This version `Mon Dec 12 2016`

Easy Shortcut Definitions

If you read and understand only this section, you'll understand $\mathcal{P}$ and $\mathcal{NP}$.

A language L is in the class $\mathcal{P}$ if there is some constant k and some machine M (A machine of any class: Turing machine, RAM, virtual C++ machine, or whatever.) which decides whether a given string w is a member of L, in at most n^k steps, where $n = |w|$.

A language L is in the class $\mathcal{NP}$ if there is some constant k and some non-deterministic machine M such that, given any string $w \in L$, there is a computation of M, with input w, which halts in at most n^k steps, where $n = |w|$, and such that, for any string $w \notin L$, there is no computation of M with input w which halts. Note that, since M is non-deterministic, there could be lots of computations of M with input w that do not halt, even if $w \in L$.

Verifier Definition of $\mathcal{NP}$

There is an equivalent definition of $\mathcal{NP}$ which is much easier to work with, the *verifier* definition, given below.

Let L be a language. Then L is $\mathcal{NP}$ if there is an integer k and a program V (called the verifier) such that

1. The input of V is an ordered pair of strings (w, c).
2. Let $n = |w|$. Then V, with input (w, c), halts within $O(n^k)$ steps, and outputs "0" or "1," regardless of the string c.
3. If $w \notin L$, the output of V with input (w, c) is "0." regardless of c.
4. If $w \in L$, then there is some string c such that $|c| \leq n^k$ and the output of V with input (w, c) is "1." (We call c a *certificate,* or a *witness.*)

Of course, you have to choose your witness carefully.

Example: The independent set problem. For any graph G, a set S of vertices of G is *independent* if no two members of S are neighbors. An instance of the independent set problem is an ordered pair (G, k), where G is a graph and k is an integer. We can write the ordered pair as a string: $\langle G, k \rangle$. Let $L_{IND} = \{\langle G, k \rangle : \text{ G has an independent set of size } k\}$.

For this example, a witness for the ordered pair $\langle G, k \rangle$ is simply an independent set S of k vertices of G, or more properly, a string $\langle S \rangle$ which encodes S.

$\mathcal{P}$ And $\mathcal{NP}$

Throughout, if we say "machine" we mean deterministic machine, unless we specially say "non-deterministic."

For a definition of the classes $\mathcal{P}$-TIME and $\mathcal{NP}$-TIME, usually abbreviated as simply $\mathcal{P}$ and $\mathcal{NP}$, respectively, read the file

http://www.egr.unlv.edu/~larmore/Courses/CSC456/NP/npcomplete.html

(There are, of course, many other discussions of this topic on the web.)

Recall that a language L is in $\mathcal{P}$ if and only if there is some machine that accepts L in polynomial time, and that a language L is in $\mathcal{NP}$ if and only if there is some non-deterministic machine that accepts L in polynomial time.

Theorem 1 *A language L is in $\mathcal{P}$ if and only if there is some machine that decides L in polynomial time.*

Actually, this is usually the definition of the class $\mathcal{P}$.

Definition 1 We say that a language L is *$\mathcal{NP}$-complete* if the following two conditions are satisfied:

1. $L \in \mathcal{NP}$
2. For any $L_2 \in \mathcal{NP}$, there exists a polynomial time reduction of L_2 to L.

Theorem 2 *If L_1 is $\mathcal{NP}$-complete, $L \in \mathcal{NP}$, and there is a polynomial time reduction of L_1 to L, then L is $\mathcal{NP}$-complete.*

Proof: Condition 1 of the definition is given. To prove Condition 2, let $L_2 \in \mathcal{NP}$. Since L_1 is $\mathcal{NP}$-complete, there is a polynomial time reduction of L_2 to L_1. By hypothesis, there is a polynomial time reduction of L_1 to L. Composing these two reductions, we obtain a polynomial time reduction of L_2 to L. □

Theorem 3 *If $L \in \mathcal{P}$ and L is $\mathcal{NP}$-complete, then $\mathcal{P} = \mathcal{NP}$.*

Proof: Trivially, $\mathcal{P} \subseteq \mathcal{NP}$. We need only show that any $L_2 \in \mathcal{NP}$ is in $\mathcal{P}$.

Suppose that $L_2 \in \mathcal{NP}$. Since $L \in \mathcal{P}$, there is a machine M_1 that decides L in polynomial time. Since L is $\mathcal{NP}$-complete, there is a polynomial time reduction of L_2 to L, computed by some machine M_2. Connecting the input of M_1 with the output of M_2, we obtain a machine that decides L_2 in polynomial time. Thus, $L_2 \in \mathcal{P}$. □

Certificates

Given an $\mathcal{NP}$ language L, we define a *certificate system* of L as follows:

1. There is a language L_C, which we call the *certificate language*.
2. Let "$\sharp$" be some symbol which is not in the alphabet of either L or L_C. There is a language $L_V \subset L_C \sharp L$, called the *verification language*, such that $w \in L$ if and only if there is some $x \in L_C$ such that $x \sharp w \in L_V$. We call x a *certificate* of w, because it certifies that $w \in L$.

