Bocconi University— 30540: Computer Science II — Handout NP4
Fabrizio Iozzi and Luca Trevisan — May 9, 2023

In this lecture we prove our fist NP-completeness result, and we show that the Circuit-SAT problem is NP-complete. Once we have one NP-complete problem, each subsequent NP-completeness result requires only one reduction. We will show a reduction from Circuit-SAT to 3SAT, which will give us our second NP-complete problem.

1 NP-completeness of Circuit SAT

A Boolean circuit is an abstract device that computes Boolean functions of Boolean inputs. The device is made of *Boolean gates* connected by *wires*. We can model a circuit as a directed acyclic graph whose nodes are gates and whose edges are wires connecting gates. The inputs of the circuit are modelled as the sources (nodes of indegree 0) of the graph and the outputs are modelled as the sinks of the graph (nodes of outdegree 0).

The *size* of a circuit is the number of gates.

One obtains different kinds of circuit depending on the kind of Boolean gates that one allows. For our purposes, we will allow AND gates, OR gates and NOT gates. A NOT gate has indegree 1, and the output is the negation of the input, that is, if the input is 0 the output is 1 and vice versa. An AND gate has indegree 2 and computes the product of the inputs, so if both inputs are 1 then the output 1, otherwise the output is 0. An OR gate has indegree 2 and the output is 1 if at least one input is 1, and the output is 0 if both inputs are 0. To describe the computation of a circuit on a given input, label the nodes corresponding to the inputs by the bits of the input, then proceed in topological order to evaluate each gate given the label of the input nodes/gates. The label of the output nodes is the output of the circuit.

A first point is that Boolean circuits, as described above, can compute any function.

Lemma 1 *Let $f : \{0,1\}^n \to \{0,1\}$ be an arbitrary Boolean function. Then there is a circuit of size $O(2^n)$ that computes f.*

Proof: We give a recursive construction of the circuit. If $n = 1$, then either $f(x) = x$, in which case it is computed by a circuit of zero gates, or $f(x) = \neg x$, which can be computed by a circuit of size one, or $f(x) = 0 = (x \wedge \neg x)$, which is computed by a circuit of size two, or $f(x) = 1 = (x \vee \neg x)$, which is also computed by a circuit of size 2.

Let now f be an arbitrary function of n variables. We can write it as

$$f(x_1, \ldots, x_n) = (x_n \wedge f(x_1, \ldots, x_{n-1}, 1)) \vee (\neg x_n \wedge f(x_1, \ldots, x_{n-1}, 0))$$

Where both $f(x_1, \ldots, x_{n-1}, 1)$ and $f(x_1, \ldots, x_{n-1}, 0)$ are functions of $n-1$ variables, that can be recursively realized by a circuit.

The size $S(n)$ of the circuit constructed this way satisfies the recursion $S(1) \leq 2$, $S(n) \leq 4 + 2S(n-1)$, which solves to $S(n) \leq 3 \cdot 2^n - 4$. $\square$

The main result that we will use (without providing a full proof) is as follows.

Theorem 2 (Circuits can Simulate Algorithms) *If Alg is an algorithm that takes worst-case time $t(n)$ on inputs of length n, then for every n there is a boolean circuit of size $O(t^2(n))$ such that, for every input x of length n, the output of the circuit on input x is the same as the output of Alg on input x. Furthermore, given the code of Alg and the parameter n, such a circuit is computable in time polynomial in $t(n)$.*

We have not built enough techniques to prove the above theorem: in particular, we have not provided a sufficiently formal definition of what is an algorithm and what is the running time of an algorithm on a particular input. The general idea is that if an algorithm uses at most $s(n)$ bits of memory and runs for at most $t(n)$ steps on inputs of length n, then the execution of one step of the algorithm can be simulated by a circuit of size $O(s(n))$, and the execution of the entire algorithm can be simulated by combining $t(n)$ such circuits, one per computational step, yielding a simulating circuit of size $O(t(n) \cdot s(n))$. Finally, one uses the fact that $s(n) \leq O(t(n))$ because each step of the algorithm affects at most $O(1)$ memory locations (such a statement will depend on the precise way in which we define "algorithm" and "time step").

