UC Berkeley—CS 170: Efficient Algorithms and Intractable Problems Handout 19
Lecturer: Michael Jordan November 23, 2005

Notes 19 for CS 170

1 Tractable and Intractable Problems

So far, almost all of the problems that we have studied have had complexities that are *polynomial*, i.e., whose running time $T(n)$ has been $O(n^k)$ for some fixed value of k. Typically k has been small, 3 or less. We will let **P** denote the class of all problems whose solution can be computed in polynomial time, i.e., $O(n^k)$ for some fixed k, whether it is 3, 100, or something else. We consider all such problems efficiently solvable, or *tractable*. Notice that this is a very relaxed definition of tractability, but our goal in this lecture and the next few ones is to understand which problems are *intractable*, a notion that we formalize as *not being solvable in polynomial time*. Notice how *not being in* **P** is certainly a strong way of being intractable.

We will focus on a class of problems, called the *NP-complete problems*, which is a class of very diverse problems, that share the following properties: we only know how to solve those problems in time much larger than polynomial, namely *exponential time*, that is $2^{O(n^k)}$ for some k; and if we could solve one NP-complete problem in polynomial time, then there is a way to solve *every* **NP**-complete problem in polynomial time.

There are two reasons to study NP-complete problems. The practical one is that if you recognize that your problem is NP-complete, then you have three choices:

1. you can use a known algorithm for it, and accept that it will take a long time to solve if n is large;

2. you can settle for *approximating* the solution, e.g., finding a nearly best solution rather than the optimum; or

3. you can change your problem formulation so that it is in **P** rather than being NP-complete, for example by restricting to work only on a subset of simpler problems.

Most of this material will concentrate on recognizing NP-complete problems (of which there are a large number, and which are often only slightly different from other, familiar, problems in **P**) and on some basic techniques that allow to solve some NP-complete problems in an *approximate* way in polynomial time (whereas an exact solution seems to require exponential time).

The other reason to study NP-completeness is that one of the most famous open problem in computer science concerns it. We stated above that "we *only know* how to solve NP-complete problems in time much larger than polynomial" not that we *have proven* that NP-complete problems require exponential time. Indeed, this is the million dollar question,[1] one of the most famous open problems in computer science, the question whether "**P** = **NP**?", or whether the class of NP-complete problems have polynomial time solutions. After

[1] This is not a figure of speech. See `http://www.claymath.org/prizeproblems`.

decades of research, everyone believes that **P**$\neq$**NP**, i.e., that no polynomial-time solutions for these very hard problems exist. But no one has proven it. If you do, you will be very famous, and moderately wealthy.

So far we have not actually defined what NP-complete problems are. This will take some time to do carefully, but we can sketch it here. First we define the larger class of problems called **NP**: these are the problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time. For example, suppose the problem is to answer the question "Does a graph have a simple path of length $|V|$?". If someone hands you a path, i.e., a sequence of vertices, and you can *check* whether this sequence of vertices is indeed a path and that it contains all vertices in polynomial time, then the problem is in **NP**. It should be intuitive that any problem in **P** is also in **NP**, because are all familiar with the fact that checking the validity of a solution is easier than coming up with a solution. For example, it is easier to get jokes than to be a comedian, it is easier to have average taste in books than to write a best-seller, it is easier to read a textbook in a math or theory course than to come up with the proofs of all the theorems by yourself. For all this reasons (and more technical ones) people believe that **P**$\neq$**NP**, although nobody has any clue how to prove it. (But once it will be proved, it will probably not be too hard to understand the proof.)

