Lecture 1:
Introduction, existence of unsolvable problems, and the Turing machine

Valentine Kabanets

September 3, 2013

1 Why this course?

1. Foundations of CS.
2. Basic questions: What is a computer? What can and can't it do?
3. Intrinsic properties of Computation.

2 Course outline

1. **Computer viruses/worms.** Write a program that prints its own source code, without using any system calls (print commands are, of course, allowed).

   Possible! Write it yourself as an exercise!

2. **Perfect virus detection software.** Write a computer program that detects whether any given program prints its own text.

   This turns out to be impossible to do! Unsolvable problems exist.

3. **Can mathematics be automatized?** Is there a computer program that would distinguish true mathematical statements (about natural numbers) from false ones?

   No, no such program can be written! Need mathematicians to do math.

4. **Hard vs. easy problems.** Some problems (like sorting n numbers) are easy to solve on a computer (in almost linear time). Other problems, e.g., scheduling classes at a university, are not known to have efficient (fast) algorithms. Complexity theory classifies problems according to the amount of computational resources (such as time and memory) that are required to solve these problems. We'll talk about this classification, with special emphasis on the "P vs. NP" problem. (The main application of complexity theory is to cryptography.)

Overall, this course is about the *power and limitations of computers*: what problems can and cannot be solved, and if they can be solved, then how fast.

We'll talk about Turing's Halting problem, Gödel's Incompleteness Theorem, and Cook-Levin's theory of NP-completeness, among other things.

3 Existence of unsolvable problems

There exists an unsolvable problem.

For now, "unsolvable" = "cannot be computed by any Java program"

By "problem" we mean a function f mapping a finite binary string to either 0 or 1 (i.e., a *decision problem*).

Claim 1. *The set of all Java programs (that is, the set of all text files of Java programs) is countable.*

Proof. Each Java program text can be converted into a finite binary string. The set of all finite binary strings is countable. It can be enumerated in the lexicographical order:

$$\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots$$

where ϵ represents the empty string.

By omitting binary sequences that do not correspond to any valid Java program, we obtain a complete enumeration of all Java programs. □

Claim 2. *The set of all computational problems (that is, functions from binary strings to 0 or 1) is uncountable.*

Proof. (by Diagonalization).

Each function f can be identified with the infinite binary string

$$f(\epsilon), f(0), f(1), f(00), f(01), f(10), f(11), f(000), \ldots$$

that is, the *truth table* of f.

Thus, it suffices to show that the set of all infinite binary strings is uncountable. Suppose, for the sake of contradiction, that one can enumerate all infinite binary sequences. Let's arrange them as rows of an infinite table: $\alpha_1, \alpha_2, \alpha_3, \ldots$, where α_i is the ith row.

We'll construct an infinite binary sequence not in this table. Hence, we contradict the assumption that the table contains all possible infinite binary sequences.

Let's define the sequence

$$\delta = \delta_1 \delta_2 \delta_3 \ldots$$

where δ_i is the ith bit of the string δ defined as follows:

$$\delta_i = \begin{cases} 0 & \text{if } (\alpha_i)_i = 1, \\ 1 & \text{if } (\alpha_i)_i = 0 \end{cases}$$

where $(\alpha_i)_i$ denotes the ith bit of the string α_i (the ith row in our table).

By definition of δ, we have δ is different from $\alpha_1, \alpha_2, \ldots$. Hence, our δ is not in the table. This proves that the set of all infinite binary sequences is not countable. □

Finally, we conclude that since the set of computational problems is much larger than the set of all Java programs, there must exist computational problems that cannot be solved by any Java program. To see this, observe that each Java program determines one particular function — namely, the function computed by this Java program. Since the set of all Java programs is countable, it

follows that the set of functions computable by Java programs is also countable! Hence, the set of Java computable functions is a strict subset of the set of all decision problems. Therefore, some decision problems exist that cannot be computed by any Java program.

This is just an "existence argument". We've argued that some uncomputable functions exist. We haven't proved yet that a specific function (like a perfect virus checker discussed above) is uncomputable. We'll get to such specific functions in the next lectures, after developing the necessary machinery.

4 Turing machine: Intuition

To argue that certain computational problems are not computable (that is, not solvable by any algorithm), one needs to have the precise notion of an algorithm. In 1936, Alan Turing defined algorithm through his Turing machine (TM).

A TM is modeled after a human computer: at each point in time, the computer

- is in one of a finite number of states of mind,
- the computer reads the contents of one page, and
- depending on the state and the contents of the page, the computer can change the contents of the page, enter a new state, and move to the next or the previous page.

5 Turing machine: Formal definition

Formalizing the intuitive definition of a human computer, we now define a *Turing machine (TM)* as a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, where

- Q is a finite set of states ("states of mind"),
- Σ is a finite input alphabet (e.g., $\Sigma = \{0, 1\}$),
- Γ is a finite tape alphabet, containing Σ as well as the special "blank symbol",
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function (where R and L mean "move right" and "move left" , respectively),
- q_0 is the initial state,
- q_{accept} is the accepting state, and
- q_{reject} is the rejecting state.

Initially, the TM is scanning the left-most symbol of the input string (over the alphabet Σ), in state q_0. The tape is infinite in both directions[1], with all tape cells beyond the input string containing blanks. At each step, the TM applies the appropriate instruction from the transition

[1]More precisely, we should think of the tape as *finite but extendable*: we can always go out and buy more memory for our computer.

function δ. For example, if we scan a in state q, and $\delta(q, a) = (p, b, R)$, then the TM replaces a with b, enters the new state p, and moves its tape head one position to the right.

The TM may run forever, or may terminate. If it terminates in state q_{accept}, we say that it accepts its input. If it terminates in q_{reject}, we say it rejects its input.

6 "Church-Turing Thesis"

Turing gave a very convincing argument that a human computer (performing symbolic manipulations with pen and paper) can be simulated by an appropriate Turing machine. Clearly, every Turing machine can be simulated by a human (who will just follow the program of the Turing machine). Thus, we have an equivalence between the intuitive notion of an algorithm (of the symbolic-manipulation kind, as discussed above) and the formal notion of a Turing machine. This equivalence is usually called the "Church-Turing thesis"[2].

[2]although some computability theorists object to the word "thesis" since they believe this correspondence between algorithmically solvable and Turing machine solvable to be a true fact which was convincingly argued by Turing in his original 1936 paper