TB Schardl
6.046J/18.410J
November 6, 2009
Recitation 8

Reductions

Recall from yesterday's lecture:

REDUCTIONS ARE USEFUL

Intuitively, suppose you have some problem A that you don't know how to solve. If you can find a way to reduce problem A to some problem B that you do know how to solve, then that's just as good as finding a way to solve A in the first place. You've seen an example of this sort of thing already, when we reduced maximum bipartite matching to maximum flow.

Formally, suppose we have an instance of problem A that we want to solve. Suppose further that we know how to transform instances of problem A into instances of problem B in polynomial time, and we have a machine to solve instances of problem B in polynomial time. We can solve an instance of A in polynomial time by transforming it to an instance of B, which takes polynomial time, solving that instance of B, which also takes polynomial time, and extracting the answer to A from the answer to B in polynomial time. This is a slight generalization from the description of reductions described yesterday. This last step is typically obvious from the original transformation, but sometimes it requires some care.

Implications of Reductions

Reductions also tell us about the relative difficulty of problems. If we have a way of quickly reducing instances of A into instances of B, then solving B is theoretically at least as difficult as solving A. After all, if you can solve B, then you can solve A by using your reduction and the B solver. I believe this is why we notate the fact that A reduces to B with the notation $A < B$ or $A \leq B$ or $A \leq_P B$. This notion is important in complexity theory. If we know that A is NP-complete, and we have a polynomial-time reduction from A to B, then we can conclude that B is also NP-complete. You'll see more of this in lecture next week.

Some Reductions

Let's look at some reductions between problems. Recall the following decision problems.

SAT: Given some boolean formula in CNF, does it have a satisfying assignment?

3-SAT: Given some boolean formula in CNF in which each clause contains 3 literals, does it have a satisfying assignment?

SAT appears to be a generalization of 3-SAT and is intuitively more difficult. However, it turns out we can reduce SAT to 3-SAT, so 3-SAT is just as hard as SAT.

Problem: Reduce SAT to 3-SAT.
Solution: We first create a parse tree of the given boolean formula. For example, suppose we are given $\phi = (x_1 \wedge x_2) \vee \neg((\neg x_1 \vee x_3) \wedge x_4 \wedge \neg x_5) \wedge \neg x_2$. The parse tree for this ϕ looks like the tree in figure 1. After obtaining this parse tree we associate a new variable with each internal node, as shown in the same figure. Using these new variables we can write a new boolean formula

$$\begin{aligned}\phi' = y_1 \wedge (y_1 &\iff (y_2 \wedge \neg x_2)) \\ \wedge (y_2 &\iff (y_3 \vee y_4)) \\ \wedge (y_3 &\iff (x_1 \wedge x_2)) \\ \wedge (y_4 &\iff \neg y_5) \\ \wedge (y_5 &\iff (y_6 \wedge \neg x_5)) \\ \wedge (y_6 &\iff (y_7 \wedge x_4)) \\ \wedge (y_7 &\iff (\neg x_1 \vee x_3))\end{aligned}$$

We now have a bunch of clauses that each have at most 3 literals. We need to convert these clauses into CNF. To do this, for each clause we will write a truth table to see when the clause is true. For instance, if clause $C_1 = (y_1 \iff (y_2 \wedge \neg x_2))$, then the truth table for C_1 is:

y_1	y_2	$\neg x_2$	C_1
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

With this truth table we will examine each row where the value of C_1 is **false**, or where $\neg C_1$ is true. In this example we see that

$$\begin{aligned}\neg C_1 = &(\neg y_1 \wedge y_2 \wedge \neg x_2) \vee \\ &(y_1 \wedge \neg y_2 \wedge x_2) \vee \\ &(y_1 \wedge \neg y_2 \wedge \neg x_2) \vee \\ &(y_1 \wedge y_2 \wedge x_2)\end{aligned}$$

We can apply DeMorgan's laws to this formula to obtain a version of this clause in CNF:

$$\begin{aligned}C_1 = \neg(\neg C_1) = &(y_1 \vee \neg y_2 \vee x_2) \wedge \\ &(\neg y_1 \vee y_2 \vee \neg x_2) \wedge \\ &(\neg y_1 \vee y_2 \vee x_2) \wedge \\ &(\neg y_1 \vee \neg y_2 \vee \neg x_2)\end{aligned}$$