We define an *$\mathcal{NP}$ certificate system* of L to be a certificate system of L which has the following additional properties:

3. There is a polynomial function $F(n)$ such that, for any $w \in L$ such that $|w| = n$, there is a certificate of w of length at most $F(n)$.

4. The verification language L_V is in $\mathcal{P}$.

We define an $\mathcal{P}$ *certificate system* of L to be a $\mathcal{NP}$ certificate system of L which has the following additional property:

5. There is a machine M which computes some $x \in L_C$ in polynomial time for any input string w, such that x is a certificate for w if $w \in L$.

Theorem 4 *A language L is in $\mathcal{NP}$ if and only if it has an $\mathcal{NP}$ certificate system.*

We first need a definition.

Definition 2 Let M be an NTM. A *configuration* of M is defined to be a string which encodes the state of M, the tape contents of M, and the position of the read/write head of M. The details do not really matter, but we can use the definition given on page 129 of the first edition of *Introduction to the Theory of Computation* by Michael Sipser, or on page 140 of the second edition, or page 168 of the third edition.

If x and y are configurations of M, we say that x *yields* y, which we denote $x \vdash y$, if, given that x is the configuration of M after t steps, y could be the configuration of M after $t + 1$ steps.[1]

Without loss of generality, the symbol "$\vdash$" is not a symbol of the tape alphabet of M, nor the name of a state of M. We define a *valid computation* of M to be a string $x^0 \vdash x^1 \vdash \ldots \vdash x^n$ such that each x^i is a configuration of M, x^0 is a start configuration of M, x^n is an accepting configuration of M, and $x^{t-1} \vdash x^t$ for each t. In addition, we say that this sequence is a *valid computation of M with input w* if x^0 is the start configuration of M on input w. The *time complexity* of the computation is defined to be n.

For convenience in our proofs, we shall say that $x \vdash x$ for any configuration x.

Note that, the way we have defined it, a valid computation is a string, differing slightly from the definition given at the bottom of page 129 of the textbook.

Proof: (of Theorem 6) Suppose that L is a language which has an $\mathcal{NP}$ certificate system. We can design a non-deterministic program that accepts L as follows. First guess $x \in L_C$ in $F(n)$ steps. Then verify that $x\sharp w \in L_V$ in polynomial time.

Conversely, suppose that $L \in \mathcal{NP}$. Let M be an NTM which accepts L in $F(n)$ time, where $F(n)$ is some polynomial time function. Let L_C be the language of all valid computations of M. We say that $x \in L_C$ is a certificate of $w \in L$ if x is a valid computation of time complexity $F(n)$ of M with input w, where $n = |w|$. Note that the length of x is $O(F(n) \cdot (F(n) + n)$, since the space complexity of a valid computation cannot exceed its time complexity, plus the size of the input. Thus, the length of x is a polynomial function of n. Clearly, w has a certificate in L_C if and only if $w \in L$, and there is a polynomial time algorithm that verifies that x is a certificate of w, simply by checking that x is a valid computation of M with input w. Thus, we have constructed an $\mathcal{NP}$ certificate system for L.

This completes the proof of Theorem 4. □

[1] Of course, if M is deterministic, for any x there can be at most one choice of y.

Theorem 5 *A language L is in $\mathcal{P}$ if and only if it has a $\mathcal{P}$ certificate system.*

Proof: Suppose that L has a $\mathcal{P}$ certificate system. If $w \in \Sigma^*$, where Σ is the alphabet of L, use M to compute $x \in L_C$ is polynomial time. If $w \in L$, then it can be verified that $x\sharp w \in L_V$ is polynomial time.

Conversely, suppose $L \in \mathcal{P}$. Let $L_C = \{1\}$, a language with just one string. We say that 1 is a certificate of every $w \in L$. Trivially, that certificate can be computed in polynomial time. Since $L \in \mathcal{P}$, the language $L_V = \{1\sharp w \mid w \in L\}$ is in $\mathcal{P}$. $\square$

Boolean Satisfiability

Generally, we define a *Boolean expression* to be an expression involving variables and operators, where all variables have Boolean type and all operators have Boolean type.[2] A *satisfying assignment* of a Boolean expression is an assignment of truth values (there are only two truth values, *true* and *false*) to each variable so that the value of the expression is *true*. If a Boolean expression has a satisfying assignment, we say it is *satisfiable*; otherwise, we say it is a *contradiction*. The *Boolean satisfiability problem* is to determine that a given boolean expression is satisfiable. This problem is $\mathcal{NP}$-complete, and in fact is our "base" $\mathcal{NP}$-complete problem, the one we shall use to determine that other problems are $\mathcal{NP}$-complete.

Formal Definition of Boolean Satisfiability

We first define a context-free language L_{bool} to consist of all boolean expressions using the following rules:

1. All variables are written in alphanumeric style, *i.e.*, are strings containing letters and digits and starting with a letter.
2. The only operators permitted are
   (a) "+" denoting *or*.
   (b) "$\cdot$" denoting *and*.
   (c) "$\sim$" denoting *not*.
   (d) "=" denoting *if and only if*.
   (e) "$\Rightarrow$" denoting *implies*.

