Learning Objectives:

- Introduce the class **NP** capturing a great many important computational problems
- **NP**-completness: evidence that a problem might be intractable.
- The **P** vs **NP** problem.

16

NP, NP completeness, and the Cook-Levin Theorem

> *"In this paper we give theorems that suggest, but do not imply, that these problems, as well as many others, will remain intractable perpetually"*, Richard Karp, 1972

> *"It is not the verifier who counts; not the man who points out how the solver of problems stumbles, or where the doer of deeds could have done them better. The credit belongs to the man who actually searches over the exponential space of possiblities, whose face is marred by dust and sweat and blood; who strives valiantly; who errs, who comes short again and again ... who at the best knows in the end the triumph of high achievement, and who at the worst, if he fails, at least fails while daring greatly, so that his place shall never be with those cold and timid souls who neither know victory nor defeat."*, paraphrasing Theodore Roosevelt (1910).

16.1 The class **NP**

So far we have shown that 3SAT is no harder than Quadratic Equations, Independent Set, Maximum Cut, and Longest Path. But to show that these problems are *computationally equivalent* we need to give reductions in the other direction, reducing each one of these problems to 3SAT as well. It turns out we can reduce all three problems to 3SAT in one fell swoop.

In fact, this result extends far beyond these particular problems. All of the problems we discussed in the previous lecture, and a great many other problems, share the same commonality: they are all

search problems, where the goal is to decide, given an instance x, whether there exists a *solution* y that satisfies some condition that can be verified in polynomial time. For example, in 3SAT, the instance is a formula and the solution is an assignment to the variable; in Max-Cut the instance is a graph and the solution is a cut in the graph; and so on and so forth. It turns out that *every* such search problem can be reduced to 3SAT.

To make this precise, we make the following mathematical definition: we define the class **NP** to contain all Boolean functions that correspond to a *search problem* of the form above— that is, functions that output 1 on x if and only if there exists a solution w such that the pair (x, w) satisfies some polynomial-time checkable condition. Formally, **NP** is defined as follows:

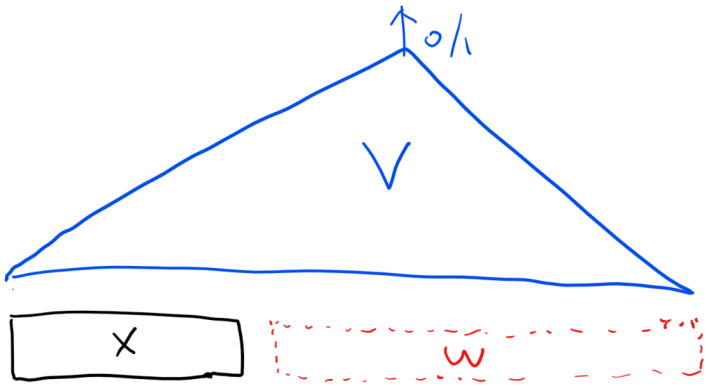

Figure 16.1: The class **NP** corresponds to problems where solutions can be *efficiently verified*. That is, this is the class of functions F such that $F(x) = 1$ if there is a "solution" w of length polynomial in $|x|$ that can be verified by a polynomial-time algorithm V.

Definition 16.1 — NP. We say that $F : \{0,1\}^* \to \{0,1\}$ is in **NP** if there exists some constants $a, b \in \mathbb{N}$ and $V : \{0,1\}^* \to \{0,1\}$ such that $V \in \mathbf{P}$ and for every $x \in \{0,1\}^n$

$$F(x) = 1 \Leftrightarrow \exists_{w \in \{0,1\}^{an^b}} \text{ s.t. } V(xw) = 1 \tag{16.1}$$

See also Fig. 16.1 for an illustration of Definition 16.1. The name **NP** stands for "nondeterministic polynomial time" and is used for historical reasons; see the bibiographical notes. The string w in Eq. (16.1) is sometimes known as a *solution*, *certificate*, or *witness* for the instance x.

