CS 3719 (Theory of Computation and Algorithms) – Lecture 5

Antonina Kolokolova*

January 18, 2013

1 Undecidable languages

1.1 Church-Turing thesis

Let's recap how it all started. In 1990, Hilbert stated a list of problems for mathematicians of the next century; some of these problems asked to "devise a procedure"; two of those problems are "devise a procedure for solving an equation over integers (Diophantine equations)" that you have seen in the first lecture, and "devise a procedure that, given a statement of mathematics, would decide if it is true or false".

Alan Turing was working on Hilbert's problem that asked for an algorithm that for any statement of mathematics would state whether it is true or false; Gödel has shown (his famous Incompleteness Theorem) that there are statements of mathematics for which such answer cannot be given, but it remained open at that time whether there is such a procedure for statements for which that answer could be given. There were several mathematicians working on this problem at that time; notably, Alonco Church solved this problem (to give a negative answer) at about the same time, by inventing lambda-calculus. Turing's approach is somewhat more computational: he defined a model of computation which we now call the Turing machine, equivalent to Church's model in terms of power, and used it to show undecidability results, thus giving a negative answer to Hilbert's problem.

Definition 13 (Church-Turing thesis). *Anything computable by an algorithm of any kind (our intuitive notion of algorithm) is computable by a Turing machine.*

Since this statement talks about an intuitive notion of algorithm we cannot really prove it; all we can do is that whenever we think of a natural notion of an algorithm, show that this can be done by a Turing machine.

*The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364. Many thanks to Richard Bajona for taking notes!

In this lecture we will show that even though Turing machines are considered to be as powerful as any algorithm we can think of, there are languages that are not computable by Turing machines. Thus, for these languages, it is likely that no algorithm we can think of would work.

We will present two proofs of existence of undecidable languages. The first proof is non-constructive, using Cantor's diagonalization. The second proof presents an actual language that is undecidable.

1.2 Diagonalization

The Diagonalization method is used to prove that two (infinite) sets have different cardinalities, that is, a set A is larger than the set B. By definition of cardinalities, this means that there is no one-to-one correspondence between elements of the two set, so the elements of A cannot be "enumerated" by elements of B. The proof is by contradiction: assume that there is such a enumeration. Then, construct an element of A which is not in the list. In our case, the larger set A will be the set of all languages (for simplicity, over $\Sigma = \{0, 1\}$, but any alphabet with at least 2 symbols will work). And B will be the set of all Turing machines.

First, let us say how we describe languages. Recall that a *characteristic string* of a set is an infinite string of 0s and 1s where, for a given order (usually lexicographic order) of elements in the set there is a 0 in i^{th} position in the string if i^{th} element in the order is not in the set and 1 if it is in the set. For example, for a set $L = \{1, 01\}$ over $\{0, 1\}^*$ the characteristic string would be 00101000...00..., since out of the lexicographic ordering $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$ of $\{0, 1\}^*$ only the 3rd and 5th elements are in L. Thus, for every language over $\{0, 1\}^*$ (or any alphabet with at least 2 elements) there is a (unique) characteristic string describing this language.

Now, we need to describe Turing machines and state how to enumerate them (show that the set of all Turing machines is countable). For that, we show that every Turing machine can be encoded by a distinct finite binary string (and is thus a subset of all finite binary strings, which is countable since every string can be treated as a binary number with the leading 1 missing).

To encode a Turing machine, it is sufficient to write, in binary, the tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$. However, we are not interested in specific names of symbols in Q, Σ, Γ; we are just interested in how many symbols are in each. A Turing machine which accepts all even-length strings over $\{a, b\}$ operates exactly the same way as a Turing machine accepting all even-length strings over $\{0, 1\}$, with a changed to 0 and b changed to 1 everywhere in its description. We assign an order to elements of Q, Σ and Γ, and refer to elements as 1st symbol of Σ, 5th state in Q and so on. So all we need to write is the number of states in Q (for simplicity, we can even rename states to have q_0 be the first state in the list, q_{accept} second and q_{reject} third, since every Turing machine will have these three states), number of symbols in Σ and Γ (say Γ consists of Σ followed by $\sqcup$ followed by possible extra symbols). Finally, we need to write out δ, using the indices of symbols in Q, Σ and Γ in the description of transitions.

Here is one way of doing the description. We can start by writing $|Q|$ 1s, then a 0, then $|\Sigma|$ 1s, then another 0, then $|\Gamma|$ 1s, and a 0 again. We could also write all elements of δ in unary (since it is finite), but we can also do so in binary, by introducing a special "separator" symbol , into Σ. If we have to stick to $\Sigma = \{0, 1\}$, then we can still write δ in binary as follows: associate "11" with ,, "00" with 0 and "01" with 1. For example, say we want to encode a transition $(q_3, a) \to (q_4, b, L)$. With separators, it can be coded in binary as $11, 0, 100, 1, 0$ (here, code L by 0 and R by 1). Using the transformation to encode the transition in binary obtain 0101110011010000011011100. Note that this method of coding allows us to talk about all sorts of objects as an input to a Turing machine, be it Java code or descriptions of graphs.