The NP-complete problems have the interesting property that if you can solve any one of them in polynomial time, then you can solve *every* problem in **NP** in polynomial time. In other words, they are at least as hard as any other problem in **NP**; this is why they are called *complete*. Thus, if you could show that *any one* of the NP-complete problems that we will study *cannot* be solved in polynomial time, then you will have not only shown that **P**$\neq$**NP**, but also that none of the **NP**-compete problems can be solved in polynomial time. Conversely, if you find a polynomial-time algorithm for just one NP-complete problem, you will have shown that **P**=**NP**.[2]

2 Decision Problems

To simplify the discussion, we will consider only problems with Yes-No answers, rather than more complicated answers. For example, consider the *Traveling Salesman Problem* (TSP) on a graph with nonnegative integer edge weights. There are two similar ways to state it:

1. Given a weighted graph, what is the minimum length cycle that visits each node exactly once? (If no such cycle exists, the minimum length is defined to be ∞.)

2. Given a weighted graph and an integer K, is there a cycle that visits each node exactly once, with weight at most K?

Question 1 above seems more general than Question 2, because if you could answer Question 1 and find the minimum length cycle, you could just compare its length to K to answer Question 2. But Question 2 has a Yes/No answer, and so will be easier for us to consider. In

[2] Which still entitles you to the million dollars, although the sweeping ability to break every cryptographic protocol and to hold the world banking and trading systems by ransom might end up being even more profitable.

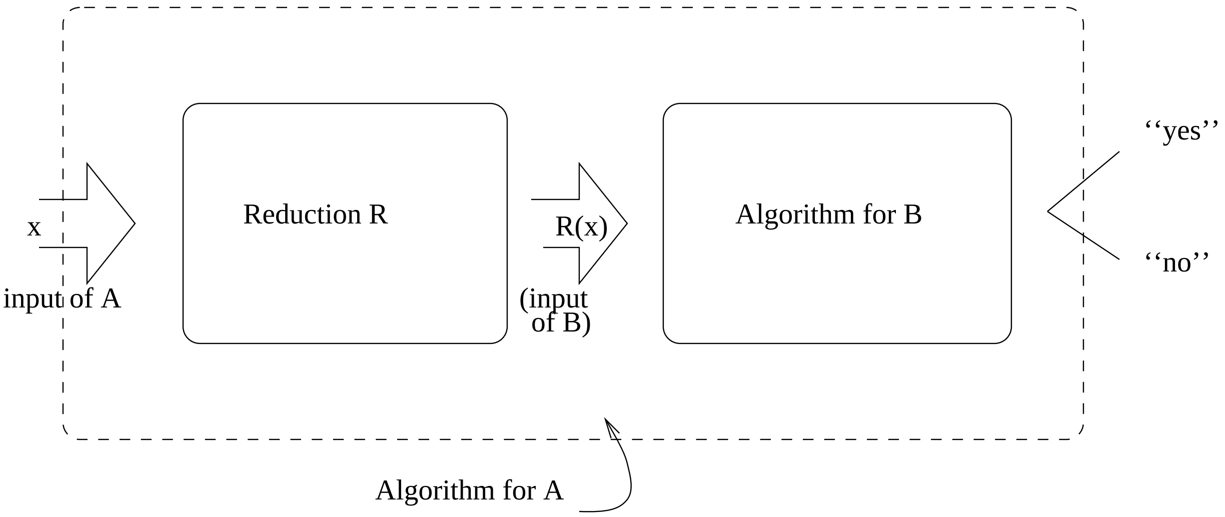

Figure 1: A reduction.

particular, if we show that Question 2 is NP-complete (it is), then that means that Question 1 is at least as hard, which will be good enough for us.[3]

Another example of a problem with a Yes-No answer is *circuit satisfiability* (which we abbreviate CSAT). Suppose we are given a Boolean circuit with n Boolean inputs $x_1, ..., x_n$ connected by AND, OR and NOT gates to one output x_{out}. Then we can ask whether there is a set of inputs (a way to assign True or False to each x_i) such that x_{out} =True. In particular, we will not ask what the values of x_i are that make x_{out} True.

If A is a Yes-No problem (also called a *decision* problem), then for an input x we denote by $A(x)$ the right Yes-No answer.