R **NP and proof systems** The definition of **NP** means that for every $F \in \mathbf{NP}$ and string $x \in \{0,1\}^*$,

$F(x) = 1$ if and only if there is a *short and efficiently verifiable proof* of this fact. That is, we can think of the function G in Definition 16.1 as a *verifier* algorithm, similar to what we've seen in Definition 12.7. The verifier checks whether a given string $w \in \{0,1\}^*$ is a valid proof for the statement "$F(x) = 1$". Essentially all proof systems considered in mathematics involve line-by-line checks that can be carried out in polynomial time. Thus the heart of **NP** is asking for statements that have *short* (i.e., polynomial in the size of the statements) proof. As we will see later on, for this reason Kurt Gödel phrased the question of whether **NP** = **P** as asking whether "the mental work of a mathematician [in proving theorems] could be completely replaced by a machine".

The Definition 16.1 is *assymetric* in the sense that there is a difference between an output of 1 and an output of 0. You should make sure you understand why this definition does *not* guarantee that if $F \in$ **NP** then the function $1 - F$ is in **NP** as well. In fact, this is believed *not* to be the case in general. This is in contrast to the class **P** which (as you should verify) *does* satisfy that if $F \in$ **P** then $1 - F$ is in **P** as well.

16.1.1 Examples:

- $3SAT$ is in **NP** since for every ℓ-variable formula φ, $3SAT(\varphi) = 1$ if and only if there exists a satisfying assignment $x \in \{0,1\}^\ell$ such that $\varphi(x) = 1$, and we can check this condition in polynomial time.[1]
- $QUADEQ$ is in **NP** since for every ℓ-variable instance of quadratic equations E, $QUADEQ(E) = 1$ if and only if there exists an assignment $x \in \{0,1\}^\ell$ that satisfies E, and we can check this condition in polynomial time.
- $ISET$ is in **NP** since for every graph G and integer k, $ISET(G,k) = 1$ if and only if there exists a set S of k vertices that contains no pair of neighbors in G, and we can check this condition in polynomial time.
- $LONGPATH$ is in **NP** since for every graph G and integer k, $LONGPATH(G,k) = 1$ if and only if there exists a simple path P in G that is of length at least k, and we can check this condition in polynomial time.

[1] Note that an ℓ variable formula φ is represented by a string of length at least ℓ, and we can use some "padding" in our encoding so that the assignment to φ's variables is encoded by a string of length exactly $|\varphi|$. We can always use this padding trick, and so one can think of the condition Eq. (16.1) as simply stipulating that the "solution" y to the problem x is of size at most $poly(|x|)$.

- $MAXCUT$ is in **NP** since for every graph G and integer k, $MAXCUT(G,k) = 1$ if and only if there exists a cut $(S, \overline{S})$ in G that cuts at least k edges, and we can check this condition in polynomial time.

16.1.2 *From* NP *to* 3SAT

There are many, many, *many*, more examples of interesting functions we would like to compute that are easily shown to be in **NP**. What is quite amazing is that if we can solve 3SAT then we can solve all of them!

The following is one of the most fundamental theorems in Computer Science:

Theorem 16.2 — Cook-Levin Theorem. For every $F \in$ **NP**, $F \leq_p 3SAT$.

We will soon show the proof of Theorem 16.2, but note that it immediately implies that $QUADEQ$, $LONGPATH$, and $MAXCUT$ all reduce to $3SAT$. In fact, combining it with the reductions we've seen, it implies that all these problems are *equivalent!* For example, to reduce $QUADEQ$ to $LONGPATH$, we can first reduce $QUADEQ$ to $3SAT$ using Theorem 16.2 and use the reduction we've seen from $3SAT$ to $LONGPATH$. There is of course nothing special about $QUADEQ$ here— by combining Theorem 16.2 with the reduction we saw, we see that just like $3SAT$, *every* $F \in$ **NP** reduces to $LONGPATH$, and the same is true for $QUADEQ$ and $MAXCUT$. All these problems are in some sense "the hardest in **NP**" since an efficient algorithm for any one of them would imply an efficient algorithm for *all* the problems in **NP**. This motivates the following definition:

Definition 16.3 — NP-hardness and NP-completeness. We say that $G : \{0,1\}^* \to \{0,1\}$ is **NP** *hard* if for every $F \in$ **NP**, $F \leq_p G$.

We say that $G : \{0,1\}^* \to \{0,1\}$ is **NP** *complete* if G is **NP** hard and G is in **NP**.

Theorem 16.2 and the reductions we've seen in the last lecture show that despite their superficial differences, 3SAT, quadratic equations, longest path, independent set, and maximum cut, are all **NP**-complete. Many thousands of additional problems have been shown to be **NP**-complete, arising from all the sciences, mathematics, economics, engineering and many other fields.[2]

[2] For some partial lists, see this Wikipedia page and this website.

