CS 3719 (Theory of Computation and Algorithms) – Lectures 23-32

Antonina Kolokolova*

March 2011

1 Scaling down to complexity

In real life, we are interested whether a problem can be solved efficiently; just knowing that something is decidable is not enough for practical purposes. The Complexity Theory studies languages from the point of view of efficient solvability. But what is efficient? The accepted definition in complexity theory equates efficiency with being computable in time polynomial in the input length (number of bits to encode a problem). So a problem is efficiently solvable (a language is efficiently decidable) if there exists an algorithm (such as a Turing machine algorithm) that solves it in time $O(n^d)$ for some constant d. Thus, an algorithm running in time $O(n^2)$ is efficient; an algorithm running in time $O(2^n)$ is not.

Recall that we defined, for some function $f : \mathbb{N} \to \mathbb{N}$, $\mathsf{Time}(f(n)) = \{L \mid L \text{ is decided by some TM in at most } f(n) \text{ steps }\}$. Now, a problem is efficiently solvable if $f(n)$ is $O(n^d)$ for some constant d.

Definition 16. *The class* P *of polynomial-time decidable languages is* $\mathsf{P} = \cup_{k\geq 1}\mathsf{Time}(n^k)$.

Another way of interpreting this definition is that P is the class of languages decidable in time comparable to the length of the input string . This is the class of problems we associate with being efficiently solvable. Another useful class to define is $\mathsf{EXP} = \cup_{k\geq 1}\mathsf{Time}(2^{n^k})$, the class of problems solvable in time comparable to the value of the string when treated as a number. For example, we don't know how to break a cryptographic encoding (think a password) in time less that password viewed as a number; however, given a number we can find if it is, say, divisible by 3 in time proportional to its binary encoding. Other examples of polynomial-time solvable problems are graph reachability (algorithms such as Depth First Search/Breadth First Search run in polynomial time), testing whether two numbers are relatively prime (Euclid's algorithm). Also, any regular language, and, moreover, any context-free language is decidable in polynomial time.

*The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364 and SFU CS 710.

2 The class NP

To go from computability to complexity we will augment the definitions of decidable, semi-decidable, co-semi-decidable and in general arithmetic hierarchy by bounding all quantifiers by a polynomial in the input length. Scaling down decidable, as we just did, we get P. Scaling down semi-decidable gives us NP; the efficient counterpart of co-semi-decidable is co-NP, and overall, arithmetic hierarchy (alternating quantifiers) scales down to polynomial-time hierarchy (polynomially-bounded alternating quantifiers). In this course we will concentrate on P, NP and the major open problem asking whether the two are distinct.

NP is a class of languages that contains all of P, but which most people think also contains many languages that aren't in P. Informally, a language L is in NP if there is a "guess-and-check" algorithm for L. That is, there has to be an efficient verification algorithm with the property that any $x \in L$ can be verified to be in L by presenting the verification algorithm with an appropriate, short "certificate" string y.

Remark: NP stands for "nondeterministic polynomial time", because one of the ways of defining the class is via nondeterministic Turing machines. NP does *not* stand for "not polynomial", and as we said, NP includes as a subset all of P. That is, many languages in NP are very simple: in particular, all regular languages and all context-free languages are in P and therefore in NP.

We now give the formal definition. For convenience, from now on we will assume that all our languages are over the fixed alphabet Σ, and we will assume $0, 1 \in \Sigma$.

Definition 17. *Let $L \subseteq \Sigma^*$. We say $L \in$NP if there is a two-place predicate $R \subseteq \Sigma^* \times \Sigma^*$ such that R is computable in polynomial time, and such that for some $c, d \in \mathbb{N}$ we have for all $x \in \Sigma^*$, $x \in L \Leftrightarrow \exists y \in \Sigma^*, |y| \leq c|x|^d$ and $R(x, y)$.*

We have to say what it means for a two-place predicate $R \subseteq \Sigma^* \times \Sigma^*$ to be computable in polynomial time. One way is to say that $\{\langle x, y\rangle \,|\, (x, y) \in R\} \in$ P, where $\langle x, y\rangle$ is our standard encoding of the pair x, y. Another, equivalent, way is to say that there is a Turing machine M which, if given $x\#y$ on its input tape, halts in time polynomial in $|x| + |y|$, and accepts if and only if $(x, y) \in R$.

Most languages that are in NP are easily shown to be in NP, since this fact usually follows immediately from their definition,

Example 1. A *clique* in a graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that there is an edge between every pair of vertices in S. That is, $\forall u, v \in V(u \in S \wedge v \in S \rightarrow E(u, v))$. The language $CLIQUE = \{< G, k > | G$ is a graph, $k \in N$, G has a clique of size $k\}$. Here, we take $n = |V|$, so the size of the input is $O(n^2)$.

We will see later that this problem is NP-complete. Now we will show that it is in NP.

Suppose that $< G, k > \in CLIQUE$. That is, G has a set S of vertices, $|S| = k$, such that for any pair $u, v \in S$, $E(u, v)$. Guess this S using an existential quantifier. It can be represented

as a binary string of length n, so its length is polynomial in the size of the input. Now, it takes k^2 checks to verify that every pair of vertices in S is connected by an edge. If the algorithm is scanning E every time, it takes $O(n^2)$ steps to check that a given pair has an edge between them. Therefore, the total time for the check is $k^2 \cdot n^2$, which is quadratic in the length of the input (since E is of size n^2, the input is of size $O(n^2)$ as well).

One common source of confusion, especially among non-computer scientists, is that NP is a class of languages, not a class of optimization problems for which a solution needs to be computed.

Example 2. Recall the Knapsack optimization problem: given a set of pairs of weights and profits $(w_1, p_1) \dots (w_n, p_n)$ and a bound B, compute a set $S \subseteq \{1 \dots n\}$ such that $\Sigma_{i \in S} w_i \leq B$ and $\Sigma_{i \in S} p_i$ is maximized. In this formulation, it is not a language and as such it does not make sense to say that it is in NP. Suppose we define it as a language by adding a parameter P to the input and considering all instances $(w_1, p_1), \dots (w_n, p_n), B, P$ such that P is maximal possible profit. In this case, it is a language; however, this problem is not known to be in NP. Even though we can easily verify, given a set S, that $\Sigma_{i \in S} w_i \leq B$ and that $\Sigma_{i \in S} p_i = P$, how would we be able to test that there is no set with a profit better than P? We don't know of any way of doing it (unless NP = co-NP, which is another major open problem; most people believe they are distinct).