Definition 3 (Circuit SAT) *The Circuit-SAT problem is defined as follows: given a boolean circuit C with a one-bit output, find an input x such that $C(x) = 1$, if such an input exists.*

Theorem 4 *Circuit-SAT is NP-complete.*

Proof: It follows from the definition that Circuit-SAT is an NP search problem: the definition of the value of $C(x)$ for a given circuit C and input x is an algorithm definition that can be implemented in linear time.

We prove that circuit-SAT is NP-hard as follows. Let X be an NP search problem, and V_X be the polynomial time computable property that defines the NP search problem X.

Given an input x for X of length n, the function f of the reduction constructs a circuit C_x such that, for every possible solution S for x, we have $C_x(S) = V_X(S)$.

The function $g(x, S)$ just outputs S. If the function f behaves as prescribed, and if it is computable in time polynomial in $len(x)$, then it is clear that what we have described is a reduction.

The polynomial time computation of f proceeds as follows. Given x, we compute $n = len(x)$, and we use the "furthermore" part of Theorem **??** to construct a circuit C_n such that for every x and every S we have $C_n(x, S) = V_X(x, S)$. Since V_X runs in time polynomial in $len(x)$, the circuit C_n can be constructed in time polynomial in n. Finally, we let C_x be the circuit C_n modified so that the first n inputs are "hard-wired" to be x. □

2 NP-completeness of 3SAT

An input of 3SAT problem is a Boolean formula over Boolean variables $x_1, \ldots, x_n$ in 3-Conjunctive Normal Form (3CNF). Conjunctive normal form means that the formula is a AND-of-OR, and the number 3 stands for the fact that each OR is over three variables or negated variables. For example the following is a 3CNF Boolean formula over Boolean variables x_1, x_2, x_3, x_4, x_5:

$$(x_1 \vee x_2 \vee x_5) \wedge (x_2 \vee \bar{x}_4 \vee x_5) \wedge (x_1 \vee \bar{x}_3 \vee \bar{x}_4) \wedge (\bar{x}_2 \vee x_3 \vee x_5) \tag{1}$$

The symbol $\wedge$ stands for AND (looks a bit like an "A" as in "And") and the symbol $\vee$ stands for OR (looks like a "v", as in "vel" which is Latin for "or" – don't look for logic in the way logicians define notation). The bar over a variable stands for negation, so $\bar{x}_2$ should be read as NOT x_2.

Given a 3CNF Boolean formula, the goal of the 3SAT problem is to find an assignment of Boolean values to the Boolean variables that makes the formula evaluate to True, if such an assignment exists (if this happens we say that the assignment *satisfies* the clause). For example, assigning every variable to True satisfies the formula in (**??**). A couple more useful pieces of terminology: the OR subformulas in a 3CNF formula (like $x_2 \vee \bar{x}_4 \vee x_5$) are called the *clauses* of the formula. A variable like x_2 and a negated variable like $\bar{x}_4$ are called *literals.* Thus a 3CNF formula is an AND of clauses, each clause is an OR of three literals, and each literal is either a variable or the negation of a variable.

We call $at - most - 3SAT$ the variant of 3SAT in which each clause contains at most 3 literals rather than exactly 3.

Theorem 5 *at-most-3SAT is NP-complete.*

Proof: It is clear that, given a Boolean formula and an assignment to the variables, we can verify in linear time if the assignment satisfies the formula, so 3SAT is an NP-search problem.

To complete the proof we want to show that Circuit-SAT $\leq$ 3SAT. Given a circuit C that has n inputs and m gates, we define a Boolean formula with $n+m$ variables $g_1, \ldots, g_{n+m}$ as follows. Let us think of the DAG with $n+m$ vertices that is associated to the circuit C: it has vertices $v_1, \ldots, v_n$ associated with the n inputs, and then m vertices $v_{n+1}, \ldots, v_{n+m}$ associated with the m gates. Assume that we numbered vertices consistently with a topological order, so that v_{n+m} corresponds to the output gate. We construct the formula F by associating a small formula to each gate and taking the AND of all the small formulas associated to the gates, and the AND of the above with g_{n+m}.

