CS 121 Reductions Guide

CS 121 Teaching Staff

Fall 2019

Contents

1 Introductions

Reductions are the main tool we have to understand the *hardness of computational problems*. Here are several equivalent ways you might hear people talk about reductions:

1. F reduces to G.
2. The problem of solving F boils down to the problem of solving G.
3. We can transform any instance of problem F into an instance of solving G, such that if we could solve G, we could also solve F.
4. F is "at most as hard" as G. If F is "hard", then G is "hard".

There are several different types of reductions, based on the type of hardness we're trying to understand. For instance, we've seen *uncomputability reductions*, which we can use to understand whether certain problems are computable (a problem is uncomputable if it can be reduced from $HALT$). We've also seen *polynomial-time reductions*, which we can use to understand things like whether certain problems are in P.

Note that reductions have a *direction*. We don't expect you to remember which one we call "to" and which one we call "from", but you should know that it is quite possible that F reduces to G but G does not reduce to F. For instance, in the case of uncomputability reductions, we observe that every computable problem reduces to $HALT$, whereas $HALT$ never reduces to a computable problem:

1. Consider the following uncomputability reduction from any computable function F to $HALT$: $R(x)$ is a Turing machine that, on input x, first computes $F(x)$. Then if $F(x) = 1$, it outputs the constant zero Turing machine; if $F(x) = 0$, it outputs the infinite loop Turing machine.

2. However, if $HALT$ reduces to any F, that means F is also uncomputable (if F were computable, then $HALT$ would also be computable).

In some sense, this implies that $HALT$ is the "hardest problem" for uncomputability reductions — by reducing $HALT$ to another problem, we are showing that that is also a "hardest problem". Similarly, in the case of polynomial-time reductions, all problems in NP reduce to $3SAT$, while we know no reduction from $3SAT$ to any problem in P:

1. We will see that all problems in NP reduce to $3SAT$; this is called the *Cook-Levin theorem*. This proof is somewhat involved. However, it is not very difficult to show that all problems in P reduce to $3SAT$. Consider the following polynomial-time reduction from any function $F \in \mathsf{P}$ to $3SAT$: $R(x)$ is a Turing machine that, on input x, first computes $F(x)$. Then if $F(x) = 1$, it outputs the formula $\varphi_1(x_1, x_2) = x_1 \wedge x_2$; if $F(x) = 0$, it outputs the formula $\varphi_0(x_1, x_2) = x_1 \wedge \overline{x}_2$. This is polynomial-time because we have assumed that F is solvable in polynomial time.

2. We do not believe that there is a reduction from $3SAT$ to any problem in P (this is equivalent to the $\mathsf{P} \overset{?}{=} \mathsf{NP}$ conjecture).

In the same way, we know by the above that $3SAT$ is the "hardest problem in NP" for polynomial-time reductions. If we reduce from $3SAT$ to any other problem in NP, then this is also a "hardest problem". All these "hardest problems in NP" are collectively called the NP*-complete problems*.

A final important observation is that the format of an *instance* depends on the particular problem. An instance is an input to the problem $F : \{0,1\}^* \to \{0,1\}$ that generates either a "yes" (1) or a "no" (0) answer. For some problems, an instance is what you'd expect — such as for SAT problems, an instance is usually a formula ϕ (and the question is "Is ϕ satisfiable?"). However, for e.g. graph problems, the instance is often more than just a graph — e.g. for $ISET$, the question is "Does G have an independent set *of size* k?", which means a single instance is a pair (G, k).

2 Goals

The goal of this document to give examples of fully fleshed-out and well-written polynomial-time reductions. As such, we will provide:

1. The general format of a solution to a "prove NP-completeness"-type question (which includes the reduction),

2. Examples of actual solutions, and

3. Some miscellaneous tips.

TL;DR:

Being in NP + NP-hardness = NP-completeness.

Three steps to prove F is NP-hard. Four steps to prove that F is NP-complete.