16.1.3 What does this mean?

Clearly **NP** $\supseteq$ **P**, since if we can decide efficiently whether $F(x) = 1$, we can simply ignore any "solution" that we are presented with. (However, it is still an excellent idea for you to pause here and verify that you see why every $F \in$ **P** will be in **NP** as per Definition 16.1.) Also, **NP** $\subseteq$ **EXP**, since all the problems in **NP** can be solved in exponential time by enumerating all the possible solutions. (Again, please verify that you understand why this follows from the definition.)

The most famous conjecture in Computer Science is that **P** $\neq$ **NP**. One way to refute this conjecture is to give a polynomial-time algorithm for even a single one of the **NP**-complete problems such as 3SAT, Max Cut, or the thousands of others that have been studied in all fields of human endeavors. The fact that these problems have been studied by so many people, and yet not a single polynomial-time algorithm for any of them has been found, supports that conjecture that indeed **P** $\neq$ **NP**. In fact, for many of these problems (including all the ones we mentioned above), we don't even know of a $2^{o(n)}$-time algorithm! However, to the frustration of computer scientists, we have not yet been able to prove that **P** $\neq$ **NP** or even rule out the existence of an $O(n)$-time algorithm for 3SAT. Resolving whether or not **P** $=$ **NP** is known as the **P vs NP problem**. A million-dollar prize has been offered for the solution of this problem, a popular book has been written, and every year a new paper comes out claiming a proof of **P** $=$ **NP** or **P** $\neq$ **NP**, only to wither under scrutiny.[3] The following 120 page survey of Aaronson, as well as chapter 3 in Wigderson's upcoming book are excellent sources for summarizing what is known about this problem.

One of the mysteries of computation is that people have observed a certain empirical "zero-one law" or "dichotomy" in the computational complexity of natural problems, in the sense that many natural problems are either in **P** (often in $TIME(O(n))$ or $TIME(O(n^2))$), or they are are **NP** hard. This is related to the fact that for most natural problems, the best known algorithm is either exponential or polynomial, with not too many examples where the best running time is some strange intermediate complexity such as $2^{2^{\sqrt{\log n}}}$. However, it is believed that there exist problems in **NP** that are neither in **P** nor are **NP**-complete, and in fact a result known as "Ladner's Theorem" shows that if **P** $\neq$ **NP** then this is indeed the case (see also Exercise 16.1 and Fig. 16.2).

4

5

[3] The following web page keeps a catalog of these failed attempts. At the time of writing it lists about 110 papers claiming to resolve the question, of which about 60 claim to prove that **P** $=$ **NP** and about 50 claim to prove that **P** $\neq$ **NP**.

[4] TODO: maybe add examples of NP hard problems as a barrier to understanding - problems from economics, physics, etc.. that prevent having a closed-form solutions

[5] TODO: maybe include knots

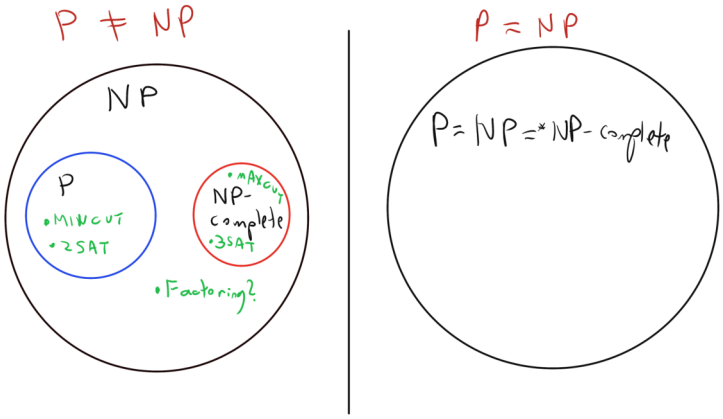

Figure 16.2: The world if $\mathbf{P} \neq \mathbf{NP}$ (left) and $\mathbf{P} = \mathbf{NP}$ (right). In the former case the set of **NP**-complete problems is disjoint from **P** and Ladner's theorem shows that there exist problems that are neither in **P** nor are **NP**-complete. (There are remarkably few natural candidates for such problems, with some prominent examples being decision variants of problems such as integer factoring, lattice shortest vector, and finding Nash equilibria.) In the latter case that $\mathbf{P} = \mathbf{NP}$ the notion of **NP**-completeness loses its meaning, as essentially all functions in **P** (save for the trivial constant zero and constant one functions) are **NP**-complete.

