

# COMP 6902 (Theory of Computation) – Computability and undecidability

Antonina Kolokolova\*

May 16, 2018

## 1 Computability

A Turing machine  $M$  *recognizes* a language  $L$  if it accepts all and only strings in  $L$ : that is,  $\forall x \in \Sigma^*$ ,  $M$  accepts  $x$  iff  $x \in L$ . As before, we write  $\mathcal{L}(M)$  for the language accepted by  $M$ .

**Definition 13.** A language  $L$  is called *semi-decidable* (also called *recursively enumerable*, r.e. or *Turing-recognizable*) if  $\exists$  a Turing machine  $M$  such that  $\mathcal{L}(M) = L$ .

A language  $L$  is called *decidable* (or *recursive*) if  $\exists$  a Turing machine  $M$  such that  $\mathcal{L}(M) = L$ , and additionally,  $M$  halts on all inputs  $x \in \Sigma^*$ . That is, for every input string  $x \in \Sigma^*$  on the tape at the start of the computation,  $M$  either enters the state  $q_{\text{accept}}$  or  $q_{\text{reject}}$  in some point in computation.

It is possible to define a Turing machine producing an output; in that case, the Turing machine halts in an accepts state, with the tape clear except for the output and head pointing to the first symbol of the output.

**Definition 14.** A function  $f: \Sigma_1^* \rightarrow \Sigma_2^*$  is *computable* if there is a Turing machine  $M$  that halts on every input  $x \in \Sigma_1^*$  with  $f(x)$  as its output on the tape.

One reason that these languages are called “recursively enumerable” is that it is possible to “enumerate” all strings in such a language by a special type of Turing machine, enumerator. This Turing machine starts on blank input and runs forever; as it runs, it outputs (e.g., on some additional tape or part of the main tape) all the strings in the language. It is allowed to print the same string multiple times, as long as it does not print anything not in the

---

\*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!

language and eventually print every string that is in the language. See Sipser's book for a formal definition and a proof of equivalence.

There is yet another definition of the class of semi-decidable languages, which is quite useful for some of the proofs. Note that to check that a Turing machine  $M$  accepted a string  $x$  it is enough to look at a run of  $M$  with input  $x$ , and confirm that it stops in an accept state after a finite number of steps when started with  $x$  as an input. However, if a machine  $M$  does not accept  $x$ , no such run exists. More generally, we can equivalently define the class of semi-decidable languages as follows.

**Theorem 15.** *A language is  $L$  semi-decidable if there exists a Turing machine  $V_L$  (where  $V$  stands for "verifier") halting on every input such that for all  $x \in \Sigma^*$ ,  $x \in L$  if and only if there exists a finite string  $y$  (over a possibly different finite alphabet) such that  $\langle x, y \rangle \in \mathcal{L}(V_L)$ . We call  $y$  a proof (or certificate) of  $x$  being in  $L$ .*

To see why every language that has such verifier is semi-decidable, consider a Turing machine  $M_L$  which, starting with an input  $x$ , tries all possible  $y$  starting from the empty string  $\lambda$  and runs the verifier on  $\langle x, y \rangle$  for each  $y$ . If for some  $y$  the pair  $\langle x, y \rangle$  is accepted by the verifier, then  $M_L$  accepts. Since each run of the verifier takes a finite amount of time,  $M_L$  never "gets stuck" on any of the  $y$ s. Otherwise, it runs forever, trying longer and longer potential proofs. Thus,  $\mathcal{L}(M_L) = L$ .

For the other direction of the equivalence, suppose that  $L$  is semi-decidable by some Turing machine  $M_L$ . We want to show that execution of  $M_L$  with an input  $x$  can be encoded by a finite string  $y$ , which can serve as an easily verifiable proof when  $M_L$  accepts  $x$ . Here is one possible way to describe such a proof, using the terminology of a "configuration" of a Turing machine.

