# Lecture 14: Encoding problems and decidability

9 March 2010

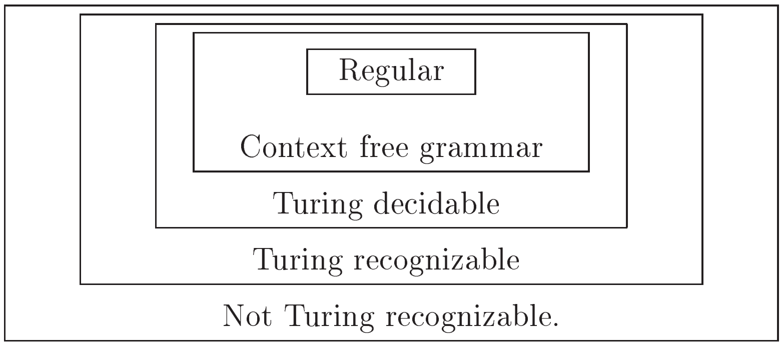
This lecture presents examples of languages that are ***Turing decidable***, and argues that existing “real” computers can be simulated by Turing machines.

## 1 Review and context

Remember that a Turing machine $\mathtt{D}$ can do three sorts of things on an input $w$. The $\mathsf{TM}$ $\mathtt{D}$ might halt and accept. It might halt and reject. Or it might never halt. A $\mathsf{TM}$ is a ***decider*** if it always halts on all inputs.

A $\mathsf{TM}$ ***recognizable*** language is a language $L$ for which there is a $\mathsf{TM}$ $\mathtt{D}$, such that $\mathsf{L}(\mathtt{D}) = L$. A $\mathsf{TM}$ decidable language is a language $L$ for which there is a decider $\mathsf{TM}$ $\mathtt{D}$, such that $\mathsf{L}(\mathtt{D}) = L$.

Here is a figure showing the hierarchy of languages.



Conceptually, when we think about algorithms in computer science, we are normally interested in code which is guaranteed to halt on all inputs. So, for questions about languages, our primary interest is in Turing decidable (not just recognizable) languages.

Any algorithmic task can be converted into decision problem about languages. Some tasks are naturally in this form, e.g. “Is the length of this string prime?”. In other cases, we have to restructure the question in one of two ways:

- A complex input object (e.g. a graph) may need to be encoded as a string.
- A construction task may have to be rephrased as a yes/no question.

## 2 $\mathsf{TM}$ example: Adding two numbers

### 2.1 A simple decision problem

For example, consider the task of adding two decimal numbers. The obvious algorithm might take two numbers $a$ and $b$ as input, and produce a number $c$ as output. We can rephrase this as a question about languages by asking "Given inputs $a$, $b$, and $c$, is $c = a + b$".

For the alphabet

$$\Sigma = \{\texttt{0}, \texttt{1}, \ldots, \texttt{9}, +, -\},$$

consider the language

$$L = \left\{ a_n a_{n-1} \ldots a_0 + b_m b_{m-1} \ldots b_0 = c_r c_{r-1} \ldots c_0 \;\middle|\; \begin{array}{l} a_i, b_j, c_k \in [0, 9] \text{ and} \\ \;\langle a_n a_{n-1} \ldots a_0 \rangle \\ \quad + \quad \langle b_m b_{m-1} \ldots b_0 \rangle \\ \qquad\qquad = \langle c_r c_{r-1} \ldots c_0 \rangle \end{array} \right\},$$

where $\langle a_n a_{n-1} \ldots a_0 \rangle = \sum_{i=0}^{n} a_i \cdot 10^i$ is the number represented in base ten by the string $a_n a_{n-1} \ldots a_0$.

We then ask whether we can build a TM which decides the language $L$.

## 2.2 A decider for addition

To build a decider for this addition problem, we will use a multi-tape TM. We showed (last class) that a multi-tape TM is equivalent to a single tape TM. First, let us build a useful helper function, which reverses the contents of one tape.

### 2.2.1 Reversing a tape

Given the content of tape ✇$_1$, we can reverse it easily in two steps using a temporary tape. First, we put a marker onto the temporary tape. Moving the heads on both tapes to the right, we copy the contents of ✇$_1$ onto the temporary tape.

Next, we put the ✇$_1$ head at the start of its tape, but the temporary tape head remains at the end of this tape. We copy the material back onto ✇$_1$, but in reverse order, moving the ✇$_1$ head rightwards and the temporary tape head leftwards.

