20191119, 20191121

NP-Completeness

We have just seen that if we could solve SAT in polynomial time, then we could solve every problem in $\mathcal{NP}$ in polynomial time. This is such a powerful property that we use a special term to describe it: we say that SAT is NP-Complete.

The formal definition of this term is:

A problem X is NP-Complete if:

- $X \in \mathcal{NP}$
- $\forall Y \in \mathcal{NP}, Y \propto X$

The practical significance of this is that virtually all computer scientists are convinced that NP-Complete problems cannot be solved by polynomial-time algorithms. When faced with a new problem, we must ask ourselves if it is reasonable to hope for a polynomial-time algorithm. If the problem is NP-Complete, then the answer is almost certainly "No"

So we turn to the question of how we can prove that a problem is NP-Complete. Fortunately we don't have to do what Cook and Levin did (i.e. show how to transform every instance of every problem in NP into an instance of our new problem in an answer-preserving way, in polynomial time.)

Given a problem X in the class $\mathcal{NP}$, if we can show that a known NP-Complete problem reduces to X then we know that X is NP-Complete as well.

Putting this more precisely:

A problem X is NP-Complete if:

- $X \in \mathcal{NP}$
- $\exists$ NP-Complete problem Y such that $Y \propto X$

Why is this equivalent to the previous definition? What we are doing is showing that all problems in $\mathcal{NP}$ reduce to X, but we are using some known NP-Complete problem to stand in for all the other problems in $\mathcal{NP}$. We can do this because we already know that all problems in $\mathcal{NP}$ reduce to Y (because we know Y is NP-Complete). If we can prove that Y $\propto$ X, then the transitivity of $\propto$ tells us immediately that all problems in $\mathcal{NP}$ reduce to X … so X satisfies the original definition of NP-Completeness.

To demonstrate this method of proving that a problem is NP-Complete we will use a variant of SAT called CNF-SAT. CNF-SAT is known to be NP-Complete.

CNF-SAT is defined as follows: Let E be a Boolean expression with k clauses and n literals (literals = Boolean variables, possibly negated), in which
- each clause contains only "OR" connectives
- the only connectives between clauses are "AND"s

CNF-SAT asks "Is there a way to assign True and False to the literals so that E is True?"

Clearly CNF-SAT is in $\mathcal{NP}$ - if the answer is YES and we are told which literals should be True and which should be False, verifying the answer is easy: we just make sure there is at least one True literal in each clause of E.

Example: Let $E = (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_4 \vee x_6) \wedge (\overline{x_3} \vee \overline{x_5})$

In this example E is satisfiable.

I'm not going to prove that CNF-SAT is NP-Complete, but the proof involves showing that any instance of SAT can be re-written in CNF . We will use the fact that CNF-SAT is known to be NP-Complete to prove that a problem called k-Clique is also NP-Complete.

The k-Clique Problem: Given a graph G and an integer k, does G contain a set of k vertices that are all mutually adjacent (connected by edges)?

(Notice that our proof does **not** involve showing that k-Clique reduces to CNF-SAT - we already know this, thanks to Cook and Levin. We need to show that CNF-SAT reduces to k-Clique.)

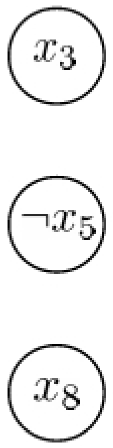
First we observe that k-Clique is in $\mathcal{NP}$: if the answer to an instance of k-Clique is YES and we know the details of the answer, we can easily verify that the required edges are present in the graph.

To show that CNF-SAT reduces to k-Clique, we start with an arbitrary instance of CNF-SAT: a boolean expression E in CNF. We need to construct an instance of k-Clique (i.e. a graph G and an integer k) in such a way that G has a k-clique iff E is satisfiable.

Note that even though we cannot put any restrictions on E, we do know that E is some specific Boolean expression in Conjunctive Normal Form. We can use all details of E when we create G and k.

To motivate the construction of G, we can reflect on the two problems we are trying to connect. If E has a satisfying truth assignment, then we must be able to set (at least) one literal to True in each clause, and of course the things we set to True must be non-contradictory – we can't have x_1 being true in one clause and $\neg x_1$ being true in another clause. A graph G has a k-clique if we can find a group of k vertices in G, such that they are all adjacent to each other. Is there a way that we can transform the literals in E to vertices in G? Can we relate "setting one literal to True in each clause, in a non-contradictory way" to "finding k vertices, all of which are adjacent"?

We can! We'll start by creating a vertex for each literal in E, and group them together according to the clause of E they come from. We'll label each vertex with the literal it represents. So a clause $(x_3 \vee \neg x_5 \vee x_8)$ would be represented by a set of vertices



To satisfy E, we need to make sure there is a true literal in each clause. To relate this to a k-clique in the graph we are building, we need to make sure the k-clique contains a vertex from each clause-group of vertices. We can ensure this by not having any edges within each clause-group. This means that any k-clique in the graph can only contain vertices from different clause-groups.

