Computational Complexity

Ian Holyer

1984

Abstract

The computational complexity of a problem is intended to be a measure of how difficult the problem is to solve. Three aspects of this discussed in the course are:

- Formal complexity measures – what properties they should have and how they compare.
- Practical complexity measures
- Practical problems – design and analysis of algorithms.
- NP-Completeness – an attempt to distinguish between problems with practical algorithms and those without.

You may find the following sources of information useful.

"An Overview of the Theory of Computational Complexity"
Hartmanis and Hopcroft
Journal of the ACM, Vol. 18, No. 3, July 1971, pp. 444-475

This is a review paper on the general properties shared by all complexity measures.

"The Design and Analysis of Computer Algorithms"
Aho, Hopcroft and Ullman (Addison-Wesley)

Don't confuse this with other books by the same authors (or subsets of them). This is an easily available general book which concentrates on (2) above. The bibliographic notes at the end of each chapter are useful, though the book is getting a bit old.

"Computers and Intractability"
Garey and Johnson (Freeman)

This is a more recent book which specialises in NP-completeness. It contains a large catalogue of problems which have been shown to be NP-complete.

Chapter 1

Formal Measures of Computational Complexity

1.1 The Definition of a Complexity Measure

In fact we cannot measure the (computational) complexity of a problem, only the complexity of a particular algorithm for solving the problem. It turns out that there are problems with no fastest algorithm, and so we cannot define the complexity of a problem as the complexity of its fastest algorithm. In addition, different measures of complexity yield different results (unlike different definitons of computability), and there are no obvious "correct" measures. Even different versions of the Turing Machine lead to different complexity measures. Thus we need to say what measures are allowed, what properties they have, and how they compare.

The first step in defining a complexity measure is to decide how the information in an instance of a problem, and the result, are to be represented or encoded as integers, so that the problem may be regarded as a number-theoretic function of one argument.

Next, we must decide how algorithms are to be represented, for instance as Turing Machines. The representation of an algorithm will presumably be finite, leading to an effective enumeration of all algorithms (and thus of all partial recursive functions). We use $a_0, a_1, a_2, \ldots$ to denote both the (representations of) the algorithms and the partial recursive functions they compute. An enumeration $a_0, a_1, a_2, \ldots$ is an effective enumeration of algorithms if, given i and n, we can find a_i and simulate its behaviour on argument n. In other words, the function $f(i, n) = a_i(n)$ is partial recursive.

Then for each algorithm and each argument for which the algorithm successfully computes a value, there must be a number c measuring the complexity of the computation. This number c must be calculable in the sense that given a bound m, we can follow the computation until either the computation stops and we have found c, or we are sure that $c > m$ (possibly undefined). This leads us to a definition of complexity measure as follows.

Definition 1.1 *A computational complexity measure A consists of an effective enumeration $a_0, a_1, a_2, \ldots$ of algorithms, together with corresponding measuring functions $A_0, A_1, A_2, \ldots$ such that*

1. *$A_i(n)$ is defined iff $a_i(n)$ is defined.*

2. The function M below is a recursive function:

$$M(i,n,m) = \begin{array}{ll} 1 & \text{if } A_i(n) \leq m \\ 0 & \text{otherwise} \end{array}$$

Note that if we find that $A_i(n) \leq m$, we can then calculate $A_i(n)$ and $a_i(n)$ because of the effectiveness of the enumeration a_i.

The first question we should ask is whether this definition includes the most obvious and natural complexity measures. Certainly the time (number of steps) taken by a Turing Machine fits this definition if we take a_i to be the i'th TM and $A_i(n)$ to be the number of steps taken by a_i on argument n. The function M is recursive since we can follow the action of a_i on argument n for m steps and put $M(i,n,m) = 1$ if the computation has stopped.

Another natural measure is the amount of space taken by a TM, say measured as the number of tape cells scanned during a computation. This fits in with (1) above provided we say that the number of cells scanned is undefined if the computation doesn't stop. However, it is not immediately clear that (2) holds, since a TM computation may continue indefinitely without scanning more than m cells. At any stage of such a computation, the cells which have been scanned form a contiguous block including the starting cell, so the computation remains within the $2m-1$ cells centred on the starting cell. Thus an instantaneous description of the computation consists of the current state, currently scanned cell, and the contents of these $2m-1$ cells. There are only a finite number of possible instantaneous descriptions, and so if we simulate a computation as a sequence of such descriptions, one of the following must happen:

1. The computation terminates, and we can determine the number $A_i(n)$ of cells used.
2. A step of the computation takes us out of the $2m-1$ cells we have been monitoring, in which case we know $A_i(n) > m$ (or undefined).
3. A description is repeated, in which case we know that the computation cycles indefinitely and $A_i(n)$ is undefined.

Now we know that the definition is not too restrictive because it includes TM time- and space-complexity. The next question is whether the definition is restrictive enough to exclude trivial measures. Bounded measures, such as $A_i(n) = 0$ if $a_i(n)$ is defined, are excluded by the following theorem.

Theorem 1.1 *Let A be any complexity measure and let g be any recursive function. Then there is a recursive function f such that any algorithm a_i for f satisfies $A_i(n) > g(n)$ for large n (all $n \geq$ some N_i).*

Proof. The idea of the proof is to simulate the algorithms a_i and to set $f \neq a_i$ at suitable moments. We then say that a_i has been dealt with. As time goes on, we can simulate more algorithms, and we can simulate them for more steps. When calculating $f(n)$, we simulate the first n algorithms for $g(n)$ steps. Let a_i be the first computation which has stopped within $g(n)$ steps, and which has not been dealt with previously, if any.

$$f(n) = \begin{array}{ll} 0 & \text{if all } a_i \text{ with } A_i(n) \leq g(n) \text{ have been dealt with} \\ 0 & \text{if } a_i(n) > 0 \\ 1 & \text{if } a_i(n) = 0 \end{array}$$

The function f is recursive since g is recursive and for each $i \leq n$ we can simulate $a_i(n)$ for $g(n)$ steps. Suppose a_i is an algorithm for f. There is an $N_i \geq i$ such that all $a_j (j < i)$ have been dealt with (if they are ever dealt with) by the time the value $f(N_i)$ is defined. Then the definition ensures that $A_i(n) > g(n)$ for all $n \geq N_i$.
□

There are thus arbitrarily complex functions which only have 0 or 1 for values – the complexity is not bounded by the values taken. On the other hand the values taken can be shown to be bounded by a recursive function of the complexity.

The Speed Up Theorem

Now we want to prove the "Speed Up Theorem" which tells us that there are functions with no fastest algorithm. In a way, this is no surprise, as we can speed up the computation of any function by having a table of the first few values. This increases the size of the algorithm and increases the speed for small arguments, but may or may not increase the speed for large arguments. In fact not all functions can be speeded up in the manner of the Speed Up Theorem, and in any case the speeded up algorithm cannot be found effectively. First we will prove it for one specific measure, Turing Machine space complexity, and then show how to extend the result to any measure.

Theorem 1.2 (Speed Up Theorem For Space Complexity.) *Let A be the complexity measure defined as the number of cells used in a Turing Machine computation, and let $g(n)$ be any recursive function. Then there is a function $f(n)$ such that given any algorithm a_i for f, there is another algorithm a_j for f with*

$$g(A_j(n)) \leq A_i(n) \quad \text{for large } n$$

Thus if $g(n) = 2^n$, we get $A_j \leq log(A_i)$. This could be repeated indefinitely.