3 Reduction format

Suppose that you want to prove that F is NP-complete. The format of your answer should look like this:

1. Prove that $F \in \mathsf{NP}$. You will need to explicitly explain these four things:

(a) What a valid certificate w for an input x looks like,

(b) What the verifying program V does, i.e. how to determine that $V(x, w) = 1$,

(c) Why V is polynomial-time (with respect to the length of its input), and

(d) Why x has a valid witness iff $F(x) = 1$, i.e.

$$(\exists w : V(x, w) = 1) \iff F(x) = 1.$$

You can give a high-level implementation of V and the corresponding high-level runtime analysis. For instance, you could say "V is $O(n)$ because it makes a single pass through the graph representation." However, do not merely state that V is polynomial-time without any justification, as obvious as that might be to you. Also, note that polynomial-time is always with respect to the total length of input, not number of vertices or number of clauses.

2. Prove that F is NP-hard, by reducing *from* a known NP-hard problem H. In other words, we reduce H to F:

$$H \leq_p F.$$

Typical choices for H include $3SAT$, $MAXCUT$, or $VERTEXCOVER$.

First, you must mention explicitly which H you chose. Let x denote an arbitrary input to H. Then your reduction will contain the following three steps:

(a) Give a high-level implementation of the transformation (program) R, which will transform the inputs of H to the inputs of F, i.e. $x' = R(x)$.

(b) Perform a runtime analysis of R and show that R is polynomial-time (w.r.t. $|x|$, the length of its input).

(c) Prove that your reduction is correct, i.e. prove that

$$H(x) = F(R(x))$$

$F(X) = 3SAT(R(X))$

(or $H(x) = F(x')$) always holds by showing both implications:

i. *Completeness*: Show that

$$H(x) = 1 \implies F(R(x)) = 1.$$

That is, assume some arbitrary input x such that $H(x) = 1$, apply your transformation $x' = R(x)$, and show that by definition of F, it must be true that $F(x') = 1$.

ii. *Soundness*: Show that

$$F(R(x)) \implies H(x) = 1.$$

That is, assume some arbitrary input x' such that $F(x') = 1$, reverse your transformation, and show that by definition of H, any x which could have been transformed into x' must be such that $H(x) = 1$.

These four steps are exactly what you have to do to show that F is NP-hard; these steps together form the definition of NP-hardness. *It is highly important that you distinguish these four steps and keep them separate in your proof.* Clearly indicate where you are proving correctness, where you are proving runtime etc. This helps with the clarity and readability of your proof, and makes it less likely to make a mistake.

4 How to devise the transformation

Even after understanding how to prove a statement that $F \in \mathsf{NP}$, it's often hard to know how to come up with the transformation. This can often seem like an unapproachable, binary problem (i.e. you either have it or you don't), and this section aims to give strategies to help you come up with transformations, hopefully making the process a bit more linear.

At a high level, when we reduce F to G, we are saying that "if we can compute G with G efficiently, then we could compute F efficiently". We do this by creating a polynomial time algorithm R such that

$$F(x) = G(R(x))$$

We can call $R(x)$ our "transformed input" since that's what we get when running our initial input (it may be a graph, CNF, etc.) through our transformer.

We will then end up proving correctness for this algorithm, which is essentially proving $F(x) = 1 \Rightarrow G(R(x)) = 1$ and $G(R(x)) = 1 \Rightarrow F(x) = 1$. Usually we prove $F(x) = 1 \Rightarrow G(R(x)) = 1$ by supposing that $F(x) = 1$, so supposing there is a solution for an input, and then using this to derive a solution to G for our transformed input. To prove $G(R(x)) = 1 \Rightarrow F(x) = 1$ we'll assume the existence of a solution on our transformed input and use this to derive a solution for our initial input. As such, a helpful way to devise the transformation is to think about similarities in solutions between F and G.

This is a little bit abstract, so let's make it concrete by doing this for a specific problem. Let's take the example of reducing $VERTEXCOVER$ to $DOMINATINGSET$ (which is solved below). The first question we ask is what a solution to either of these problems looks like.

