Intractable Problems

One of the claims made in the last section was that there are lots and lots of NP-complete problems that are of interest to the practical computer scientist. Now it is time to fulfill this prophecy and demonstrate this. We shall examine some of the popular NP-complete problems from various computational areas.

Logicians should be quite pleased that satisfiability for the propositional calculus is NP-complete. It means that they will still be needed to prove theorems since it seems unlikely that anyone will develop a computer program to do so. But we, as computer scientists need to see problems that are closer to home. This is also more than a theoretical exercise because we know that any problem which is NP-complete is a candidate for approximation since no subexponential time bounded algorithms are known for these problems.

First, we shall review the process of proving a problem NP-complete. We could do it from scratch like we did for SAT. But that is far too time consuming, especially when we have a nifty technique like reduction. All we need to do is:

a) show that the problem is in NP,
b) reduce an NP-complete problem to it, and
c) show that the reduction is a polynomial time function.

That's not too bad at all. All we basically must accomplish is to transform an NP-complete problem to a new one. As a first example, let us simplify satisfiability by specifying exactly how many literals must be in each clause. Then we shall reduce this problem to others.

> **Satisfiability with 3 literals per clause (3-SAT).** *Given a finite set of clauses, each containing exactly three literals, is there some truth assignment for the variables which satisfies all of the clauses?*

> **Theorem 1.** *3-SAT is NP-complete.*

> **Proof.** We know that since 3-SAT is merely a special case of SAT, it must be in NP. (That is, we can verify that a truth assignment satisfies all of the clauses as fast as we can read the clauses.)
>
> To show that it 3-SAT hard for NP, we will reduce SAT to it by transforming any instance of the satisfiability problem to an instance of

3-SAT. This means we must demonstrate how to convert clauses that do not contain exactly three literals into ones that do. It is easy if a clause contains two literals. Let us take (x_1, x_2) as an example. This is equivalent to the pair:

$$(x_1,x_2,u),(x_1,x_2,\overline{u})$$

where u is a new variable. Note that each clause of the pair contains exactly three literals and that.

So far, so good. Now we will transform clauses such as (x) which contain one literal. This will require two steps. We begin by converting it to the pair of two literal clauses:

$$(x,u_1),(x,\overline{u_1})$$

much as before. Then we change each of these just as before and get:

$$(x,u_1,u_2),(x,u_1,\overline{u_2}),(x,\overline{u_1},u_2),(x,\overline{u_1},\overline{u_2})$$

This was easy. (But you'd better plug in all possible truth-values for the literals and fully check it out.)

One case remains. We might have a clause such as $(x_1, \ldots, x_k)$ which contains more than three literals. We shall arrange these literals as a cascade of three literal clauses. Consider the sequence of clauses:

$$(x_1,x_2,u_1),(x_3,\overline{u_1},u_2),\ldots,(x_{k-2},\overline{u_{k-4}},u_{k-3}),(x_{k-1},x_k,\overline{u_{k-3}})$$

Let us look at this. If the original clause were satisfiable then one of the x_i's had to be true. Let us set all of the u_i's to true up to the point in the sequence where x_i was encountered and false thereafter. A little thought convinces us that this works just fine since it provides a truth assignment that satisfies the collection of clauses. So, if the original clause was satisfiable, this collection is satisfiable too.

Now for the other part of the proof. Suppose the original clause is not satisfiable. This means that all of the x_i's are false. We claim that in this case the collection of clauses we constructed is unsatisfiable also. Assume that there is some way to satisfy the sequence of clauses. For it to be satisfiable, the last clause must be satisfiable. For the last clause to be satisfied, u_{k-3} must be false since x_{k-1} and x_k are false. This in turn forces u_{k-4} to be false. Thus all of the u_i's all the way down the line have got to be false. And when we reach the first clause we are in big trouble

since u_1 is false. So, if the x_i's are all false there is nothing we can do with the truth-values for the u_i's that satisfies all of the clauses.

Note that the above transformation is indeed a polynomial time mapping. Thus SAT $\leq_p$ 3-SAT and we are done.

