CS 466/666 Notes
NP-Completeness

Over the years several models of computation have been proposed, all essentially permitting a constant amount of information to be performed in "one step". These include

Random Access Machine: our "usual machine model". The memory consists of an arbitrarily large number of words, of fixed size (that we often permit to be large enough to store the problem size, so lg n bits (OK this is not quite "fixed"). The operations include +.-,*,div, and, or, etc as well as comparisons and conditional branching. From these we have most standard programming languages.

Turing Machine (multitape if you like): The storage medium is a tape that is as long as we require. Each square of the tape contains a symbol from some finite alphabet. Initially, the input is written on the first n tape cells. There is a read/write head that, at any time, is over some tape square. The "program" is embodied in a finite state control. Based on the symbol under the read head and the state the control is in, the machine maps to a sate, writes a symbol on the square it is currently inspecting and moves the head left or right or (the entire machine) halts. Note that if we were to restrict the tape length, say to the space needed for the input, then these "linear bounded automata" would still have quite a bit of compute power. However, this linear space bound is clearly an important restriction and it is obvious that there is an algorithm to determine whether or not a "linear bounded automaton" halts on a given input. This is not true for Turing machines in general…1 tape or multipape. Indeed it is fairly easy to prove that a multitape Turing can be simulated by a single tape machine in TS steps, where T denotes the time for the multitape machine and S its storage. This means that a single tape machine can simulate a multitape machine and at most square the run time.

Recursive functions: These are functions over the natural numbers. We start with definitions of 0 and the successor function (i.e. add 1) and some basic conditionals (i.e. test for 0. Then functions are defined recursively. If a recursive function is defined in terms of calls to the function with a smaller parameter, then it is clear that all such functions are well defined (i.e. "halt"). This type of recursion is called "primitive recursion". More generally we can define a function in terms of calls to the same function with larger (as well as smaller) parameter values.

These models can not only simulate one another, but indeed do so without the number of steps growing "too much". In particular Turing machines and Random Action machines can simulate each other without the run time more than squaring.

Another approach to computation is through grammars. You have seen regular grammars (nonterminal ::= terminal; or nonterminal ::= terminal nonterminal) and context free grammars (nonterminal::= non empty string of terminal and non terminals). These forms can easily be generalized to context sensitive (non empty string of non terminals ::= non empty string of terminal and non terminals that is at least as long as the left side) and general (or type 0) grammars (string of non terminals ::= any strong of terminals and non

terminals). With grammars we are clearly doing "language recognition", we accept an input string or we don't. This can be thought of as a Boolean function, and indeed all the previous models can be thought of in the "accept" or not context. There is an interesting relation between these grammars and (Turing-like) machine models, it's called the Chomsky hierarchy. The correspondence is between the grammars and nondeterministic versions of several types of automata. Recall that with a nondeterministic automaton, at any step the machine may have several possible moves, if any sequence of legal moves leads to acceptance, the machine accepts its input. A grammar is very much the same, at any stage in performing a derivation there can be several productions which could be applied. If there is a way to start with the start symbol and apply productions in a legal manner that ultimately generates the string we want … it's in the language. Finite automata (here there is no difference between the power of deterministic and nondeterministic versions) accept exactly the languages generated by regular grammars. Context free grammars correspond to nondeterministic pushdown automata; context sensitive grammars to nondeterministic linear bounded automata; and type 0 grammars correspond to nondeterministic Turing machines. But another twist comes into play. Nondeterministic Turing machines accept exactly the same class of languages as deterministic Turing machines; however the "obvious" simulation can cause the runtime to increase exponentially.

Another point of view of a nondeterministic computation is that of a proof. Often we can think of a nondeterministic computation as "making a few guesses" and verify that they were the right choices to prove the point. For example if I ask whether 944871836856449473 is prime or composite, I simply guess that it is composite and that indeed its factors are 961748941 and 982451653. By the wonders of grade 3 arithmetic we see that this is true. (About 30 years ago Pratt showed that the primes are also recognized quickly by a nondeterministic procedure; of course we now know the primes are recognizable "quickly" by a deterministic machine (Agrawal et al 2004)) Our notion of a "quick", "efficient" or "good" computational procedure will be one that runs in time polynomial in the length of the input.

The Class P

If we are just a bit more careful about the models: Random Access Machine with fixed sized words and Turing Machine, multitape if you like, but deterministic. Then the runtimes differ by "at most" a "squaring or so", so a polynomial time algorithm on one machine gives one on the other.
P is the class of recognition problems (formally languages) for which we have algorithms that take time $O(n^k)$ on a TM or RAM with fixed sized word, where n is the number of bits of input and k is a specific constant for each problem.