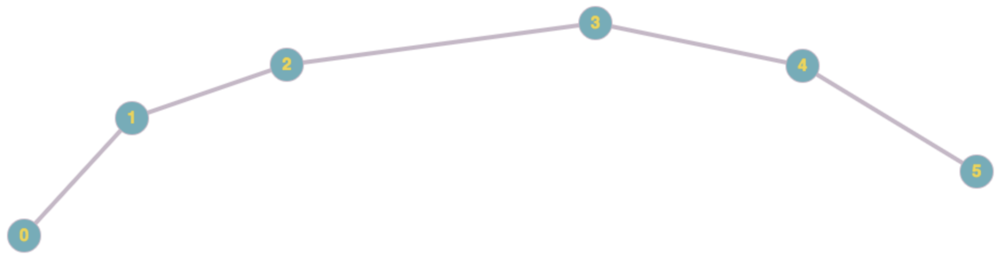
	VERTEXCOVER	DOMINATINGSET
What does a solution to this problem look like?	A set of k vertices S such that $\forall\{u, v\} \in E$, either $u \in S$ or $v \in S$	A set of k vertices S such that $\forall v \in V - S, \exists s \in S : \{v, s\} \in E$

We want to transform a given instance of $VERTEXCOVER$ ("Does graph G have a vertex cover of size k"?) to an instance of $DOMINATINGSET$ ("Does graph G' have a dominating set of size k'?"). We immediately see a similarity here, in that the solution to both problems is a set of size k, so maybe the "solution for the initial input" will be the same as the "solution for the transformed input". But how can we choose the transformation?

Comparing solutions here is immediately helpful because we see that in a graph with no isolated vertices (which we can assume because a vertex cover is preserved adding any number of isolated vertices), a solution to $VERTEXCOVER$ is actually a solution to $DOMINATINGSET$! This is because if we have a set of k vertices S satisfying the $VERTEXCOVER$ properties, $\forall v \in V - S$, we know v must be part of an edge, and every edge has an endpoint in S. This implies an idea for a transformation: Do nothing (i.e. have $G' = G$)![1] Then a $VERTEXCOVER$ on the input into a solution for $DOMINATINGSET$ on the transformed input.

However, we need to think about $G(R(x)) = 1 \Rightarrow F(x) = 1$. Suppose that we have a solution for $DOMINATINGSET$ on the transformed input G'. What does that imply about the properties of the graph G' under $VERTEXCOVER$? We see that a solution to $DOMINATINGSET$ is almost a solution to $VERTEXCOVER$, but there is one issue. A solution to $DOMINATINGSET$ essentially needs to touch all *vertices*, but it doesn't have to touch all *edges*. For example, consider the graph below, in which $\{1, 4\}$ is a dominating set that is not a vertex cover:

[1] It's often a great strategy to try and pick the most naïve approach possible to solve a problem and then see why it fails.

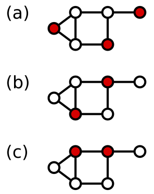


Thus, “doing nothing” for our reduction is not good enough. The core issue is that a $DOMINATINGSET$ solution for G' does not always ensure a $VERTEXCOVER$ solution for G' because it may miss edges between vertices it dominates (but are not included in S). At this point, we might brainstorm ideas to solve this problem (i.e. how can we set up G' such that G' has a $DOMINATINGET$ solution of some size iff G has a $VERTEXCOVER$ solution). What if for each vertex v in the initial graph G we made another vertex v' and added the edge $\{v, v'\}$? Maybe we could connect all the vertices to a new central node? We would see both of these ideas fail, but it’s good to try and generate ideas to get the creative juices flowing. The core issue is that a “dominating set” doesn’t have to cover all the edges, so we could “force” it to cover all the edges by for each edge $\{u, v\} \in E$ adding another point z and the edges $\{u, z\}, \{v, z\}$. Evaluating this, we see it works!