Sketch Proof. Again we simulate the first n algorithms for a number of steps. Instead of using a uniform bound $g(n)$ for the number of steps, we use a different bound $g_i(n)$ for each algorithm a_i. The bounds g_i are discussed in more detail below. They decrease rapidly with i so that f is more likely to have "large" algorithms. We define f to be unequal to the first computation $a_i, i \leq n$, which has stopped within $g_i(n)$ cells and which has not been dealt with previously.

$$\begin{array}{rcll} f(n) & = & 0 & \text{if all } a_i \text{ with } A_i(n) \leq g_i(n) \text{ dealt with} \\ & & 0 & \text{if } a_i(n) > 0 \\ & & 1 & \text{if } a_i(n) = 0 \end{array}$$

The function f is recursive since we can calculate $g_i(n)$ for each *ilen*, and simulate the $a_i(n)$'s for $g_i(n)$ steps. Now if a_i is any algorithm for f, we have

$$A_i(n) > g_i(n) \quad \text{for large } n \ (\forall n \geq N_i) \tag{1.1}$$

Next we want to show that given k, we can calculate f within g_k cells, i.e. given k there is an algorithm a_j for f with

$$A_j(n) \leq g_k(n) \quad \text{for large } n \tag{1.2}$$

Note that j is likely to be much larger than k. The algorithm has a table of answers for $a_0, a_1, \ldots a_k$, and simulates $a_{k+1}, a_{k+2}, \ldots$ For each $i \leq k$, if a_i is ever dealt with, let N_i be the argument for which $f(N_i)$ is made unequal to $a_i(N_i)$. Let N be $max(N_i)$ over those i, so that if a_i is dealt with for $i \leq k$, it happens for an argument $\leq N$. Notice that although we know N exists, we cannot calculate it.

The TM for f stores, in its finite state control, details of all computations $a_i(n)$ within $g_i(n)$ cells for $i \leq k$ and $n \leq N$. For each $n \leq k$, the TM can thus immediately print out $f(n)$. If $n > k$, the TM acts as follows. It simulates each TM a_i for $k < i \leq n$, for each argument $m \leq n$, allowing $g_i(m)$ cells, determining the order in which the a_i are dealt with.

For this algorithm to be possible within $g_k(n)$ cells, we need to re-use the same space for the different simulations, we need $g_{k+1}, g_{k+2}, \ldots << g_k$, we need $g_i(n)$ to be "tape-constructible" so that we can mark out a block of $g_i(n)$ cells without going outside those cells, and then check that subsequent computations stay within the block, and we need the TM's carefully ordered so that simulation does not take up too much space. See Hartmanis and Hopcroft for details.

The g_i are arranged so that $g(g_{i+1}(n)) < g_i(n)$ for large n. Thus the g_i can be pictured roughly as

$$\begin{array}{llllllll} g_0 & 0, & g, & g^2, & g^3, & g^4, & g^5, & \ldots \\ g_1 & 0, & 0, & g, & g^2, & g^3, & g^4, & \ldots \\ g_2 & 0, & 0, & 0, & g, & g^2, & g^3, & \ldots \\ g_3 & 0, & 0, & 0, & 0, & g, & g^2, & \ldots \\ \ldots & & & & & & & \end{array}$$

Suppose that a_i is an algorithm for f. Then by equation 1.1 we have $A_i > g_i$ for large n. Let a_j be an algorithm for f as above with $A_j \leq g_{i+1}$. We have

$$g(A_j) \leq g(g_{i+1}) < g_i < A_i \quad \text{for large } n$$

as required, assuming that g is monotonic increasing.
□

To prove the Speed Up Theorem more generally, we need to show that any two complexity measures A and B are recursively related, so that "easy" problems in one measure are "easy" in all measures. To avoid discussion about comparing algorithms represented in different ways, we assume that the enumerations $\{a_i\}, \{b_i\}$ have been suitably adjusted so that a_i and b_i compute the same function.

Theorem 1.3 *Let A and B be any two complexity measures for which a_i and b_i compute the same partial recursive function. Then there is a recursive function $r(n, m)$ such that*

$$A_i(n) \leq r(n, B_i(n)) \quad \text{for large } n$$

Proof. We simulate the first n algorithms b_i for m steps and put

$$r(n, m) = max(A_i(n)) \quad \text{over } i \leq n \text{ with } B_i(n) \leq m$$

This is recursive because if $B_i(n) \leq m$ then $b_i(n)$ is defined, so $a_i(n)$ is defined, so we can calculate $A_i(n)$. Then $A_i(n) \leq r(n, B_i(n))$ whenever $n \geq i$ and either side is defined. Note that we can replace $r(n, m)$ by just $r(m)$ if $B_i(n) \geq n$.

□

Theorem 1.4 (General Speed Up Theorem.) *Let A be any measure and g any recursive function. Then there is a recursive function f such that if a_i is any algorithm for f, there is an algorithm a_j for f such that*

$$g(A_j(n)) \leq A_i(n) \quad \text{for large } n$$

Proof. We can suppose that $g(n)$ is monotonic increasing (otherwise replace it by $max(g(i))$ over $i \leq n$). Let B be the TM space complexity measure, and assume that a_i and b_i compute the same partial recursive function, that $A_i(n) \geq n$ and $B_i(n) \geq n$. Then the previous theorem guarantees increasing recursive functions r and s with

$$A_i \leq r(B_i) \;\; \textit{and} \;\; B_i \leq s(A_i) \quad \text{for large } n$$

Using the Speed Up Theorem for TM space complexity, we can find f such that given any algorithm a_i for f, there is an algorithm a_j for f with

$$s(g(r(B_j))) \leq B_i \quad \text{for large } n$$

Then

$$s(g(A_j)) \leq s(g(r(B_j))) \leq B_i \leq s(A_i) \quad \text{for large } n$$

giving

$$g(A_j) \leq A_i \quad \text{for large } n$$

as required.
□

The results of this section may not have been particularly surprising, but the fact that they follow from such a weak definition of complexity measure is surprising. For more results about general complexity measures, and further properties which are desirable for them, see Hartmanis and Hopcroft.

Chapter 2

Practical Complexity Measures

2.1 Representation of Problem Instances

In this chapter we are interested in complexity measures which capture the informal notion of the time taken to solve a problem. We want to regard the complexity as a function of the size of an instance of a problem, and this brings us back to the question of how instances are represented or encoded as integers.

The convention we take is that the size of an instance is the amount of information (number of TM cells) needed to specify the instance in a natural but *compact* way. Thus if the problem is a number theoretic one such as "is the number n prime" the argument n must be represented in a compact way, say in some number base b without leading zeros, in which case the representation requires about $log_b(n)$ digits. We never worry about constant factors, so the base of the logarithm does not matter. The complexity of such a number-theoretic problem is regarded as a function of $log(n)$. In the case of graphs, one natural compact representation is as a list of edges, each edge being a pair of vertex labels. If the graph has v vertices and e edges, and numbers 1 to v are used as vertex labels, the size of the instance will be about $e * log(v)$. If the graphs are dense, we might prefer an adjacency matrix giving a size of about v^2.

2.2 Turing Machine and RAM Complexity Measures

One of the powerful intuitive arguments suggesting that Turing Machines compute all computable functions is that we can liken a TM to a mathematician with a large notebook and eraser. Her actions are determined by her state of mind and the contents of the page open in front of her. Her basic actions are to change what is on the page, turn the page and/or change her state of mind. This argument also suggests that the number of steps taken by a TM is a good time-complexity measure. TM time-complexity is thus taken as a yardstick against which to compare other time-complexity measures. When comparing functions, we use "big O" notation as follows.

Definition 2.1 *We say $f(n) = O(g(n))$ to mean there is a constant k such that $f(n) \leq k * g(n)$ for all sufficiently large n.*

Different versions of the TM model give time-complexities differing usually only quadratically (that is $T_1 = O(T_2^2)$ and $T_2 = O(T_1^2)$). Since we have to live with differences like this

anyway, and since working out TM time-complexity is incredibly tedious, we actually use a more convenient measure as follows.