Let **ReverseTape**($t$) denote the TM mechanism (i.e. procedure) that reverses the $t$th tape. We are going to buildup TM by putting together such procedures.

### 2.2.2 Adding two numbers

Now, let us assemble the addition algorithm. We will use five tapes: the input (✇$_1$), three tapes to hold numbers (✇$_2$, ✇$_3$, and ✇$_4$), and a scratch tape used for the reversal operation.

The TM will first scan the input tape (i.e., ✇$_1$), and copy the first number to ✇$_2$, and the second number to ✇$_3$. Next, we do **ReverseTape**(2) and **ReverseTape**(3). Now, we move the head of ✇$_2$ and ✇$_3$ to the beginning of the tapes, and we start moving them together computing the sum of the digits under the two heads, writing the output to ✇$_4$, and moving the three heads to the right. Naturally, we have a carry over digit, which we encode in the current state of the TM controller (the carry over digit is either $0, 1$ or $2$).

If one of the heads of ✇$_2$ or ✇$_3$ reaches the end of the tape, then we continue moving it, interpreting ␣ as a 0. We halt when the heads on both tapes see ␣.

Next, we move the head of ✇$_4$ back to the beginning of the tape, and do **ReverseTape**(4). Finally, we compare the content of ✇$_4$ with the number written on ✇$_1$ after the = character. If they are equal, the TM accepts, otherwise it rejects.

## 3 Encoding a graph problem

As the above example demonstrates, the ***coding scheme*** used for the input has big impact on the complexity of our algorithm. The addition algorithm would have been easier if the numbers were written in reverse order, or if they had been in binary. Such details may affect the running time of the algorithm, but they do not change whether the problem is Turing decidable or not.

When algorithms operate on objects that are not strings, these objects need to be ***encoded*** into strings before we can make the algorithm into a decision problem. For example, consider the following situation. We are given a ***directed graph*** $G = (V, E)$, and two vertices $s, t \in V$, and we would like to decide if there is a way to reach $t$ from $s$.

All sorts of encodings are possible. But it is easiest to understand if we use encodings that look like standard `ASCII` file, of the sort you might use as input to your `Java` or `C++` program. `ASCII` files look like they are two-dimensional. But remember that they are actually one-dimensional strings inside the computer. Line breaks display in a special way, but they are underlyingly just a special separator character (<`NL`> on a unix system), very similar to the \$ or # that we've used to subdivide items in our string examples.

To make things easy, we will number the vertices of $V$ from 1 to $n = |V|$. To specify that there is an edge between two vertices $u$ and $v$, we then specify the two indices of $u$ and $v$. We will use the notation $(u, v)$. Thus, to specify a graph as a text file, we could use the following format, where $n$ is the number of vertices and $m$ is the number of edges in the graph.

$$\begin{array}{l} n \\ m \\ (n_1, n'_1) \\ (n_2, n'_2) \\ \vdots \\ (n_m, n'_m) \end{array}$$

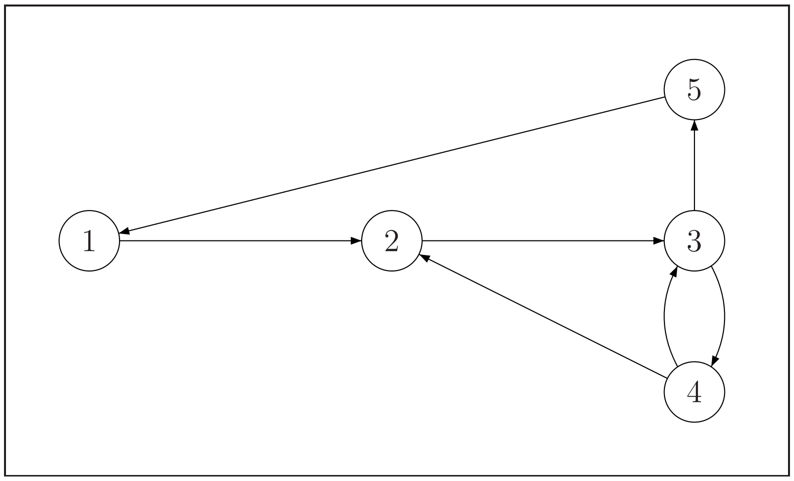
Namely, the first line of the file, will contain the number (written explicitly using `ASCII`), next the second line is the number of edges of $G$ (i.e., $m$). Then, every line specify one edge of the graph, by specifying the two numbers that are the vertices of the edge. As a concrete example, consider the following graph.

