20191114, 20191115

Advanced Complexity Analysis

Up to this point I have described some problems as being easy and some as being difficult. It's time to put that distinction on a solid scientific footing. We need to talk about different classes of problems.

When computers first became widely available, people all over the world started to create algorithms to solve computational problems. It was soon recognized that the problems they were working on fell into two loose categories: ones that were **easy** to solve with fast algorithms, and ones for which it seemed to be **impossible** to find any efficient algorithm at all.

Researchers looked for some way to formalize these concepts. The concept of computational complexity was applied, and it turned out that the "easy" problems can all be solved in polynomial time – ie they have algorithms that run in O(n^t) time for some constant t. None of the hard problems have polynomial time algorithms at all, even for very large values of k. That could be because we are not smart enough to find fast algorithms for these problems, or it could be because such algorithms don't exist. How can we tell?

The study of the difficulty of problems experienced an enormous breakthrough in the early 1970's. **Cook** and **Levin**, working independently and virtually simultaneously, proved a theorem that revolutionized our understanding of the relationship between easy problems and hard problems, and also provided us with an immensely powerful tool for identifying computationally hard problems. But we need to work up to that ...

Definition: a problem X is a **decision problem** if
- the answer to any instance of X is either Yes or No
- the answer to any instance of X is completely determined by the details of the instance

For example, let X = "Does the set S contain the value 3?" where S is a well-defined set such as {1, 10, 7, 3, 8}. X is a decision problem.

Another example: let X = "Does program P enter an infinite loop when the input is I?". This is the well-known Halting Problem, and it is a decision problem (even though we know there is no algorithm that can solve it for all possible programs and inputs). This illustrates the important point that not all decision problems are created equal!

Another example: let X = "Will this coin land Heads-up the next time it is tossed after this question is asked?". Here the answer is either Yes or No, but the answer is **not** determined by the details of the instance - tossing the coin is a random event and its outcome cannot be predicted with 100% accuracy (unless the coin is the same on both sides, or unless the universe is completely deterministic). This X is not a decision problem.

Definition: The class $\mathcal{P}$ is the set of all decision problems that can be solved (i.e. the complete details of the solution, if there is one, can be found) using a "real" computer in O(n^k) time, where k is constant for each problem.

For example, the problem "Given a graph on n vertices, are there three vertices that are all mutually adjacent?" can be solved by an algorithm that examines each of the $\binom{n}{3}$ possible solutions. Since such an algorithm clearly runs in O(n^3) time, this problem is in the class $\mathcal{P}$.

Definition: The class $\mathcal{NP}$: This class is a bit more complex than $\mathcal{P}$. First, we distinguish between **solving** a problem - actually finding the answer - and

verifying a solution - checking the details to make sure they are correct. For example, if the problem is "Given a set of integers, is the number 17 in the set?" the answer might be "Yes, it is in position 5 in the set". To verify this we would check the appropriate value to see if it really is 17.

Second, we imagine a type of computer, called a **Non-Deterministic Turing Machine** (this is where the $\mathcal{N}$ in $\mathcal{NP}$ comes from) which has the magic ability to guess the right answer to any decision problem, and which will provide us with the details if the answer is **Yes**. Once the NDTM has performed this magic trick, it relapses back into being a normal computer.

Now we can define $\mathcal{NP}$. $\mathcal{NP}$ is the set of all decision problems that can be solved by a NDTM which **verifies all Yes** answers in O(n^k) time, where k is constant for each problem.

Equivalently (and perhaps more comprehensibly), $\mathcal{NP}$ is the set of all decision problems for which if someone tells you the answer, if the answer is No you don't have to do anything, but if the answer is Yes and they give you the details of the answer, you can verify the correctness of that answer in polynomial time.

The first response when one hears about the class $\mathcal{NP}$ is often "What is the point of talking about problems that are solved on imaginary magical computers?"

We'll answer that question in two steps. First, it is important to see that $\mathcal{P}$ is contained in $\mathcal{NP}$. If we have an algorithm that correctly solves a problem and provides details of the solution, then we could use that same algorithm to verify the correctness of the Yes/No solution "guessed" by an NDTM.