Therefore, when we talk about the decision problem we will call General Knapsack Decision Problem in the context of NP, we will define it as a language

GKD=$\{\langle (w_1, p_1), \cdots, (w_n, p_n), B, P\rangle \mid \exists S \subseteq \{1, \cdots, n\}, \sum_{i \in S} w_i \leq B \text{ and } \sum_{i \in S} p_i \geq P\}$.

To write it in a more readable form,

GKD (General Knapsack Decision Problem).
Instance:
$\langle (w_1, p_1), \cdots, (w_n, p_n), B, P\rangle$ (with all integers nonnegative represented in binary).
Acceptance Condition:
In **GKD** if there is an $S \subseteq \{1, \cdots, n\}$ such that $\sum_{i \in S} w_i \leq B$ and $\sum_{i \in S} p_i \geq P$.

2.1 Alternative definition of NP

A different way to define NP is via non-deterministic Turing machines. Recall that whereas deterministic models of computation have a single transition from every configuration, non-deterministic models of computation, including Turing machines, have a set of possible transitions, and as such, many possible computation paths. Non-deterministic Turing machines are a mathematical abstraction; we cannot build them in practice. And contrary to some of the popular media says, quantum computation is not (proved or believed to be) the same as non-deterministic computation.

It was clear what a deterministic time-bounded Turing machine was: on any input of length

n, it would take time at most $t(n)$. But how do we define computation time for a non-deterministic Turing machine? In this case, we will require that *every* computational path, whether accepting or rejecting, is bounded by $t(n)$.

Definition 18. $\mathsf{NTime}(f(n))$=$\{L \mid$ *some (multi-tape) non-deterministic Turing machine* M *decides* L *in time at most* $f(n)\}$. *That is, for every string* x, $|x| \leq n$, *and every sequence of non-deterministic choices the time taken by* M *on* x *is at most* $f(n)$.

Definition 19. *A language* L *is in* NP *if some non-deterministic Turing machine decides this language in time bounded by* $c|x|^d$ *for some constants* c, d.

Theorem 22. *Definitions 17 and 19 are equivalent.*

Proof. Suppose that L is in NP according to the first definition. Then there exists a verifier, polynomial time-computable relation $R(x, y)$ and constants c, d such that $x \in L \iff \exists y, |y| \leq c|x|^d$ and $R(x, y)$. Now, design a (multi-tape) non-deterministic Turing machine that works as follows. First, it writes # and then a string y on the tape after x, where for every step there is a choice of transitions one writing a 0 and moving right and another writing 1 and moving right; a third possibility is to finish writing y and move to the next stage of the computation. It keeps a counter to make sure $|y| \leq c|x|^d$. Now, simulate a deterministic computation of a polynomial-time Turing machine computing $R(x, y)$. Alternatively, we can describe the computation as non-deterministically guessing y and then computing $R(x, y)$.

For the other direction, suppose L is decided by a non-deterministic Turing machine M. Let s be the maximum number of possibilities on any given transition. Since there is only a constant number of states, symbols in Γ and $\{L, R\}$, s is a constant. Let $t(n) \leq |x|^d$ be the time M takes on its longest branch over all inputs of length n. Now, guess y to be a sequence of symbols $\{1...s\}$, denoting the non-deterministic choices M makes. Guessing y is the same as guessing a computation branch of M. The length of y would be bounded by $t(n)$. So if it takes c bits to encode one symbol of y, the length of y will be bounded by $c|x|^k$. It remains to define $R(x, y)$: just make it a relation that checks every step of the computation of M on x with non-deterministic choices y for correctness, and for ending in accepting state.This is clearly polynomial-time computable since checking one step takes constant time and there are at most $|x|^d$ steps. Now, we have described c, d and $R(x, y)$, it remains to argue that $x \in L \iff \exists y, |y| \leq c|x|^d, R(x, y)$. Notice that if x is in L then there exists a sequence of choices leading to accept state, and that sequence can be encoded by y; alternatively, if there exists a y which is recognized by R to be a correct sequence of transitions to reach an accept state, then this computation branch M would be accepting. □

2.2 Decidability of NP

Theorem 23. $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{EXP}$. *In particular, any problem in* NP *can be solved in exponential time.*

Proof. To see that $\mathsf{P} \subseteq \mathsf{NP}$, just notice that you can take a polynomial-time computable relation $R(x, y)$ which ignores y and computes the answer on x.

For the second inclusion, notice that there are at most $2^{c|x|^d}$ possible strings y. So in time $2^{c|x|^d}$ multiplied by the time needed to compute R all possible "guesses" can be checked. □

Corollary 24. *Any language $L \in \mathsf{NP}$ is decidable.*

3 P vs. NP problem and its consequences.

We now come to one of the biggest open questions in Computer Science. This question was listed as one of the seven main questions in mathematics for the new millennium by the Clay Mathematical Institute, who offered a million dollars for resolution of each of these questions.

Major Open Problem: $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$. That is, is P equal to NP?

This is a problem of "Generating a solution vs. Recognizing a solution". Some examples: student vs. grader; composer vs. listener; writer vs. reader; mathematician vs. computer.

Unfortunately, we are currently unable to prove whether or not P is equal to NP. However, it seems very unlikely that these classes are equal. If they were equal, most combinatorial optimization problems such as Knapsack optimization problem would be solvable in polynomial time. Related to this is the following lemma. This lemma says that if $\mathsf{P} = \mathsf{NP}$, then if $L \in \mathsf{NP}$, then for every string x, not only would we be able to compute in polynomial-time whether or not $x \in L$, but in the case that $x \in L$, we would also be able to actually find a certificate y that demonstrates this fact. That is, for every language in NP, the associated search problem would be polynomial-time computable. That is, whenever we are interested in whether a short string y exists satisfying a particular (easy to test) property, we would automatically be able to efficiently find out if such a string exists, and we would automatically be able to efficiently find such a string if one exists.

Theorem 25. *Assume that $\mathsf{P} = \mathsf{NP}$. Let $R \subseteq \Sigma^* \times \Sigma^*$ be a polynomial-time computable two place predicate, let $c, d \in \mathbb{N}$, and let $L = \{x \mid$ there exists $y \in \Sigma^*, |y| \leq c|x|^d$ and $R(x, y)\}$.*

Then there is a polynomial-time Turing M machine with the following property. For every $x \in L$, if M is given x, then M outputs a string y such that $|y| \leq c|x|^d$ and $R(x, y)$.

Proof: Let L and R be as in the statement of the theorem. Consider the language $L' = \{\langle x, w\rangle \mid$ there exists z such that $|wz| \leq c|x|^d$ and $R(x, wz)\}$. It is easy to see that $L' \in \mathsf{NP}$ (Exercise!), so by hypothesis, $L' \in \mathsf{P}$.