The number of edges is a bit redundant, because we could just stop reading at the end of the file. But it is convenient for algorithm design.

See Figure 1, for an example of a graph its encoding using these scheme.

## 4 Algorithm for graph reachability

To encode an instance of the ***s, t-reachability problem***, our `ASCII` file will need to contain not only the graph but also the vertices $s$ and $t$. The input tape for our TM would contain all this information, laid out in 1D (i.e. imagine the line break displayed as an ordinary separator character).



Graph encoding

```
5
7
(1,2)
(2,3)
(3,5)
(5,1)
(3,4)
(4,3)
(4,2)
```

Figure 1: A graph encoded as text. The string encoding the graph is in fact "`5`⟨`NL`⟩`7`⟨`NL`⟩`(1,2)`⟨`NL`⟩`(2,3)`⟨`NL`⟩`(3,5)`⟨`NL`⟩`(5,1)`⟨`NL`⟩`(3,4)`⟨`NL`⟩`(4,3)`⟨`NL`⟩`(4,2)`". Here ⟨`NL`⟩ denotes the spacial new-line character.

To solve this problem, we will need to search the graph, starting with node $s$. The TM accepts iff this search finds the node $t$. We will store information on four TM tapes, in addition to the input tape. The TM would have the following tapes:

✇$_1$: Input tape

✇$_2$: Target node $t$.

✇$_3$: Edge list.

✇$_4$: *Done list*: list of nodes that we've finished processing

✇$_5$: *To-do list*: list of nodes whose outgoing edges have not been followed

Given the graph, the TM reads the graph (checking that the input is in the right format). It puts the list of edges onto tape ✇$_3$, puts $t$ onto its own tape (i.e., ✇$_2$), and puts the node $s$ onto the to-do list tape (i.e., ✇$_5$).

Next, the TM loops. In each iteration, it removes the first node $x$ from the to-do list. If $x = t$, the TM halts and accepts. Otherwise, $x$ is added to the done list (i.e., ✇$_4$). Then the TM searches the Edge list for all edges going outwards from $x$. Suppose an outgoing edge goes from $x$ to $y$. Then if $y$ is not already on the finished list or the to-do list, then $y$ is added to the to-do list.

If there is nothing left on on the to-do list, the TM halts and rejects.

This algorithm is a ***graph search*** algorithm. It is breadth-first search if the new nodes are added to the end of the to-do list and depth-first search if they are added in the start of the list. (Or, said another way, the to-do list operates as either a queue or a stack.)

The separate visited list is necessary to prevent the algorithm from going into an infinite loop if the graph contains cycles.

## 5 Some decidable DFA problems

If $\mathtt{D}$ is a DFA, the string encoding of $\mathtt{D}$ is written as $\langle \mathtt{D} \rangle$.

The string encoding of a DFA is similar to the encoding of a directed graph except that our encoding has to have a label for each edge, specify the start state, and list the final states.

**Emptiness of DFA.** Consider the language

$$\mathsf{E}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D} \rangle \;\middle|\; \mathtt{D} \text{ is a DFA, and } \mathsf{L}(\mathtt{D}) = \emptyset \right\}.$$

This language is decidable. Namely, given an instance $\langle \mathtt{D} \rangle$, there is a TM that reads $\langle \mathtt{D} \rangle$, this TM always stops, and accepts if and only if $\mathsf{L}(\mathtt{D})$ is empty. Indeed, do a graph search on the DFA (as above) starting at the start state of $\mathtt{D}$, and check whether any of the final states is reachable. If so, the $\mathsf{L}(\mathtt{D}) \neq \emptyset$.

**Lemma 5.1** *The language $\mathsf{E}_{\mathsf{DFA}}$ is decidable.*

**Emptiness of NFA.** Consider the following language

$$\mathsf{E}_{\mathsf{NFA}} = \left\{ \langle \mathtt{D} \rangle \;\middle|\; \mathtt{D} \text{ is a NFA, and } \mathsf{L}(\mathtt{D}) = \emptyset \right\}.$$

This language is decidable. Indeed, convert the given NFA into a DFA (as done in class, long time ago) and then call the code for $E_{\mathsf{DFA}}$ on the encoded DFA. Notice that the first step in this algorithm takes the encoded version of $\mathtt{D}$ and writes the encoding for the corresponding DFA. You can imagine this as taking a state diagram as input and producing a new state diagram as output.

