NP or not NP

Dylan Zwick

Tuesday, January 29th 2008

In these notes and in the talk given from these notes I discuss the basic concept of algorithmic complexity, and present some examples of different types of algorithms for solving the same problem, and their different levels of complexity. I then introduce some famous problems, and discuss some aspects of their complexity classes. This leads to the concept of the complexity classes of P and NP. I then discuss the set of NP-complete problems, and give an example. I then close with the open question of whether $P \neq NP$.

1 Algorithmic Complexity

1.1 Sorting

Suppose you have a set of numbers like:

$$(3, 5, 7, 2, 8, 6, 1, 4)$$

and you want to sort them. A simple algorithm for doing this would be to just read through the sequence in order, reading two at a time and moving one forward each time, and switching the two you're examining if they're out of order. So, for example, here we'd read $(3, 5)$, find nothing wrong and so then move onto $(5, 7)$, again find nothing wrong and move onto $(7, 2)$ which we'd then switch to $(2, 7)$. Once the algorithm had made

one pass through the list it would make another pass, and continue to do this until it had to make no switches, and therefore the list was sorted. So, for example, after one pass our list would look like:

$$(3, 5, 2, 7, 6, 1, 4, 8).$$

This is commonly called "bubble sort". Now, at best this method would only have to read through a list of n numbers one time. At worst, it would have to read through the list n times. (The best case scenario would be if the list were already sorted, and the worst case scenario would be if the list were already sorted, but sorted the wrong way!).

Now, another way to sort these numbers would be to first divide the list up into two:

$$(3, 5, 2, 7)(6, 1, 4, 8)$$

and then divide these newly created smaller lists in two again:

$$(3, 5)(2, 7)(6, 1)(4, 8)$$

and continue to do this until all our sublists have two elements in them. (We're assuming that there are 2^k elements in our list. This might seem a bit artificial, but as algorithmic complexity is only concerned with large scale (asymptotic) behavior we'll find that this restriction doesn't matter.)

Now, once all our our sublists are only of size 2, we perform bubble sort on all of them. In other words, we exchange the numbers in a list if they're backwards, and do nothing if they're correct.

Once this is done, we merge our lists in the reverse order of how we split them. Now, here's the important thing. If we're merging two lists, and we know that each list is internally sorted, then to merge them and keep the two lists sorted we only need to perform one comparison for each element in the two lists. For example, if we've got the two lists:

$$(2, 3, 5, 7)(1, 4, 6, 8)$$

we look at the top two elements and grab the number 1. So, 4 becomes the top element in the right list. Then, we look at the top two elements and grab the number 2, so 3 becomes to top element in the left list. We then look at the top two elements and grab 3, so 5 becomes the top element in the left list. The important thing is that, because we know our two lists are internally sorted, we know the list we're forming from them in this way will be sorted itself. This type of search is typically called "merge sort".

How long will this take? If the list is of length n then chopping the list up will require $\log_2 n$ steps, running the initial sorts will require $\frac{n}{2}$ steps, and then each merge will require n steps. However, there will be a total of $log_2 n$ such merges. Therefore, the total running time of this algorithm will be:

$$\log_2 n + \frac{n}{2} + n \log_2 n + C$$

where C handles any constant time concerns (like starting up the computer).

Well, so what? The important idea is that both of these algorithms do the same thing, but in very different ways, and one way of doing it can take longer than another.

1.2 The Big O

When we talk about algorithmic complexity, we're basically talking about how long it takes to run an algorithm as a function of the size of the input to that algorithm. In our two examples up above, the size of our input was the number of elements in our list. Now, when talking about algorithmic complexity, we want to know the large scale behavior of the algorithm, which means we want to know how fast it grows as our input size gets bigger. This means that we're only cocerned with the fastest growing term in our time function. In the case of the second sorting algorithm above as n gets really big the term that will dominate will be $n \log n$ (dropping the base, as it won't matter). So, we say that this algorithm has a running time that is proportional, on a large scale, to $n \log n$. The way we would write this would be:

$$C(mergesort) = O(n \log n).$$

But what about our first example, with bubble sort? There we saw that for some inputs the time required could be proportional to n^2, while for other inputs it could be proportional to n. So, which one do we use? Well, for big-O notation, we always use the worst case scenario input. So, for bubble sort:

$$C(bubblesort) = O(n^2).$$

As $n \log n < n^2$ for large n we would say that merge sort is a more efficient algorithm. Is there a way that we could capture the idea that, at best, bubble sort is more efficient? Yes, we use big-Omega notation. For bubble sort we'd say:

$$C_B(bubblesort) = \Omega(n)$$

which says that, at best, bubble sort take a time proportional to n. Now, note that merge sort does exactly the same thing, and takes exactly the same time, not matter what input we give it. So, in this case the best case scenario would be the same as the worst case scenario. In this case we have:

$$C_B(mergesort) = \Omega(n \log n).$$

If an algorithm has $C = C_B$ that we express that in big-Theta. So, if both the best case and worst case running times for the algorithm are the same, like with merge sort, we would say:

$$C(mergesort) = \Theta(n \log n).^1$$

As mentioned earlier, we're concerned with worst case running times, so we always use big-O to determine which algorithm is "better".

[1] There are many other sorting algorithms, such as heap sort and quicksort. It has been proven for any "comparison sort" algorithm that the best running time possible is $n \log n$.

1.3 Two Algorithms with Massively Different Running Times

In most beginning progreamming classes you learn about the concept or recursion. That is a funciton that calls itself, and continues to call itself until it gets down to some base case. It then builds its final solution up from this base case. An example:

```
Fibonacci_{1} (int n)
{
  if(n == 0) return 0;
  else if(n == 1) return 1;
  else return Fibonacci(n-1) + Fibonacci(n-2);
}
```

This program will compute the nth term in the Fibonacci sequence for any non-negative integer n.

Pretty slick, huh? There's only one problem. This is a horrible way to write this program. It's incredibly slow. We can see why if we take a look at how many times the program has to call the function Fibonacci for a given input n. If $n = 0$ or $n = 1$ the answer is 1. Otherwise, the answer is:

$$CF(n) = CF(n-1) + CF(n-2) + 1$$

This is slightly more than $Fibonacci(n)$. Now, the closed-form solution for the Fibonacci terms is:

$$Fib(n) = \frac{(1+\sqrt{5})^n - (1-\sqrt{5})^n}{2^n\sqrt{5}}.$$

This funciton increases exponentially with n, and so our Fibonacci function has a running time that is exponential as a function of it input value n.

On the other hand, here's an example of a program that does the same thing, only much more efficiently:

```
Fibonacci_{2} (n)
{
  int i, j, k, temp;
  i = 0;
  j = 1;
  k = 0;

  if (n == 0) return 0;
  else{
    while(k < n)
    {
      temp = j;
      j = i+j;
      i = temp;
      k = k+1;
    }

    return j
  }
}
```

This program does exactly the same thing as our earlier Fibonacci program, in that for the same input it will always returns the same output, but it calculates the output in time that is directly proportional to the input n. This is a *huge* (linear vs. exponential) improvement over our first program. The second example here is a very basic example of a programming method called dynamic programming that can be used, if you're clever enough, to reduce the complexity of a program enormously.

Using our earlier notation for algorithmic complexity, here viewed as a functin of n, we have:

$$O(Fibonacci_1) = \left(\frac{1+\sqrt{5}}{2}\right)^n$$

while

$$O(Fibonacci_2) = n$$

2 P vs. NP

2.1 Euler and Hamilton Paths

Supposedly the oldest theorem in graph theory, which presaged the study of topology, is the Konigsberg bridge problem. During Euler's time the city of Koningsberg was divided by the river Pregel, with two large islands in the middle of the river. There were seven bridges connecting the city and the two islands, and schematically the situation looked like this:

The question was, is there a way of crossing all the bridges of Konigsberg such that you only have to cross each bridge once? In 1736 Euler proved that it's not possible.

