Lecture 12

Turing machines

In this lecture we will discuss the *Turing machine* model of computation. This model is named after Alan Turing (1912–1954), who proposed it in 1936. It is difficult to overstate Alan Turing's influence on the subject of this course—theoretical computer science effectively started with Turing's work, and for this reason he is sometimes referred to as the father of theoretical computer science.

The Church–Turing thesis

The intention of the Turing machine model is to provide a simple mathematical abstraction of general computations. The idea that Turing machine computations are representative of a fully general computational model is called the *Church–Turing thesis*. Here is one statement of this thesis, but understand that it is the idea rather than the exact choice of words that is important.

Church–Turing thesis: Any function that can be computed by a mechanical process can be computed by a Turing machine.

Note that this is not a mathematical statement that can be proved or disproved. If you wanted to try to prove a statement along these lines, the first thing you would most likely do is to look for a mathematical definition of what it means for a function to be "computed by a mechanical process," and this is precisely what the Turing machine model was intended to provide.

There are alternative models of computation that offer abstractions of general computations. One example is λ-calculus, which was proposed by Alonzo Church a short time prior to Turing's introduction of what we now call Turing machines. These two models, Turing machines and λ-calculus, are equivalent in the sense that any computation that can be performed by one of them can be performed by the other. Turing sketched a proof of this fact in his 1936 paper. We will see another example in Lecture 14 when we show that a model called the *stack machine* model is equivalent to the Turing machine model. Based on the stack machine model, or directly on the Turing machine model, it is not conceptually difficult to envision the simulation of a model of computation abstracting the notion of a *random access machine*.

While machines behaving like Turing machines have been built, this is mainly a recreational activity. The Turing machine model was never intended to serve as a practical approach to performing computations, but rather was intended to provide a rigorous mathematical foundation for reasoning about computation—and it has served this purpose very well since its introduction.

12.1 Definition of the Turing machine model

We will begin with an informal description of the Turing machine model before stating the formal definition. There are three components of a Turing machine:

1. The *finite state control*. This component is in one of a finite number of states at each instant.
2. The *tape*. This component consists of an infinite number of *tape squares*, each of which can store one of a finite number of *tape symbols* at each instant. The tape is infinite both to the left and to the right.
3. The *tape head*. The tape head can move left and right on the tape, and is understood to be scanning exactly one of the tape squares at the start of each computational step. The tape head can read which symbol is stored in the square it scans, and it can write a new symbol into that square.

Figure 12.1 illustrates these three components. It is natural to imagine that the tape head is connected in some way to the finite state control.

The idea is that the action of a Turing machine at each instant is determined by the state of the finite state control together with the single symbol stored in the tape square that the tape head is currently reading. Thus, the action is determined by a finite number of possible alternatives: one action for each state/symbol pair. Depending on the state and the symbol being scanned, the action that the machine is to perform may involve changing the state of the finite state control, changing the symbol in the tape square being scanned, and moving the tape head to the left or right. Once this action is performed, the machine will again have some state for its finite state control and will be reading some symbol on its tape, and the process continues. One may consider both deterministic and nondeterministic variants of the Turing machine model, but our main focus will be on the deterministic variant of the model.

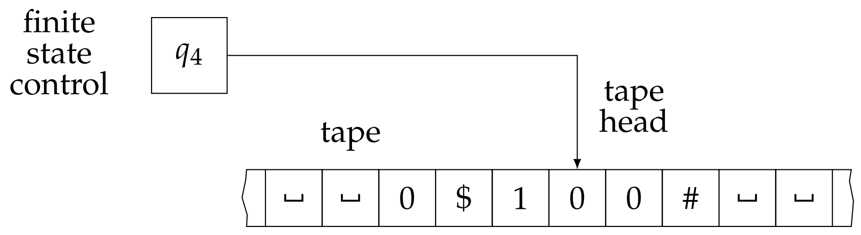

Figure 12.1: An illustration of the three components of a Turing machine: the finite state control, the tape head, and the tape. The tape is infinite in both directions (although it appears that this Turing machine's tape was torn at both ends to fit it into the figure).

When a Turing machine begins a computation, an input string is written on its tape, and every other tape square is initialized to a special *blank* symbol, which may not be included in the input alphabet. We need an actual symbol to represent the blank symbol in these notes, and we will use the symbol ␣ for this purpose. More generally, we will allow the possible symbols written on the tape to include other non-input symbols in addition to the blank symbol, as it is sometimes convenient to allow this possibility.