3 Reductions

Let A and B be two problems whose instances require Yes/No answers, such as TSP and CSAT. A *reduction* from A to B is a polynomial-time algorithm R which transforms inputs of A to equivalent inputs of B. That is, given an input x to problem A, R will produce an input $R(x)$ to problem B, such that x is a "yes" input of A if and only if $R(x)$ is a "yes" input of B. In a compact notation, if R is a reduction from A to B, then for every input x we have $A(x) = B(R(x))$.

A reduction from A to B, together with a polynomial time algorithm for B, constitute a polynomial algorithm for A (see Figure1). For any input x of A of size n, the reduction R takes time $p(n)$—a polynomial—to produce an equivalent input $R(x)$ of B. Now, this input $R(x)$ can have size at most $p(n)$—since this is the largest input R can conceivably construct in $p(n)$ time. If we now submit this input to the assumed algorithm for B, running in time $q(m)$ on inputs of size m, where q is another polynomial, then we get the right answer of x, within a total number of steps at most $p(n) + q(p(n))$—also a polynomial!

[3] It is, in fact, possible to prove that Questions 1 and 2 are equally hard.

We have seen many reductions so far, establishing that problems are easy (e.g., from matching to max-flow). In this part of the class we shall use reductions in a more sophisticated and counterintuitive context, in order to prove that certain problems are hard. If we reduce A to B, we are essentially establishing that, *give or take a polynomial, A is no harder than B.* We could write this as

$$A \leq B$$

an inequality between the complexities of the two problems. If we know B is easy, this establishes that A is easy. If we know A is hard, this establishes B is hard. It is this latter implication that we shall be using soon.

4 Definition of Some Problems

Before giving the formal definition of **NP** and of **NP**-complete problem, we define some problems that are **NP**-complete, to get a sense of their diversity, and of their similarity to some polynomial time solvable problems.

In fact, we will look at pairs of very similar problems, where in each pair a problem is solvable in polynomial time, and the other is presumably not.

- **minimum spanning tree:** Given a weighted graph and an integer K, is there a tree that connects all nodes of the graph whose total weight is K or less?
- **travelling salesman problem:** Given a weighted graph and an integer K, is there a cycle that visits all nodes of the graph whose total weight is K or less?

Notice that we have converted each one of these familiar problems into a decision problem, a "yes-no" question, by supplying a goal K and asking if the goal can be met. Any optimization problem can be so converted

If we can solve the optimization problem, we can certainly solve the decision version (actually, the converse is in general also true). Therefore, proving a negative complexity result about the decision problem (for example, proving that it cannot be solved in polynomial time) immediately implies the same negative result for the optimization problem.

By considering the decision versions, we can study optimization problems side-by-side with decision problems (see the next examples). This is a great convenience in the theory of complexity which we are about to develop.

- **Eulerian graph:** Given a directed graph, is there a closed path that visits each edge of the graph exactly once?
- **Hamilitonian graph:** Given a directed graph, is there a closed path that visits each *node* of the graph exactly once?

A graph is Eulerian if and only if it is strongly connected and each node has equal in-degree and out-degree; so the problem is squarely in **P** There is no known such characterization—or algorithm—for the Hamilton problem (and notice its similarity with the TSP).

- **circuit value:** Given a Boolean circuit, and its inputs, is the output T?
- **circuit SAT:** Given a Boolean circuit, is there a way to set the inputs so that the output is T? (Equivalently: If we are given *some* of its inputs, is there a way to set the remaining inputs so that the output is T.)

We know that **circuit value** is in **P**: also, the naïve algorithm for that evaluates all gates bottom-up is polynomial. How about **circuit SAT?** There is no obvious way to solve this problem, sort of trying all input combinations for the unset inputs—and this is an exponential algorithm.

