Lecture 21

The Cook–Levin theorem

The purpose of this lecture is to state and prove the Cook–Levin theorem, which gives us a primordial NP-complete language, from which many other languages can be proved NP-complete. There are thousands of interesting NP-complete languages known, and the proofs that they are NP-complete all rely on the Cook–Levin theorem. We won't go further down this path in this course, but this is done in CS 341; the process of proving languages (or their associated computational problems) to be NP-complete, based on reductions from other NP-complete problems, is covered in that course.

We're going to split the Cook–Levin theorem into two pieces, mainly for the sake of clarity and exposition. While the actual statement of the second piece is what people usually mean when they refer to "the Cook–Levin theorem," the proof of the first piece is an important part of it.

21.1 Circuit satisfiability

The first piece of the Cook–Levin theorem is concerned with the following language:

$$\text{CIRCUIT-SAT} = \left\{ \langle C \rangle \,:\, \begin{array}{l} C \text{ is an } m\text{-input, 1-output Boolean circuit, and} \\ \text{there exists } x \in \{0,1\}^m \text{ such that } C(x) = 1 \end{array} \right\}. \qquad (21.1)$$

Here $\langle C \rangle$ is the encoding of a Boolean circuit C, which we may assume is over the binary alphabet $\Sigma = \{0,1\}$ for simplicity. As usual, it should be assumed that this is an efficient encoding scheme.

Let us first observe that CIRCUIT-SAT $\in$ NP, which turns out to be rather easy. Consider the following language:

$$B = \left\{ \langle \langle C \rangle, y \rangle \,:\, \begin{array}{l} C \text{ is an } m\text{-input, 1-output Boolean circuit,} \\ y \in \Sigma^*,\ |y| \geq m, \text{ and } C(y_1, \ldots, y_m) = 1 \end{array} \right\}. \qquad (21.2)$$

The language B is contained in P—it is almost the same as the circuit value problem that we observed is in P in the previous lecture. If we set $f(n) = n^2$ for instance, which is a polynomially-bounded time constructible function, we see that

$$\text{CIRCUIT-SAT} = \left\{ x \in \Sigma^* : \begin{array}{l} \text{there exists } y \in \Sigma^* \text{ such that} \\ |y| = f(|x|) \text{ and } \langle x, y \rangle \in B \end{array} \right\}. \tag{21.3}$$

This establishes that CIRCUIT-SAT $\in$ NP.

The first piece of the Cook–Levin theorem establishes that CIRCUIT-SAT is an NP-complete language.

Theorem 21.1 (Cook–Levin theorem, part 1)**.** CIRCUIT-SAT *is NP-complete.*

Proof. We have already observed that CIRCUIT-SAT $\in$ NP, so it remains to prove that CIRCUIT-SAT is NP-hard. To this end, let $A \subseteq \Gamma^*$ be any language contained in NP. Our goal is to prove $A \leq_m^p$ CIRCUIT-SAT.

Because $A \in$ NP, there must exist a language $B \in$ P and a polynomially bounded, time-constructible function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that

$$x \in A \Leftrightarrow \left(\exists y \in \Sigma^*\right) \left[|y| = f(|x|) \ \wedge \langle x, y \rangle \in B \right] \tag{21.4}$$

for every $x \in \Gamma^*$. Because B is decided by a polynomial-time DTM, we may invoke the theorem we proved at the end of the previous lecture to conclude[1] that for every $n \in \mathbb{N}$, there exists a Boolean circuit C_n as follows:

$$C_n(\langle x, y \rangle) = \begin{cases} 1 & \text{if } \langle x, y \rangle \in B \\ 0 & \text{if } \langle x, y \rangle \notin B, \end{cases} \tag{21.5}$$

for every choice of strings $x \in \Gamma^n$ and $y \in \Sigma^{f(n)}$. Moreover, it is possible to compute C_n from x in polynomial time. (In fact, C_n can be computed from 0^n, where $n = |x|$, in polynomial time, as one may conclude from the theorem.) Note that the number of input bits of C_n is not generally n, but rather is the length of the encoding $\langle x, y \rangle$ for x having length n and y having length $f(n)$.

Next, for each string $x \in \Gamma^*$, let us consider a very simple Boolean circuit D_x that computes the Boolean function

$$E_x(y) = \langle x, y \rangle \tag{21.6}$$

[1] Note that we are making use of the assumptions we stated a couple of lectures back on our choice of an encoding scheme, whereby two strings x and y are efficiently encoded as a binary string $\langle x, y \rangle$ whose length $|\langle x, y \rangle|$ only depends on the lengths of x and y, and not on the particular bits in x and y.