Recall that at any step of the computation of a Turing machine its subsequent behaviour depends only on its state, content of its tape (where non-blank part is always finite) and a position of its head. For compactness, let's write a given configuration as a string encoding non-blank part of the tape (more specifically, the part between the start of the tape and last non-blank symbol, if any), with an additional symbol denoting the state inserted before (to the left) of the symbol being read. For example, if the tape contains string 010 surrounded by blanks, and the Turing machine is in a state  $q_7$  reading the second symbol (the 1), we can encode it as a finite string  $0q_710$ . Now, an accepting (or rejecting) computation of a Turing machine can be written as a sequence of its configurations, starting with  $q_0x_1 \dots x_n$  for the input  $x = x_1 \dots x_n$ , and ending in a string containing  $q_a$  (respectively,  $q_r$ ). Also, it is very easy to check that each subsequent configuration was obtained from a previous by a valid transition of this Turing machine: just check that the configuration is essentially the same, except the character under the head and the state might have changed, with the head moving left or right (that is, state character swapping places with a symbol to the left, or a (potentially new) symbol to the right).

Thus, verifying that a given string  $y$  encodes a correct computation of a given Turing machine is a decidable problem. So if a decider  $V_L$  (which already knows  $M_L$ ) gets  $x$  and then a  $y$  encoding a computation of  $M_L$  on  $x$  as above, it can quickly verify that the sequence starts with  $q_0x_1 \dots x_n$ , ends with a string containing  $q_a$ , and all steps are valid. This completes the proof that for any semi-decidable  $L$  there exists a verifier  $V_L$  as above.

## 1.1 Church-Turing thesis

Let's recap how it all started. In 1900, 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, Alonzo 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 15** (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.

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 sets, so the elements of  $A$  cannot be “enumerated” by elements of  $B$ . The proof is by contradiction: assume that there is such an 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  $i^{\text{th}}$  position in the string if  $i^{\text{th}}$  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). Rather than giving a full proof here, we will refer to the intuition that any “piece of code” (including a Turing machine) becomes a different string of 0s and 1s in a computer memory; whichever encoding achieves that, suffices for our purposes.

**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  $w$ ;  $\langle D \rangle$  encoding a finite automaton  $D$ ,  $\langle G \rangle$  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  $i^{\text{th}}$  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  $k^{\text{th}}$  row of the table; however, it differs from the diagonal language in  $k^{\text{th}}$  element.

	$\lambda$	0	1	00	01	10	11	000	001	...
$M_1$	<b>0</b>	0	1	1	0	1	1	0	1	....
$M_2$	1	<b>1</b>	1	1	1	0	0	1	1	....
$M_3$	1	0	<b>0</b>	0	0	1	1	1	1	....
$M_4$	1	1	0	<b>1</b>	1	0	0	1	1	....
$M_5$	0	0	1	1	<b>1</b>	1	1	0	0	....
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$D$	<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>0</b>	<b>1</b>	....

## 2 Undecidable languages

### 2.1 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 16.** *The language  $A_{TM} = \{\langle M, w \rangle \mid M \text{ is a Turing machine and } w \text{ is a string over the input alphabet of } M \text{ and } M \text{ accepts } w\}$*

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

**Theorem 16.**  *$A_{TM}$  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:

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

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  $Diag$ :  $H(\langle \langle M, \langle M \rangle \rangle \rangle)$  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  $H(\langle \langle M, \langle M \rangle \rangle \rangle)$  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".

□

## 2.2 Beyond semi-decidable

Suppose you are given two languages,  $L_1$  and  $L_2$ . What can you say about a language  $L_1 \cup L_2 = \{x \mid x \in L_1 \text{ or } x \in L_2\}$ ? For example, suppose you are interested in all strings either code Turing machines or are prime numbers when viewed as a binary number; in this case, your  $L_1$  could be all encodings of Turing machines and  $L_2$  all prime numbers. Similarly, you may want to ask about  $L_1 \cap L_2$  which contains strings which are in both languages (that is, encodings of TMs which are primes when viewed as binary numbers, in our example.) Suppose you know that  $L_1$  and  $L_2$  are both decidable, or both semi-decidable – what does it tell you about decidability (semi-decidability) of their union and intersection?

