Complexity (23)

Michael L. Littman

December 8th, 1998

1 COMPLEXITY

1.1 Efficient Algorithms

We've looked at many different algorithms with varying run times.
Some problems, like topological sort and stable marriage, admitted low-overhead linear-time (in the size of the input) algorithms. Others, like solving a system of linear equations and sorting, took more time.
Many of the problems also had simple exponential-time algorithms, but we generally felt that these were not the ones we wanted.
These kind of observations have led theoreticians to make a distinction between polynomial-time and exponential-time algorithms, with exponential meaning "bad".

1.2 Hard Problems

We can use the notion of efficient algorithms to categorize problems: a hard problem is one that has no efficient (polynomial-time) algorithm. (This is "algorithms as a taxonomic tool!")
A natural question to ask is: Are there any hard problems?
Yes. You can study them (EXP-hard problems) in the complexity course. In fact, there are some problems whose answers cannot be computed!
Amazingly, there is a huge class of important problems whose status as hard or easy is unknown. What we do know is that they are all easy or all hard!
These are the NP-complete problems. Today will be a quick survey of why these problems are important in the study of algorithms. Next time we'll look at developing algorithms for NP-complete problems.

1.3 Degrees of Hardness

Like stones, we can categorize problems by how hard they are:

- not computable: halting problem.

- computable, but not in less than exponential time: succinct circuit value, generalized checkers.
- computable in polynomial time, but not in linear time: sorting using only comparisons (Section 9.1).
- computable in linear time: topological sort.

All of these are tight: can't do any better. The NP-complete problems are frustrating because we know how to solve them in exponential time, but we don't know we *can't* solve them in polynomial time. Big gap!

2 NP-COMPLETENESS

2.1 Decision Problems

To keep things simple, we will mainly concern ourselves with decision problems. These problems only require a single bit output: "yes" and "no".
How would you solve the following decision problems?

- Is this directed graph acyclic?

  DFS, look for visit to a gray node. Linear time.

- Is there a spanning tree of this undirected graph with total weight less than w?

  MST, compare resulting value to w. Polynomial time.

- Does this bipartite graph have a perfect (all nodes matched) matching?

  Find maximum matching and see if all nodes are matched. Polynomial time.

- Does the pattern p appear as a substring in text t?

  String search. Polynomial time.

2.2 Decision to Optimization

In the previous examples, we find an optimal answer to solve the decision problem.
How would you use a decision problem to solve the optimization problem?
For example, imagine we have a subroutine for telling us whether there is a spanning tree of a graph that has weight less than w. How can we find the minimum weight efficiently?

Binary search.

Now, how do we find the MST?

Self reducibility. Once we know the size of the MST, we can drop out an edge and see if we're still ok. Drop the edges that are ok to drop and keep the others. When we're through, we'll have the MST!

2.3 P

P is the set of decision problems that can be solved in worst-case polynomial time:

- If the input is of size n, the running time must be $O(n^k)$.
- Note that k can depend on the problem class, but not the particular instance.

All the decision problems mentioned above are in P.

2.4 Nice Puzzle

The class NP (meaning non-deterministic polynomial time) is the set of problems that might appear in a puzzle magazine: "Nice puzzle."
What makes these problems special is that they might be hard to solve, but a short answer can always be printed in the back, and it is easy to see that the answer is correct once you see it.
Example... Does matrix A have an LU decomposition?
No guarantee if answer is "no".

2.5 NP

Technically speaking:

- A problem is in NP if it has a short accepting certificate.
- An *accepting certificate* is something that we can use to quickly show that the answer is "yes" (if it is yes).
- *Quickly* means in polynomial time.
- *Short* means polynomial size.

This means that all problems in P are in NP (since we don't even need a certificate to quickly show the answer is "yes").
But other problems in NP may not be in P. Given an integer x, is it composite? How do we know this is in NP?

> Can prove it is by writing down its factors. Can check in polynomial time using multiplication.

2.6 Good Guessing

Another way of thinking of NP is it is the set of problems that can solved efficiently by a really good guesser.
The guesser essentially picks the accepting certificate out of the air (Non-deterministic Polynomial time). It can then convince itself that it is correct using a polynomial time algorithm. (Like a right-brain, left-brain sort of thing.)
Clearly this isn't a practically useful characterization: how could we build such a machine? (Although, researchers have been recently entranced by the idea that superposition properties of quantum mechanics can simultaneously try out all guesses quickly. But, I don't think this panned out.)

2.7 Exponential Upperbound

Another useful property of the class NP is that all NP problems can be solved in exponential time (EXP).
This is because we can always list out all short certificates in exponential time and check all $O(2^{n^k})$ of them.
Thus, P is in NP, and NP is in EXP. Although we know that P is not equal to EXP, it is possible that NP = P, or EXP, or neither. Frustrating!

2.8 NP-hardness

As we will see, some problems are at least as hard to solve as any problem in NP. We call such problems *NP-hard*.
How might we argue that problem X is at least as hard (to within a polynomial factor) as problem Y?

> Show we can use an algorithm for X to solve Y.

If X is at least as hard as Y, how would we expect an algorithm that is able to solve X to behave?

> Shouldn't take any longer to solve Y. Like a shortest path argument.

2.9 Reducibility

The idea of reducibility is really just a matter of using a solution to one problem as a subroutine in solving another.
We say problem Y is (polynomial) reducible to X if we can solve Y using a polynomial number of calls to an algorithm for X.
This implies that Y is as easy as X (X is at least as hard as Y).
We used this idea when we talked about least squares and solving linear equations: we can reduce least squares to solving linear equations very directly.