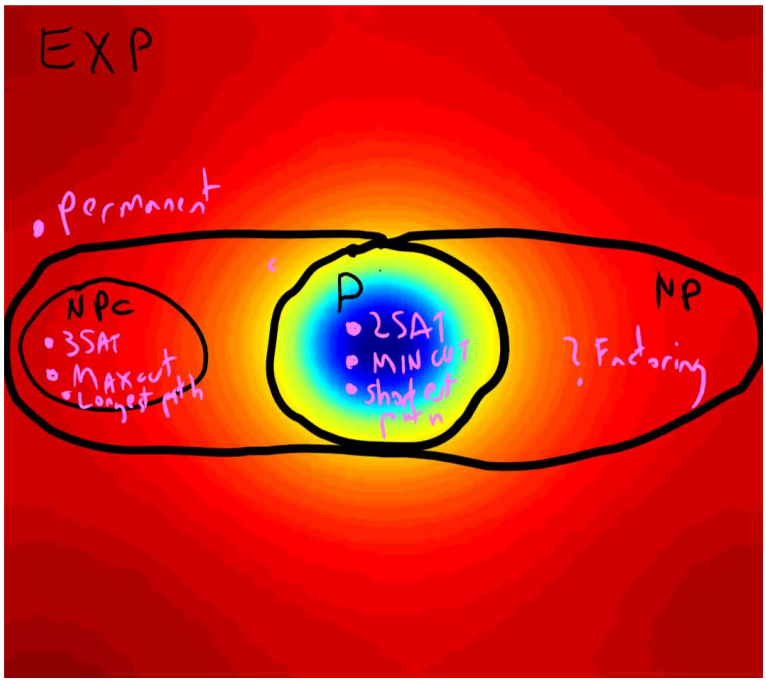

Figure 16.3: A rough illustration of the (conjectured) status of problems in exponential time. Darker colors correspond to higher running time, and the circle in the middle is the problems in **P**. **NP** is a (conjectured to be proper) superclass of **P** and the **NP**-complete problems (or **NPC** for short) are the "hardest" problems in **NP**, in the sense that a solution for one of them implies a solution for all other problems in **NP**. It is conjectured that all the **NP**-complete problems require at least $\exp(n^{\epsilon})$ time to solve for a constant $\epsilon > 0$, and many require $\exp(\Omega(n))$ time. The *permanent* is not believed to be contained in **NP** though it is **NP**-hard, which means that a polynomial-time algorithm for it implies that $\mathbf{P} = \mathbf{NP}$.

16.2 The Cook-Levin Theorem

We will now prove the Cook-Levin Theorem, which is the underpinning to a great web of reductions from 3SAT to thousands of problems across great many fields. Some problems that have been shown to be **NP**-complete include: minimum-energy protein folding, minimum surface-area foam configuration, map coloring, optimal Nash equilibrium, quantum state entanglement, minimum supersequence of a genome, minimum codeword problem, shortest vector in a lattice, minimum genus knots, positive Diophantine equations, integer programming, and many many more. The worst-case complexity of all these problems is (up to polynomial factors) equivalent to that of 3SAT, and through the Cook-Levin Theorem, to all problems in **NP**.

To prove Theorem 16.2 we need to show that $F \leq_p 3SAT$ for every $F \in \mathbf{NP}$. We will do so in three stages. We define two intermediate problems: $NANDSAT$ and $3NAND$. We will shortly show the definitions of these two problems, but Theorem 16.2 will follow from the following three lemmas:

Lemma 16.4 $NANDSAT$ is **NP**-hard.

Lemma 16.5 $NANDSAT \leq_p 3NAND$.

Lemma 16.6 $3NAND \leq_p 3SAT$.

From the transitivity of reductions, Lemma 16.4, Lemma 16.5, and Lemma 16.6 together immediately imply that $3SAT$ is **NP**-hard, hence establishing Theorem 16.2. (Can you see why?) We now prove these three lemmas one by one, providing the requisite definitions as we go along.

