Outline

1 Review

In this recitation, we'll be talking about how to prove that a problem is NP-hard. In order to do so, it's a good idea to first review some definitions. The definitions of NP and NP-hardness can be very tricky to get at first, so it's a good thing to make sure that we're all on the same page.

The first thing to take care of is actually even more basic: how do we define a problem? Is it very general or very specific? For instance, 3-SAT is a problem. But there's a number of related problems, such as the problem of whether the boolean formula $(x_1 \vee \overline{x_2}) \wedge (x_2 \vee \overline{x_1})$ can be satisfied. For the purposes of this recitation, we'll be calling 3-SAT a problem, and boolean formulae like the one above are called *problem instances*.

1.1 About NP

With that out of the way, we can review the definition of NP. All problems in NP are *decision problems*, not search problems: the answer to a particular instance of the problem is always either TRUE or FALSE. However, for any decision problem there's usually a related search problem. And if you're trying to show that a search problem is NP-hard, it's sufficient to show that the related decision problem is NP-hard.

About NP.
- *Decision problems*: solution to every problem instance is TRUE or FALSE.
- Formally, there exists a poly-time algorithm A such that the answer to instance x is TRUE iff there exists a poly-size *witness* y with $A(x, y) =$ TRUE.

Informally, a problem is in NP if and only if it has a polynomial-time verifier. In other words, it is in NP if you can check whether the answer is correct in polynomial time. However, there's a bit of a complication here. The answer to a decision problem is either TRUE or FALSE. So "checking" an answer seems like it would consist of taking as input the problem instance x and the answer TRUE, and verifying that that was the correct answer. But it seems like if you could do that in polynomial-time, you ought to be able to solve x in polynomial-time. (Just try running the verifier with x and TRUE as inputs.)

That's where the idea of witnesses comes in. A witness y for a problem instance x is basically secret information that lets you verify that the answer to x is TRUE. It's usually the answer to the related search problem. For example, consider the question of whether the boolean formula $(x_1 \vee \overline{x_2}) \wedge (x_2 \vee \overline{x_1})$ can be satisfied. This is clearly an instance of the 3-SAT problem. The related search problem asks for a particular assignment that satisfies that formula. So the witness for the satisfiability of that problem would be an assignment such as $x_1 =$ TRUE, $x_2 =$ TRUE.

There are a couple of things to note. First of all, the size of the witness y should be polynomial in the size of the problem instance x. Second, if a problem is in NP, then there are witnesses that attest to a particular problem instance being TRUE, but there might not be any witnesses that attest to a particular problem instance being FALSE. The decision problem is usually defined as, "Is there an answer to this related search problem?" So finding an answer to the related search problem is clearly a witness to the decision problem. But if there is no answer, there might not be an efficient way to check.

1.2 About NP-Hardness

Now that we've reviewed the definition of NP, we can move on to the definition of NP-hardness. A problem is NP-hard if all problems in NP can be reduced to it in polynomial time. So if any NP-hard problem can be solved in polynomial time, then any problem in NP can also be solved in polynomial time. Unfortunately, this means that it seems rather difficult to show that anything is NP-hard. (Having to show a reduction from every problem in NP from scratch every time seems like it would be very difficult.) Luckily, there's a shortcut: if you know that a problem A is NP-hard, then you can show that a problem B is NP-hard by giving a polynomial-time reduction from any instance of problem A to an instance of problem B.

NP-Hardness and NP-Completeness.
- A problem is NP-hard if all problems in NP can be reduced to it.
- To show NP-hardness, sufficient to reduce a known NP-hard problem to it.
- A problem is NP-complete if it is NP-hard and in NP.

2 Proving NP-Hardness

This brings us to the main topic of this particular recitation: how to show that a problem is NP-hard. We'll be approaching this from an algorithmic design perspective: not just giving an example of a proof of NP-hardness, but also showing how to come up with such a proof.

The technique that we'll be using boils down to the four steps listed to the right. We'll be using those four steps in an attempt to prove that the SUBSET-SUM problem is NP-hard.

Recipe for NP-Hardness.
1. Pick a problem to reduce from.
2. Relate the witnesses.
3. Add constraints.
4. Formalize.

2.1 Pick a Problem

The very first step in showing NP-hardness is to pick a problem to reduce from. Technically, there exists a polynomial-time reduction

from every NP-complete problem to every NP-hard problem, so we could pick pretty much any problem. But some problems are a lot easier than others. It's usually a good idea to pick a problem that is in some way related.