5 Intermediate graph example: Dominating set

Problem: Let $DOMINATINGSET : (G(V, E), k) \to \{0, 1\}$ be the function that takes an undirected graph G and an integer k, and outputs 1 iff there is a dominating set with k vertices. A dominating set is a set of vertices X such that every vertex in G is either *in* X, or *connected to* a vertex of X by an edge, or both. Show that $DOMINATINGSET$ is NP-complete.

Solution: It can be a little hard to visualize a dominating set, so here is a picture:



Thus $DOMINATINGSET(G, 2) = 1$ for the graph in the above examples.

1. Proof that $DOMINATINGSET \in NP$:

   We choose the certificate w for any graph with a dominating set of k vertices to be the set of k vertices that comprises the dominating set. We denote this set to be S. (In the picture example, it would be any pair of 2 red vertices).

   The verifier program $VERIFY$ (renamed because V represents vertices of our graph) will take as input the graph G and S. $VERIFY$ will do the following: Iterate over each vertex $v \in V$ and check if the vertex is either in the dominating set ($v \in S$), or adjacent to a vertex in the dominating set (which we can check by seeing if all the neighbors of the current vertex are in the dominating set). If we find a vertex that doesn’t satisfy either condition, break the loop and return 0 - the given set is not dominating.

Runtime analysis of $VERIFY$: This takes at most $|V|^3k$ steps. This is because we are looping through all the vertices (V steps), and for every vertex we are at most checking every edge in the graph to see if that edge connects the vertex to each node in the dominating set (V^2k steps).

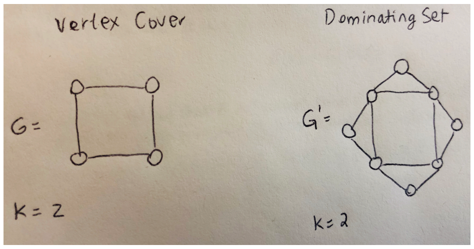
2. Reduction from $VERTEXCOVER$ to $DOMINATINGSET$:

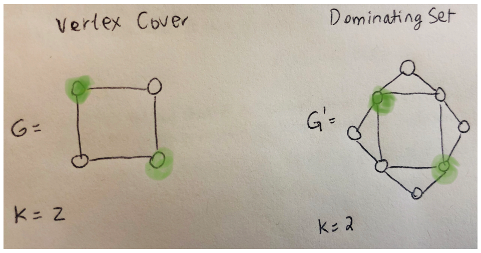
   (As an aside, how do you know to reduce from $VERTEXCOVER$, out of all the NP-hard problems? In this case, it's because both are graph-covering-esque problems - only in $VERTEXCOVER$ you're trying to select the set of vertices that covers all the edges, and in $DOMINATINGSET$ you're trying to select the set of vertices whose neighbors include all the vertices.)

   (Another aside: recall that $VERTEXCOVER(G, k) = 1$ if there exists some set of k vertices such that every edge in the graph G has at least one endpoint in the cover. In the example graphs shown above, a vertex cover would consist of at least 4 vertices).

   We want to show that if we can compute $DOMINATINGSET$, we can compute $VERTEXCOVER$ - this would prove $VERTEXCOVER$ is no harder than $DOMINATINGSET$. Our transformation R will therefore take as input a graph G and input k (the input to $VERTEXCOVER$), and output a transformed graph G' and the same number k (as the input to $DOMINATINGSET$). We construct G' as follows:

   - G' has all edges and vertices of G.
   - For every edge $\{u, v\} \in G$, we add an intermediate node on a parallel path in G'. For example, we'd keep $\{u, v\}$ intact in G' and also add vertex w and edges $\{u, w\}$ and $\{w, v\}$ in G'.





   Now, for the required verifications:

3. Runtime analysis of R: After copying the graph, we iterate over all E edges and do $O(2)$ work on each edge to construct a new vertex and two new edges. So V is $O(|E|)$ and linear time in number of edges.

4. Proof of correctness:
   We claim that G has a vertex cover of size k if and only if the transformed G' has a dominating set of the same size.