2.10 NP-Completeness

For a problem X to be NP-complete, it needs to satisfy two properties:

- X is in NP (short accepting certificate).
- X is NP-hard (an NP-hard problem can be reduced to it).

The NP-complete problems are the hardest problems in NP. If even one of them can be solved in polynomial time, then they *all* can.
Books of such problems exist...

3 NP-COMPLETE PROBLEMS

3.1 Set Covering

We want to make a small committee to look into global warming, but we want to make sure all of the following 7 groups are represented: Women, Homosexuals, Jews, African Americans, Tall people, Native Americans, Republicans.

Mick	F	F	F	F	F	T	F
Harvey	F	T	T	F	T	F	T
Wilt	F	F	F	F	T	T	F
Sue	T	T	T	F	F	T	F
Barney	F	F	F	F	T	F	F
Myrna	T	F	T	F	F	F	T
Tiffany	T	F	F	T	T	T	F

Minimal set cover: No one redundant. How solve this efficiently?

> One algorithm would be: Start with the full set of people. While there is a redundant person, remove him/her. When we're done, we cover the same set but no one can be removed.

Minimum set cover: Smallest possible committee. How solve this efficiently?

> Ah, there's the rub.

3.2 Decision-Problem Version

The decision-problem version is "Is there a set cover of size k or smaller?"
How could you use an algorithm for this to find the minimum set cover? Assume n people.

- Binary search: $\log n$ queries.
- Self-reduction: n queries to see who is needed.

Try it.

3.3 Hard Problem

There is no known polynomial-time algorithm for set cover.
Even more striking, the problem is NP-complete. This means that, unless P = NP, there is no polynomial-time algorithm for set cover.
We'll prove this shortly. NP-complete = in NP + NP-hard. Argue that it is in NP. We'll do hardness shortly.

> If I give you a possible cover, we can check that it has no more than k people and covers the set.

3.4 SAT

To prove a problem is NP-hard, it helps to have other NP-hardness problems around to reduce to it.
Cook proved the first NP-hard problem.
3CNF-SAT (sometimes just SAT) is a popular version of the problem:

- Given a Boolean formula in conjunctive normal form with at most 3 variables per clause, is there an assignments to its variables that makes the formula true?

Example...
SAT is in NP: assignment is the certificate.
SAT is NP-hard essentially because we can "build" a special guessing computer out of logic gates. We can write any problem in NP as a program for this computer.
This computer has some non-deterministic components that express an accepting certificate.
The formula will be satisfiable only if the computer will print "yes" for some certificate.

3.5 Hardness of Set Cover

Now we have one NP-complete problem.
Can prove set cover is no easier by showing how an algorithm for set cover could be used to solve SAT.
How can we determine satisfiabilty with a polynomial number of calls to an algorithm for set cover?

- First, augment the formula with a clause for all n variables (contains positive and negative literal).
- Now, each clause becomes something to be covered (a "minority group").
- Each literal (positive or negative variable) becomes a potential committee member that "covers" the clauses that it appears in.
- Problem has a size n cover if and only if formula is satisfiable.

Example...

3.6 Fine Points

- Can't have a set cover of size smaller than n because of the "variable" clauses.
- Further, a size n cover will not include a positive and negative literal for the same variable. Therefore, it corresponds to an assignment.
- A size n cover will have a true variable in every clause, therefore it will correspond to a satisfying assignment.

Amazing how SAT can hide itself in other problems: set covering is just SAT in disguise!

3.7 Graph Coloring

We'd like to take a map consisting of a set of non-overlapping regions and "color" it (assign a color to each region) so that no two contiguous regions have the same color. Want to minimize the number of colors.
Important in scheduling problems.
What's the natural decision problem?

> k-colorability.

3.8 Solving 3-colorability

Algorithm for solving it?
In NP?

> Sure... show me the colors.

How use an algorithm for 3-colorability to solve SAT?

3.9 Reduction

Not too bad.

- Triangle: true, false, red.
- Each literal becomes a node with an edge to "red." Its color (which can only be "true" or "false") represents its assignment. A literal and its complement are connected by an edge to assure that they don't get the same color.
- Each clause becomes a set of 5 nodes (widget): insures that all 3 literals are not "false." (Proof by exhaustion.)

```
x-o
  |\
y-o-o-o
      |\
z-----o-true

 x
 |\
~x-red
```

Graph has a 3 coloring if and only if formula has a satisfying assignment.

3.10 Colorability Complexity for Planar Graphs

A planar graph is one that you can draw on paper without edge crossings. Example: $K_{3,2}$ vs. $K_{3,3}$.
Complexity of determining k-colorability for planar graphs as a function of k:

- 1: $O(1)$ (can't do it if there are any edges at all).
- 2: $O(|V| + |E|)$ (BFS!).
- 3: NP-complete (special widget to take general reduction and eliminate edge crossings).
- 4: $O(1)$ (famous 4-color map theorem, always "yes").

The number 3 comes up in many NP-complete problems!

4 PRACTICE PROBLEMS

- Explain how to use a solution to the bipartite matching decision problem to find a maximum bipartite matching of a graph. Hint: First use a binary search to find the size of the maximum matching, then use self-reducibility to find the set of edges that attains this value.
- CLR Exercise 36.4-7 (pg. 946).
- CLR Exercise 36.5-2 (pg. 960).