CS 341 - Algorithms, Spring 2017. University of Waterloo

Lecture 18: NP-completeness

We study the general definition of the class of NP problems and show that 3SAT is NP-complete.

NP

As we discussed at the end of last lecture, we could do reductions between different problems and slowly build up the huge map of the relations of all the problems.

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

Would it 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 show 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 large class of problems.

Short proofs

One 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, but we can easily verify that it is an YES instance if someone tells us a Hamiltonian cycle, and we just need to confirm that all the edges in the cycle are really present in the graph.

As another example, given a 3SAT formula, it is easy to check 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 it is easy to verify a solution is correct.

Definition (NP) For a problem X, we represent an instance of X as a binary string s.

A problem X is in NP if there is a polynomial time verification algorithm B 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 $B(s,t)$ returns YES.

Note that the key points are: B is a polynomial time algorithm, and t is a short proof of length $\text{poly}(|s|)$.

This definition is a bit abstract, so let's see some concrete examples.

Example 1 (vertex cover): Given the input graph $G=(V,E)$ of n vertices, the verification algorithm expects the proof to be a solution, i.e. a subset S of $\leq k$ vertices, and then the algorithm will go through all the edges and check if S indeeds cover all the edges. Clearly, the proof is of length $\text{poly}(n)$ (short) and the verification algorithm runs in polynomial time. Finally, the verification algorithm will only accept YES-instances; for a graph with no vertex cover of at most k vertices, there exists no proof that will make it accept.

Example 2 (3SAT) Given a 3SAT formula, the verification algorithm expects the proof to be a satisfying assignment and will check whether it satisfies all the clauses. Clearly, the proof is short (polysize), the verification is quick (polytime), and it only accepts YES instances.

Exercises Check that Clique, IS, HC, HP, SUBSET-SUM are all in NP.

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

Remark 1 Just before you think that all problems are in NP. Let us see some non-examples. Suppose the problem is to ask you to determine whether a graph is non-Hamiltonian. Then no one knows of a polytime verification algorithm to check that a graph is non-Hamiltonian (i.e. no Hamiltonian cycles). We don't know how to certify that an instance is a NO-instance of HC efficiently. Contrast this with maximum bipartite matching, where we have a short proof that a graph has no matching $>k$, by showing a vertex cover of size k. A problem with short proofs for both YES and NO instances belongs to NP$\cap$co-NP. The common belief is that not every problem in NP belongs to NP$\cap$co-NP.

Remark 2 Clearly, every polynomial time solvable decision problem is in NP. The verification algorithm is simply the algorithm to solve the problem, and there is no need for a proof. Let P denote the class of decision problems solvable in polynomial time. Then $P \subseteq NP$.

Remark 3 The name NP comes from nondeterministic polynomial time. A nondeterministic machine, roughly speaking, has the power to correctly guess a solution, or an accepting path. So, as long as there is a short solution, the machine will magically find it.

Remark 4 It is the most important problem in theoretical computer science (if not in all computer science) whether $P = NP$ or $P \subset NP$. It is now one of the seven open problems in mathematics posted by Clay Mathematics Institute, with a one-million dollar award for each solution. The common belief is that $P \neq NP$. An intuitive heuristic argument is that if we can efficiently verify a solution, it shouldn't automatically imply that we can also efficiently find a solution. For example, if someone can verify good music, it doesn't imply that he/she is also a good composer. As another example closer to us, if someone knows how to verify a mathematical proof, it doesn't imply that he/she can come up with the proof.

NP-completeness

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

Definition (NP-completeness) A problem $X \in NP$ is NP complete if $Y \leq_p X \ \forall Y \in NP$.

From the definition, we can formally say that an NP-complete problem is the hardest problem in NP.

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

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

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

Since reductions are transitive, if we can prove that $3SAT \leq_p X$, then X is also NP-complete. For example, we have proved in last lecture that $3SAT \leq_p IS$, and so the independent set

problem is also a hardest problem in NP, a good reason that we couldn't solve it in polytime. Then, it also follows that VC and clique are NP-complete.

To prove that a problem X is NP-complete, we just need to find an NP-complete problem Y, and prove that $Y \leq_p X$.

The reduction is a polynomial time algorithm.

So, in a sense, we prove a problem is hard by providing an algorithm.

But notice that it is quite different than finding an algorithm to solve X.

When we try to find an algorithm to solve X, we use the algorithms that we know (say bipartite matching) and try to apply them to solve X.

When we try to prove X is NP-complete, we assume we know how to solve X, and we need to search for an NP-complete problem Y and use X to Y.

It is a quite 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.

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

Cook-Levin theorem

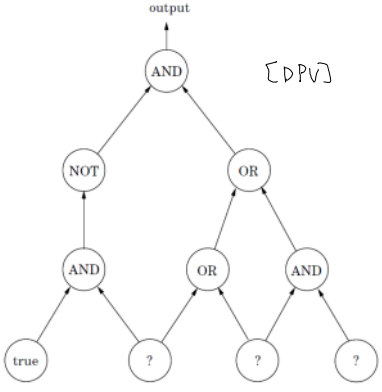
We would like to prove that 3SAT is NP-complete. The original proof directly does it.