**Theorem 17.** *The class of semi-decidable languages is closed under union and intersection operations.*

*Proof.* Let  $L_1$  and  $L_2$  be two semi-decidable languages, and let  $M_1, M_2$  be Turing machines such that  $\mathcal{L}(M_1) = L_1$  and  $\mathcal{L}(M_2) = L_2$ . We will construct Turing machines  $M_{L_1 \cup L_2}$  and  $M_{L_1 \cap L_2}$  accepting union and intersection of  $L_1$  and  $L_2$ , respectively.

Consider the union operation first; intersection will be similar. Let  $x$  be the input for which we are trying to decide whether it is in  $L_1 \cup L_2$ . The first idea could be to try to run  $M_1$

on  $x$ , and if it does not accept, then run  $M_2$  on  $x$ . But  $M_1$  is not guaranteed to stop on  $x$ , and we would still like to accept  $x$  if  $M_2$  accepts it. So the solution is to run  $M_1$  and  $M_2$  in parallel, switching between executing one or the other. If at some point in the computation either  $M_1$  or  $M_2$  accepts, we accept; if neither accepts, can run forever – but this is OK, because if neither  $M_1$  nor  $M_2$  accepts  $x$  then  $x \notin L_1 \cup L_2$ . So we define  $M_{L_1 \cup L_2}$  as follows:

$M_{L_1 \cup L_2}$  : On input  $x$   
 For  $i = 1$  to  $\infty$   
     Run  $M_1$  on  $x$  for  $i$  steps. If  $M_1$  accepts, accept.  
     Run  $M_2$  on  $x$  for  $i$  steps. If  $M_2$  accepts, accept.

The intersection, in this case, is very similar. The only difference is that we accept at stage  $i$  if not just one, but both  $M_1$  and  $M_2$  accepted in  $i$  steps.

□

**Corollary 18.**  $\overline{A_{TM}}$  is not semi-decidable. Moreover, complement of any semi-decidable, but undecidable language is not semi-decidable.

*Proof.* Otherwise, running Turing machines  $M_{A_{TM}}$  and  $M_{\overline{A_{TM}}}$  simultaneously, as in the proof above, we could decide  $A_{TM}$ . Same holds for any semi-decidable, but undecidable language.

□

This shows that the class of semi-decidable languages is different (incomparable) from the class of languages which are complements of semi-decidable ones. Also, there are languages that are neither. For example, consider a simple language  $0 - 1A_{tm} = \{\langle M, w \rangle \mid \text{TM } M \text{ accepts } 01 \text{ and loops on } 1w\}$ .

Intuitively, testing if  $\langle M, w \rangle$  is in the language requires solving an  $A_{TM}$  problem and a  $\overline{A_{TM}}$  problem. The second makes it not semi-decidable, and the first makes its complement not semi-decidable.

A convenient way to talk about complexity of such languages is to look at the quantifiers present in the language definition, and then use an extension of the verifier characterization of semi-decidable. First, recall that negation of an existential quantifier is a universal quantifier; so if a language is defined as "  $x$  for which there exists  $y$  such that a condition  $V(x, y)$  holds" (e.g., "there exists an accepting computation of  $M$  on  $x$ "), then its complement consists of strings that are not in a valid form (e.g., for  $A_{TM}$ , strings that are not  $\langle M, w \rangle$  for any  $M, w$ ), together with strings  $x$  in valid form for which condition does not hold for any  $y$  (e.g, there does not exist an accepting computation  $y$  of  $M$  on  $w$ ). Moving the negation inside the quantifier, get that if  $L = \{x \mid \exists y V(x, y)\}$ , then  $\bar{L} = \{x \mid \forall y \neg V(x, y)\}$ . Similarly, the language

$0 - 1A_{tm}$  above can be written as  $\{\langle M, w \rangle \mid \exists y_1 \text{ such that } y_1 \text{ is an accepting computation of } M \text{ on } 01 \text{ and } \forall y_2, y_2 \text{ is not a halting computation of } M \text{ on } 1w\}$ .