   We apply the following precedence rules:
   (a) "$\sim$" has precedence over all other operators.
   (b) "$\cdot$" has precedence over all operators except "$\sim$"
   (c) "+" has precedence over all operators except "$\sim$" and "$\cdot$"

[2] This definition is much more restrictive than the definition of a Boolean expression in a programming language, which could contain other things, such as the C++ expression "`n == 5`" where `n` is a variable of integer type.

(d) "=" and "⇒" have equal precedence and cannot associate, to avoid confusion. That is, $a = b = c$, $a \Rightarrow b \Rightarrow c$, $a = b \Rightarrow c$, and $a \Rightarrow b = c$ are disallowed.

It is an easy exercise to design a context-free grammar for L_{bool}, using the alphabet consisting of the operator symbols listed above, letters and digits, and right and left parentheses.[3]

Certificates for L_{sat} A certificate for any $w \in L_{\text{sat}}$ is a satisfying assignment. For example, $x = 1, y = 0$ is a certificate for the boolean expression $(x + y) \cdot (\sim x + \sim y)$, while the boolean expression $(x + \sim x)$ has no certificate. Since a certificate for any $w \in L$ is no longer than w itself, and since it can be verified in linear time, we have an $\mathcal{NP}$ certificate system for L_{sat}.

Conjunctive normal form. We say that $w \in L_{\text{bool}}$ is in *conjunctive normal form* (abbreviated CNF) if it is the conjunction of clauses, where each clause is the disjunction of terms, and each term is either a variable or the negation of a variable.

Definition 3

1. *L_{sat} is the subset of L_{bool} consisting of all satisfiable boolean expressions.*
2. *L_{3cnf} is the subset of L_{bool} consisting of all boolean expressions in conjunctive normal form where each clause has exactly three terms.*
3. *$L_{\text{3cnf-sat}} = L_{\text{3cnf}} \cap L_{\text{sat}}$.*

We give context-free grammars for both L_{bool} and L_{3cnf} in the appendix.

Theorem 6 *L_{sat} is $\mathcal{NP}$-complete.*

We define a *simple clause* of a boolean expression to be a boolean expression of one of the following three forms:

1. (v) for some variable v
2. $(\sim v)$ for some variable v
3. $(u = v)$ for some variables u, v

If M is an NTM, then a *configuration* of M is a string of the form xqy, where x and y are strings of tape symbols (including the symbol which represents blank) and q is a state. We can "pad" a configuration, making it longer, by adding as many additional blanks as we wish to y.

A configuration can then be translated into a string over the boolean alphabet $\Sigma = \{0, 1\}$, by fixing binary encodings of the tape symbols and states. It is important for our purposes that there is some fixed k such that each tape symbol and each state is encoded by exactly k binary symbols.

[3] In class, I did not list the operators "=" and "⇒" but I have decided that including these operators makes the later discussion easier.

A string of binary symbols of length n can then be thought of as an ordered n-tuple of truth values, where 0 represents *false* and 1 represents *true*. If $v[1..n]$ is an array of type boolean, we can think of a string of binary symbols of length n as an *assignment* of that array, where $v[i]$ is assigned the truth value corresponding to the i^{th} symbol of the string.

If x, y are binary encodings of configurations of M of the same length, say n, (which can always be made true by padding) then the statement that x yields y, written $x \vdash y$, is then a function of boolean type with $2n$ boolean parameters, which we call `yields(x[1],...x[kn],y[1],...y[kn])`, or simply `yields(x,y)`. For any fixed n, you could write that function using a single statement in the body, which would be a return statement where the right hand side is a boolean expression with $2n$ variables. (You probably wouldn't do it that way, but you could.)

If a configuration is of the form aqb where a, b are tape symbols (including possibly the blank symbol) and $q \in Q$, we say that it is a *window* configuration. We also say that qb is a window configuration. (We need this case to handle the situations where the head is on the rightmost cell.)

Lemma 1 *For any one transition rule of M, there is a boolean function W with $6k$ boolean parameters such that $W(x[1], \ldots x[3k], y[1], \ldots y[3k])$ has the value true if and only if x is a 3-symbol window configuration of M and $x \vdash y$, where that one transition rule is used to derive y from x. Further, W is the conjunction of no more than $5k$ simple clauses.*

Proof: The tape head is over the cell that contains b, and the state is q. If the transition rule is to write c, move right, and change to state r, then the boolean function W will be designed to be true if and only if there is some a such that $x = aqb$ and $y = acr$.

If the transition rule is to write c, move left, and change to state r, then the boolean function W will be designed to be true if and only if there is some a such that $x = aqb$ and $y = rac$.

In either case, W is the conjunction of $5k$ simple clauses, and has $6k$ terms. □

Lemma 2 *For any one transition rule of M which causes a right movement of the head, there is a boolean function W with $4k$ boolean parameters such that $W(x[1], \ldots x[2k], y[1], \ldots y[2k])$ has the value true if and only if x is a 2-symbol window configuration of M and $x \vdash y$, where that one transition rule is used to derive y from x. Further, W is the conjunction of no more than $4k$ simple clauses.*

We omit the proof of Lemma 2, which is similar to that of Lemma 1.