We construct the machine M to work as follows. Let $x \in L$. Using a polynomial-time algorithm for L', we will construct a certificate $y = b_1 b_2 \cdots$ for x, one bit at a time. We

begin by checking if $(x, \epsilon) \in R$, where ϵ is the empty string; if so, we let $y = \epsilon$. Otherwise, we check if $\langle x, 0\rangle \in L'$; if so, we let $b_1 = 0$, and if not, we let $b_1 = 1$. We now check if $(x, b_1) \in R$; if so, we let $y = b_1$. Otherwise, we check if $\langle x, b_1 0\rangle \in L'$; if so, we let $b_2 = 0$, and if not, we let $b_2 = 1$. We now check if $(x, b_1 b_2) \in R$; if so, we let $y = b_1 b_2$. Continuing in this way, we compute bits $b_1, b_2, \cdots$ until we have a certificate y for x, $y = b_1 b_2 \cdots$. $\square$

This lemma has some amazing consequences. It implies that if $\mathsf{P} = \mathsf{NP}$ (in an efficient enough way) then virtually *all* cryptography would be easily broken. The reason for this is that for most cryptography (except for a special case called "one-time pads"), if we are lucky enough to guess the secret key, then we can verify that we have the right key; thus, if $\mathsf{P} = \mathsf{NP}$ then we can actually *find* the secret key, break the cryptosystem, and transfer Bill Gates' money into our private account. (How to avoid getting caught is a more difficult matter.) The point is that the world would become a very different place if $\mathsf{P} = \mathsf{NP}$.

For the above reasons, most computer scientists conjecture that $\mathsf{P} \neq \mathsf{NP}$.

4 Polynomial-time reductions

Just as we scaled down decidable and semi-decidable classes of languages to obtain P and NP, we need to scale down many-one reducibility. It is enough to make the complexity of the reduction at most as hard as the easier problems we will consider. Since here we will mainly be reducing NP problems to each other, it would be sufficient for us to define polynomial-time reductions by slightly modifying the definition of many-one reduction.

Definition 20. *Let $L_1, L_2 \subseteq \Sigma^*$. We say that $L_1 \leq_p L_2$ if there is a* polynomial-time *computable function $f : \Sigma^* \to \Sigma^*$ such that for all $x \in \Sigma^*$, $x \in L_1 \Leftrightarrow f(x) \in L_2$.*

As before, the intuitive meaning is that L_2 is at least as hard as L_1. So if $L_2 \in \mathsf{P}$, then L_1 must be in P.

Definition 21. *A language L is NP-complete if two conditions hold:*

1) (easiness condition) $L \in \mathsf{NP}$

2) (hardness condition) L is NP-hard, that is, $\forall L' \in \mathsf{NP}$, $L' \leq_p L$.

This definition can be generalized to apply to any complexity class such as P or EXP or semi-decidable; however, we need to make sure the notion of reduction is scaled accordingly..

Note that there is a very different type of reduction, called Turing reduction, that is sometimes used to show hardness of optimization problems (not necessarily languages). In that reduction, a question being asked is whether it is possible to solve a problem A using a solver for a problem B as a subroutine; the solver can be called multiple times and returns a

definite answer (e.g., "yes/no") each time. We will touch upon this type of reductions when we talk about search-to-decision reductions and computing solutions from decision-problem solver. Often when people are saying "Knapsack optimization problem is NP-hard" this is what they mean. However, we will only use this type of reduction when talking about computing a solution to a problem, since for the languages this would result in mixing up NP with co-NP.

Lemma 26. *Polynomial-time reducibility is transitive, that is, if* $L_1 \leq_p L_2$ *and* $L_2 \leq_p L_3$*, then* $L_1 \leq_p L_3$*.*

The proof is an easy exercise. One corollary of it is that if A is NP-hard, $A \leq_p B$, $B \leq_p C$ and $C \in \mathsf{NP}$, then all three of these problems are NP-complete.

4.1 Examples of reductions

Here we will give several examples of reductions. Later, when we show that SAT is NP-complete, these reductions will give us NP-completeness for the corresponding problems.

The problem Independent Set is defined as follows $IndSet = \{< G, k > | \exists S \subseteq V, |S| = k, \forall u, v \in S \neg E(u, v)\}$. That is, there are k vertices in the graph such that neither of them is connected to another (of course, they can be connected to vertices outside the set).

Example 3. $Clique \leq_p IndSet$. We need a polynomial-time computable function f, with $f(< G, k >) =< G', k' >$ such that $< G, k > \in Clique \iff < G', k' > \in IndSet$. Let f be such that $k' = k$ and $G' = \bar{G}$, that is, taking G to complement of G and leaving k the same. This function is computable in polynomial time since all it does is taking a complement of the relation E. Now, suppose that there was a clique of size k in G. Call it S. Then the same vertices, in S, will be an independent set in G'. For the other direction, suppose that G' has an independent set of size k. Then G must have had a clique on the same vertices.

In general, proving NP-completeness of a language L by reduction consists of the following steps.

1) Show that the language L is in NP
2) Choose an NP-complete L language from which the reduction will go, that is, $L' \leq_p L$.
3) Describe the reduction function and argue that it is computable in polynomial time.
4) Argue that if an instance x was in L', then $f(x) \in L$.
5) Argue that if $f(x) \in L$ then $x \in L'$.

4.2 Propositional satisfiability problem

One of the most important (classes of) problems in complexity theory is the propositional satisfiability problem. Here, we will define it for CNFs, in the form convenient for NP-completeness reductions. This is the problem which we will show directly is NP-complete and will use as a basis for subsequent reductions.

Definition 22 (Conjunctive normal form)**.** *A propositional formula (that is, a formula where all variables have two values true/false, also written as 1/0) is in CNF if it is a conjunction (AND) of disjunctions (OR) of literals (variables or their negations). A formula is in kCNF, e.g., 3CNF, if the maximal number of variables per disjunction is k. The disjunctions are also called "clauses", so we talk about a formula as a "set of clauses". We use the letter n to be the number of variables (e.g., $x_1 \dots x_n$), and m for the number of clauses.*

Example 4. The following formula is in 4CNF: $(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \neg x_4) \wedge (\neg x_1 \vee \neg x_3 \vee x_4 \vee \neg x_5)$

A *truth assignment* is an assignment of values true/false (1/0) to the variables of a propositional formula. A *satisfying assignment* is a truth assignment that makes the formula true. For example, in the formula above one possible satisfying assignment is $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 0$. In this case, the first clause is satisfied (made true) by either of its literals, the second clause by either of x_1 or $\neg x_2$, and the last by $\neg x_5$.