We also give Turing machines an opportunity to stop the computational process and produce an output by requiring them to have two special states: an *accept* state q_{acc} and a *reject* state q_{rej}. These two states are deemed *halting states*, and all other states are *non-halting states*. If the machine enters a halting state, the computation immediately stops and *accepts* or *rejects* accordingly. When we discuss language recognition, our focus is naturally on whether or not a given Turing machine eventually reaches one of the states q_{acc} or q_{rej}, but we can also use the Turing machine model to describe the computation of functions by taking into account the contents of the tape if and when a halting state is reached.

Formal definition of DTMs

With the informal description of Turing machines from above in mind, we will now proceed to the formal definition.

Definition 12.1. A *deterministic Turing machine* (or DTM, for short) is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\mathrm{acc}}, q_{\mathrm{rej}}), \tag{12.1}$$

where Q is a finite and nonempty set of *states*, Σ is an alphabet called the *input alphabet*, which may not include the blank symbol ␣, Γ is an alphabet called the

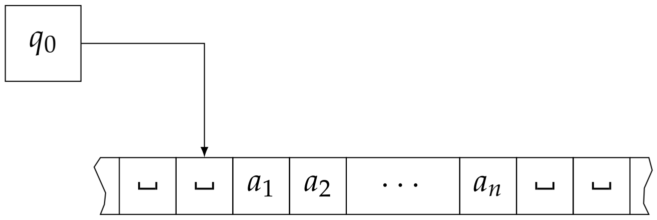

Figure 12.2: The initial configuration of a DTM when run on input $w = a_1 a_2 \cdots a_n$.

tape alphabet, which must satisfy $\Sigma \cup \{\sqcup\} \subseteq \Gamma$, δ is a transition function having the form

$$\delta : Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}, \tag{12.2}$$

$q_0 \in Q$ is the *initial state*, and $q_{\text{acc}}, q_{\text{rej}} \in Q$ are the *accept* and *reject* states, which must satisfy $q_{\text{acc}} \neq q_{\text{rej}}$.

The interpretation of the transition function is as follows. Suppose the DTM is currently in a state $p \in Q$, the symbol stored in the tape square being scanned by the tape head is $a \in \Gamma$, and it is the case that

$$\delta(p, a) = (q, b, D) \tag{12.3}$$

for $D \in \{\leftarrow, \rightarrow\}$. The action performed by the DTM is then to

1. change state to q,
2. overwrite the contents of the tape square being scanned by the tape head with b, and
3. move the tape head in direction D (either left or right).

In the case that the state is q_{acc} or q_{rej}, the transition function does not specify an action, because we assume that the DTM halts once it reaches one of these two states.

Turing machine computations

If we have a DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, and we wish to consider its operation on some input string $w \in \Sigma^*$, we assume that it is started with its components initialized as illustrated in Figure 12.2. That is, the input string is written on the tape, one symbol per square, with every other tape square containing the blank symbol, and with the tape head scanning the tape square immediately to the left of the first input symbol. In the case that the input string is ε, all of the tape squares start out storing blanks.

Once the initial arrangement of the DTM is set up, the DTM begins taking *steps*, as determined by the transition function δ in the manner suggested above. So long as the DTM does not enter one of the two states q_{acc} or q_{rej}, the computation continues. If the DTM eventually enters the state q_{acc}, it *accepts* the input string, and if it eventually enters the state q_{rej}, it *rejects* the input string. Thus, there are three possible alternatives for a DTM M on a given input string w:

1. M accepts w.
2. M rejects w.
3. M runs forever on input w.

In some cases one can design a particular DTM so that the third alternative does not occur, but in general it might.

Representing configurations of DTMs

In order to speak more precisely about Turing machines and state a formal definition concerning their behavior, we will require a bit more terminology. When we speak of a *configuration* of a DTM, we are speaking of a description of all of the Turing machine's components at some instant. This includes

1. the *state* of the finite state control,
2. the *contents* of the entire tape, and
3. the *tape head position* on the tape.

Rather than drawing pictures depicting the different parts of Turing machines, like in Figure 12.2, we use the following compact notation to represent configurations. If we have a DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, and we wish to refer to a configuration of this DTM, we express it in the form

$$u(q,a)v \tag{12.4}$$

for some state $q \in Q$, a tape symbol $a \in \Gamma$, and (possibly empty) strings of tape symbols u and v such that

$$u \in \Gamma^* \backslash \{\sqcup\}\Gamma^* \quad \text{and} \quad v \in \Gamma^* \backslash \Gamma^*\{\sqcup\}. \tag{12.5}$$

In words, u and v are strings of tape symbols, u does not start with a blank, and v does not end with a blank. The interpretation of the expression (12.4) is that it refers to the configuration in which the string uav is written on the tape in consecutive squares, with all other tape squares containing the blank symbol, the state of M

is q, and the tape head of M is positioned over the symbol a that occurs between u and v. For example, the configuration of the DTM in Figure 12.1 is expressed as

$$0\$1(q_4,0)0\# \tag{12.6}$$

while the configuration of the DTM in Figure 12.2 is

$$(q_0,\sqcup)w \tag{12.7}$$

for $w = a_1 \cdots a_n$.