We use RAM programs in which the basic instructions are simple arithmetic operations, indexing (using a variable z to access a variable $y[z]$) and simple logical tests :-

```
x := 42
x := y + 42
x := y + z
x := y * z

x  :=  y[z]           (z a variable)
y[z] :=  x

if x = 0 then goto a
if x > y then goto a
```

RAM time-complexity is then the number of basic instructions executed. This measure is *not* equivalent to TM complexity, not even within some polynomial factor since numbers of the order of 2^{2^n} can be calculated on a RAM by squaring $O(n)$ times, whereas they need about 2^n cells just to be represented on a Turing Machine. The assumptions being made are that the values of the variables are bounded (the limit on an actual computer is often about 2^{31}) so that there is a bound on the cost of arithmetic, and that the number of variables is bounded so that the cost of accessing a variable $x[i]$ is bounded. RAM complexity is an acceptable measure provided that we check in each case what effect these assumptions have.

In order to justify this claim and compare RAM's with TM's in a direct way, we can define another complexity measure, logarithmic RAM complexity, as follows. We put all the variables of the RAM into a single sequence $x[i]$ and charge a cost $log(i)+log(x[i])$ for accessing a value $x[i]$. The cost for the addition $x+y$ is $log(x)+log(y)$, and for the multiplication $x*y$ is (say) $log(x)*log(y)$. This will lead to costs such as the following (where $log(x[i])$ refers to the *new* value of $x[i]$ etc.)

```
x[i] := x[j]          log(j) + 2log(x[j]) + log(i)
... x[j] + x[k]       log(j) + log(k) + 2log(x[j]) + 2log(x[k]) +...
x[x[i]] := ...        log(i) + 2log(x[i]) + log(x[x[i]]) +...
if x[i]=0 goto a      log(i) + log(x[i])
```

Theorem 2.1 *Turing Machine complexity and logarithmic RAM complexity are polynomially related measures.*

Proof. The polynomial bounds depend on the exact versions of the Turing Machine and RAM, but are small, say $T_1 = O(T_2^2)$ or $T_1 = O(T_2^4)$. First suppose we have a RAM program which takes k steps on a particular input (using logarithmic cost). We simulate it using a Turing Machine as follows. The TM's tape contains, to the right of the origin, a description of the current state of those RAM variables which are non-zero in the form

```
... # i # x[i] # j # x[j] # ...
```

where # is some special symbol marking the beginning and end of numbers and where each number i and value $x[i]$ are in some number base without leading zeros. Working space is reserved to the left of the origin. Each pair (except, say, 0 & $x[0]$ which represent the input) must have been created by a separate RAM instruction of the form $x[i] := \ldots$ which has logarithmic cost $log(i) + log(x[i]) + \ldots$, i.e. approximately the length of the representation `#i#x[i]#`. Thus the length of the representation of all the non-zero variables is $O(k)$. To access a variable $x[i]$, the TM has i in its workspace, looks for a pair `#i#x[i]#` and then moves $x[i]$ to its workspace. If it does not find such a pair, it knows $x[i] = 0$. This can be done within $O(k^2)$ steps on a Turing machine with one head, or $O(k)$ on a TM with two heads. (Two heads are better than one!) With one head, the TM has to shuttle between the representation of variables and the workspace comparing or copying one digit at a time, and making marks on the tape to show where it has got up to. Similarly, updating a value $x[i]$ involves finding and deleting any old pair `#i#x[i]#` and replacing it with a new one, which can also be done within $O(k^2)$ steps. Addition and multiplication are done in the workspace by "school" algorithms which do one digit at a time and keep track of carries. It should be clear that the Turing Machine can do this within some polynomial of the number of digits involved, and thus within some polynomial of k, say $O(k^3)$. The RAM simulates each instruction within $O(k^3)$, and it simulates $O(k)$ instructions, so the total time is $O(k^4)$.

Using a RAM to simulate a Turing Machine is easier. We merely have an array of variables $x[i]$ to represent the contents of the cells, (ignoring the problem of negative cell numbers), and a variable i representing the current position of the head. For each state s there is a block of RAM instructions labelled s. Suppose the transition table for the TM indicates that in state s, with current cell containing a, the TM changes the cell to b, moves right and goes into state t. Then the block of instructions for s might begin

```
s:   if x[i] /= a goto s1
     x[i] := b
     i := i + 1
     goto t
s1:  if x[i] /= b goto s2
     ...
```

As i is the only variable which is not bounded by a constant, and $i = O(k)$ where k is the number of steps taken by the TM, the cost of simulating this one step of the TM is $O(log(k))$ and simulating the whole computation is $O(k * log(k)) = O(k^2)$.

□

Chapter 3

Practical Problems

3.1 Algorithm Design – The Sorting Problem

Since in practice we approximate the time-complexity by using the (uniform cost) RAM complexity measure, we may as well approximate the size of the input to a problem as well. Thus in the problem of sorting n items taken from a total order into order (say numbers into numerical order or words into alphabetical order), we take the size of the problem instance to be n.

Remember, we are assuming that the items have some bounded size, and that they all fit into our internal storage space (i.e. the space available for variables). In the literature, this is sometimes called internal sorting to distinguish it from the case when we can't make these assumptions. Turing Machine or logarithmic RAM complexity is then more appropriate, but the exact version also matters.

Our first stab at an algorithm for the sorting problem assumes for simplicity that we are sorting numbers. Our approach is to find the largest, put it last, and then sort the remaining $n-1$.

```
     To sort items x[1] to x[n] into ascending order.
1.   for i=1 to n-1 do step 2
2.      if x[i] > x[i+1] then swap the values x[i] and x[i+1]
3.   sort the items x[1] to x[n-1]
```

It should be clear how to translate each step into primitive RAM instructions. The only question is how the sorting in line 3 is to be carried out. Clearly, the intention is that it should be carried out in the same way as the overall algorithm. In other words, we have specified the algorithm recursively. This can be very simply achieved in this case by the instructions

```
3a.  n := n - 1
3b.  if n >= 2 then goto step 1
```

The effect of recursion can always be achieved, though in harder cases one might need to keep a list of intermediate results and of partially executed subroutines.

Step 2 above can be done with some fixed number of instructions, i.e. in time $O(1)$. Steps 1 and 3 involve repeating these $n-1$ times, then $n-2$ times and so on, giving a total of $n*(n-1)/2$ times, so the total time taken is $O(n^2)$. We clearly cannot get a better bound than $O(n^2)$ using this algorithm.

If we drop our assumptions and use logarithmic RAM complexity, the input size is $O(n * log(n))$ (assuming we are sorting numbers between 1 and n). Step 2 takes $O(log(n))$ time and is repeated $O(n^2)$ times, giving a total time of $O(n^2 * log(n))$, regarded as a function of $m = n * log(n)$. This is a nasty function of m, but it is clearly $O(m^2)$, so our assumptions were justified. In many problems, the values of the variables are $O(n)$ and the uniform and logarithmic complexities differ by no more than a factor $(log(n))^k$ for some small constant k.

To get a better bound than n^2 on the complexity of sorting, we use the "divide and conquer" approach – we split the problem in two and solve the halves. We ignore the problems of odd n for the moment.

```
     To sort items x[1] to x[n]
1.   sort items x[1] to x[n/2]
2.   sort items x[n/2+1] to x[n]
3.   merge the two sorted lists
```

Again we have a recursively specified algorithm. The recursion can be handled by keeping a list of pairs (i, j), each representing a sequence of items $x[i]$ to $x[j]$ which need sorting. Assuming this can be done with only a constant factor increase in the time taken for each step, (you can always assume this), we can analyse the complexity as follows. Step 3 can be done by running through the two lists simultaneously, copying the lesser of the two current items to a second set of variables $y[i]$, and then copying the resulting joint list back into the $x[i]$. This takes $O(n)$ time, say $k*n$. If T_n is the time taken by this algorithm to sort n items, we get a recurrence equation

$$T_n = 2 * T_{n/2} + k * n$$

We can absorb any overhead involved in subroutine calling in steps 1 and 2 in the term $k * n$. From this, it is easy to show by induction that $T_n = O(n * log(n))$, an improvement on our $O(n^2)$ bound above. The problem of odd n is most easily dealt with by pushing n up to the next power of 2 by adding a few "infinite" items on the end. We have no more than doubled n, so if the time taken for arbitrary n is S_n, then $S_n = O(T_{2*n})$ which is still $O(n * log(n))$.