Definition 23.

$$SAT = \{\phi | \phi \textit{ is a propositional formula which has a satisfying assignment}\}$$

Respectively, $3SAT \subset SAT$ contains only formulae in 3CNF form.

Example 5. The following formula is *unsatisfiable*, that is, false for all possible truth assignments: $(\neg x \vee \neg y) \wedge x \wedge y$.

Example 6. Here we will show that although 3SAT is a subset of SAT, it has the same complexity: $SAT \leq_p 3SAT$. We will start with a formula in CNF form and obtain a 3CNF formula. Consider one clause of a formula, for example, $(\neg x_1 \vee x_3 \vee x_4 \vee \neg x_5)$. We want to convert this clause into a 3CNF formula. The idea is to introduce a new variable y encoding the last two literals of the clause. Then, the formula becomes $(\neg x_1 \vee x_3 \vee y) \wedge (y \iff (x_4 \vee \neg x_5))$. Now, opening the $\iff$ and bringing it into CNF form, we obtain $(\neg x_1 \vee x_3 \vee y) \wedge (\neg y \vee x_4 \vee \neg x_5) \wedge (y \vee \neg x_4) \wedge (y \vee x_5)$. Now apply this transformation to the formula until every clause has at most 3 literals. You can check the equivalence of the two formulae as an exercise. Now it remains to be shown that the transformation can be done in polynomial time. For every extra literal in a clause there needs to be a y introduced. Therefore, since there can be as many as n literals in a clause, there can be as many as $n-2$ such replacements per clause, for m clauses, and each introducing 3 new clauses. Therefore, the new formula will be of size at most $4nm$, which is definitely a polynomial in the size of the original formula. Moreover, each step takes only constant time. Therefore, f is computable in polynomial time.

5 NP Completeness

We will give several version of the first NP-completeness proof. First, we will briefly sketch the proof hat Circuit-SAT is NP-complete. Then we will do the original Cook's proof that SAT is NP-complete, and show how to modify it to obtain Fagin's theorem that existential second-order logic captures NP.

Circuit-SAT $= \{C \mid C$ is a satisfiable Boolean circuit$\}$

Theorem 27. *Circuit-SAT is* NP*-complete.*

Proof. We need to prove that

1) Circuit-SAT is in NP, and

2) Circuit-SAT is NP-hard (i.e., every language $L \in$ NP reduces to Circuit-SAT).

The fact that Circuit-SAT is in NP is easy: Given a circuit C on variables $x_1, \ldots, x_n$, nondeterministically guess an assignment to $x_1, \ldots, x_n$ and verify that this assignment is satisfying; this verification can be done in time polynomial in the size of the circuit. In other words,

$$Circuit - SAT = \{C \mid \exists x, |x| \leq |C|, R'(C, x)\}$$

where $R'(C, x)$ is True iff $C(x) = 1$ (i.e., C on input x evaluates to 1).

Now we prove NP-hardness. Take an arbitrary $L \in$ NP. Say

$$L = \{x \mid \exists y, |y| \leq |x|^c, R(x, y)\}$$

for some constant c and $R \in$ P. Let's suppose that $R(x, y)$ is computable in time $N = (|x| + |y|)^d$, for some constant d.

Consider N steps of computation of the Turing machine deciding R on input x, y. This computation can be pictured as a sequence of N configurations. A configuration at time t is a sequence of symbols $y_1 \ldots y_m$, where each y_j contains the following information: the contents of tape cell j at time t, whether or not tape cell is being scanned by the TM at time t, and if it is, then what is the state of a TM at time t.

The crucial observation is that the computation of a TM has the following "locality property": the value of symbol y_i at time $t+1$ depends only on the values of symbols y_{i-1}, y_i, y_{i+1} at time t (as well as the transition function of the TM).

We can construct a constant-size circuit *Step* that computes the value of y_i at time $t + 1$ from the values of y_{i-1}, y_i, y_{i+1} at time t. Now, we construct a big circuit $C(x, y)$ by replacing each symbol y_i in every configuration at time t by a copy of the circuit *Step* whose inputs

are the outputs of the corresponding three copies of *Step* from the previous configuration. We also modify the circuit so that it outputs 1 on x, y iff the last configuration is accepting.

The size of the constructed circuit will be at most $N * N * |Step|$ (N configurations, at most N copies of *Step* in each), which is polynomial in $|x|$.

Our reduction from L to Circuit-SAT is the following: Given x, construct the circuit $C_x(y) = C(x, y)$ as explained above (with x hardwired into C). It is easy to verify that $x \in L$ iff there is y such that $C_x(y) = 1$. So this is a correct reduction. □

Theorem 28 (Cook-Levin)**.** *SAT is* NP*-complete.*

Proof. One way to prove it would be show $Circuit - SAT \leq_p SAT$. But instead we will go through the proof as it was originally done, without any references to circuits.

Example 7. Consider a very simple non-deterministic Turing machine with 3 states q_0, q_a, q_r and the following transition table:

$$(q_0, 0) \to (q_0, \_, R) \quad (q_0, 1) \to (q_0, \_, R) \quad (q_0, 1) \to (q_a, \_, R) \quad (q_0, \_) \to (q_r, \_, R)$$

That is, TM accepts iff there is a symbol 1 in the input string. The non-determinism comes from the fact that any of the 1s would lead to an accept state in some computation.

To talk about Turing machine computations we will define a notion of *tableau*.

Definition 24. *A* tableau *for a Turing machine M on input w running in time n^k is a $n^k \times n^k$ table whose rows are configurations if a branch of the computation of M on input w.*

That is, any possible path in the computation tree of M on w can be encoded as a tableau. To simplify our formulae, we will assume that the first and the last column of the tableau contains a special symbol #.

The TM from example 7 has the following 3 tableau on string 0110:

T_1=

#	q_0	0	1	1	0	_	#
#	_	q_0	1	1	0	_	#
#	_	_	q_a	1	0	_	#

T_2=

#	q_0	0	1	1	0	_	#
#	_	q_0	1	1	0	_	#
#	_	_	q_0	1	0	_	#
#	_	_	_	q_a	0	_	#

T_3=

#	q_0	0	1	1	0	_	#
#	_	q_0	1	1	0	_	#
#	_	_	q_0	1	0	_	#
#	_	_	_	q_0	0	_	#
#	_	_	_	_	q_0	_	#
#	_	_	_	_	_	q_r	#

The first two correspond to the accepting branches, the last one to the rejecting. So a NTM accepts a string iff it has an accepting tableau on that string.

