CS512 LECTURE NOTES - LECTURE 21

1 Complexity Classes

1.1 The class P

Definition

We say that a problem $\Pi \in P$ iff ı can be solved in polynomial time by a deterministic Turing Machine (DTM).

The problems that we have talked about so far are in P.

1.2 NP

There are some problems for which it is possible to verify a given solution in polynomial time.

Satisfiability problem (SAT)

Given a proposition, Is there an assignment of truth variables that will make the proposition true?

Notice that a proposition can be written as a function of several variables:

$$P(q_1, q_2, q_3, q_4, \ldots, q_n)$$

So the question is whether there is an assignment to each one of the variables, such as:

$$\begin{aligned} q_1 &= T \\ q_2 &= F \\ q_3 &= F \\ &\vdots \\ q_n &= T \end{aligned}$$

that makes the proposition true?

Example

In Conjunctive Normal Form (Conjunction of disjunctions):

Determine if the following formula is satisfiable:

$$(p \vee \neg q) \wedge (q \vee \neg r) \wedge (r \vee \neg p)$$

Start, for example, with $p = T$.
What do we need to show to prove that a proposition is NOT satisfiable?

1.2.1 Verification

In the case of SAT, given a boolean formula ϕ on variables $x_1, \ldots, x_n$, if you tell me that the answer is YES (it is satisfiable) how can you convince me?

You just have to show me a truth assignments of the variables that satisfies ϕ, and I can verify that the formula is in fact satisfiable.

1.2.2 NP definition

If given an instance I of a problem Π it is possible to verify a certificate C in polynomial by a deterministic Turing Machine, we say that the problem is in NP.

This definition is equivalent to saying that the problem can be solved by a non-deterministic Turing Machine in polynomial time.

The equivalence of these two definitions can be easily seen by noticing that the tree of possible solutions and verifications has a polynomial bounded depth, so it can be computed in polynomial time by a non-deterministic Turing Machine.

Definition (NP)
We say that a problem $\Pi \in NP$ if and only if there exists a deterministic Turing Machine M that accepts $V(I, C)$ in time $O(|I|^k)$ for some $k > 0$, where $V(I, C)$ verifies that C is a certificate of instance I.

Alternative Definition (NP)
We say that a problem $\Pi \in NP$ if and only if there exists a non-deterministic Turing Machine M that accepts Π in (non-deterministic) polynomial time.

The name NP comes from this second definition, meaning Non-deterministic Polynomial.
Observation: If a problem $Pi \in P$ then it can be solved by a deterministic TM in polynomial time and also can be verified by a deterministic TM in polynomial time, i.e. $\Pi \in NP$, therefore

$$P \subseteq NP$$

1.3 Polynomial Time Reducibility

We will use the same concept of reducibility that we used in the case of maximum bipartite matching, but in this case we will use a simpler version for decision problems.
Definition (polynomial time reducibility)

We say that problem Π_1 is polynomially reducible to problem Π_2 denoted by $\Pi_1 \leq_p \Pi_2 \Leftrightarrow \exists\ f : \Sigma^* \to \Sigma^*$ such that

- f is polynomially computable by a deterministic Turing Machine
- if I_1 is an instance of Π_1 then $f(I_1)$ is an instance of Π_2
- $I_1 \in \Pi_1 \Leftrightarrow I_2 \in \Pi_2$ (answer YES for instance $I_1 \Leftrightarrow$ answer YES for instance I_2)

The fact that f is polynomial-time computable implies that the size of $f(I_1)$ must be also bounded by a polynomial on I_1, since the Turing Machine computing f can only move at most one position right/left with each operation, so we have:

$$|f(I_1)| \leq g(|I_1|) \text{ where } g \text{ is a polynomial}$$

Theorem 1.

If $\Pi_1 \leq_p \Pi_2$ and $\Pi_2 \in P \Rightarrow \Pi_1 \in P$

proof.

We build a Turing machine that solves Pi_1 using the following algorithm on an instance I_1 of Pi_1:

1. $I_2 = f(I_1)$
2. Since Π_2 is in P, there is a Turing Machine M that solves I_2 in time $\leq h(|I_2|)$, where h is a polynomial
3. output the answer of M on input I_2, which we know is the answer to I_1, since $\Pi_1 \leq_p \Pi_2$

Notice that the time t this algorithm takes to solve I_1 is:

$$t \leq\leq h(|I_2|)$$

And since $I_2 = f(I_1)$, we have that

$$|I_2| = |f(I_1)| \leq g(|I_1|)$$

where g is a polynomial, therefore we get:

$$t \leq h(|I_2|) \leq h(g(|I_1|))$$

and since g and h are polynomials, and we know that the composition of two polynomials is a polynomial, then there exists a deterministic TM that accepts Pi_1 in polynomial time

$$\therefore \quad Pi_1 \in P$$

1.4 NP-completeness

There is a class of problems called NP-hard such that if a problem $\Pi \in$NP-hard then all the problems in NP can be polynomially reduced to Π.