Second, it fairly quickly became apparent that most of those "really hard" problems I have alluded to – the ones that nobody could find good algorithms for – actually do have the property that "Yes" answers can be verified quite easily. So the NDTM is a useful *conceptual* type of computer for discussing the solution of these problems.

Remember that the goal of this line of research was to find a practical way to distinguish between easy and hard problems. So far all we have is an imaginary magical computer – not very useful. But stay with me – this all leads to a very practical and essential tool in modern computer science.

The question to be addressed is, are there any problems in $\mathcal{NP}$ that are not in $\mathcal{P}$? In other words, is there any problem X such that X can be verified using an NDTM but X cannot be solved using a "real" computer? It would seem that there absolutely must be such problems - after all, the NDTM can magically guess right answers to everything, and all it has to do is verify the Yes answers. Surely being able to do magic must give you an advantage.

If there are no such problems, then the two classes $\mathcal{P}$ and $\mathcal{NP}$ are equal – which virtually nobody believes – because if $\mathcal{P} = \mathcal{NP}$ then real computers are just as powerful as magic computers.

To address the problem "Does $\mathcal{P} = \mathcal{NP}$?" people started trying to find the most difficult problems in $\mathcal{NP}$ - these are surely the best candidates for problems that in $\mathcal{NP}$ but not in $\mathcal{P}$. By the same token, if we can prove that the most difficult problems in $\mathcal{NP}$ are also in $\mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

But how can we identify the hardest problems in $\mathcal{NP}$? Our measure of problem difficulty so far has been the computational complexity of the best algorithm for solving the problem ... but for the problems we are interested in, we have **no idea** what the best algorithm is. We are in the uncomfortable position of trying to compare the difficulty of problems that we don't know how to solve.

It turns out we can do this in a clever way. We imagine two problems X and Y. We would like to show that X is "easier" than Y (or, more precisely, that X is "not harder" than Y). We do this indirectly by showing that **IF** we could find an efficient algorithm for Y, this would immediately give us an efficient algorithm for X.

To show this relationship between X and Y, we demonstrate that any instance of X (i.e. any specific set of values or objects that X applies to) can be transformed **in polynomial time** into an instance of Y (i.e. a specific set of values or objects that Y applies to) in an **answer-preserving** way. That is, if the answer to X on that specific set of values is **YES**, then the answer to Y on the transformed set of values is also **YES** (and similarly for **NO**).

Here's a simple example:

P_1: Given a set S of n integers, does S contain the value 4?

P_2: Given a set S of n integers and a target integer k, does S contain the value k?

Obviously, both of these problems are so simple that we can immediately see good algorithms for solving them. Ignore that for the moment - we are focusing on the relationship between the problems.

Suppose we are given an instance of P_1 (a specific set of integers), and suppose we have no idea how to solve it. We ask "If we knew how to solve P_2, how could we use that knowledge to solve P_1?"

Well, if we know how to solve P_2 for any set S and any integer k, we can write down an algorithm **solve_P_2(S,k)** for this problem. Then we could create an algorithm for P_1 like this:

```
def solve_$P_1$(S):
    return solve_$P_2$(S,4)
```

Putting it formally, the instance of P_1 is transformed into an instance of P_2 in constant time simply by assigning k the value 4, and the transformation is answer-preserving: the call to solve_P_2() returns "Yes" iff the correct answer to the instance of P_1 is "Yes".

This is actually the "proper" way to express what we are doing here. Given S which is an instance of P_1, we construct (S,4) which is an instance of P_2. We observe that the instance of P_2 is constructed in constant time, and that the answer to P_2(S,4) is identical to the answer to P_1(S).

Thus we can say that **if** we could solve P_2 efficiently, **then** we could also solve P_1 efficiently.

It is also possible to turn this around: if we could solve P_1 efficiently, we could also solve P_2. That is, any instance of P_2 (search for k) can be easily transformed into an instance of P_1 (search for 4). In class we discussed methods for doing this – it's a beneficial exercise which I recommend for anyone who was not in class when we discussed this.

