The Classes P and NP

We now shift gears slightly and restrict our attention to the examination of two families of problems which are very important to computer scientists. These families constitute the bulk of our practical computational problems and have been central to the theory of computation for years.

The first is a class which contains all of the problems we actually solve using computers. If we think about the problems we normally present to the computer, we note that not too many computations require more than $O(n^3)$ or $O(n^4)$ time. In fact, most of the important algorithms are somewhere in the linear to $O(n^3)$ range. With this in mind, we state that practical computation resides within polynomial time bounds. There is a name for this class of problems.

> **Definition.** *The class of* ***polynomially solvable problems****:* ***P,*** *contains all sets in which membership may be decided by an algorithm whose running time is bounded by a polynomial.*

Besides containing all of what we have decided to consider practical computational tasks, the class P has another attractive attribute. Its use allows us to not worry about our machine model since all reasonable models of computation (including programs and Turing machines) have time complexities (running times) which are polynomially related.

That was the class of problems we actually compute. There is another important class. This one is the class of problems that we would love to solve but are unable to do so exactly. Since that sounds strange, let's look at an example: final examination scheduling. A school has n courses and five days in which to schedule examinations. The optimal schedule would be one where no student has to take two examinations on the same day. This seems like an easy problem. But, there are $O(5^n)$ possible different schedules. If we looked at them all with a computer which could check a million schedules every second, the time spent checking for a value of n = 50 would be more than

200,000,000,000,000,000,000 years!

Yes, that's a long time. Obviously this will not be done between registration and the end of the semester.

One might wonder if the above analysis was needed, because after all, who would look at *all* of the schedules? You should only need to check a few of the obvious ones. Or do you? Think back over all of the examination schedules you have seen. Were there *any* which were optimal? No! This indicates that there must be a problem somewhere.

Let us think a little more about examination schedules. While it might be very difficult to find a good one, it is easy to check a schedule to see how near perfect it is. This process is called *verification* and allows us to know almost immediately if we stumble upon a good schedule.

Consider another problem, that of finding a minimal length tour of n cities where we begin and end at the same place. (We call this the closed city tour problem.) Again, there are many solutions, in fact, there are n-1 factorial different tours. But, once more, if we have a tour, we can easily check to see how long it is. Thus if we want a tour of less than some fixed length, we can quickly check candidates to see if they qualify.

It is very interesting to have some problems that seem to require exponential time to solve, but candidates for solution can be checked or verified rather quickly. This provides some hope of solving problems of this kind. If we can determine the worth of an answer, then maybe we can investigate promising solutions and keep the best one.

Let us consider a class of problems which all seem very complex, but have solutions which are easily checked. Here is a class which contains the problems for which solutions can be *verified* in polynomial time.

> **Definition.** *The class of* ***nondeterministic polynomially acceptable problems: NP,*** *contains all sets in which membership can be verified in polynomial time.*

This may seem to be quite a bizarre collection of problems. But think for a moment. The examination scheduling problem does fit here. If we were to find a solution, it could be checked out very quickly. Lots of other problems fall into this category. Closed tours of groups of cities for instance. Plus, many graph problems used in CAD algorithms for computer chip design. Also, most scheduling problems. This turns out to be a very interesting and useful collection of problems.

One might wonder about the time actually involved in solving membership in this class. The only known relationship between NP and deterministic time is the following result.

Theorem 1. *For every set A in NP there is a polynomial p(n) such the problem of whether a data item of size n is a member of A can be solved in* $2^{p(n)}$ *time.*

A useful tool in studying the relationships between members of a class is to translate or map one into another. If we can translate one set into another, we can often deduce properties of one by the properties that we know the other possesses. This is called reducibility, is pictured in Figure 1, and defined below.

Definition. *The set A is* ***many-one polynomial-time reducible*** *to the set B (to be written as* $A \leq_p B$*) if and only if there is a recursive function g(x) which can be computed in polynomial time such that for all x:* $x \in A$ *if and only if* $g(x) \in B$.

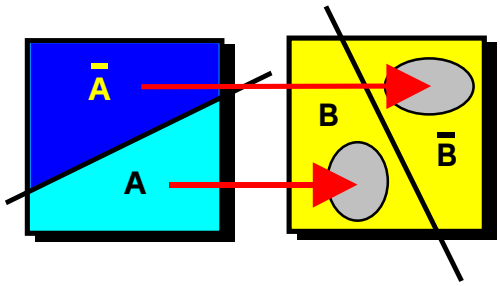

Figure 1 - A many to one mapping between sets

Note that all of the members of A map into a portion of B and all elements not in A map into a part of B's complement. This gives us a way to solve membership in A if we know how to solve membership in B. If A is reducible to B via the function g(x), then all we need do to determine if x is in A is to check to see if g(x) is in B.