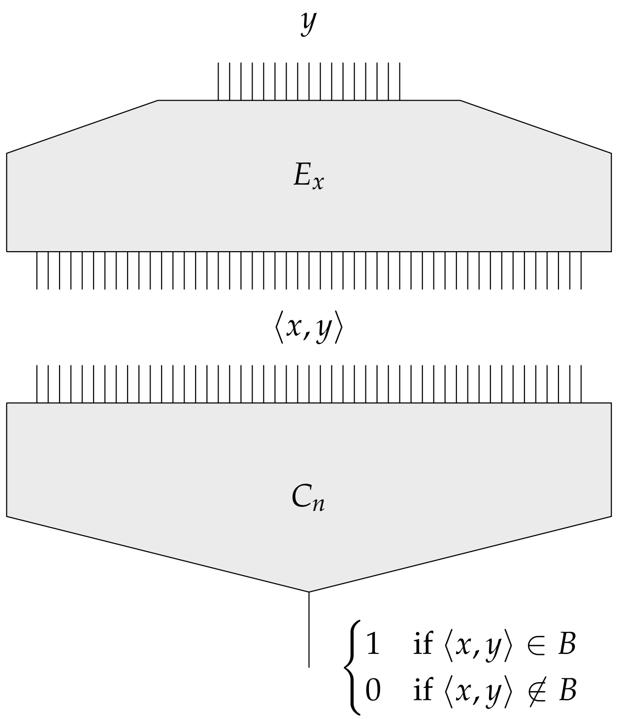
y
E_x
$\langle x, y\rangle$
C_n
$\begin{cases}1 & \text{if } \langle x,y\rangle \in B\\ 0 & \text{if } \langle x,y\rangle \notin B\end{cases}$

Figure 21.1: The circuit D_x is obtained by composing E_x with C_n, for $n = |x|$. It holds that $D_x(y) = 1$ if and only if $\langle x, y\rangle \in B$.

for all strings $y \in \Sigma^{f(n)}$, where $n = |x|$. If x has length n, then this Boolean circuit takes a binary input string y of length $f(n)$, and effectively hard-codes the string x, obtaining the encoding $\langle x, y\rangle$. The function $g : \Gamma^* \to \Sigma^*$ defined as

$$g(x) = \langle E_x \rangle \tag{21.7}$$

is certainly a polynomial-time computable function.

Finally, for each $x \in \Gamma^*$, define a Boolean circuit D_x as suggested in Figure 21.1. That is, the circuit D_x is a composition of E_x with C_n, and it possesses the property that

$$D_x(y) = \begin{cases}1 & \text{if } \langle x,y\rangle \in B\\ 0 & \text{if } \langle x,y\rangle \notin B.\end{cases} \tag{21.8}$$

The function $f : \Gamma^* \to \Sigma^*$ defined as

$$f(x) = \langle D_x \rangle \tag{21.9}$$

for all $x \in \Gamma^*$ is polynomial-time computable, by virtue of the fact that both C_n and E_x can be computed from x in polynomial time.

The function f is a polynomial-time reduction from A to CIRCUIT-SAT, as $x \in A$ if and only if there exists a string $y \in \Sigma^{f(n)}$ such that $\langle x, y \rangle \in B$, which is equivalent to the existence of a string $y \in \Sigma^{f(n)}$ such that $D_x(y) = 1$. As $A \subseteq \Gamma^*$ was an arbitrarily chosen language in NP, and it has been established that $A \leq_m^p$ CIRCUIT-SAT, the proof is complete. □

21.2 Satisfiability of 3CNF formulas

The second part of the Cook–Levin theorem is concerned with *Boolean formulas* in place of Boolean circuits. Recall that a Boolean formula ϕ in Boolean variables $X_1, \ldots, X_n$ is said to be in *conjunctive normal form* (or CNF for short) if it is the case that

$$\phi = \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_m \tag{21.10}$$

where each ψ_k is an OR of a collection of variables or their negations.

Hereafter, whenever we wish to refer to either a variable or the negation of a variable, we will call such a thing a *literal*. For any variable X_j, we often write $\overline{X_j}$ in place of $\neg X_j$, so that the set

$$\{X_1, \overline{X_1}, X_2, \overline{X_2}, \ldots, X_n, \overline{X_n}\} \tag{21.11}$$

represents the possible literals from which each ψ_k above may be formed. Each ψ_k, being an OR of a collection of literals, is called a *clause*. You can think of a clause in a CNF formula as being similar to a clause in a contract or legal agreement. That is, because the Boolean formula ϕ is an AND of the clauses $\psi_1, \ldots, \psi_m$, it must be that all of the clauses are true in order for ϕ to be true—just like all of the clauses in a legal document represent things that must be obeyed.