To the right there is a list of the problems that are most commonly used to show NP-hardness. Some of these you may already know. However, for the sake of completeness, we're going to repeat the definition of each.

Known NP-Complete Problems.
- 3-SAT
- SUBSET-SUM
- VERTEX-COVER
- CLIQUE
- GRAPH-COLORING

Definition 1. *The* 3-SAT *problem is defined as follows:*

- ***Input:*** *A boolean formula in conjunctive normal form such that there are at most three literals in each clause.*
- ***Output:*** TRUE *if and only if the formula is satisfiable.*

Definition 2. *The* SUBSET-SUM *problem is defined as follows:*

- ***Input:*** *A set $S = \{s_1, \ldots, s_n\}$ of non-negative integers and a target value t.*
- ***Output:*** TRUE *if and only if there exists a subset of S that sums to t.*

Definition 3. *The* VERTEX-COVER *problem is defined as follows:*

- ***Input:*** *A graph $G = (V, E)$ and a non-negative integer k.*
- ***Output:*** TRUE *if and only if there exists a subset S of vertices such that $|S| = k$ and all edges are incident to at least one vertex in S.*

Definition 4. *The* CLIQUE *problem is defined as follows:*

- ***Input:*** *A graph $G = (V, E)$ and a non-negative integer k.*
- ***Output:*** TRUE *if and only if there exists a subset S of vertices such that $|S| = k$ and the subgraph of G induced by S is complete.*

Definition 5. *The* GRAPH-COLORING *problem is defined as follows:*

- ***Input:*** *A graph $G = (V, E)$ and a number of colors $k \geq 3$.*
- ***Output:*** TRUE *if and only if there exists a way to assign each vertex a color such that no edge connects two vertices of the same color.*

These problems fall into three categories: booleans, numbers, and graphs. These categories can be useful in picking which problem to reduce from. If we're trying to show that some problem related to booleans is NP-hard, then it might make sense to reduce from 3-SAT. If we're trying to show that some problem related to numbers is NP-hard, then it might make sense to reduce from SUBSET-SUM. If we're

trying to show that some problem related to graphs is NP-hard, then it might make sense to reduce from whichever of Vertex-Cover, Clique, or Graph-Coloring seems more closely related.

In this recitation, we're trying to show that Subset-Sum is NP-hard. That's a number-related problem, but unfortunately we can't show that Subset-Sum is NP-hard by reducing from Subset-Sum. Instead, we're going to do it by reduction from 3-Sat. Note that 3-Sat can often be a sort of catch-all — even when the problem doesn't seem to be related to booleans, the reduction from 3-Sat can be quite straightforward in some cases.

2.2 Relate the Witnesses

Now that we've picked a problem, it's time to start constructing our reduction. Due to some complications with the definition of NP, we cannot use general Cook reductions to show that a problem is NP-hard. Instead, we must use Karp reductions.

Recall that a Karp reduction from a problem A to a problem B is a function f mapping instances of problem A to instances of problem B such that for any instance x of A, the answer to x is True if and only if the answer to $f(x)$ is True. If A and B are problems in NP, then we can use the definition of NP to rewrite the definition of a Karp reduction as follows.

Let V_A be the poly-time verifier for A, and let V_B be the poly-time verifier for B. Then f is a Karp reduction from A to B if and only if for all instances x of A, there is a poly-size witness y for the truth of x if and only if there is a poly-size witness z for the truth of $f(x)$. More formally:

$$\begin{array}{c}\text{There exists a poly-size}\\ \text{witness } y \text{ such that}\\ V_A(x,y) = \textsc{True}.\end{array} \iff \begin{array}{c}\text{There exists a poly-size}\\ \text{witness } z \text{ such that}\\ V_B(f(x),z) = \textsc{True}.\end{array}$$

What does this mean? Well, it means that if a witness exists for x, then there must be a witness for $f(x)$, and vice versa. Logically, it seems like maybe these witnesses should be connected somehow. So that is our first step in building a reduction: find a mapping between the witnesses that seems like it might be useful.

Let's discuss what properties might be useful. The problem instance x can be any instance of problem A, so we can't place any constraints on the witness y. The problem instance $f(x)$, on the other hand, is strictly determined by our reduction, and so we may be able to place constraints on the witness z.