One of the properties preserved by reducibility is complexity. Recall that to decide whether x was in A, we had to:

a) Compute g(x), and
b) Check to see if g(x) was in B.

Thus the complexity of deciding membership in A is the sum of the complexities of computing g(x) and deciding membership in B. If computing g(x) does not take very long then we can say that A is no more complex than A. From this discussion we can state the following theorem.

Theorem 2. *If* $A \leq_p B$ *and B is in P, then A is in P also.*

And of course if $A \leq_p B$ and B is in *NP*, then A is in *NP* for exactly the same reasons. This brings up another concept.

> **Definition.** *The set A is **hard** for a class of sets if and only if every set in the class is many-one polynomial time reducible to A.*

Note that if some members of the class require at least polynomial time to decide, then the set A is more complex than any of the members of the class it is hard for. Thus an *NP*-hard set would be more difficult to decide membership in than any set in *NP*. If it were also in *NP*, then it would be the most complex set in the class. We have a name for this.

> **Definition.** *A set is **complete** for a class if and only if it is a member of the class and hard for the class.*

The corollary to the following theorem is a very important observation about NP-complete sets and polynomial reducibilities. It will be our major tool in proving sets *NP*-complete. It follows from the fact that polynomial time reducibility is transitive. (That is, if $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$.)

> **Theorem 3.** *If $A \leq_p B$ and A is NP-hard, then B is NP-hard also.*

> **Corollary.** *If $A \leq_p B$ for a set B in NP, and A is NP-complete, then B is NP-complete also.*

Polynomial reducibilities also may be used to place upper bounds upon sets in P. For example, the following result is based on this.

> **Theorem 4.** *If A is NP-complete then A is a member of P if and only if P is the same class as NP.*

> **Proof.** Almost obvious. If A is a member of P then every set polynomially reducible to A is also in P. Thus the NP-completeness of A forces every single one of the sets in NP to be members of P.
>
> On the other hand, if P = NP then of course A is a member of P as well..

This is very interesting. If we know that one NP-complete set can be decided in polynomial time then we know that *every set in NP* can be decided using some polynomial algorithm! Since we know of no sub-exponential algorithms for membership in any of these sets this means that we can get them all for the price of one. But, it is felt that this is highly unlikely since nobody has done it yet and the problem has been around for a while.

Of course for any of the above discussion to be worthwhile we need to see an NP-complete set. Or at least prove that there is one. The following definitions from the propositional calculus lead to our first NP-complete problem.

> **Definition.** *A **clause** is a finite collection of **literal**s, which in turn are Boolean variables or their complements.*

> **Definition.** *A clause is **satisfiable** if and only if at least one literal in the clause is true.*

Suppose we examine the clause $(v_1, v_3, \overline{v_7})$ which comes from the set of Boolean variables $\{ v_1, \ldots, v_n \}$. This clause is satisfiable if either v_1 or v_3 are true or v_7 is false. Now let us look at the collection of clauses:

$$(v_1, \overline{v_2}, v_3) \; (v_2) \; (\overline{v_1}, v_3)$$

All three are true (at once) when all three variables are true. Thus we shall say that a collection of clauses is *satisfiable if and only if there is some assignment of truth values to the variables which makes all of the clauses true simultaneously*. The collection:

$$(v_1, \overline{v_2}, v_3) \; (v_2) \; (\overline{v_2}, \overline{v_3}) \; (\overline{v_1}, v_3)$$

is not satisfiable because at least one of the three clauses will be false no matter how the truth values are assigned to the variables. Now for the first decision problem which is NP-complete. It is central to theorem proving procedures and the propositional calculus.

> **The Satisfiability Problem (SAT).** *Given a set of clauses, is there an assignment of truth values to the variables such that the collection of clauses is satisfiable?*

Since some collections are satisfiable and some are not, this is obviously a nontrivial decision problem. And it just happens to be NP-complete! By the way, it is not the general satisfiability problem for propositional calculus, but the *conjunctive normal form* satisfiability problem. Here is the theorem and its proof.

> **Theorem 5.** *The satisfiability problem is NP-complete.*

> **Proof Sketch.** The first part of the proof is to show that the satisfiability problem is in NP. This is simple. A machine which checks this merely

jots down a truth value for each Boolean variable in a nondeterministic manner, plugs these into each clause, and then checks to see if one literal per clause is true. A Turing machine can do this as quickly as it can read the clauses.