Euler's result, which we'll explain in just a moment, applies to any graph. A path that crosses each edge of a graph once and only once is called an Euler path for the graph in honor of Euler. An Euler path that begins and ends on the same node is called an Euler circuit.

Euler proved that it's possible to find an Euler path if either every node has an even number of edges connected to it, or if two and only two nodes have an odd number of edges connected to them. There's even an algorithm for finding an Euler path if you know one exists. It works as follows:

1. If every node has an even number of edges, begin anywhere, if two nodes have an odd number of edges, begin on one of these nodes.

2. Every time you land on a node record it. So, each node should have a record of the number of times you've landed on it. When you're at a node, choose a edge from the available edge that takes you to

another node with the minimal number of previous visits. If there's a tie, any choice will do.

3. After you've crossed an edge, remove it from the set of available edges.

It's not very hard to prove, although we won't do it here, that this algorithm will always give you an Euler path if one exists. (Run through example)

Now, a question that might at first appear very similar is the question of a Hamiltonian path. A Hamiltonian path is like an Euler path, only instead of hitting every edge once and only once, a Hamiltonian path must hit every vertex once and only once. In our earlier example a Hamiltonian path is pretty easy to find, however if we just remove one edge to get the following graph:

a Hamiltonian path no longer exists! The question of if an algorithm exists for finding a Hamiltonian path and, if so, how complex that algorithm is leads us into the concept of NP problems.

2.2 P and NP

Let's take a look at the Hamiltonian path problem in a little bit more detail. First, we note that if we're given a candidate for a Hamiltonian path (a sequence of vertices) we can quickly check to see if that sequence is a Hamiltonian path. Second, we note that there obviously is an algorithm for figuring out whether or not there's a Hamiltonian path. You just enumerate all n! possible sequences of vertices (where n is the number of vertices) and you just check them all sequentially.

If we view the input size of a graph to just be the number of vertices and the number of edges (there are some technicalities we're sweeping under the rug here, that you can look up if you're interested, but they're not that important) we can see pretty quick that we can *verify* a solution (a sequence of edges) in a time that is polynomial in the size of the graph. However, checking all n! possibilities potentially requires an amount of time that is exponential in the size of the graph. So, at least using the enumerate and check approach, we cannot determine whether there's a Hamiltonian path in an amount of time that is polynomial in the size of the input.

Now, problems that are decidable (give a yes/no answer) in an amount of time that is polynomial in the size of the input are considered to be in a complexity class called P, which stands for polynomial. Problems that are verifiable (you can check if a given solution works) in an amount of time that is polynomial in the size of the input are considered to be in a complexity class called NP.[2]

[2]NP stands for nondeterministic polynomial. This basically means that if you could check all possible solutions *simultaneously* you could solve the problem in a polynomial amount of time. It's equivalent to being able to verify a solution in a polynomial amount of time.

2.3 NP-Completeness

Now, a very interesting property of this set of NP problems are the problems that are called NP-complete. The idea behind these problems is this. Suppose you have two problems K_1 and K_2, and for any instance of a problem in K_1 you have an algorithm, f, that would take that instance and convert it into an instance of a problem in K_2 in a polynomial amount of time. Furthermore, for any instance $x \in K_1$ the answer for that problem is yes if and only if the answer to the problem $f(x) \in K_2$ is also yes.

Well, what would that mean? That would mean that any solution algorithm in K_2 could be turned into a solution algorithm in K_1. Furthermore, a polynomial solution algorithm in K_2 would automatically give rise to a polynomial solution algorithm in K_1. Just map the problem from K_1 to K_2, and solve it using your polynomial time algorithm in K_2.

Now, the amazing thing is that there exists a class of problems, called NP-complete problems, that have the property that *any* other problem in NP can be converted into an NP-complete problem in a polynomial amount of time. This set of problems is, in a sense, the hardest possible set of NP problems in that if you can find a polynomial time solution to *any* of the NP-complete problems then you've got a polynomial time solution to *every* NP problem!