We will show how to encode a run of a NTM M on a string w by a propositional formula, in a sense that the formula will have a satisfying assignment iff there exists an accepting tableau of M on w. Our formula will have propositional variables $x_{i,j,s}$ where i is the name of the row, j is the column and s is a symbol of $C = Q \cup \Gamma \cup \{\#\}$. The intention is that a variable $x_{i,j,s}$ is set to True iff symbol s is in the cell i, j.

The final formula will consist of a conjunction of 4 parts, call them ϕ_{cell}, ϕ_{start}, ϕ_{accept} and ϕ_{move}.

1) The first formula says that there is exactly one symbol in every cell:

$$\phi_{cell} = \bigwedge_{1 \le i,j < n^k} \left(\left(\bigvee_{s \in C} x_{i,j,s} \right) \wedge \left(\bigwedge_{s,t \in C, s \neq t} (\overline{x_{i,j,s}} \vee \overline{x_{i,j,t}}) \right) \right)$$

2) The second formula forces the values of starting configuration by taking a conjunction of the corresponding variables. In the example 7, on string 0110, the formula will look like this:

$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,0} \wedge x_{1,4,1} \wedge x_{1,5,1} \wedge x_{1,6,0} \wedge x_{1,7,\_} \wedge x_{1,8,\#}$$

3) The third formula says that the tableau corresponds to an accepting computation:

$$\phi_{accept} = \bigvee_{1 \le i,j < n^k} x_{i,j,q_a}$$

4) The last formula ϕ_{move} is the most complicated part. It has to encode the fact that every line in the tableau came from the previous line by a valid transition of M. This formula encodes, for every symbol $x_{i,j,s}$, all possible transitions that can lead to cell $[i, j]$ having a value s. The crucial property that allows us to encode that is that every symbol in the configuration can be determined by looking at the three symbols in the previous line: symbol above and symbols to the left and right of it. That is, to determine the value of $x_{i+1,j,s}$ it is sufficient to know, for all symbols, values of $x_{i,j-1,?}$, $x_{i,j,?}$ and $x_{i,j+1,?}$.Here, ? runs through all possible symbols. If there is no head position symbol among these three, then the value of $x_{i,j,s} = x_{i+1,j,s}$. In the example 7 there are two possibilities for the content of cell [3,4], either q_0 or q_a. This is encoded by the following formula

$$x_{2,3,q_0} \wedge x_{2,4,1} \wedge x_{2,5,1} \rightarrow (x_{3,4,q_0} \vee x_{3,4,q_a})$$

Notice here that the only time there is a $\vee$ on the right side of the implication is when there is a non-deterministic choice of a transition.

Now, suppose that the formula has a satisfying assignment. That assignment encodes precisely a tableau of an accepting computation of M. For the other direction, take an accepting computation of M and set exactly those $x_{i,j,s}$ corresponding to the symbols in the accepting tableau.

Finally, to show that our reduction is polynomial, we will show that the resulting formula is of polynomial size. The number of variables is $N = n^k \times n^k \times |Q \cup \Gamma \cup \{\#\}|$. In ϕ_{cell} there are $O(N)$ variables, same for ϕ_{accept}. Only n^k variables are in ϕ_{start}. Finally, ϕ_{move} has $O(N \times |\delta|)$ clauses. Therefore, the size of the formula is polynomial. □

5.1 Importance of the Cook-Levin Theorem

There is a trivial NP-complete language:

$$L_u = \{(M, x, 1^k) \mid \text{NTM } M \text{ accepts } x \text{ in} \leq k \text{ steps}\}$$

Exercise: Show that L_u is NP-complete.

The language L_u is not particularly interesting, whereas SAT is extremely interesting since it's a well-known and well-studied natural problem in logic. After Cook and Levin showed NP-completeness of SAT, literally hundreds of other important and natural problems were also shown to be NP-complete. It is this abundance of natural complete problems which makes the notion of NP-completeness so important, and the "P vs. NP" question so fundamental.

5.2 Co-NP

We say that a language $L \in \text{coNP}$ if the complement of L is in NP.

In other words, if $L \in \text{coNP}$, then there is a NTM M with the property: if $x \in L$, then *every* computation path of M on x is accepting; if $x \notin L$, then at least one computation path of M on x is rejecting.

Another, equivalent definition of coNP is as follows. A language $L \in \text{coNP}$ if there exist a constant c and a polynomial-time computable relation $R \in \text{P}$ such that

$$L = \{x \mid \forall y, |y| \leq |x|^c, R(x, y)\}$$

Open Question: $\text{NP} \stackrel{?}{=} \text{coNP}$

Define the language

$$TAUT = \{\phi \mid \phi \text{ is a tautology (i.e., identically true)}\}$$

Theorem 29. *TAUT is* coNP*-complete.*

Proof. The proof is easy once you realize that TAUT is essentially the same as the complement of SAT. Since SAT is NP-complete, its complement is coNP-complete. (Check this!) □

The "NP vs. coNP" question is about the existence of short proofs that a given formula is a tautology.

The common belief is that NP $\neq$ coNP, but it is believed less strongly than that P $\neq$ NP.

Lemma 30. *If* P = NP*, then* NP = coNP*.*

Proof. First observe that P = coP (check it!) Now we have NP = P implies coNP = coP. We know that coP = P, and that P = NP. Putting all this together yields coNP = NP. □

The contrapositive of this lemma says that: NP $\neq$ coNP implies P $\neq$ NP.

Thus, to prove P $\neq$ NP, it is enough to prove that NP $\neq$ coNP. This means that resolving the "NP vs. coNP" question is probably even harder than resolving the "P vs. NP" question.

6 Some NP-complete problems

Recall that proving NP-completeness of a language L by reduction consists of the following steps.

1) Show that the language A is in NP
2) Show that A is NP-hard, via the following steps:
   (a) Choose an NP-complete B language from which the reduction will go, that is, $B \leq_p A$.
   (b) Describe the reduction function f
   (c) Argue that if an instance x was in B, then $f(x) \in A$.
   (d) Argue that if $f(x) \in A$ then $x \in B$.
   (e) Briefly explain why is f computable in polytime.

Usually the bulk of the proof is 2a, we often skip 1 and 6 when they are trivial. But make sure you convince yourself they are! Without step 1, your are only proving NP-hardness; with hard enough f you can prove that anything reduces to anything.

6.1 Clique-related problems.

We talked before about Clique and IndependentSet problems and showed that IndependentSet reduces to Clique. Now, if we show that IndSet is NP-complete then by transitivity of $\leq_p$ and the fact that Clique $\in$ NP we get that Clique is NP-complete as well.

Lemma 31. $3SAT \leq_p IS$.