The Class NP

NP is the class of recognition problems for which we have procedures that take time $O(n^k)$ on a nondeterministic Turing machine, where n is the number of bits of input and k is a specific constant for each problem. This translates to "there exists a proof of polynomial length, and a polynomial time algorithm to check that proof"

There are a lot of problems in NP that may not appear to be in P. Deciding whether a graph has a Hamiltonian cycle (a simple cycle of length n) is such a problem, hence the generalization to weighted graphs (the Travelling Salesman Problem) is similarly difficult. There is also no known algorithm to determine whether two graphs are isomorphic. We have noted, however, that there is a polynomial time algorithm to recognize the primes. If we are interested in getting polynomial time algorithms for various problems, NP would seem a good place to look. Indeed it is reasonable to ask whether there are problems in NP that are not in P.

Cook's Theorem

In 1971, Steven Cook showed how to construct a Boolean expression from the description of a nondeterministic Turing machine, its runtime and its input such that the expression is true if and only if the Turing machine halts within the stated time bound accepting that input; furthermore the Boolean expression is of size a polynomial in the length of the Turing machine description and the input. This means that a polynomial time algorithm for Boolean expression satisfiablity would give such an algorithm for any problem in NP. We say such a problem is NP-hard. Furthermore, satisfiability is clearly in NP and we say a problem that is in NP and also NP-hard is NP-complete. He also showed a couple of other problems, including subgraph isomorphism, were NP-complete. Cook's original result was actually worded in terms of determining whether a Boolean expression was a tautology (i.e. true for all variable assignments) and his proof dealt with Booleans expressions in Disjunctive Normal Form (DNF), that is "or-ing" together clauses consisting of the "and" of symbols and their negations. ($(a \wedge \neg b \wedge c) \vee (\neg a \wedge \neg b \wedge c)$ is in DNF) Flipping things around, most subsequent work has dealt with satisfiablity of Boolean expressions in Conjunctive Normal Form (CNF) the "and-ing" of clauses consisting of symbols and their negations "or-ed" together. ($(a \vee \neg b \vee c) \wedge (\neg a \vee \neg b \vee c)$ is in DNF.)

Cook's proof (modified as suggested above)

The proof is mechanical … great insight to get it … but no technically hard parts, though admittedly long … and we will just give a sketch:
Inputs: TM description, its input w, of length n and the runtime p(n), a specific polynomial
If the TM accepts w , there is at least one sequence of TM id's
$Q_0, Q_1, \ldots Q_q$ $(q \leq p(n))$ describing
Tape contents
Head Position
State

A few variables help:
C<i,j,t> is 1 iff ith tape cell contains xj at time t $\{1 \leq i \leq p(n),\ 1 \leq j \leq m,\ 0 \leq t \leq p(n)\}$
S<k,t> is 1 iff TM in state qk at time t $\{1 \leq k \leq s,\ 0 \leq t \leq p(n)\}$
H<i,t> is 1 iff head is scanning tape square t at time i $\{\ 0 \leq t \leq p(n)\}$
So $O(p(n)^2)$ variables.

Picky point: times and tape squares represented in binary, so an extra factor of lg n in description size.

Useful formula: Exactly one true
$U(x1,\ldots xr) = (x1+x2+\ldots xr)\ (\Pi_{i\neq j}(\neg xi+\neg xj))$ (size $O(r^2)$)

We now construct one big formula, ABCDEFG, that is true iff the TM accepts the input:
The first 4 say "it looks like a Turing machine to me"
A asserts that TM scans exactly one square at time t, $\forall t$. So $A = \Pi A_t$
$A_t = U(H<1,t>, H<2,t> .., H<p(n),t>)$
B asserts each tape square has exactly 1 symbol in it at time t, $\forall t$.
C asserts TM is in 1 state at time t, $\forall t$.
D asserts at most one tape square (the correct one) changes from 1 step to the next
$D = \Pi_{i,j,t}\ [C<i,j,t> \equiv C<i,j,t+1>+H<i,t>]$

E asserts the moves of the TM from one ID to the next are allowed by the next move function:
E_{ijkt} asserts at least one of:
- i^{th} cell does not contain symbol j at time t
- Head is not on i^{th} cell at time t
- M is not in state k at time t
- ID of TM obtained from previous ID by legal transformation