Notation 1. We will use the notation $\langle M \rangle$ to mean a binary string encoding of a Turing machine M. We can use the same notation to talk about encodings of other objects, e.g. $\langle M, w \rangle$ encodes a pair Turing machine M and a string w; $\langle N \rangle$ encoding a NFA N, $\langle G \rangle$ for a graph G and so on,

Now, notice that for every Turing machine there is a finite binary description. Treating this description as a binary number, obtain an enumeration (by a subset of $\mathbb{N}$ of all Turing machines. Finally, we can do the diagonalization argument. Start by assuming that it is possible to enumerate all languages by Turing machines. Write elements of characteristic strings as columns, and Turing machine descriptions as rows. Put a 1 in cell (i, j) if the i^{th} Turing machine M_i accepts string number j in the enumeration, and 0 if it does not accept this string. We obtain a table as in the following example (for different enumerations of Turing machines the 0s and 1s would be different), and use diagonalization argument to construct a language not recognized by any Turing machine. Indeed, if that language were recognized by some Turing machine, say M_k, it would be the string in the k^{th} row of the table; however, it differs from the diagonal language in k^{th} element.

M_1	**0**	0	1	1	0	1	1	0	1	....
M_2	1	**1**	1	1	1	0	0	1	1	....
M_3	1	0	**0**	0	0	1	1	1	1	....
M_4	1	1	0	**1**	1	0	0	1	1	....
M_5	0	0	1	1	**1**	1	1	0	0	....
ots										
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
D	**1**	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	....

1.3 Universal Turing machine and undecidability of A_{TM}

In this section we will present a specific, very natural problem and show that it is undecidable. It will lead us to a whole class of problems of similar complexity.

Definition 14. *The language* $A_{TM} = \{\langle M, w\rangle | M$ *is a Turing machine and* w *is a string over the input alphabet of* M *and* M *accepts* $w\}$

That is, the language A_{TM} consists of all pairs M, w of Turing machine + a string in $\mathcal{L}(M)$.

Theorem 15. A_{TM} *is semi-decidable, but not decidable.*

Proof. Let us first show that A_{TM} is semi-decidable. That is, there exists a Turing machine $M_{A_{TM}}$ accepting all and only strings in A_{TM}. Note that if M does not halt on w, neither does $M_{A_{TM}}$ on $\langle M, w\rangle$

$M_{A_{TM}}$: On input $\langle M, w\rangle$
Simulate M on w. If M accepts w, accept. If M rejects w, reject.

Note that the above algorithm is essentially an interpreter; that is a program which takes as input both a program P and an input w to that program, and simulates P on input w. In this case the program P is given by a Turing machine M. In particular, the Turing machine "interpreter" $M_{A_{TM}}$ is known under the name of a *Universal Turing Machine.* Turing described a universal Turing machine in some detail in his original 1936 paper, an ideal which paved the way for later interpreters operating on real computers. This is quite a meta-mathematical concept, though: a single Turing machine, and a simple one at that, that could "do the job" of any other Turing machine provided it is given the description of the TM it is supposed to simulate and a string to work on.

Now, let us show that A_{TM} is not decidable. Assume for the sake of contradiction that it is, so there is a Turing machine H that takes as an input $\langle M, w\rangle$ and halts either accepting (if M accepted w) or rejecting (if M did not accept w). Now, define the following language:

$Diag = \{\langle M\rangle | M$ is a Turing machine and $\langle M\rangle \notin \mathcal{L}(M)\}$.

That is, *Diag* is a language of all descriptions of Turing machines that do not accept a string that is their own encoding. This is exactly the diagonal language from our diagonalization table.

Now, notice that H deciding A_{TM} can also be used to decide *Diag*: $H(\langle(M, \langle M\rangle)\rangle)$ halts and accepts if M accepts its own encoding and rejects if M does not accept its own encoding. A decider H_{Diag} for *Diag* would run $H(\langle(M, \langle M\rangle)\rangle)$ and accept if H rejects, reject if H accepts. But what should it do on input $\langle H_{Diag}\rangle$? It cannot accept this input, since that would mean that H_{Diag} accepts its own encoding, so it should not be in *Diag*. And it cannot reject its own encoding, since it would make it a Turing machine not accepting its own encoding and thus it has to be in *Diag*. Contradiction.

This contradiction is akin to Russell's paradox from logic, and other self-referential paradoxes of the form "I am lying".

□