The hard part is showing that every set in NP is reducible to the satisfiability problem. Let's start. First of all, if a set is in NP then there is some one tape Turing machine M_i with alphabet = {0, 1, *b*} which recognizes members (i.e., verifies membership) of the set within time p(n) for a polynomial p(n). What we wish is to design a polynomial time computable recursive function $g_i(x)$ such that:

$$M_i \text{ recognizes } x \quad \text{iff } g_i(x) \in \text{SAT}.$$

For $g_i(x)$ to be a member of SAT, it must be some collection of clauses which contain at least one true literal per clause under some assignment of truth values. This means that g_i must produce a logical expression which states that M_i accepts x. Let us recall what we know about computations and arithmetization. Now examine the following collections of assertions.

a) When M_i begins computation:
- **#x is on the tape,**
- **the tape head is on square one, and**
- **instruction *I1* is about to be executed.**

b) At each step of M_i 's computation:
- **only one instruction is about to be executed,**
- **only one tape square is being scanned, and**
- **every tape square contains exactly one symbol.**

c) At each computational step, the instruction being executed and the symbol on the square being scanned completely determine:
- **the symbol written on the square being read,**
- **the next position of the head, and**
- **the next instruction to be executed.**

d) Before p(n) steps, M_i is in a halting configuration.

These assertions tell us about the computation of $M_i(x)$. So, if we can determine how to transform x into a collection of clauses which mean exactly the same things as the assertions written above, we have indeed found our $g_i(x)$. And, if $g_i(x)$ is polynomially computable we are done.

First let us review our parameters for the Turing machine M_i. It uses the alphabet {0, 1, *b*, #} (where # is used only as an endmarker) and has m instructions. Since the computation time is bounded by the polynomial p(n) we know that only p(n) squares of tape may be written upon.

Now let us examine the variables used in the clauses we are about to generate. There are three families of them. For all tape squares from 1 to p(n) and computational steps from time 0 to time p(n), we have the collection of Boolean variables of the form

HEAD[s, t] which is true if M_i has its tape head positioned on tape square s at time t.

(Note that there are $p(n)^2$ of these variables.) For the same time bounds and all instructions, we have the variables of the form

INSTR[i, t] which is true if M_i is about to execute instruction number i at time t.

There are only m∗p(n) of these variables. The last family contains variables of the form

CHAR[c, s, t] which is true if character c in {0, 1, *b*, #} is found upon tape square s at time t.

So, we have $O(p(n)^2)$ variables in all. This is still a polynomial.

Now let's build the clauses which mean the same as the above assertions. First, the machine must begin properly. At time 0 we have #x on the tape. If x = 0110 then the clauses which state this are:

(CHAR[#,1,0]), (CHAR[0,2,0]), (CHAR[1,3,0]),
(CHAR[1,4,0]), (CHAR[0,5,0])

and blanks are placed upon the remainder of the tape with:

(CHAR[*b*,6,0]), ... , (CHAR[*b*,p(n),0]).

Since the machine begins on square one with instruction 1, we also include:

(HEAD[1,0]), (INSTR[1,0]).

That finishes our first assertion. Note that all of the variables in these clauses must be true for $g_i(x)$ to be satisfiable since each clause contains

exactly one literal. This starts $M_i(x)$ off properly. Also note that there are $p(n)+2$ of these particular one variable clauses.

(NB. We shall keep count of the total number of literals used so far as we go so that we will know $|g_i(x)|$.)

During computation one instruction may be executed at each step.
But, if the computation has halted then no more instructions can be executed. To remedy this we introduce a bogus instruction numbered 0 and make M_i switch to it whenever a halt instruction is encountered. Since M_i remains on instruction 0 from then on, at each step *exactly* one instruction is executed.

The family of clauses (one for each time $t \leq p(n)$) of the form:

$$(INSTR[0,t], INSTR[1,t], \ldots , INSTR[m,t])$$

maintain that M_i is executing *at least* one instruction during each computational step. There are $(m+1)*p(n)$ literals in these. We can outlaw pairs of instructions (or more) at each step by including a clause of the form:

$$(\overline{INSTR[i,t]}\ ,\ \overline{INSTR[j,t]}\ )$$

for each instruction pair i and j (where $i < j$) and each time t. These clauses state that no *pair* of instructions can be executed at once and there are about $p(n)*m^2$ literals in them.

Clauses which mandate the tape head to be on one and only one square at each step are very much the same. So are the clauses which state that exactly one symbol is written upon each tape square at each step of the computation. The number of literals in these clauses is on the order of $p(n)^2$. (So, we still have a polynomial number of literals in our clauses to date.)

Now we must describe the action of M_i when it changes from configuration to configuration during computation. Consider the Turing machine instruction:

I27:	**0**	**1**	**right**	**I42**
	1	**1**	**halt**	

Thus if M_i is to execute instruction 27 at step 239 and is reading a 0 on square 45 we would state the following implication:

if (INSTR[27,239] *and* HEAD[45,239] *and* CHAR[0,45,239])
then (CHAR[1,45,240] *and* HEAD[46,240] *and* INSTR[42,240]).