Following the textbooks, 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: whether there is a truth assignment of the unknown input gates so that the output is TRUE.



[DPV]

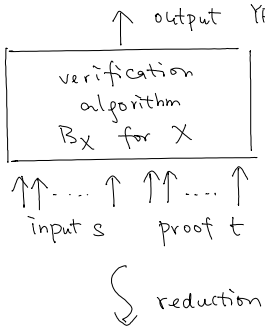
We can assume that the input circuit is a directed acyclic graph, and each AND/OR gate has only two incoming edges.

Claim Circuit-SAT is NP-complete.

proof (sketch) To prove such a general statement, we need to start with the general and abstract definition of the class NP.

We want to show that for any $X \in NP$, $X \leq_p$ Circuit-SAT.

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 $B_X(s,t)$ accepts; otherwise, $B_X(s,t)$ rejects for all t.



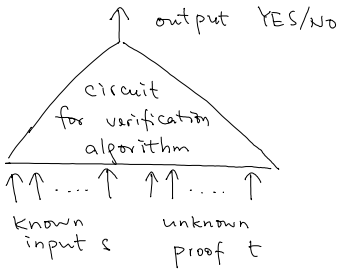
Let's think about what is an algorithm.

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

A machine can be represented as a circuit.

If the verification algorithms 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.

reduction



We are being sketchy about this 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 (think of it as some kind of a compiling procedure, to translate an algorithm into a circuit).

You can probably learn about the formal proof in CS 360 or CS 365.

Once we accept this reduction, the remaining is straightforward.

If s is an YES-instance, then there is a proof that will make the algorithm and hence the circuit to return YES.

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

So, s is a YES-instance if and only if Circuit-SAT says YES, so $X \leq_p$ circuit-SAT for any $X \in NP$. □

Once we have translated the abstract notion of an algorithm to a concrete object of a circuit, it is much easier to reduce it to other (combinatorial) problems.

From circuit to formula

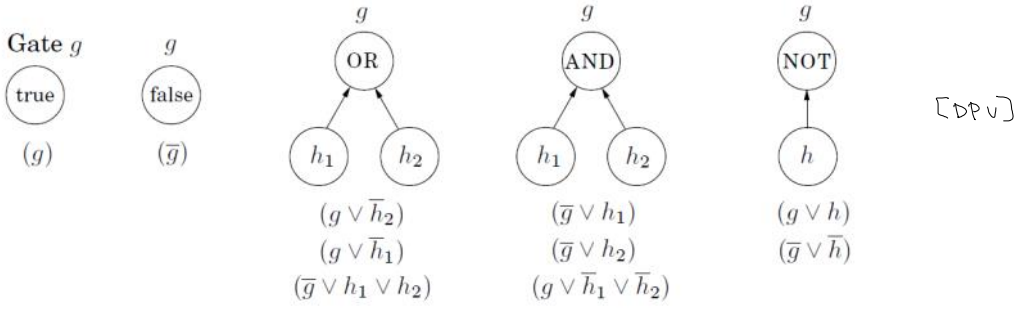
Now, we just need to show that a formula has the same expressive power as a circuit.

Claim Circuit-SAT $\leq_p$ 3SAT.

proof Given a circuit of n gates, we will construct a formula of $O(n)$ variables so that the circuit is satisfiable if and only if the formula is satisfiable.

We create one variable in the formula for each gate in the circuit.

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



[DPV]

If there is a true gate g, we add a one variable clause (g) so that to satisfy the formula we must set g to be True, faithfully simulating the circuit.

Similarly, for a (known) false gate g, we add a clause $(\overline{g})$ to force it to be false.

For a NOT gate, we want the value of g and h to be different, and we can enforce it by adding two clauses $(g \vee h) \wedge (\overline{g} \vee \overline{h})$.

For an AND gate, we want if 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 we want if h_1 and h_2 are true, then g is true, so we add $(g \vee \overline{h}_1 \vee \overline{h}_2)$.

Similarly, for an OR gate, we add $(g \vee \overline{h}_1) \wedge (g \vee \overline{h}_2)$ so that if one of h_1, h_2 is true, then g must be true to satisfy the formula; otherwise when both false, we add $(\overline{g} \vee h_1 \vee h_2)$ to enforce that g to be false.

Finally, to make sure the output is T, we add a variable o and add a clause (o) to make sure that it will be evaluated to T.

By this construction, it is clear that the transformation is polynomial time, as we can just do this "local replacement" for each gate by clauses.

Also, by the construction, it should be clear that the circuit is satisfiable if and only if the formula is satisfiable, as there is a one-to-one correspondance between the variables and the gates and the above simulations work. □

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

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