General circuits connected in arbitrary ways are hard to reason about, so we will consider them in a certain standard form, called *conjunctive normal form (CNF)*: Let $x_1, ..., x_n$ be the input Boolean variables, and x_{out} be the output Boolean variable. Then a Boolean expression for x_{out} in terms of $x_1, ..., x_n$ is in CNF if it is the AND of a set of *clauses*, each of which is the OR of some subset of the set of *literals* $\{x_1, ..., x_n, \neg x_1, ..., \neg x_n\}$. (Recall that "conjunction" means "and", whence the name CNF.) For example,

$$x_{out} = (x_1 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee \neg x_1) \wedge (x_1 \vee x_2) \wedge (x_3)$$

is in CNF. This can be translated into a circuit straightforwardly, with one gate per logical operation. Furthermore, we say that an expression is in **2-CNF** if each clause has two distinct literals. Thus the above expression is not 2-CNF but the following one is:

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

3-CNF is defined similarly, but with 3 distinct literals per clause:

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

- **2SAT:** Given a Boolean formula in **2-CNF**, is there a satisfying truth assignment to the input variables?
- **3SAT:** Given a Boolean formula in **3-CNF** is there a satisfying truth assignment to the input variables?

2SAT can be solved by graph-theoretic techniques in polynomial time. For **3SAT**, no such techniques are available, and the best algorithms known for this problems are exponential in the worst case, and they run in time roughly $(1.4)^n$, where n is the number of variables. (Already a non-trivial improvement over 2^n, which is the time needed to check all possible assignments of values to the variables.)

- **matching:** Given a boys-girls compatibility graph, is there a complete matching?
- **3D matching:** Given a boys-girls-homes compatibility relation (that is, a set of boy-girl-home "triangles"), is there a complete matching (a set of disjoint triangles that covers all boys, all girls, and all homes)?

We know that matching can be solved by a reduction to max-flow. For **3D matching** there is a reduction too. Unfortunately, the reduction is *from 3SAT to* **3D matching**—and this is bad news for **3D matching**. . .

- **unary subset-sum:** Given integers $a_1, \ldots, a_n$, and another integer K *in unary*, is there a subset of these integers that sum exactly to K?
- **subset-sum:** Given integers $a_1, \ldots, a_n$, and another integer K *in binary*, is there a subset of these integers that sum exactly to K?

unary subset-sum is in **P**—simply because the input is represented so wastefully, with about $n + K$ bits, so that a $O(n^2K)$ dynamic programming algorithm, which would be exponential *in the length of the input* if K were represented in binary, is bounded by a polynomial in the length of the input. There is no polynomial algorithm known for the real **subset-sum** problem. This illustrates that you have to represent your input in a sensible way, binary instead of unary, to draw meaningful conclusions.

5 NP, NP-completeness

Intuitively, a problem is in **NP** if it can be formulated as the problem of whether there is a *certificate*:

- They are *small*. In each case the certificate would never have to be longer than a polynomial in the length of the input.
- They are *easily checkable*. In each case there is a polynomial algorithm which takes as inputs the input of the problem and the alleged certificate, and checks whether the certificate is a valid one for this input. In the case of **3SAT**, the algorithm would just check that the truth assignment indeed satisfies all clauses. In the case of *Hamilton cycle* whether the given closed path indeed visits every node once. And so on.
- Every "yes" input to the problem has at least one certificate (possibly many), and each "no" input has none.

Not all decision problems have such certificates. Consider, for example, the problem **non-Hamiltonian graph:** Given a graph G, is it true that there is no Hamilton cycle in G? How would you prove to a suspicious person that a given large, dense, complex graph has *no* Hamilton cycle? Short of listing all cycles and pointing out that none visits all nodes once (a certificate that is certainly not succinct)?

These are examples of problems in NP:

- Given a graph G and an integer k, is there a simple path of length at least k in G?
- Given a set of integers $a_1, \ldots, a_n$, is there a subset S of them such that $\sum_{a \in S} a = \sum_{a \notin S} a$?

We now come to the formal definition.