Lemma 3 *If M has R transition rules, there is a boolean function W with $6k$ boolean parameters such that $W(x[1], \ldots x_[3k], y[1], \ldots x[3k])$ has the value true if and only if x is a window configuration of M and $x \vdash y$. Further, W has no more than $6k(R+1)$ terms.*

Proof: Let $W_1, \ldots W_R$ be the boolean functions given by Lemma 1 for each rule. Let W_0 be the boolean function that says that $x = y$, namely $(x[1] = y[1]) \cdot (x[2] = y[2]) \cdot \cdots \cdot (x[3k] = y[3k])$. Now let $W = W_0 + W_1 + \cdots + W_R$. □

Lemma 4 *If M has R transition rules, there is a boolean function W with $4k$ boolean parameters such that $W(x[1], \ldots x_[2k], y[1], \ldots x[2k])$ has the value true if and only if x is a window configuration of M and $x \vdash y$. Further, W has no more than $4k(R+1)$ terms.*

We omit the proof of Lemma 4, which is similar to that of Lemma 3.

Lemma 5 *For each n, there is a boolean function $e(x_1, \ldots x_{kn}, y_1, \ldots y_{kn})$ where, if x, y are binary encodings of configurations of M of the same length n, $e(x[1], \ldots x[kn], y[1], \ldots y[kn])$ is true if and only if $x \vdash y$, and if the last symbol of x is not a state symbol. Furthermore, the function e can be written as a boolean expression whose number of terms is polynomial in n.*

Proof: For each $1 < i < n$, let w_i^x be the substring of x consisting of the three symbols in positions $i-1$ through $i+1$. Let e be the boolean function given by Lemma Lemma 3. Then define $e_i(x[1], \ldots x[kn], y[1], \ldots y[kn])$ to be the function $((x[1] = y[1]) \cdot \cdots \cdot (x[ki - 2k] = y[ki - 2k])) \cdot e(x[ki - 2k + 1], \ldots x[ki + k], y[ki - 2k + 1], \ldots, y[ki + k]) \cdot ((x[ki - 2k + 1] = y[ki - 2k + 1]) \cdot \cdots \cdot (x[kn] = y[kn]))$ which is true if and only if x is a configuration of M, $x \vdash y$, and x has a state symbol in position i. Similarly, using Lemma 2, we can define a function

$e_1(x[1], \ldots x[kn], y[1], \ldots y[kn])$ which is true if and only if x is a configuration of M, $x \vdash y$, and x has a state symbol in position 1. By construction, we can see that each e_i can be written as a boolean expression whose number of terms is polynomial in n. We now define $e = e_1 + e_2 + \cdots + e_{n-1}$. □

Proof: (of Theorem 6.) Our method is to show that, for any NTM M, for any polynomial function F, and for every string w, where we write $n = |w|$, there is a boolean expression $E = E_{M,w,F}$ whose length is $O(F(n)^3 \log(F(n)))$, and which is satisfiable if and only if M accepts w in $F(n)$ time. Furthermore, $E_{M,w,F}$ can be calculated, given input w, in $G(n)$ time for some polynomial function G. Our method will be that E is satisfiable if and only if there is a valid computation of M of length at most $F(n)$ which starts with the initial configuration of M with input w.

We can assume that each configuration has length exactly $F(n) + 2$, no configuration in a valid computation than the length of the computation plus 2, and we can "pad" any shorter configuration by adding blanks. Pick a constant k such that 2^k is at least the sum of the number of states of M and the size of the tape alphabet of M, including the blank symbol. Then, encode all states and tape symbols, including blank, as unique binary strings of length k. Let $N = k(F(n) + 2)$. We can thus represent every configuration of M that could be used in our proof as a binary string of length N.

Write $T = F(n)$. We can assume that a valid computation of at most length T has length exactly T, and thus exactly $T + 1$ configurations, by repeating the accepting configuration as many times as necessary.

A binary string of length N can be considered to be an array of length N of boolean type, *i.e.*, an array of N boolean variables. E will have exactly $N(T+1)$ distinct variables, where the variable $\{v_{t,i}\}$ represents the i^{th} bit of the configuration x^t.

There is then a one-to-one correspondence between assignments of these variables, and strings of the form $x^0 \vdash x^1 \vdash \cdots \vdash x^T$, where each x^t is a boolean string of length N. The boolean expression E will be designed in such a way that a particular assignment of its variables is satisfying if and only if the corresponding string is a valid computation of M with input w.

□

Theorem 7 $L_{\text{3cnf-sat}}$ *is* $\mathcal{NP}$*-complete.*

Proof: We will use Theorems 2 and 6.

It is trivial that $L_{\text{3cnf-sat}} \in \mathcal{NP}$, since a satisfying assignment of any $w \in L_{\text{3cnf-sat}}$ can be verified in linear time. We now give a polynomial time reduction of L_{sat} to $L_{\text{3cnf-sat}}$. We will use the context-free grammar for L_{bool} given in the appendix. Let Σ be the alphabet of L_{bool}. For any $w \in L_{\text{bool}}$, we will construct a string $F(w) \in L_{\text{3cnf}}$ such that $w \in L_{\text{sat}}$ if and only if $F(w) \in L_{\text{3cnf-sat}}$, and such that F can be computed in polynomial time.

To use Theorem 2, we need to compute $F(w)$ for all $w \in \Sigma^*$. If $w \notin L_{\text{bool}}$, we define $F(w) = \epsilon$. Since L_{bool} is a context-free language, this part of the program can be done in polynomial time.