Given

$$\phi = (x \vee y \vee \neg z) \wedge (\neg x \vee w \vee z) \wedge \ldots ()$$

with m clauses, produce the graph G_ϕ that contains a triangle for each clause, with vertices of the triangle labeled by the literals of the clause. Plus, add an edge between any two complementary literals from different triangles. Finally, set $k = m$.

In our example, we have triangles on $x, y, \neg z$ and on $\neg x, w, z$, plus the edges $(x, \neg x)$ and $(\neg z, z)$ (see Figure 6.1).

It remains to be shown that ϕ is satisfiable iff G_ϕ has an independent set of size at least k.

Proof. We need to prove two directions. First, if ϕ is satisfiable, then G_ϕ has an independent set of size at least k. Secondly, if G_ϕ has an independent set of size at least k, then ϕ is satisfiable. (Note that the latter is the contrapositive of the implication "if ϕ is not satisfiable, then G_ϕ does not have an independent set of size at least k".)

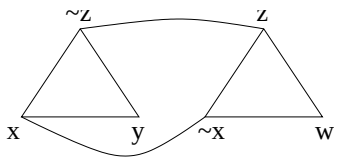
For the first (easy) direction, consider a satisfying assignment for ϕ. Take one true literal from every clause, and put the corresponding graph vertex into a set S. Observe that S is an independent set of size k (where k is the number of clauses in ϕ).

For the other direction, take an independent set S of size k in G_ϕ. Observe that S contains exactly one vertex from each triangle (clause), and that S does not contain any conflicting pair of literals x and $\bar{x}$ (since any such pair of conflicting literals are connected by an edge in G_ϕ). Hence, we can assign the value True to all these literals in the set S, and thereby satisfy the formula ϕ. □

Note that in the proof above, we say things like "take a satisfying assignment to ϕ" or "take an independent set of G_ϕ". These are not efficient steps! But, we don't need them to be efficient, as they are done only as part of the analysis of the reduction (i.e., the proof of correctness of our reduction), and they are not part of the reduction itself. So the point is that the *reduction* must be efficient, whereas the *analysis* of correctness of the reduction may involve inefficient (or even non-computable) steps.

Example 8. Consider the following problem: on a network, we want to put monitoring devices on the nodes of the network so that for every link in the network there is a monitoring device on at least one end. The problem of finding a minimal set of vertices enough to cover all

Figure 1: Reduction from 3SAT to IS.



vertices is called the Minimal Vertex Cover problem. We are interested in the corresponding NP language: similarly to defining GKD from Knapsack, we define the language VC as follows.

$VC = \{\langle G, k\rangle \mid \exists S \subseteq V, |S| \leq k \text{ and } \forall \{u, v\} \in E, u \neq v, \text{ either } u \in S \text{ or } v \in S\}$.

Notice that $VC \in$ NP: given S, it takes $O(|E| * |S|)$ time to check that every edge in G has at least one end in S. To show hardness, we will show that $IndSet \leq_p VC$. We define $f(\langle G, k\rangle) = \langle G', k'\rangle$ by $G' = G$ and $k' = n - k$. Now, we need to argue that the reduction works. Let S be a largest independent set (can be several of the same size). Then we will show that $V - S$ is the smallest vertex cover. First, why is V-S a vertex cover? Suppose it is not; then there is an edge both endpoints of which are in S. But this is impossible, since two endpoints the same edge cannot be in an independent set together. Now, why would it be the smallest vertex cover? Suppose it is possible to throw out of $V - S$ some vertex v and still keep the property that $V - S$ is a vertex cover. This means that the vertex v has all of its neighbours in $V - S$. But then there is no reason why it shouldn't be in S, since it has no neighbours in there. That would contradict the maximality of S as an independent set. For the other direction we can similarly argue that if $V - S$ is a vertex cover, then S must be an independent set, and if $V - S$ is minimal, then S must be maximal.

6.2 Satisfiability problems

We already showed that SAT and $3SAT$ are NP-complete. Another NP-complete variant of SAT is Not-all-equal SAT, where the acceptance requirement is that an assignment not only satisfies at least one literal per clause, but also falsifies at least one literal per clause (in each clause, not all literals are the same: hence the name).

However, not all variants of SAT are necessarily NP-complete. Consider the following 3 variants:

1) 2SAT, where there are at most 2 variables per clause.

2) HornSat, where every clause contains at most one true literal (equivalently, it is of the same form as Prolog language rules: $l_1 \leftarrow l_2 \ldots l_k$).

3) XOR-SAT and XOR-2SAT, where instead of $\vee$ there is an exclusive OR operation.

All of the following classes are solvable in polynomial time, by different algorithms. For example, XOR-SAT can be represented as a system of linear equations over 0,1 and solved using Gaussian Elimination. There is a beautiful theorem due to Schaefer ("Schaefer Dichotomy Theorem") that states that those are pretty much the only possible subclasses of SAT, defining by putting a restriction on how a clause might look like, and there is nothing in between. In particular, for SAT over Boolean variables there is no class of formulas with

complexity between P and NP, although in general such problem exists, provided P $\neq$ NP. Things get interesting, though, if instead of having Boolean variables you allow more than 2 values for the x's (think fuzzy logic or databases). The case of formulas over 3-value variables have been solved in 2002 by Bulatov, how have showed, with a lot of work and a help of a computer, that indeed there is a dichotomy in that case as well. For the 4-valued variables and higher, the problem is wide open.

6.3 Hamiltonicity problems

Consider two very similar problems. One of them is asking, given a graph, if there is a path through the graph that passes every edge exactly once. Another asks if there is a path that visits every vertex exactly once. Although these problems seem very similar, in fact their complexity is believed to be vastly different. The first problem is the famous Euler's problem (the "bridges of Köninsberg" problem): the answer is that there is a cycle through a graph visiting every edge exactly once iff every vertex has an even degree; if exactly two vertices have odd degrees then there is a path, but not a cycle. The second problem, called Hamiltonian Path problem, is NP-complete.

Definition 25. *A* Hamiltonian cycle (path, s-t path) *is a simple cycle (path, path from vertex s to vertex t) in an undirected graph which touches all vertices in the graph. The languages HamCycle, HamPath and stHamPath are sets of graphs which have the corresponding property (e.g., a Hamiltonian cycle).*

We omit the proof that HamPath is NP-complete (see Sipser's book page 286). Instead, we will do a much simpler reduction. Assuming that we know that HamCycle is NP-complete, we will prove that stHamPath is NP-complete. It is easy to see that all problems in this class are in NP: given a sequence of n vertices one can verify in polynomial time that no vertex repeats in the sequence and there is an edge between every pair of subsequent vertices.

