “at least once” problem
“At least once” $\mapsto$ “precisely once” with the triangle inequality

- assume that vertices may be visited more than once
- the graph can be made complete and the triangle inequality hold without changing the length of the shortest cycle by adding for every connection $u_1 \to u_2 \to \ldots \to u_n$ that is as short as possible the edge $u_1 \to u_n$ (if it does not yet exist) and letting its length be $\delta(u_1, u_2) + \delta(u_2, u_3) + \cdots + \delta(u_{n-1}, u_n)$
  - succeeds in polynomial time—how?
- now extra visits to vertices in a shortest cycle can be replaced by short cuts without changing the length of the cycle

$\Rightarrow$ the “at least once” problem was reduced to the “precisely once” problem with the triangle inequality

The name “travelling salesman problem” is indeed used of the “precisely once” version, although it is difficult to justify why a real salesman must not visit the same city more than once

- if the products are so bad that he is not brave enough to show his face to the customers afterwards, let him only sell them during his last visit to the city . . .
Both the travelling salesman problem and the travelling salesman problem with the triangle inequality are \textbf{NP}-complete.

The \textit{optimization version} of the travelling salesman problem (with or without the triangle inequality) asks for a shortest cycle that meets the requirements:

- when necessary, the ordinary version is called the \textit{decision problem version} to distinguish it from the optimization version.
- it is obvious that if the optimization version of the travelling salesman problem (with or without the triangle inequality) can be solved in polynomial time, then also the decision problem version can be solved in polynomial time and \( \textbf{P} = \textbf{NP} \).
- in Section 1 we reasoned that also the opposite holds: if the decision problem version can be solved in polynomial time (that is, if \( \textbf{P} = \textbf{NP} \)), then also the optimization version can be solved in polynomial time.
5.2 On the application of NP concepts to optimization problems

In an optimization problem, the best solution is being sought for

- e.g., the shortest cycle that meets the requirements

Because either none or both of the decision and optimization versions of the travelling salesman problem can be solved in polynomial time, one may feel tempted to call also the optimization version **NP-hard**

The definition of **NP-completeness** as such does not allow that, however

- it talks about decision problems

For many optimization problems it is easy to invent **NP-hard** decision problems that are solved immediately, if a solution to the optimization problem is obtained

An example: “what is the shortest cycle” → “how long is the shortest cycle” →

- “is the length of the shortest cycle this number”
- “is there a cycle whose length is at most this number”

⇒ These optimization problems can be called **NP-hard** without causing confusion
On the other hand, application of the concept “is in $\text{NP}$” to optimization problems is problematic

- the natural corresponding decision problem does not necessarily belong to $\text{NP}$

An example: “is the length of the shortest cycle this number”

- it is easy to design a polynomial time certificate system for the claim “there is a cycle of precisely this length”

- however, the example involves also the requirement that there are no shorter cycles

- it is not easy to design a polynomial time certificate system for this claim!
  - it would prove that $\text{NP} = \text{co-NP}$
On the other hand, the problem “is there a cycle whose length is at most this number” is \( \text{NP} \)-complete

- to demonstrate that it holds it is not necessary to show that cycles of a certain kind do not exist

\( \Rightarrow \) Care is needed when formulating an \( \text{NP} \)-complete problem that corresponds to an optimization problem

\( \Rightarrow \) Calling an optimization problem \( \text{NP} \)-complete easily leads to misconceptions

- the audience does not necessarily realize the correct formulation

\( \Rightarrow \) Not recommended, not even as loose language
5.3 The travelling salesman problem (without the triangle inequality) does not have good approximate solutions

In practice, the travelling salesman does not need the best cycle

• a cycle that is at most 0.1% longer than the best is usually perfectly fine

Unfortunately, it is possible to prove that if \( P \neq NP \), then, let \( c \) be any positive real number, there is no polynomial time algorithm such that the length of the cycle it finds is always at most \( c \cdot \) the length of the shortest cycle

Let us recall the definition of the hamiltonian cycle problem:

“Does a given undirected graph have a cycle that visits every vertex precisely once?”

• the hamiltonian cycle problem is \( NP \)-complete
The proof is based on reducing the hamiltonian cycle problem to the travelling salesman problem in such a way that the existence of even an approximate solution tells whether a hamiltonian cycle exists

- let the length of each edge of the graph in question be 1
- convert the graph to a complete one by adding each missing edge and giving it the length $c|V| + 1$
- if the original graph has a hamiltonian cycle, then the modified graph has a travelling salesman cycle whose length is $|V|$
- otherwise the length of the shortest travelling salesman cycle of the modified graph is at least $|V| - 1 + (c|V| + 1) = (c + 1)|V|$

$\Rightarrow$ a solution whose length is at most $c \cdot$ the length of the shortest solution is by necessity a solution of the hamiltonian cycle

$\Rightarrow$ Polynomial time does not always suffice for finding even a solution, whose length is at most 10 000 times the length of the optimal solution (except if $P = NP$)!
5.4 The triangle inequality guarantees reasonably good approximate solutions

Adding the triangle inequality changes the situation altogether.

It invalidates the impossibility proof presented above (how?)

A minimal spanning tree of a graph is a collection of edges such that

- they are a subset of the edges of the graph
- from every vertex there is a path to every vertex along those edges
- they do not constitute a cycle (so there are $|V| - 1$ of them)
- the sum of their lengths is as small as possible

Many polynomial time algorithms are known for finding a minimal spanning tree

- also fast in practice

The shortest cycle is at least as long as the sum of the lengths of the edges of the minimal spanning tree

- if an edge is removed from the shortest cycle, a spanning tree is obtained
Systematic traversal of a tree is a cycle, where each edge is taken precisely once in each direction

⇒ Its length is at most $2 \cdot$ the length of the shortest cycle

⇒ It is an approximate solution with a factor of the length of $\leq 2$

A polynomial time algorithm that finds an even better approximate solution is known

Conclusion:

*NP*-completeness tells a lot about the cost of finding an optimal solution, but not about the cost of finding a good approximate solution.

—— End of case 3 ——