**Definition 17.** A language  $L$  is called *co-semi-decidable* if its complement  $\bar{L}$  is semi-decidable, or, equivalently, if there is a verifier  $V_L$  which halts on all inputs and such that for every  $x$ ,  $x \in L$  if and only if  $\forall y$ ,  $V_L$  accepts  $\langle x, y \rangle$ . Generally, languages that can be represented by a definition with  $k$  alternations of quantifiers are said to be at the  $k^{\text{th}}$  level of the arithmetical hierarchy. If the definition starts with  $\exists$ , we call that level  $\Sigma_k$  (not to be confused with  $\Sigma$  as an alphabet), and if it starts with  $\forall$ , then  $\Pi_k$ .

For example, a language  $\text{Empty} = \{\langle M \rangle \mid M \text{ is a Turing Machine and } \mathcal{L}(M) = \emptyset\}$  is co-semi-decidable:  $\text{Empty} = \{\langle M \rangle \mid \forall w \forall s \text{ } s \text{ is not an accepting computation of } M \text{ on } w\}$ . And a language  $\text{All} = \{\langle M \rangle \mid M \text{ is a Turing Machine that accepts every input}\}$  is in the level  $\Pi_2$  of the arithmetic hierarchy: it can be written as  $\text{All} = \{\langle M \rangle \mid \forall w \exists s \text{ such that } s \text{ is an accepting computation of } M \text{ on } w\}$ .

### 3 Reductions

Now we will proceed to show that many problems are undecidable (and some of them are not semi-decidable, not co-semi-decidable or even neither semi- nor co-semi-decidable). Rather than adapting the proof of undecidability of  $A_{TM}$  to other problems, we will use a concept which is going to be used a lot for the rest of this course: the notion of a *reduction*. A reduction is a method of “disguising” one problem as another, so if we can solve the disguised one it can give us the solution to the original. This method is very useful for proving that problems are hard: if you can disguise a hard problem as one in hand, then solving this problem is at least as hard as solving the hard one.

**Definition 18.** Let  $L_1, L_2 \subseteq \Sigma^*$ . We say that  $L_1 \leq_m L_2$  if there is a computable function  $f : \Sigma^* \rightarrow \Sigma^*$  such that for all  $x \in \Sigma^*$ ,  $x \in L_1 \Leftrightarrow f(x) \in L_2$ .

Here, we need  $f$  to be computable so that it always gives us an answer. The notation  $\leq_m$  stands for “many-one reduction” or “mapping reduction”. It is many-one since  $f$  may map many different instances of a problem to a single output.

**Theorem 19.** Let  $L_1, L_2 \subseteq \Sigma^*$  such that  $L_1 \leq_m L_2$ . Then

- 1)  $\bar{L}_1 \leq_m \bar{L}_2$
- 2) If  $L_2$  is decidable then  $L_1$  is decidable.  
(And hence, if  $L_1$  is not decidable then  $L_2$  is not decidable either).



- 3) If  $L_2$  is semi-decidable then  $L_1$  is semi-decidable.  
 (And hence, if  $L_1$  is not semi-decidable then neither is  $L_2$ .)

*Proof.* **1)** Say that  $L_1 \leq_m L_2$  via the computable function  $f$ . Then we also have  $\overline{L_1} \leq_m \overline{L_2}$  via  $f$ , since  $x \in L_1 \Leftrightarrow f(x) \in L_2$  implies that  $x \in \overline{L_1} \Leftrightarrow f(x) \in \overline{L_2}$ .

