NP, NP-hardness, and NP-completeness

CMSC 27230: Honors Theory of Algorithms

February 20, 2023

Corresponding section(s) of Kleinberg-Tardos: 8.1,8.3

1 NP

NP, which stands for non-deterministic polynomial time (NOT not polynomial time) is an extremely important class of problems in complexity theory. We will first give a more intuitive description of NP and then give more precise definitions of NP.

Definition 1.1. *We say a YES/NO problem is in P if there is a polynomial time algorithm to solve the problem.*

Definition 1.2 (Intuitive description of NP)**.** *We say that a YES/NO problem is in NP if when someone gives you a solution to the problem showing that the answer is yes, it is easy to verify this solution.*

To illustrate what this means, we consider the 3-coloring problem.

Problem 1.3 (3-coloring)**.** *In the 3-coloring problem, we are given an undirected graph G and we are asked whether it is possible to color each vertex of G with one of three different colors so that no two adjacent vertices have the same color.*

Example 1.4. *If G is a triangle (i.e. $V(G) = \{v_1, v_2, v_3\}$ and $E(G) = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_2, v_3\}\}$) then we can color v_1 red, v_2 white, and v_3 blue.*

Example 1.5. *If G is a square (i.e. $V(G) = \{v_1, v_2, v_3, v_4\}$ and $E(G) = \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_1, v_4\}\}$) then we can color v_1 red, v_2 white, v_3 red, and v_4 white.*

Example 1.6. *If $G = K_4$ (i.e. $V(G) = \{v_1, v_2, v_3, v_4\}$ and $E(G) = \{\{v_i, v_j\} : i < j \in [4]\}$) then we cannot 3-color G because each vertex would have to have its own color but there are only 3 colors.*

We now make the following observation. If someone gives you a huge graph and asks you if it is 3-colorable, this might be extremely hard to solve. In fact, we currently do not know of any methods which are guaranteed to be much faster than a brtue force search. However, if someone gives you a huge graph, claims that it is 3-colorable, and gives you the 3-coloring, this is relatively easy to check. You just need to go through all of the edges $e = \{u, v\} \in E(G)$ and confirm that u and v are assigned different colors.

We now describe how to define NP more precisely.

Definition 1.7 (Non-deterministic polynomial time algorithms)**.** *A non-deterministic polynomial time algorithm for a YES/NO problem P is an algorithm A (which may use random coin flips) such that*

1. *Whenever the answer to P is NO, A always returns NO (A never gives a false positive).*
2. *Whenever the answer to P is YES, $Pr[A \text{ returns YES}] > 0$ (whenever the answer is YES, A has some chance, which may be extremely small, of returning YES).*
3. *$\exists B, c > 0$ such that A always terminates in time Bn^c (A always takes polynomial time).*

Definition 1.8 (More precise definition of NP)**.** *We say that a YES/NO problem P is in NP if there is a non-deterministic polynomial time algorithm for P.*

Example 1.9. *A non-deterministic polynomial time algorithm for 3-coloring is as follows:*

1. *Guess a color for each vertex.*
2. *Check if the coloring we guessed is a 3-coloring. If it is a 3-coloring, we output YES. If it is not a 3-coloring, then we output NO.*

To see that this is a non-deterministic polynomial time algorithm, note that if the graph G has no 3-coloring, we will always output NO regardless of what we guess and if G has a 3-coloring, then there is an extremely small but nonzero chance that we will output YES.

Remark 1.10. *Note that we only make a guess and check it ONCE. Here we are not trying to accurately answer the problem. Instead, we just need a nonzero chance of outputting YES when the answer is YES and one guess is sufficient for this.*

We now show why this definition is equivalent to the description of NP as the class of problems where it is easy to verify a solution. To do this, we first need to make this description more precise.

Definition 1.11 (Polynomial time verifiers)**.** *A polynomial time verifier for a YES/NO problem is an algorithm V such that*

1. *The input to V is the input to the problem $x \in \{0,1\}^n$ and a potential solution $s \in \{0,1\}^m$ where m is polynomial in n.*
2. *If the answer to the problem is YES, there exists a solution s such that $V(x,s) = 1$. Conversely, if the answer to the problem is NO then for all $s \in \{0,1\}^m$, $V(x,s) = 0$.*
3. *V is a polynomial time algorithm.*

Definition 1.12 (More precise definition of NP with verifiers)**.** *We say that a YES/NO problem P is in NP if there is a polynomial time verifier for P.*

Lemma 1.13. *These two definitions of NP are equivalent. In other words, there is a non-deterministic time algorithm A for a problem P if and only if there is a polynomial time verifier for P.*

Proof. If there is a polynomial time verifier V for P, then we have the following non-deterministic polynomial time algorithm for P:

1. Guess a solution $s \in \{0,1\}^m$.

2. Output YES if $V(x,s) = 1$ and output NO if $V(x,s) = 0$.

To see that this is non-deterministic polynomial time, observe that if the answer to the problem is YES, there exists a solution $s \in \{0,1\}^m$ such that $V(x,s) = 1$ so there is a nonzero probability that A will guess this s and output YES. Conversely, if the answer to the problem is NO, then $V(x,s) = 0$ regardless of s so A will always output NO.

