NP-completeness and complexity of problems
The meaning of “polynomial time”

To we measure the running time as a function of the input instance, we need some universal standards.

Actually, there is always a variety of ways of measuring both, the input size and the running time. But if our goals are “loose”, we can be “sloppy”.

Our goals are loose because we are interested in the following issue: can an algorithm solve our problem in time that is polynomial as the function of the input size. This statement will remain equivalent if we change the way we describe/code our problem, provided that the new encoding has a length that is “polynomially related” to the old one.

Example: we can code a graph as an adjacency matrix, $n^2$ bits, or as the list of edges, pairs of numbers in the range 0 to $n - 1$. This other encoding has to have more than $n$ bits. For this reason, we are not overly sloppy if we neglect totally how we encode the graph but take $n$ as the measure of the input size.

Similarly, when our algorithm performs an “operation”, we actually need to perform some work with every bit that is involved, be it part of an object (say, a number or a string that is part of our data or intermediate results) or an address (be it explicitly computed or a pointer). Again, we are not overly sloppy if we count an operation with unit cost as long as the number of bits is a polynomial of the input size.

Now we have an informal, but sufficiently precise understanding of class $P$, problems that can be computed in polynomial time.
The meaning of “non-deterministic polynomial time”

The meaning of non-deterministic solution to a problem is not obvious in general. However, for decision problems both the definition and intuition is clear.

A decision problem requires us to answer if the input has a certain property, and the answer is either yes or no.

A property/problem is in class \textbf{NP} if the positive answer have “short proofs”. What is a proof? Something that can be verified by a polynomial time algorithm. More formally, a problem in \textbf{NP} has the following general form:

\[ A(x) \equiv \exists y \ |y| < p(|x|) \land B(x, y) \]

where \( p \) is a polynomial and \( B \) is a polynomial time property.

Example 1: \( A(x) \) says that \( x \) is a bipartite graph with \( 2n \) nodes that has a matching of size \( n \); \( y \) is a matching with \( n \) edges, we can check easily if this is so (hence, property \( B(x, y) \) is in \textbf{P}).

Example 2: \( A(x) \) says that \( x \) is a bipartite graph with \( 2n \) nodes that has no matching of size \( n \); \( y \) is a node cover with less than \( n \) nodes, we can check easily if this is indeed a matching.

Example 3: \( x \) is a general graph with \( n \) nodes, \( k \) is a number, \( A(x) \) says that \( x \) has a node cover with \( k \) nodes. Obviously, \( A \) is in \textbf{NP}.

Example 4: \( x \) is a general graph with \( n \) nodes, \( k \) is a number, \( A(x) \) says that \( x \) has no node cover with \( k \) nodes. It is “widely believed” that \( A(x) \) DOES NOT have short proofs.
We have seen a Node Cover problem, conjectured to be hard, and Node Cover problem restricted to bipartite graphs, known to be easy. How to make proper conjectures about thousands of problems?

The idea is to make only several conjectures, like these two

\( P \neq \mathbf{NP} \), at least one problem in \( \mathbf{NP} \) has no polynomial time algorithm.

\( \mathbf{NP} \neq \mathbf{coNP} \), at least one problem in \( \mathbf{coNP} \) is not in \( \mathbf{NP} \).

The method to replace thousands of conjecture with one or two is to use polynomial time reductions

A polynomial time function \( t \) is a reduction of problem \( A \) to problem \( A' \) is \( A(x) \equiv A'(t(x)) \).

**Positive implication.** If \( A' \) has a polynomial time algorithm, then so does \( A \): composition of \( t \) and the algorithm for \( A' \) forms an algorithm for \( A \).

**Contra-positive implication.** If \( A \) has no polynomial time algorithm, neither does \( A' \).

How can it replace thousands of conjectures with just one? Because there exists \( \mathbf{NP} \)-complete problems. Problem \( A \) is \( \mathbf{NP} \)-complete if

(a) \( A \) belongs to \( \mathbf{NP} \)
(b) for every problem \( A' \) in \( \mathbf{NP} \) there exists a polynomial time reduction from \( A' \) to \( A \).

Thus if even one \( \mathbf{NP} \)-complete has a polynomial time algorithm, \( P = \mathbf{NP} \).