Here is an example of a CNF formula in the variables X_1, X_2, X_3, X_4:

$$\phi = (X_1 \vee X_2 \vee X_3 \vee \overline{X_4}) \wedge (\overline{X_2} \vee \overline{X_3}) \wedge (X_1 \vee X_3 \vee X_4) \wedge (\overline{X_1} \vee X_3). \tag{21.12}$$

There are four clauses in this formula:

$$\begin{aligned} \psi_1 &= (X_1 \vee X_2 \vee X_3 \vee \overline{X_4}), \\ \psi_2 &= (\overline{X_2} \vee \overline{X_3}), \\ \psi_3 &= (X_1 \vee X_3 \vee X_4), \\ \psi_4 &= (\overline{X_1} \vee X_3). \end{aligned} \tag{21.13}$$

No restrictions are placed on the number of literals in each clause of a CNF formula—you could even have a clause consisting of just one literal if you like. A formula is said to be a *3CNF formula* if it is the case that it is a CNF formula with exactly three literals in each clause. For example, the formula (21.12) is not a 3CNF formula, but the following formula is:

$$\psi = \bigl(X_2 \vee X_3 \vee \overline{X_4}\bigr) \wedge \bigl(X_1 \vee \overline{X_2} \vee \overline{X_3}\bigr) \wedge \bigl(X_1 \vee X_3 \vee X_4\bigr) \wedge \bigl(\overline{X_1} \vee X_2 \vee X_3\bigr). \tag{21.14}$$

A Boolean formula ϕ in the variables $X_1, \ldots, X_n$ is *satisfiable* if there exists a Boolean assignment for the variables that causes the formula to evaluate to 1. For example, both of the formulas ϕ and ψ given above are satisfiable: the assignment $X_1 = 1$, $X_2 = 0$, $X_3 = 1$, $X_4 = 0$ works for both of them.

Now let us define a new language:

$$\text{3SAT} = \bigl\{\langle\phi\rangle \,:\, \phi \text{ is a satisfiable 3CNF formula}\bigr\}. \tag{21.15}$$

The second part of the Cook–Levin theorem establishes that this language is NP-complete.

Theorem 21.2 (Cook–Levin theorem, part 2)**.** 3SAT *is NP-complete.*

Proof. The fact that 3SAT $\in$ NP is similar to CIRCUIT-SAT $\in$ NP, which we have already observed. It therefore remains to prove that 3SAT is NP-hard.

To this end we will prove that

$$\text{CIRCUIT-SAT} \leq_m^p \text{3SAT}. \tag{21.16}$$

In order to establish this reduction, we will devise a polynomial-time computable function f such that, for every Boolean circuit C with m inputs and 1 outputs, it holds that

$$f(\langle C\rangle) = \langle\phi_C\rangle \tag{21.17}$$

for some 3CNF formula ϕ_C with the property that ϕ_C is satisfiable if and only if C is satisfiable.

We cannot directly express a Boolean circuit directly as a 3CNF formula—it could be that the smallest 3CNF formula equivalent to a given circuit has an exponential number of clauses. So, instead of trying to represent a circuit directly as a 3CNF formula we will use a simple trick: for a given Boolean circuit C, we construct a 3CNF formula ϕ_C having a separate variable for every *node* of C (as opposed to just the nodes corresponding to inputs).

More specifically, suppose that C is a Boolean circuit, and let us define a 3CNF formula ϕ_C as follows. We introduce one Boolean variable for each node of C according to the following pattern:

1. The input nodes of C, labeled by its n inputs $X_1, \ldots, X_n$, will correspond to n variables of ϕ_C that are also denoted $X_1, \ldots, X_n$; no confusion will result from using the same names for the inputs of C and the variables of ϕ_C corresponding to these inputs.

2. The constant nodes of C will have associated variables $Y_1, \dots, Y_k$.
3. The gate nodes of C will have associated variables $Z_1, \dots, Z_m$.