This mapping between witnesses should be two-way. We want to be able to construct the appropriate z given any y, and we want to be able to extract the value of y given the appropriate z. This is an encoding of sorts: we want all of the information in y to show up somewhere in z preferably in a way that's pretty easy to extract.

Reduction Form.

- For NP-hardness, must use Karp.
- A Karp reduction from Problem A to Problem B is a function f mapping instances of A to instances of B such that:

$$\begin{array}{c}\text{answer to } x\\ \text{is True}\end{array} \Leftrightarrow \begin{array}{c}\text{answer to } f(x)\\ \text{is True}\end{array}$$

- If A and B are problems in NP with V_A the verifier for A and V_B the verifier for B, equivalent to:

$$\begin{array}{c}\text{exists witness } y:\\ V_A(x,y) = \textsc{True}\end{array} \Leftrightarrow \begin{array}{c}\text{exists witness } z:\\ V_B(f(x),z) = \textsc{True}\end{array}$$

It can be hard to understand what this process of relating witnesses looks like without an example. So let's return to our reduction from 3-SAT to SUBSET-SUM. A witness for 3-SAT assigns each variable a value of TRUE or FALSE. A witness for SUBSET-SUM is a subset of numbers. So if we want to encode a variable assignment in a subset, it makes sense to create for each variable in our 3-SAT problem, a number in our SUBSET-SUM problem that will be included in the subset if and only if the variable is TRUE.

Witnesses Example.
- 3-SAT Witness: Satisfying assignment of TRUE or FALSE to all variables.
- SUBSET-SUM Witness: Subset of numbers that sums to the target.

Does this idea work? Well, let's do an example. Suppose that we start with the 3-SAT instance $(x_1 \vee \overline{x_2}) \wedge (x_2 \vee \overline{x_1})$. There are two variables x_1 and x_2, so we would create two numbers in our SUBSET-SUM instance: s_1 and s_2. The assignment $x_1 =$ TRUE, $x_2 =$ TRUE is a witness for 3-SAT, so the corresponding SUBSET-SUM witness would be the set $\{s_1, s_2\}$. To make this work properly, we'd probably want to set $t = s_1 + s_2$.

First Idea.
- For each variable in the 3-SAT instance, make a corresponding number in the SUBSET-SUM instance.
- Variable is TRUE iff corresponding number is included in SUBSET-SUM witness.

But this doesn't quite work properly. There's another witness for the boolean formula above: $x_1 =$ FALSE and $x_2 =$ FALSE. This would correspond to the empty subset. So to make this work properly, we'd probably want to set $t = 0$. Combined with the constraints on non-negativity, this means that $s_1 = s_2 = 0$. This seems wrong.

How can we fix this? Well, the problem seems to be that when we set a variable to false, nothing shows up in the corresponding subset. So intuitively, it seems like the target value we're looking for should be roughly proportional to the number of variables that need to be true to create a satisfying assignment. (If we assume that all of the numbers are roughly the same size.) But we don't know how many variables need to be true to create a satisfying assignment.

So let's take a different approach. Let's make each variable x_i have *two* numbers s_{iT} and s_{iF} in the corresponding SUBSET-SUM instance. The number s_{iT} should be included in the subset if and only if x_i is set to TRUE. The number s_{iF} should be included in the subset if and only if x_i is set to FALSE. We'll have to figure out how to enforce those constraints later, but in the meantime this seems to fix our problem. Now the witness $x_1 =$ TRUE, $x_2 =$ FALSE corresponds to the subset $\{s_{1T}, s_{2T}\}$, while the witness $x_1 =$ FALSE, $x_2 =$ FALSE corresponds to the subset $\{s_{1F}, s_{2F}\}$. And it's not out of the realm of possibility to have $s_{1T} + s_{2T} = t = s_{1F} + s_{2F}$.

Second Idea.
- For each variable x_i in the 3-SAT instance, make two corresponding numbers s_{iT} and s_{iF} in the SUBSET-SUM instance.
- $x_i =$ TRUE $\Leftrightarrow$ s_{iT} included in subset
- $x_i =$ FALSE $\Leftrightarrow$ s_{iF} included in subset

2.3 Add Constraints

We have made a decision about how we want the witnesses y and z to relate to each other. Now it's time to make sure that y is a witness for x only when z is a witness for $f(x)$, and vice versa. This step usually requires rephrasing the constraints of Problem A (the problem we're reducing from) in terms of the constraints of Problem B (the problem we're reducing to). Essentially, there are certain constraints that determine whether or not a given string y is a witness for the problem instance x. We have to somehow simulate

Add Constraints.
- Figure out constraints imposed on witness y by Problem A.
- Figure out constraints imposed on witness z by Problem B.
- Want to simulate constraints of Problem A using constraints from Problem B.

those constraints using the constraints that determine whether or not a given string z is a witness for the problem instance $f(x)$.