There are many $O(n * log(n))$ sorting algorithms known, and the question arises whether this is the best possible. Lower bounds on complexity are notoriously difficult to obtain, but sorting is one problem where we have a clear-cut answer. We have to assume that nothing is known about the total order from which the items are taken, and that all the information which the algorithm gathers comes from direct comparisons between pairs of items, together with inferences from the total order axioms. A lower bound on the complexity of sorting can then be obtained by finding the minimum number of comparisons which can be made before the order of the items is determined. There are $n!$ possibilities to be distinguished, and any one comparison can at best halve the number of remaining possibilities. (Remember that we are interested in worst-case complexity.) Thus we need at least $log(n!)$ comparisons. Since $n! > (n/2)^{n/2}$, $log(n!) > k * n * log(n)$ for some k.

3.2 Data Structure Design – Set Operations

Many algorithms require the handling of sets during their operation. As an example of this, consider the minimum spanning tree problem. We are given a graph, each of whose edges has

a cost associated with it. The graph might represent towns, the edges being possible routes for building railway lines between pairs of towns. The edge cost would be the cost of building such a rail link. The minimum spanning tree would represent the railway network of least cost linking all the towns.

One algorithm for finding the minimum spanning tree begins by sorting the edges into order of increasing cost. The first (lowest cost) edge is put into the tree. At each stage, the next edge is put into the tree if it does not create a cycle, otherwise it is discarded. There is a simple graph-theory proof (exercise) that this does indeed yield the minimum spanning tree. The algorithm can be implemented by having a collection of sets $T[i]$ which partition the vertices such that two vertices are in the same set iff they are currently joined by tree edges. The next edge can then be tested to see if its ends are in the same set. If so, it is discarded, otherwise they are in sets $T[i]$ and $T[j]$, say. The edge is added to the tree, and the the sets $T[i]$ and $T[j]$ are replaced by their union.

We need to design a way of representing these sets (i.e. a data structure for them) so that we can cope with a sequence of set operations in the most efficient way. The set operations which we need for the above problem are:

```
FIND        Given an item i, find the set containing it.
UNION       Form the (disjoint) union of two given sets.
```

We assume that the items are the numbers $1, \ldots, n$ and that the sets partition the items. The names of the sets are of no importance for this problem. In other problems where the names are important, translations between internal names and external names can be provided separately. Thus we will name the sets $T[i]$ with numbers $i = 1, \ldots, n$ in such a way that if $T[i]$ is non-empty, i is its "first" element. Initially the sets consist of n singletons. We assume that there will be $n-1$ UNION operations and $O(n)$ FIND operations in the course of the algorithm.

The best data structure for the FIND operations is undoubtedly an array of variables $s[i]$ such that $s[i]$ represents the set containing item i. Thus for $n = 10$, the array

```
i    = 1  2  3  4  5  6  7  8  9 10
s[i] = 1  2  1  2  5  2  2  1  2  5
```

represents the non-empty sets

```
T[1] = {1, 3, 8}
T[2] = {2, 4, 6, 7, 9}
T[5] = {5, 10}
```

The cost of finding the set containing i is just the cost of accessing the variable $s[i]$ which is $O(1)$. On the other hand, the cost of forming a union is high. Forming the union of $T[1]$ and $T[5]$ above, for example, involves scanning all the variables $s[i]$, and changing all those with value 5 to value 1. The cost of this is $O(n)$, so the cost of $O(n)$ operations is $O(n^2)$.

The best data structure for the UNION operations is one in which the items in any one set are in the form of a list, so that two lists may merely be concatenated. We need not insist that the items in the list be in any particular order. For example, we could keep track of the first and last elements in each set with variables $f[i]$ and $l[i]$, and for each item record the next item in the same set in variables $n[i]$. Undefined values can be represented by zero. The above sets would be represented by:

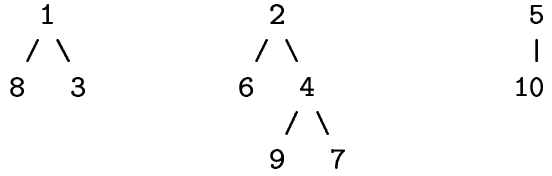
```
i    = 1  2  3  4  5  6  7  8  9 10
f[i] = 1  2  0  0  5  0  0  0  0  0
l[i] = 8  9  0  0 10  0  0  0  0  0
n[i] = 3  4  8  6 10  7  9  0  0  0
```

The union of sets $T[i]$ and $T[j]$ can be carried out by the algorithm

```
    To form the UNION of non-empty sets T[i] and T[j]
1.  n[l[i]] := f[j]
2.  l[i] := l[j]
3.  f[j] := 0
4.  l[j] := 0
```

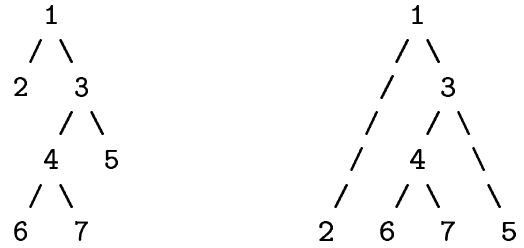
It is easy enough (exercise) to add checks to cope with sets which may be empty. Thus a UNION operation can be carried out in a time which is $O(1)$. However, a FIND operation involves searching all the lists, or following the list the item is in to the end and searching for the last element among the $l[i]$. This is an $O(n)$ operation and so once more the cost of $O(n)$ operations is $O(n^2)$.

To do better than this, we need some compromise data structure in which neither UNION nor FIND is too expensive. The answer is to use a tree to represent each set. The sets in our example above might be represented as trees with roots 1, 2 and 5 thus.



Terminology

When trees are used as data structures, they are traditionally drawn upside-down with the root at the top, and the following terminology is used. The neighbours of a vertex which are further from the root (lower) are called its children. The neighbour which is nearer to the root (higher) is called the parent. The depth of a vertex is the distance from the root (the root having depth 0). The height of a vertex is the maximum distance from any of its descendents. The height of a tree is the height of its root. The weight of a tree is the number of vertices in it. The following picture illustrates a tree drawn with the levels representing depths, and with the levels representing heights.