Example 9. $HamCycle \leq_p stHamPath$

Proof. Let $f(G) = (G', s, t)$ be the reduction function. Define it as follows. Choose an arbitrary vertex of G (say, labeled v). Suppose that there is no vertex in G called v'. Now, set vertices of G' to be $V' = V \cup \{v'\}$, and edges of G' to be $E' = E \cup \{(u, v') \mid (u, v) \in E\}$. That is, the new vertex v' is a "copy" of v in a sense that it is connected to exactly the same vertices as v. Then, set $s = v$ and $t = v'$.

Now, suppose that there is a Hamiltonian cycle in G. Without loss of generality, suppose that it starts with v, so it is $v = v_1, v_2, \ldots v_n, v$. Here, it would be more correct to use numbering of the form $v_{i_1} \ldots v_{i_n}$, but for simplicity we assume that the vertices are renumbered. Now, replacing the final v with v' we get a Hamiltonian path from $s = v$ to $t = v'$ in G'.

For the other direction, suppose that G' has a Hamiltonian path starting from s and ending in t. Then since s and t correspond to the same vertex in G, this path will be a Hamiltonian cycle in G.

Lastly, since f does no computation and only adds 1 vertex and at most n edges the reduction is polynomial-time. □

Note that this reduction would not work if we were reducing to HamPath rather than stHamPath. Then the part 1c of the proof would break: it might be possible to have a Hamiltonian path in G' but not a ham. cycle in G if we allow v and v' to be in different parts of the path.

Example 10 (Travelling salesperson problem)**.** In the Traveling Salesperson Problem (TSP) the "story" is about a traveling salesperson who wants to visit n cities. For every pair of the cities there is a direct connection; however, those connections have very different costs (think flights between cities and their costs). In the optimization problem, the salesperson wants to visit all cities and come back as cheaply as possible; in the decision version, it is a question of whether it is possible to visit all cities given a fixed budget B which should cover all the costs.

To model TSP, consider an undirected graph in which all possible edges $\{u, v\}$ (for $u \neq v$) are present, and for which we have a nonnegative integer valued cost function c on the edges. A *tour* is a simple cycle containing all the vertices (exactly once) – that is, a Hamiltonian cycle – and the *cost* of the tour is the sum of the costs of the edges in the cycle.

TSP
Instance:
$\langle G, c, B\rangle$ where G is an undirected graph with all edges present , c is a nonnegative integer cost function on the edges of G, and B is a nonnegative integer.
Acceptance Condition:
Accept if G has a tour of cost $\leq B$.

Theorem 32. *TSP is* NP*-Complete.*

Proof. It is easy to see that TSP $\in$ NP.
We will show that HamCycle $\leq_p$ TSP.
Let α be an input for HamCycle, and as above assume that α is an instance of HamCycle, $\alpha = \langle G\rangle$, $G = (V, E)$. Let
$f(\alpha) = \langle G', c, 0\rangle$ where:
$G' = (V, E')$ where E' consists of all possible edges $\{u, v\}$;
for each edge $e \in E'$, $c(e) = 0$ if $e \in E$, and $c(e) = 1$ if $e \notin E$.

It is easy to see that G has a Hamiltonian cycle $\Leftrightarrow$ G' has a tour of cost ≤ 0. □

Note that the above proof implies that TSP is NP-complete, even if we restrict the edge costs to be in $\{0, 1\}$.

6.4 SubsetSum, Partition and Knapsack

SubsetSum
Instance:
$\langle a_1, a_2, \cdots, a_m, t\rangle$ where t and all the a_i are nonnegative integers presented in binary.
Acceptance Condition:
Accept if there is an $S \subseteq \{1, \cdots, m\}$ such that $\sum_{i\in S} a_i = t$.

We will postpone the proof that SubsetSum is NP-complete until the next lecture. For now, we will give a simpler reduction from SubsetSum to a related problem Partition.

PARTITION
Instance:
$\langle a_1, a_2, \cdots, a_m\rangle$ where all the a_i are nonnegative integers presented in binary.
Acceptance Condition:
Accept if there is an $S \subseteq \{1, \cdots, m\}$ such that $\sum_{i\in S} a_i = \sum_{j\notin S} a_j$.

Lemma 33. *PARTITION is* NP*-Complete.*

Proof. It is easy to see that PARTITION $\in$ NP.
We will prove SubsetSum $\leq_p$ PARTITION. Let x be an input for SubsetSum. Assume that x is an Instance of SubsetSum, otherwise we can just let $f(x)$ be some string not in PARTITION. So $x = \langle a_1, a_2, \cdots, a_m, t\rangle$ where t and all the a_i are nonnegative integers presented in binary. Let $a = \sum_{1\leq i\leq m} a_i$.

Case 1: $2t \geq a$.
Let $f(x) = \langle a_1, a_2, \cdots, a_m, a_{m+1}\rangle$ where $a_{m+1} = 2t - a$. It is clear that f is computable in polynomial time. We wish to show that
$x \in$ SubsetSum $\Leftrightarrow f(x) \in$ PARTITION.

To prove $\Rightarrow$, say that $x \in$ SubsetSum. Let $S \subseteq \{1, \cdots, m\}$ such that $\sum_{i\in S} a_i = t$. Letting $T = \{1, \cdots, m\} - S$, we have $\sum_{j\in T} a_i = a - t$. Letting $T' = \{1, \cdots, m+1\} - S$, we have $\sum_{j\in T'} a_i = (a-t) + a_{m+1} = (a-t) + (2t-a) = t = \sum_{i\in S} a_i$. So $f(x) \in$ PARTITION.

To prove $\Leftarrow$, say that $f(x) \in$ PARTITION. So there exists $S \subseteq \{1, \cdots, m+1\}$ such that letting $T = \{1, \cdots, m+1\} - S$, we have $\sum_{i\in S} a_i = \sum_{j\in T} a_j = [a + (2t-a)]/2 = t$. Without loss of generality, assume $m+1 \in T$. So we have $S \subseteq \{1, \cdots, m\}$ and $\sum_{i\in S} a_i = t$, so
$x \in$ SubsetSum.

Case 2: $2t \leq a$. You can check that adding $a_{m+1} = a - 2t$ works. □

Warning: Students often make the following serious mistake when trying to prove that $L_1 \leq_p L_2$. When given a string x, we are supposed to show how to construct (in polynomial time) a string $f(x)$ such that $x \in L_1$ if and only if $f(x) \in L_2$. We are supposed to construct $f(x)$ without knowing whether or not $x \in L_1$; indeed, this is the whole point. However, often

students assume that $x \in L_1$, and even assume that we are given a certificate showing that $x \in L_1$; this is completely missing the point.

