CS 3510 Algorithms Last updated: November 7, 2023

Lecture 20: Introduction to Reductions

Lecturer: Sahil Singla *Scribe(s): Joseph Gulian, Allen Chang*

In the last lecture, we discussed the classes P, NP, Exp, NP-Hard, and NP-Complete. In this lecture we will show more of the problems in these classes. Recall that P is the class of problems that can be solved in polynomial time, NP is the class of problems that have a polynomial time verifier, and Exp is the class of problems that can be solved in exponential time. The class NP-Hard is the class of problems that if solved in polynomial time mean that all problems in NP are also in P (and also that if P $\neq$ NP, there is no polynomial time algorithm for these problems). The final class is NP-Complete which is the class of problems that are NP-Hard and NP.

1 Reductions

As we hinted at in the last lecture, the main technique we'll be using to show more problems are in some class is "reduction". For two problems A and B, when you make a reduction from problem A to B, we're showing that B is at least as hard as problem A. To think about reductions, you could think about if someone gave you some blackbox algorithm for problem B, you want to use that algorithm to solve problem A. If we can take some instance of problem A, transform it into some instance of problem B and solve it with that blackbox algorithm, then B must be at least as hard as A because having a polynomial time algorithm for B would give us a polynomial time algorithm for A. [1]

We have actually already seen a simple example of this with Max-Flow and Maximum Bipartite Matching. Let's forget Ford-Fulkerson and try to make the argument that if such an algorithm existed, this would imply that Maximum Matching also takes polynomial time. So, given a blackbox algorithm to find the maximum flow, let's show that we could use that algorithm to solve Bipartite Matching. We can translate the Bipartite Graph into a flow network by connecting all the nodes in L to a source and all the nodes in R to the sink, and give each edge a capacity of one. This will take polynomial time to do since there are order $|V|$ nodes and $|E| + |V|$ edges. [2] Now that we have our reduction, we know that Maximum Flow must be at least as hard as Maximum Matching because if we can solve Maximum-Flow, then we can solve Maximum Matching.

[1] For this class, we'll be focusing on polynomial time reductions, so turning an instance of problem A into problem B can take at most polynomial time. Again, if you think about what we're trying to show "if B has a polynomial time algorithm, then A must also have a polynomial time algorithm", then it makes sense that your transformation should not take more than polynomial time.

[2] It's important to show that your transformation is polynomial because again, if your transformation is non-polynomial, than the implication that having a blackbox polynomial time algorithm for B gives a polynomial time algorithm for A falls apart.

2 NP-Hard Redefined

Now that we have an appreciation for reductions, let's take another look at the class NP-Hard. In the last lecture we said that solving any problem in NP-Hard would give a polynomial time algorithm for all problems in NP. In other words, all problems in NP reduce to any problem in NP-Hard, meaning that any problem in NP-Hard is at least as hard as all problems in NP. It may seem hard to show a reduction from all problems in NP to any one problem, but this was actually done in the Cook-Levin Theorem (1972) which showed 3-Sat is NP-Complete (NP-Hard and NP). Having this one problem made showing that other problems were NP-Hard much easier because we could simply show that if problem A is at least as hard as problem B and B is at least as hard as all problems in NP, then A must also be at least as hard as all problems in NP and itself be NP-Hard. Karp used this logic to show that 21 other problems were NP-Complete. As you can see here, showing that one problem is NP-Hard through reduction is much easier than showing the problem is NP-Hard by definition. You might guess that, we will mostly be showing problems are NP-Hard by reduction.