In the above representation, UNION's are implemented by making one tree a subtree of the other, and to keep the trees well balanced (so the depth of an item does not grow too fast) we should make the smaller tree a subtree of the larger. FIND's can be implemented by following the path from the item to the root. Thus we only need to store the parent of each item in variables $p[i]$, and the weight of each tree in $w[i]$.

```
i    = 1  2  3  4  5  6  7  8  9 10
p[i] = 0  0  1  2  0  2  4  1  4  5
w[i] = 3  5  0  0  2  0  0  0  0  0
```

Note that the non-zero values of these sets of variables match nicely so that in a practical algorithm, they could be merged (say making the weights negative to distinguish them). It is easy to see inductively that because we make the smaller tree a subtree of the larger, the height of the trees is $O(log(n))$, so the time taken for $O(n)$ operations is $O(n * log(n))$. However, we can do better still by *path compression*. Whenever we FIND an item, and follow the path to the root, we can make the items on the path children of the root to speed up later FIND's. Thus the algorithms for UNION and FIND are as follows.

```
    To form the UNION of non-empty sets T[i] and T[j]
1.  If w[i] < w[j] then exchange i and j
2.  w[i] := w[i] + w[j]
3.  w[j] := 0
4.  p[j] := i

    To FIND the set containing an item i
    Use variables j, l[j] to keep a list of items on the path to the root
1.  j := 0
2.  while p[i] /= 0 do steps 3,4 and 5
3.     j := j + 1
4.     l[j] := i
5.     i := p[i]
6.  for k = 1,...,j do p[l[k]] := i
7.  the item is in set T[i]
```

Definition 3.1 *The fast growing function $F(n)$ and the complementary slow growing function $G(n)$ are defined as follows. Note that $G(n) \leq 5$ for "practical" values of n, i.e. for $n \leq 2^{65536}$.*

$$\begin{aligned} F(0) &= 1 \\ F(n+1) &= 2^{F(n)} \\ G(n) &= \text{minimum } k \text{ with } F(k) \geq n \end{aligned}$$

Theorem 3.1 *The above data structure and algorithms allow $O(n)$ UNION and FIND operations on n items to be carried out in a time which is $O(n * G(n))$, i.e. almost linear.*

Proof. For the purposes of the proof, we pretend that the algorithm keeps track of the *parent* of each item, and also the (highest known) *ancestor* of each item. The FIND operations are assumed to use and alter only the ancestor variables and not the parent variables, so that the

tree structure at any stage is determined entirely by the UNION operations. Throughout the proof a tree T_i will have root r_i, height h_i and weight w_i.

The first step in the proof is to show by induction that at any stage if an item has height h, then it has at least 2^h descendents. Suppose at some stage we form the UNION of trees T_1, T_2 where $w_1 \geq w_2$. Then r_1 becomes the parent of r_2, and r_1 is the only item whose height or number of descendents have changed. The height of r_1 becomes $max(h_1, h_2 + 1)$ and the number of descendents becomes $w_1 + w_2$. By induction,

$$w_1 \geq 2^{h_1} \text{ and } w_2 \geq 2^{h_2}$$

so

$$w_1 + w_2 \geq w_1 \geq 2^{h_1}$$

$$w_1 + w_2 \geq 2 * w_2 \geq 2 * 2^{h_2} \geq 2^{h_2+1}$$

Since there are n items, the above result shows that the height of any item never exceeds $log_2(n)$. Moreover, there are at most $n/2^h$ items of height h, since any two items at the same height have distinct sets of descendents. Now we partition the items into groups at each stage according to their height by defining $g(i) = G(h(i))$ where $h(i)$ is the height of item i. In other words $g(i) = j$ where $F(j-1) \leq h(i) \leq F(j)$. Thus items of height 0 or 1 go into group 0, items of height 2 into group 1, items of height 3 or 4 into group 2, items of height 5,...,16 into group 3 and so on. As the height of an item never exceeds $log_2(n)$, the group number is at most $G(log_2(n)) \leq G(n) - 1$, so there are at most $G(n)$ groups.

A UNION operation takes $O(1)$ time, so $O(n)$ UNION operations take $O(n)$ time. Any constant overhead in a FIND operation similarly contributes $O(n)$ time to the total. A FIND operation may thus be regarded as taking a constant amount of time for each item whose ancestor variable is altered. First consider those items whose ancestor belongs to a higher group when the alteration is made. As the ancestor variables are followed during a FIND operation, the height and thus the group of the items increases monotonically. As there are at most $G(n)$ groups, the number of items encountered whose ancestor is in a higher group is at most $G(n)$. These items thus contribute $O(G(n))$ to the time taken by a FIND operation, and $O(n * G(n))$ to the total time taken.

It remains to consider those items whose ancestor is in the same group before the ancestor is altered. Note that the height of the ancestor is strictly increased when it is altered, so the ancestor of a particular item in group j is altered at most $F(j) - F(j-1) \leq F(j)$ times by FIND operations before being in a higher group. Now as the number of items at height h is at most $n/2^h$, the number of items in group j is bounded by

$$\begin{aligned} & n/2^{F(j-1)+1} + n/2^{F(j-1)+2} + \ldots + n/2^{F(j)} \\ \leq\ & n/2^{F(j-1)+1}(1 + 1/2 + 1/4 + 1/8 + \ldots) \\ \leq\ & n/2^{F(j-1)} \\ \leq\ & n/F(j) \end{aligned}$$

Thus the total time taken on items in group j is $O(n)$. This applies to all the groups, and so the total time for these items is $O(n * G(n))$. Since we have split the time taken into pieces each of which is $O(n * G(n))$, the total time taken is $O(n * G(n))$ as required.

□

Chapter 4

NP-Completeness

4.1 The Travelling Salesman Problem

In this chapter, we are looking for a means of distinguishing between those problems with practical algorithms and those without. The subject was stimulated by a number of problems which had no known practical algorithm, the most famous of which is probably the Travelling Salesman Problem (TSP).

The problem involves a number of towns. The distance between each pair of towns is known. The distance may be infinite if no direct route between the two towns is available. The Travelling Salesman's job is to start out from his home town, visit all the other towns exactly once each, then return home. His problem is to find the shortest route. If he is more interested in economy, he can replace the inter-town distances by inter-town costs.

All the known algorithms for solving this problem exactly (i.e. for finding the minimum route rather than just finding a reasonable route) have exponential complexity. The most obvious algorithm involves searching through all $n!$ permutations of the n towms. An immense amount of effort has been spent on this and similar problems in an attempt to find practical algorithms. It would clearly save a lot of further effort if it could be shown that this problem has no efficient algorithm for its exact solution. This has not been done, but the problem has been shown to be NP-complete, which for practical purposes may be taken to mean that it has no efficient algorithm.

The idea behind NP-completeness is to define a wide class NP of problems. The class includes most combinatorial problems which arise in practice, and in particular includes the Travelling Salesman Problem and many other long-standing unsolved problems. It is then shown that the TSP and the other unsolved problems are as hard as possible in the class NP, that is NP-Complete, in the sense that any problem in the class NP can be solved efficiently from them. If any one of these NP-Complete problems could be solved in polynomial time, then all problems in the class NP could be solved in polynomial time. This is now regarded as extremely unlikely, so "NP-Complete" may be taken to imply "not having a polynomial algorithm".

4.2 The Class P of Polynomial Problems

It is instructive to contrast problems involving "exponential search", i.e. involving searching through a number of items which grows exponentially with the size of a problem instance,

with those having polynomial algorithms, that is algorithms which take an amount of time bounded by some polynomial in the size of the problem instance.

If we compare the time taken by algorithms of complexity n^5 and 2^n microseconds for instances of size $n = 10, 20, 30, 40, 50, 60$, we get

n	$n^5\mu$		$2^n\mu$	
10	0.1	*sec*	.001	*sec*
20	3.2	*sec*	1	*sec*
30	24.3	*sec*	17.9	*min*
40	1.7	*min*	12.7	*days*
50	5.2	*min*	35.7	*years*
60	13.0	*min*	366	*centuries*

Exponential algorithms thus quickly become impractical even for modest sizes of problem instance. Moreover, if we increase the power of our computer, either with better hardware or better software, by a factor of 100, the size of problem we can cope with goes up by a factor of 2.5 for the n^5 algorithm, whereas for the 2^n algorithm the size merely increases by adding 6 or 7. It is clear that for practical purposes, the n^5 algorithm is acceptable, at least as a basis on which to improve, but the 2^n algorithm is not. In practice, few algorithms fall between $O(n^5)$ and $O(2^n)$. Of course, there are functions such as $2^{n/log(n)}$ which are more than polynomial but less than exponential. It would seem natural to regard algorithms with such complexities as unacceptable. We thus make the definitions:

Definition 4.1 *A problem is called a decision problem (or "language" or "predicate") if it has a yes or no answer, i.e. if the corresponding number-theoretic function takes values 0 or 1. The rest of this section is restricted to decision problems for technical simplicity.*

Definition 4.2 *A decision problem is said to be polynomial if it has an algorithm of time-complexity bounded by some polynomial in the size of a problem instance, i.e. of complexity $O(n^k)$ for some constant k. The class P is defined to be the class of polynomial decision problems.*

Note that most problems can easily be seen to be at least as hard as some corresponding decision problem, so that to prove the problem non-polynomial, it suffices to show that the corresponding decision problem is not in P. For example the TSP "find the minimum tour in the weighted graph G" has a corresponding decision problem "does the weighted graph G have a tour of weight at most k".