F asserts initial conditions are OK,
$F = S<1,0>\ H<1,0>\ \Pi_{1\leq i\leq n}\ C<i,j_i,0>\ \Pi_{n<i\leq p(n)}\ C<i,1,0>$
j_i indicates i^{th} input symbol

G asserts TM eventually enters accept state, i.e. $G=S< q_s,p(n)>$ where q_s is accept state.
Indeed entire formula size is $O(p(n)^3)$

As noted, Cook also showed subgraph isomorphism is as hard as SAT.
Karp ('72) showed about a dozen other problems were in the same class
Then … the term NP-Complete
Note: NP and CoNP, hence NP-Complete and Co-NP-Complete
NP-hard
Then came 1000's of problems
And what to do with them

We have proved that SAT is NP-Complete. Indeed, if you look at the proof, you see that only a special case of SAT is used, namely the product (and) of sums (or) of literals (variables and their negations). This $\Pi\ \Sigma$ (literals) form is called **conjunctive normal form.** (Similarly, the $\Sigma\ \Pi$ (literals) form is called **conjunctive normal form**) One can convert an arbitrary Boolean formula to either of these forms, but it may result in a dramatic (exponential) increase in size. The first question we ask, though, is how much more we can restrict the form of a Boolean expression and still have its Sat problem NP-

complete. Curiously, 3 literals per clause is as hard as the general problem. We can reduce general CNF-SAT to CNF-3SAT (which we will simply call 3-SAT). Consider an arbitrary CNF expression $\Pi\, C_i$, where $C_i = (a_{i1} \vee a_{i2} \vee \ldots a_{ik})$ and a_{ij} denotes a literal. The reduction is quite straightforward and causes only a linear increase in size.
Write a clause $C = (a_1 \vee a_2 \vee \ldots a_k)$ as $(a_1 \vee a_2 \vee b_1)\ \Pi_{i=1,..k-4}\ (\neg b_i \vee a_{i+2} \vee b_{i+1})\ (\neg b_{k-3} \vee a_{k-1} \vee a_k)$. Here the b's are new variables not used anywhere else in the construction. If C is made true by assigning a_j true, then the new form is made true by making a_j true, all b_i that occur earlier than the clause where a_j are made true and all that come later than this clause are made false.
So how about just 2 literals per clause? Things become dramatically easier:
2-Sat is easily solved in polynomial time. Simply make an arbitrary assignment of one variable. This satisfies some clauses and forces the assignment of other variables if certain other clauses are to be satisfied. The process of forced assignment and clause elimination continues until we either eliminated some clauses and no remaining variables are forced, or we have a contradiction (the same variable is forced to be true in one clause and false in another). In the former case, we keep the assignments and repeat the process until no clauses are remain unsatisfied. In the latter case, our tentative assignment does not work, so we must make the opposite one … which goes through a similar forcing process and leads either to the conclusion that the expression is not satisfiable or finds a satisfying assignment. It is not hard to eliminate a single variable in O(n), but indeed then entire process can be done in linear time.

Input: CNF expression of n clauses each contains 2 literals
(A literal is a Boolean variable or its negative)

Output: A satisfying assignment of variables or, essentially, a proof if these are none.

Note: There are most 2n variables

Data Structures:
For each variable x_i create
doubly linked list of clauses containing x_i (without ¬) and
doubly linked list of clauses containing $\neg\, x_i$.

Each clause representation has 2 sets of flags to give status of each variable (T, F,?) and the clauses (T,?)

Algorithm:
Choose an arbitrary variable x_i from a clause still unsatisfied. Simultaneously (e.g. you could alternate steps) run 2 processes one makes tentative assignment x_i =T, the other makes x_i =F, (Data structures marked independently by the two)

All clauses made true by the assignment are flagged as such. For all clauses in which the negation of the assigned value occurs, the other literal of that clause is forced to hold. Continue with these forced assignments (successively) until either:

i) All assignments have been made; hence every clause is satisfied or has no assignment to either term.

or

ii) We discover some variable is forced to both T & F. Hence a contradiction to original assignment.

Case (i): Halt the other process and run through it again undoing all its markings. (This takes same amount of time as making assignment). Assignments made by the properly terminating process are "made permanent". This leaves us with a subset of clauses & variables, all these remaining clauses in original form. The time taken is the same for each process; the number of clauses removed is proportional to time taken, as a clause is inspected at most twice by properly termination process. Reapply procedure to remaining clauses starting with some remaining variables (dual) assignment.

Case (ii) Terminating assignment leads to contradiction. Let other assignment process continue to completion. Satisfaction if it finds one.