   (a) Completeness: We want to show that if G has a vertex cover of size k, G' must have a dominating set of size k. Let S be the vertex cover in G. We will show that S is a dominating set in G': every vertex is covered by S by either being part of S or by being adjacent to a vertex in S. Because S is a vertex cover, every edge in G has at least one of its endpoints in S. Now consider vertex $x \in G'$.

   If x is an original node in G, either x is in the cover or is connected by an edge to a vertex in the cover. So x is covered by some element in S.

   If v is a newly added node in G', x has two adjacent vertices $u, v \in G$ and by the above argument, at least one of those vertices is in S. So the additional nodes are also covered by S.

   So if G has a vertex cover, then G' has a dominating set of at most the same size (in fact the same set itself would do, see annotated illustration with green circles denoting the cover).

   (b) Soundness: We want to show that if G' has a dominating set of size k, G must have a vertex cover of size k. Let D be the dominating set cover in G'; we will construct the vertex cover for G.

For every vertex w we added to construct G' that is in D, w must be connected to two other vertices $u, v \in G$. In the vertex cover we can replace w with u or v - it will still allow us to cover the vertices w used to dominate.

So we can eliminate all the additional vertices as above. Since all the additional vertices correspond to one of the edges in G, and since all of the additional vertices are covered by the modified D, this means that all the edges in G are covered by the set. So if G' has a dominating set of size k, then G has a vertex cover of size at most k.

(Illustrating using our example - if the dominating set in G' consisted of vertices we added in the transformation, like the top-most and bottom-most vertices, we could replace each added vertex with one of its neighbors from the original graph, like the two green vertices we selected. Those two original vertices comprise a vertex cover on the original graph G.)

Thus, we have proven both directions of correctness:
$VERTEXCOVER(G, k) = 1 \iff DOMINATINGSET(R(G), k) = 1$

6 Intermediate SAT example: 4-SAT

Problem: Let $4SAT : \{0,1\}^* \to \{0,1\}$ be the function that takes a Boolean formula in 4-CNF (e.g. $\phi(x) = (x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_2 \vee x_3 \vee \overline{x}_4 \vee x_5))$, and outputs 1 iff ϕ is satisfiable. Show that $4SAT \in \mathsf{NP}$, and reduce it to $3SAT$ (without using the Cook-Levin theorem).

Solution:

1. Proof that $4SAT \in \mathsf{NP}$:

   We will choose the certificate w for any 4-satisfiable formula ϕ to simply be a satisfying assignment.

   The verifier program V will take as a formula ϕ with n variables and m clauses and an assignment w (w can be represented as a bit-string of length n). V will do the following: Iterate over the clauses of ϕ. Break the loop and return 0 if some clause is not satisfied by the assignment, otherwise, return 1 at the end of the loop.

   Runtime analysis of V: Iterating over all clauses takes $O(m)$ time, and checking whether a particular clause is satisfiable is $O(1)$. Hence V is $O(m)$ and thus it is linear-time with respect to the length of its inputs.

   Finally, we observe that such a w exists iff ϕ is satisfiable by definition, and V accepts a candidate assignment w iff it is a satisfying assignment.

2. Reduction from $4SAT$ to $3SAT$:

   Recall that $3SAT(\phi) = 1$ if there is a satisfying assignment for the 3-CNF formula ϕ.

   Our transformation R will take as input some 4-SAT formula ϕ (with n variables and m clauses) and output a 3-SAT formula ψ (with $n + m$ variables and $2m$ clauses) as follows:

   (a) Let $i = 1, \ldots, m$. If the i-th clause of ϕ is $a_1 \vee a_2 \vee a_3 \vee a_4$ (each of these is a *literal*, such as x_3 or $\overline{x}_7$), create the new variable z_i, and add the two clauses

   $$(a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4).$$

   Now, for the required verifications:

   (a) Runtime analysis of R: We create $O(m)$ new variables, and do $O(1)$ work for each of the m clauses. The overall runtime is $O(m)$ which is linear-time with respect to the input length of R.

