

## CDM

### Hardness and Completeness

Klaus Sutner  
Carnegie Mellon University  
www.cs.cmu.edu/~sutner

Hardness

1

### Battleplan

- Comparing Problems
- Polynomial Time Turing Reductions
- Polynomial Time Reductions
- NP-Hardness and Completeness
- Cook-Levin Theorem

Hardness

2

### Classifying Problems

Ideally, one would like to be able to pinpoint the complexity of a algorithmic problem by giving an upper and lower bound for the resources needed so solve it. For example, with regard to time complexity one would like results like:

- The Bandersnatch Problem can be solved in time  $O(n^3)$ .
- Furthermore, the Bandersnatch Problem requires time  $\Omega(n^3)$ .

Unfortunately, such tight bounds are often impossible to obtain.

Instead, one has to rely on results of the following kind:

- The Bandersnatch Problem lies in complexity class  $\mathcal{C}$ .
- If it also lies in class  $\mathcal{C}'$  then all hell will break lose.

We will shortly make clear what kind of hell we have in mind.

Hardness

3

### How About $\mathbb{P}$ versus $\mathbb{NP}$ ?

Recall the list of problems in  $\mathbb{NP}$  from last time.

- Vertex Cover
- Traveling Salesman
- Hamiltonian Cycle
- Polynomial Equations over  $\mathbb{Z}_2$
- Pebbling

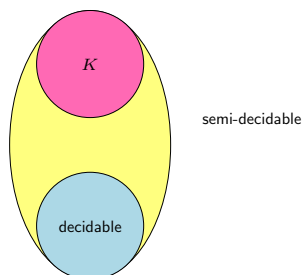
No interesting lower bounds are known for any of these problems, though they all appear to require something like exponential time.

While there are no lower bounds, we still claim that within  $\mathbb{NP}$  these problems play a role similar to the Halting set  $K$  in the semi-decidable sets.

Hardness

4

### The Classical World of Complexity



$K$  is the Halting Problem.

Hardness

5

### A Strict Hierarchy

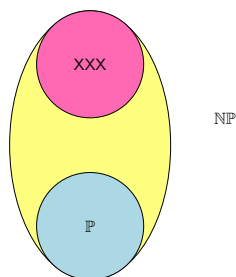
The point here is that we can **prove** that  $K$  is undecidable (Cantor style diagonalization), the picture is not just a conjecture or a suggestion.

In fact one can show that there is space between  $K$  and the decidable problems: There provably are intermediate problems:

Problems that are undecidable, but not as difficult as the full Halting problem.

The proof of this fact is rather difficult (Friedberg-Muchnik priority method), see below.

## The Probable World of (Nearly) Feasible Computation



Here XXX stands for any of our  $\text{NP}$  problems such as Vertex Cover.

## The Difference

There is one crucial difference: At the time of this writing (2005/08/28 09:30:29) no one can prove that Vertex Cover is not in  $\text{P}$ .

So how can we convince ourselves that these problems really cannot be handled in polynomial time?

We need to exploit another analogy:  $K$  is the most complicated r.e. set because every other r.e. set can be *reduced* to it: for every r.e. set  $A$  there is a recursive function  $f$  such that

$$x \in A \iff f(x) \in K.$$

This is the key property of  $K$  that can be scaled down to  $\text{NP}$ .

First, a slightly more general look at methods that allow comparisons of the relative difficulty of problems.

## Comparing Problems

We need to compare the relative difficulty of problems.

The basic idea is simple (but won't work for  $\text{NP}$  in the end): problem  $A$  is easier than  $B$  if we could use an algorithm for problem  $B$  to construct an algorithm for  $A$ .

**Example 1. Unique:** Given a list of items, eliminate all duplicates.

The brute force algorithm takes  $O(n^2)$  steps where  $n$  is the length of the list.

```
for i = 1 .. n do
  for j = i+1 .. n do
    if ( a[i] == a[j] )
      remove a[j]
```

## Reduction to Sorting

We can speed this up significantly if the items are ordered and easily comparable (integers for example are fine).

- First sort the list, and then
- scan the sorted list to eliminate adjacent duplicates.

The scan is linear time, and sorting takes time  $O(n \log n)$ , so this is much better than brute force. In fact, if we use a linear time sorting method the whole running time is just linear (but careful, we have to count bits; logarithmic size model).

The important point is that the sorting algorithm is used as a subroutine: using a sorting algorithm we can easily implement an algorithm for Unique.