16.2.1 The $NANDSAT$ Problem, and why it is **NP** hard.

We define the $NANDSAT$ problem as follows. On input a string $Q \in \{0,1\}^*$, we define $NANDSAT(Q) = 1$ if and only if Q is a valid representation of an n-input and single-output NAND program and there exists some $w \in \{0,1\}^n$ such that $Q(w) = 1$. While we don't need this to prove Lemma 16.4, note that $NANDSAT$ is in **NP** since we can verify that $Q(w) = 1$ using the polyonmial-time algorithm for evaluating NAND programs.[6] We now present the proof of Lemma 16.4.

[6] Q is a NAND program and not a NAND++ program, and hence it is only defined on inputs of some particular size n. Evaluating Q on any input $w \in \{0,1\}^n$ can be done in time polynomial in the number of lines of Q.

Proof Idea: To prove Lemma 16.4 we need to show that for every $F \in \mathbf{NP}$, $F \leq_p NANDSAT$. The high-level idea is that by the definition

of **NP**, there is some NAND++ program P^* and some polynomial $T(\cdot)$ such that $F(x) = 1$ if and only if there exists some w such that $P^*(xw)$ outputs 1 within $T(|x|)$ steps. Now by "unrolling the loop" of the NAND++ program P^* we can convert it into a NAND program Q that on input w will simulate $P^*(xw)$ for $T(|x|)$ steps. We will then get that $NANDSAT(Q) = 1$ if and only if $F(x) = 1$.

Proof of Lemma 16.4. We now present the details. Let $F \in \mathbf{NP}$. By Definition 16.1 there exists $G \in \mathbf{P}$ and $a, b \in \mathbb{N}$ such that for every $x \in \{0,1\}^*$, $F(x) = 1$ if and only if there exists $w \in \{0,1\}^{a|x|^b}$ such that $G(xw) = 1$. Since $G \in \mathbf{P}$ there is some NAND++ program P^* that computes G in at most n'^c time for some constant c where n' is the size of its input. Moreover, as shown in Theorem 7.3, we can assume without loss of generality that P^* is simple in the sense of Definition 7.2.

To prove Lemma 16.4 we need to give a polynomial-time computable map of every $x^* \in \{0,1\}^*$ to a NAND program Q such that $F(x^*) = NANDSAT(Q)$. Let $x^* \in \{0,1\}^*$ be such a string and let $n = |x^*|$ be its length. In time polynomial in n, we can obtain a NAND program Q^* of $n + an^b$ inputs and $|P^*| \cdot (n + an^b)^c$ lines (where $|P^*|$ denotes the number of lines of P^*) such that $Q^*(xw) = P^*(xw)$ for every $x \in \{0,1\}^n$ and $w \in \{0,1\}^{an^b}$. Indeed, we can do this by simply copying and pasting $(n + an^b)^c$ times the code of P^* one after the other, and replacing all references to `i` in the j-th copy with $INDEX(j)$.[7] We also replace references to `validx_`$\langle k \rangle$ with `one` if $k < n + an^b$ and `zero` otherwise. By the definition of NAND++ and the fact that the original program P^* was simple and halted within at most $(n + an^b)^c$ steps, the NAND program Q^* agrees with P^* on every input of the form $xw \in \{0,1\}^{n+an^b}$.[8]

Now we transform Q^* into Q by "hardwiring" its first n inputs to corresond to x^*. That is, we obtain a program Q such that $Q(w) = Q^*(x^*w)$ for every $w \in \{0,1\}^{an^b}$ by replacing all references to the variables `x_`$\langle j \rangle$ for $j < n$ with either `one` or `zero` depending on the value of x^*_j. (We also map `x_`$\langle n \rangle$,...,`x_`$\langle n + an^b \rangle$ to `x_`$\langle 0 \rangle$, ..., `x_`$\langle an^b - 1 \rangle$ so that the number of inputs is reduced from $n + an^b$ to an^b.) You can verify that by this construction, Q has an^b inputs and for every $w \in \{0,1\}^{an^b}$, $Q(w) = Q^*(x^*w)$. We now claim that $NANDSAT(Q) = F(x^*)$. Indeed note that $F(x^*) = 1$ if and only if there exists $w \in \{0,1\}^{an^b}$ s.t. $P^*(x^*w) = 1$. But since $Q^*(xw) = P^*(xw)$ for every x, w of these lengths, and $Q(w) = Q^*(x^*w)$ it follows that this holds if and only if there exists $w \in \{0,1\}^{an^b}$ such that $Q(w) = 1$. But the latter condition holds

[7] Recall that $INDEX(j)$ is the value of the `i` index variable in the j-th iteration. The particular formula for $INDEX(j)$ was given in Eq. (7.4) but all we care is that it is computable in time polynomial in j.

