Time Complexity: P and NP

**DISCLAIMER:** These notes are not an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

- Time Complexity and P.
- Non-determinism and NP.
- NP-Completeness.

Last lecture we exhibited more undecidable problems by reducing the Turing machine acceptance problem to them. In this lecture, we show that the notion of hardness of computation can be more fine-grained: we can also take the number of steps required by the Turing machine into account.

## 1 Time Complexity and P

Formally, the running time of a Turing machine is a function \( f : \mathbb{N} \to \mathbb{N} \) where \( f(n) \) is the maximum number of steps that the Turing machine can take on an input of size \( n \).

This in turn leads to the definition of time complexity classes. We use \( \text{Time}(g(n)) \) to denote the set of all languages decidable by \( O(g(n)) \) time Turing machines.

Then \( P \), the set of all poly time decidable languages, is given by

\[
P = \bigcup_{k} \text{Time}(n^k).
\]

Note that this definition fits well with Turing machines because operations such as rewinding the tape give extra factors of \( n \) (actually, the length of the input, which may be as big as \( n^k \)) in the running times. Treating such overheads as in the same class allows us to ignore such details when considering algorithms.

We show a language is in \( P \) by exhibiting a Turing machine that decides it in \( O(n^k) \) time. Examples in the text book include:

1. \( \{0^k1^k \mid k \geq 0\} \in \text{Time}(n^2) \).
2. \( \text{PATH} \in P \).
3. \( \text{RELATIVEPRIME} \in P \).
4. ANY context-free language \( L \) is in \( P \) (via dynamic programming).

The less obvious ones are PRIME and COMPOSITE.
2 Non-Determinism and \( NP \)

We skipped over the notion of non-determinism when introducing Turing machines. A non-deterministic operation is simply where \( \delta(q, x) \) can give multiple outcomes. On such a machine, an input is accepted if and only if there is a sequence of valid transitions that take things to an accepting state. For running time, we only consider the shortest sequence of transitions that lead to an accepting state.

This leads to the notion of non-deterministic time: We use \( \text{NTime}(g(n)) \) to denote the set of all languages decidable by \( O(g(n)) \) time non-deterministic Turing machines. It also gives the definition of \( NP \), non-deterministic polynomial time:

\[
NP = \bigcup_k \text{NTIME}(n^k).
\]

Examples of problems in NP discussed in the textbook are CLIQUE and HAMILTON-PATH. For each such problem, the following alternate characterization of NP is useful:

**Definition 2.1.** A verifier of a language \( A \) is an algorithm \( V \) where

\[
A = \{ w \mid V \text{ accepts } <w, c> \text{ for some string } c \}.
\]

**Theorem 2.2.** \( NP \) is precisely the languages with poly-time verifiers. That is, a verifier \( V \) that runs in time \( O(n^k) \) when \( w \) has length \( n \).

Note that the poly-time behavior of \( V \) also means the length of \( c \) must also be poly \( n \). This equivalence is proven by:

1. (for showing a language with a poly-time verifier is in \( NP \)) Having a non-deterministic Turing machine that first writes down the string \( c \), and then invokes \( V \). This is akin to ‘trying all strings’.

2. (for showing any language in \( NP \) has a poly-time verifier) Letting the string \( c \) denote the non-deterministic choices of the Turing machine made during its operations, and letting \( V \) be the (deterministic) simulation of the non-deterministic Turing machine using the choices given by \( c \).

3 P vs. NP and NP-Completeness

Note that because we can simulate over all \( c \) by brute force, we also have

\[
NP \subseteq \bigcup_k \text{Time}(2^k).
\]

Somewhat surprisingly, this is the best that’s known to date in terms of the relation between \( P \) and \( NP \).

What we do know instead, is a large class of problems such that solving any one of them in \( P \) will imply \( P = NP \). \( NP \)-completeness is defined via \( NP \)-hardness, which in turn requires the notion fo poly-time reducibility.
Definition 3.1. A is poly-time reducible to B, or \( A \leq_P B \) if there is a poly-time function \( f \) such that

\[ w \in A \iff f(w) \in B. \]

Note that this in particular means that if \( B \) is in \( P \), \( A \) would be in \( P \) as well: we simply run \( B \) on \( f(w) \), whose length is at most \( poly(|w|) \) because \( f \) runs in poly-time. This leads to the notion of NP-hard:

Definition 3.2. A is NP-hard if for every \( B \in NP \), we have \( B \leq_P A \).

and furthermore, a language is NP-complete if it’s both in \( NP \) and also \( NP \)-hard.

Note that if \( A \) is \( NP \)-hard, and we have \( A \leq_P B \), then \( B \) is also \( NP \)-hard. This gives the general strategy for carving out a space of \( NP \)-hard problems: starting with one problem, and reduce it to other problems.

The problem in question is 3-SAT: it asks whether there is an assignment of variables \( x_1 \ldots x_n \) that satisfies all clauses simultaneously. A clause is the OR of three literals, and each literal is either some variable, or its negation.

The Cook-Levin theorem gives that 3-SAT is NP-hard.

Theorem 3.3 (Cook-Levin Theorem). For any problem \( A \) in \( NP \), we have \( A \leq_P 3\text{-SAT} \).

As 3-SAT is in \( NP \), it is also NP-complete.

4 Clique is NP-Complete

If time allows, I will do a proof of \( 3\text{-SAT} \leq_P \text{CLIQUE} \). The optimization version of maximum clique asks for the maximum subset of vertices such that every pair is connected by an edge. Its decision version can be formalized as \( \text{CLIQUE}(G, k) \) being true if \( G \) has a clique of size at least \( k \).

The main idea is that the structure of 3-SAT is rich enough for the literals/clauses to be interpreted as (groups) of vertices. This then allows us to convert instances of 3-SAT to instances of graph theoretic problems.

To do so, we start with the easier direction: Note that given a candidate set \( S \), we can check whether the size of \( y \) is at least \( k \), and whether all pairs of vertices in \( y \) are connected. Therefore it’s easier to have a verifier \( C((G, k), S) \).

For the harder direction we want:

\[ 3\text{-SAT} \rightarrow \text{CLIQUE}. \]

That is, we want to create a function that takes any 3-SAT instance, and output a CLIQUE instance that’s true iff the 3-SAT instance is true.

Given a 3-SAT instance with clauses \( c_1 \ldots c_m \) and variables \( x_1 \ldots x_n \), we create a graph with \( 3m \) vertices as follows:

1. For each clause \( C_r = l_{r,1} \lor l_{r,2} \lor l_{r,3} \), create one vertex for each of \( l_{r,1}, l_{r,2} \) and \( l_{r,3} \).
2. Place an edge between two vertices \( l_{r,i} \) and \( l_{s,j} \) if and only if:

- \( r \neq s \), the literals are from different clauses, and
- \( l_{r,i} \neq \neg l_{s,j} \), aka. they are consistent.

Since each the 3 vertices corresponding to each clause are not connected from each other, at most one of them can be in a clique at a time. This means the maximum clique size is at most \( m \). We now show that a clique of size \( m \) exist if and only if the 3-SAT instance is satisfiable.

1. If we have such a clique, then take the vertices involved. Since none of them conflict with each other, we can set the variables accordingly. As we have one such literal per clause, each clause would then have a satisfied literal, making this a satisfying assignment.

2. If we have an satisfying assignment, then each clause has one literal that’s satisfied. Since none of these literals conflict with each other, there are edges between them, giving the clique of size \( m \).