CSC 364S Notes **University of Toronto, Spring, 2003**

NP and NP-Completeness

NP

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 an alternative way of defining the class uses a notion of a "nondeterministic" Turing machine. It 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.

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 1. *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 $x \in \Sigma^*$*
$x \in L \Leftrightarrow$ there 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
$\{x; y \,|\, (x, y) \in R\} \in$**P**, where ";" is a symbol not in Σ.
An equivalent 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 $x\not{b}y$ on its input tape ($x, y \in \Sigma^*$), 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, Consider, for example, the language SDPDD.

SDPDD (Scheduling with Deadlines, Profits and Durations Decision Problem).
<u>Instance:</u>
$\langle (d_1, g_1, t_1), \cdots, (d_m, g_m, t_m), B\rangle$ (with all integers represented in binary).

<u>Acceptance Condition:</u>
Accept if there is a feasible schedule with profit $\geq B$.

We can easily see that SDPDD$\in$**NP** by letting R be the set of $(x, y) \in \Sigma^* \times \Sigma^*$ such that $x = \langle (d_1, g_1, t_1), \cdots, (d_m, g_m, t_m), B\rangle$ is an instance of SDPDD and y is a feasible schedule

with profit $\geq B$. It is easy to see that membership in R can be computed in polynomial time. Since for most reasonable encodings of this problem a schedule can be represented so that its length is less than the length of the input, we have
$x \in \text{SDPDD} \Leftrightarrow$ there exists $y \in \Sigma^*, |y| \leq |x|$ and $R(x, y)$.

The point is that the language is *defined* by saying that x is in the language if and only if some y exists such that (x, y) satisfies some simple, given property; it will usually be clear that if such a certificate y exists, then a "short" such y exists.

A slightly more interesting example is the language of composite numbers COMP:

COMP
Instance:
An integer $x \geq 2$ presented in binary.

Acceptance Condition:
Accept if x is not a prime number.

We can easily see that COMP$\in$**NP** by letting R be the set of $(x, y) \in \Sigma^* \times \Sigma^*$ such that x and y are integers (in binary) and $2 \leq y < x$ and y divides x. It is easy to see that membership in R can be computed in polynomial time, and for every $x \in \Sigma^*$
$x \in \text{COMP} \Leftrightarrow$ there exists $y \in \Sigma^*, |y| \leq |x|$ and $R(x, y)$.

A much more interesting example is the language PRIME consisting of prime numbers:

PRIME
Instance:
An integer $x \geq 2$ presented in binary.

Acceptance Condition:
Accept if x is a prime number.

It turns out that PRIME $\in$ **NP**, but this is a more difficult theorem; the proof involves concepts from number theory, and will not be given here.

It is not, however, hard to prove that **NP** includes all of **P**.

Theorem 1. P $\subseteq$ NP

Proof: Let $L \subseteq \Sigma^*$, $L \in \mathbf{P}$. Let $R = \{(x, y) \,|\, x \in L\}$. Then since $L \in \mathbf{P}$, R is computable in polynomial time. It is also clear that for every $x \in \Sigma^*$
$x \in L \Leftrightarrow$ there exists $y \in \Sigma^*, |y| \leq |x|$ and $R(x, y)$. $\square$

We now come to one of the biggest open questions in Computer Science.

Open Question: Is **P** equal to **NP**?

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, all of the search problems mentioned so far in these notes would be solvable in polynomial time, because they are reducible (in the sense of $\stackrel{p}{\longrightarrow}$) to a language in $\mathbf{NP}$. More generally, most combinatorial optimization problems would be solvable in polynomial time, for essentially the same reason. Related to this is the following lemma. This lemma says that if $\mathbf{P} = \mathbf{NP}$, then if $L \in \mathbf{NP}$, 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 $\mathbf{NP}$, 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.

Lemma 1. *Assume that* $\mathbf{P} = \mathbf{NP}$.
Let $R \subseteq \Sigma^* \times \Sigma^*$ *be a polynomial-time computable two place predicate, let* $c, d \in \mathbb{N}$, *and let* $L = \{x \mid \text{there exists } y \in \Sigma^*, |y| \leq c|x|^d \text{ and } R(x, y)\}$.

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 Lemma. Consider the language $L' = \{\langle x, w\rangle \mid \text{there exists } z \text{ such that } |wz| \leq c|x|^d \text{ and } R(x, wz)\}$. It is easy to see that $L' \in \mathbf{NP}$ (Exercise!), so by hypothesis, $L' \in \mathbf{P}$.

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 $\mathbf{P} = \mathbf{NP}$ (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 $\mathbf{P} = \mathbf{NP}$ 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 $\mathbf{P} = \mathbf{NP}$.