Note also that the definition does not depend too crucially on the time-complexity measure chosen. Any reasonable version of Turing Machine complexity or logarithmic RAM complexity will give rise to the same class of problems. Indeed one might conjecture (along the lines of Church's thesis) that any "reasonable" time-complexity measures are polynomially related and so define the same class P.

In order to tell whether a problem is practical or not, it suffices either to find a polynomial algorithm or to show that the problem is not in class P. However, the only problems known not to be in class P are either artificially constructed, or else are much worse than exponential.

4.3 The Class NP (Non-deterministic Polynomial)

To make some headway with the problem of showing that some problems are intractable, we define a class NP of decision problems which can be solved in a time $O(2^{n^k})$ for some k. This can be regarded as the class of problems solvable with one level of exponential search.

Informally, we can define this class NP in terms of a two-stage "non-deterministic" algorithm. The first stage takes the input i, and from it produces some structure g by guessing. The second stage conventionally processes the inputs i and g. The solution to the original problem is taken to be *yes* iff there is a guess g for which the second stage halts with answer *yes*. The algorithm is said to be polynomial, or a member of NP, if the second stage is a deterministic (conventional) algorithm with a complexity which is polynomial in the length of input i. This implies a polynomial bound on the length of the guessed structure g.

For instance, in the TSP "does weighted graph G have tour of length at most k", the guessing stage can produce a permutation of the vertices of G. The second stage can, in polynomial time, check whether the permutation defines a tour of length at most k. This shows that the TSP is in the class NP.

Note that the definition of NP is not symmetrical in *yes* and *no* in the way in which the class P is. NP can be regarded as a class of existence problems ("Is there a ...such that ..."). In other words, whenever the answer to an NP problem is *yes*, there is a polynomially checkable certificate of the fact. The complement of the TSP, namely "does the weighted graph G have *no* tour of weight at most k", is not obviously a problem in NP, having no obvious compact certificate of a *yes* answer, and indeed it is not known to be in NP.

4.4 Other Definitions Of The Class NP

The above definition of NP can easily be formalised in terms of a Turing Machine which takes input i, follows it by an arbitrary number g, and then acts conventionally on the pair (i, g).

There are two variations on the definition of non-deterministic algorithm which apparently yield greater power. First, the second part of the algorithm is often only required to stop in the case of a *yes* answer. In other words, the overall problem is said to be in NP if whenever the overall problem has answer *yes*, there is a guess for which the second part of the algorithm delivers the answer *yes* within polynomial time. Suppose we have such an algorithm which works within a time n^k. We can arrange another algorithm for the problem as follows. We add a mechanism which counts the number of steps taken in the first algorithm and stops after n^k steps, returning the answer *no*, if the first algorithm has not yet stopped. If we are dealing with Turing Machines, the second algorithm can simulate one step of the first, including step-counting, in $O(n^k)$ steps, and thus can simulate the whole algorithm in $O(n^k * n^k) = O(n^{2k})$ steps. Thus the new algorithm is fully polynomial, and gives the same overall answers to the problem instances. The new algorithm can only be found effectively if a particular polynomial bound is known for the old one, but it always exists.

The second (in fact the older) variation of the definition of non-deterministic algorithm is as follows. At each step of the algorithm, instead of there being just one possible next step, there is a finite set of possible next steps. The algorithm is deemed to give the answer *yes* if any one of the possible sequences of steps which the algorithm could take yields the answer *yes*. Thus the "guessing" or "non-determinism" is spread through the computation rather than being concentrated at the beginning.

A useful mental picture one can use in this situation is as follows. Imagine an unlimited supply of machines, initially dormant. One machine is given the problem to solve. It makes the first step of the computation and discovers that there are several possible next steps, all of which must be explored. It chooses several dormant machines to which it passes the problem, telling each which one of the possible steps to concentrate on, then waits for their replies. The "child" machines take one step each, and then pass out sub-problems to more dormant machines, and so on. Each machine passes back the answer *yes* to its parent if it receives a *yes* back from any of its children.

Given a polynomial bound n^k on the number of steps taken by such an algorithm, we can simulate the action of the algorithm by first guessing the sequence of n^k choices which will be made at each step, then simulating the deterministic algorithm which those choices define. If the algorithm does not stop within n^k steps, give the answer *no*.

Thus *for the purposes of defining NP* these variations on the non-deterministic theme are equivalent.

4.5 Building A Non-deterministic Machine

Because of the picture given above of machines passing each other sub-problems, many people seem to think that a parallel computer with an effectively unlimited number of processors would be able to "crack" the NP-complete problems. In particular the human brain can be regarded as parallel computer with (very roughly) 10^{10} processors. Unfortunately, this is not true.

To see why, define a parallel computer as follows. There is an infinite number of identical processors (or a finite number of different types, which amounts to the same thing). Each processor is an "automaton" or "finite machine", that is a Turing Machine with a finite tape. The processors are arranged in 3- (or finite-) dimensional space. Each takes up a fixed minimum volume, and they cannot overlap. Each can only communicate with those others within a fixed distance (or else one can take the finite speed of communication into account). These restrictions make it clear that after a polynomially bounded time, only a polynomially bounded number of processors will be taking part in the computation, whereas our definition of a non-deterministic algorithm above requires exponentially many processors to be involved. Our "practical" parallel computer in fact defines the same class P as a humble Turing Machine! See a recent issue of Theoretical Computer Science for more details.

4.6 Polynomial Transformations And NP-Completeness

The classes P and NP are now on a sound footing, and we can get down to the business of NP-completeness. The basic tool of NP-completeness is the polynomial transformation.

Definition 4.3 *A problem X is polynomially transformable to a problem Y if there is a polynomial algorithm p which transforms an input i to problem X into an input $p(i)$ to problem Y in such a way that input $p(i)$ is accepted by Y (has answer yes) if and only if input i is accepted by X.*

Definition 4.4 *Two problems are polynomially equivalent if each can be polynomially transformed into the other.*

Definition 4.5 *A problem X is NP-complete if it is in the class NP and all other problems in the class NP polynomially transform to it.*

If X is polynomially transformable to Y, we can say that X is no harder than Y (up to a polynomial), and indeed if Y has a polynomial algorithm for its solution, then so does X. The converse of this is what is used in NP-completeness proofs, so we grace it with the title of theorem.

Theorem 4.1 *If problem X polynomially transforms to problem Y in NP, and if X is NP-complete, then Y is also NP-complete.*

Proof. If we take any problem in the class NP, it is polynomially transformable to X. As X is polynomially transformable to Y, we can combine the two transformations to get a polynomial transformation from the problem to Y. Y is thus NP-complete.
□

This allows us to prove an unknown problem NP-complete by transforming a known NP-complete problem to it. Note which way round this goes – we do not transform the unknown problem to something else as we would do if we were trying to show that it was easy.

There is a (historically older) notion of polynomial reduction which can also be used to define NP-completeness. Instead of transforming an input to problem X into an input of problem Y to get a compound algorithm for X, X is said to reduce to Y if there is a polynomial algorithm for X which uses the problem Y as a subroutine possibly many times (each use counting as one step). This apparently more powerful definition might lead to a larger class of NP-complete problems, though this is not known. All currently known NP-complete problems can be shown NP-complete using transformations, and as this is usual practice, we use transformations as the basis for our definition.