**Equal languages for DFAs.** Consider the language

$$\mathsf{EQ}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D}, \mathtt{C} \rangle \;\middle|\; \mathtt{D} \text{ and } \mathtt{C} \text{ are NFAs, and } \mathsf{L}(\mathtt{D}) = \mathsf{L}(\mathtt{C}) \right\}.$$

This language is also decidable. Remember that the ***symmetric difference*** of two sets $X$ and $Y$ is $X \oplus Y = (X \cap \overline{Y}) \cup (Y \cap \overline{X})$. The set $X \oplus Y$ is empty if and only if the two sets are equal. But, given a DFA, we know how to make a DFA recognizing the complement of its language. And we also know how to take two DFA's and make a DFA recognizing the union or intersection of their languages.

So, given the encodings for $\mathtt{D}$ and $\mathtt{C}$, our TM will construct the encoding of a DFA $\langle \mathtt{B} \rangle$ recognizing the symmetric difference of their languages. Then it would call the code for deciding if $\langle \mathtt{B} \rangle \in \mathsf{E}_{\mathsf{DFA}}$.

Informally, problems involving regular languages are always decidable, because they are so easy to manipulate. Problems involving context-free languages are sometimes decidable. And only the simplest problems involving Turing machines are decidable.

## 6 The acceptance problem for DFA's

The following language is also decidable:

$$\mathsf{A}_{\mathsf{DFA}} = \left\{ \langle \mathtt{D}, w \rangle \;\middle|\; \mathtt{D} \text{ is a } \mathsf{DFA},\ w \text{ is a word, and } \mathtt{D} \text{ accepts } w. \right\}.$$

As before, the notation $\langle \mathtt{D}, w \rangle$ is the encoding of the DFA $\mathtt{D}$ and the word $w$; that is, it is the pair $\langle \mathtt{D} \rangle$ and $\langle w \rangle$. For example, if $\langle w \rangle$ is just $w$ (it's already a string), then $\langle \mathtt{D}, w \rangle$ might be $\langle \mathtt{D} \rangle \# w$ where # is some separator character. Or it might be $(\langle \mathtt{D} \rangle, w)$. Or anything similar that encodes the input well. We will just assume that it is in some such reasonable encoding of a pair and that the low-level code for our TM (which we will not spell out in detail) knows what it is.

A Turing machine deciding $\mathsf{A}_{\mathsf{DFA}}$ needs to be able to take the code for some arbitrary DFA, plus some arbitrary string, and decide if that DFA accepts that string. So it will need to contain a general-purpose DFA simulator. This is called the ***acceptance problem*** for DFA's.

It's useful to contrast this with a similar-sounding claim. If $\mathtt{D}$ is any DFA, then $\mathsf{L}(\mathtt{D})$ is Turing-decidable. Indeed, to build a TM that accepts $\mathsf{L}(\mathtt{D})$, we simply move the TM head to the right over the input, using the TM's controller to simulate the controller of the DFA directly.

In this case, we are given a specific fixed DFA $\mathtt{D}$ and we only need to cook up a TM that recognizes strings from this one particular language. This is much easier than $\mathsf{A}_{\mathsf{DFA}}$.

To decide $\mathsf{A}_{\mathsf{DFA}}$, our TM will use five tapes:

✇$_1$: input: $\langle \mathtt{D}, w \rangle$,

✇$_2$: state,

✇$_3$: final states

✇$_4$: transition triples

✇$_5$: input string.

The simulator then runs as follows:

(1) Check the format of the input. Copy the start state to tape ✇$_2$. Copy the input string to tape ✇$_5$. Copy the transition triples and final states of the input machine $\langle \mathtt{D} \rangle$ to tapes ✇$_3$ and ✇$_4$.

(2) Put the tape ✇$_5$ head at the beginning of the tape.

(3) Find a transition triple $p \xrightarrow{\mathtt{c}} q$ (written on tape ✇$_4$) whose input state and character match the state written on tape ✇$_1$ (i.e., $p$) and the character (i.e., $c$) under the head on tape ✇$_5$.

