Notes 22 for CS 170

1 NP-completeness of Circuit-SAT

We will prove that the circuit satisfiability problem CSAT described in the previous notes is **NP**-complete.

Proving that it is in **NP** is easy enough: The algorithm $V()$ takes in input the description of a circuit C and a sequence of n Boolean values $x_1, \dots x_n$, and $V(C, x_1, \dots, x_n) = C(x_1, \dots, x_n)$. I.e. V *simulates* or *evaluates* the circuit.

Now we have to prove that for every decision problem A in **NP**, we can find a reduction from A to CSAT. This is a difficult result to prove, and it is impossible to prove it really formally without introducing the *Turing machine* model of computation. We will prove the result based on the following fact, of which we only give an informal proof.

Theorem 1
Suppose A is a decision problem that is solvable in $p(n)$ time by some program P, where n is the length of the input. Also assume that the input is represented as a sequence of bits.

Then, for every fixed n, there is a circuit C_n of size about $O((p(n)^2) \cdot (\log p(n))^{O(1)})$ such that for every input $x = (x_1, \dots, x_n)$ of length n, we have

$$A(x) = C_n(x_1, \dots, x_n)$$

That is, circuit C_n solves problem A on all the inputs of length n.

Furthermore, there exists an efficient algorithm (running in time polynomial in $p(n)$) that on input n and the description of P produces C_n.

The algorithm in the "furthermore" part of the theorem can be seen as the ultimate CAD tool, that on input, say, a C++ program that computes a boolean function, returns the description of a circuit that computes the same boolean function. Of course the generality is paid in terms of inefficiency, and the resulting circuits are fairly big.

Proof: [Sketch] Without loss of generality, we can assume that the language in which P is written is some very low-level machine language (as otherwise we can compile it).

Let us restrict ourselves to inputs of length n. Then P runs in at most $p(n)$ steps. It then accesses at most $p(n)$ cells of memory.

At any step, the "global state" of the program is given by the content of such $p(n)$ cells plus $O(1)$ registers such as program counter etc. No register/memory cell needs to contain numbers bigger than $\log p(n) = O(\log n)$. Let $q(n) = (p(n) + O(1))O(\log n)$ denote the size of the whole global state.

We maintain a $q(n) \times p(n)$ "tableau" that describes the computation. The row i of the tableau is the global state at time i. Each row of the tableau can be computed starting from the previous one by means of a small circuit (of size about $O(q(n))$). In fact the microprocessor that executes our machine language is such a circuit (this is not totally accurate). $\square$

Now we can argue about the **NP**-completeness of CSAT. Let us first think of how the proof would go if, say, we want to reduce the Hamiltonian cycle problem to CSAT. Then, given a graph G with n vertices and m edges we would construct a circuit that, given in input a sequence of n vertices of G, outputs 1 if and only if the sequence of vertices is a Hamiltonian cycle in G. How can we construct such a circuit? There is a computer program that given G and the sequence checks if the sequence is a Hamiltonian cycle, so there is also a circuit that given G and the sequence does the same check. Then we "hard-wire" G into the circuit and we are done. Now it remains to observe that the circuit is a Yes-instance of CSAT if and only if the graph is Hamiltonian.

The example should give an idea of how the general proof goes. Take an arbitrary problem A in **NP**. We show how to reduce A to Circuit Satisfiability.

Since A is in **NP**, there is some polynomial-time computable algorithm V_A and a polynomial p_A such that $A(x) =$ YES if and only if there exists a y, with $length(y) \leq p_A(length(x))$, such that $V(x, y)$ outputs YES.

Consider now the following reduction. On input x of length n, we construct a circuit C that on input y of length $p(n)$ decides whether $V(x, y)$ outputs YES or NOT.

Since V runs in time polynomial in $n + p(n)$, the construction can be done in polynomial time. Now we have that the circuit is satisfiable if and only if $x \in A$.

2 Proving More NP-completeness Results

Now that we have one **NP**-complete problem, we do not need to start "from scratch" in order to prove more **NP**-completeness results. Indeed, the following result clearly holds:

Lemma 2
If A reduces to B, and B reduces to C, then A reduces to C.