One of the reasons that showing that 3-SAT is NP-complete is not too difficult is that it is a restricted version of the satisfiability problem. This allowed us to merely modify a group of clauses when we did the reduction. In the future we shall use 3-SAT in reductions and be very pleased with the fact that having only three literals per clause makes our proofs less cumbersome.

Of course having only two literals per clause would be better yet. But attempting to change clauses with three literals into equivalent two literal clauses is very difficult. Try this. I'll bet you cannot do it. One reason is because 2-SAT is in P. In fact, if you could reduce 3-SAT to 2-SAT by translating clauses with three literals into clauses with two literals, you would have shown that P = NP.

Let us return to introducing more NP-complete problems. We immediately use 3-SAT for the reduction to our next NP-complete problem that comes from the field of mathematical programming and operations research. It is a variant of integer programming.

0-1 Integer Programming (0-1 INT). *Given a matrix A and a vector b, is there a vector x with values from {0, 1} such that $Ax \geq b$?*

If we did not require the vector x to have integer values, then this is the linear programming problem and is solvable in polynomial time. This one is more difficult.

Theorem 2. *0-1 INT is NP-complete.*

Proof. As usual it is easy to show that 0-1 INT is in NP. Just guess the values in x and multiply it out. (The exact degree of the polynomial in the time bound is left as an exercise.)

A reduction from 3-SAT finishes the proof. In order to develop the mapping from clauses to a matrix we must change a problem in logic into an exercise in arithmetic. Examine the following chart. It is just a spreadsheet with values for the variables x_1, x_2, and x_3 and values for some expressions formed from them.

Expressions	Values							
X_1	0	0	0	0	1	1	1	1
X_2	0	0	1	1	0	0	1	1
X_3	0	1	0	1	0	1	0	1
$+ X_1 + X_2 + X_3$	0	1	1	2	1	2	2	3
$+ X_1 + X_2 - X_3$	0	-1	1	0	1	0	2	1
$+ X_1 - X_2 - X_3$	0	-1	-1	-2	1	0	0	-1
$- X_1 - X_2 - X_3$	0	-1	-1	-2	-1	-2	-2	-3

Above is a table of values for arithmetic expressions. Now we shall interpret the expressions in a logical framework. Let the plus signs mean *true* and the minus signs mean *false*. Place *or*'s between the variables. So, $+x_1 + x_2 - x_3$ now means that

x_1 is *true*, *or* x_2 is *true*, *or* x_3 is *false*.

If 1 denotes *true* and 0 means *false*, then we could read the expression as $x_1=1$ *or* $x_2=1$ *or* $x_3=0$.

Now note that in each row headed by an arithmetic expression there is a minimum value and it occurs exactly once. Find exactly which column contains this minimum value. The first expression row has a zero in the column where each x_i is also zero. Look at the expression. Recall that $+x_1 + x_2 + x_3$ means that at least one of the x_i should have the value 1. So, the minimum value occurs when the expression is *not satisfied*.

Look at the row headed by $+x_1 - x_2 - x_3$. This expression means that x_1 should be a 1 or one of the others should be 0. In the column containing the minimum value this is again not the case.

The points to remember now for each expression row are:

a) Each has *exactly* one column of minimum value.
b) This column corresponds to a nonsatisfying truth assignment.
c) Every other column satisfies the expression.
d) All other columns have higher values.

Here is how we build a matrix from a set of clauses. First let the columns of the matrix correspond to the variables from the clauses. The rows of the matrix represent the clauses - one row for each one. For each clause, put a 1 under each variable that is not complemented and a -1 under those that are. Fill in the rest of the row with zeros. Or we could say:

$$a_{i,j} = \begin{cases} 1 \text{ if } v_j \in \text{ clause i} \\ -1 \text{ if } \overline{v_j} \in \text{ clause i} \\ 0 \text{ otherwise} \end{cases}$$

The vector b is merely made up of the appropriate minimum values plus one from the above chart. In other words:

b_i = 1 - (the number of complemented variables in clause i).