Unfortunately, this step is by far the most difficult to understand in its full generality. So let's see how it applies to the problem that we've been working on.

Let's try to understand the types of constraints in 3-SAT and in SUBSET-SUM. Any witness for 3-SAT has the following properties:

1. Variables are assigned to be either TRUE or FALSE. No variable can be given more than one value. All variables must be given a value.

2. In each clause of the boolean formula, at least one of the literals must be true.

The constraints for SUBSET-SUM are:

1. The numbers in the witness must all come from S, the set of numbers in the SUBSET-SUM instance. No number can be used more than once.

2. All of the numbers in the witness must sum to t.

At first, this seems like a rather tough task. The constraints on 3-SAT are myriad: one independent constraint for each clause. The constraints on SUBSET-SUM are comparatively simple: the witness must be a subset, and it must sum to the correct value.

Let's start by figuring out how to simulate the first 3-SAT constraint: that all variables must be assigned to either TRUE or FALSE, but not both. We have decided that for a variable x_i, the number s_{iT} should be included in the SUBSET-SUM witness if and only if $x_i =$ TRUE in the corresponding 3-SAT witness. Similarly, the number s_{iF} should be included in the SUBSET-SUM witness if and only if $x_i =$ FALSE in the corresponding 3-SAT witness. So the equivalent of making sure that x_i is assigned to either TRUE or FALSE but not both would be to make sure that either s_{iT} or s_{iF} shows up in the witness z, but not both.

To tackle this problem, let's look at a smaller case. Suppose we only have one variable x_1. Then there are two numbers in the SUBSET-SUM problem: s_{1T} and s_{1F}. We want to make sure that exactly one of them is selected for the subset. If we don't have to satisfy any other constraints, then it's sufficient to set $s_{1T} = s_{1F} = t$. That way, we pick exactly one for the subset.

We may generalize this as depicted at right: each variable x_i has a unique power of 2 associated with it. If s_{1T} and s_{1F} are both included in some subset, then the sum of the subset must be even. But t is odd, so exactly one of s_{1T} and s_{1F} must be included in the subset. By induction, we can show that for each variable x_i, exactly one of s_{iT} and s_{iF} must be included to make the sum work out properly.

Understanding Constraints.

- Constraints on 3-SAT witnesses:
  1. all variables TRUE or FALSE
  2. cannot be both
  3. all clauses contain a true literal
- Constraints on SUBSET-SUM witnesses:
  1. must be a subset
  2. must sum to t

Variables Not Both True and False.

- **Simple Example:** Only one variable, x_1.
  - Want the subset summing to t to contain exactly one of s_{1T} and s_{1F}.
  - **Idea:** Make $s_{1T}, s_{1F} = t$.
- **General Case:** Extend to more variables.
  - **Idea:** For each variable x_i, set $s_{iT} = s_{iF} = 2^{i-1}$. Then set $t = 2^n - 1$.
  - Written out in binary:

$s_{1T} =$	0	0	0	1
$s_{1F} =$	0	0	0	1
$s_{2T} =$	0	0	1	0
$s_{2F} =$	0	0	1	0
$s_{3T} =$	0	1	0	0
$s_{3F} =$	0	1	0	0
$s_{4T} =$	1	0	0	0
$s_{4F} =$	1	0	0	0
$t =$	1	1	1	1

  - **Realization:** Can use different bits in numbers to impose different constraints.

This construction won't completely work, in practice. (All numbers in the SUBSET-SUM problem must be unique.) But it gives us some intuition on how to construct constraints: if we can avoid or limit carries, then different bits of the number can be used to impose different constraints. We can use this idea in an attempt to impose all of the 3-SAT constraints.

The other kind of 3-SAT constraints we need to simulate are the clause constraints. For each clause, there are three assignments of the form x_i = TRUE or x_i = FALSE that could cause the clause to be satisfied. At least one of those assignments must be included in the set of assignments used to satisfy the whole formula. We can rephrase this in terms of the SUBSET-SUM instance: for each clause, there are three numbers that, when included in the subset, correspond to satisfying the clause. We want to add constraints to our SUBSET-SUM instance to ensure that at least one of these three numbers is included in the subset.

