CS 341 Algorithms. Spring 2025. University of Waterloo

Lecture 18: NP-completeness

We study the class of NP problems and show that 3-SAT is NP-Complete.

The Class NP

As we discussed last time, we could do reductions between different problems and slowly build up a huge map showing the relations of all the problems.

But it would be exhausting to do so many reductions, or even just to look up the relations.

It would be nice if we could identify the "hardest" problem X of a large class.

Then, when we have a new problem Y, we just need to prove that $X \leq_p Y$ and then Y would also be at least as hard as all the problems in the large class.

This sounds very good, but how can we show that a problem is the hardest among a large class?

For this, we need a general and more abstract definition that captures a very large class of problems.

Short Proofs

A general feature of all the problems that we have seen is that although it may be difficult to determine whether an instance is a YES-instance or not, it is easy to verify that it is a YES-instance in the sense that there is a short proof for this fact.

For example, given a graph, we don't know how to determine if it has a Hamiltonian cycle efficiently, but we can easily verify that it is a YES-instance if someone tells us a Hamiltonian cycle and we just need to verify that all the edges in the cycle are really present in the graph to confirm.

As another example, given a 3-SAT formula, it is easy to confirm that it is a YES-instance if someone tells us a satisfying assignment.

In short, although it may be computationally difficult to find a solution for these problems, it is easy to verify that a solution is correct.

Informally, NP is the class of problems for which there is a "short proof" of its YES-instances that can be checked efficiently.

Definition (NP) For a problem X, each instance of X is represented by a binary string s. A problem X is in NP if there is a polynomial time verification algorithm B_X such that the input s is a YES-instance if and only if there is a proof t which is a binary string of length $\text{poly}(|s|)$ so that $B_X(s,t)$ returns YES.

The key points are: B_X is a polynomial time algorithm and t is a short proof of length poly($|S|$).
In most problems, t is simply a solution and B_X is an efficient algorithm to check if t is indeed a solution.
This definition is a bit abstract, so let's see some concrete examples.

Example 1 (vertex cover): The problem X is whether a graph has a vertex cover of size at most k.
The input string s is a binary string representation of an input graph $G=(V,E)$
The proof string t is a binary string representation of a subset $S \subseteq V$ of at most k vertices.
The verification algorithm B_X will take s and t as input, go through all the edges in G and return YES if and only if S indeed covers all the edges in G.
Clearly, the proof is of length poly($|V|$) (short) and the verification algorithm runs in polytime (efficient).
Finally, the verification algorithm will only accept YES-instances, i.e. for a graph with no vertex cover of size at most k, there exists no proof that will make the verification algorithm to return YES.
So, the decision version of the vertex cover problem is in the class NP.

Example 2 (3-SAT): Given a 3-SAT formula, the verification algorithm expects the proof to be a satisfying assignment and will return YES if it indeed satisfies all the clauses. Clearly, the proof is short (polysize), the verification is efficient (polytime), and it only accepts YES instances.

Exercises Show that Clique, IS, HC, HP, Subset-Sum are all in the class NP.

It should be apparent that the class NP captures all the problems considered in this course, since they all have short solutions that are easy to verify.

Remark 1 (non-examples)
Suppose the problem is to determine whether a graph is non-Hamiltonian, i.e. no Hamiltonian cycles.
Then no one knows whether this problem is in NP, and the common belief is that it is not in NP.
In other words, we don't know how to efficiently check that a graph is a NO-instance of HC.
In fact, we don't know how to do much better than enumerating all potential Hamiltonian cycles and check.

Remark 2 (co-NP)
Contrast the above remark with the bipartite matching problem, for which we know how to efficiently check whether a graph is a NO-instance (i.e. no matching of size $>k$), by checking a vertex cover of size $\leq k$.
A problem with short and easy-to-check proofs for both YES and NO instances is in NP $\cap$ co-NP.
For example, a min-max theorem such as König's theorem would show that a problem is in NP $\cap$ co-NP.

But the common belief is that most problems in NP are not in NP ∩ co-NP (so, e.g. no mix-max theorem).

Remark 3 ($P \subseteq NP$)

Clearly, every polynomial time solvable decision problem is in NP: The verification algorithm is simply the algorithm to determine whether an input is a YES-instance, without the need of a proof string.

Let P denote the class of decision problem solvable in polynomial time. Then, $P \subseteq NP$.

Remark 4 (non-deterministic polynomial time)

NP is the class of problems solvable by a non-deterministic polytime algorithm, hence the name NP.

