Computability and Computational Complexity

Jan Hązła

ETH Zürich

February 11, 2014

Introduction

So we get a computational problem. What can we do with it?

- Find a fast algorithm $\rightarrow$ IOI
- Prove there is no fast algorithm $\rightarrow$ Complexity theory
- Prove there is no algorithm at all $\rightarrow$ Computability theory

Outline

Introduction

We need a formal setting:

- Pick your favourite programming language.
- Program and data can always be converted to a string of bits.

Introduction – problems

But what is a problem, anyway? Examples:

- Given integer, decide if it is prime or composite (PRIMES).
- Given simple graph, decide if it has a cycle that visits every vertex exactly once (HAMILTON-CYCLE).
- Given simple graph, decide if it has a cycle that visits every edge exactly once (EULER-CYCLE).
- Given program A and input x, decide if $A(x)$ stops or runs forever (HALTING).

Introduction – problems

Further examples:

- Given set of integers and another integer k, decide if it has non-empty subset with total sum k (SUBSET-SUM).
- Given simple graph and k, decide if there is a set of k vertices that "touches" all edges (VERTEX-COVER).
- Given n tasks that take time $t_1, \ldots, t_n$ and a deadline T, decide if you can divide them between two processors such that processing is finished before the deadline (JOB-SCHEDULING).
- Given polynomial with integer coefficients $p(x_1, \ldots, x_n)$, decide if there exist integer coordinates $z_1, \ldots, z_n$ such that $p(z_1, \ldots, z_n) = 0$ (POLY-INT-ZERO).

Introduction – problems

To sum up:

- We are interested in decision problems, with input and "yes"/"no" answer.
- There is usually a natural notion of input size (useful later).
- Worst-case analysis: we want programs that are correct on *every* input.

Exercise

Q: What about non-decision problems?

A: Handled by standard tricks. For example, take VERTEX-COVER. How to compute size of smallest VC if we can decide if there exists VC of size k? How to compute smallest VC if we can compute size of smallest VC?

Amazing fact: there are problems for which no algorithm exists!

Halting problem

Given program A and input x, decide if $A(x)$ stops or runs forever (HALTING).

- For sure we can run $A(x)$ and see if it stops...
- But how long do we wait?
- Maybe if the program loops we can detect it... But have you ever heard of busy beavers?

For some nasty programs runs forever is a better phrase than loops forever.

Proof of undecidability

Proof by contradiction! Assume there exists program B that solves halting problem on every input.

Remember that both programs and inputs are just strings of bits. We can make an (infinite) list of them all:
$\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$

Assuming this order we can interpret this list as enumeration of all possible programs $A_1, A_2, \ldots$ or of all possible inputs $x_1, x_2, \ldots$

Proof of undecidability

With that in mind we write the following program:

```
C(x_i) {
    if (B(A_i, x_i)) {
        run forever;
    } else {
        stop;
    }
}
```

Proof of undecidability

Let us make a huge (infinite) 2D table. Rows are inputs, columns are programs. In row x_i and column A_j we put 1 if $A_j(x_i)$ stops, and 0 otherwise:

	A_1	A_2	A_3	A_4	$\cdots$
x_1	1	0	1	1	$\cdots$
x_2	1	0	1	0	$\cdots$
x_3	1	1	1	1	$\cdots$
x_4	1	0	1	0	$\cdots$

$\cdots$

Proof of undecidability

Let us make a huge (infinite) 2D table. Rows are inputs, columns are programs. In row x_i column A_j we put 1 if $A_j(x_i)$ stops, and 0 otherwise:

	A_1	A_2	A_3	A_4	$\cdots$
x_1	1	0	1	1	$\cdots$
x_2	1	0	1	0	$\cdots$
x_3	1	1	1	1	$\cdots$
x_4	1	0	1	0	$\cdots$
		$\cdots$			

C	A_1	A_2	A_3	A_4	$\cdots$
x_1	1/0	0	1	1	$\cdots$
x_2	1	0/1	1	0	$\cdots$
x_3	1	1	1/0	1	$\cdots$
x_4	1	0	1	0/1	$\cdots$
		$\cdots$			

C cannot be any of $A_1, A_2, \ldots$! Therefore, C cannot exist. Therefore, B cannot exist.

This is called the diagonal method.

Some comments

This sounds like cheating, right?

Exercise

We just showed that every program fails to solve HALTING on *at least one input*. Show that every program fails to solve HALTING on *infinite number of inputs*.

Fact

Q: One more of example programs is undecidable. Which one? A: POLY-INT-ZERO

Preliminaries

To prove there is no fast algorithm for a problem, we need to define “fast” first.

- Assign size n to every input.
- In principle size should be a number of bits used to describe the input, but it is ok if it is another (polynomially related) notion.
- Worst-case analysis: maximum running time of the algorithm over all inputs of size n.
- $O()$-notation: constant factors and low-order terms ignored: $100n^2 + 10n = O(n^2)$.

Running times

Assume you have an algorithm that runs in time t for inputs of size n.

- Good: polynomial time, e.g., $O(n^2)$ — to compute input of size $2n$ in time t you need four times faster machine.
- Bad: exponential time, e.g.: $O(3^n)$ — to compute input of size $n+1$ in time t you need three times faster machine.
- Polynomial time is $n^c = 2^{c \log n}$. Exponential time is $c^n = 2^{(\log c)n}$. There is a lot of functions "in between", e.g., $n^{\log n} = 2^{\log^2 n}$, $2^{\sqrt{n}}$, etc. They are also "bad".

Running times

Problem

Q: What about runtimes $10^{20} \cdot n$, or $10^{-3} \cdot (1.000001)^n$?