Definition

$Pi \in$NP-hard $\Leftrightarrow \forall\ \Pi_x \in NP\ \Pi_x \leq_p \Pi$

The idea is that NP-hard contains those problems that are "so hard" that if we can solve one of those problems, we could use it to solve any problem in NP. But these problems (NP-hard) might be even harder than those in NP, i.e. it is possible that there are NP-hard problems that are not even in NP.

We are interested in those problems that are the hardest problems in NP, i.e those that are NP-hard, but are also in NP.

Definition (NP-complete)

A problem $\Pi \in$NP-complete $\Leftrightarrow$

1. $\Pi \in NP$
2. $\Pi \in$NP-hard

What would happen if **one** of these (hard) problems could be solved in polynomial time by a deterministic Turing Machine?

Well what would happen is that then $NP \subseteq P$

Theorem

If $\Pi \in$NP-complete and $\Pi \in P \Rightarrow NP \subseteq P$

Proof

Assume that $\Pi \in$NP-complete and $\Pi \in P$.

Let $\Pi_x \in NP$, since $\Pi \in$NP-hard then $\Pi_x \leq_p \Pi$

From **theorem 1** we have that $\Pi \in P$

□

2 NP-complete problems

The question now is: How do we find a problem in NP-complete?

It seems really difficult to show that **every** problem in NP can be reduced to it!

What we will do is see what all problems in NP have in common: they all have a non-deterministic Turing Machine that accepts instances in the language.

So we will look at the computation of a non-deterministic Turing Machine on instance I and provide all the conditions that the computation must satisfy. We will then see that all of these conditions must represent an instance of the SAT (satisfiability) problem, and we would have our first NP-completeness reduction.
Theorem SAT $\in$ NP-complete

proof

1. SAT$\in$ NP. It ie easy to see that given a boolean formula $\phi(x_1, \ldots, x_n)$ and a truth assignment to the variables, a deterministic Turing Machine can verify that the given truth assignment satisfies ϕ in polynomial time.

2. SAT$\in$NP-hard (Cook's Theorem)

   A polynomial-time computation of a Turing Machine M can be represented as a matrix where each row corresponds to an instantaneous configuration:

Δ	a	q_3	b	Δ	Δ

   We will call this matrix the *Computation Matrix* and will be represented by T_{ij}.

   Assume that the input to M is $x \in \Sigma$. Since the computation takes polynomial time $|x|^k$ then the maximum possible size of the computation matrix is $|x|^k \times |x|^k$.

   The symbols that can be used on the computation matrix are $C = Q \cup \Gamma$

   In order to represent the computation using boolean variables, we will use $X_{i,j,s}$ that is true if and only if the symbol on the computation matrix in position i, j is $s \in C$.

   $$X_{i,j,s} = \begin{cases} \text{True} & \text{if} \quad T_{i,j} = s \\ \text{False} & \text{otherwise} \end{cases}$$

   For the computation matrix to represent an accepting computation the following conditions must be satisfied by the variables $X_{i,j,s}$:

(a) Each entry of the computation matrix has exactly one element from C. This condition can be separated in two conditions:

i. Each entry has at least one symbol from C, i.e. entry i, j satisfies:

$$\vee_{s \in C} X_{i,j,s}$$

Since all the matrix elements must satisfy the condition we can use the *and* operator over all matrix elements:

$$\wedge_{i,j} (\vee_{s \in C} X_{i,j,s})$$

ii. Each entry has at most one symbol from C, i.e. in every pair symbols for the same entry at most one can be true. This can be described as saying that for every pair of variables for the same entry representing different symbols, the negation of both cannot be true.

$$\neg X_{i,j,s} \vee \neg X_{i,j,t} \;\; \forall i, j; \; t \neq s$$

Again we can use the *and* operator over all elements.