**2)** Say that  $L_2 = \mathcal{L}(M_2)$  where  $M_2$  is a Turing machine that halts on every input. Let  $M$  be a Turing machine that computes  $f$ . We now define Turing machine  $M_1$  as follows. On input  $x$ ,  $M_1$  runs  $M$  on  $x$  to get  $f(x)$ , and then runs  $M_2$  on  $f(x)$ , accepting or rejecting as  $M_2$  does. Clearly  $M_1$  halts on every input, and  $L_1 = \mathcal{L}(M_1)$ , so  $L_1$  is decidable.

**3)** Say that  $L_2 = \mathcal{L}(M_2)$  where  $M_2$  is a Turing machine. Let  $M$  be a Turing machine that computes  $f$ . We now define Turing machine  $M_1$  as follows. On input  $x$ ,  $M_1$  runs  $M$  on  $x$  to get  $f(x)$ , and then runs  $M_2$  on  $f(x)$ , accepting or rejecting as  $M_2$  does if and when  $M_2$  halts. Clearly  $L_1 = \mathcal{L}(M_1)$ , so  $L_1$  is semi-decidable.  $\square$

Question: is it true that  $A_{TM} \leq_m \overline{A_{TM}}$ ? The answer is No: if it were true, then by the first property above we would have  $\overline{A_{TM}} \leq_m A_{TM}$ . But by 3) above, that would mean that  $\overline{A_{TM}}$  is semi-decidable. But if both a language and its complement are semi-decidable, then, as we saw in the last class, the language would have to be decidable – which is a contradiction, since  $A_{TM}$  is undecidable. Thus,  $A_{TM}$  is not reducible to  $\overline{A_{TM}}$ , and in fact, it is not reducible to any co-semi-decidable language. So please don't make a mistake of assuming that you always prove undecidability by reducing  $A_{TM}$  to a problem in hand – sometimes you have to use  $\overline{A_{TM}}$ , if the problem you are working with is co-semi-decidable. Or, conceptually easier, if the problem in hand is co-semi-decidable, then work with its complement all the way.

Now we can use this notion of reduction to prove that some languages are undecidable by reducing languages for which we already know that (such as  $A_{TM}$ ) to them.

**Example 1.** Let  $\text{HaltB} = \{\langle M \rangle \mid \text{TM } M \text{ halts on blank input}\}$ . We will show that  $A_{TM} \leq_m \text{HaltB}$

Let  $x \in \Sigma^*$ , and assume that  $x = \langle M, w \rangle$  where  $M$  is a Turing Machine.

(If  $x$  is not of this form, then we can let  $f(x)$  be anything not in  $\text{HaltB}$ . In general we will assume that the input is “well-formed” since this will always be easy to test for.)

We will let  $f(x) = \langle M' \rangle$  where  $M'$  works as follows on a blank tape (we don't care what  $M'$  does on a non-blank tape).

$M'$  : on input  $x'$   
 write  $w$  on the tape  
 simulate  $M$  running on input  $w$ ;  
 if and when  $M$  halts and accepts,  $M'$  halts and accepts;  
 if and when  $M$  halts and rejects,  $M'$  goes into an infinite loop.

Clearly  $f$  is computable. It is also easy to see that  $x \in A_{TM} \Leftrightarrow f(x) \in \text{HaltB}$ , since  $M$  accepts  $w \Leftrightarrow M'$  halts on blank tape.

**Corollary 20.**  $\overline{\text{HaltB}}$  is not semi-decidable.

*Proof.* We know  $\text{HaltB}$  is semi-decidable but not decidable, so  $\overline{\text{HaltB}}$  is not semi-decidable.  $\square$

**Example 2.** Let  $\text{NE}$  be the language consisting of Turing Machines that accept a nonempty language. That is,  $\text{NE} = \{\langle M \rangle \mid M \text{ is a Turing Machine and } \mathcal{L}(M) \neq \emptyset\}$ .