Definition 1 *A problem A is* **NP** *if there exist a polynomial p and a polynomial-time algorithm $V()$ such that x is a YES-input for problem A if and only if there exists a certificate y, with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.*

We also call **P** the set of decision problems that are solvable in polynomial time. Observe every problem in **P** is also in **NP**.

We say that a problem A is **NP**-hard if for every N in **NP**, N is reducible to A, and that a problem A is **NP**-complete if it is **NP**-hard *and* it is contained in **NP**. As an exercise to understand the formal definitions, you can try to prove the following simple fact, that is one of the fundamental reasons why **NP**-completeness is interesting.

Lemma 1
*If A is **NP**-complete, then A is in **P** if and only if **P** =**NP**.*

So now, if we are dealing with some problem A that we can prove to be **NP**-complete, there are only two possibilities:

- A has no efficient algorithm.
- All the infinitely many problems in **NP**, including factoring and all conceivable optimization problems are in **P**.

If **P** =**NP**, then, given the statement of a theorem, we can find a proof in time polynomial in the number of pages that it takes to write the proof down.

If it was so easy to find proof, why do papers in mathematics journal have theorems *and* proofs, instead of just having theorems. And why theorems that had reasonably short proofs have been open questions for centuries? Why do newspapers publish solutions for crossword puzzles? If **P** =**NP**, whatever exists can be found efficiently. It is too bizarre to be true.

In conclusion, it is safe to assume **P** $\neq$ **NP**, or at least that the contrary will not be proved by anybody in the next decade, and it is *really* safe to assume that the contrary will not be proved by us in the next month. So, if our short-term plan involves finding an efficient algorithm for a certain problem, and the problem turns out to be **NP**-hard, then we should change the plan.

6 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 2
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 $x = (x_1, \ldots, x_n)$ of length n, we have

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

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 $p(n)$, or equivalently, no register/memory cell needs to hold more than $\lg p(n) = O(\log n)$ bits. 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). □

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 $\mathit{length}(y) \leq p_A(\mathit{length}(x))$, 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 $x \in A$.

7 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 3
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. □

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 4
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.

8 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.

9 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

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

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

$$(x_1 \vee x_2 \vee y_1) \wedge (\bar{y}_1 \vee x_3 \vee y_2) \wedge (\bar{y}_2 \vee x_4 \vee y_3) \wedge \cdots \wedge (\bar{y}_{k-3} \vee x_{k-1} \vee x_k)$$

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.

10 Some NP-complete Graph Problems

10.1 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 for which one is the complement of the other.

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.

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 $\phi \to (G_\phi, m)$ 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.

10.2 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.

10.3 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 5
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. □

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.

Some NP-complete Numerical Problems

10.4 Subset Sum

The **Subset Sum** problem is defined as follows:

- Given a sequence of integers $a_1, \ldots, a_n$ and a parameter k,
- Decide whether there is a subset of the integers whose sum is exactly k. Formally, decide whether there is a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = k$.

Subset Sum is a true *decision problem*, not an optimization problem forced to become a decision problem. It is easy to see that Subset Sum is in NP.

We prove that Subset Sum is NP-complete by reduction from Vertex Cover. We have to proceed as follows:

- Start from a graph G and a parameter k.
- Create a sequence of integers and a parameter k'.
- Prove that the graph has vertex cover with k vertices iff there is a subset of the integers that sum to k'.

Let then $G = (V, E)$ be our input graph with n vertices, and let us assume for simplicity that $V = \{1, \ldots, n\}$, and let k be the parameter of the vertex cover problem.

We define integers $a_1, \ldots, a_n$, one for every vertex; and also integers $b_{(i,j)}$, one for every edge $(i, j) \in E$; and finally a parameter k'. We will define the integers a_i and $b_{(i,j)}$ so that if we have a subset of the a_i and the $b_{(i,j)}$ that sums to k', then: the subset of the a_i corresponds to a vertex cover C in the graph; and the subset of the $b_{(i,j)}$ corresponds to the edges in the graph such that exactly one of their endpoints is in C. Furthermore the construction will force C to be of size k.