Now, proving a problem is NP-complete isn't that hard as long as you've got another problem that you know is NP-complete. To prove a problem K is NP-complete we must:

1. Prove $K \in NP$.
2. Select a known NP-complete problem K'.
3. Construct a polynomial time mapping from K' to K such that the mapping outputs a solvable instance in K if and only if the image instance in K' is solvable.

Usually, that's not very hard. An undergraduate class in complexity theory will have you figure out many of these.

2.4 Two NP-Complete Problems: SAT and 3-SAT

Now, there's a question here that's just begging to be answered. How do we find the first NP-complete problem? Well, as you might imagine, that's not a trivial problem. In 1971 Stephen Cook proved that the problem SAT is NP-complete. The proof is not too hard, but it's kind of long. If you want to read it check out the book: *Introduction to Automata Theory, Language, and Computation* by Hopcroft, Motwani and Ullman.

Now, the SAT problem can be stated as follows: given a set of variables and a logical sentence, is there a truth assignment for these variables that makes the sentence true? From our point of view a logical sentence will be a sentence in a form that looks like:

$$(x_1 \vee !x_2) \wedge (x_2 \vee x_3 \vee x_4 \vee x_5) \wedge (x_3 \vee x_4 \vee !x_5)$$

In other words, a bunch of variables or their negations, grouped into parenthetical groups by OR operators, with the parenthetical groups connected by AND operators. [3]

Now, a truth assignment that satisfies this sentence is easy. Just set every variable to true. However, for general logical sentences we don't know of any algorithm that is asymptotically better than just checking all possible truth combinations until we find one that works. Also, it's pretty obvious that whether or not a given truth assignment works can be verified in polynomial time, so this set of problems is in NP.

Now, the 3-SAT problem is exactly the same as the SAT problem, only instead of allowing arbitrary sets of variables connected by OR operators, we only allow sets of 3 variables connected by OR operators. So, for example:

$$(x_1 \vee x_2 \vee !x_1) \wedge (x_1 \vee !x_2 \vee x_2)$$

Now, the important idea here is that for any instance of SAT, we can convert it into an instance of 3-SAT. Here's how. Take a set of variables connected by OR operators in a sentence in SAT:

[3] This is how we're *defining* a logical sentence. It can be shown that the more common understanding of a logical sentence can always be converted into this form.

$$\{x_1, \ldots, x_n\}$$

Here we're putting any set of variables connected by OR operators into a set. If we have more than one such set, we assume these sets are themselves connected by AND operators.

Convert this set into $n-1$ sets utilizing $n-1$ new variables $z_1, \ldots, z_{n-3}$:

$$\{x_1, x_2, z_1\}, \{x_3, !z_1, z_2\}, \ldots, \{x_n, !z_{n-2}, z_{n-1}\}$$

Now, if there a truth assignment for the variables $x_1, \ldots, x_n$ that satisfies the original set then at least one of them must be true. Call that variable x_r. Then if we set z_{r-2} to TRUE and the other values of z_k to FALSE then we satisfy each of the new sets of variables. Conversely, if the new set of variables has an assignment that makes them true, at least one of the x_k must also be true. This conversion procedure obviously works in time that is polynomial in the size of the original set, so we have an appropriate mapping of an NP-complete problem into 3-SAT. 3-SAT is also clearly polynomial verifiable, so it's in NP, and therefore 3-SAT is NP complete.

2.5 The Big Question

So, right now nobody knows of any algorithm that solves an NP problem in polynomial time. However, nobody has yet been able to prove that no such algorithm exists. In fact, proving no such algorithm exists has proven to be a very difficult problem. So hard, in fact, that's in a milennium problem. In other words, if you can find a polynomial time algorithm for any NP problem, or prove that one doesn't exist, you'll get $1,000,000 and fame. Put mathematically, we want to know if the set containment:

$$P \subseteq NP$$

is proper.

If you found this talk interesting and would like to know more, you can take a computer science class on algorithms. At the U there is one taught every fall by Suresh Venkatasubramanian. You can check out the class webpage through his homepage.