Henceforth, we assume that $w \in L_{\text{bool}}$. Let $\mathcal{T}$ be a parse tree for w, using the grammar given in the appendix. If $\mathcal{N}$ is any internal node of the parse tree, we write leaves$(\mathcal{N}) \in \Sigma^*$ to be the string of all the terminals in the subtree rooted at N, in left-to-right order. Thus, for example, if $\mathcal{N}$ is the root of $\mathcal{T}$, then leaves$(\mathcal{N}) = w$.

We first assign a unique name to each internal node of $\mathcal{T}$, which will be a subscripted grammar symbol. For example, the nodes whose grammar symbol is S will be given the names $S_1, S_2, \ldots$ and so forth. We can assume that S_1 is the root node of $\mathcal{T}$.

Each internal node whose grammar symbol is S, E, T, or F, corresponds to a derivation where the variable grammar symbols have subscripts. For example, the internal node E_5 might correspond to the subscripted derivation $E_5 \to E_6 + T_9$.

We then assign a string variable$(\mathcal{N})$, which we will use as the name of a variable of boolean type, to each internal node of $\mathcal{N}$ of $\mathcal{T}$ whose grammar symbol is S, E, T, F, or V. (We do not assign strings to the internal nodes whose grammar symbols are P, A, or N.) We use the following rules in making this assignment:

1. variable$(S_i) = s\langle i \rangle$ for each i, where $\langle i \rangle$ is the decimal numeral for i. (For example, variable$(S_{27}) = s27$.)
2. variable$(E_i) = e\langle i \rangle$ for each i.
3. variable$(T_i) = t\langle i \rangle$ for each i.
4. variable$(F_i) = f\langle i \rangle$ for each i.
5. variable$(V_i) =$ leaves(V_i)

We call the variables assigned to the S, E, T, and F nodes *internal parse tree variables*.

We now define a string $u \in L_{\text{bool}}$ which consists of the conjunction of a number of clauses, one corresponding to each internal node of the parse tree, plus one more, which we call the *top clause*. Those clauses are defined as follows:

1. $(s1)$ is a clause of u. This is the top clause of u.
2. If there is a subscripted derivation $S_i \to E_j = E_k$, then $(s\langle i \rangle = (e\langle j \rangle = e\langle k \rangle))$ is a clause of u.

3. If there is a subscripted derivation $S_i \to E_j \Rightarrow E_k$, then $(s\langle i\rangle = (e\langle j\rangle \Rightarrow e\langle k\rangle))$ is a clause of u.

4. If there is a subscripted derivation $S_i \to E_j$, then $(s\langle i\rangle = e\langle i\rangle)$ is a clause of u.

5. If there is a subscripted derivation $E_i \to T_j$, then $(e\langle i\rangle = t\langle j\rangle)$ is a clause of u.

6. If there is a subscripted derivation $E_i \to E_j + T_k$, then $(e\langle i\rangle = (e\langle j\rangle + t\langle k\rangle))$ is a clause of u.

7. If there is a subscripted derivation $T_i \to F_j$, then $(t\langle i\rangle = f\langle j\rangle)$ is a clause of u.

8. If there is a subscripted derivation $T_i \to T_j \cdot F_k$, then $(t\langle i\rangle = t\langle j\rangle \cdot f\langle k\rangle)$ is a clause of u.

9. If there is a subscripted derivation $F_i \to\sim F_j$, then $(f\langle i\rangle =\sim f\langle k\rangle)$ is a clause of u.

10. If there is a subscripted derivation $F_i \to (S_j)$, then $(f\langle i\rangle = s\langle j\rangle)$ is a clause of u.

11. If there is a subscripted derivation $F_i \to V_j$, then $(f\langle i\rangle = \text{variable}(V_j))$ is a clause of u.

All the clauses of u other than the top clause we call *internal clauses*. The variable to the left side of the first "=" in an internal clause we call the *left hand side* of the clause. The expression to the right of the first "=" in an internal clause we call the *right hand side* of the clause.

Lemma 6 *u is satisfiable if and only if w is satisfiable.*

Proof: The set of variables in u is the set of variables of w together with the internal parse tree variables. For any assignment of truth values of the variables of w, there is a unique assignment of the internal parse tree variables that makes all the internal clauses true, which can be computed as follows. There is one internal clause for each internal node of the tree labeled S, E, T, or F. Visit these in bottom-up order. For each such clause, we evaluate the right hand side, since the truth values of all variables in the right hand side have been assigned. We then assign that same value to the left hand side of the clause. These assignments, together with the assignments of the truth values of the variables of w, constitute an assignment of the truth values of all variables of u. All internal clauses of u are satisfied by this assignment, so the assignment satisfies u if and only if it satisfies the top clause $(s1)$. But that is true if and only if the original assignment of the variables of w is a satisfying assignment.

This concludes the proof of Lemma 6. □

We now construct a boolean expression v in conjunctive normal form which is equivalent to u, by replacing each internal clause of u with an equivalent boolean expression in conjunctive normal form, and where each clause has at most three terms.

- The top clause (s1) is not replaced.
- A clause of the form $(x = y)$ is replaced by $(x+ \sim y) \cdot (y+ \sim x)$.

