ICS 161: Design and Analysis of Algorithms Lecture notes for March 12, 1996

NP-Completeness

So far we've seen a lot of good news: such-and-such a problem can be solved quickly (in close to linear time, or at least a time that is some small polynomial function of the input size).

NP-completeness is a form of bad news: evidence that many important problems can't be solved quickly.

Why should we care?

These NP-complete problems really come up all the time. Knowing they're hard lets you stop beating your head against a wall trying to solve them, and do something better:

- Use a heuristic. If you can't quickly solve the problem with a good worst case time, maybe you can come up with a method for solving a reasonable fraction of the common cases.
- Solve the problem approximately instead of exactly. A lot of the time it is possible to come up with a provably fast algorithm, that doesn't solve the problem exactly but comes up with a solution you can prove is close to right.
- Use an exponential time solution anyway. If you really have to solve the problem exactly, you can settle down to writing an exponential time algorithm and stop worrying about finding a better solution.
- Choose a better abstraction. The NP-complete abstract problem you're trying to solve presumably comes from ignoring some of the seemingly unimportant details of a more complicated real world problem. Perhaps some of those details shouldn't have been ignored, and make the difference between what you can and can't solve.

Classification of problems

The subject of *computational complexity theory* is dedicated to classifying problems by how hard they are. There are many different classifications; some of the most common and useful are the following. (One technical point: these are all really defined in terms of yes-or-no problems -- does a certain structure exist rather than how do I find the structure.)

- **P**. Problems that can be solved in polynomial time. ("P" stands for polynomial.) These problems have formed the main material of this course.
- **NP**. This stands for "nondeterministic polynomial time" where nondeterministic is just a fancy way of talking about guessing a solution. A problem is in NP if you can quickly (in polynomial time) test whether a solution is correct (without worrying about how hard it might be to find the solution). Problems in NP are still relatively easy: if only we could guess the right solution, we could then quickly test it.

NP does not stand for "non-polynomial". There are many complexity classes that are much harder than

NP.

- **PSPACE**. Problems that can be solved using a reasonable amount of memory (again defined formally as a polynomial in the input size) without regard to how much time the solution takes.
- **EXPTIME**. Problems that can be solved in exponential time. This class contains most problems you are likely to run into, including everything in the previous three classes. It may be surprising that this class is not all-inclusive: there are problems for which the best algorithms take even more than exponential time.
- **Undecidable**. For some problems, we can prove that there is no algorithm that always solves them, no matter how much time or space is allowed. One very uninformative proof of this is based on the fact that there are as many problems as there real numbers, and only as many programs as there are integers, so there are not enough programs to solve all the problems. But we can also define explicit and useful problems which can't be solved.

Although defined theoretically, many of these classes have practical implications. For instance P is a very good approximation to the class of problems which can be solved quickly in practice -- usually if this is true, we can prove a polynomial worst case time bound, and conversely the polynomial time bounds we can prove are usually small enough that the corresponding algorithms really are practical. NP-completeness theory is concerned with the distinction between the first two classes, P and NP.

Examples of problems in different classes

Example 1: Long simple paths.

A *simple path* in a graph is just one without any repeated edges or vertices. To describe the problem of finding long paths in terms of complexity theory, we need to formalize it as a yes-or-no question: given a graph G, vertices s and t, and a number k, does there exist a simple path from s to t with at least k edges? A solution to this problem would then consist of such a path.

Why is this in NP? If you're given a path, you can quickly look at it and add up the length, double-checking that it really is a path with length at least k. This can all be done in linear time, so certainly it can be done in polynomial time.

However we don't know whether this problem is in P; I haven't told you a good way for finding such a path (with time polynomial in m,n, and K). And in fact this problem is NP-complete, so we believe that no such algorithm exists.

There are algorithms that solve the problem; for instance, list all 2^m subsets of edges and check whether any of them solves the problem. But as far as we know there is no algorithm that runs in polynomial time.

Example 2: Cryptography.