**Lemma 21.**  $\text{NE}$  is semi-decidable, but not decidable. (Hence,  $\overline{\text{NE}}$  is not semi-decidable.)

*Proof.* We will design a Turing Machine  $M_{\text{NE}}$  such that  $\text{NE} = \mathcal{L}(M_{\text{NE}})$   $M_{\text{NE}}$  behaves as follows:

$M_{\text{NE}}$ : on input  $x$   
 if  $x$  is not of the form  $\langle M \rangle$ , reject  
 for  $i = 1$  to  $\infty$   
     run  $M$  on all inputs of length  $\leq i$  for  $i$  steps;  
     if and when it is discovered that  $M$  accepts some input,  $M_{\text{NE}}$  halts and accepts.

That is,  $M_{\text{NE}}$  does the following (let  $\Sigma = \{0, 1\}$ ): run  $M$  on all inputs of length  $\leq 1$  for 1 steps; if  $M$  accepted  $\epsilon$  or 0 or 1 in one transition then accept; otherwise run  $M$  on all inputs of length  $\leq 2$  for 2 steps; if  $M$  accepted one of  $\epsilon, 0, 1, 00, 01, 10, 11$  then accept; otherwise run  $M$  on all inputs of length  $\leq 3$  for 3 steps; etc.

Clearly  $\text{NE} = \mathcal{L}(M_{\text{NE}})$ , so  $\text{NE}$  is semi-decidable.

To show that  $\text{NE}$  is not decidable we will prove that  $\text{HaltB} \leq_m \text{NE}$ .

Let  $x$  be an input for  $\text{HaltB}$ , and assume that  $x$  is well-formed, that is,  $x = \langle M \rangle$ .

Define  $f(x) = \langle M' \rangle$  where  $M'$  works as follows.

$M'$  on input  $x$   
 erase  $x$  and run  $M$  on the blank tape;  
 if and when  $M$  halts,  $M'$  halts and accepts.

Clearly  $\langle M \rangle \in \text{HaltB} \Leftrightarrow \langle M' \rangle \in \text{NE}$ , so  $\text{HaltB} \leq_m \text{NE}$ , so  $\text{NE}$  is not decidable.

This type of reduction, doing the same on all inputs can help with a surprising number of problems. I like to call them “All-or-nothing” reductions. The resulting language is either  $\Sigma^*$  (and thus includes every subset one might be interested in) or  $\emptyset$ .  $\square$

You might have noticed that it is possible to reduce  $A_{TM}$  to  $HaltB$  and  $NE$  as well as the other way around: thus, the complexity of these three languages is the same. This leads us to the following definition:

**Definition 19.** A language  $L$  is hard for a class of languages  $\mathcal{C}$  (under a given type of reductions) if for every  $L' \in \mathcal{C}$ ,  $L'$  reduces to  $L$ . A language  $L$  is complete for  $\mathcal{C}$  if additionally  $L \in \mathcal{C}$ .

**Theorem 22.**  $A_{TM}$  is complete for the class of semi-decidable languages.

We have showed that  $A_{TM}$  is semi-decidable, so it is in the class. To show that every semi-decidable language reduces to  $A_{TM}$ , use the universal Turing machine.

### 3.1 More examples of reductions

Recall that  $A \leq_m B$  iff there exists a computable function  $f$  such that  $\forall x \in \Sigma_A^* x \in A$  iff  $f(x) \in B$ . The notation suggests that “A is at most as hard to solve as B”. Often we use the reduction to prove hardness for problems of comparable complexity, but sometimes it is not the case:  $A$  can be a lot simpler than  $B$ .