- A clause of the form $(x = \sim y)$ is replaced by $(x + y) \cdot (\sim y + \sim x)$.
- A clause of the form $(x = y + z)$ is replaced by $(x + \sim y) \cdot (x + \sim z) \cdot (\sim x + y + z)$.
- A clause of the form $(x = y \cdot z)$ is replaced by $(x + \sim y + \sim z) \cdot (\sim x + y) \cdot (\sim x + z)$.
- A clause of the form $(x = (y = z))$ is replaced by $(x + y + z) \cdot (x + \sim y + \sim z) \cdot (\sim x + y + \sim z) \cdot (\sim x + \sim y + z)$
- A clause of the form $(x = (y \Rightarrow z))$ is replaced by $(x + y) \cdot (x + \sim z) \cdot (\sim x + \sim y + z)$.

Since v is equivalent to u, then, by Lemma 6, v is satisfiable if and only if w is satisfiable. We now define $F(w)$ by expanding each clause of v so that it has exactly three terms, *e.g.*, we replace $(x + y)$ by $(x + x + y)$. Thus, $F(w)$ is satisfiable if and only if w is satisfiable.

Finally, we must show that $F(w)$ can be computed in polynomial time. Suppose $|w| = n$. Then $\mathcal{T}$ has $O(n)$ nodes, and can be computed in $O(n)$ time by a lexical analyzer followed by an LALR parser. Then u can be computed from $\mathcal{T}$ in $O(n)$ time, v can be computed from u in $O(n)$ time, and $F(w)$ can be computed from v in $O(n)$ time.

This concludes the proof of Theorem 7. □

Other $\mathcal{NP}$-Complete Problems

The most common method of proving that a given problem (*i.e.*, language) is $\mathcal{NP}$-complete is to use Theorem 2, where L_1 is taken to be a problem (*i.e.*, language) already known to be $\mathcal{NP}$-complete. The problem $L_{\text{3cnf-sat}}$ is one of the more popular choices used for this purpose.[4]

The Independent Set Problem

Given a graph G and a number k, an *independent set* of G is defined to be a set $\mathcal{I}$ of vertices of G such that no two members of $\mathcal{I}$ are connected by an edge of G. The *order* of $\mathcal{I}$ is defined to be its size, *i.e.*., simply how many vertices it contains.

An instance of the *independent set problem* consists of a graph G and a number k. The question is, "Does G have an independent set of order k?"

The language L_{ind}. Pick an alphabet Σ (without loss of generality, Σ is set of Ascii symbols, including an "end-of-file" symbol) and decide on a method of encoding graphs. For any graph G, let $\langle G \rangle$ be the encoding of G by that method. We insist that $\langle G \rangle$ have a recognizable and unique end of file suffix, *i.e.*, it has a suffix that appears as a substring nowhere else in $\langle G \rangle$. For any number k, let $\langle k \rangle$ be an encoding of k.

Now, let $L_{\text{instance-ind}} = \{\langle G \rangle \langle k \rangle\}$, and let $L_{\text{ind}} \subseteq L_{\text{instance-ind}}$ be the set of all $\langle G \rangle \langle k \rangle$ such that G has an independent set of order k. The purpose of the end of file suffix

[4]The precise definition of the problem described in this handout as $L_{\text{3cnf-sat}}$ differs from book to book, but they are all equivalent.

is to enable a machine reading $w = \langle G \rangle \langle k \rangle \in L_{\text{instance-ind}}$ to tell where $\langle G \rangle$ ends and $\langle k \rangle$ begins.[5]

Theorem 8 *L_{ind} is $\mathcal{NP}$ complete.*

Proof: We first define a certificate system for L_{ind}. For any $w = \langle G \rangle \langle k \rangle \in L_{\text{ind}}$, a certificate for w is an encoding $\langle \mathcal{I} \rangle$ of some independent set $\mathcal{I}$ of G of order k. Since that encoding can be made shorter than $\langle G \rangle$ itself, and since a simple program can easily verify a certificate, we have, by Theorem 4, that $L_{\text{ind}} \in \mathcal{NP}$.

We will use Theorems 2 and 7 to complete the proof, by showing that there is a polynomial time reduction of $L_{\text{3cnf-sat}}$ to L_{ind}. Let Σ be an alphabet which includes the alphabet of L_{3cnf} and also the alphabet of $L_{\text{instance-ind}}$. For any $w \in \Sigma^*$, we will define a string $F(w) \in \Sigma^*$ such that:

1. If $w \in L_{\text{3cnf}}$, then $F(w) \in L_{\text{instance-ind}}$, else $F(w)$ is the empty string.
2. $w \in L_{\text{3cnf-sat}}$ if and only if $F(w) \in L_{\text{ind}}$.

We now proceed to define $F(w)$ for a given $w \in L_{\text{3cnf}}$. We write $w = C_1 \cdot C_2 \cdot \cdots \cdot C_k$, where $C_i = (t_{i,1} + t_{i,2} + t_{i,3})$, where each $t_{i,j}$ is either x or $\sim x$, where x is a string which represents a variable.