Suppose we have an encryption function e.g.

```
    code=RSA(key,text)
```

The "RSA" encryption works by performing some simple integer arithmetic on the code and the key, which consists of a pair (p,q) of large prime numbers. One can perform the encryption only knowing the product pq; but to decrypt the code you instead need to know a different product, (p-1)(q-1).

A standard assumption in cryptography is the "known plaintext attack": we have the code for some message, and we know (or can guess) the text of that message. We want to use that information to discover the key, so we can decrypt other messages sent using the same key.

Formalized as an NP problem, we simply want to find a key for which code=RSA(key,text). If you're given a key, you can test it by doing the encryption yourself, so this is in NP.

The hard question is, how do you find the key? For the code to be strong we hope it isn't possible to do much better than a brute force search.

Another common use of RSA involves "public key cryptography": a user of the system publishes the product pq, but doesn't publish p, q, or (p-1)(q-1). That way anyone can send a message to that user by using the RSA encryption, but only the user can decrypt it. Breaking this scheme can also be thought of as a different NP problem: given a composite number pq, find a factorization into smaller numbers.

One can test a factorization quickly (just multiply the factors back together again), so the problem is in NP. Finding a factorization seems to be difficult, and we think it may not be in P. However there is some strong evidence that it is not NP-complete either; it seems to be one of the (very rare) examples of problems between P and NP-complete in difficulty.

Example 3: Chess.

We've seen in the news recently a match between the world chess champion, Gary Kasparov, and a very fast chess computer, Deep Blue. The computer lost the match, but won one game and tied others.

What is involved in chess programming? Essentially the sequences of possible moves form a tree: The first player has a choice of 20 different moves (most of which are not very good), after each of which the second player has a choice of many responses, and so on. Chess playing programs work by traversing this tree finding what the possible consequences would be of each different move.

The tree of moves is not very deep -- a typical chess game might last 40 moves, and it is rare for one to reach 200 moves. Since each move involves a step by each player, there are at most 400 positions involved in most games. If we traversed the tree of chess positions only to that depth, we would only need enough memory to store the 400 positions on a single path at a time. This much memory is easily available on the smallest computers you are likely to use.

So perfect chess playing is a problem in PSPACE. (Actually one must be more careful in definitions. There is only a finite number of positions in chess, so in principle you could write down the solution in constant time. But that constant would be very large. Generalized versions of chess on larger boards are in PSPACE.)

The reason this deep game-tree search method can't be used in practice is that the tree of moves is very bushy, so that even though it is not deep it has an enormous number of vertices. We won't run out of space if we try to traverse it, but we will run out of time before we get even a small fraction of the way through. Some pruning methods, notably "alpha-beta search" can help reduce the portion of the tree that needs to be examined, but not enough to solve this difficulty. For this reason, actual chess programs instead only search a much smaller depth (such as up to 7 moves), at which point they don't have enough information to evaluate the true consequences of the moves and are forced to guess by using heuristic "evaluation functions" that measure simple quantities such as the total number of pieces left.

Example 4: Knots.

If I give you a three-dimensional polygon (e.g. as a sequence of vertex coordinate triples), is there some way of twisting and bending the polygon around until it becomes flat? Or is it knotted?

There is an algorithm for solving this problem, which is very complicated and has not really been adequately analyzed. However it runs in at least exponential time.

One way of proving that certain polygons are not knots is to find a collection of triangles forming a surface with the polygon as its boundary. However this is not always possible (without adding exponentially many new vertices) and even when possible it's NP-complete to find these triangles.

There are also some heuristics based on finding a non-Euclidean geometry for the space outside of a knot that work very well for many knots, but are not known to work for all knots. So this is one of the rare examples of a problem that can often be solved efficiently in practice even though it is theoretically not known to be in P.

Certain related problems in higher dimensions (is this four-dimensional surface equivalent to a four-dimensional sphere) are provably undecidable.

Example 5: Halting problem.

Suppose you're working on a lab for a programming class, have written your program, and start to run it. After five minutes, it is still going. Does this mean it's in an infinite loop, or is it just slow?