[8] We only used the fact that P^* is simple to ensure that we have access to the `one` and `zero` variables, and that assignments to the output variable `y_0` are "guarded" in the sense that adding extra copies of P^* after it has already halted will not change the output. It is not hard to ensure these properties, as shown in Theorem 7.3.

exactly when $NANDSAT(Q) = 1$. ■

P The proof above is a little bit technical but ultimately follows quite directly from the definition of **NP**, as well as of NAND and NAND++ programs. If you find it confusing, try to pause here and work out the proof yourself from these definitions, using the idea of "unrolling the loop" of a NAND++ program. It might also be useful for you to think how you would implement in your favorite programming language the function `expand` which on input a NAND++ program P and numbers T, n would output an n-input NAND program Q of $O(|T|)$ lines such that for every input $x \in \{0,1\}^n$, if P halts on x within at most T steps and outputs y, then $Q(x) = y$.

16.2.2 *The 3NAND problem*

The $3NAND$ problem is defined as follows: the input is a logical formula φ on a set of variables $z_0, \ldots, z_{r-1}$ which is an AND of constraints of the form $z_i = NAND(z_j, z_k)$. For example, the following is a $3NAND$ formula with 5 variables and 3 constraints:

$$(z_3 = NAND(z_0, z_2)) \wedge (z_1 = NAND(z_0, z_2)) \wedge (z_4 = NAND(z_3, z_1)) \tag{16.2}$$

The output of $3NAND$ on input φ is 1 if and only if there is an assignment to the variables of φ that makes it evaluate to "true" (that is, there is some assignment $z \in \{0,1\}^r$ satisfying all of the constraints of φ). As usual, we can represent φ as a string, and so think of $3NAND$ as a function mapping $\{0,1\}^*$ to $\{0,1\}$. We now prove Lemma 16.5.

Proof Idea: To prove Lemma 16.5 we need to give a polynomial-time map from every NAND program Q to a 3NAND formula φ such that there exists w such that $Q(w) = 1$ if and only if there exists z satisfying φ. This will actually follow directly from our notion of "modification logs" or "deltas" of NAND++ programs (see Definition 7.6). We will have a variable of φ corresponding to every line of Q, with a constraint ensuring that if line i has the form `foo := bar NAND blah` then the variable corresponding to line i should be the NAND of the variables corresponding to the lines in which `bar` and `blah` were just written to. We will also have variables associated with

the input w, and use them in lines such as `foo := x_17 NAND x_33` or `foo := bar NAND x_55`. Finally we add a constraint that requires the last assignment to `y_0` to equal 1. By construction, satisfying assignments to our formula φ will correspond to valid modification logs of executions of Q that end with it outputting 1. Hence in particular there exists a satisfying assignment to φ if and only if there is some input $w \in \{0,1\}^n$ on which the execution of Q on w ends in 1.

Proof of Lemma 16.5. To prove Lemma 16.5 we need to give a reduction from $NANDSAT$ to $3NAND$. Let Q be a NAND program with n inputs, one output, and m lines. We can assume without loss of generality that Q contains the variables `one` and `zero` by adding the following lines in its beginning if needed:

```
notx_0 := x_0 NAND x_0
one := x_0 NAND notx_0
zero := one NAND one
```

We map Q to a $3NAND$ formula φ as follows:

- φ has $m+n$ variables $z_0, \ldots, z_{m+n-1}$
- For every $\ell \in \{n, n+1, \ldots, n+m\}$, if the $\ell - n$-th line of the program Q is `foo := bar NAND blah` then we add to φ the constraint $z_\ell = NAND(z_j, z_k)$ where $j-n$ and $k-n$ correspond to the last lines in which the variables `bar` and `blah` (respectively) were written to. If one or both of `bar` and `blah` was not written to before then we use z_{ℓ_0} instead of the corresponding value z_j or z_k in the constraint, where $\ell_0 - n$ is the line in which `zero` is assigned a value. If one or both of `bar` and `blah` is an input variable `x_i` then we we use z_i in the constraint.
- Let ℓ^* be the last line in which the output `y_0` is assigned a value. Then we add the constraint $z_{\ell^*} = NAND(z_{\ell_0}, z_{\ell_0})$ where $\ell_0 - n$ is as above the last line in which `zero` is assigned a value. Note that this is effectively the constraint $z_{\ell^*} = NAND(0,0) = 1$.