## Recall: Oracles and Reductions

We have already seen machinery that allows one to compare problems according to their difficulty in classical computability theory: Turing reductions, many-one and one-one reductions.

In order to adapt these to a resource-bounded environment we need to impose bounds on the amount of computation that can occur in the reduction itself.

For example, if we use an oracle Turing machine, we require this machine to run in polynomial time.

Note that in a computation with oracle  $B$  we only charge for the steps taken by the ordinary part of the Turing machine, the solutions provided by the oracle do not count towards the time complexity.

It seems reasonable to scale things down from CCT to complexity theory by imposing polynomial time bounds wherever appropriate.

## Not So Fast . . .

The classical machinery rests on two essential facts, clearly recognized by Turing:

- One can effectively list all (say, one-tape) Turing machines, and the listing can be handled by a Turing machine.
- There exists a universal Turing machine that can simulate the machines in this listing.

We can easily scale the listing by attaching an explicit clock to the machine. A popular choice is to have the computation of machine  $M_k$  be automatically truncated after  $n^k + k$  steps.

Then

$$\text{P} = \{ \mathcal{L}(M_k) \mid k \geq 0 \}$$

and a similar enumeration of nondeterministic machines produces

$$\text{NP} = \{ \mathcal{L}(N_k) \mid k \geq 0 \}.$$

## Universal Machines

There is a universal, deterministic machine  $U$  that simulates  $M_k$  with only a polynomial slowdown. More precisely,  $U$  on input  $e\#x$  simulates  $M_e$  on  $x$  in running time  $q(n^c + e)$  where  $q$  is some polynomial (even low degree).

But note that  $U$  itself is not polynomial time, the running time increases for different choices of  $e$ .

Likewise, there is a universal, nondeterministic machine  $U'$  that simulates  $N_k$  with only a polynomial slowdown in the same sense as above. Again,  $U'$  itself is not polynomial time.

So, one has to be a bit careful where the polynomial time bounds should go.

## Polynomial Time Turing Reductions

At any rate, consider two combinatorial problems  $A$  and  $B$ .

**Definition 1.**  $A$  is **polynomial time Turing reducible** to  $B$  if a solution for  $A$  can be computed in polynomial time using  $B$  as an oracle.

In symbols,  $A \leq_T^p B$ .

In other words, our algorithm performs polynomially many steps, plus gets solutions for instances for  $B$  for free.

While we are mostly interested in decision problems, this makes perfect sense for function and search problems. In fact, the examples above are all of the more general type.

## Example 1

**Proposition 1.** *The decision version of Vertex Cover is polynomial time Turing reducible to the function version.*

This is not hard:

- On input  $G$  and  $k$  hand over  $G$  to the oracle.
- Get back a minimal vertex cover  $C$ .
- Check whether  $|C| \leq k$ .

This is clearly linear time in the size of the input – recall that we charge nothing for the work of the oracle.

## Example 2

**Proposition 2.** *The function version of Vertex Cover is polynomial time Turing reducible to the decision version.*

This requires some work. Let  $n$  be the number of vertices in  $G$ , say,  $V = \{v_1, \dots, v_n\}$ .

- First, do a search on  $k = 1, \dots, n$  to find the size  $k_0$  of a minimal cover, using the oracle for the decision problem.
- Then find the least  $i$  such that  $G - v_i$  has a cover of size  $k_0 - 1$ . Place  $v_i$  into  $C$  and remove it from  $G$ . Recurse to produce a full cover.

This is no longer linear but still polynomial time.

**Exercise 1.** *Determine the exact running time of this oracle algorithm.*

## Simplify

The last argument is fairly typical: many function and search problems can be rephrased as decision problems without losing much: they are polynomial time Turing reducible to the decision version.

Hence for difficult problems we can focus on the decision version and still get a good idea what the complexity of the more natural function/search version is.

BTW: I could not figure out how to reduce the function version of the Pebbling Problem to the decision version. Excellent extra credit project.

## Properties

So let's focus on decision problems from now on.

**Proposition 3.** *Polynomial time Turing reducibility is a quasi-order (reflexive and transitive).*

For transitivity, this works since polynomials are closed under substitution. Hence we can from equivalence classes as usual, the polynomial time Turing degrees. We won't pursue this idea here.

**Proposition 4.**  $B \in \mathbb{P}$  and  $A \leq_T^p B$  implies  $A \in \mathbb{P}$ .

Here we can simply replace the oracle by a real polynomial algorithm.