The above chart provides the needed ammunition for the proof that our construction is correct. The proper vector x is merely the truth assignment to the variables which satisfies all of the clauses. If there is such a truth assignment then each value in the vector Ax will indeed be greater than the minimum value in the appropriate chart column.

If a 0-1 valued vector x does exist such that $Ax \geq b$, then it from the chart we can easily see that it is a truth assignment for the variables which satisfies each and every clause. If not, then one of the values of the Ax vector will always be less than the corresponding value in b. This means that the that at least one clause is not satisfied for any truth assignment.

Here is a quick example. If we have the three clauses:

$$(x_1, \overline{x_3}, x_4), (\overline{x_2}, \overline{x_3}, x_4), (x_1, x_2, x_3)$$

then according to the above algorithm we build A and b as follows.

$$\begin{bmatrix} 1 & 0 & -1 & 1 \\ 0 & -1 & -1 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \quad \\ \quad \\ \quad \\ \quad \end{bmatrix} \geq \begin{bmatrix} 0 \\ -1 \\ 1 \end{bmatrix}$$

Note that everything comes out fine if the proper values for the x_i are put in place. If x_3 is 0 then the first entry of Ax cannot come out less than 0 nor can the second ever be below -1. And if either x_2 or x_1 is 1 then the third entry will be at least 1.

Problems in graph theory are always interesting, and seem to pop up in lots of application areas in computing. So let us move to graph theory for our next problem.

CLIQUE. *Given a graph and an integer k, are there k vertices in the graph which are all adjacent to each other?*

This does not sound like a very practical problem, does it? Interesting, yes, but practical? Consider this. Suppose that you had a graph whose nodes were wires on a silicon chip. And there was an edge between any two nodes whose wires might overlap if placed on the same horizontal coordinate of the chip. Finding the cliques tells the designer how much horizontal room is needed to route all of the wires.

Theorem 3. *CLIQUE is NP-complete.*

Proof. Again, it is easy to verify that a graph has a clique of size k if we guess the vertices forming the clique. We merely examine the edges. This can be done in polynomial time.

We shall now reduce 3-SAT to CLIQUE. We are given a set of k clauses and must build a graph, which has a clique if and only if the clauses are satisfiable. The literals from the clauses become the graph's vertices. And collections of true literals shall make up the clique in the graph we build. Then a truth assignment which makes at least one literal true per clause will force a clique of size k to appear in the graph. And, if no truth assignment satisfies all of the clauses, there will not be a clique of size k in the graph.

To do this, let every literal in every clause be a vertex of the graph we are building. We wish to be able to connect true literals, but not two from the same clause. And two that are complements cannot both be true at once. So, connect all of the literals that are not in the same clause and are not complements of each other. We are building the graph G = (V, E) where:

$$V = \{\langle x, i\rangle \mid x \text{ is in the i-th clause}\}$$
$$E = \{(\langle x, i\rangle, \langle y, j\rangle) \mid x \neq \overline{y} \text{ and } i \neq j\}$$

Now we shall claim that if there were k clauses and there is some truth assignment to the variables which satisfies them, then there is a clique of size k in our graph. If the clauses are satisfiable then one literal from each clause is true. That is the clique. Why? Because a collection of literals (one from each clause) which are all true cannot contain a literal and its complement. And they are all connected by edges because we connected literals not in the same clause (except for complements).

On the other hand, suppose that there is a clique of size k in the graph. These k vertices must have come from different clauses since no two literals from the same clause are connected. And, no literal and its complement are in the clique, so setting the truth assignment to make the literals in the clique true provides satisfaction.

A small inspection reveals that the above transformation can indeed be carried out in polynomial time. (The degree will again be left as an exercise.) Thus the CLIQUE problem has been shown to be NP-hard just as we wished.

One of the neat things about graph problems is that asking a question about a graph is often equivalent to asking quite a different one about the graph's complement. Such is the case for the clique problem. Consider the next problem that inquires as to how many vertices must be in any set that is connected to or *covers* all of the edges.