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

Conjecture: $\mathbf{P} \neq \mathbf{NP}$

Given that we can't prove that $\mathbf{P} \neq \mathbf{NP}$, is there any way we can gain confidence that a particular language $L \in \mathbf{NP}$ is not in $\mathbf{P}$? It will turn out that if we can prove that L is

"**NP**-Complete", this will imply that if L is in **P**, then $\mathbf{P} = \mathbf{NP}$; we will take this as being very strong evidence that $L \notin \mathbf{P}$.

NP-Completeness

Definition 2. *Let $L \subseteq \Sigma^*$. Then we say L is* **NP**-Complete *if:*

1) *$L \in \mathbf{NP}$ and*

2) *For all $L' \in \mathbf{NP}$, $L' \leq_p L$.*

If L satisfies the second condition in the definition of **NP**-completeness, we say
L is *(ManyOne)*-**NP**-*Hard*.

A weaker condition we could have used is "(Turing)-**NP**-Hard", which applies to search problems as well as to languages. Let S be a search problem or a language. We say S is *Turing*-**NP**-*Hard* if for every language $L' \in \mathbf{NP}$, $L' \xrightarrow{p} S$.
It is clear that if L is a language, then if L is (ManyOne)-**NP**-Hard, then L is (Turing)-**NP**-Hard. (The reader should not be concerned with the historical reasons for the choice of the qualifiers "ManyOne" and "Turing".)

Lemma 2. *Let S be a search problem or a language that is (Turing)-*NP*-Hard. Then if S is solvable in polynomial time, then $\mathbf{NP} = \mathbf{P}$. (Hence, if any (ManyOne)***NP***-Hard language is in* **P***, then $\mathbf{NP} = \mathbf{P}$. Hence, if any* **NP***-Complete language is in* **P***, then $\mathbf{NP} = \mathbf{P}$.)*

Proof: Say that S is (Turing)-*NP*-Hard and solvable in polynomial time. Consider an arbitrary language $L' \in \mathbf{NP}$. We have $L' \xrightarrow{p} S$ and S is solvable in polynomial time, so by Theorem 2 in the last set of notes, $L' \in \mathbf{P}$. So $\mathbf{NP} \subseteq \mathbf{P}$, so $\mathbf{NP} = \mathbf{P}$. $\square$

Because of this Lemma, if a problem is **NP**-Hard, we view this as very strong evidence that it is not solvable in polynomial time. In particular, it is very unlikely that any **NP**-Complete language is in **P**. We also have the following theorem.

Theorem 2. *Let L be an* **NP***-Complete language. Then*
$L \in \mathbf{P} \Leftrightarrow \mathbf{P} = \mathbf{NP}$.

Proof: Say that L is **NP**-Complete.
This implies $L \in \mathbf{NP}$, so if $\mathbf{P} = \mathbf{NP}$, then $L \in \mathbf{P}$.
Conversely, assume $L \in \mathbf{P}$. The the previous Lemma implies $\mathbf{P} = \mathbf{NP}$. $\square$

Note that we have not proven that *any* language is **NP**-Complete. We will now define the language CircuitSat, and prove that it is **NP**-Complete.

First we have to define what a "Boolean, combinational circuit" (or for conciseness, just "Circuit") is. A Circuit is just a "hardwired" algorithm that works on a fixed-length string of n Boolean inputs. Each gate is either an "and" gate, an "or" gate, or a "not" gate; a gate has either one or two inputs, each of which is either an input bit, or the output of a previous gate. The last gate is considered the output gate. There is no "feedback", so a circuit is essentially a directed, acyclic graph, where n nodes are input nodes and all the other nodes are labelled with a Boolean function. (Note that our circuits differ slightly from those of the text CLR, since the text allows "and" and "or" gates to have more than two inputs.)

More formally, we can view a circuit C with n input bits as follows. We view the i-th gate as computing a bit value into a variable x_i. It is convenient to view the inputs as gates, so that the first n gate values are $x_1, x_2, \cdots, x_n$. Each of the other gates compute bit values into variables $x_{n+1}, x_{n+2}, \cdots, x_m$. For $n < i \leq k$, the i-th gate will perform one of the following computations:
$x_i \leftarrow x_j \vee x_k$ where $1 \leq j, k < i$, or
$x_i \leftarrow x_j \wedge x_k$ where $1 \leq j, k < i$, or
$x_i \leftarrow \neg x_j$ where $1 \leq j < i$.
The output of C is the value of x_m, and we say C *accepts* $a_1 a_2 \cdots a_n \in \{0,1\}^n$ if, when $x_1, x_2, \cdots, x_n$ are assigned the values $a_1, a_2, \cdots, a_n$, x_m gets assigned the value 1. We can now define the language CircuitSat as follows.

CircuitSat
Instance:
$\langle C \rangle$ where C is a circuit on n input bits.

Acceptance Condition:
Accept if there is some input $\in \{0,1\}^n$ on which C accepts (that is, outputs 1).