### So How About NP?

Our polynomial time Turing reductions are very natural, but a bit too powerful for NP: it seems that co-NP is different from NP, but these reductions do not distinguish between a set and its complement.

$$\overline{A} \leq_T^p A \quad \text{and} \quad A \leq_T^p \overline{A}$$

for any set  $A$ .

So  $B \in \text{NP}$  and  $A \leq_T^p B$  is not known to imply  $A \in \text{NP}$ , which is a bit awkward.

In fact, there is no reason to assume that NP is closed under complementation: it appears rather likely that  $\text{NP} \neq \text{co-NP}$ .

### Weak Reductions

As before in CRT, we can consider weaker reductions instead.

Let  $A$  and  $B$  be two decision problems, with Yes-instances  $Y_A$  and  $Y_B$ , respectively.

**Definition 2.**  $A$  is **polynomial time reducible** to  $B$  if there is a polynomial time computable function  $f$  such that  $x \in Y_A \leftrightarrow f(x) \in Y_B$ .

Notation:  $A \leq_m^p B$ .

Don't confuse this with polynomial time Turing reductions  $A \leq_T^p B$ .

Another plausible reduction is based on linear time computable functions.

### Closure Properties

**Lemma 1.**

- If  $B$  is in  $\mathbb{P}$  and  $A \leq_m^p B$  then  $A$  is also in  $\mathbb{P}$ .
- If  $B$  is in NP and  $A \leq_m^p B$  then  $A$  is also in NP.

*Proof.*

- (1) Replace the oracle  $B$  by a polynomial time algorithm.
- (2) Combine the transducer for the reduction and the nondeterministic acceptor for  $B$ : produces a nondeterministic polynomial time acceptor for  $A$ .

□

### Hardness and Completeness

**Definition 3.**

$B$  is **NP-hard** if for all  $A$  in NP:  $A \leq_m^p B$ .

$B$  is **NP-complete** if  $B$  is in NP and is also NP-hard.

It is not at all clear that NP-complete sets exist, but note the following:

**Proposition 5.** If  $B$  in NP-complete and  $B$  is in  $\mathbb{P}$  then  $\mathbb{P} = \text{NP}$ .

If indeed  $\mathbb{P} \neq \text{NP}$  as expected, then no NP-complete problem can have a polynomial time algorithm.

### Constructing an NP-complete Problem

How do we get our hands on an NP-complete problem?

It is tempting to use the universal machine  $U$  that can simulate machines in the enumeration  $(N_e)$  of nondeterministic, polynomial time Turing machines already mentioned.

We want to scale down  $K$ , the Halting set.

The first attempt would be to use

$$K' = \{ e\#x \mid x \text{ accepted by } N_e \} \subseteq \Sigma^*$$

It is easy to see that  $K'$  is NP-hard, but there is no reason why it should be in NP; simulation of  $N_e$  is not a task that can be handled within a fixed polynomial time bound.

So simple Cantor-style diagonalization fails here.

### An Ugly Set

We can fix the problem by padding the input so that we can compensate for the running time of  $N_e$ .

$$K' = \{ e\#x\#1^t \mid x \text{ accepted by } N_e \text{ in } t = |x|^c + |x| \text{ steps} \} \subseteq \Sigma^*$$

**Proposition 6.**  $K'$  is in NP.

*Proof.*

To see this note that the slowdown by  $U$  is polynomial, say, the simulation takes  $q(n^c + e)$  steps.

But then  $U$  can test, in time polynomial in  $|e\#x\#1^t| = |x| + t + c$ , whether  $N_e$  indeed accepts  $x$ .

□

### Hardness

**Proposition 7.**  $K'$  is NP-hard.

*Proof.*

Consider  $A = \mathcal{L}(N_e) \in \text{NP}$  arbitrary. Then the function

$$x \mapsto e\#x\#1^t \quad \text{where } t = |x|^e + |x|$$

is polynomial time computable and shows that  $A \leq_m^p K'$ .

□

Hence,  $K'$  is indeed NP-complete.

### It Works, But . . .

So we have the desired theorem.

**Theorem 1.** *There is an NP-complete problem.*

Alas, this result is perfectly useless when it comes to our list of NP problems: they bear no resemblance whatsoever to  $K'$ .

We have a foothold in the world of NP-completeness, but to show that one of these natural problems is NP-complete we would have to find a reduction from  $K'$  to, say, Pebbling or Vertex Cover.

Good luck on that.

### Boolean Circuits to the Rescue

How about the circuits we encountered in SLP and Pebbling?