Theorem 34. *SubsetSum is NP-complete*

Proof. We already have seen that SubsetSum is in NP (guess S, check that the sum is equal to t). Now we will show that SubsetSum is NP-complete by reducing a known NP-complete problem $3SAT \leq_p SubsetSum$.

Given a 3cnf on n variables and m clauses, we define the following matrix of decimal digits. The rows are labeled by literals (i.e., x and $\bar{x}$ for each variable x), the first n columns are labeled by variables, and another m columns by clauses.

For each of the first n columns, say the one labeled by x, we put 1's in the two rows labeled by x and $\bar{x}$. For each of the last m columns, say the one corresponding to the clause $\{x, \bar{y}, z\}$, we put 1's in the three rows corresponding to the literals occurring in that clause, i.e., rows x, $\bar{y}$, and z. We also add $2m$ new rows to our table, and for each clause put two 1's in the corresponding column so that each new row has exactly one 1. Finally, we create the last row to contain 1's in the first n columns and 3 in the last m columns.

The $2n + 2m$ rows of the constructed table are interpreted as decimal representations of $k = 2n + 2m$ numbers $a_1, \ldots, a_k$, and the last row as the decimal representation of the number T. The output of the reduction is $a_1, \ldots, a_k, T$.

Now we prove the correctness of the described reduction. Suppose we start with a satisfying assignment to the formula. We specify the subset S as follows: For every literal assigned the value True (by the given satisfying assignment), put into S the corresponding row. That is, if x_i is set to True, add to S the number corresponding to the row labeled with x_i; otherwise, put into S the number corresponding to the row labeled with $\bar{x_i}$. Next, for every clause, if that clause has 3 satisfied literals (under our satisfying assignment), don't put anything in S. If the clause has 1 or 2 satisfied literals, then add to S 2 or 1 of the dummy rows corresponding to that clause. It is easy to check that the described subset S is such that the sum of the numbers yields exactly the target T.

For the other direction, suppose we have a subset S that makes the subset sum equal to T. Since the first n digits in T are 1, we conclude that the subset S contains exactly one of the two rows corresponding to variable x_i, for each $i = 1, \ldots, n$. We make a truth assignment by setting to True those x_i which were picked by S, and to False those x_i such that the row $\bar{x_i}$ was picked by S. We need to argue that this assignment is satisfying. For every clause, the corresponding digit in T is 3. Even if S contains 1 or 2 dummy rows corresponding to that clause, S must contain at least one row corresponding to the variables, thereby ensuring that the clause has at least one true literal. □

Corollary 35. *Partition is NP-complete*

Proof. In the last lecture we have showed that $SubsetSum \leq_p Partition$. Since SubsetSum is NP-complete, so is Partition. □

7 "Search-to-Decision" Reductions

Suppose that $\mathsf{P} = \mathsf{NP}$. That would mean that all NP languages can be decided in deterministic polytime. For example, given a graph, we could decide in deterministic polytime whether that graph is 3-colorable. But could we find an actual 3-coloring? It turns out that yes, we can. In general, we can define an NP *search problem*: Given a polytime relation R, a constant c, and a string x, find a string y, $|y| \leq |x|^c$, such that $R(x, y)$ is true, if such a y exists. We already saw (recall theorem 25 on page 5) that if $\mathsf{P} = \mathsf{NP}$, then it is possible to compute the certificate in polynomial time. Now we will expand on this, showing how the ability to get access to the answer to the decision problem allows us to solve the search problem.

This is not as trivial matter as it seems from the simplicity of the reductions. This method produces *some* certificate of the membership, not necessarily one you might like to know. As a prime (pun intended) example, remember that testing whether a number is prime can be done in polynomial time; however, computing a factorization of a composite number is believed to be hard, and is a basis for some cryptographic schemes (think RSA).

Theorem 36. *If there is a way to check, for any formula ϕ, if ϕ is satisfiable, then there is a polynomial-time (not counting the check) algorithm that, given a formula $\phi(y_1, \ldots, y_n)$, finds a satisfying assignment to ϕ, if such an assignment exists.*

Proof. We use a kind of binary search to look for a satisfying assignment to ϕ. First, we check if $\phi(x_1, \ldots, x_n) \in SAT$. Then we check if $\phi(0, x_2, \ldots, x_n) \in SAT$, i.e., if ϕ with x_1 set to False is still satisfiable. If it is, then we set a_1 to be 0; otherwise, we make $a_1 = 1$. In the next step, we check if $\phi(a_1, 0, x_3, \ldots, x_n) \in SAT$. If it is, we set $a_2 = 0$; otherwise, we set $a_2 = 1$. We continue this way for n steps. By the end, we have a complete assignment $a_1, \ldots, a_n$ to variables $x_1, \ldots, x_n$, and by construction, this assignment must be satisfying.

The amount of time our algorithm takes is polynomial in the size of ϕ: we have n steps, where at each step we must answer a SAT question. Since we did not count the time to answer a SAT question in our computation time, the rest of each step takes polytime. □

One way of thinking about this type of problem is that if $\mathsf{P} = \mathsf{NP}$, then a certificate to any $x \in L$ for any $L \in \mathsf{NP}$ can be computed in polynomial time. Another way is to imagine a giant precomputed database somewhere in the cloud storing answers to, say, all instances of SAT up to a certain (large enough) size. Suppose that our algorithm can send queries to this database and get back the answers. Since these answers are already precomputed, there is no time (and at least no our time) spent retrieving them. Of course, this dataset would be enormous.

As another example of the search-to-decision reduction, consider the Clique search problem. Assuming that there is a procedure $CliqueD(G, k)$ which given a graph G and a number k says "yes" if G has a clique of size k and "no" otherwise, design an algorithm $CliqueS(G)$

finding a clique of maximum size as follows. First, ask $CliqueD(G, n)$. If the answer is "yes", then G is a complete graph and we are done. If not, proceed doing a binary search on k to determine the size of the largest clique (it does not quite matter in this problem whether it is a binary search or a linear search, but it does matter for other applications). Suppose k_{max} is the size of the maximum clique. Now, find an actual clique by repeatedly throwing out vertices and checking if there is still a clique of size k_{max} in the remaining graph; if not, you know that the vertex just thrown out was a part of the clique and should be kept in G for the following iterations.

It should be stressed that we are interested in *efficient* (i.e., polytime) search-to-decision reductions. Such efficient reductions allow us to say that if the decision version of our problem is in P, then there is also a polytime algorithm solving the corresponding search version of the problem.