1. Introduction

Our goal in these notes is to review reductions. We’ll go over the definitions of reductions and then prove one example reduction to hopefully convey some intuition for constructing reductions.

2. Definition of Reductions

Before attempting to do reductions, make absolutely sure you understand how to construct a reduction. We review the definitions below.

The first kind of reduction we learned was Turing reducibility. We did not go over a formal definition of this in class (although it is in section 6.3 in Sipser’s textbook), but the idea of this type of reduction is as follows: we suppose problem \( B \) is decidable for sake of contradiction and show that, as a result, problem \( A \) is decidable. We knew that problem \( A \) was not decidable beforehand, so this gives us a contradiction and shows us that \( B \) is not decidable. Note that this type of reduction cannot prove Turing-unrecognizability. This type of reduction will also not show NP-completeness, PSPACE-completeness, NL-completeness, or EXPSPACE-completeness. Essentially, this reduction is only useful for proving undecidability.

We say \( A \) is mapping reducible to \( B \) (written \( A \leq_M B \)) if there exists a computable function \( f : \Sigma^* \rightarrow \Sigma^* \) such that \( f(w) \in B \) if and only if \( w \in A \). That is, \( f \) must send strings in \( A \) to strings in \( B \), and \( f \) must send strings outside of \( A \) to strings outside of \( B \). The key here is that \( f \) must be computable; that means there must exist a Turing machine that, on every input \( w \), halts with the output \( f(w) \) on its tape. This means, importantly, that that Turing machine cannot loop on some input. An example of mapping reducibility that we proved is \( A_{\text{TM}} \leq_M \overline{E_{\text{TM}}} \). Mapping reducibility can prove Turing-unrecognizability.

We say \( A \) is polynomial time mapping reducible, or just polynomial time reducible to \( B \) (written \( A \leq_P B \)) if there exists a polynomial time computable function \( f : \Sigma^* \rightarrow \Sigma^* \) such that \( f(w) \in B \) if and only if \( w \in A \). Note here that this is just a specific type of mapping reducibility with the extra condition that \( f \) must be computable in polynomial time. Examples of this type of reduction are \( A \leq_P \text{SAT} \) for every \( A \in \text{NP} \), \( 3\text{SAT} \leq_P \text{CLIQUE} \), \( 3\text{SAT} \leq_P \text{HAMPATH} \), \( \text{TQBF} \leq_P \text{GENERALIZED \ GEOPGRAPHY} \), and \( A \leq_P \text{EQ}_{\text{REX}} \) for every \( A \in \text{EXPSPACE} \).

Finally, we say \( A \) is log space reducible to \( B \) (written \( A \leq_L B \)) if there exists a log space computable function \( f : \Sigma^* \rightarrow \Sigma^* \) such that \( f(w) \in B \) if and only if \( w \in A \). Examples of this reduction are \( A \leq_L \text{PATH} \) for every \( A \in \text{NL} \).

Notice the pattern between these last two types of reducibility: they are just specialized types of mapping reducibility. That is, they are just mapping reducibility with extra constraints (polynomial time and log space respectively).
We focus on the mapping reductions in this review (specifically polynomial time reducibility and log space reducibility). Notice the structure of these reductions. To show $A \leq B$, we are proving there exists a function $f$ that sends strings to strings such that $f(w) \in B$ if and only if $w \in A$. That’s the key: when proving mapping reducibility, polynomial time reducibility, or log space reducibility, you should be describing a function that sends strings to other strings with the constraint mentioned above.

How this generally goes is we focus on problem instances of $A$ and $B$. For instance, if $A$ is 3SAT and $B$ is CLIQUE, we focus on functions that send 3cnf boolean formulas $\phi$ to pairs $\langle G, k \rangle$, where $G$ is a graph and $k$ is a positive integer. We want $G$ to have a $k$-clique if and only if $\phi$ is satisfiable. You might ask, “what happens if $\phi$ is actually some garbage input that can’t be read as a boolean formula at all?” We don’t really focus on this case, since the mapping reduction $f$ is assumed to check the input and make sure it’s reasonable first. If the input is not reasonable, $f$ then immediately knows to output some string that is not in the target language $B$ (note that in all the types of mapping reducibility we discussed, the function $f$ is capable of reading the entire input).

3. 3COLOR

In this section, we show the 3COLOR problem is NP-complete.

This is problem 7.29 in the textbook. The 3COLOR problem is a very interesting and famous problem—you might have heard of it already. The problem is as follows: given a graph, we ask if there is a way to color all the nodes of the graph using at most three different colors such that if two nodes are connected by an edge, they are not the same color. Put differently, we want to color the graph so the same color is never adjacent to itself. We call such a coloring a 3-coloring.