It would be convenient if your compiler could tell you that your program has an infinite loop. However this is an undecidable problem: there is no program that will always correctly detect infinite loops.

Some people have used this idea as evidence that people are inherently smarter than computers, since it shows that there are problems computers can't solve. However it's not clear to me that people can solve them either. Here's an example:

```
main()
{
int x = 3;
for (;;) {
    for (int a = 1; a <= x; a++)
    for (int b = 1; b <= x; b++)
        for (int c = 1; c <= x; c++)
        for (int i = 3; i <= x; i++)
            if(pow(a,i) + pow(b,i) == pow(c,i))
            exit;
    x++;
}
}
```

This program searches for solutions to Fermat's last theorem. Does it halt? (You can assume I'm using a multiple-precision integer package instead of built in integers, so don't worry about arithmetic overflow complications.) To be able to answer this, you have to understand the recent proof of Fermat's last theorem. There are many similar problems for which no proof is known, so we are clueless whether the corresponding programs halt.

Problems of complexity theory

The most famous open problem in theoretical science is whether P = NP. In other words, if it's always easy to check a solution, should it also be easy to find the solution?

We have no reason to believe it should be true, so the expectation among most theoreticians is that it's false. But we also don't have a proof...

So we have this nice construction of complexity classes P and NP but we can't even say that there's one problem in NP and not in P. So what good is the theory if it can't tell us how hard any particular problem is to solve?

NP-completeness

The theory of NP-completeness is a solution to the practical problem of applying complexity theory to individual problems. NP-complete problems are defined in a precise sense as the hardest problems in P. Even though we don't know whether there is any problem in NP that is not in P, we can point to an NP-complete problem and say that if there are any hard problems in NP, that problems is one of the hard ones.

(Conversely if everything in NP is easy, those problems are easy. So NP-completeness can be thought of as a way of making the big P=NP question equivalent to smaller questions about the hardness of individual problems.)

So if we believe that P and NP are unequal, and we prove that some problem is NP-complete, we should believe that it doesn't have a fast algorithm.

For unknown reasons, most problems we've looked at in NP turn out either to be in P or NP-complete. So the theory of NP-completeness turns out to be a good way of showing that a problem is likely to be hard, because it applies to a lot of problems. But there are problems that are in NP, not known to be in P, and not likely to be NP-complete; for instance the code-breaking example I gave earlier.

Reduction

Formally, NP-completeness is defined in terms of "reduction" which is just a complicated way of saying one problem is easier than another.

We say that A is easier than B, and write A < B, if we can write down an algorithm for solving A that uses a small number of calls to a subroutine for B (with everything outside the subroutine calls being fast, polynomial time). There are several minor variations of this definition depending on the detailed meaning of "small" -- it may be a polynomial number of calls, a fixed constant number, or just one call.

Then if A < B, and B is in P, so is A: we can write down a polynomial algorithm for A by expanding the subroutine calls to use the fast algorithm for B.

So "easier" in this context means that if one problem can be solved in polynomial time, so can the other. It is possible for the algorithms for A to be slower than those for B, even though A < B.

As an example, consider the Hamiltonian cycle problem. Does a given graph have a cycle visiting each vertex exactly once? Here's a solution, using longest path as a subroutine:

```
for each edge (u,v) of G
if there is a simple path of length n-1 from u to v
    return yes      // path + edge form a cycle
    return no
```

This algorithm makes m calls to a longest path subroutine, and does O(m) work outside those subroutine calls, so it shows that Hamiltonian cycle < longest path. (It doesn't show that Hamiltonian cycle is in P, because we don't know how to solve the longest path subproblems quickly.)

As a second example, consider a polynomial time problem such as the minimum spanning tree. Then for every other problem B, B < minimum spanning tree, since there is a fast algorithm for minimum spanning trees using a subroutine for B. (We don't actually have to call the subroutine, or we can call it and ignore its results.)