**Example 3.**  $\{uu|u \in \{0,1\}^*\} \leq_m A_{TM}$

We need to define  $f(x) = \langle M, w \rangle$ . Take a Turing machine, say, which accepts an empty string and rejects everything else. So the description of  $M$  is simple:  $Q = \{q_0, q_{accept}, q_{reject}\}$ ,  $\Sigma = \{0, 1\}$ ,  $\Gamma = \{0, 1, \sqcup\}$   $\delta = \{(q_0, 0) \rightarrow (q_{reject}, 0, R), (q_0, 1) \rightarrow (q_{reject}, 0, R), (q_0, \sqcup) \rightarrow (q_{accept}, 0, R)\}$ . Now, define  $f(x) = \langle M, \epsilon \rangle$  if  $x = uu$  for some  $u \in \{0, 1\}^*$  and  $f(x) = \langle M, 0 \rangle$  otherwise. This is a computable function, and it has the desired property that  $x \in \{uu|u \in \{0, 1\}^*\} \iff f(x) \in A_{TM}$ .

Similarly, for the rest of this lecture we will use reductions to show that certain problems are even harder than ones we encountered so far. Recall that semi-decidable languages are ones for which there is a Turing machine which halts (and accepts) on all strings in the language; for co-semi-decidable languages, there is a Turing machine halting on all inputs not in the language. However, there are some languages which are neither semi-decidable nor co-semi-decidable, but belong higher in arithmetic hierarchy, as it is called. The best way to think about them is using quantifiers: semi-decidable languages correspond to an existential quantifier (or several existential quantifiers) over a decidable predicate (e.g., “exists a string on which there exists an accepting computation” – here, the decidable predicate is the check that the existential quantifiers indeed guessed a string and a correct computation of this Turing machine on this string ending in an accept state). Similarly, a co-semi-decidable language can be described using a universal quantifier, just by negating a formula describing the language (e.g. “for any string any computation (finite and correct) is not accepting”). The languages beyond semi-decidable and co-semi-decidable are, thus, described using a

combination of quantifiers. The number of quantifier alternations corresponds to the levels of this (strict) hierarchy.

In this lecture we will only talk about languages described with just one quantifier alternation. But already in this case we cannot talk at all about a Turing machine corresponding to this language (or its complement).

**Example 4.** Let  $L_{01} = \{\langle M \rangle \mid M \text{ loops on } 0 \text{ and accepts } 1\}$ . Note that the description of this language has both a universal quantifier ("loop" = "all finite computations are wrong") and an existential quantifier ("accept" = "exists correct accepting computation").

To prove that this language is neither semi-decidable nor co-semi-decidable we will use two reductions. Let us use  $A_{TM}$  as the "hard problem". To show that  $L_{01}$  is not co-semi-decidable, we will reduce a not co-semi-decidable  $A_{TM}$  to  $L_{01}$ . Then, to show that  $L_{01}$  is not semi-decidable we will reduce a non-semi-decidable  $\overline{A_{TM}}$  to  $L_{01}$ .

First we will show that  $L_{01}$  is not co-semi-decidable by showing  $A_{TM} \leq_m L_{01}$ . For that, by definition of reduction, we will describe a computable function  $f(\langle M, w \rangle) = \langle M' \rangle$  that for any pair  $M, w$  constructs  $M'$  such that  $M$  accepts  $w$  if and only if  $M'$  loops on 0 and accepts 1. Since we are reducing a semi-decidable language (existential quantifier), we will force  $M'$  to always loop on 0, and will make its behaviour on 1 depend on whether  $M$  accepts  $w$ .

$M'$ : on input  $x$

for  $x \neq 0, x \neq 1$  it does not matter what  $M'$  does.

It could accept, or run  $M$  on  $w$ , anything. Say  $M'$  rejects.

if  $x = 0$  then loop

if  $x = 1$  then run  $M$  on  $w$ .

if  $M$  accepts, accept. If  $M$  rejects, loop (here, reject would also be correct).

Thus,  $M'$  always loops on 0, and accepts 1 if and only if  $M$  accepts  $w$ , just as we wanted. This reduction shows that  $L_{01}$  is at least as hard as  $A_{TM}$ , in particular, since  $A_{TM}$  is not co-semi-decidable, then neither is  $L_{01}$ .