Theorem 3. *CircuitSat is* **NP**-*Complete.*

Proof Outline: We first show that CircuitSat $\in$ **NP**. As explained above, this is easy. We just let $R = \{(\alpha, a) \mid \alpha = \langle C \rangle, C$ a circuit, and a is an input bit string that C accepts$\}$. It is easy to see that R is computable in polynomial time, and for every $\alpha \in \{\Sigma^*\}$,
$\alpha \in$ CircuitSat $\Leftrightarrow$ there exists $a \in \Sigma^*, |a| \leq |\alpha|$ and $R(\alpha, a)$.

We now show that CircuitSat is (ManyOne)-**NP**-Hard; this is the interesting part. Let $L' \in$ **NP**. So there is a polynomial time computable 2-place relation R and integers c, d such that for all $w \in \Sigma^*$,
$w \in L' \Leftrightarrow$ there exists $t \in \Sigma^*, |t| \leq c|w|^d$ and $R(w, t)$.
Say that M is a polynomial time machine that accepts R. There is a constant e such that on all inputs (w, t) such that $|t| \leq c|w|^d$, M halts in at most $|w|^e$ steps (for sufficiently long w).

Now let $w \in \Sigma^*$, $|w| = n$. We want to compute a string $f(w)$ in polynomial time such that $w \in L' \Leftrightarrow f(w) \in$ CircuitSat. The idea is that $f(w)$ will be $\langle C_w \rangle$, where C_w is a circuit that has w built into it, that interprets its input as a string t over Σ of length $\leq cn^d$, and that simulates M computing on inputs (w, t). There are a number of technicalities here. One problem is that C_w will have a fixed number of input bits, whereas we want to be able to view the possible inputs to C as corresponding to all the strings over Σ of length $\leq cn^d$. However it is not hard to invent a convention (at least for sufficiently large n) whereby each string over Σ of length $\leq cn^d$ corresponds to a string of bits of length exactly cn^{d+1}. So C_w will have cn^{d+1} input bits, and will view its input as a string t over Σ of length $\leq cn^d$.

It remains to say how C_w will simulate M running on inputs (w, t). This should be easy to see for a computer scientist/engineer who is used to designing hardware to simulate software. We can design C_w as follows. C_w will proceed in n^e stages, where the j-th stage simulates M running for j steps; this simulation will compute the values in all the squares of M that are within distance n^e of the square the head was initially pointing at; the simulation will also keep track of the current state of M and the current head position. It is not hard to design circuitry to compute the information for stage $j+1$ from the information for stage j. $\square$

SAT and Other NP-Complete Languages

The theory of **NP**-Completeness was begun (independently) by Cook and Levin in 1971, who showed that the language SAT of satisfiable formulas of the propositional calculus is **NP**-Complete. In subsequent years, many other languages were shown (by many different people) to be **NP**-Complete; these languages come from many different domains such as mathematical logic, graph theory, scheduling and operations research, and compiler optimization. Before the theory of **NP**-Completeness, each of these problems was worked on independently in the hopes of coming up with an efficient algorithm. Now we know that since any two **NP**-Complete languages are polynomial-time transformable to one another, it is reasonable to view them as merely being "restatements" of one another, and it is unlikely that any of them have polynomial time algorithms. With this knowledge we can approach each of these problems in a new way: we can try to solve a slightly different version of the problem, or we can try to solve only special cases, or we can try to find "approximate" solutions.

To define SAT, recall the definition of a formula of the propositional calculus. We have atoms $x, y, z, x_1, y_1, z_1, \cdots$ and connectives $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$. An example of a formula F is $((x \vee y) \wedge \neg(x \wedge z)) \rightarrow (x \leftrightarrow y)$. A truth assignment τ assigns "true" or "false" (that is, 1 or 0) to each atom, thereby giving a truth value to the formula. We say τ satisfies F if τ makes F true, and we say F is *satisfiable* if there is some truth assignment that satisfies it. For example, the previous formula F is satisfied by the truth assignment:
$\tau(x) = 1, \tau(y) = 0, \tau(z) = 1$.

SAT
Instance:
$\langle F \rangle$ where F is a formula of the propositional calculus.

Acceptance Condition:
Accept if F is satisfiable.

Another version of this language involves formulas of a special form. We first define a *literal*

to be an atom or the negation of an atom, and we define a *clause* to be a disjunction of one or more literals. We say a formula is in CNF (for "conjunctive normal form") if it is a conjunction of one or more clauses. We say a formula is in 3CNF if it is in CNF, and each clause contains at most 3 literals. For example, the following formula is in 3CNF:
$(x \vee y) \wedge (\neg x) \wedge (\neg y \vee \neg w \vee \neg x) \wedge (y)$. We define the language 3SAT as follows.

3SAT
Instance:
$\langle F \rangle$ where F is a formula of the propositional calculus in 3CNF.

Acceptance Condition:
Accept if F is satisfiable.

The reader should note that our definition of SAT is the same as the text's, but different from some other definitions in the literature. Some sources will use "SAT" to refer to the set of satisfiable formulas that are in CNF. In any case, this variant is **NP**-Complete as well.