P vs. NP

P is a class of all problems that have algorithms that run in $O(n^c)$ for some c. For example, PRIMES has an algorithm running in $O(n^6)$, hence it is in P.

Another very important class is called NP. These are problems that are "easy-to-verify".

Definition of NP

The inputs for each problem can be divided into "yes"-instances and "no"-instances.

A problem is in NP if there exists an efficient (polynomial-time) way of verifying instances, such that:

- For each "yes"-instance there exists *some* proof.
- For each "no"-instance there is *no* proof.

What is a "proof"? A bitstring. How do you verify it? By a program that takes proof as additional input.

Definition of NP – example

Given simple graph, decide if it has a cycle that visits every vertex exactly once (Hamilton cycle).

A proof is a cycle in the graph and we verify it by checking if it is a Hamilton cycle.

Definition of NP – example

Given set of integers and another integer k, decide if it has non-empty subset with total sum k (subset sum).

A proof is a subset and we verify it by checking if its sum is k.

Definition of NP – exercises

Exercise

- Is primality testing in NP?
- Show that P $\subseteq$ NP.
- Is halting problem in NP?

P vs. NP

Problem

Are P and NP equal? Good question.

But why is the answer important?

NP-completeness

So far we do not have high hopes for proving either $\mathrm{P} = \mathrm{NP}$ or $\mathrm{P} \neq \mathrm{NP}$.

But we can prove this: there exists a huge range (hundreds) of problems that are "NP-complete".

- If $\mathrm{P} = \mathrm{NP}$, *all* those problems are in P.
- If $\mathrm{P} \neq \mathrm{NP}$, *all* of them are outside P.

In particular, Hamilton cycle, subset sum, vertex cover and job scheduling are all NP-complete.

Proving NP-completeness

How do you prove NP-completeness for a problem L?

- First, show $L \in \mathrm{NP}$. That way $\mathrm{P} = \mathrm{NP} \implies L \in \mathrm{P}$.
- Second, take some problem L' that you already know is NP-complete and show that $L \in \mathrm{P} \implies L' \in \mathrm{P}$ (in a sense you show that L' is not harder than L).

Problem

Q: But you need to start with the first NP-complete problem? How do you get it?
A: Well. . . somehow. It is called Cook's theorem.

Proving NP-completeness – example

Example. Recall:

- Given simple graph and k, decide if there is a set of k vertices that "touches" all edges (VERTEX-COVER).
- Given set of integers and another integer k, decide if it has non-empty subset with total sum k (SUBSET-SUM).

Assume we know that VERTEX-COVER is NP-complete. How to show that SUBSET-SUM is NP-complete?

Does SUBSET-SUM $\in$ NP? Yes!

Proving NP-completeness – reductions

To show: SUBSET-SUM $\in$ P $\implies$ VERTEX-COVER $\in$ P.

How to prove this step? Using a reduction.

Assume you have a program A that puts SUBSET-SUM in P.
Write a program (reduction) that will call A as subprocedure and (efficiently) solve VERTEX-COVER.

Reduction VERTEX-COVER $\leq_p$ SUBSET-SUM

Given G with n vertices and m edges and $0 \leq k < n$ the reduction constructs SUBSET-SUM instance as follows:

There are $n + m$ numbers, each of them with $m + 1$ digits *in base n*.

Reduction VERTEX-COVER $\leq_p$ SUBSET-SUM

For each vertex u we construct number s_u:

	d_0	d_1	d_2	$\cdots$	d_e	$\cdots$	d_m
s_u	1	0	1	$\cdots$	1	$\cdots$	0

where d_0 is always 1 and $d_e = 1$ if and only if u is one of the endpoints of e.

For each edge e we construct number t_e:

	d_0	d_1	d_2	$\cdots$	d_e	$\cdots$	d_m
t_e	0	0	0	$\cdots$	1	$\cdots$	0

with a single 1 at digit e.

Reduction VERTEX-COVER $\leq_p$ SUBSET-SUM

In total, we want:

$+$	d_0	d_1	d_2	$\cdots$	d_e	$\cdots$	d_m
			$\cdots$				
s_u	1	0	1	$\cdots$	1	$\cdots$	0
			$\cdots$				
t_e	0	0	0	$\cdots$	1	$\cdots$	0
			$\cdots$				
$k' =$	k	2	2	$\cdots$	2	$\cdots$	2

Reduction VERTEX-COVER $\leq_p$ SUBSET-SUM

We (efficiently) constructed a SUBSET-SUM instance from VERTEX-COVER instance.

If we prove:

- "yes"-instance of VERTEX-COVER is always mapped to "yes"-instance of SUBSET-SUM.
- "no"-instance of VERTEX-COVER is always mapped to "no"-instance of SUBSET-SUM,

we are done! We can solve VERTEX-COVER by mapping it to SUBSET-SUM and returning whatever SUBSET-SUM algorithm returns.

Exercise

Prove mapping property from the previous slide.

Congratulations! You proved a problem NP-complete. There is a whole theory of various smart ("gadget") reductions like that.

P vs. NP – some consequences

1. If $\mathrm{P} = \mathrm{NP}$:
    - Huge progress in solving problems.
    - "Creativity" not significantly harder than "verifying".
    - Almost all cryptography used in practice is broken.
2. If $\mathrm{P} \neq \mathrm{NP}$:
    - Cryptography is provably safe.
    - Randomized algorithms do not give significant speedup.

Both bullet points require stronger assumptions than $\mathrm{P} \neq \mathrm{NP}$.

What else?

Oracles, circuits, randomized algorithms, space complexity, approximation algorithms, interactive proofs, quantum algorithms, cryptography, communication complexity, natural proofs. . .

THANKS!