Here's another, slightly more complex example. Consider these problems X and Y:

X: Given a set S of n integers, are there more positive than negative integers in the set? (NB: X is a decision problem, and it is in $\mathcal{NP}$)

Y: Given a set S of n integers, is the sum of the set positive? (Y is also a decision problem, and Y is also in $\mathcal{NP}$)

(Once again, it is obvious that we can solve both of these problems easily. I have chosen simple problems for these examples so that we can focus on the transformation process.)

Solving X requires counting, but solving Y involves adding, so how can we transform an instance of X into an instance of Y? The key insight is that counting is equivalent to adding 1's. So if we transform the instance of X into an instance of Y by replacing every positive integer by 1, and every negative integer by -1, then solving Y on the transformed set will give us a YES answer if and only if the answer to X on the original set is YES. So our transformation is answer-preserving, as required ... but is it a polynomial-time transformation? Yes it is, because all we need to do is make a single, constant-time change to each element of the set - the entire transformation requires O(n) time.

Thus if we can find an efficient algorithm for Y, we will also have an efficient algorithm for X.

Our term for this kind of transformation is **reduction**. We say **X reduces to Y**. This is confusing to many people because an intuitive interpretation of the word "reduce" often suggests "simplify". Here, the reduction goes from the "easier" problem (X) to the "harder" or "more general" problem (Y).

It is useful to remind ourselves exactly what we mean by **X reduces to Y: if we could solve Y, then we could also solve X** or **we can solve X by solving Y**

Reduction is NOT about:

- showing that there is an efficient algorithm for Y
- showing that there is an efficient algorithm for X
- showing that X is difficult or easy
- showing that Y is difficult or easy

Reduction IS about:

- showing that **IF** we could solve Y in polynomial time, **THEN** we could also solve X in polynomial time by transforming instances of X into instances of Y.

When we reduce problem X to problem Y, we often have no definite knowledge about efficient algorithms for either of the two problems.

The standard notation for reduction is $\propto$. So for our problems above, we write

$$X \propto Y$$

The next step in the argument is to observe that reduction is **transitive**. If $X \propto Y$, and $Y \propto Z$, then $X \propto Z$ - the transformation now takes two steps, but it is still polynomial-time, and (this is crucial) it is still answer-preserving.

Now we can imagine long chains of problems linked by reduction. The first problems in each chain would be quite easy, and as we move along the chains the problems get harder and harder. The problems we are searching for, the most difficult problems in $\mathcal{NP}$, will be at the far ends of the chains.

A natural question to ask is "Do these chains go on forever, or do they reach an end?" My (very satisfying) answer is "Both. The chains do go on forever, and they also reach an end." What I mean by this is that there is always another problem that we can reduce to, but the *difficulty* of these problems (as long as we stay in the class $\mathcal{NP}$) reaches a maximum level and stays there.

Finally, we can get back to Cook and Levin. To describe their amazing discovery and its importance, we need one or two more definitions.

SAT: SAT is a problem in $\mathcal{NP}$, defined as follows: Let E be a Boolean expression with n **literals** (a literal is just a Boolean variable, possibly negated). Each literal may occur more than once in E. Is there a way to assign True and False to the literals in E so that E is true?

Example: Let $E = (x_1 \wedge \neg x_2) \vee \neg x_1$ If we let $x_1 = True$ and $x_2 = False$, E evaluates to $True$, so E is satisfiable.

Example: Let $E = (x_1 \wedge \neg x_2) \wedge (x_2 \vee \neg x_1)$. E is not satisfiable ... you can verify this for yourself.

And now at last the jewel in the crown – the most important result in the history of the study of algorithms (which is the most important part of computer science ... but maybe I'm biased):

The Cook-Levin Theorem:

Let X be any problem in $\mathcal{NP}$. Then X reduces to SAT.

Reminder: This means that for every instance of every problem in $\mathcal{NP}$ we can construct a Boolean expression in polynomial time such that the original problem instance has answer "Yes" if and only if the Boolean expression is satisfiable.

****

Optional expansion on the significance of **Cook-Levin** – skip this if you are not quite as excited about this as I am!   The main story picks up again after another row of asterisks.