A non-deterministic machine, roughly speaking, has the power to correctly guess a solution or an accepting path.

So, as long as there is a short solution, a non-deterministic machine will magically find it.

Remark 5 ($P = NP$?)

It is the most important open problem in theoretical computer science to answer whether $P = NP$ or $P \subset NP$.

It is now one of the seven open problems in mathematics posted by Clay Math Institute, with $1000000 award.

The common belief is that $P \neq NP$, and the intuition is that an efficient algorithm to verify a solution should not automatically implies an efficient algorithm to find a solution.

For examples, someone who can recognize good music doesn't imply that they are a good composer, and someone who can check a mathematical proof doesn't imply they could come up with the proof.

NP-completeness

Informally, we say a problem is NP-Complete if it is a hardest problem in NP.

Definition (NP-completeness) A problem $X \in NP$ is NP-Complete if $Y \leq_p X$ for all $Y \in NP$.

From the definition, we can formally show that an NP-Complete problem is a hardest problem in NP.

Proposition $P = NP$ if and only if an NP-Complete problem can be solved in polynomial time.

Cook and independently Levin formulated the class of NP problems and proved the following important theorem.

Theorem (Cook-Levin) 3-SAT is NP-Complete.

Since polynomial time reductions are transitive, if we can prove that $3\text{-SAT} \leq_p X \in NP$, then X is also NP-Complete.

For example, we have proved in L17 that $3\text{-SAT} \leq_p IS$, and so the independent set problem

is also a hardest problem in NP, a good reason that we couldn't solve it in polynomial time.

Then, it also follows from the results in L17 that VC and Clique are NP-Complete.

Proving NP-Completeness

To prove that a problem X is NP-Complete, first we show that $X \in NP$, then we find an an NP-Complete problem Y and prove that $Y \leq_p X$.

The reduction $Y \leq_p X$ is by a polynomial time algorithm, so interestingly we prove the hardness of a problem X by providing an efficient algorithm to use X to solve a hard problem Y.

But note that this is quite different from finding an efficient algorithm to solve X.

When we try to find an algorithm to solve X, we use the algorithms that we know (eg. bipartite matching) and try to apply them to design an efficient algorithm for solving X.

When we try to prove X is NP-Complete, we assume that we know how to solve X, and we need to search for an NP-Complete problem Y and use X to design an efficient algorithm for Y.

This is a very different experience, and often we don't know what Y to start with.

We will do a few NP-Completeness proofs in the next lectures, many look quite magical, e.g. $3\text{-SAT} \leq_p IS$.

It will take a lot of practices to acquire the skill to prove NP-completeness result.

Cook-Levin Theorem

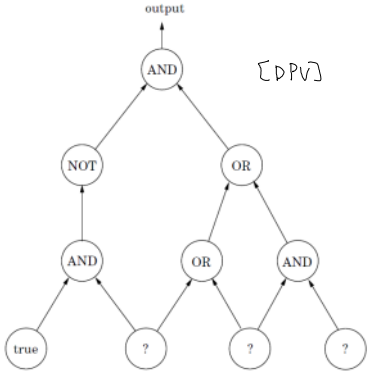
We would like to prove that 3-SAT is NP-Complete. The original proof directly does it.

Following the reference books, it would be easier to introduce some intermediate problems to do so.

Circuit-SAT

Input: a circuit with AND/OR/NOT gates, some known input gates, and some unknown input gates.

Output: is there a truth assignment to the unknown input gates so that the output is True?



[DPV]

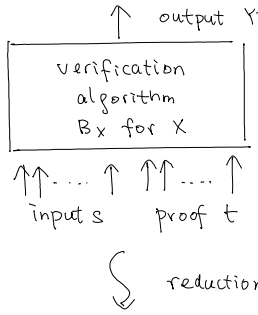
We can assume the input circuit is a directed acyclic graph, and each AND/OR gate has only two incoming edges.

Theorem Circuit-SAT is NP-Complete

Proof (sketch) To prove such a general statement, we need to start from the abstract definition of NP.

Our goal is to prove that for any $X \in NP$, $X \leq_p$ Circuit-SAT.

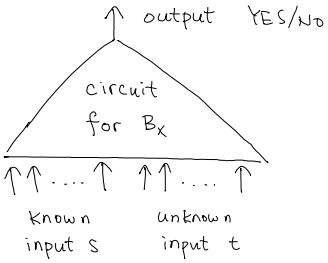
Since $X \in NP$, by definition, there is a polynomial time verification algorithm B_X such that if an instance s is a YES-instance, there exists a short proof t such that $B_X(s,t)$ returns YES; otherwise $B_X(s,t)$ returns NO for all t.



Let's think about what is an algorithm.

It can be written as a program and executed on a machine.

If the verification algorithm runs in poly(|s|) time, then there is a machine that executes it in poly(|s|) time, and so there is a circuit with only poly(|s|) size implementing the algorithm as it only requires poly(|s|) operations.



We are being sketchy about the reduction from an algorithm to a circuit. To do it precisely, we need a formal definition of a nondeterministic Turing machine and the details are quite tedious.

The main conceptual idea here is that a circuit is as general as an algorithm.

The reduction can be carried out in polynomial time (we can think of it as some kind of a compiler procedure that translates an algorithm into a circuit).

The original proofs of Cook and Levin directly transforms a non-deterministic Turing machine for the verification algorithm into a CNF formula (I may learn from CS 360 or CS 365).

Once we accept this reduction can be done in polynomial time, the rest is straightforward.

If s is a YES-instance, then there is a proof string t that will make the algorithm return YES, and hence make the circuit output True.

If s is a NO-instance, then no input t of the circuit will make it output True.

So, s is a YES-instance to problem X iff there is a satisfying assignment to the Circuit-SAT problem.

Therefore, $X \leq_p$ Circuit-SAT for any $X \in NP$. □

From circuit to formula

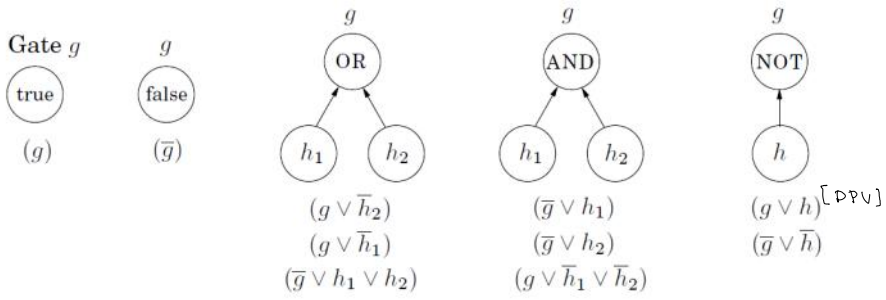
Now, we just need to show that a CNF formula has the same expressive power as a circuit.

Proposition Circuit-SAT $\leq_p$ 3-SAT.

Proof Given a circuit of n gates, we will construct a formula with $O(n)$ number of variables so that the circuit is satisfiable if and only if the 3-CNF formula is satisfiable.

For each gate in the circuit, we create one variable in the formula.

Then, we add clauses to the formula so as to simulate the function of the circuit as follows:



[DPV]

1. If there is a True gate g, we add a one literal clause (g) so that to satisfy the formula we must set g to be True, faithfully simulating the circuit.
2. Similarly, for a known False gate g, we add a clause $(\overline{g})$ to force g to be set to False.
3. For a NOT gate g, we want the value of g and its input h to be different, and we can enforce this by adding two clauses $\neg(g \wedge h) \wedge \neg(\overline{g} \wedge \overline{h}) = (\overline{g} \vee \overline{h}) \wedge (g \vee h)$
4. For an AND gate g, we want if the two inputs h_1 or h_2 is false, then g is false, so we add $(\overline{g} \vee h_1) \wedge (\overline{g} \vee h_2)$, and if h_1 and h_2 are true then g is true, so we add $(g \vee \overline{h}_1 \vee \overline{h}_2)$.
5. Similarly, for an OR gate g, we add $(g \vee \overline{h}_1) \wedge (g \vee \overline{h}_2)$ so that if h_1 or h_2 is true then g must be set to True to satisfy the formula, and we add $(\overline{g} \vee h_1 \vee h_2)$ so that if h_1 and h_2 are both False then g must also be set to False too.

Finally, to ensure that the output of the circuit is True, we add a variable o and add a clause (o) to ensure that o will be set to True.

It should be clear that this transformation from a circuit to a 3-CNF formula can be done in polynomial time, as we can just do "local replacement" for each gate by clauses.

Also, it should be clear that the circuit is satisfiable if and only if the formula is satisfiable, as there is a one-to-one correspondence between the variables and the gates, and the clauses enforce faithful simulation of the gates of the circuit. □

With the Cook-Levin theorem, we have a firm foundation to prove that a problem is NP-hard. We will grow our list of NP-complete problems in the next lecture.

References: [KT 8.3, 8.4], [DPV 8.3]