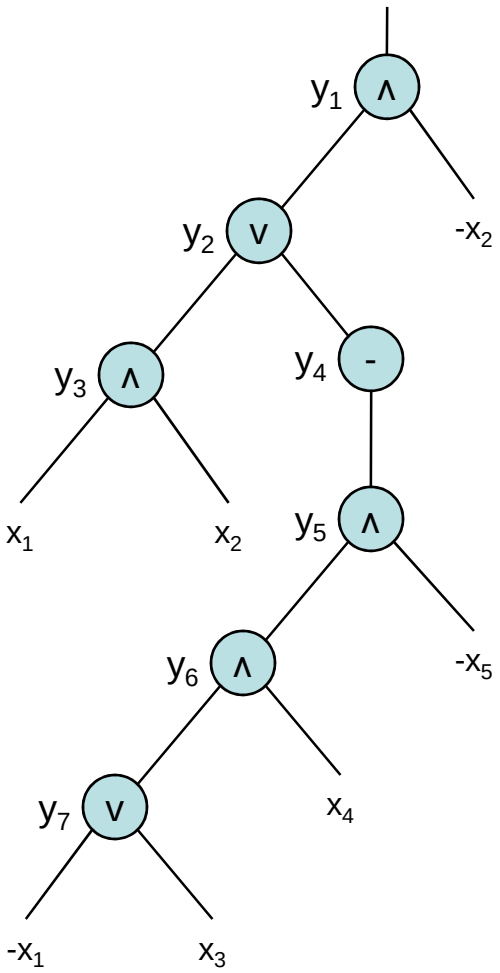

Figure 1: Parse tree for boolean expression $\phi = (x_1 \wedge x_2) \vee \neg((\neg x_1 \vee x_3) \wedge x_4 \wedge \neg x_5) \wedge \neg x_2$. Each internal node has been associated with a new variable in order to define a new ϕ' boolean expression with clauses containing at most three literals.

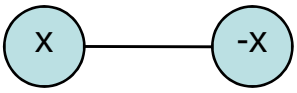

Figure 2: Variable gadget for 3-SAT to k-Vertex Cover reduction. The variable x in a ϕ for 3-SAT is replaced by two nodes in a graph. One node represents the literal x, while the other represents the literal $\neg x$.

To finish this reduction we have to deal with the clauses that have fewer than 3 literals in them. We can use a simple padding technique on these clauses. Suppose for instance that we have some clause $(x \vee y)$. This clause is equivalent to $(x \vee y \vee p) \wedge (x \vee y \vee \neg p)$ because, no matter what value of p we choose, one of x or y must be true for these clauses to be true, which is exactly the condition that satisfies the original clause. We can therefore pad clauses with 2 variables by adding an additional variable and replicating the clause appropriately. A similar technique can be used to handle clauses with a single variable.

The correctness of this reduction can be observed from the correctness of each step. We have to verify that this reduction creates a boolean formula for 3-SAT that is polynomial in size, and that this process takes polynomial time. Notice that parsing the original boolean formula into the tree takes polynomial time and generates at most a binary tree. A binary tree with n leaves contains $2n$ nodes in total, so by transforming each internal node of this tree into a new clause we create a only a polynomial number of clauses. Transforming each clause into CNF requires examining at most 8 entries in a truth table since each of these clauses contains at most 3 literals. Therefore the transformation to CNF multiplies the number of clauses by at most 8. Finally the padding step at most doubles or quadruples the number of clauses to handle clauses that were too small. Each step takes polynomial time and multiplies the number of clauses in the formula by at most a polynomial amount, and therefore the result is polynomial in size as desired.

With this reduction we have proven that 3-SAT is at least as hard as SAT. This is useful for showing that other problems are at least as hard as SAT. Recall the following decision problem.

Vertex Cover: Given an arbitrary graph, can we find a subset of k nodes in the graph such that every edge touches one of the nodes in this subset?

Problem: Reduce 3-SAT to Vertex Cover.
Solution: Suppose our boolean formula for 3-SAT, ϕ, contains m variables and l clauses. For our vertex cover we will use $k = m + 2l$. We will use the following two gadgets to perform our reduction. For each variable in ϕ we will use the variable gadget described in figure 2. For each clause in ϕ we will use the clause gadget described in figure 3. Connect the literals in the variable gadgets to the matching literals in the clause gadgets with an edge. For example, if we have $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \geq x_3)$ then we are looking for a vertex cover of size $k = 3 + 2(3) = 9$ on the graph in figure 4. At this point the reduction is complete.