4.7 Cook's Theorem – Satisfiability is NP-Complete

It is now clear that once we have one NP-complete problem, we can transform it to others and prove many problems NP-complete. However, at least one problem has to be shown NP-complete the hard way – by showing that all problems in NP transform to it. The honour of being the "first" NP-complete problem goes to a problem in propositional logic – the satisfiability problem.

Definition 4.6 *The satisfiability problem is as follows. Given an expression or formula in propositional logic formed from variables $x, y, z, \ldots$, connectives $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\Rightarrow$ (implies) and $\Leftrightarrow$ (if and only if), and brackets, is there an assignment of the values true and false to the variables which makes the resulting expression have value true. If so, the expression is called satisfiable.*

The size of an instance of the problem is the number of symbols in the expression. The problem is slightly more general than Cook's original one, as he insisted that the expression be in a particular normal form. Because of the symmetry between *true* and *false* in propositional expressions, the satisfiability problem is (computationally) equivalent to the problem of telling whether an expression is *not* a tautology. Telling whether an expression is a tautology is not (obviously) in NP as it (apparently) has no brief certificate of the fact.

Theorem 4.2 (Cook's Theorem.) *The satisfiability problem is NP-complete.*

Proof. The satisfiability problem is clearly in NP since we can guess an assignment of *true* and *false* to the variables, then calculate the value of the resulting expression in polynomial time.

Let X be an arbitrary problem in the class NP. We must show how to transform it to the satisfiability problem. Suppose that X is presented as a Turing Machine which takes an input i, makes a guess g, then acts in the normal way on the pair (i, g). Suppose that it takes at most $N = n^k$ steps on an input i of length n. We may also suppose that the tape cells only contain symbols T or F and that the Turing Machine never moves to the left of the input. (The number of tape cells can always be reduced to two at the expense of the number of states, and the input can be moved a suitable distance to the right before starting). Suppose that the TM has K states.

The instantaneous state of the TM at any time can be described by the contents of the first N tape cells, the current state, and the currently scanned cell. The computation can thus be described by the values of the following propositional variables.

$S_{t,s}$	$1 \leq t \leq N, \quad 1 \leq s \leq K$	(State)
$P_{t,c}$	$1 \leq t \leq N, \quad 1 \leq c \leq N$	(Position)
$C_{t,c}$	$1 \leq t \leq N, \quad 1 \leq c \leq N$	(Contents)

$S_{t,s}$	means the TM is in state s at time t
$P_{t,c}$	means the TM is scanning tape cell c at time t
$C_{t,c}$	means the tape cell c holds T at time t

The input i determines the initial values $C_{1,c} = i_c, 1 \leq c \leq n$, of the first n tape cells, and the guessed number g affects the initial values of the remaining cells $C_{1,c}, n < c \leq N$. The state at time t is thus described by $2 * N + K$ variables, and the whole computation by $N * (2 * N + K) = O(n^{2k})$ variables.

We can now construct a propositional expression from these variables which will mean "the variables represent the computation of the TM for X on input i for some guess g, and the computation stopped with answer *yes* within N steps". The expression can be regarded as having independent variables $C_{1,c}$ for $n < c \leq N$, i.e. the guess g, all other variables being determined from these. The expression is thus satisfiable if and only if there is a guess g which leads to an answer *yes*, i.e. iff the input i to problem X has answer *yes*. The expression is constructed from the following clauses by putting brackets round each and joining them all together with $\wedge$.

$S_{1,1} \wedge P_{1,1}$	clauses ensuring: initial state correct,
$C_{1,c} \Leftrightarrow i_c$	computation has input i $(1 \leq c \leq n)$,
$S_{t,1} + \ldots + S_{t,K}$	TM in at least one state $(1 \leq t \leq N)$,
$\neg(S_{t,s1} \wedge S_{t,s2})$	and at most one state $(1 \leq t \leq N, 1 \leq s1 \neq s2 \leq K)$,
$P_{t,1} + \ldots + P_{t,N}$	TM scanning at least one cell $(1 \leq t \leq N)$,
$\neg(P_{t,c1} \wedge P_{t,c2})$	and at most one cell $(1 \leq t \leq N, 1 \leq c1 \neq c2 \leq N)$,
$S_{N,K} \wedge C_{N,1}$	TM stops (state K) by time N with *yes* in cell 1,
$(S_{t,s1} \wedge P_{t,c} \wedge C_{t,c}) \Rightarrow$ $(S_{t+1,s2} \wedge P_{t+1,c+1} \wedge \neg C_{t,c})$	TM transitions, e.g. in state s_1 scanning T, write F, move R, enter state s_2 $(1 \leq t < N, 1 \leq c < N)$

It is easy to check that the resulting expression is made up from a number of variables polynomial in n, that each clause has polynomial length, that there are a polynomial number of clauses, and that the construction of the whole expression from i takes a time polynomial in the length n of i. The existence of a polynomial algorithm has thus been shown, though effectively producing it requires knowledge of a particular non-deterministic Turing Machine algorithm for X together with a bound n^k on its running time.

It is clear that the resulting expression is an input to the satisfiability problem, and that it is satisfiable if and only if input i to problem X has answer *yes*. So any problem X in the class NP is polynomially transformable to the satisfiability problem, which is thus NP-complete. □

4.8 NP-completeness of the Travelling Salesman Problem

Now that we have one NP-complete problem under our belt, we can quite quickly transform it into others. The first step is to transform a general propositional expression into a more suitable normal form.

Definition 4.7 *The 3-satisfiability problem is the problem of telling whether a propositional expression is satisfiable, the expression being of a form such as*

$$(a \vee \neg b \vee c) \wedge (\neg d \vee \neg a \vee e) \wedge \ldots$$

i.e. a conjunction of clauses, each of which is a disjunction of three terms, each of which is a simple variable or its negation.

Theorem 4.3 *The 3-satisfiability problem is NP-complete.*

Proof. The problem is clearly in NP, being a restriction of satisfiability. Methods for transforming expressions to various normal forms are well-known by logicians, computer scientists,

and hardware designers. The only subtlety is that exponential growth must be avoided to make the transformation polynomial. For example in eliminating $\Leftrightarrow$, we cannot merely replace

$$e_1 \Leftrightarrow e_2$$

by

$$(e_1 \Rightarrow e_2) \wedge (e_2 \Rightarrow e_1)$$

and then expand the four sub-expressions, since we have doubled the length of the original expression, and we may double it again in expanding the four subexpressions and so on. Instead, new variables must be introduced to represent each subexpression, and the expression must broken down into short clauses, the first representing the expression as a whole, and the others defining the added variables. For example, $(a \wedge b) \Leftrightarrow (c \Rightarrow d)$ might become

$$\begin{array}{l}(e \Leftrightarrow f) \wedge \\ (e \Leftrightarrow (a \wedge b)) \wedge \\ (f \Leftrightarrow (c \Rightarrow d))\end{array}$$

The clauses are bounded in length, (at most 7 symbols), and can be put into normal form without expanding the length more than polynomially.
□

Now we have to move from propositional logic to graph theory. The first graph theory problem we prove NP-complete is the vertex cover problem.