So we define $3\text{COLOR} = \{\langle G \rangle | G$ is colorable with 3 colors $\}$. Our goal is to show that $3\text{COLOR}$ is NP-complete.

Here’s our plan: to show a language is NP-complete, we need to show that it is in NP and that any language in NP reduces to it (i.e. that the language is NP-hard). Let’s tackle the first part.

**Lemma 3.1.** $3\text{COLOR}$ is in NP.

*Proof.* For any graph $G$, we can define its certificate $c$ to be a valid 3-coloring of $G$. This certificate can be verified in polynomial time in the length of $G$ because we can check that every node has a different color from all of its neighbors. □

Next, the more complicated part: showing that everything in NP is polynomial-time reducible to $3\text{COLOR}$. Remember that our main strategy for showing a problem is NP-hard is reducing some NP-complete problem to it. But how do we pick such a problem?

The first and most important thing to note is that picking any NP-complete problem works in theory. This must be true since we are trying to prove that some language (in this case $3\text{COLOR}$) is NP-complete, so every language in NP must be reducible to it (importantly, this includes the NP-complete languages).

However, the second most important thing to note is that some problems will be easier to reduce from than others. There may be some inherent structure of one problem that is very similar to the problem we are trying to show is NP-complete. Ultimately, a reduction $A \leq B$ is viewing a problem instance of $A$ as a problem
instance of $B$ in a reasonable way. What does reasonable mean? It just means that the latter problem instance is in $B$ if and only if the former problem instance is in $A$. This "viewing" is the tricky part of reductions, but it tends to be easier to do if you choose a good language to reduce from.

Let’s see this in practice: 3COLOR is a language about graphs, so we might think to reduce from another NP-complete language about graphs (e.g. CLIQUE or HAMPATH). But if we look at little closer, it’s not really the graph aspect of the problem that is most important: it’s the satisfiability of the colorings of the graphs. Indeed, to show $G \in 3COLOR$, we want to show that $G$ has some valid coloring (the key word here being valid). This is very similar to showing $\phi \in 3SAT$, where we want to show that $\phi$ has some satisfying assignment.

So, let’s try reduce $3SAT \leq_p 3COLOR$. How do we do this? It’s always helpful to start with the end in mind: we want to map a boolean formula $\phi$ to a graph $f(\phi)$ such that $f(\phi)$ has a valid 3-coloring if and only if $\phi$ is satisfiable (where $f$ is our mapping reduction function). This seems difficult to do when you try to look at the whole boolean formula, so let’s break it down into parts.

Breaking the reduction down into parts is the key idea behind gadgets. Gadgets are ways of representing key substructures of the problem instance $w$ as key substructures of the problem instance $f(w)$. By combining these substructures in $f(w)$ in an equivalent way to how they were combined in $w$, we will end up with a valid reduction.

What could be our gadgets for this reduction? Well, the boolean formula $\phi$ is comprised of variables arranged in 3cnf form (since $\phi \in 3SAT$). So let’s first think of a variable gadget (i.e. how to represent each variable of $\phi$ as some subgraph in $f(\phi)$).

What are the key characteristics that should be exhibited by a variable gadget in order to reflect the characteristics that its associated variable has in $\phi$?

The first one we might think of is that we can assign each variable to be True or False. That’s critical–and more importantly, if we assign the variable $x_i$ to be True, then the literal $x_i$ will be True, and the literal $\overline{x_i}$ will be False.

How do we embody these key characteristics in a variable gadget? Let’s first think of each variable as two nodes connected by a horizontal edge. Let’s designate the left node to be representing $x_i$ and the right node to be representing $\overline{x_i}$. Since the two nodes are connected by an edge, if we want to try to make a satisfying 3-coloring of the graph, the two nodes must take on different colors, which is what we wanted. But now we have another small problem: there are 3 colors but only two values a boolean variable can take on (True or False). So let’s prevent the two nodes here from being a specific color.

How do we do that? See Fig 1.

Note that we don’t actually color the two nodes labeled $T$ and $F$–we don’t precolor any nodes of the graph. It’s just a way to think about those two nodes, since we’ll be connecting all the variable gadgets to that third (bottom right node). This way, no matter which colors the $T$ and $F$ nodes will be, the variable nodes will be those same colors. Remember that we’re viewing the left node as $x_i$ and the right node as $\overline{x_i}$–this will be a bit important later on. We’ll also come back to the palette part of Figure 1 soon.