Proving that a problem is \( \mathbf{NP} \)-complete from the first principle is complicated and beyond the scope of the course. But it is relatively easy to prove that a problem \( A' \) is \( \mathbf{NP} \)-complete if we can assume that some problem \( A \) is \( \mathbf{NP} \)-complete: we just verify property (a) and show a reduction from \( A \) to \( A' \).
Important/classic \textbf{NP}-complete problems

SATISFIABILITY, or CNF-satisfiability
3SAT
Hamiltonian circuit
Node Cover (Vertex Cover), Independent Set, Clique
Subset Sum, Knapsack, Bin-Packing
Set Cover
Exact 3-Cover

We will look at the definitions and reductions.

We will mention easy restricted versions for some of these problems.

Some of these decision problems correspond to optimization problems and we will describe classic heuristics (e.g. for the Subset Sum, Bin Packing, Set Cover and Node Cover).

\textbf{3SAT}

This is SATISFIABILITY restricted to the case when clauses have at most 3 literals.

Suppose that a clause $c$ has $k > 3$ literals, then it is a disjunction of two shorter clauses, $c' \lor c''$ where $c'$ has $k - 2$ literals and $c''$ has 2 literals. Then we can introduce a new variable $x$ and

$$c \equiv \exists x (c' \lor x) \land (c'' \lor \neg x)$$

Thus if a SATISFIABILITY instance has a clause with $k > 3$ literals, we introduce a new variable and replace it with two clauses, one with $k - 1$ literals and the other, with 3 literals. The number of literals in “long clauses” goes down by 1.

One can see that if we start with a CNF formula that has an “excess” of $a$ literals, we obtain a formula that is satisfiable if and only if the starting one was, and which has no “long” clauses, and we increase the number of clauses and variables by $a$. 
INDEPENDENT SET

An instance is an undirected graph \((V, E)\) and integer \(k\). An independent set is a set of nodes \(A \subseteq V\) such that no edge of \(E\) connects a pair of nodes in \(A\). The question to decide is: does \(V\) contain an independent set of size \(k\)?

A reduction from SATISFIABILITY creates a graph from an instance/formula.

Suppose that the instance has \(m\) clauses and \(n\) occurrences of literals. We make a graph with \(n\) nodes and we will ask: does it have an independent set of size \(k\)?

The nodes are occurrences of literals. The edges are of two kinds:
(a) between occurrences of literals in the same clause;
(b) between occurrences of literals that cannot be true in the same time, i.e. if they have the same variable and one is of the form of \(x\) and the other, \(\neg x\).

Suppose that there exists an independent set \(A\) with \(m\) nodes. Because of (a), \(A\) has at most one node in a clause, so exactly one node in every clause. Because of (b), we can assign all literals in \(A\) to true (if an occurrence of \(x\) belongs to \(A\), we set \(x\) to true, if an occurrence of \(\neg x\) belongs to \(A\), we set \(x\) to false, and there is no conflict between these two rules). This assignment of values satisfies the formula.

Suppose that the formula is satisfied by some assignment of values. Then in every clause we can select a literal occurrence that is true. These selections form an independent set with \(m\) nodes.

CONCLUSION: the new graph has an independent set of size \(m\) if and only if the original formula has a satisfying assignment.
CLIQUE

Given a graph \((V, E)\), a **clique** is a set of nodes \(A \subseteq V\) such that every pair of nodes in \(A\) is connected with an edge of \(E\).

We can define a **complement edge set** \(\bar{E}\) that connect exactly those pair of nodes that edges of \(E\) do not connect. By the very definition,

\[
A \text{ is an independent set in } (V, E) \equiv A \text{ is a clique in } (V, \bar{E})
\]

Thus the decision problem for CLIQUE has a very simple reduction from the problem of INDEPENDENT SET.

NODE COVER

Given a graph \((V, E)\), a **node cover** (also called vertex cover) is a set of nodes \(B\) such that every edge \(e \in E\) contains a node from \(B\).

**OBSERVATION:** \(B\) is a node cover if and only if \(V - B\) is an independent set.

**COROLLARY:** for a graph \((V, E)\) there exists a node cover of size \(m\) if and only if there exists an independent set of size \(|V| - m\).

Thus the decision problem for NODE COVER has a very simple reduction from the problem of INDEPENDENT SET.

SET COVER

In this problem we have a set of elements \(U\), a collection of subsets of \(U\), \(S = \{S_1, \ldots, S_m\}\). We also have an integer \(k\).