For each $i = 1, \ldots, m$, the small formula associated to the vertex v_{n+i}, that is, to the i-th gate is:

- If v_{n+i} corresponds to a NOT gate and the incoming edge comes from v_j, then the formula is
$$(g_{n+i} \vee \bar{g}_j) \wedge (\bar{g}_{n+i} \vee g_j)$$
which is equivalent to the equation $g_{n+i} = \bar{g}_j$.

- if v_{n+i} corresponds to an AND gate, and the incoming edges come from v_j and v_h, then the formula is
$$(g_{n+i} \vee \bar{g}_j \vee \bar{g}_h) \wedge (\bar{g}_{n+i} \vee g_j \vee g_h) \wedge (\bar{g}_{n+i} \vee \bar{g}_j \vee g_h) \wedge (\bar{g}_{n+i} \vee g_j \vee \bar{g}_h)$$
which is equivalent to the equation $g_{n+i} = g_j \wedge g_h$.

- if v_{n+i} corresponds to an OR gate, and the incoming edges come from v_j and v_h, then the formula is
$$(g_{n+i} \vee \bar{g}_j \vee \bar{g}_h) \wedge (\bar{g}_{n+i} \vee g_j \vee g_h) \wedge (g_{n+i} \vee \bar{g}_j \vee g_h) \wedge (g_{n+i} \vee g_j \vee \bar{g}_h)$$
which is equivalent to the equation $g_{n+i} = g_j \vee g_h$.

Now we have the following claims.

1. If $(g_1, \ldots, g_{n+m}) = (b_1, \ldots, b_{n+m})$ is an assignment that satisfies the formula F defined above, then, on input $(b_1, \ldots, b_n)$, the i-th gate of circuit C evaluates to b_{n+i}, as can be verified by induction on i. Furthermore, $b_{n+m} = 1$, and so the circuit outputs 1.

2. If $x_1, \ldots, x_n$ is an input such that $C(x_1, \ldots, x_n) = 1$, and if y_i is the value of gate i of circuit C on input $x_1, \ldots, x_n$, then $(g_1, \ldots, g_{n+m}) = (x_1, \ldots, x_n, y_1, \ldots, y_m)$ satisfies the formula F.

We claim that if we define $f(C)$ to be the formula F described above, and

$$g(C, b_1, \ldots, b_{n+m}) = b_1, \ldots, b_n$$

then we have a reduction from Circuit-SAT to at-most-3SAT.

The point is that the first claim above shows that if $b_1, \ldots, b_{n+m}$ is a valid solution for F, then $b_1, \ldots, b_n$ is a valid solution for C. The second claim above shows that if there is a solution for C then there is a solution for F, and so if there is no solution for F then there is no solution for C either. $\Box$

Theorem 6 *3SAT is NP-complete*

PROOF: We show that at-most-3SAT $\leq$ 3SAT.

Given a CNF formula F in which every clause has at most three literals, we construct a formula F' in which every clause has exactly three literals, and such that F is, in an appropriate sense, equivalent to F'.

We repeatedly apply the following procedure until all clauses have exactly three literals:

- If there is a clause $(x_i \wedge x_j)$ with two literals, remove it, and replace it with the two clauses
$$(x_i \vee x_j \vee y) \wedge (x_i \vee x_j \vee \bar{y})$$
where y is a new variable that does not appear anywhere else in the formula (similarly if x_i and/or x_j are complemented in the clause);
- If there is a clause (x_i) with a single literal, remove it, and replace it with the four clauses
$$(x_i \vee y \vee y') \wedge (x_i \vee \bar{y} \vee y') \wedge (x_i \vee y \vee \bar{y}') \wedge (x_i \vee \bar{y} \vee \bar{y}')$$
where y and y' are new variables that appear nowhere else in the formula (similarly if the clause is $\bar{x}_i$).

At each step of this transformation, we maintain the invariant that if there is an assignment to the x variables that satisfies F, then it can be extended with an appropriate assignment to the y variables to create an assignment that satisfies the new formula, and if there is an assignment to the x variables and the y variables that satisfies the new formula, then restricting this assignment to the x variables satisfies F.

The reduction maps F to F' and an assignment to the x and y variables that satisfies F' to the restriction of the assignment to only the x variables. $\Box$