Vertex Cover (VC). *Given a graph and an integer k, is there a collection of k vertices such that each edge is connected to one of the vertices in the collection?*

It turns out that if a graph with n vertices contains a clique consisting of k vertices then the size of the vertex cover of the graph's complement is exactly n-k. Convenient. For an example of this, examine the graphs in figure 1. Note that there is a 4-clique (consisting of vertices a, b, d, and f) in the graph on the left. Note also that the vertices not in this clique (namely c and e) do form a cover for the complement of this graph (which appears on the right).

Since the proof of VC's NP-completeness depends upon proving the relationship between CLIQUE and VC, we shall leave it as an exercise and just state the theorem.

Theorem 4. *VC is NP-complete.*

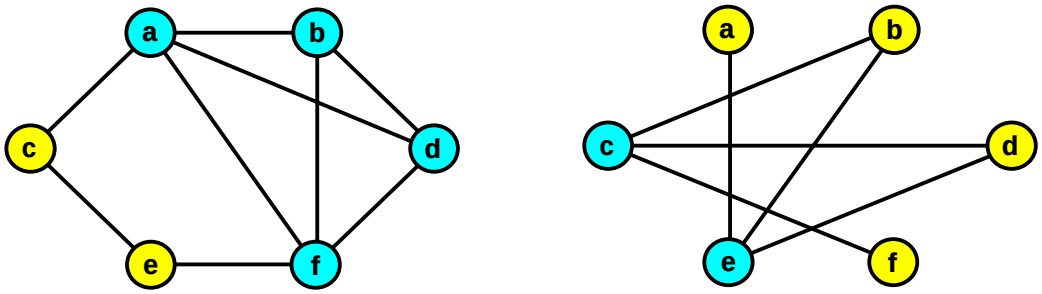

Figure 1 - A graph and its complement.

On to another graph problem. This time we shall examine one of a very different nature. In this problem we ask about coloring the vertices of a graph so that adjacent ones are distinct. Here is the definition.

Chromatic Number (COLOR). *Given a graph and an integer k, is there a way to color the vertices with k colors such that adjacent vertices are colored differently?*

This is the general problem for coloring. A special case, map coloring can always be done with four colors. But as we shall see presently, the general problem is NP-complete when we must use more than four colors.

Theorem 5. *COLOR is NP-complete.*

Proof. To show that COLOR is in NP, again just guess the method of coloring vertices and check it out.

We shall reduce 3-SAT to COLOR. Suppose that we have r clauses that contain $n \geq 3$ variables. We need to construct a graph that can be colored with n+1 colors if and only if the clauses are satisfiable.

Begin by making all of the variables $\{v_1, \ldots, v_n\}$ and their complements vertices of the graph. Then connect each variable to its complement. They must be colored differently, so color one of each pair *false* and the other *true*.

Now we will force the *true* colors to be different from each other. Introduce a new collection of vertices $\{x_1, \ldots, x_n\}$ and connect them all together. The n x_i's now form a clique. Connect each x_i to all of the v_j and their complements except when i = j. Thus if we have n different *true* colors (call them $t_1, \ldots, t_n$) we may color the x_i's with these. And, since neither v_j or its complement is connected to x_i one of these may also be colored with t_i. So far we have colored:

a) each x_i with t_i,
b) either v_i or its complement with t_i, and the other *false*.

An example for three variables is depicted in figure 2. Since shades of gray are difficult to see, we have used three for the *true* colors and have drawn as squares all of the vertices to be colored with the false color. Note that v_1 and v_2 are true while v_3 is false.