   (b) Proof of correctness:

i. Completeness: Assume that $4SAT(\phi) = 1$, so that a satisfying assignment for ϕ exists. Say this assignment is $x_1 \cdots x_n$. We want to show that ψ has a corresponding satisfying assignment, so that $3SAT(\psi) = 1$. We will construct a particular example of this assignment based on $x_1 \cdots x_n$ by iterating through the clauses of ϕ. Suppose the i-th clause of ϕ is $a_1 \vee a_2 \vee a_3 \vee a_4$, where each a_i is again a literal. Let $j \in \{1, 2, 3, 4\}$ be the smallest j such that $a_j = 1$ for our satisfying assignment $x_1 \cdots x_n$. (We know such a j must exist because every clause is satisfied). Then set $z_i = 0$ if $j \in \{1, 2\}$ and $z_i = 1$ if $j \in \{3, 4\}$. Thus, the two clauses in ψ coming from clause i in ϕ will both be satisfied: If $j \in \{1, 2\}$, then the first clause is satisfied since either $a_1 = 1$ or $a_2 = 1$ and the second clause is satisfied since $z_i = 0$; and if $j \in \{3, 4\}$, the first clause is satisfied since $z_i = 1$ and the second clause is satisfied since either $a_3 = 1$ or $a_4 = 1$. Thus, all clauses of ψ are satisfied by the assignment consisting of $x_1 \cdots x_n$ as well as our chosen values $z_1 \cdots z_m$, and so $3SAT(\psi) = 1$.

ii. Soundness: This direction proceeds similarly to the above. If $3SAT(\psi) = 1$, we have some satisfying assignment $x_1 \cdots x_n z_1 \cdots z_m$ for ψ. Then we claim that $x_1 \cdots x_n$ is a satisfying assignment for ϕ, so that $4SAT(\phi) = 1$. Consider the i-th clause $a_1 \vee a_2 \vee a_3 \vee a_4$ (where each a_i is a literal) of ϕ. There are two cases: If $z_i = 0$, then the clause $a_1 \vee a_2 \vee z_i$ of ψ must be satisfied, we have $a_1 \vee a_2 = 1$; similarly, if $z_i = 1$, then the clause $\overline{z}_i \vee a_3 \vee a_4$ of ψ must be satisfied, we have $a_3 \vee a_4 = 1$. In either case, $a_1 \vee a_2 \vee a_3 \vee a_4 = 1$, and so all clauses of ϕ are satisfied, and so $4SAT(\phi) = 1$.

7 Advanced example: 3-coloring

Problem: $3COLOR : \{0, 1\}^* \to \{0, 1\}$ is the function that takes as input a graph $G = (V, E)$, and outputs 1 iff there is a 3-coloring of G. Show that $3COLOR$ is NP-complete.

Informally, a k-coloring of a graph is an assignment of a color to each vertex of the graph, using at most k colors, such that no two adjacent vertices (connected by an edge) share the same color. In our case, $k = 3$. Formally, $3COLOR(G) = 1$ if and only if there exists a mapping $C : V \to \{1, 2, 3\}$ such that for any edge $(u, v) \in E, C(u) \neq C(v)$.

Solution:

1. Proof that $3COLOR \in$ NP:

   We will choose the certificate w for any 3-colorable input G to be the mapping function C. The verifier program $VERIFY$ (re-named to avoid confusion with V, the set of vertices) will take as inputs G and C (where C can be represented as, for example, an array of length $|V|$, such that $C[v]$ represents the color of vertex v). $VERIFY$ will do the following: Iterate over each $(u, v) \in E$, and check if $C[u] = C[v]$. Break the loop and return 0 if $C[u] = C[v]$ for some (u, v), otherwise, return 1 at the end of the loop.

   Runtime analysis of $VERIFY$: Iterating over all edges takes $O(|E|)$ time, and checking $c[u]$ and $c[v]$ is $O(1)$. Hence $VERIFY$ is $O(|E|)$ and thus it is linear-time with respect to the length of its inputs.

   Finally, we observe that such a C exists iff G is 3-colorable by definition, and $VERIFY$ accepts a candidate C iff it is a valid coloring.