We now define a graph $G = (V, E)$, and then define $F(w) = \langle G \rangle \langle k \rangle$. We let V be a set of vertices $\{v_{i,j} | 1 \leq i \leq k, 1 \leq j \leq 3\}$. E is defined as follows:

1. For each i, there is an edge from $v_{i,j}$ to $v_{i,j'}$ for all $1 \leq j < j' \leq 3$.
2. If $t_{i,j} = x$ and $t_{i',j'} = \sim x$ for some string x, there is an edge from $v_{i,j}$ to $v_{i',j'}$.
3. There are no other edges.

We now show that $F(w) \in L_{\text{ind}}$ if and only if w is satisfiable. For each i, let K_i be the subgraph of G consisting of the three vertices $v_{i,1}$, $v_{i,2}$, $v_{i,3}$, and the edges connecting them. Suppose that $\mathcal{I}$ is an independent set of G of order of k. Since $\mathcal{I}$ is independent, it cannot contain more than one vertex from each K_i. Since it has order k, it must then contain exactly one vertex from each K_i. For each i, let $j(i) \in \{1, 2, 3\}$ be defined such that $v_{i,j(i)} \in \mathcal{I}$. We now define an assignment of w, as follows:

1. For any i, if $t_{i,j(i)} = x$ for some variable x, we assign $x =$ *true*.
2. For any i, if $t_{i,j(i)} = \sim x$ for some variable x, we assign $x =$ *false*.
3. If a variable x used in w is not assigned using the above two rules, we assign $x =$ *false*. (Actually, we could assign x either truth value.)

[5] If Σ is the Ascii alphabet, we can require that the end of file suffix be the single end-of-file symbol, and that the encoding of a number be its decimal numeral. Alternatively, we could have said that G and k must be encoded in binary, and that the end of file suffix is the string '111111', or some other string we choose in advance. But whether we use Ascii or binary or some other encoding does not matter.

We claim that the above assignment is well-defined and satisfies w. Suppose the assignment is not well-defined, *i.e.*, the same variable x is assigned to be both *true* and *false*. That implies that $t_{i,j(i)} = x$ and $t_{i',j(i')} = \sim x$, where $i \neq i'$. But this is impossible since, in G, there is an edge from $v_{i,j(i)}$ to $v_{i',j(i')}$. The assignment satisfies w because it satisfies every clause; for each i, C_i is satisfied because the term $t_{i,j(i)}$ is true under the assignment.

Conversely, suppose that w has a satisfying assignment. For each i, at least one term of C_i must be true under the assignment. Let $j(i) \in \{1, 2, 3\}$ be defined such that $t_{i,j(i)}$ is true under the assignment. If more than one term of C_i is true under the assignment, $j(i)$ can be chosen arbitrarily. Now, define $\mathcal{I} = \left\{v_{i,j(i)}\right\}$, which clearly has order k. If $\mathcal{I}$ were not independent, there would have to be an edge between two members of $\mathcal{I}$. Since $\mathcal{I}$ contains just one member of each K_i, this edge would have to be from $v_{i,j(i)}$ to $v_{i',j(i')}$, where $t_{i,j(i)} = x$ and $t_{i',j(i')=\sim x}$ for some variable x, and where $i \neq i'$. Under the given assignment, both of those terms must be true, which is a contradiction.

This completes the proof of Theorem 8. □

The Knapsack Problem

Informally, the knapsack problem is whether there is a subset of a given set of items that fits exactly into a given knapsack. Formally, an instance of the knapsack problem is a finite sequence $x_1, \ldots x_k$ of non-negative numbers and a single number B. This instance is a member of the language L_{Knapsack} if there is some subsequence of $\{x_i\}$ whose sum is B.

Theorem 9 *The knapsack problem is $\mathcal{NP}$-complete.*

Proof: Trivially, the knapsack problem satisfies the verifiability definition of $\mathcal{NP}$.

We reduce the independent set problem to the knapsack problem.

Suppose $\langle G \rangle \langle k \rangle$ is an instance of the independent set problem. Let $v_0, \ldots v_{n-1}$ and $e_0 \ldots e_{m-1}$ be the vertices and edges of G. Our technique is to choose a number B, and to make each vertex and each edge of G into an item, assigning weights in such a way that a set of items S has weight B if and only if the following conditions hold:

1. If e_j is any edge, then e_j is either in S or is incident to a vertex in S, but not both.
2. S contains exactly k vertices.

We assign weights as follows.

- $w(e_j) = 4^j$
- The weight of any vertex is 4^m, plus the sum of the weights of its incident edges.
- $B = 4^m\, k + \sum_{j=0}^{m-1} 4^j$

It helps to write all numbers in base 4. Then $B = k111 \ldots 1111$, where we allow the k on the left end to overflow into other places. The j^{th} edge has weight $0000 \ldots 0001000 \ldots 00000$, where the single digit 1 is in the j^{th} place. The weight of vertex is written as a base 4 numeral with $(\delta + 1)$ 1's, where δ is the degree of the vertex. For example, if $m = 9$ and v_4 is incident to e_2, e_5, and e_6, then $w(v_4)$ is written is base 4 as 1000110010. □

Appendix

Context-free grammar for L_{bool}

The start symbol is S.