But we do need edges! Remember that in a satisfying truth assignment for E, the only restriction is that we can't use a literal as True in one clause and False in another. So the corresponding restriction on a k-clique in G is that it cannot contain two vertices with contradictory labels (such as x_4 and $\neg x_4$). To make sure this doesn't happen, we simply omit edges between contradictory vertices.

In summary, we add an edge between every pair of vertices **except for**

- vertices in the same clause-group
- vertices whose labels are direct contractions of each other

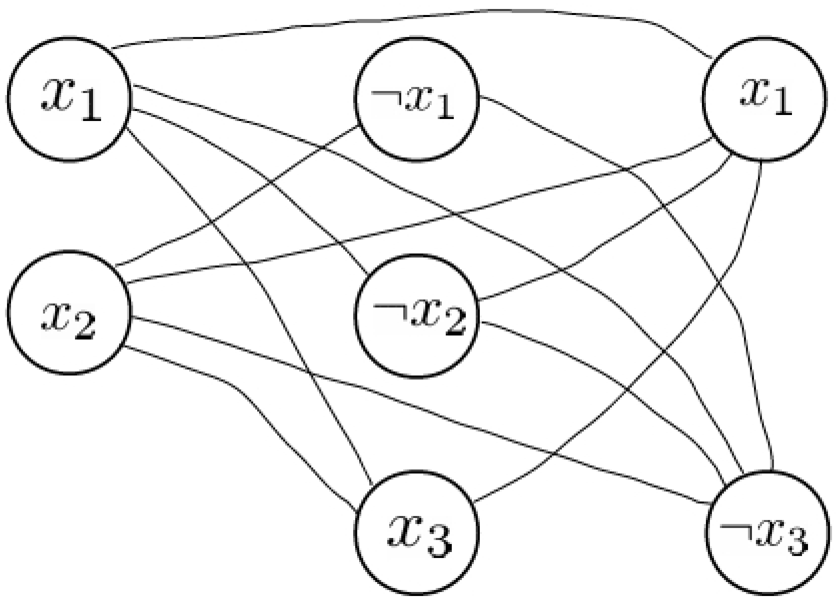
Clearly G can be constructed in O(n^2) time where n is the length of E.

To complete the constructed instance of k-Clique we need the integer k. Note that any clique in G can contain at most one vertex from each of the clause-groups, so the number of clauses in E is an upper bound on the size of the largest clique in G. And if we can find a clique that *does* contain one vertex from each clause-group, that would correspond to choosing one literal from each clause of E. This is exactly what we want to do in a satisfying truth assignment. We let k = the number of clauses in E.

The graph G and the integer k form an instance of k-Clique.

Here is an example (mercifully smaller than the one we did in class):

Let $E = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3)$



So far we have shown that we can transform any instance of CNF-SAT into an instance of k-Clique in polynomial time. Now we have to show that the transformation is answer-preserving.

We will show that the answer to the instance of CNF-SAT is YES iff the answer to the instance of k-Clique is YES.

First direction: Suppose the CNF-SAT answer is YES - i.e. suppose E is satisfiable. Consider any assignment of True and False to the literals in E that makes E true. There must be at least one literal in each clause that is True. For each clause, choose a literal in that clause that is True, and select the corresponding vertex in G.

Claim: these vertices must form a k-clique. First, there are certainly k of them (one from each clause). Suppose that some pair of these vertices are not adjacent. That could only happen if they were in the same clause-group (they aren't) or their labels contradicted each other. But if their labels contradicted each other, that would mean some literal was both true and false in the truth assignment that satisfies E - which can't happen. Thus all of the selected vertices are adjacent to each other, and they form a k-clique.

Second direction: Suppose G contains a k-clique. Due to the way we constructed G, the k-clique must contain one vertex from each clause-group. For each of the vertices in the k-clique, select the corresponding literal in the corresponding clause of E. Set all these literals to be True. There is no danger of attempting to assign True to a variable and to its negation, because the k-clique in G cannot contain any edges between things that negate each other. Now we have one True literal in each clause. Assign True and False to any unassigned literals in any consistent way (i.e. don't attempt to assign both True and False to the same variable) . These assignments are effectively irrelevant because we already have one True literal in each clause of E, which means that E is satisfiable.

Thus our polynomial time reduction from CNF-SAT to k-Clique is answer preserving. Therefore k-Clique is NP-Complete.

Note that when we said "suppose E is satisfiable", we never said anything about how we might actually find the truth assignment that we then use to select vertices in G. Similarly when we said "Suppose G contains a k-clique" we never said anything about how we find the k-clique that lets us choose literals in E. There is no suggestion in this proof that either of these problems can be solved in polynomial time (in fact, the implication is that they cannot). What the proof shows is that IF the answer to either instance is YES, then the answer to the other instance is also YES.

We are still no closer to solving either of these problems but now that we know they are both NP-Complete, we know that finding a polynomial-time algorithm for either one would also give us a polynomial-time algorithm for all problems in NP ... and in practice, we conclude that k-Clique is almost certainly not solvable in polynomial time.

You may ask the perfectly reasonable question "How did anyone come up with that reduction?" The answer, as for so many other difficult problems, is that it probably took a long time, with many false starts. The relationship between these two problems is certainly not an obvious one.