We want to prove that 3SAT is **NP**-Complete. From now on, however, when we prove a language **NP**-Complete, it will be useful to use the fact that some other language has already been proven **NP**-Complete. We will use the following lemma.

Lemma 3. *Let L be a language over Σ such that*

*1) $L \in$ **NP**, and*

*2) For some **NP**-Complete language L', $L' \leq_p L$*

*Then L is **NP**-Complete.*

Proof: We are given that $L \in$ **NP**, so we just have to show that that for every language $L'' \in$ **NP**, $L'' \leq_p L$. So let $L'' \in$ **NP**. We know that $L'' \leq_p L'$ since L' is **NP**-Complete, and we are given that $L' \leq_p L$. By the transitivity of $\leq_p$ (proven earlier), we have $L'' \leq_p L$. $\square$

Theorem 4. *3SAT is **NP**-Complete.*

Proof: We can easily see that 3SAT$\in$**NP** by letting R be the set of $(x, y) \in \Sigma^* \times \Sigma^*$ such that $x = \langle F \rangle$ where F is a formula, and y is a string of bits representing a truth assignment τ to the atoms of F, and τ satisfies F. It is clear that membership in R can be computed in polynomial time, since it is easy to check whether or not a given truth assignment satisfies a given formula. It is also clear that for every $x \in \Sigma^*$,
$x \in$ 3SAT $\Leftrightarrow$ there exists $y \in \Sigma^*$, $|y| \leq |x|$ and $R(x, y)$.

By the previous Lemma, it is now sufficient to prove that CircuitSat $\leq_p$ 3SAT.
Let α be an input for CircuitSat. Assume that $\alpha = \langle C \rangle$ where C is a circuit (otherwise we

just let $f(\alpha)$ be some string not in 3SAT). We wish to compute a formula $f(C) = F_C$ such that
C is a satisfiable circuit $\Leftrightarrow$ F is a satisfiable formula.

Recall that we can view C as having n inputs $x_1, \cdots, x_n$, and $m - n$ gates that assign values to $x_{n+1}, \cdots, x_m$, where x_m is the output of the circuit. The formula F_C we construct will have variables $x_1, \cdots, x_m$; it will have a subformula for each gate in C, asserting that the variable corresponding to the output of that gate bears the proper relationship to the variables corresponding to the inputs. For each i, $n + 1 \leq i \leq m$, we define the formula G_i as follows:
If for gate i we have $x_i \leftarrow x_j \vee x_k$, then G_i is $x_i \leftrightarrow (x_j \vee x_k)$.
If for gate i we have $x_i \leftarrow x_j \wedge x_k$, then G_i is $x_i \leftrightarrow (x_j \wedge x_k)$.
If for gate i we have $x_i \leftarrow \neg x_j$, then G_i is $x_i \leftrightarrow \neg x_j$.

Of course, G_i is not in 3CNF. But since each G_i involves at most 3 atoms, we can construct formulas G'_i such that G'_i is logically equivalent to G_i, and such that G'_i is in 3CNF with at most 8 clauses. For example, if G_i is $x_i \leftrightarrow (x_j \vee x_k)$, we can let G'_i be
$(\neg x_i \vee x_j \vee x_k) \wedge (x_i \vee \neg x_j) \wedge (x_i \vee \neg x_k)$.

We then let $F_C = G'_{n+1} \wedge G'_{n+2} \wedge \cdots \wedge G'_m \wedge x_m$. The clauses in G'_{n+1} through G'_m assert that the variables get values corresponding to the gate outputs, and the last clause x_m asserts that C outputs 1. It is hopefully now clear that
C is satisfiable $\Leftrightarrow$ F_C is satisfiable.

To prove $\Rightarrow$, consider an assignment $a_1, \cdots, a_n$ to the inputs of C that makes C output 1. This induces an assignment $a_1, \cdots, a_m$ to all the gates of C. We now check that the truth assignment that for each i assigns a_i to x_i satisfies all the clauses of F_C, and hence satisfies F_C.

To prove $\Leftarrow$, consider a truth assignment τ that assigns a_i to x_i for each i, $1 \leq i \leq m$, and that satisfies F_C. Consider the assignment of $a_1, \cdots, a_n$ to the inputs of C. Using the fact that τ satisfies F_C, we can prove by induction on i, $n + 1 \leq i \leq m$, that C assigns a_i to x_i. Hence, C assigns a_m to x_m. Since τ satisfies F_C, $a_m = 1$. So C outputs 1, so C is satisfiable.
$\Box$

Theorem 5. *SAT is* **NP***-Complete.*

Proof: It is easy to show that SAT $\in$ **NP**. (Exercise.)
We will now show 3SAT $\leq_p$ SAT. This is easy as well. Intuitively, it is clear that this holds, since 3SAT is just a special case of SAT. More formally, let $x \in \Sigma^*$ be an input. If x is not equal to $\langle F \rangle$, for some 3CNF formula F, then let $f(x)$ be any string not in SAT; otherwise, let $f(x) = x$.

f is computable in polynomial time, and for all x, $x \in$ 3SAT $\Leftrightarrow$ $x \in$ SAT. $\Box$

This previous Theorem is just a special case of the following lemma.