$S \to E = E \mid E \Rightarrow E \mid E$
$E \to E + T \mid T$
$T \to T \cdot F \mid F$
$F \to \sim F \mid V \mid (S)$
$V \to AP$
$P \to AP \mid NP \mid \epsilon$
$A \to a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
$N \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Context-free grammar for L_{3cnf}

The start symbol is E.

$E \to E \cdot C \mid C$
$C \to (T + T + T)$
$T \to \sim V \mid V$
$V \to AP$
$P \to AP \mid NP \mid \epsilon$
$A \to a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$
$N \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Completing the Lecture of November 30, 2016

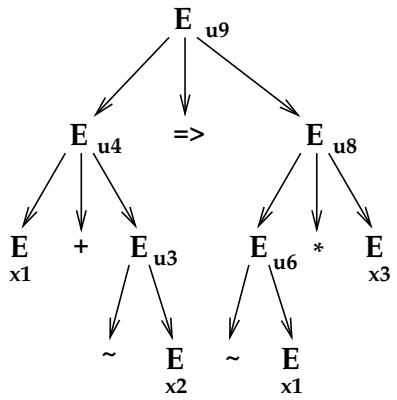
The notation I used during this semester (Fall 2016) differs slightly from the notation in the $\mathcal{NP}$ document, but not so much that you should be confused by it.

The problem was to show a reduction of SAT to 3SAT. In the $\mathcal{NP}$ document I refer to 3SAT $L_{\text{3cnf-sat}}$. That reduction R is completely explained in the $\mathcal{NP}$ document. However, I did not finish with the one example I had on the board on November 30.

That was to give $R(e)$, where:

$$e \equiv (x1+ \sim x2) \Rightarrow \sim x1 * x3$$

The parse tree for e, using an ambiguous version of context-free grammar given in the $\mathcal{NP}$ handout:



Our next step is to translate the parse tree into the conjunction of clauses:

$$(u4 = x1 + u3) * (u3 = \sim x2) * (u6 = \sim x1) * (u8 = u6 * x3) * (u9 = u4 \Rightarrow u8) * u9$$

Finally, we translate into 3-CNF form by replacing each of those clauses by the conjunction of CNF clauses of at most three terms:
$(u4+ \sim x1) * (u4+ \sim u3) * (\sim u4 + x1 + u3) *$
$(u3 + x2) * (\sim u3+ \sim x2) *$
$(u6 + x1) * (\sim u6+ \sim x1) *$
$(u8+ \sim u6+ \sim x3) * (\sim u8 + u6) * (\sim u8 + x3) *$
$(u9 + u4) * (u9+ \sim u8) * (\sim u9+ \sim u4 + u8) *$
$u9$

We can then "pad" each clause of length less than three by duplicating terms, so that each clause has exactly three terms. For example, we can replace $(u4+ \sim x1)$ with $(u4 + u4+ \sim x1)$, and $u9$ with $(u9 + u9 + u9)$.

Reduction of Knapsack to Partition

The knapsack problem can be shown to be $\mathcal{NP}$-complete by reducing 3-SAT to Knapsack. I did not do this in class during the Fall 2016 semester. We can then show that the partition problem is $\mathcal{NP}$-complete by reducing Knapsack to Partition.

Recall that an instance of the Knapsack problem is a number followed by a sequence of positive numbers, namely $\langle K, x_1, x_2, \ldots x_m \rangle$, and the answer to such an instance is **true** if and only if there is some subsequence of $x_1, \ldots x_m$ whose total is K.

Similarly, an instance of the Partition problem is a sequence of positive numbers, namely $\langle y_1, \ldots y_\ell \rangle$, and the answer to such an instance is **true** if and only if there is some subsequence of $y_1, \ldots y_\ell$ whose sum is half the total, *i.e.* $\frac{1}{2}\sum_{j=1}^{\ell} y_j$.

The partition problem is trivially in the class $\mathcal{NP}$, since the solution, if any, can be verified in polynomial time.

Reduction

Given an instance $I_K = \langle K, x_1, x_2, \ldots x_m \rangle$ of Knapsack, define $L = \sum_{i=1}^{m} x_i - K$. Let $R(I_K) = I_P$ be the following instance of Partition:

$$I_P = \langle K + 1, x_1, \ldots x_m, L + 1 \rangle$$

That is, $I_P = \langle y_1, \ldots y_\ell \rangle$ where $\ell = m + 2$, $y_1 = K + 1$, $y_\ell = L + 1$, and $y_j = x_{i+1}$ for all other j.

We need to prove that this reduction works, that is, that I_P is true if and only if I_K is true. There are thus two directions to the proof.

Note that the sum of the items of I_P is $2K + 2L + 2$.

Suppose that I_K is true. There is some subsequence of $\{x_i\}$ whose total is K. The items of this subsequence, together with $L + 1$, have total $K + L + 1$, and thus I_P is true.

Conversely, suppose some subsequence S of $K + 1, x_1, \ldots x_m, L + 1$ has total $K + L + 1$. Clearly that subsequence cannot contain both $K + 1$ and $L + 1$, since they would exceed the needed value. Similarly, and by symmetry, S must contain either $K + 1$ or $L + 1$. Without loss of generality, S contains $L + 1$. The remaining members of S are a subsequence of $x_1, \ldots x_m$ whose total is K, and we are done.