Proof: If A reduces to B, there is a polynomial time computable function f such that $A(x) = B(f(x))$; if B reduces to C it means that there is a polynomial time computable function g such that $B(y) = C(g(y))$. Then we can conclude that we have $A(x) = C(g(f(x))$, where $g(f())$ is computable in polynomial time. So A does indeed reduce to C. $\Box$

Suppose that we have some problem A in **NP** that we are studying, and that we are able to prove that CSAT reduces to A. Then we have that every problem N in **NP** reduces to CSAT, which we have just proved, and CSAT reduces to A, so it is also true that every problem in **NP** reduces to A, that is, A is **NP**-hard. This is very convenient: a single reduction from CSAT to A shows the existence of all the infinitely many reductions needed to establish the **NP**-hardness of A. This is a general method:

Lemma 3
Let C be an **NP**-*complete problem and A be a problem in* **NP**. *If we can prove that C reduces to A, then it follows that A is* **NP**-*complete.*

Right now, literally thousands of problems are known to be **NP**-complete, and each one (except for a few "root" problems like CSAT) has been proved **NP**-complete by way of a single reduction from another problem previously proved to be **NP**-complete. By the definition, all **NP**-complete problems reduce to each other, so the body of work that lead to the proof of the currently known thousands of **NP**-complete problems, actually implies *millions* of pairwise reductions between such problems.

3 NP-completeness of SAT

We defined the CNF Satisfiability Problem (abbreviated SAT) above. SAT is clearly in **NP**. In fact it is a special case of Circuit Satisfiability. (Can you see why?) We want to prove that SAT it is **NP**-hard, and we will do so by reducing from Circuit Satisfiability.

First of all, let us see how *not* to do the reduction. We might be tempted to use the following approach: given a circuit, transform it into a Boolean CNF formula that computes the same Boolean function. Unfortunately, this approach cannot lead to a polynomial time reduction. Consider the Boolean function that is 1 iff an odd number of inputs is 1. There is a circuit of size $O(n)$ that computes this function for inputs of length n. But the smallest CNF for this function has size more than 2^n.

This means we cannot translate a circuit into a CNF formula of comparable size that computes the same function, but we may still be able to transform a circuit into a CNF formula such that the circuit is satisfiable iff the formula is satifiable (although the circuit and the formula do compute somewhat different Boolean functions).

We now show how to implement the above idea. We will need to add new variables. Suppose the circuit C has m gates, including input gates, then we introduce new variables $g_1, \ldots, g_m$, with the intended meaning that variable g_j corresponds to the output of gate j.

We make a formula F which is the AND of $m+1$ sub-expression. There is a sub-expression for every gate j, saying that the value of the variable for that gate is set in accordance to the value of the variables corresponding to inputs for gate j.

We also have a $(m+1)$-th term that says that the output gate outputs 1. There is no sub-expression for the input gates.

For a gate j, which is a NOT applied to the output of gate i, we have the sub-expression

$$(g_i \vee g_j) \wedge (\bar{g}_i \vee \bar{g}_j)$$

For a gate j, which is a AND applied to the output of gates i and l, we have the sub-expression

$$(\bar{g}_j \vee g_i) \wedge (\bar{g}_j \vee g_l) \wedge (g_j \vee \bar{g}_i \vee \bar{g}_l)$$

Similarly for OR.

This completes the description of the reduction. We now have to show that it works. Suppose C is satisfiable, then consider setting g_j being equal to the output of the j-th gate of C when a satisfying set of values is given in input. Such a setting for $g_1, \ldots, g_m$ satisfies F.

Suppose F is satisfiable, and give in input to C the part of the assignment to F corresponding to input gates of C. We can prove by induction that the output of gate j in C is also equal to g_j, and therefore the output gate of C outputs 1.

So C is satisfiable if and only if F is satisfiable.

4 NP-completeness of 3SAT

SAT is a much simpler problem than Circuit Satisfiability, if we want to use it as a starting point of **NP**-completeness proofs. We can use an even simpler starting point: 3-CNF Formula Satisfiability, abbreviated 3SAT. The 3SAT problem is the same as SAT, except that each OR is on precisely 3 (possibly negates) variables. For example, the following is an instance of 3SAT:

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

Certainly, 3SAT is in **NP**, just because it's a special case of SAT.

In the following we will need some terminology. Each little OR in a SAT formula is called a *clause*. Each occurrence of a variable, complemented or not, is called a *literal*.