How do we define the integers in the subset sum instance so that the above properties hold? We represent the integers in a matrix. Each integer is a row, and the row should be seen as the base-4 representation of the integer, with $|E| + 1$ digits.

The first column of the matrix (the "most significant digit" of each integer) is a special one. It contains 1 for the a_is and 0 for the $b_{(i,j)}$s.

Then there is a column (or digit) for every edge. The column (i, j) has a 1 in a_i, a_j and $b_{(i,j)}$, and all 0s elsewhere.

The parameter k' is defined as

$$k' := k \cdot 4^{|E|} + \sum_{j=0}^{|E|-1} 2 \cdot 4^i$$

This completes the description of the reduction. Let us now proceed to analyze it.

From Covers to Subsets Suppose there is a vertex cover C of size k in G. Then we choose all the integers a_i such that $i \in C$ and all the integers $b_{(i,j)}$ such that exactly one of i and j is in C. Then, when we sum these integers, doing the operation in base 4, we have a 2 in all digits except for the most significant one. In the most significant digit, we are summing one $|C| = k$ times. The sum of the integers is thus k'.

From Subsets to Covers Suppose we find a subset $C \subseteq V$ and $E' \subseteq E$ such that

$$\sum_{i \in C} a_i + \sum_{(i,j) \in E'} b_{(i,j)} = k'$$

First note that we never have a carry in the $|E|$ less significant digits: operations are in base 4 and there are at most 3 ones in every column. Since the $b_{(i,j)}$ can contribute at most one 1 in every column, and k' has a 2 in all the $|E|$ less significant digits, it means that for every edge (i, j) C must contain either i or j. So C is a cover. Every a_i is at least $4^{|E|}$, and k' gives a quotient of k when divided by $4^{|E|}$. So C cannot contain more than k elements.

10.5 Partition

The **Partition** problem is defined as follows:

- Given a sequence of integers $a_1, \ldots, a_n$.
- Determine whether there is a partition of the integers into two subsets such the sum of the elements in one subset is equal to the sum of the elements in the other.

  Formally, determine whether there exists $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = (\sum_{i=1}^{n} a_i)/2$.

Clearly, Partition is a special case of Subset Sum. We will prove that Partition is is NP-hard by reduction from Subset Sum.[4]

Given an instance of Subset Sum we have to construct an instance of Partition. Let the instance of Subset Sum have items of size $a_1, \ldots, a_n$ and a parameter k, and let $A = \sum_{i=1}^{n} a_i$.

Consider the instance of Partition $a_1, \ldots, a_n, b, c$ where $b = 2A - k$ and $c = A + k$.

Then the total size of the items of the Partition instance is $4A$ and we are looking for the existence of a subset of $a_1, \ldots, a_n, b, c$ that sums to $2A$.

It is easy to prove that the partition exists if and only if there exists $I \subseteq \{1, \ldots, n\}$ such that $\sum_i a_i = k$.

10.6 Bin Packing

The **Bin Packing** problem is one of the most studied optimization problems in Computer Science and Operation Research, possibly the second most studied after TSP. It is defined as follows:

- Given items of size $a_1, \ldots, a_n$, and given unlimited supply of bins of size B, we want to pack the items into the bins so as to use the minimum possible number of bins.

You can think of bins/items as being CDs and MP3 files; breaks and commercials; bandwidth and packets, and so on.

The decision version of the problem is:

- Given items of size $a_1, \ldots, a_n$, given bin size B, and parameter k,
- Determine whether it is possible to pack all the items in k bins of size B.

Clearly the problem is in NP. We prove that it is NP-hard by reduction from Partition.

Given items of size $a_1, \ldots, a_n$, make an instance of Bin Packing with items of the same size and bins of size $(\sum_i a_i)/2$. Let $k = 2$.

There is a solution for Bin Packing that uses 2 bins if and only if there is a solution for the Partition problem.

[4] The reduction goes in the non-trivial direction!