The question is if there exists a collection of \(k\) sets, \(C = \{C_1, \ldots, C_k\} \subseteq S\) that covers \(U\), i.e.

\[
U = \bigcup_{i=1}^{k} C_i
\]

Again, a reduction from NODE COVER to SET COVER is very simple. Discuss.
SUBSET SUM

We are given a set of numbers \( S = \{ w_1, \cdots, w_n \} \) and the target number \( t \). The question is if there exists a subset \( T \subset S \) such that \( t = \sum_{i \in T} w_i \).

We can show a reduction from 3SAT. Consider an instance with \( m \) clauses and \( n \) variables. Our numbers will have \( n + 2m \) bits, and we will describe their bits.

The target has 1 on every bit position, so it equal \( 2^{n+2m} - 1 \).

We will think about our instance as a matrix of bits, with rows for numbers and columns for bit positions, so we describe a matrix with \( 2n + m \) columns and \( 2n + 2m \) rows. When we do not mention a position in that matrix, it is 0. For every variable \( x \) we have two numbers \( w_x \) and \( w_{\neg x} \).

In the column dedicated to the consistency of \( x \) both \( w_x \) and \( w_{\neg x} \) have 1.

In the pair of consecutive columns dedicated to the consistency of a clause, literals of that clause have 01. The two numbers dedicated to the consistency of that clause have both 01.

This is the full construction. If the original formula is satisfiable, we construct a subset \( T \) from the satisfying assignment. If literal \( a \) is true, \( T \) has the number \( w_a \). If \( b \) literals in a clause are true, \( T \) gets \( 3 - b \) numbers dedicated it the consistency of that clause. CHECK THAT IT IS CORRECT.

Now it remains to show the converse, namely, if we have a subset \( T \) that sums to \( t \) then we have a satisfying assignment. We start by showing that in every column/pair of columns dedicated to some consistency we do not have a “carry”. Consider the least significant such carry.

If this was a column dedicated to a variable, we had 1, 1, and the result in the column is 0, incorrect.

If this was a column dedicated to a clause, we had three to five of 01, so the result with a carry is 100 or 101, incorrect.

Without a carry, for each variable \( x \) our set \( T \) contains exactly one number from the pair \( w_x \) and \( w_{\neg x} \), so we know which literal we assign to true.

Moreover, we can see that for a clause, we must have between 1 and 3 true literals in that assignment. Thus all clauses are true.
HAMILTONIAN CIRCUIT

The input to this problem is a graph \((V, E)\). A Hamiltonian circuit is a simple cycle that contains all the nodes. The question is if the graph has a Hamiltonian circuit.

One can show that HAMILTONIAN CIRCUIT is NP-hard using a reduction from 3SAT. Given an instance of 3SAT with \(n\) variables and \(m\) clauses, we will make a graph with \(3n + 18m\) nodes, such that it has a Hamiltonian circuit if and only if the initial 3SAT instance has a satisfying assignment (of values of the variables).

The idea is that the graph will consist of \(m\) clause gadgets, each with 18 nodes and \(m\) variable gadgets, each with 3 nodes. For each literal, \(x\) and \(\neg x\) we will have a thread. The only way to make a Hamiltonian cycle is to select exactly one thread for each variable, so we have to decide which of the two literals is true, and a clause gadget has to be covered by selected threads of its literals, so we must have at least one of them selected/true.

First we design a clause gadget that can be traversed only in the form of threads: if an “entrance” of a thread is used, the “exit” is also used and they are connected by the traversal, and if we can use any combination of the threads. The picture below shows how it works. Edges with dark blue background are necessary for every Hamiltonian cycle, because of nodes of degree 2. We can assume that we use some combination of the middle edges, light blue — assume used, yellow — assume not used. From that we get implication which connection between threads are used. From that we can conclude which entrance/exit edges are used. If we assume that no middle edges are used we get a contradiction: cycles separated from the rest of nodes. In other cases, we get every possible cover of the gadget with threads.
We connect threads of literals together by exits of gadgets with appropriate entrances. Then we connect the threads of literals using simple gadgets \((a, b, c)\) where the central node \(b\) has only two neighbors, \(a\) has two exits, from threads of \(x_i\) and \(\neg x_i\), and \(b\) has to exits, to threads of \(x_{i+1}\) and \(\neg x_{i+1}\).