3 Cook-Levin

Before we do reductions, we'll try to give a flavor of what the Cook-Levin theorem did. As we said Cook-Levin operated on 3-Sat, so let's first define the problem k-Sat,

3.1 Sat And It's Variations

Recall in true and false, boolean values can either be true (also 1) or false (also 0). On these values we have the operations and $\wedge$, or $\vee$, and not $\overline{x}$ (we'll use overline of a variable to notate not). A boolean formula is a formula involving variables and these operations, so for instance on the variables x, y, z, the formula $(\overline{x} \vee y) \wedge (x \vee \overline{z})$. Each of the groups collected by ANDs are called clauses, so for the above example $(\overline{x} \wedge y)$ and $(x \wedge \overline{z})$. Our boolean formula consists of many of these clauses ORed together. In general satisfying problems, given some boolean formula, we want to assign variables to values such that the formula evaluates to true; this is more generally known as Sat. For the example above, a valid satisfying assignment is $x = 0, y = 1, z = 0$ which makes the formula evaluate to true.

Now that we have an understanding of Sat, k-Sat is simple. In k-Sat problems, each clause can have at most k literals. It turns out that 2-Sat is in P, but for all k-Sat with k greater than 2, the problem is NP-Hard. You should also see that k-Sat problems are in NP because given variable assignments, we can evaluate the formula in polynomial time.

3.2 The Theorem

As discussed above, the Cook-Levin theorem gives a reduction from all problems in NP to the k-Sat. The theorem is a little beyond the scope of the class, so we'll give a broad overview of what it says instead. A problem being in NP means that there is a nondeterministic polynomial time algorithm to solve that problem. We can encode that into a satisfying

expression which could be solved if we had a polynomial time solver for SAT. [3] This essentially shows that SAT is NP-HARD; 3-SAT is also NP-HARD because we can turn any SAT problem into 3-SAT. Since 3-SAT is also in NP, this means 3-SAT is NP-COMPLETE.

What is the significance of this result? As we've said previously, now that we know 3-SAT is NP-COMPLETE, we can use this to show that other problems are NP-HARD. Now we will begin to build out our known NP-COMPLETE problems. We'll discuss several graphical problems, but this can be extended to many more that we won't discuss. To see a broader variety of NP-COMPLETE problems, you could look at Karp's 21 problems.

4 Independent-Set is NP-Complete

The first problem we'll show is NP-COMPLETE is the maximum INDEPENDENT-SET problem on general graphs. With trees, this problem could be done in polynomial time, but on general graphs, it's actually NP-COMPLETE. Recall that for a given graph $G = (V, E)$, a set $S \subseteq V$ is independent if $(u, v) \notin E$ for all $u, v \in S$. Of course we've formulated this as an optimization problem, but we can change it to a decision problem by asking whether there is an independent set of size at least k.

Now, to show that INDEPENDENT-SET, we will show that it is in NP by giving a polynomial time verifier, and show that the problem is NP-HARD by giving a polynomial time reduction from 3-SAT. For the former task, we'll use the chosen vertices S as our certificate and we can check the following things in polynomial time.

- Check in polynomial time whether $\exists (u, v) \in E$ where $u, v \in S$ (whether or not S is even an independent set)
- $|S| \geq k$ (whether or not S is valid under the constraint)

Now to show the problem is NP-HARD, we will take a 3-SAT problem and turn it into a INDEPENDENT-SET. In each clause, turn each literal (e.g. x or $\overline{x}$) and fully connect these literals nodes. Now for a given variable like x, connect all nodes that are x to all nodes that are $\overline{x}$.

Our goal in this construction is to check whether the maximum independent set is exactly size m (the number of clauses). The connections in the first step (between literals in a clause) ensures that only one node in each clause will be selected. Adding this constraint creates the implication that an independent set of size $|S|$, means that exactly $|S|$ clauses can be satisfied, allowing us to check whether $|S| = m$. The second connections (between variables and their complements) ensures that a node can either be true or false, but not both.

We'll continue this next time with an example.

[3] This is simplified to the point of not saying anything but it's hard to explain the true theorem meaning of these without appreciation for languages and models of computation which we don't have time to cover. The main takeaway you should probably have from this section is that SAT and 3-SAT are NP-COMPLETE.