2. Proof that $3COLOR$ is NP-hard:

   We will reduce $3SAT$ to $3COLOR$. Recall that $3SAT(\phi) = 1$ if there is a satisfying assignment for the 3-CNF formula ϕ.

   Our transformation R will take as input some formula ϕ (with N variables and M clauses) and output a graph G as follows:

   (a) We first create 3 vertices, named $TRUE$, $FALSE$, and $NULL$, and connect them all to each other (forming a clique). We call this the *original clique*.

(b) For each variable $x_i \in \phi$, we create two vertices named v_{2i} and v_{2i+1}, respectively, representing x_i and $\overline{x_i}$.

(c) We create add edges to create cliques between v_{2i}, v_{2i+1} and $NULL$. So we've created N more cliques, which we call the *variable cliques.*

(d) For each clause in ϕ, we add a subgraph called an OR-*gadget* (refer to Figure 1). An OR-gadget contains 6 vertices that are connected to the 3 variables of the clause, e.g. if the clause is $x_0 \vee x_1 \vee \overline{x_2}$ then the OR-gadget is connected to v_0, v_2 and v_5. One of the 6 vertices is the output vertex, which also forms a clique with $FALSE$ and $NULL$. The OR-gadget basically simulates an OR-gate of 3 inputs. Refer to Figure 1 for an example of what G will look like.

Now, for the required verifications:

(a) Runtime analysis of R: Creating $TRUE$, $FALSE$, and $NULL$ is clearly $O(1)$. Creating a clique for each variable of ϕ is $O(N)$. Creating an OR-gadget for each clause is $O(M)$. The overall runtime is $O(N + M)$ which is linear-time with respect to the input length of R.

(b) Proof of correctness:

i. Completeness: Suppose we have some ϕ such that $3SAT(\phi) = 1$, i.e. there is a valid assignment for ϕ. Consider a particular valid assignment. Then consider the following 3-coloring of G: color the $TRUE$ vertex green, as well as v_{2i} for all x_i in ϕ whose valid assignment is 1. Color the $FALSE$ vertex red, as well as v_{2i+1} for all x_i in ϕ whose valid assignment is 0. Color the $NULL$ vertex grey.

It is clear that all N variable cliques as well as the original clique are colored in a valid way, since either $x_i = 1$ or $x_i = 0$ for any valid assignment, and so one of v_{2i} and v_{2i+1} must be green and the other, red. Now let's look at the OR-gadgets. Since there is a valid assignment that satisfies the clause, one of the 3 variable vertices must be colored green. It is clear that there is a way of coloring the OR-gadget such that the output node is green (and hence forms a valid coloring with $NULL$ and $FALSE$). Hence there is a valid 3-coloring of G and so $3COLOR(G) = 1$.

ii. Soundness: Now, assume that $3COLOR(G) = 1$, i.e. a 3-coloring of G exists. We want to show that the 3-CNF formula ϕ corresponding to G has a valid assignment. Consider a particular 3-coloring of G. WLOG let the $TRUE$ vertex be green, the $FALSE$ vertex be red, and the $NULL$ vertex be grey. (They must all be different for the original clique to be validly colored.) Now, since all variable cliques are also validly colored, then for each i, one of v_{2i} and v_{2i+1} must be red and the other must be green. Our claim now is that the coloring of the variable vertices is a valid assignment for ϕ. First, it is clear that each x_i is well-defined and must be either 1 or 0. Next, we need to show that one of the 3 variable vertices connected to any OR-gadget must be green. We know that the output vertex of the OR-gadget is green. It is possible to show by contradiction that one of the 3 variable vertices must be green. Hence the coloring of G is a valid assignment for ϕ and hence $3SAT(\phi) = 1$.

Note: This reduction is subtle, in the sense that coming up with something like the OR-gadget takes time, and it is not something we would expect you to do in an exam setting. We also understand that it is not immediately intuitive how the colors of G should correspond to the clauses/variables of ϕ. We intentionally chose a non-trivial reduction because we hope that going through this example can help you understand the general structure of reductions, as well as for you to go back and make sure you understand what (if any) were the issues in your problem set solutions.