To complete the proof we need to show that there exists $w \in \{0,1\}^n$ s.t. $Q(w) = 1$ if and only if there exists $z \in \{0,1\}^{n+m}$ that satisfies all constraints in φ. We now show both sides of this equivalence.

- **Completeness:** Suppose that there is $w \in \{0,1\}^n$ s.t. $Q(w) = 1$. Let $z \in \{0,1\}^{n+m}$ be defined as follows: for $i \in [n]$, $z_i = w_i$ and for $i \in \{n, n+1, \ldots, n+m\}$ z_i equals the value that is assigned in the $(i-n)$-th line of Q when executed on w. Then by construction

z satisfies all of the constraints of φ (including the constraint that $z_{\ell^*} = NAND(0,0) = 1$ since $Q(w) = 1$.)

- **Soundness:** Suppose that there exists $z \in \{0,1\}^{n+m}$ satisfying φ. Soundness will follow by showing that $Q(z_0,\ldots,z_{n-1}) = 1$ (and hence in particular there exists $w \in \{0,1\}^n$, namely $w = z_0 \cdots z_{n-1}$, such that $Q(w) = 1$). To do this we will prove the following claim $(*)$: for every $\ell \in [m]$, $z_{\ell+n}$ equals the value assigned in the ℓ-th step of the execution of the program Q on $z_0,\ldots,z_{n-1}$. Note that because z satisfies the constraints of φ, $(*)$ is sufficient to prove the soundness condition since these constraints imply that the last value assigned to the variable `y_0` in the execution of Q on $z_0 \cdots w_{n-1}$ is equal to 1. To prove $(*)$ suppose, towards a contradiction, that it is false, and let ℓ be the smallest number such that $z_{\ell+n}$ is *not* equal to the value assigned in the ℓ-th step of the exeuction of Q on $z_0,\ldots,z_{n-1}$. But since z satisfies the constraints of φ, we get that $z_{\ell+n} = NAND(z_i,z_j)$ where (by the assumption above that ℓ is *smallest* with this property) these values *do* correspond to the values last assigned to the variables on the righthand side of the assignment operator in the ℓ-th line of the program. But this means that the value assigned in the ℓ-th step is indeed simply the NAND of z_i and z_j, contradicting our assumption on the choice of ℓ.

■

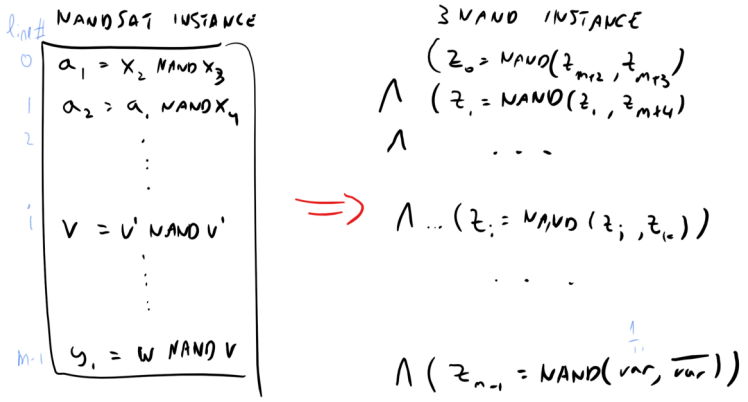

Figure 16.4: We reduce $NANDSAT$ to $3NAND$ by mapping a program P to a formula ψ where we have a variable for each line and input variable of P, and add a constraint to ensure that the variables are consistent with the program. We also add a constraint that the final output is 1. One can show that there is an input x such that $P(x) = 1$ if and only if there is a satisfying assignment for ψ. (NOTATION IN FIGURE NEEDS TO BE UPDATED)

16.2.3 *From 3NAND to 3SAT*

To conclude the proof of Theorem 16.2, we need to show Lemma 16.6 and show that $3NAND \leq_p 3SAT$. We now do so.