Lemma 4. *Let L_1, L_2, L_3 be languages over Σ such that*
L_1 is **NP**-*Complete, $\Sigma^* \neq L_2 \in$* **NP**, *$L_3 \in$* **P** *and $L_1 = L_2 \cap L_3$.*
Then L_2 is **NP**-*Complete.*

Proof: Exercise. □

It should be noted that the text CLRS proves these theorems in a different order. After showing that CircuitSat is **NP**-Complete, they show CircuitSat $\leq_p$ SAT, and then SAT $\leq_p$ 3SAT. The proof given in the text that SAT $\leq_p$ 3SAT is very interesting, and is worth studying.

It also follows from what we have done that SAT $\leq_p$ 3SAT, since we have shown that SAT $\leq_p$ CircuitSat (since CircuitSat is **NP**-Complete), and that CircuitSat $\leq_p$ 3SAT. It is interesting to put these proofs together to see how, given a formula F, we create (in polynomial time) a 3CNF formula G so that F is satisfiable $\Leftrightarrow$ G is satisfiable. We do this as follows.
Given F, we first create a circuit C that simulates a Turing machine that tests whether its input satisfies F; we then create a 3CNF formula G that (essentially) simulates C. It is important to understand that G is *not logically equivalent* to F, but it is true that
F is satisfiable $\Leftrightarrow$ G is satisfiable.

Using 3SAT, it is possible to show that many other languages are **NP**-Complete. Texts such as CLRS typically define **NP** languages CLIQUE, VertexCover, IndependentSet, SubsetSum, PARTITION, HamCycle, and TSP (Traveling Salesman) and prove them **NP**-Complete by proving transformations such as:
3SAT $\leq_p$ CLIQUE $\leq_p$ IndependentSet $\leq_p$ VertexCover and
3SAT $\leq_p$ HamCycle $\leq_p$ TSP and
3SAT $\leq_p$ SubsetSum $\leq_p$ PARTITION $\leq_p$ GKD.

We will now define some of these languages. It is convenient to first define a number of concepts relating to undirected graphs.

Definition 3. *Let $G = (V, E)$ be an undirected graph, and let $S \subseteq V$ be a set of vertices.*
We say S is a Clique *if for all $u, v \in S$ such that $u \neq v$, $\{u, v\} \in E$.*
We say S is an Independent Set *if for all $u, v \in S$ such that $u \neq v$, $\{u, v\} \notin E$.*
We say S is a Vertex Cover *if for all $\{u, v\} \in E$ such that $u \neq v$, $u \in S$ or $v \in S$.*

We now define some languages. Note that in each of these three languages, it doesn't matter if k is expressed in unary or binary, since whenever k is bigger than the number of vertices, the answer is trivial. The proof below that CLIQUE is **NP**-hard is very clever, but the **NP**-hardness of IndependentSet and VertexCover follows easily from the **NP**-hardness of CLIQUE.

CLIQUE
Instance:
$\langle G, k\rangle$ where G is an undirected graph and k is a positive integer.

Acceptance Condition:
Accept if G has a clique of size $\geq k$.

IndependentSet
Instance:
$\langle G, k\rangle$ where G is an undirected graph and k is a positive integer.

Acceptance Condition:
Accept if G has an independent set of size $\geq k$.

VertexCover
Instance:
$\langle G, k\rangle$ where G is an undirected graph and k is a positive integer.

Acceptance Condition:
Accept if G has a vertex cover of size $\leq k$.

Theorem 6. *CLIQUE is* **NP***-Complete.*

Proof: It is easy to see that CLIQUE $\in$ **NP**.
We will show that 3SAT $\leq_p$ CLIQUE. Let α be an input for 3SAT. Assume that α is an instance of 3SAT, otherwise we can just let $f(\alpha)$ be some string not in CLIQUE.
So $\alpha = \langle F\rangle$ where F is a propositional calculus formula in 3CNF. By duplicating literals if necessary, we can assume that every clause contains exactly 3 literals, and write
$F = (L_{1,1} \vee L_{1,2} \vee L_{1,3}) \wedge (L_{2,1} \vee L_{2,2} \vee L_{2,3}) \wedge \ldots \wedge (L_{m,1} \vee L_{m,2} \vee L_{m,3})$.

We will let $f(\alpha) = \langle G, m\rangle$ where $G = (V, E)$ is the following graph. The idea is that V will have one vertex for each occurrence of a literal in F; there will be an edge between two vertices if and only if the corresponding literals are *consistent*. We define two literals to be *consistent* if there is no propositional letter x such that one of the literals is x and the other is $\neg x$. We define
$V = \{v_{i,j} \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq 3\}$, and
$E = \{\{v_{i,j}, v_{i',j'}\} \mid i \neq \imath' \text{ and } L_{i,j} \text{ is consistent with } L_{i',j'}\}$.
It is easy to see that f is computable in polynomial time.
We claim F is satisfiable $\Leftrightarrow$ G has a clique of size m.