Arithmetic circuits are a bit too specialized, but we could use Boolean circuits instead.

So the data are just single bits, and the operations are just the standard Boolean operations

$$\wedge, \vee \text{ and } \neg.$$

Since any digital computer can ultimately be implemented in terms of these operations (in a manner of speaking, at least) there is good hope that we still wind up with a problem that is expressive enough to be NP-hard.

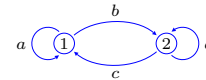
On the other hand, verifying that such a circuit produces some output given the input is clearly polynomial time.

So there is hope . . .

### Example: Transitive Closure

As a small example, consider a circuit that computes the transitive closure of a  $2 \times 2$  Boolean matrix.

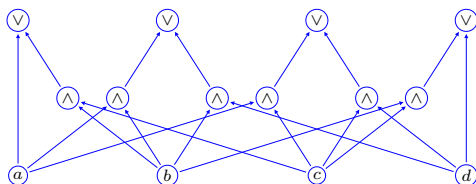
The matrix  $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$  can be considered as the adjacency matrix for a subgraph of



We need a Boolean circuit with 4 inputs and 4 outputs corresponding to the entries in  $A^*$ .

### The Closure Circuit

The output nodes are on top.



For the reflexive transitive closure we can set  $a = d = 1$ .

### Evaluating a Circuit

A Boolean circuit can be modeled as a directed acyclic graph.

The indegree 0 nodes are the input nodes, and the outdegree 0 nodes are the output nodes.

Given a Boolean circuit with input nodes  $x_1, \dots, x_n$  and an assignment  $\alpha$  of truth values to these nodes we can easily compute the values of the output nodes in polynomial time (even in linear time).

But what if we don't have an assignment of truth values?

What if we ask whether an assignment exists that produce true at the output nodes?

## History

For historical reasons one often talks about a *Boolean formula* or a *Boolean expression* rather than a circuit. There is really no difference, except that formulae fit more nicely into the classical framework of logic.

Hence, by a Boolean formula we mean an expression built from true, false, Boolean variables, and logical connectives  $\wedge$ ,  $\vee$  and  $\neg$ .

On occasion one also adds other Boolean operations such as implication or exclusive-or.

A truth assignment for a formula is a map that assigns a truth-value to each variable.

## Evaluation

Given a truth assignment for the variables, one can easily evaluate the formula in polynomial time.

For example, we can think of the formula as a DAG, and the values percolating up to the root.

This is very similar to the Pebbling game.

We write

$$\varphi[V]$$

for the value of  $\varphi$  under the assignment  $T$ .

## Satisfaction

### Definition 4.

Truth assignment  $V$  **satisfies**  $\varphi$  iff  $\varphi[V] = 1$ .

A formula  $\varphi$  is called a

- **tautology**:  $\varphi[V] = 1$  for all  $V$
- **contradiction**:  $\varphi[V] = 0$  for all  $V$
- **contingency** (or **satisfiable**):  $\varphi[V] = 1$  for some  $V$

For example,

$$(p \wedge q \rightarrow r) \wedge (\neg p \vee q) \rightarrow (p \rightarrow r)$$

is a tautology, but

$$(p \rightarrow \neg p) \wedge (\neg p \rightarrow p)$$

is a contradiction.

## Assorted Decision Problems

All these notions translate into corresponding decision problems. Notably

Problem: **Satisfiability**

Instance: A Boolean formula  $\varphi$ .

Question: Is  $\varphi$  satisfiable?

Problem: **Tautology**

Instance: A Boolean formula  $\varphi$ .

Question: Is  $\varphi$  a tautology?

## Some Time Estimates

It is easy to check in polynomial time if a given truth assignment satisfies a formula.

But for these problems we have to search over all truth assignment, an exponentially large set.

Baby case: 100 variables. This produces

$$1.27 \cdot 10^{30} \text{ truth assignments}$$

Assume universe is 15 billion years old, and we can check one assignment in a micro second. Then, since the Big Bang, we would have checked only

$$4.73 \cdot 10^{23} \text{ truth assignments}$$

Just about nothing.

## Cook-Levin Theorem

Of course, there might be some other, clever, structural approach to satisfiability testing. That seems unlikely in light of the following result.

**Theorem 2.** *Cook-Levin 1971/1973*

*The Satisfiability Problem is NP-complete.*

Membership in NP is easy using the standard guess-and-verify approach.

But hardness takes work: we have to express the computation of a Turing machine as a Boolean formula.