(b) **Start configuration** The start configuration should be:

q_0	x_1	x_2	$\ldots$	x_n	Δ	$\ldots$	Δ

Which can be written as:

$X_{1,1,q_0} \vee X_{1,2,x_1} \vee \ldots X_{1,3,x_2} \vee \ldots \vee X_{1,n+1,x_n} \vee X_{1,n+2,\Delta} \vee X_{1,|x|^k,\Delta}$

(c) **Transition function** In order to make sure that the computation is valid, we must model the operation of the transition function. The easiest way to do this is to write each possible 2×3 section of matrix that would represent a valid computation using the transition function, and use a boolean formula to represent it

For example, if $\delta(q_3, a) = (q_6, b, R)$ then the following submatrix would represent a valid computation:

c	q_3	a
c	b	q_6

This situation can be represented as

$X_{i+0,j+0,c} \wedge X_{i+0,j+1,q_3} \wedge X_{i+0,j+2,a} X_{i+1,j+0,c} \wedge X_{i+1,j+1,b} \wedge X_{i+1,j+2,q_6}$

and this should be true for all i, j within the boundaries of the computation matrix.

Notice that since the number of elements in C is constant with respect to the length of the input, and since the submatrix that is being used has only 6 entries, the total number of combinations is constant with respect to the length of the input so in total we would have $A|x|^{2k}$ conditions, which is polynomial because A is a constant.

(d) **Accepting configuration:** The last row must be an accepting configuration:

$$\vee_{j=1}^{|x|^k} X_{|x|^k,j,q_f}$$

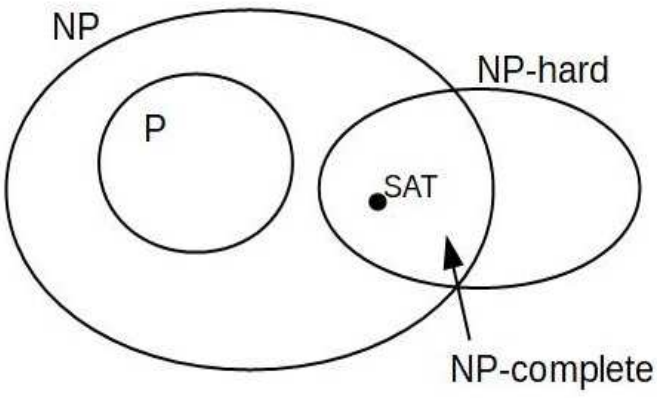
Taking the conjunction of all these conditions we have a boolean formula $\Pi(X_{i,j,s}\ldots)$

Therefore, given a TM M that accepts a problem Π, an instance of the problem x, and the transformation described above it is not hard to show that

There is a computation of M that accepts instance x if and only if the bolean formula $\phi(X_{i,j,s}\ldots)$ is satisfiable.

□

This proves that there are problems that are NP-complete. At least SAT is such a problem. So a Venn diagram of the complexity classes that we have seen so far looks like this:



3 NP Completeness Reductions

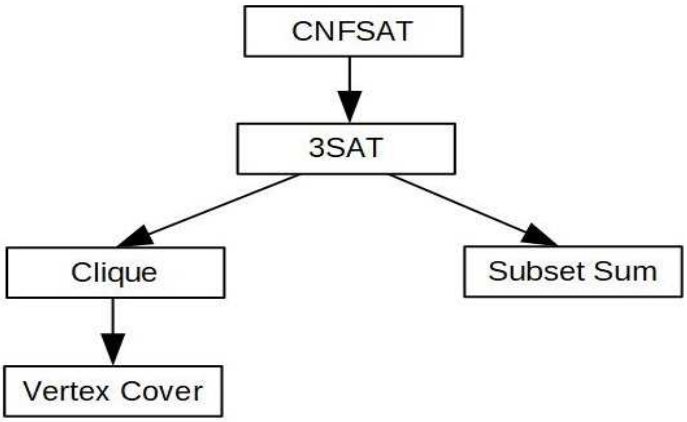
Now that we have our first NP-complete problem, the question is How do we find more? It was hard to transform the computation of a Turing Machine into a boolean formula. Fortunately we do not have to do the same for every other problem that we want to show that is NP-complete.

Since we know that $\forall \Pi_x \in NP$, $\Pi_x \leq_p SAT$, if we can show that $SAT \leq_p \Pi$ then Π is also NP-hard, since the composition of polynomials is also a polynomial.

So, the strategy to prove that a problem Π is NP-hard will be:

Find a polynomial time reduction from a known NP-hard problem Π_1 to Π, i.e. $\Pi_1 \leq_p \Pi_2$

We will provide several examples of these reductions:



where each arrow represents $\leq_p$.