Now that we’ve represented the variables, let’s make a gadget for the clauses (the other key substructure of a boolean variable $\phi$). All of the variables in a cnf clause are OR-ed together, so let’s create an OR gadget.
What’s the key characteristic of the OR boolean function ($\lor$)? If at least one variable is True, the OR evaluates to True, otherwise, it doesn’t. To make things easier to think about, let’s say we are trying to color the graph with the 3 colors red, blue, and green. Let’s suppose that red means True, blue means False, and green is just the third color.

So we want two red nodes—let’s call them $A$ and $B$—to somehow result in a third node $C$ being red. We can’t just connect $A$ and $B$ to $C$, because then $C$ won’t be red for sure. So let’s first connect $A$ and $B$ to intermediate nodes $A'$ and $B'$, respectively. Then $A'$ and $B'$ can be both connected to $C$ so that if $A'$ is blue and $B'$ is green, then $C$ must be red! Moreover, $C$ will be red if $A'$ is green and $B'$ is blue. But how do we guarantee that $A'$ and $B'$ are different colors? We connect them with an edge.

The structure we just described is given in Figure 2. This gives us our desired behavior: if $A$ and $B$ are both red, then $C$ can be red. If $A$ or $B$ is red, then $C$ can be red. I say "can" because we could also color $C$ blue or green in these later two cases. To get rid of the green color possibility, let’s connect the output node to the bottom right node of the palette (not shown in the figure). Later, we’ll see why the blue color possibility does not matter.

How do the variable gadgets tie into the OR gadget? We’ll explain by example: for a clause $x_1 \lor x_2 \lor \overline{x_3}$, we first use the OR-gadget on the literals $x_1$ and $x_2$: that is, the left nodes of the $x_1$ and $x_2$ variable gadgets are exactly nodes $A$ and $B$ in Figure 2. Then we take the $C$ node of this OR-gadget and use it as the $A$ node of a new OR gadget, where the $B$ node is the right node of the $x_3$ variable gadget. (Try drawing this out for yourself, then see Figure 3.)

Note that this sort of structure is exactly why we’d expect with the OR-gadget given that we designed it to represent the OR-function: just like with boolean ORs, we can view $x_1 \lor x_2 \lor \overline{x_3}$ as $((x_1 \lor x_2) \lor \overline{x_3}$.

This explains how each clause in $\phi$ is represented in $f(\phi)$. Now, we put the clauses together. There is indeed a possible AND gadget we could make, but we can do something even simpler: we connect all the output nodes of the clause gadgets to a single, new node. We also connect this single, new node to the bottom right node of the palette (in Figure 1. We’ll call this single, new node the final node.

Now, we’ve finished the reduction; here’s why: the final node will only be colorable if all of the output nodes of the clauses are the same color (remember that the output

![Figure 1](image-url)
Figure 2. $A$ is the bottom left node, $B$ is the bottom right node, $A'$ is the middle left node, $B'$ is the middle right node, and $C$ is the top node. Connection to palette node not shown. Sipser, Introduction to Theory of Computation Problem 7.29 Hint.

nodes cannot be the color of the bottom right node of the palette). Continuing with our red, blue, green visualization, the nodes must be then all red or all blue. This means that if one clause has all blue literal nodes and another clause has all red literal nodes, then the graph will not be 3-colorable overall.

Let’s now prove validity of the reduction.

Proof. If $\phi$ is satisfiable, then we can color all the true literal nodes red and all the false literal nodes blue. Then there will be at least one red node per clause gadget, so the output node can be colored red. Then the final node can be colored blue, and the graph is 3-colorable.

If $f(\phi)$ is 3-colorable, then all the output nodes must be the same color. Take that color to be the True color: assign all literals with that color to be True. Then there must exist one True literal in every clause, so $\phi$ is satisfiable. □

4. Closing Remarks

In the 3COLOR proof, the key points were choosing to reduce from 3SAT (since the validity aspect of both problems were very similar) and creating gadgets in our reduction (to build the reduction up from smaller, easier to understand parts). When trying to do a reduction, you may find it helpful to ask two question. The first is, “which problem is similar to the problem I’m trying to reduce to?” The second question, which may help in constructing parts of the reduction (gadgets or not) is: “why are these two problems similar?” Leveraging the answer to the second question can help guide you to a valid reduction.
Figure 3. Two OR gadgets combined to make a clause gadget. The output node is the topmost node. Connections to the palette node are not shown. Sipser, *Introduction to Theory of Computation* Problem 7.29 Hint.