It's not remarkable that we can construct a Boolean expression that is satisfiable iff the problem instance has answer "Yes".   For example, consider the extremely simple decision problem "Given an integer value x, does x = 3?" Since we are talking about an instance of this problem, we are given the value of x and we can convert it to binary. Suppose x consists of just 3 bits $b_2 b_1 b_0$. Of course 3 in binary is 011 so we need a Boolean expression that is satisfiable iff $b_2 == 0$, $b_1 == 1$ and $b_0 == 1$

For each of these requirements we create a Boolean clause. If the requirement is satisfied we create the clause (v), and if the requirement is not satisfied we create the clause $(v \wedge \neg v)$ where v is a Boolean variable. Conjoining these clauses with "$\wedge$" operators gives us a Boolean expression that can be satisfied iff x = 3.

For example, suppose x is 2. Here $b_2 == 0, b_1 == 1, b_0 == 0$

The requirement $b_2 == 0$ is satisfied so our first Boolean clause is (v)

The requirement $b_1 == 1$ is satisfied so our next Boolean clause is (v)

The requirement $b_0 == 1$ is not satisfied so our last Boolean clause is $(v \wedge \neg v)$

Our complete Boolean expression is E = $(v) \wedge (v) \wedge (v \wedge \neg v)$

It's easy to see that this E is not satisfiable – it contains only one Boolean variable so there are only two possible truth assignments: either $v = True$ or $v = False$. For each of these, E evaluates to *False*.

Of course if x does equal 3 then the three requirements would all be satisfied so the Boolean expression would be E = $(v) \wedge (v) \wedge (v)$ which is obviously satisfiable.

This may seem like a lot of work to establish something pretty trivial, but it's a good illustration of the principle. We used the information available in the instance of the problem to construct an instance of SAT, and we did it in an answer-preserving way.

You might want to think about how you would approach something a bit more complex, such as "Given two integers x and y, does $x + y == 10$?" You may assume again that x and y are 3-bit numbers. You can consider all the possible pairs of x and y values that give the desired sum (eg x == 6, y == 4) and use the sort of expression we developed above to check the value of a variable to see if the given values of x and y fit any of the pairs that sum to 10.

There are just a few of these pairs : $(3,7), (4,6), (5,5), (6,4), (7,3)$ You can probably see how to build a Boolean expression that is satisfiable iff x and y match one of these pairs. Then you can "or" these pieces together to see if $x + y == 10$ is true or false. The final Boolean expression is going to be quite a bit longer than the Boolean expression we used for the "$x == 3$?" question.

And there's the rub! If we take this approach of building a Boolean expression by enumerating the possible ways the answer can be "Yes", building a Boolean expression for each them, and then sticking them all together to get our final Boolean expression, we will soon encounter exponential growth in the size of the Boolean expression. Any question to which the "Yes" answer can be triggered by some subset of a set (such as the Subset Sum Problem) would contain sub-expressions for testing each possible solution ... so it would have to have 2^n sub-expressions.

This is what makes **Cook-Levin** such an astonishing, brilliant, powerful result. They showed that for **any** instance of **any** problem in $\mathcal{NP}$, we can construct a Boolean expression that preserves the answer, and the size of the Boolean expression is a **polynomial** function of the size of the instance of the problem. We don't need an exponentially large Boolean expression to deal with exponentially many possible solutions.

Properly describing the Cook-Levin method for constructing the Boolean expression (which is completely different from the little examples I did above) would require a dive into formal languages and Turing Machines - there is no time in our course to do that. If you are interested, I strongly encourage you to explore this topic.

****

Picking up where we left off ... Cook and Levin showed that every problem in $\mathcal{NP}$ reduces to SAT.

This means if we could solve SAT in polynomial time then we could solve every problem in $\mathcal{NP}$ in polynomial time too. That would have a couple of interesting implications. For one thing, it would mean that all of the hard decision problems that have defeated everyone who has attempted to find good algorithms for them for the last 50 years, actually do have polynomial time algorithms. For another thing, it would mean $\mathcal{P} = \mathcal{NP}$ ... which means that our normal, real-world computers have just as much power as magical, guess-the-right-answer-every-time Non-Deterministic computers. Most people don't believe either of these things ... and so most people believe that SAT simply cannot be solved in polynomial time.

In other words, most people believe $\mathcal{P} \neq \mathcal{NP}$

... but you could be the one to prove us all wrong.