For the other direction, given a non-deterministic polynomial time algorithm A for P, we can construct a polynomial time verifier as follows. We take $V(x,r)$ to be $A(x)$ where r gives the results of the random coin flips for A. To see that this is a polynomial time verifier, observe that if the answer to the problem is NO then A always outputs NO so $V(x,r) = 0$ regardless of r. On the other hand, if the answer to the problem is YES then there is a nonzero chance that A outputs YES so there must be some r such that $V(x,r) = 1$ □

2 Reductions

Reductions, where we transform a problem A into another problem B, are an extremely important idea in complexity theory and in mathematics. Reductions have several uses. First, if we can reduce problem A to problem B and we know how to solve problem B, then this gives us a way to solve problem A. More subtly, if we know or have good reason to believe that problem A is hard and we can reduce problem A to problem B, this shows that problem B must be hard as well. Finally, if problem A and problem B are reducible to each other, this means that problem A and problem B are essentially equivalent.

Definition 2.1. *We say that problem A is poly-time reducible to problem B if there is a polynomial time algorithm R which takes an instance x of problem A of size n and returns an instance y of problem B of size $poly(n)$ such that $A(x) = B(y)$ (the problem has the same answer after the reduction)*

Proposition 2.2. *If problem A is poly-time reducible to problem B and problem B is poly-time reducible to problem C then problem A is poly-time reducible to problem C.*

A simple example of two problems which are poly-time reducible to each other are independent set and vertex cover (as we'll see later, both of these problems are NP-complete)

Problem 2.3 (Independent Set)**.** *In the independent set problem is as follows. Given a graph G, is there an independent set of size k in G? In other words, is there a set of vertices $I \subseteq V(G)$ such that $|I| = k$ and no two vertices in I are adjacent to each other?*

Problem 2.4 (Vertex Cover)**.** *The vertex cover problem is as follows. Given a graph G, is there a vertex cover of size k in G? In other words, is there a set of vertices $V \subseteq V(G)$ such that for every edge $(u,v) \in E(G)$, either $u \in V$ or $v \in V$?*

Theorem 2.5. *The independent set and vertex cover problems are poly-time reducible to each other.*

Proof. We make the following observation.

Lemma 2.6. *There is an independent set of size k in G if and only if there is a vertex cover of size $n-k$ in G.*

Proof. If I is an independent set of G of size k then we take $V = V(G) \setminus I$. Since I is an independent set, for any edge $(u,v) \in E(G)$, either $u \notin I$ or $v \notin I$ so either $u \in V$ or $v \in V$.

Conversely, if V is a vertex cover of G then we take $I = V(G) \setminus V$. For any two vertices $(u,v) \in I$, $(u,v) \notin E(G)$ as otherwise V would not be a vertex cover of G. □

Using this lemma, to reduce independent set to vertex cover (and vice versa), all we have to do is replace k by $n-k$. □

3 NP-hardness and NP-completeness

Definition 3.1. *We say that a problem P is NP-hard if every problem P' in NP is poly-time reducible to P.*

Remark 3.2. *Problems which are NP-hard are not necessarily in NP. If P is NP-hard, this only guarantees that P is at least as hard as any problem in NP, but P may be much harder.*

Definition 3.3. *We say that a problem P is NP-complete if P is both in NP and NP-hard.*

Proposition 3.4. *If P and P' are two NP-complete problems, then P and P' are poly-time reducible to each other.*

Proof. Since P is in NP and P' is NP-hard, there must be a polynomial time reduction from P to P'. Conversely, since P' is in NP and P is NP-hard, there must be a polynomial time reduction from P' to P. □

3.1 Showing problems are NP-complete

In order to show that a problem P is NP-complete, we must do two things:

1. Show that P is in NP.
2. Give a polynomial time reduction from a problem P' which is NP-complete to P.

If we can do this then given another problem P'' in NP, since P' is NP-hard, P'' is poly-time reducible to P'. Since P' is poly-time reducible to P, this implies that P'' is poly-time reducible to P. Thus, P is NP-hard. Since P is in NP as well, P is NP-complete.

However, in order for this process to work, we need a starting problem which is NP-complete. For this, we will use the Cook-Levin theorem which says that circuit SAT is NP-complete.