8 Miscellaneous tips

1. Always prove the $F \in \mathsf{NP}$ part before the actual reduction. These two parts are independent — you can still prove that $F \in \mathsf{NP}$ even if you can't immediately figure out the reduction. In fact, this part is (almost always) significantly easier and might help you understand F better.

2. Be very clear about what the direction of the reduction is, before you start thinking about the reduction. You are reducing *from* a *known* NP-hard problem H *to* the function of interest F.

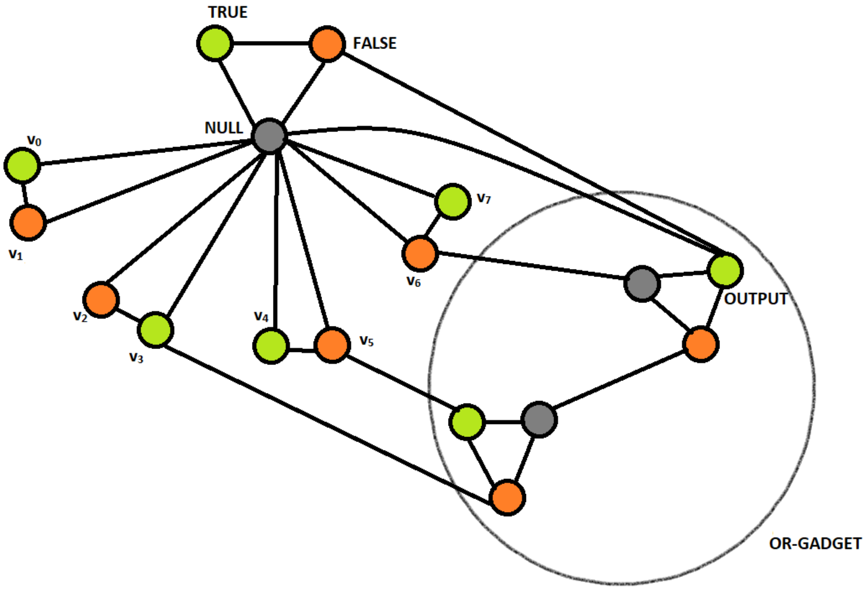


Figure 1: An example of the OR-gadget, here applied to the example clause $x_1 \vee \overline{x_2} \vee \overline{x_3}$. Recall that a literal is green iff it is 1 in the corresponding assignment (e.g. since v_7 is green, then $\not{x}_3 = 1$). Note that the vertex labeled $OUTPUT$ must be green since it is placed in a triangle with $FALSE$ and $NULL$. Observe that if v_3, v_5, and v_6 were all red, we could not 3-color the vertices of the OR-gadget (in particular, the endpoints of the edge connecting the triangles would both have to be red); conversely, if at least one of v_3, v_5, or v_6 is green, we can always 3-color the vertices of the OR-gadget.

9 Helpful resources/additional practice

1. Chapter 13 from Boaz's textbook - contains reductions from 3SAT to ISET (Section 13.4), ISET to MAXCUT (Section 13.5), 3SAT to LONGESTPATH (Section 13.6)

2. For further practice on graph-based reductions: From UMD's CS Class

3. For some simple reductions, see Section 7 Notes, which includes a worked-through reduction from the book, as well as several simple reductions in the practice problems (Questions 2 and 4)

4. Really helpful video showing 3SAT to 3COLOR Reduction

5. Chapter 7 of Sipser's book contains many solved examples

6. The teaching staff - we are always here for you! Please let us know what concepts confuse you and we will do our best to provide more resources and explain it.

10 Acknowledgements

The sections on reduction formats and 3-coloring were written by Boaz Barak, Wanqian Yang, and Brian Sapozhnikov in Fall 2018. The rest was written and edited by Alex Chin, Noah Singer, and Michele Wang in Fall 2019.