To prove $\Rightarrow$, assume τ is a truth assignment satisfying F.
So there exist literals $L_{1,j_1}, L_{2,j_2}, \ldots, \mathrm{L}_{m,j_m}$ that are all made true by τ, and so every pair of these literals are consistent. So $S = \{v_{1,j_1}, v_{2,j_2}, \ldots, v_{m,j_m}\}$ is a clique in G of size m.

To prove $\Leftarrow$, Let S be a clique in G of size m.
By the definition of G we must be able to write $S = \{v_{1,j_1}, v_{2,j_2}, \ldots, v_{m,j_m}\}$ where L_{i,j_i} is

consistent with $L_{i',j_{i'}}$ for $i \neq i'$.
Define the set of literals $U = \{L_{1,j_1}, L_{2,j_2}, \ldots, L_{m,j_m}\}$.
Define the truth assignment τ as follows on each propositional letter x:
$\tau(x) = \text{TRUE}$ if x is one of the literals in U, otherwise $\tau(x) = \text{FALSE}$.
We claim τ makes each literal in U true (and hence satisfies F):
if L_{i,j_i} is a propositional letter x, then this is true by definition of τ;
If L_{i,j_i} is of the form $\neg x$, then x is not a literal in U (otherwise two literals in U would be inconsistent and so S wouldn't be a clique), so $\tau(x) = \text{FALSE}$, so $\tau(\neg x) = \text{TRUE}$. $\square$

Theorem 7. *IndependentSet is* **NP**-*Complete.*

Proof: It is easy to see that IndependentSet $\in$ **NP**.
We will show that CLIQUE $\leq_p$ IndependentSet.
Let α be an input for CLIQUE, and as above assume that α is an instance of CLIQUE, $\alpha = \langle G, k\rangle$, $G = (V, E)$. Let $f(\alpha) = \langle G', k\rangle$ where $G' = (V, E')$, where E' consists of $\{u, v\}$ such that $u \neq v$ and $\{u, v\} \notin E$.

We leave it as an exercise to show that G has a clique of size $k \Leftrightarrow G'$ has an independent set of size k. $\square$

Theorem 8. *VertexCover is* **NP**-*Complete.*

Proof: It is easy to see that VertexCover $\in$ **NP**.
We will show that IndependentSet $\leq_p$ VertexCover.
Let α be an input for VertexCover, and as above assume that α is an instance of VertexCover, $\alpha = \langle G, k\rangle$, $G = (V, E)$. Let $f(\alpha) = \langle G, |V| - k\rangle$.

We leave it as an exercise to show that a set $S \subseteq V$ is an independent set for G if and only if $V - S$ is an vertex cover. It follows that
G has an independent set of size $k \Leftrightarrow G$ has an vertex cover of size $|V| - k$.$\square$

We now discuss the Hamiltonian Cycle (HamCycle) and Travelling Salesperson (TSP) problems.

In an undirected graph G, a *Hamiltonian cycle* is a simple cycle containing all the vertices. We shall omit here the difficult proof that HamCycle is **NP**-complete.

For 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 edges (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.

HamCycle
Instance:
$\langle G \rangle$ where G is an undirected graph.

Acceptance Condition:
Accept if G has a Hamiltonian 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 9. *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 $e \notin E$.

We leave it as an exercise to show that
G has a Hamiltonian cycle $\Leftrightarrow$ G' has a tour of cost ≤ 0.$\square$

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

We will omit the proof that SubsetSum is **NP**-complete, and instead will concern ourselves now with some consequences of the **NP**-Completeness of SubsetSum.

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 $\sum_{i \in S} a_i = t$.

Recall the Simple and General Knapsack decision problems:

SKD (Simple Knapsack Decision Problem).
Instance:
$\langle w_1, \cdots, w_m, W, B\rangle$ (with all integers nonnegative and represented in binary).

Acceptance Condition:
Accept if there is an $S \subseteq \{1, \cdots, m\}$ such that $B \leq \sum_{i \in S} w_i \leq W$.

GKD (General Knapsack Decision Problem).
Instance:
$\langle (w_1, g_1), \cdots, (w_m, g_m), W, B\rangle$ (with all integers nonnegative represented in binary).

Acceptance Condition:
Accept if there is an $S \subseteq \{1, \cdots, m\}$ such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} g_i \geq B$.

Theorem 10. *The languages SKD, GKD, and SDPDD are all* **NP**-*Complete.*

Proof: It is easy to see that all three languages are in **NP**.
We will show that SubsetSum $\leq_p$ SKD. Since we have already seen that
SKD $\leq_p$ GKD $\leq_p$ SDPDD, the theorem follows from the **NP**-Completeness of SubsetSum.