We can see that this reduction requires polynomial time and polynomial space. This reduction requires one gadget of 3 vertices and 3 edges for each clause in ϕ, and one gadget of 2 vertices and 1

Re

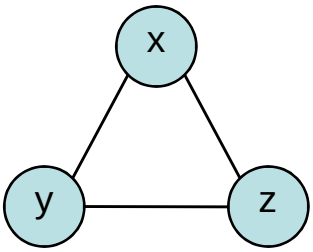

Figure 3: Clause gadget for 3-SAT to k-Vertex Cover reduction. The 3-SAT clause is represented as a clique of 3 nodes, where each node represents a literal in the clause. The clause represented here is $(x \vee y \vee z)$.

edge for each variable in ϕ. Each literal in each clause gadget is connected with a single additional edge to a literal in a variable gadget, so we add 3 edges for each clause gadget to connect the clause and variable gadgets. Therefore this reduction has polynomial size and requires polynomial time to compute.

We now have to prove that this reduction is correct, meaning that a vertex cover of size $k = m + 2l$ exists if and only if the given ϕ is satisfiable. Suppose we have a satisfying assignment for ϕ. For each variable, add the node corresponding to the true version of the literal for that variable to the vertex cover. Then, for each clause, select one true literal in the clause and add the remaining two literals to the vertex cover. We have used 2 vertices in each clause gadget and 1 vertex in each variable gadget for this cover, which meets the size requirement for the cover. Each edge in a variable gadget is covered by the node selected from that gadget. All three edges in each clause are covered by the two nodes we selected in that clause gadget. One true literal in each clause gadget is left out of the cover, but because it is a true literal it is connected by an edge to the node corresponding to the true literal in a variable clause. Therefore these nodes cover all of the edges in the graph, and we have a valid vertex cover.

We must still prove the other direction of this implication. If we have a vertex cover of size k for this graph, that cover must contain one node in each variable gadget and two nodes in each clause gadget to cover the edges in those gadgets. This requires exactly $k = m+2l$ nodes. Suppose we take literal corresponding to a covered node in a variable gadget to be true. This assignment satisfies ϕ because, for each clause gadget, each of the three edges connecting the clause gadget to the variable gadgets is covered, and only two nodes in the clause gadget are in the cover. Therefore one of these clause-gadget edges must be covered by a node in a variable gadget, and so the assignment of that covered variable gadget literal to true in ϕ will satisfy the clause. This holds for every clause gadget and clause, so this assignment satisfies ϕ. Therefore a k-covering of this graph corresponds to a satisfying assignment for ϕ, while a satisfying assignment for ϕ corresponds to a k-covering of this graph, and this reduction is correct.

We also know that reductions are transitive. If we can quickly reduce A to B, and we can quickly reduce B to C, then we have an obvious way to quickly reduce A to C. The implication on problem difficulty is also transitive. If $A \leq B$ and $B \leq C$, then $A \leq C$.