At right, we depict one way to do this. Each clause gets several bits associated with it, and the numbers associated with satisfying that clause have a value of 1 in the bits associated with that clause. But what should the target value be? Well, the number of assignments that satisfy the clause should be at least one, but it could be as high as three. So we need some way to make sure that no matter how many true literals the clause contains for a given satisfying assignment, the sum works out properly. To do so, we add extra values to fill the gap for each clause. If the extra values sum to at most 3 within the bits devoted to that clause, then by setting the target value to 4 within those bits, we can ensure that each clause must contain at least one true literal.

Clause Constraints.

- Assign each clause several bits.
- For each variable assignment x_i = TRUE or x_i = FALSE, modify the variable s_{iT} or s_{iF} to have a 1 in all of the clauses that the assignment would satisfy.
- **Example:** $(x_1 \vee \overline{x_2}) \wedge (x_2 \vee \overline{x_1}) \wedge (x_1 \vee x_2)$
  - Clause $c_1 = (x_1 \vee \overline{x_2})$.
  - Clause $c_2 = (x_2 \vee \overline{x_1})$.
  - Clause $c_3 = (x_1 \vee x_2)$.

	x_2	x_1	c_3	c_2	c_1
$s_{1T} =$	000	001	001	000	001
$s_{1F} =$	000	001	000	001	000
$s_{2T} =$	001	000	001	001	000
$s_{2F} =$	001	000	000	000	001
$t =$	001	001	???	???	???

2.4 Formalize

Now that we've figured out most of the details, it's time to formalize the reduction. Suppose that we're given an instance of 3-SAT with n variables $x_1, \ldots, x_n$ and m clauses $c_1, \ldots, c_m$. Then we construct the numbers in our SUBSET-SUM instance as follows:

- For each variable x_i, let $C_{iT} = \{j \mid x_i \in c_j\}$ be the indices of the clauses that contain the literal x_i. Then define the number s_{iT} as follows:

$$s_{iT} = 1 \cdot 2^{3(i+m-1)} + \sum_{j \in C_{iT}} 1 \cdot 2^{3(j-1)}$$

- For each variable x_i, let $C_{iF} = \{j \mid \overline{x_i} \in c_j\}$ be the indices of the clauses that contain the literal $\overline{x_i}$. Then define the number s_{iF} as follows:

$$s_{iF} = 1 \cdot 2^{3(i+m-1)} + \sum_{j \in C_{iF}} 1 \cdot 2^{3(j-1)}$$

- For each clause c_j, define two numbers:

$$r_{j1} = 1 \cdot 2^{3(j-1)}$$
$$r_{j2} = 2 \cdot 2^{3(j-1)}$$

We then define the target sum t to be:

$$t = \sum_{i=1}^{n} 1 \cdot 2^{3(i+m-1)} + \sum_{j=1}^{m} 4 \cdot 2^{3(j-1)}$$

All that remains now is to prove that this construction is correct. We may do so by arguing about the witnesses: that a witness for the original 3-SAT instance can be transformed into a witness for the corresponding SUBSET-SUM instance, and vice versa.

3 Strong and Weak NP-Hardness

One thing to note about the above reduction is the magnitude of the numbers that we use. Note that the magnitude of the numbers is exponential in m and n. If those numbers are written in binary, this is not a problem: it takes $O(m+n)$ time to write down a number with $O(m+n)$ bits. So this reduction requires polynomial time if the numbers are written in binary.

Suppose that we wanted to write those numbers in unary. Writing a number in unary means writing down a sequence of symbols whose length is equal to the magnitude of the number. This would require $O(2^{O(m+n)})$ time. So the reduction would stop being polynomial-time if the numbers were written in unary.

Consider a variant of the SUBSET-SUM problem where all input numbers are written in unary. Call this new problem SUBSET-SUM-UNARY. This problem can actually be solved in time polynomial in the size of the input, using a simple dynamic programming algorithm. As a result, the SUBSET-SUM problem is what's known as *weakly NP-hard* — if the numbers are written in unary, it's not NP-hard at all. By contrast, a problem is *strongly NP-hard* if the problem is hard even when the numbers are written in unary.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012