To prove that SubsetSum $\leq_p$ SKD, 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 SKD. 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 $f(x) = \langle a_1, a_2, \cdots, a_m, t, t\rangle$. It is clear that f is computable in polynomial time, and $x \in$ SubsetSum $\Leftrightarrow f(x) \in$ SKD. $\square$

Related to SubsetSum is the language 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 $\sum_{i \in S} a_i = \sum_{j \notin S} a_j$.

Theorem 11. *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
$x \in \text{SubsetSum} \Leftrightarrow f(x) \in \text{PARTITION}$.

To prove $\Rightarrow$, say that $x \in \text{SubsetSum}$. 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 $f(x) \in \text{PARTITION}$.

To prove $\Leftarrow$, say that $f(x) \in \text{PARTITION}$. 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
$x \in \text{SubsetSum}$.

Case 2: $2t \leq a$. (Exercise.) $\square$

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.

NP-Completeness of Graph 3-Colorability

Before we discuss the problem of graph colorability, it will be convenient to first deal with a language more closely related to SAT, namely SymSAT (standing for "Symmetric SAT").

3SymSAT
Instance:
$\langle F \rangle$ where F is a 3CNF propositional calculus formula with exactly 3 literals per clause.

Acceptance Condition:
Accept if there is a truth assignment τ such that for every clause C of F, τ satisfies at least one literal of C and τ falsifies at least one literal of C.

Theorem 12. *3SymSAT is* **NP**-*Complete.*

Proof: It is easy to see that $\text{3SymSAT} \in \mathbf{NP}$.
We will prove $\text{3SAT} \leq_p \text{3SymSAT}$. Let $\langle F \rangle$ be an instance of 3SAT, and assume that every clause of F contains exactly 3 literals. (If not, we can always "fill out" a clause that contains one or two literals by merely repeating a literal; this will not change the satisfiability of F.) We will now show how to construct (in polynomial time) a 3CNF formula F' such that F is satisfiable if and only if $F' \in \text{3SymSAT}$.

Say that F is $C_1 \wedge C_2 \wedge \ldots \wedge C_m$. The formula F' will contain all the variables of F, as well as a new variable x, as well as a new variable y_i for each clause C_i.
F' will be $D_1 \wedge D_2 \wedge \ldots \wedge D_m$, where D_i is the following 3CNF formula:

Say that C_i is $(L_{i,1} \vee L_{i,2} \vee L_{i,3})$;
then D_i will be $(\neg y_i \vee L_{i,1} \vee L_{i,2}) \wedge (\neg L_{i,1} \vee y_i \vee x) \wedge (\neg L_{i,2} \vee y_i \vee x) \wedge (y_i \vee L_{i,3} \vee x)$.

We wish to show that
F is satisfiable $\Leftrightarrow F' \in$ 3SymSAT.

To show $\Rightarrow$, let τ be a truth assignment that satisfies F. Extend τ to a truth assignment τ' on the variables of F' by letting $\tau'(y_i) = \tau(L_{i,1} \vee L_{i,2})$ and $\tau'(x) =$ "false". Then it is easy to check that τ' satisfies at least one literal of each clause of F', and that τ' falsifies at least one variable of each clause of F'.

To show $\Leftarrow$, let τ be a truth assignment to the variables of F' such that τ satisfies at least one literal of each clause of F' and τ falsifies at least one variable of each clause of F'. If $\tau(x) =$ "false", then it is easy to check that τ satisfies every clause of F. If $\tau(x) =$ "true", then define τ' to be the truth assignment that reverses the value of τ on every variable. Then τ' must also satisfy at least one literal in every clause of F' and falsify at least one literal in every clause of F', and $\tau'(x) =$ "false"; so, as above, τ' satisfies F. $\square$

We will now discuss the problem of graph colorability. Let $G = (V, E)$ be an undirected graph, and let k be a positive integer. A k-coloring of G is a way of coloring the vertices of G such that for every edge $\{u, v\}$, u and v get different colors. 3COL is the language consisting of graphs that can be colored with 3 colors. More formally:

Definition 4. *A* k-coloring *of $G = (V, E)$ is defined to be a function $c : V \rightarrow \{1, 2, \ldots, k\}$ such that for every $\{u, v\} \in E$, $c(v) \neq c(u)$. We say G is k-colorable if there exists a k-coloring of G.*

3COL
Instance:
$\langle G \rangle$ where G is an undirected graph.

Acceptance Condition:
Accept if G is 3-colorable.

Theorem 13. *3COL is* **NP**-*Complete.*

Proof: It is easy to see that 3COL $\in$ **NP**.
We will prove 3SymSAT $\leq_p$ 3COL. Let $\langle F \rangle$ be an instance of 3SymSAT, where
$F = (L_{1,1} \vee L_{1,2} \vee L_{1,3}) \wedge (L_{2,1} \vee L_{2,2} \vee L_{2,3}) \wedge \ldots \wedge (L_{m,1} \vee L_{m,2} \vee L_{m,3})$. We will now construct (in polynomial time) a graph $G = (V, E)$ such that $F \in 3SymSAT \Leftrightarrow G$ is 3-colorable.