We now include a small collection of clauses in ϕ_C for each node of C, with the exception of the nodes labeled $X_1, \dots, X_n$. These clauses represent constraints that would necessarily be satisfied if the variable associated with each node were equal to that node's Boolean value. In addition, we will include one clause that ensures that the circuit output is 1. The way this is done depends on the type of each gate, in the following way:

1. *AND gates.* If Z is the variable corresponding to an AND gate, and the two input nodes for this gate have corresponding variables W_1 and W_2, then these four clauses (expressed as an AND of clauses) are included in ϕ_C:

$$\bigl(W_1 \vee W_2 \vee \overline{Z}\bigr) \wedge \bigl(W_1 \vee \overline{W_2} \vee \overline{Z}\bigr) \wedge \bigl(\overline{W_1} \vee W_2 \vee \overline{Z}\bigr) \wedge \bigl(\overline{W_1} \vee \overline{W_2} \vee Z\bigr). \tag{21.18}$$

   Note that the Boolean formula corresponding to the AND of these four clauses evaluates to 1 if and only if $W_1 \wedge W_2 = Z$.

2. *OR gates.* If Z is the variable corresponding to an OR gate, and the two input nodes for this gate have corresponding variables W_1 and W_2, then these four clauses (again expressed as an AND of clauses) are included in ϕ_C:

$$\bigl(W_1 \vee W_2 \vee \overline{Z}\bigr) \wedge \bigl(W_1 \vee \overline{W_2} \vee Z\bigr) \wedge \bigl(\overline{W_1} \vee W_2 \vee Z\bigr) \wedge \bigl(\overline{W_1} \vee \overline{W_2} \vee Z\bigr). \tag{21.19}$$

   Note that the Boolean formula corresponding to the AND of these four clauses evaluates to 1 if and only if $W_1 \vee W_2 = Z$.

3. *NOT gates.* If Z is the variable corresponding to a NOT gate, and the input node for this gate has corresponding variable W, then these two clauses[2] (again expressed as an AND of clauses) are included in ϕ_C:

$$\bigl(W \vee W \vee Z\bigr) \wedge \bigl(\overline{W} \vee \overline{W} \vee \overline{Z}\bigr). \tag{21.20}$$

   Note that the Boolean formula corresponding to the AND of these two clauses evaluates to 1 if and only if $Z = \neg W$.

[2] The literals W and $\overline{W}$ have been repeated so that each clause contain 3 literals, as is required by a 3CNF formula. If for some reason we wish to disallow repeated literals, we can always introduce a dummy variable instead. For example, the clauses $(W \vee Z \vee D) \wedge (W \vee Z \vee \overline{D})$ can substitute $(W \vee W \vee Z)$. A similar replacement (using two dummy variables and four clauses) is possible for a clause of the form $(Z \vee Z \vee Z)$ where a single literal appears three times.

4. *Constant nodes.* If Z is the variable corresponding to a constant node that is set to 0, then this clause is included in ϕ_C:
$$(\overline{\mathsf{Z}} \vee \overline{\mathsf{Z}} \vee \overline{\mathsf{Z}}). \tag{21.21}$$
If Z is the variable corresponding to a constant node that is set to 1, then this clause is included in ϕ_C:
$$(\mathsf{Z} \vee \mathsf{Z} \vee \mathsf{Z}). \tag{21.22}$$
In each case, the clause evaluates to 1 if and only if the variable equals the constant value corresponding to the node.

5. *Output node.* If Z is the variable corresponding to the node that has been designated as the output node of C, then this clause is included in ϕ_C:
$$(\mathsf{Z} \vee \mathsf{Z} \vee \mathsf{Z}). \tag{21.23}$$
The clause evaluates to 1 if and only if the variable is set to 1.

The formula ϕ_C is the AND of all of the clauses described above. It is evident that $\langle \phi_C \rangle$ can be computed from $\langle C \rangle$ in polynomial time—it is a very simple computational procedure to examine the nodes of C, in an arbitrary order, and output the clauses described above.

Finally, we observe that ϕ_C is satisfiable if and only if C is satisfiable. The reason is that for any setting of the variables $X_1, \ldots, X_n$, there is a unique assignment to the remaining variables that satisfy all of the clauses introduced above, with the possible exception of the last clause corresponding to the output node of C. This is because the clauses constructed above force each variable to take a value that is consistent with an evaluation of each node of the circuit C. Consequently, if C is satisfiable, then a satisfying assignment to ϕ_C is obtained by first setting the input variables $X_1, \ldots, X_n$ of ϕ_C to any values that satisfy C, and then setting every other variable in a way that is consistent with an evaluation of the circuit. Conversely, if ϕ_C is satisfiable, then whatever values this assignment gives to the variables $X_1, \ldots, X_n$ must necessarily satisfy C; for an evaluation of C must be consistent with the setting of the variables that satisfied ϕ_C. □