When working with descriptions of configurations, it is convenient to define a few functions as follows. We define $\alpha : \Gamma^* \rightarrow \Gamma^* \backslash \{\sqcup\}\Gamma^*$ and $\beta : \Gamma^* \rightarrow \Gamma^* \backslash \Gamma^*\{\sqcup\}$ recursively as

$$\begin{aligned} \alpha(w) &= w && (\text{for } w \in \Gamma^* \backslash \{\sqcup\}\Gamma^*) \\ \alpha(\sqcup w) &= \alpha(w) && (\text{for } w \in \Gamma^*) \end{aligned} \tag{12.8}$$

and

$$\begin{aligned} \beta(w) &= w && (\text{for } w \in \Gamma^* \backslash \Gamma^*\{\sqcup\}) \\ \beta(w\sqcup) &= \beta(w) && (\text{for } w \in \Gamma^*), \end{aligned} \tag{12.9}$$

and we define

$$\gamma : \Gamma^*(Q \times \Gamma)\Gamma^* \rightarrow \big(\Gamma^* \backslash \{\sqcup\}\Gamma^*\big)(Q \times \Gamma)\big(\Gamma^* \backslash \Gamma^*\{\sqcup\}\big) \tag{12.10}$$

as

$$\gamma(u(q,a)v) = \alpha(u)(q,a)\beta(v) \tag{12.11}$$

for all $u, v \in \Gamma^*$, $q \in Q$, and $a \in \Gamma$. This is not as complicated as it might appear: the function γ just throws away all blank symbols on the left-most end of u and the right-most end of v, so that a proper expression of a configuration remains.

A yields relation for DTMs

In order to formally define what it means for a DTM to accept, reject, compute a function, and so on, we will define a *yields relation*, similar to what we did for context-free grammars.

Definition 12.2. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a DTM. The *yields* relation $\vdash_M$, defined on pairs of configurations of M, includes exactly these pairs:

1. *Moving right.* For every choice of $p \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

$$\delta(p,a) = (q,b,\rightarrow), \tag{12.12}$$

the yields relation includes these pairs for all $u \in \Gamma^* \backslash \{\sqcup\}\Gamma^*$, $v \in \Gamma^* \backslash \Gamma^*\{\sqcup\}$, and $c \in \Gamma$:

$$\begin{aligned} u(p,a)cv &\vdash_M \gamma\bigl(ub(q,c)v\bigr) \\ u(p,a) &\vdash_M \gamma\bigl(ub(q,\sqcup)\bigr). \end{aligned} \tag{12.13}$$

2. *Moving left.* For every choice of $p \in Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

$$\delta(p,a) = (q, b, \leftarrow), \tag{12.14}$$

the yields relation includes these pairs for all $u \in \Gamma^* \backslash \{\sqcup\}\Gamma^*$, $v \in \Gamma^* \backslash \Gamma^*\{\sqcup\}$, and $c \in \Gamma$:

$$\begin{aligned} uc(p,a)v &\vdash_M \gamma\bigl(u(q,c)bv\bigr) \\ (p,a)v &\vdash_M \gamma\bigl((q,\sqcup)bv\bigr). \end{aligned} \tag{12.15}$$

We also let $\vdash_M^*$ denote the reflexive, transitive closure of $\vdash_M$. That is, we have

$$u(p,a)v \vdash_M^* y(q,b)z \tag{12.16}$$

if and only if there exists an integer $m \geq 1$, strings $w_1, \ldots, w_m, x_1, \ldots, x_m \in \Gamma^*$, symbols $c_1, \ldots, c_m \in \Gamma$, and states $r_1, \ldots, r_m \in Q$ such that $u(p,a)v = w_1(r_1, c_1)x_1$, $y(q,b)v = w_m(r_m, c_m)x_m$, and

$$w_k(r_k, c_k)x_k \vdash_M w_{k+1}(r_{k+1}, c_{k+1})x_{k+1} \tag{12.17}$$

for all $k \in \{1, \ldots, m-1\}$.

A more intuitive description of these relations is that the expression

$$u(p,a)v \vdash_M y(q,b)z \tag{12.18}$$

means that by running M for one step we move from the configuration $u(p,a)v$ to the configuration $y(q,b)z$; and

$$u(p,a)v \vdash_M^* y(q,b)z \tag{12.19}$$

means that by running M for some number of steps, possibly zero steps, we will move from the configuration $u(p,a)v$ to the configuration $y(q,b)z$.

12.2 Semidecidable and decidable languages; computable functions

Now we will define the classes of *semidecidable* and *decidable* languages as well as the notion of a *computable function*.

To define the classes of decidable and semidecidable languages, we must first express formally, in terms of the yields relation defined in the previous section, what it means for a DTM to accept or reject.

Definition 12.3. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a DTM and let $w \in \Sigma^*$ be a string. If there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_0, \sqcup)w \vdash_M^* u(q_{\text{acc}}, a)v, \tag{12.20}$$

then M *accepts* w. If there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_0, \sqcup)w \vdash_M^* u(q_{\text{rej}}, a)v, \tag{12.21}$$

then M *rejects* w. If neither of these conditions hold, then M *runs forever* on input w.

In words, if a DTM is set in its initial configuration, for some input string w, and starts running, it *accepts* w if it eventually enters its accept state, it *rejects* w if it eventually enters its reject state, and it *run forever* if neither of these possibilities holds. It is perhaps obvious, but nevertheless worth noting, that accepting and rejecting are mutually exclusive—because DTMs are deterministic, each configuration has a unique next configuration, and it follows that a DTM M cannot simultaneously accept a string w and reject w.

Similar to what we have done for other computational models, we write $\mathrm{L}(M)$ to denote the language of all strings that are accepted by a DTM M. In the case of DTMs, the language $\mathrm{L}(M)$ does not really tell the whole story—a string $w \notin \mathrm{L}(M)$ might either be rejected or it may cause M to run forever—but the notation is useful nevertheless.

We now define the class of semidecidable languages to be those languages recognized by some DTM.

Definition 12.4. Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language A is *semidecidable* if there exists a DTM M such that $A = \mathrm{L}(M)$.

The name *semidecidable* reflects the fact that if $A = \mathrm{L}(M)$ for some DTM M, and $w \in A$, then running M on w will necessarily lead to acceptance; but if $w \notin A$, then M might either reject or run forever on input w. That is, M does not really decide whether a string w is in A or not, it only "semidecides"; for if $w \notin A$, you might never learn this with certainty as a result of running M on w. There are several alternative names that people often use in place of semidecidable, including *Turing recognizable*, *partially decidable*, and *recursively enumerable* (or *r.e.* for short).

Next, as the following definition makes clear, we define the class of *decidable* languages to be those languages for which there exists a DTM that correctly answers whether or not a given string is in the language, never running forever.

Definition 12.5. Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language A is *decidable* if there exists a DTM M with these two properties:

1. M accepts every string $w \in A$.
2. M rejects every string $w \in \overline{A}$.

The names *recursive* and *computable* are sometimes used in place of *decidable*.

Finally, let us define what it means for a function to be *computable*. We do this for functions mapping strings to strings, but not necessarily having the same input and output alphabets, as this generality will be important in future lectures.

Definition 12.6. Let Σ and Γ be alphabets and let $f : \Sigma^* \to \Gamma^*$ be a function. The function f is *computable* if there exists a DTM $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ such that the relation

$$(q_0, \sqcup)w \vdash_M^* (q_{\text{acc}}, \sqcup)f(w) \tag{12.22}$$

holds for every string $w \in \Sigma^*$. In this case we also say that DTM M *computes* f.

In this definition, Δ can be any tape alphabet; by definition it must include all of the symbols in the input alphabet Σ, and it must also include all symbols from the alphabet Γ that appear in any output string in order for the relation (12.22) to hold.

In words, a function is computed by a DTM if, when run on any choice of an input string to that function, it eventually accepts, leaving the correct output written on the tape surrounded by blanks, with the tape head one square left of this output string.

12.3 A simple example of a Turing machine

Let us now see an example of a DTM, which we will describe using a state diagram. In the DTM case, we can represent the property that the transition function satisfies $\delta(p, a) = (q, b, \rightarrow)$ with a transition of the form

$$\begin{tikzcd} p \arrow[r, "{a, b \rightarrow}"] & q \end{tikzcd}$$

and similarly we represent the property that $\delta(p, a) = (q, b, \leftarrow)$ with a transition of the form

$$\begin{tikzcd} p \arrow[r, "{a, b \leftarrow}"] & q \end{tikzcd}$$

The state diagram for the example is given in Figure 12.3. The DTM M described by this diagram recognizes the language

$$\text{SAME} = \{0^n 1^n : n \in \mathbb{N}\}. \tag{12.23}$$

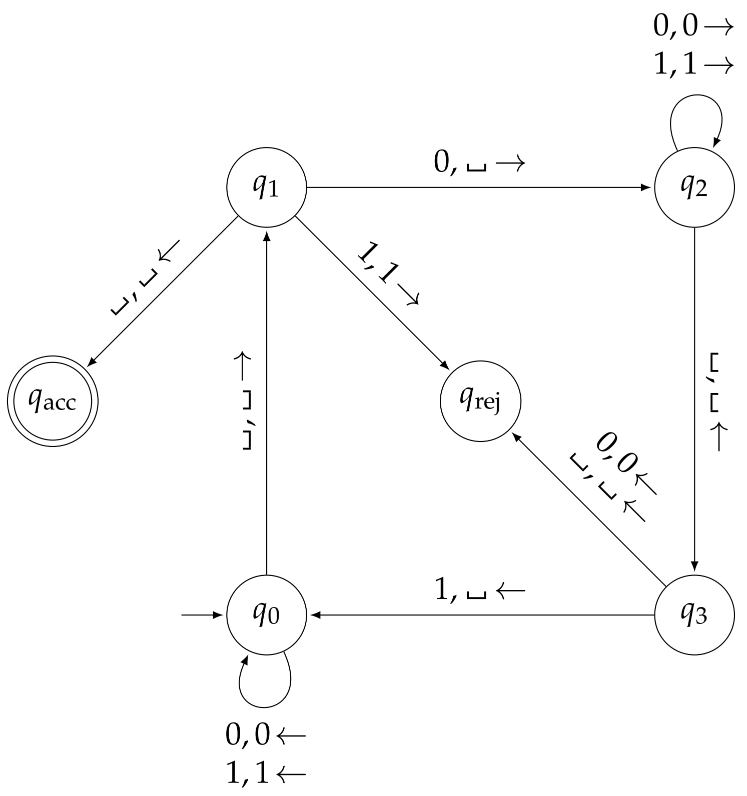

Figure 12.3: A DTM M for the language SAME $= \{0^n 1^n : n \in \mathbb{N}\}$.

To be more precise, M accepts every string in SAME and rejects every string in $\overline{\text{SAME}}$, so never does the DTM run forever. Thus, SAME is decidable.

The specific way that the DTM M works can be summarized as follows. The DTM M starts out with its tape head scanning the blank symbol immediately to the left of its input. It moves the tape head right, and if it sees a 1 it rejects: the input string must not be of the form $0^n 1^n$ if this happens. On the other hand, if it sees another blank symbol, it accepts: the input must be the empty string, which corresponds to the $n = 0$ case in the description of SAME. Otherwise, it must have seen the symbol 0, and in this case the 0 is erased (meaning that it replaces it with the blank symbol), the tape head repeatedly moves right until a blank is found, and then it moves one square back to the left. If a 1 is not found at this location the DTM rejects: there were not enough 1s at the right end of the string. Otherwise, if a 1 is found, it is erased, and the tape head moves all the way back to the left, where we essentially recurse on a slightly shorter string.

Of course, the summary just suggested does not tell you precisely how the DTM works—but if you did not already have the state diagram from Figure 12.3, the summary would probably be enough to give you a good idea for how to come up with the state diagram (or perhaps a slightly different one operating in a similar way).

In fact, an even higher-level summary is enough to convey the basic idea of how this DTM operates. We could, for instance, describe the functioning of the DTM M as follows:

1. Accept if the input is the empty string.
2. Check that the left-most non-blank symbol on the tape is a 0 and that the right-most non-blank symbol is a 1. Reject if this is not the case, and otherwise erase these symbols and goto step 1.

There will, of course, be several specific ways to implement this algorithm with a DTM, with the DTM M from Figure 12.3 being one of them.

The DTM M being discussed is very simple, which makes it atypical. The DTMs we will be most interested in will almost always be much more complicated—so complicated, in fact, that the idea of representing them by state diagrams would be absurd. The reality is that state diagrams turn out to be almost totally useless for describing DTMs, and so we will rarely employ them. Doing so would be analogous to describing a complex program using machine language instructions.

The more usual way to describe DTMs is in terms of *algorithms*, often expressed in the form of pseudo-code or high-level descriptions like the last description of M above. It may not be immediately apparent precisely which high-level algorithm descriptions can be run on Turing machines, but as an intuitive guide one may have confidence that if an algorithm can be implemented using your favorite programming language, then it can also be run on a deterministic Turing machine. The discussions in the two lectures to follow this one are primarily intended to help to build this intuition.