Recalling that the phrase (*if* A *then* B) is equivalent to (*not*(A) *or* B), we now translate the above statement into the clauses:

($\overline{\text{INSTR[27,239]}}$, $\overline{\text{HEAD[45,239]}}$, $\overline{\text{CHAR[0,45,239]}}$, CHAR[1,45,240])
($\overline{\text{INSTR[27,239]}}$, $\overline{\text{HEAD[45,239]}}$, $\overline{\text{CHAR[0,45,239]}}$, HEAD[46,240])
($\overline{\text{INSTR[27,239]}}$, $\overline{\text{HEAD[45,239]}}$, $\overline{\text{CHAR[0,45,239]}}$, INSTR[42,240]).

Note that the second line of instruction 27 contains a halt. In this case we switch to instruction 0 and place the tape head on a bogus tape square (square number 0). This would be something like:

($\overline{\text{INSTR[27,239]}}$, $\overline{\text{HEAD[45,239]}}$, $\overline{\text{CHAR[1,45,239]}}$, CHAR[1,45,240])
($\overline{\text{INSTR[27,239]}}$, $\overline{\text{HEAD[45,239]}}$, $\overline{\text{CHAR[1,45,239]}}$, HEAD[0,240])
($\overline{\text{INSTR[27,239]}}$, $\overline{\text{HEAD[45,239]}}$, $\overline{\text{CHAR[1,45,239]}}$, INSTR[0,240])

(These clauses are not very intuitive, but they do mean exactly the same as the *if-then* way of saying it. And besides, we've got it in clauses just like we needed to. This was quite convenient.)

In general, we need trios of clauses like the above for every line of each instruction, at every time, for all of the tape squares. Again, $O(p(n)^2)$ literals are involved in this.

To make sure that the rest of the symbols (those not changed by the instruction) remain on the tape for the next step, we need to state things like:

if ($\overline{\text{HEAD[45,239]}}$ *and* CHAR[0,45,239]) *then* CHAR[0,45,240],

which become clauses such as:

(HEAD[45,239], $\overline{\text{CHAR[0,45,239]}}$, CHAR[0,45,240]).

These must be jotted down for each tape square and each symbol, for every single time unit. Again, we have $O(p(n)^2)$ literals.

When M_i halts we pretend that it goes to instruction 0 and place the head on square 0. Since the machine should stay in that configuration for the rest of the computation, we need to state for all times t:

$$(\overline{\text{INSTR}[0,t]}, \text{INSTR}[0,t+1])$$
$$(\overline{\text{HEAD}[0,t]}, \text{HEAD}[0,t+1])$$

(this was another *if-then* statement) and note that there are O(p(n)) literals here.

One more assertion and we are done. Before p(n) steps, M_i must halt if it is going to accept. This is an easy one since the machine goes to instruction 0 only if it halts. This is merely the clause

$$(\text{INSTR}[0,p(n)]).$$

Of course this one must be true if the entire collection of clauses is to be satisfiable.

That is the construction of $g_i(x)$. We need to show that it can be done in polynomial time. So, think about it. Given the machine and the time bound p(n), it is easy (long and tedious, but easy) to read the description of the Turing machine and generate the above clauses. In fact we could write them down in a steady stream as we counted to p(n) in loops such as the following.

```
for time = 1 to p(n)
    for tape square = 1 to p(n)
        for scratch parameter = 1 to p(n)
            jot down the necessary clauses
```

So, computing $g_i(x)$ takes about as much time to compute as it does to write it down. Thus its complexity is $O(|g_i(x)|)$. The same as the length of all of the literals in the clauses. Since there are $O(p(n)^2)$ of these and the length of a literal will not exceed $\log_2(p(n))$ we arrive at polynomial time complexity for the computation of $g_i(x)$.

The remainder of the proof is to show that

$$M_i \text{ accepts } x \text{ iff } g_i(x) \in \text{SAT}.$$

While not completely trivial it does follow from an examination of the definitions of how Turing machines operate compared to the satisfiability of the clauses in the above construction. The first part of the proof is to argue that if M_i accepts x, then there is a sequence of configurations which M_i progresses through. Setting the HEAD, CHAR, and INSTR variables so that they describe these configurations makes the set of clauses computed by $g_i(x)$ satisfiable. The remainder of the proof is to

argue that if $g_i(x)$ can be satisfied then there is an accepting computation for $M_i(x)$.

That was our first NP-complete problem. It may not be quite everyone's favorite, but at least we have shown that one does indeed exist. And now we are able to state a result having to do with the P = NP question in very explicit terms. In fact the satisfiability problem has become central to that question. And by the second corollary, an aid to proving NP-completeness.

Corollary. *SAT is in P if and only if P = NP.*

Corollary. *If A Y NP and SAT $\leq_p$ A then A is NP-complete.*

So, all we need to do is determine the complexity of the satisfiability problem and we have discovered whether P and NP are the same. Unfortunately this seems much easier said than done!