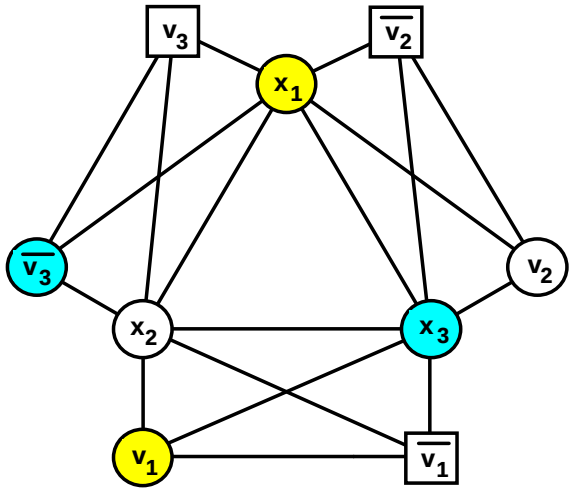

Figure 2 - Variables, True and False Colors

So far, so good. We have constructed a graph that cannot be colored with fewer than n+1 colors. And, the coloring scheme outlined above is the only one that will work. This is because the x_i's must be different colors and either v_i or its complement has to be the (n+1)-st (*false*) color.

3-SAT enters at this point. Add a vertex for each clause and name them c_1, ... , c_r. Connect each of them to all the variables and their complements except for the three literals that are in the clause. We now have the following edges in our graph for all i and j between 1 and n, and k between 1 and r, except where otherwise noted.

a) $< v_i, \overline{v_i} >$,

b) $< x_i, x_j >$ for all $i \neq j$,

c) $< v_i, x_j >$ and $< \overline{v_i}, x_j >$ for all $i \neq j$, and

d) $< v_i, c_k >$ and $< \overline{v_i}, c_k >$ for literals not in c_k

Here's a recap. One of each variable and complement pair must be *false* and the other, one of the *true* colors. These *true*'s must be different because the x_i's form a clique. Then, the clauses (the c_i's) are connected to all of the literals *not* in the clause.

Suppose that there is a truth assignment to the variables which satisfies all of the clauses. Color each true literal with the appropriate t_i and color its complement *false*. Examine one of the clauses (say, c_j). One of its literals must have been colored with one of the *true* colors since the clause is satisfied. The vertex c_j can be colored that way too since it is not connected to that literal. That makes exactly n+1 colors for all the vertices of the graph.

If there is no truth assignment which satisfies all of the clauses, then for each of these assignments there must be a clause (again, say c_i) which has all its literals colored with the *false* or (n+1)-st color. (Because otherwise we would have a satisfying truth assignment and one of each literal pair must be colored false if n+1 colors are to suffice.) This means that c_i is connected to vertices of every *true* color since it is connected to all those it does not contain. And since it is connected to all but three of the literal vertices, it must be connected to a vertex colored *false* also since there are at least three variables. Thus the graph cannot be colored with only n+1 colors.

Since constructing the graph takes polynomial time, we have shown that 3-SAT $\leq_p$ COLOR and thus COLOR is NP-complete.

An interesting aspect of the COLOR problem is that it can be almost immediately converted into a scheduling problem. In fact, one that is very familiar to anyone who has spent some time in academe. It is the problem of scheduling final examinations that we examined earlier.

Examination Scheduling (EXAM). *Given a list of courses, a list of conflicts between them, and an integer k; is there an exam schedule consisting of k dates such that there are no conflicts between courses which have examinations on the same date?*

Here is how we shall set up the problem. Assign courses to vertices, place edges between courses if someone takes both, and color the courses by their examination dates, so that no two courses taken by the same person have the same color.

We have looked at seven problems and shown them to be NP-complete. These are problems that require exponential time in order to find an optimal solution. This means that we must approximate them when we encounter them. There just happen to be many more in areas of computer science such as systems programming, VLSI design, and database systems. Thus it is important to be able to recognize them when they pop up. And, since their solutions are related, methods to approximate them often work for other problems.

In closing, here are three more NP-complete problems.

Closed Tour (TOUR). *Given n cities and an integer k, is there a tour, of length less than k, of the cities which begins and ends at the same city?*

Rectilinear Steiner Spanning Tree (STEINER). *Given n points in Euclidean space and an integer k, is there a collection of vertical and horizontal lines of total length less than k that spans the points?*

Knapsack. *Given n items, each with a weight and a value, and two integers k and m, is there a collection of items with total weight less than k, which has a total value greater than m?*