Cook's Theorem

We are now ready to formally define NP-completeness. We say that a problem A in NP is NP-complete when, for every other problem B in NP, B < A.

This seems like a very strong definition. After all, the notion of reduction we've defined above seems to imply that if B < A, then the two problems are very closely related; for instance Hamiltonian cycle and longest path are both about finding very similar structures in graphs. Why should there be a problem that closely related to all the different problems in NP?

Theorem: an NP-complete problem exists.

We prove this by example. One NP-complete problem can be found by modifying the halting problem (which without modification is undecidable).

> **Bounded halting**. This problem takes as input a program X and a number K. The problem is to find data which, when given as input to X, causes it to stop in at most K steps.

To be precise, this needs some more careful definition: what language is X written in? What constitutes a single step? Also for technical reasons K should be specified in *unary* notation, so that the length of that part of the input is K itself rather than O(log K).

For reasonable ways of filling in the details, this is in NP: to test if data is a correct solution, just simulate the program for K steps. This takes time polynomial in K and in the length of program. (Here's one point at which we need to be careful: the program can not perform unreasonable operations such as arithmetic on very large integers, because then we wouldn't be able to simulate it quickly enough.)

To finish the proof that this is NP-complete, we need to show that it's harder than anything else in NP. Suppose we have a problem A in NP. This means that we can write a program PA that tests solutions to A, and halts within polynomial time p(n) with a yes or no answer depending on whether the given solution is really a solution to the given problem. We can then easily form a modified program PA' to enter an infinite loop whenever it would halt with a no answer. If we could solve bounded halting, we could solve A by passing PA' and p(n) as arguments to a subroutine for bounded halting. So A < bounded halting. But this argument works for every problem in NP, so bounded halting is NP-complete.

How to prove NP-completeness in practice

The proof above of NP-completeness for bounded halting is great for the theory of NP-completeness, but doesn't help us understand other more abstract problems such as the Hamiltonian cycle problem.

Most proofs of NP-completeness don't look like the one above; it would be too difficult to prove anything else that way. Instead, they are based on the observation that if A < B and B < C, then A < C. (Recall that these relations are defined in terms of the existence of an algorithm that calls subroutines. Given an algorithm that solves A with a subroutine for B, and an algorithm that solves B with a subroutine for C, we can just use the second algorithm to expand the subroutine calls of the first algorithm, and get an algorithm that solves A with a subroutine for C.)

As a consequence of this observation, if A is NP-complete, B is in NP, and A < B, B is NP-complete. In practice that's how we prove NP-completeness: We start with one specific problem that we prove NP-complete, and we then prove that it's easier than lots of others which must therefore also be NP-complete.

So e.g. since Hamiltonian cycle is known to be NP-complete, and Hamiltonian cycle < longest path, we can deduce that longest path is also NP-complete.

Starting from the bounded halting problem we can show that it's reducible to a problem of simulating circuits (we know that computers can be built out of circuits, so any problem involving simulating computers can be translated to one about simulating circuits). So various circuit simulation problems are NP-complete, in particular Satisfiability, which asks whether there is an input to a Boolean circuit that causes its output to be one.

Circuits look a lot like graphs, so from there it's another easy step to proving that many graph problems are NP-complete. Most of these proofs rely on constructing *gadgets*, small subgraphs that act (in the context of the graph problem under consideration) like Boolean gates and other components of circuits.

There are many problems already known to be NP-complete, and listed in the bible of the subject:

> Computers and Intractibility:
> A guide to the theory of NP-completeness
> Michael R. Garey and David S. Johnson
> W. H. Freeman, 1979.

If you suspect a problem you're looking at is NP-complete, the first step is to look for it in Garey and Johnson. The second step is to find as similar a problem as you can in Garey and Johnson, and prove a reduction showing that similar problem to be easier than the one you want to solve. If neither of these works, you could always go back to the methods described in the rest of this class, and try to find an efficient algorithm...

ICS 161 -- Dept. Information & Computer Science -- UC Irvine
Last update: