1 The Universal Turing Machine

In the last class, we saw that the definition of TM is robust: changing the alphabet, # of tapes, etc. do not change its computation power. Hence, w.l.o.g. assume a TM $M$ is composed of an alphabet $\Gamma$, a set of states $Q$, a transition function $\delta : Q \times \Gamma^3 \to Q \times \Gamma^2 \times \{L,S,R\}$, and 3 tapes: INPUT, WORK, and OUTPUT. A universal TM can be viewed as a virtual machine.

Definition 1 (Universal TM)

- A normal TM $\langle M \rangle \stackrel{\text{def}}{=} (\Gamma; Q; \delta)$ can be treated as a string in $\{0,1\}^*$.
- A TM $U$ is a universal machine if $U(\langle M \rangle, x) = M(x)$. □

An analogy is a compiler. Note that a compiler is of fixed-size, but it can compile code of any length. The input to $U$ is the description (encoding) of a TM, for example: $\Gamma \parallel Q \parallel \delta \parallel x$ ("\parallel" stands for concatenation in some unambiguous way). $U$ simulates the input TM.

We can have some convention $C$ about how to interpret an arbitrary string in $\{0,1\}^*$ as a TM. W.l.o.g., if a string doesn’t make any sense in our pre-specified convention $C$, we just treat it as a special TM that always output nothing and halts immediately after start. $C$ doesn’t map each string in $\{0,1\}^*$ into a meaningful TM just in the same way as some real-world programs don’t compile.

Encoding $C$: TM $\to$ Strings $\{0,1\}^*$

\[ M \mapsto \langle M \rangle \]

\[ M' \leftarrow x \]

Then, how to construct a universal TM $U$?

Description of $U$. $U$ is simply an interpreter. It has two parts of input $M$ encoded as $\langle M \rangle$ and input $x$ fed to $M$. $U$ uses a work tape to keep track of the runtime configuration of $M$. $U$ simulates each step of $M(x)$. At each such step, $U$ scans $M$’s current configuration on $U$’s work tape and it scans the transition function in $\langle M \rangle$ to find the according action. $U$ performs the corresponding action by updating the configuration. Finally, output of $M(x)$ is written out to the output tape.

Theorem 2 (Existence of $U$) The universal TM exists. □

2 The Halting Problem

Definition 3 (Halting problem)

\[ \text{Halt} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\} \]

\[ \text{Halt}(\langle M \rangle, x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{If } M(x) \text{ halts.} \\ 1 & \text{If } M(x) \text{ loops forever.} \end{cases} \]
Intuitive implication: You can prove whether some specific program (say, a program for finding the shortest path) halts or not eventually. But you cannot write a “universal” program that works for all programs. This entails that no automation for deciding TM halting exists.

Theorem 4 Halt is not computable.

Proof Assume towards contradiction that Halt is computable by a TM $M'$. Define

$$ g : \{0, 1\}^* \to \{0, 1\} $$

$$ g(x) = \begin{cases} 0 & \text{Halt}(x, x) = 1 \\ \text{loops forever} & \text{Halt}(x, x) = 0 \end{cases} $$

If Halt is computable then so is $g$. Suppose $g$ is computable by a TM $M$. What is $g(\langle M \rangle)$?

- If $g(\langle M \rangle) = 0$, then by the definition of $g(\cdot)$, $\text{Halt}(\langle M \rangle, \langle M \rangle) = 1$, which means that $M(\langle M \rangle)$ loops forever. So $g(\langle M \rangle) \neq M(\langle M \rangle)$.

- If $g(\langle M \rangle)$ loops forever, then by the definition of $g(\cdot)$, $\text{Halt}(\langle M \rangle, \langle M \rangle) = 0$, which means that $M(\langle M \rangle)$ halts. So $g(\langle M \rangle) \neq M(\langle M \rangle)$.

Digression. The proof we saw is by a method called diagonalization. It was invented by Georg Cantor to prove that the cardinality of real numbers ($\mathbb{R}$) is bigger than that of rational numbers ($\mathbb{Q}$).

3 Reductions

Halt can be used as the stepping stone to prove that many other problems are incomputable, too. The basic idea is to obtain contradiction by reducing Halt to a targeted problem. The assumption that the target problem is computable will imply that Halt is computable, which is a contradiction.

Definition 5 (Busy Beaver) Consider TM with just one tape. Alphabet $\Gamma = \{0, 1\}$. Have no input. Initial tape is 00...0. If they halt, content of tape is output. $BB(n) \triangleq$ maximal number of 1's on the tape output by a halting TM with $n$ states.

Historical notes. A busy beaver is a TM that attains the maximum number of steps performed or number of non-blank symbols finally on the tape among all TMs in a certain class. The TMs in this class must meet certain design specifications and are required to eventually halt after being started with a blank tape.

A busy beaver function quantifies these upper limits on a given measure, and is a non-computable function. In fact, a busy beaver function can be shown to grow faster asymptotically than any computable function. The concept was first introduced by Tibor Radó [2]. For example, let $\Sigma(n, 2)$ denote the largest # of 1's printable by an $n$-state, 2-symbol (i.e., $\{0, 1\}$) TM started on an initially blank tape before halting. Table 1 gives a sense how fast this function skyrockets.

Theorem 6 $BB$ is uncomputable.

\footnote{From “Busy Beaver”’s Wiki entry: \url{http://en.wikipedia.org/wiki/Busy_beaver}}
<table>
<thead>
<tr>
<th>2-state</th>
<th>3-state</th>
<th>4-state</th>
<th>5-state</th>
<th>6-state</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-symbol</td>
<td>4</td>
<td>6</td>
<td>13</td>
<td>$\geq 4,098$</td>
</tr>
</tbody>
</table>

**Proof**  Assume BB is computable by a TM $M$ with $k$ states. Let DBL be a TM with $k'$ states that performs the following operation

\[
\begin{array}{c}
\begin{array}{c}
11110\cdots0 \\
\underbrace{n}
\end{array}
\end{array}
\rightarrow
\begin{array}{c}
\begin{array}{c}
111110\cdots0 \\
\underbrace{2n}
\end{array}
\end{array}
\]

Let $n$ be large enough to be specific later. Consider the following Turing machine $M_n$:

- Writes $n$ 1's on tape: $00\cdots\rightarrow 11110\cdots$.
- Apply DBL: $11110\cdots\rightarrow 1\cdots10\cdots$.
  
  $\text{BB}(2n)$
- Apply $M$: $1\cdots110\cdots$.
  
  $\text{BB}(\text{BB}(2n))$
- Apply $M$ again: $1\cdots110\cdots$.

How many states does $M_n$ have? The first step takes $n$ states; the second $k'$ states; and the third and fourth $k$ states. So in total $n + 2k + k'$ states. We get that

\[\text{BB}(n + 2k + k') \geq \text{BB}(\text{BB}(2n)).\]

Note that BB is monotone increasing because $\text{BB}(n + 1) \geq \text{BB}(n)$ for any $n \in \mathbb{N}$. So $\text{BB}(2n) \leq n + 2k + k'$ which is clearly impossible if $n$ is large enough (say, $n > 2k + k'$).

It turns out that there are no non-trivial computable function of the output of a computation.

**Theorem 7 (Rice’s Theorem)** Let $f : \{0, 1\}^* \times \{0, 1\}^* \rightarrow \{0, 1\}$ be a computable function so that $f(\langle M \rangle, x)$ depends just on $M(x)$. Then $f$ must be a constant function.

Rice’s Theorem is in Exercise 1.12 in [1].

**Conjecture 8 (The 3n + 1 Problem, aka Collatz conjecture)** Consider the following function $f : \mathbb{N} \rightarrow \mathbb{N}$.

\[
f(n) = \begin{cases} 
  n/2 & \text{n even} \\
  3n + 1 & \text{n odd}
\end{cases}
\]

It is conjectured that for any starting value $n$, if we apply $f$ enough times we will reach 1 eventually (note that $f(f(f(1))) = 1$, e.g. 1 is in a cycle). This was verified for many values of $n$, but it is unknown if it is true for all $n$.

**Definition 9 (The Generalized 3n + 1 Problem)** Input: $m, a_0, b_0 \cdots, a_{m-1}, b_{m-1}$ natural numbers. Consider a function of the form

\[
f(n) = \begin{cases} 
  a_0n + b_0 & \text{n mod m = 0} \\
  a_1n + b_1 & \text{n mod m = 1} \\
  \cdots \\
  a_{m-1}n + b_{m-1} & \text{n mod m = m - 1}
\end{cases}
\]
For example, letting $m = 2$ and $a_0 = \frac{1}{2}, b_0 = 0, a_1 = 3, a_1 = 1$ yields the $3n + 1$ problem.

**Theorem 10** Consider the following function

$$F(m, a_0, b_0, \ldots, a_{m-1}, b_{m-1}) = \text{the generalized } 3n + 1 \text{ problem always reaches } 1 \text{ from any starting value}$$

is uncomputable. □

Here is another example of an uncomputable function.

**Definition 11** (Hilbert’s 10\textsuperscript{th} problem\footnote{http://en.wikipedia.org/wiki/Hilbert’s_problems}) Given a multivariate polynomial $p(x_1, \ldots, x_n)$ with integer coefficients, figure out if $p$ has an integer solution. □

**Theorem 12** This is uncomputable. □

### 4 Time Complexity

**Definition 13** (Time complexity) TM $M$ has time complexity $T : \mathbb{N} \rightarrow \mathbb{N}$ if for all input $x \in \{0, 1\}^*$,

- $M(x)$ always halts;
- $M(x)$ halts after at most $T(|x|)$ steps, where $|x|$ denotes the length of the input $x$. □

$f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ has time complexity $T$ if there exists a TM $M$ computing $f$ which has time complexity $T$.

**Definition 14** (TIME) \(\text{TIME}(T(n))\)\textsuperscript{def} = functions computable in time $O(T(n))$. □

For example, the problem of deciding whether a given graph is bipartite ($f: \text{graph} \rightarrow \text{Is it bipartite?}$) is \(\text{in} \ \text{TIME}(n^2)\). Be cautious that this is an upperbound instead of a lowerbound.

**Definition 15** ($\mathbb{P}$ and $\mathbb{EXP}$)
- \(\mathbb{P}\)\textsuperscript{def} = \bigcup_{c \geq 1} \text{TIME}(n^c). \text{E.g., shortest paths, linear programming.} □
- \(\mathbb{EXP}\)\textsuperscript{def} = \bigcup_{c \geq 1} \text{TIME}(2^{nc}). □
- \(\mathbb{E}\)\textsuperscript{def} = \bigcup_{c \geq 1} \text{TIME}(2^{cn}). \text{E.g., SAT.} □

**Remark \#1:** Why do we care about polynomial-time computation? Why care about polynomial time and not specifically on linear, quadratic, or cubic time? The answer is, in reality, the degrees of polynomials that upperbound the running time of problems in $\mathbb{P}$ are usually very small. $\mathbb{P}$ captures the intuition of efficiently computation. However, in some scenarios, people do want to find algorithms that run in a more restricted time bound, say, linear. For example, algorithms for property testing may require sub-linear time, which means that it cannot even read the whole input.

**Remark \#2:** The definition of $\mathbb{EXP}$ is robust. Some graph algorithms, for $n$-vertex graph takes $2^n$ time. How do we encode the graph (say, it’s sparse) matters:

- List of edges: $\sim n$ (algorithm runs in $O(2^n)$ steps).
- Adjacency matrix: $\sim n^2$ (algorithm runs in $O(2^{\sqrt{n}})$ steps).

$\mathbb{EXP}$ is more robust than $\mathbb{E}$ in that a problem cannot escape $\mathbb{EXP}$ even its description’s size is blown up from $n$ to $n^2$. Usually, we take natural encoding for a problem and they fit into $\mathbb{EXP}$. 
References