Note that the following proof should be read just once. Then throw it away and reconstruct your own proof.

### Proof Idea

Let  $A$  be an arbitrary set (of Yes-instances) in  $\text{NIP}$ .

There is a deterministic polynomial time Turing machine  $M$  such that  $M$  accepts  $(x, w)$  iff  $x \in A$  where  $w$  is some witness of length polynomial in  $n = |x|$ .

The idea is to construct a (rather large) Boolean formula  $\Phi_x$  such that

$$\Phi_x \text{ is satisfiable} \iff M \text{ accepts } (x, w) \text{ for some } w.$$

While the formula is fairly complicated, it can clearly be constructed in time polynomial in  $n$ .

It has lots of variables that express information about states, head position and tape inscriptions: the satisfying truth assignment translates directly into an accepting computation of  $M$ .

### Coding Time

First off, let  $N = p(n)$  be the running time of the machine.

If we have a list of Boolean variables

$$X_0, X_1, \dots, X_N$$

and a truth assignment  $V$  we can think of  $V(X_t)$  as the value of variable  $X$  at time  $t$ .

We can use logic to pin down the value of  $X_{t+1}$  in terms of  $X_t$  (and other variables).

$$\bigwedge_{t < N} X_{t+1} \iff \varphi(X_t, \dots)$$

### Coding Numbers

If we need to code a number  $r$  in a certain range, say  $1 \leq r \leq m$ , we can simply use variables

$$X(1), X(2), \dots, X(m)$$

plus a stipulation that exactly one of them is true under  $V$ :

$$\text{EO}_m(X(1), X(2), \dots, X(m))$$

Here

$$\text{EO}_k(x_1, \dots, x_k) = (x_1 \vee x_2 \vee \dots \vee x_k) \wedge \bigwedge_{1 \leq i < j \leq k} \neg(x_i \wedge x_j).$$

Note that the size of  $\text{EO}_k$  is  $O(k^2)$ .

### Coding Hardware

Combining these two ideas we can set up polynomially many Boolean variables

$$\text{states} \quad S_t(p), 0 \leq t \leq N, 1 \leq p \leq m,$$

$$\text{head position} \quad H_t(i), 0 \leq t, i \leq N,$$

$$\text{tape inscription} \quad C_t(i), 0 \leq t, i \leq N.$$

that express, for each time  $0 \leq t \leq N$ , which state the machine is in, where the head is, and what's on the tape.

For simplicity, we assume here that the only tape symbols are 0 and 1, it is not hard to deal with the general situation.

### Coding Moves

We then have to express the constraint that the variables change from time  $t$  to time  $t + 1$  only in accordance with the transition function of the Turing machine.

For example

$$\bigwedge_{t < N} H_t(i) \rightarrow \text{EO}_3(H_{t+1}(i-1), H_{t+1}(i), H_{t+1}(i+1))$$

$$\bigwedge_{t < N} S_t(p) \wedge H_t(i) \wedge \neg C_t(i) \rightarrow S_{t+1}(q_0)$$

$$\bigwedge_{t < N} S_t(p) \wedge H_t(i) \wedge C_t(i) \rightarrow S_{t+1}(q_1)$$

And so on and on.

### Start and Stop

Initially the input is on the first part of the tape

$$H_0(0) \wedge S_0(q_0) \wedge C_0(1) = x_1 \wedge C_0(2) = x_2 \wedge \dots \wedge C_0(n) = x_n$$

and at the end we accept:

$$S_N(q_Y)$$

Note that  $C_0(n+1), \dots, C_0(N)$  is not fixed and can be set arbitrarily by  $V$ .

It is not too hard to see that the whole formula  $\Phi_x$  in the end has size polynomial in  $n$ .

### And It Works

Now suppose  $\Phi_x$  is satisfied by truth assignment  $V$ .

Then  $M$  accepts the original tape inscription  $xw$  where  $x$  is the real input and  $w$  is the bits chosen freely by  $V$ .

Since  $\Phi_x$  forces the values of all the variables to correspond to a computation of  $M$  on input  $(x, w)$  we have the desired witness and  $x$  must be a Yes-instance.

Conversely, every witness plus corresponding accepting computation can be translated into a satisfying truth assignment  $V$ .

That's it.

### Improvements

The hardness of Satisfiability holds up even when the formulae in question are rather restricted.