(4) Change the current state of the simulated DFA from $p$ to $q$.

Specifically, copy the state $q$ (written on the triple we just found on ✇$_4$), to tape ✇$_2$.

(5) Move the tape $\text{✇}_5$ head to the right (i.e., the simulation handled this input character).

(6) Goto step (3).

(7) Halt the loop when the tape $\text{✇}_5$ head sees a blank. Accept if and only if the state on tape $\text{✇}_2$ is one of the states on list of final states, stored on tape $\text{✇}_3$.

## 7 Simulating a real computer with a Turing machine

We would like to argue that we can simulate a "real" world computer on a Turing machine. Here are some key program features that we would like to simulate on a TM.

- **Numbers & arithmetic**: We already saw in previous lecture how some basic integer operations can be handled. It is not too hard to extend these to negative integers and perform all required numerical operations if we allow a TM with multiple tapes. As such, we can assume that we can implement any standard numerical operation.

  Of course, can also do floating point operations on a TM. The details are overwhelming but they are quite doable. In fact, until 20 years[1] ago, many computers implemented floating point operations using integer arithmetic. Hardware implementation of floating point-operations became mainstream, when Intel introduced the i486 in 1989 that had FPU (floating-point unit). You would probably will see/seen how floating point arithmetic works in computer architecture courses.

- **Stored constant strings**: The program we are trying to translate into a TM might have strings and constants in it. For example, it might check if the input contains the (all important) string `UIUC`. As we saw above, we can encode such strings in the states. Initially, on power-up, the TM starts by writing out such strings, onto a special tape that we use for this purpose.

- **Random-access memory**: We will use an associative memory. Here, consider the memory as having a unique label to identify it (i.e., its address), and content. Thus, if cell 17 contains the value `abc`, we will consider it as storing the pair $(17, \texttt{abc})$. We can store the memory on a tape as a list of such pairs. Thus, the tape might look like:

  $$(17, \texttt{abc})\$(1, \texttt{samuel})\$(85, \texttt{noclue})\$ \ldots (11, \texttt{stamp})\$␣␣␣␣␣ \ldots$$

  Here, address 17 stores the string `abc`, address 1 stores the string `samuel`, and so on.

  Reading the value of address $x$ from the tape is easy. Suppose $x$ is written on $\text{✇}_i$, and we would like to find the value associated with $x$ on the memory tape and write it onto $\text{✇}_j$. To do this, the TM scans $\text{✇}_{mem}$ the memory tape (i.e., the tape we use to simulate the associative memory) from the beginning, till the TM encounter a pair in $\text{✇}_{mem}$ having $x$ as its first argument. It then copies the second part of the pair to the output tape $\text{✇}_j$.

[1] This number keep changing. Very irritating.

Storing new value $(x, y)$ in memory is almost as easy. If a pair having $x$ as first element exists you delete it out (by writing a special cross-out character over it), and then you write the new pair $(x, y)$ in the end of the tape ✇$_{mem}$.

If you wanted to use memory more efficiently, the new value could be written into the original location, whenever the original location had enough room. You could also write new pairs into crossed-out regions, if they have enough room. Implementations of C `malloc`/`free` and Java garbage collection use slightly more sophisticated versions of these ideas. However, TM designers rarely care about efficiency.

- **Subroutine calls**: To simulate a real program, we need to be able to do calls (and recursive calls). The standard way to implement such things is by having a stack. It is clear how to implement a stack on its own TM tape.

  We need to store three pieces of information for each procedure call:

  (i) private working space,

  (ii) the return value,

  (iii) and the name of the state to return to after the call is done.

  The private working space needs to be implemented with a stack, because a set of nested procedure calls might be active all at once, including several recursive calls to the same procedure.

  The return value can be handled by just putting it onto a designated register tape, say ✇$_{24}$.

  Right before we give control over to a procedure, we need to store the name of the state it should return to when it is done. This allows us to call a single fixed piece of code from several different places in our TM. Again, these return points need to be put on a stack, to handle nested procedure calls.

  After it returns from a procedure, the TM reads the state name to return to. A special set of TM states handle reading a state name and transitioning to the corresponding TM state.

These are just the most essential features for a very simple general-purpose computer. In some computer architecture class, you will see how to implement fancier program features (e.g. garbage collection, objects) on top of this simple model.