We now prove that 3SAT is **NP**-complete, by reduction from SAT. Take a formula F of SAT. We transform it into a formula F' of 3SAT such that F' is satisfiable if and only if F is satisfiable.

Each clause of F is transformed into a sub-expression of F'. Clauses of length 3 are left unchanged.

A clause of length 1, such as (x) is changed as follows

$$(x \vee y_1 \vee y_2) \wedge (x \vee y_1 \vee \bar{y}_2)(x \vee \bar{y}_1 \vee y_2) \wedge (x \vee \bar{y}_1 \vee \bar{y}_2)$$

where y_1 and y_2 are two new variables added specifically for the transformation of that clause.

A clause of length 2, such as $x_1 \vee x_2$ is changed as follows

$$(x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \bar{y})$$

where y is a new variable added specifically for the transformation of that clause.

For a clause of length $k \geq 4$, such as $(x_1 \vee \cdots \vee x_k)$, we change it as follows

$$(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee y_2) \wedge (\bar{y}_2 \vee x_4 \vee y_4) \wedge \cdots \wedge (\bar{y}_{k-3} \vee x_{k-1} \vee x_k)$$

where $y_1, \cdots, y_{k-3}$ are new variables added specifically for the transformation of that clause.

We now have to prove the correctness of the reduction.

- We first argue that if F is satisfiable, then there is an assignment that satisfies F'.

  For the shorter clauses, we just set the y-variables arbitrarily. For the longer clause it is slightly more tricky.

- We then argue that if F is not satisfiable, then F' is not satisfiable.

  Fix an assignment to the x variables. Then there is a clause in F that is not satisfied. We argue that one of the derived clauses in F' is not satisfied.

5 Some NP-complete Graph Problems

5.1 Independent Set

Given an undirected non-weighted graph $G = (V, E)$, an *independent set* is a subset $I \subseteq V$ of the vertices such that no two vertices of I are adjacent. (This is similar to the notion of a *matching*, except that it involves vertices and not edges.)

We will be interested in the following optimization problem: given a graph, find a largest independent set. We have seen that this problem is easily solvable in forests. In the general case, unfortunately, it is much harder.

The problem models the execution of conflicting tasks, it is related to the construction of error-correcting codes, and it is a special case of more interesting problems. We are going to prove that it is not solvable in polynomial time unless $\mathbf{P} = \mathbf{NP}$.

First of all, we need to formulate it as a decision problem:

- Given a graph G and an integer k, does there exist an independent set in G with at least k vertices?

It is easy to see that the problem is in NP. We have to see that it is NP-hard. We will reduce 3SAT to Maximum Independent Set.

Starting from a formula ϕ with n variables $x_1, \ldots, x_n$ and m clauses, we generate a graph G_ϕ with $3m$ vertices, and we show that the graph has an independent set with at least m vertices if and only if the formula is satisfiable. (In fact we show that the size of the largest independent set in G_ϕ is equal to the maximum number of clauses of ϕ that can be simultaneously satisfied. — This is more than what is required to prove the NP-completeness of the problem)

The graph G_ϕ has a triangle for every clause in ϕ. The vertices in the triangle correspond to the three literals of the clause.

Vertices in different triangles are joined by an edge iff they correspond to two literals that are one the complement of the other. In Figure 1 we see the graph resulting by applying the reduction to the following formula:

$$(x_1 \vee \neg x_5 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (x_3 \vee x_2 \vee x_4)$$

To prove the correctness of the reduction, we need to show that:

- If ϕ is satisfiable, then there is an independent set in G_ϕ with at least m vertices.
- If there is an independent set in G with at least m vertices, then ϕ is satisfiable.

From Satisfaction to Independence. Suppose we have an assignment of Boolean values to the variables $x_1, \ldots, x_n$ of ϕ such that all the clauses of ϕ are satisfied. This means that for every clause, at least on of its literals is satisfied by the assignment. We construct an independent set as follows: for every triangle we pick a node that corresponds to a satisfied literal (we break ties arbitrarily). It is impossible that two such nodes are adjacent, since only nodes that corresponds to a literal and its negation are adjacent; and they cannot be both satisfied by the assignment.