**Definition 5.** A *literal* is a variable or negated variable. A formula is in **conjunctive normal form (CNF)** if it is a conjunction of disjunctions of literals:  $\Phi_1 \wedge \Phi_2 \wedge \dots \wedge \Phi_n$  where  $\Phi_i = z_{i,1} \vee z_{i,2} \vee \dots \vee z_{i,k(i)}$ ,  $z_{i,j}$  a literal.

It is in  $k$ -CNF if  $k(i) = 3$  for all  $i$ .

**Theorem 3.** The Satisfiability Problem is  $\text{NP}$ -complete for formulae in 3-CNF.

### Beachhead

The last result can be used to establish the  $\text{NP}$ -hardness of many problems such as Vertex Cover.

It now suffices to find a polynomial time computable function

$$f : 3\text{-CNF} \longrightarrow \text{Graphs, Integers}$$

such that  $\varphi$  in 3-CNF is satisfiable iff for  $f(\varphi) = (G, k)$  the graph  $G$  has a vertex cover of size  $k$ .

This is much, much easier than having to deal with general formulae.

### Vertex Cover

**Theorem 4.** Vertex Cover is  $\text{NP}$ -complete.

*Proof.*

Suppose we have a 3-CNF formula  $\Phi = \Phi_1 \wedge \Phi_2 \wedge \dots \wedge \Phi_m$  where  $\Phi_i = z_{i,1} \vee z_{i,2} \vee z_{i,3}$ .

The Boolean variables are  $x_1, \dots, x_n$ .

We start with a graph  $G'$  on  $2n + 3m$  vertices.

- Vertices:  $x_i, \bar{x}_i$  for  $i = 1, \dots, n$  and  $u_{i,1}, u_{i,2}, u_{i,3}$  for  $i = 1, \dots, m$ .
- Edges: one edge between  $x_i$  and  $\bar{x}_i$ , and three edges that turn  $u_{i,1}, u_{i,2}, u_{i,3}$  into a triangle.

### Proof, contd.

It is easy to see that every vertex cover of  $G'$  must have at least  $n + 2m$  vertices (one for each  $x$ -edge, and two for each triangle).

And such covers exist, lots of them: you can pick at random one of  $x_i$  or  $\bar{x}_i$ , and exactly two of  $u_{i,1}, u_{i,2}, u_{i,3}$ .

So far we have only used  $n$  and  $m$ , but not the formula itself.

Let  $G$  be the graph obtained by adding  $3m$  more edges to  $G'$ :

connect  $u_{i,j}$  to  $x_s$  if  $z_{i,j} = x_s$  and connect  $u_{i,j}$  to  $\bar{x}_s$  if  $z_{i,j} = \neg x_s$ .

Lastly, set the bound to  $k = n + 2m$ .

### Proof, contd.

**Claim.**  $G$  has a cover of size  $k = n + 2m$  iff the formula is satisfiable.

To see this, note that any cover  $C$  defines an assignment  $V$ :

$$V(x_i) = \begin{cases} 1 & \text{if } x_i \in C, \\ 0 & \text{otherwise.} \end{cases}$$

Then  $V$  satisfies the formula.

Conversely, every satisfying assignment translates into a cover.

□

### Important Points

For this construction to work we need two crucial ingredients:

- The graph  $G$  and the bound  $k$  can be computed from  $\Phi$  in polynomial time.
- $G$  has a vertex cover of size  $k$  if, and only if,  $\Phi$  is satisfiable.

Many other completeness proofs look very similar.

### Tautology

So Satisfiability is NP-complete. How about Tautology?

Note that  $\varphi$  is satisfiable iff  $\neg\varphi$  fails to be a tautology.

*Diré Warning:* This does not establish a polynomial time reduction between the two problems.

It does produce a polynomial time Turing reduction, but that type of reduction is too strong already, it clobbers NP.

All we can say easily is that Tautology is co-NP-complete.

### And In Reality?

While Satisfiability is hard for some carefully constructed formulae, it turns out to be perfectly tractable for "normal" formulae (Davis-Putnam algorithm and variants).

Since it is easy to express combinatorial problems as a satisfiability problem, good algorithms are quite valuable.

There are algorithms that routinely deal with formulae containing many thousands of variables.

More later.

### Summary

- The notion of hardness and completeness allows one to compare problems even when their exact complexity is unknown.
- Hardness results are strong indication that certain algorithms fail to exist.
- Many hundreds of combinatorial problems are known to be NP-complete.
- If you cannot find a fast algorithm for a problem, it is good style to at least provide a hardness argument.
- The complexity landscape seems to be very similar to the classical computability landscape, except that no one is able to prove it.