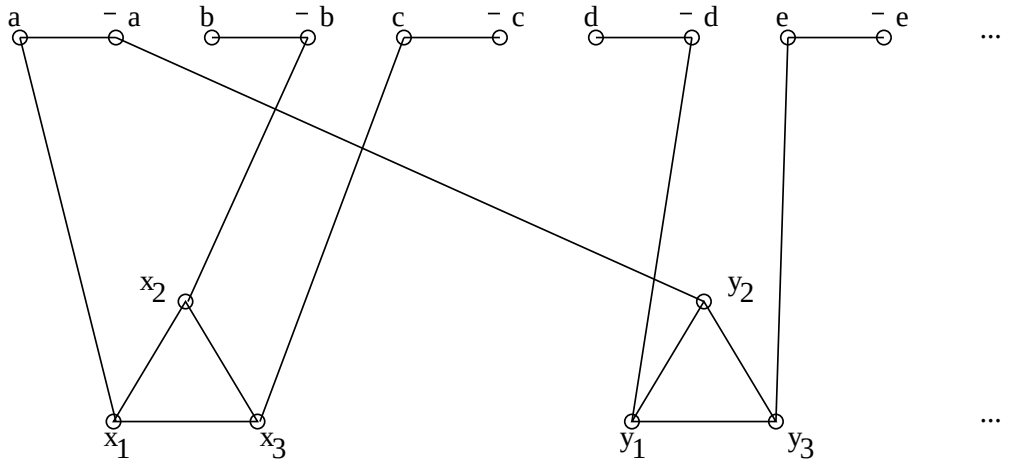
Definition 4.8 *The vertex cover problem is as follows. Given a graph G and a number k, is there a set V of vertices of G of size at most k which covers all the edges, that is such that each edge has at least one end in V.*

Theorem 4.4 *The vertex cover problem is NP-complete.*

Proof. The problem is in NP as the set V can be guessed, and then checked within polynomial time. The 3-satisfiability problem can be tranformed into the vertex cover problem as follows. Suppose an instance of the 3-satisfiability problem has the form

$$(a \vee \neg b \vee c) \wedge (\neg d \vee \neg a \vee e) \wedge \ldots$$

From it, construct a graph G having two vertices for each variable, labelled $a, \neg a, b, \neg b, \ldots$, and three vertices for each clause x labelled x_1, x_2, x_3. The graph has *truth-setting* edges connecting a and $\neg a$, b and $\neg b$ etc., so called because any vertex cover will have to include at least one of $a, \neg a$ and at least one of $b, \neg b$ etc. It also has three *satisfaction-testing* edges for each clause x, joining x_1, x_2 and x_3. Any vertex cover must contain at least two of these three vertices. Finally, there are interconnecting edges connecting vertices x_i to the three variables (or negated variables) in clause x. Thus the above example instance of 3-satisfiability yields a graph G pictured below.



The instance of vertex cover consists of the graph G, together with the number $k = v+2*c$ where v is the number of variables, and c the number of clauses. It is easy to see that the construction can be carried out in polynomial time. It remains to show that the graph G has a vertex cover of size at most k if and only if the expression from which it is transformed is satisfiable.

First suppose that V is a vertex cover of G of size at most k. Then V must contain one vertex from each pair $a, \neg a$, and two from each triple x_1, x_2, x_3. This makes $v + 2 * c = k$ vertices, so there can be no more. The set V must also cover the connecting edges. Define an assignment of *true* and *false* to the variables a, b, c by setting variable a *true* if V contains vertex a, *false* otherwise etc. A clause represented by x_i has at least one vertex, say x_1, not in V, but V covers the connecting edge between x_1 and a (say), and so V must contain a, a has value *true*, and the clause has value *true*. This holds for each clause, so the expression is satisfiable.

Conversely, suppose that the expression is satisfiable with a particular assignment of *true* and *false* to each variable. Define a set V of vertices of G as follows. Put a variable a into the set if a has value *true*, otherwise put $\neg a$ into V. Each clause x has value *true*, so at least one of its three terms x_i has value *true*, say x_1. Put x_2 and x_3 into the set V. The set V has k members, and is a vertex cover for G as required.
□

Next, we tackle the Hamiltonian circuit problem.

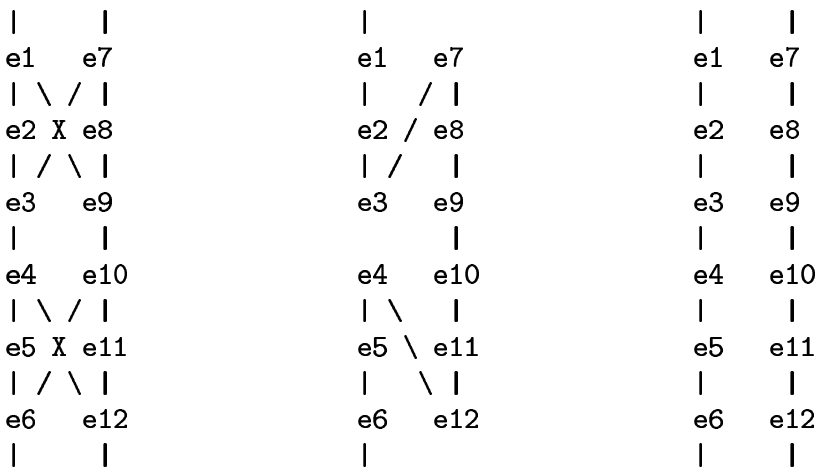
Definition 4.9 *The Hamiltonian circuit problem is as follows. Given a graph G, is there a tour which visits each vertex exactly once and returns to the starting vertex. This can be regarded as a Travelling Salesman Problem in which each distance is 1 or infinite, and a finite length tour is required.*

Theorem 4.5 *The Hamiltonian circuit problem is NP-complete.*

Proof. The problem is in NP since a permutation of the vertices can be guessed, and then checked for being a tour in polynomial time. The vertex cover problem can be transformed into the Hamiltonian circuit problem as follows. Start with a graph G and number k representing an instance of the vertex cover problem. From it, we create a graph H, an instance of the Hamiltonian circuit problem.

The vertices of H are as follows. There are k vertices $s_1, \ldots, s_k$ which are intended to select k vertices from G. Then for each edge e of G joining vertex a to vertex b there are 12 vertices e_1 to e_{12}, with e_1 to e_6 being associated with a, and e_7 to e_{12} with b. (Some arbitrary order (a, b) is decided on for the vertices of e).

The edges among the e_i are as shown below. In addition, e_1, e_6, e_7, e_{12} may be connected to other vertices. Also shown, are the only two possible ways, bar symmetry, in which the edges among the e_i could be included in a tour of H



Suppose that a vertex a is incident with edges $e, f, \ldots, g$ (in any order). Suppose that e_1 to e_6, and f_7 to f_{12},..., and g_1 to g_6 are associated with a (for example). Then add edges from each of the s_i to e_1, from e_6 to f_7, from f_{12} to ... to g_1 and from g_6 to each of the s_i. This completes the graph H, and the transformation from G can be done in polynomial time.

Suppose that G has a vertex cover $V = \{a, b, c, \ldots\}$ of size k. Then H has a tour which starts at s_1, follows the vertices in each edge set e_i associated with a, returns to s_2, follows the vertices in each edge set e_i associated with b, returns to s_3, and so on, finally returning to the starting vertex s_1. Since V is a vertex cover, each edge set e_i will be entered at least once corresponding to one of its end vertices. If it entered only once, a path such as the middle one above is taken through the set, if twice, a path such as the right hand one above is taken. In this way, the tour described passes exactly once through each vertex of H.

On the other hand, a tour of H has no choice but to follow all the vertices in each edge set associated with a particular vertex a of G between visits to the vertices s_i, and so defines a set of k vertices of G which can be seen to be a vertex cover.

□

At last we reach the Travelling Salesman Problem again.

Theorem 4.6 *The Travelling Salesman Problem is NP-complete.*

Proof. The problem is in NP. The Hamiltonian circuit problem transforms to the TSP by defining distances between the vertices to be 1 if they are joined by an edge, and infinite (or 2) if they are not. The TSP is then to find a tour of length at most n, where n is the number of vertices.

□