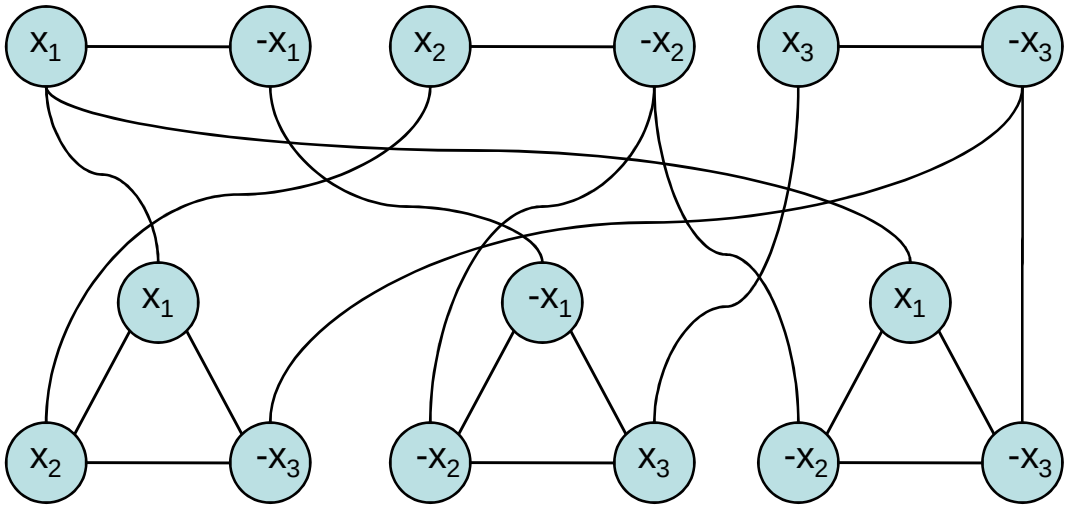

Figure 4: Example graph from 3-SAT to k-Vertex Cover reduction. The boolean formula for this example is $\phi = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \geq x_3)$. The variable and clause gadgets for this boolean formula are visible in this graph, as well as the edges connecting nodes from the literals in the variable gadgets to matching literals in the clause gadgets.

Another Kind of Reduction

We can develop a more general kind of reduction than what we've described before. Let's suppose you're stuck on a problem on a problem set. Since you need to finish the problem set soon, you decide to go to office hours to get some advice on solving the problem. However, you know that the TA is very strict and will not tell you the solution directly. Instead he will only answer very carefully crafted yes-no questions.

What you could do is first craft some yes-no question from your problem, ask the TA, and get some answer. Based on that answer you can craft another question for the TA, ask the TA, and get another answer. Then, based on that second question and answer, along with all your previous knowledge about the problem, you craft a third question for the TA, ask the TA, and get another answer. You can continue this process for a whole bunch of iterations until you get the answer to your problem, and if you can ensure that your sequence of questions is short enough, then you can finish the problem set on time. This is basically the framework for another kind of reduction.

We can formalize this kind of reduction further. Suppose we have some algorithm for solving instances of problem B in polynomial time. If we can reduce an instance of problem A to solving a polynomial number of instances of B, then we can also solve A in polynomial time. Furthermore, the instances of B that we use may depend on the results of previous reductions. This kind of reduction is called the "adaptive Cook reduction."

For instance, we may solve an instance of A, x, using an algorithm that solves instances of B, T_B, and a reducer R that takes an instance of A and an arbitrary number of previous reductions from A to B with their results as follows.

1. $x \to R(x) = x_1$. Run $T_B(x_1) = y_1$.
2. $x \to R(x, x_1, y_1) = x_2$. Run $T_B(x_2) = y_2$.
3. $x \to R(x, x_1, y_1, x_2, y_2) = x_3$. Run $T_B(x_3) = y_3$.
4. Repeat at most a polynomial number of times.
5. Return the answer to x.

This kind of reduction is very useful for reducing search problems to decision problems. For example, if we have an algorithm $T_{SAT?}$ that tells us if some boolean formula has a satisfying assignment, we can determine what that satisfying assignment is using this kind of reduction. (How?)

Let's examine one last problem that demonstrates this kind of reduction.

Problem: Suppose I have an algorithm $T_{k-VC?}$ that, given a graph $G = (V, E)$ and a parameter k, returns whether or not the graph has a k vertex cover. How can find a k-node vertex cover of the graph G?

Solution: First ask $T_{k-VC?}$ if there is a k vertex cover for G. If it says no, then no k vertex cover exists for G, so give up. Otherwise, pick some vertex $x \in V$, and remove x and all edges incident to x from G. Ask $T_{k-VC?}$ if this new graph, $G-x$, has a $k-1$ vertex cover. If so, then we know that x can be in a k vertex cover of the original graph G, so we add x to our vertex cover and recurse this whole process on $G - x$ and $k - 1$. If $T_{k-VC?}(G - x, k - 1)$ returns "no," then we know that x cannot be in a k vertex cover of G. By definition of vertex cover, we therefore know that some neighbor of x must be in the k-vertex cover of G. Consequently we can remove $y \in N(x)$, one of x's neighbors from G, include y in our vertex cover, and recurse this process on $G - y$ and $k - 1$.

After k iterations we will have found k vertices that cover G, so assuming that $T_{k-VC?}$ runs in polynomial time, then this entire process runs in polynomial time. Thus we have reduced the problem of finding a k vertex cover of a graph to the problem of deciding if a k vertex cover exists for that graph.