Proof Idea: To prove Lemma 16.6 we need to map a 3NAND formula φ into a 3SAT formula ψ such that φ is satisfiable if and only if ψ is. The idea is that we can transform every NAND constraint of the form $a = NAND(b,c)$ into the AND of ORs involving the variables a, b, c and their negations, where each of the ORs contains at most three terms. The construction is fairly straightforward, and the details are given below.

> P It is a good exercise for you to try to find a 3CNF formula ξ on three variables a, b, c such that $\xi(a,b,c)$ is true if and only if $a = NAND(b,c)$. Once you do so, try to see why this implies a reduction from $3NAND$ to $3SAT$, and hence completes the proof of Lemma 16.6

Proof of Lemma 16.6. The constraint

$$z_i = NAND(z_j, z_k) \tag{16.3}$$

is satisfied if $z_i = 1$ whenever $(z_j, z_k) \neq (1,1)$. By going through all cases, we can verify that Eq. (16.3) is equivalent to the constraint

$$(\overline{z_i} \vee \overline{z_j} \vee \overline{z_k}) \wedge (z_i \vee z_j) \wedge (z_i \vee z_k) \ . \tag{16.4}$$

Indeed if $z_j = z_k = 1$ then the first constraint of Eq. (16.4) is only true if $z_i = 0$. On the other hand, if either of z_j or z_k equals 0 then unless $z_i = 1$ either the second or third constraints will fail. This means that, given any 3NAND formula φ over n variables $z_0, \ldots, z_{n-1}$, we can obtain a 3SAT formula ψ over the same variables by replacing every $3NAND$ constraint of φ with three $3OR$ constraints as in Eq. (16.4).[9] Because of the equivalence of Eq. (16.3) and Eq. (16.4), the formula ψ satisfies that $\psi(z_0, \ldots, z_{n-1}) = \varphi(z_0, \ldots, z_{n-1})$ for every assignment $z_0, \ldots, z_{n-1} \in \{0,1\}^n$ to the variables. In particular ψ is satisfiable if and only if φ is, thus completing the proof. ■

[9] The resulting forumula will have some of the OR's involving only two variables. If we wanted to insist on each formula involving three distinct variables we can always add a "dummy variable" z_{n+m} and include it in all the OR's involving only two variables, and add a constraint requiring this dummy variable to be zero.

16.2.4 *Wrapping up*

We have shown that for every function F in **NP**, $F \leq_p NANDSAT \leq_p 3NAND \leq_p 3SAT$, and so $3SAT$ is **NP**-hard. Since in the previous

lecture we saw that $3SAT \leq_p QUADEQ$, $3SAT \leq_p ISET$, $3SAT \leq_p MAXCUT$ and $3SAT \leq_p LONGPATH$, all these problems are **NP**-hard as well. Finally, since all the aforementioned problems are in **NP**, they are all in fact **NP**-complete and have equivalent complexity. There are thousands of other natural problems that are **NP**-complete as well. Finding a polynomial-time algorithm for any one of them will imply a polynomial-time algorithm for all of them.

16.3 Lecture summary

- Many of the problems for which we don't know polynomial-time algorithms are **NP**-complete, which means that finding a polynomial-time algorithm for one of them would imply a polynomial-time algorithm for *all* of them.
- It is conjectured that **NP** $\neq$ **P** which means that we believe that polynomial-time algorithms for these problems are not merely *unknown* but are *nonexistent*.
- While an **NP**-hardness result means for example that a full-fledged "textbook" solution to a problem such as MAX-CUT that is as clean and general as the algorithm for MIN-CUT probably does not exist, it does not mean that we need to give up whenever we see a MAX-CUT instance. Later in this course we will discuss several strategies to deal with **NP**-hardness, including *average-case complexity* and *approximation algorithms*.

16.4 Exercises

Exercise 16.1 — Poor man's Ladner's Theorem. Prove that if there is no $n^{O(\log^2 n)}$ time algorithm for $3SAT$ then there is some $F \in$ **NP** such that $F \notin$ **P** and F is not **NP** complete.[10] ■

[10] **Hint:** Use the function F that on input a formula φ and a string of the form 1^t, outputs 1 if and only if φ is satisfiable and $t = |\varphi|^{\log|\varphi|}$.

16.5 Bibliographical notes

[11]

[11] TODO: credit surveys of Avi, Madhu

16.6 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

16.7 Acknowledgements