It remains to show that  $L_{01}$  is not semi-decidable. We will do it by reduction  $\overline{A_{TM}} \leq_m L_{01}$ . That is, we will construct a computable function  $f$  which on input  $\langle M, w \rangle$  produces  $M'$  which now will loop on 0 and accept 1 if and only if  $M$  does not accept  $w$ . Another technicality is that since we are talking about complement of  $A_{TM}$ , the language we are reducing from contains all the "garbage" – strings that do not encode Turing machines. That is,  $\overline{A_{TM}} = \{s \mid s \neq \langle M, w \rangle \text{ or } s = \langle M, w \rangle \text{ and } M \text{ does not accept } w\}$ . So  $f(s)$ , for  $s \neq \langle M, w \rangle$  would output something in  $L_{01}$ : for example a description of a Turing machine with transitions  $(q_0, 1) \rightarrow (q_{accept})$  and  $(q_0, 0) \rightarrow (q_{loop}, 0, R)$  where all transitions from  $q_{loop}$  go to  $q_{loop}$ . This machine accepts 1 and loops on 0.

$M'$ : on input  $x$

for  $x \neq 0, x \neq 1$  it does not matter what  $M'$  does.

It could accept, or run  $M$  on  $w$ , anything. Say  $M'$  rejects.

if  $x = 1$  then accept

if  $x = 0$  then run  $M$  on  $w$ .

if  $M$  accepts, accept (reject is OK, just don't loop). If  $M$  rejects, loop (here, it has to be loop).

**Example 5.** Let  $T$  be the language of “total” machines, that is, of machines that halt on every input.  $T = \{\langle M \rangle \mid M \text{ is a Turing Machine that halts on every input}\}$ .

**Lemma 23.** *Neither  $T$  nor  $\overline{T}$  is semi-decidable.*

*Proof.* We first show that  $\text{HaltB} \leq_m T$ , implying that  $\overline{\text{HaltB}} \leq_m \overline{T}$ , implying that  $\overline{T}$  is not semi-decidable.

Let  $f(\langle M \rangle) = \langle M' \rangle$  (and if input to  $f$  is not a proper encoding of a Turing machine, it is an  $M'$  that just loops on every input). We will do an “all-or-nothing” reduction again:

$M'$  : on input  $x$

erase input and run  $M$  on the blank tape.

if  $M$  accepts, accept. If  $M$  rejects, reject.

We have  $M$  halts on the blank tape  $\Leftrightarrow M'$  halts on every input, so we are done.

We next show that  $\text{HaltB} \leq_m \overline{T}$ , implying that  $\overline{\text{HaltB}} \leq_m T$ , implying that  $T$  is not semi-decidable. This reduction is a bit tricky.

Assume that the input for  $\text{HaltB}$ , and assume is well-formed:  $\langle M \rangle$ .

Let  $f(\langle M \rangle) = \langle M' \rangle$  where  $M'$  is as follows.

$M'$  : On input  $x$

Simulate  $M$  on the blank tape for  $|x|$  steps;

if  $M$  halts within  $|x|$  steps, then  $M'$  goes into an infinite loop;

if  $M$  doesn't halt within  $|x|$  steps, then  $M'$  halts (and, say, accepts).

Clearly  $M$  halts on the blank tape  $\Leftrightarrow M'$  is not a total machine. □

**Example 6.** Let  $All = \{\langle M \rangle \mid M \text{ is a Turing Machine that accepts every input}\}$ .

We will show in one shot that  $All$  is neither semi-decidable nor co-semi-decidable by reducing  $T$  to it:  $T \leq All$ . Now,  $f(\langle M \rangle) = \langle M' \rangle$  where  $M'$  is the same as  $M$  except every occurrence of  $q_{reject}$  in  $M$  is changed to  $q_{accept}$ .