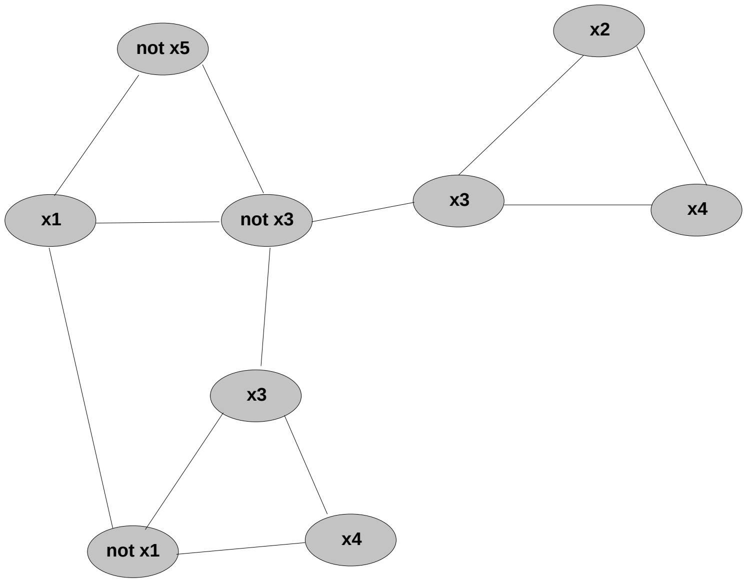

Figure 1: The reduction from 3SAT to Independent Set.

From Independence to Satisfaction. Suppose we have an independent set I with m vertices. We better have exactly one vertex in I for every triangle. (Two vertices in the same triangle are always adjacent.) Let us fix an assignment that satisfies all the literals that correspond to vertices of I. (Assign values to the other variables arbitrarily.) This is a consistent rule to generate an assignment, because we cannot have a literal and its negation in the independent set). Finally, we note how every clause is satisfied by this assignment.

Wrapping up:

- We showed a reduction $\phi \rightarrow (G_\phi, m)$ that given an instance of 3SAT produces an instance of the decision version of Maximum Independent Set.
- We have the property that ϕ is satisfiable (answer YES for the 3SAT problem) if and only if G_ϕ has an independent set of size at least m.
- We knew 3SAT is NP-hard.
- Then also Max Independent Set is NP-hard; and so also NP-complete.

5.2 Maximum Clique

Given a (undirected non-weighted) graph $G = (V, E)$, a *clique* K is a set of vertices $K \subseteq V$ such that *any two* vertices in K are adjacent. In the Maximum Clique problem, given a graph G we want to find a largest clique.

In the decision version, given G and a parameter k, we want to know whether or not G contains a clique of size at least k. It should be clear that the problem is in NP.

We can prove that Maximum Clique is NP-hard by reduction from Maximum Independent Set. Take a graph G and a parameter k, and consider the graph G', such that two vertices in G' are connected by an edge if and only if they are not connected by an edge in G. We can observe that every independent set in G is a clique in G', and every clique in G' is an independent set in G. Therefore, G has an independent set of size at least k if and only if G' has a clique of size at least k.

5.3 Minimum Vertex Cover

Given a (undirected non-weighted) graph $G = (V, E)$, a *vertex cover* C is a set of vertices $C \subseteq V$ such that for every edge $(u, v) \in E$, either $u \in C$ or $v \in C$ (or, possibly, both). In the MINIMUM VERTEX COVER problem, given a graph G we want to find a smallest vertex cover.

In the decision version, given G and a parameter k, we want to know whether or not G contains a vertex cover of size at most k. It should be clear that the problem is in NP.

We can prove that Minimum Vertex Cover is NP-hard by reduction from Maximum Independent Set. The reduction is based on the following observation:

LEMMA 4
If I is an independent set in a graph $G = (V, E)$, then the set of vertices $C = V - I$ that are not *in I is a vertex cover in G. Furthermore, if C is a vertex cover in G, then $I = V - C$ is an independent set in G.*
PROOF: Suppose C is not a vertex cover: then there is some edge (u, v) neither of whose endpoints is in C. This means both the endpoints are in I and so I is not an independent set, which is a contradiction. For the "furthermore" part, suppose I is not an independent set: then there is some edge $(u, v) \in E$ such that $u \in I$ and $v \in I$, but then we have an edge in E neither of whose endpoints are in C, and so C is not a vertex cover, which is a contradiction. $\Box$

Now the reduction is very easy: starting from an instance (G, k) of Maximum Independent set we produce an instance $(G, n - k)$ of Minimum Vertex Cover.