V will contain a special vertex w. In addition, for each variable x of F, V will contain a vertex x and a vertex $\neg x$; the vertices $w, x, \neg x$ will form a triangle, that is, we will have (undirected) edges $\{w, x\}, \{x, \neg x\}, \{\neg x, w\}$. It is convenient to think of our three colors as being "white", "true", and "false". Without loss of generality we can choose to always give node w the color "white"; thus, a 3-coloring will always color one of x and $\neg x$ "true" and the other one " false".

For each clause of F we will have three additional nodes. Corresponding to the ith clause of F we will have vertices $v_{i,1}, v_{i,2}, v_{i,3}$; these will be formed into a triangle with edges $\{v_{i,1}, v_{i,2}\}, \{v_{i,2}, v_{i,3}\}, \{v_{i,3}, v_{i,1}\}$. If the ith clause of F is $(L_{i,1} \vee L_{i,2} \vee L_{i,3})$, then we will also have edges $\{L_{i,1}, v_{i,1}\}, \{L_{i,2}, v_{i,2}\}, \{L_{i,3}, v_{i,3}\}$.
(For an example of this construction, see the figure below.)

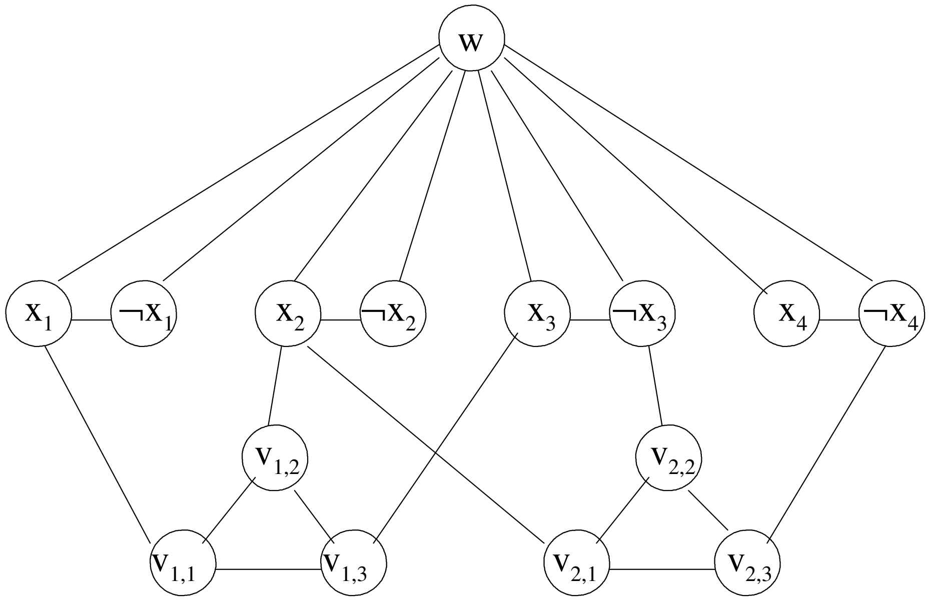
We wish to show that
$F \in$ 3SymSAT $\Leftrightarrow G \in$ 3COL.

To show $\Rightarrow$, let τ be a truth assignment that satisfies at least one literal of every clause of F, and falsifies at least one literal of every clause of F. We now define a mapping
$c : V \rightarrow \{$ "true", "false", "white" $\}$ as follows.
Let $c(w) =$ "white".
For every literal L, let $c(L) = \tau(L)$.

It remains to assign colors to the vertices corresponding to the clauses.
Consider the ith clause.
If $\tau(L_{i,1}) =$ "true" , $\tau(L_{i,2}) =$ "false" , $\tau(L_{i,3}) =$ "true" , then assign
$c(v_{i,1}) =$ "false", $c(v_{i,2}) =$ "true", $c(v_{i,3}) =$ "white";
the result is a (legal) coloring of G.

The other five cases are similar, and are left as an exercise. For example,
if $\tau(L_{i,1}) =$ "true" , $\tau(L_{i,2}) =$ "true" , $\tau(L_{i,3}) =$ "false" , then assign
$c(v_{i,1}) =$ "false", $c(v_{i,2}) =$ "white", $c(v_{i,3}) =$ "true".

To show $\Leftarrow$, let $c : V \rightarrow \{$ "true", "false", "white" $\}$ be a (legal) coloring of G; assume (without loss of generality) that $c(w) =$ "white". We define the truth assignment τ, by $\tau(x) =$ "true" $\Leftrightarrow c(x) =$ "true". We leave it as an exercise to prove that τ satisfies at least one literal of every clause of F, and falsifies at least one literal of every clause of F. $\square$



Graph Constructed From the Formula
$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$