1 Acceptance, Rejection, and I/O for Turing Machines

Definition 1.1 (Initial Configuration) *If $M = (K, \Sigma, \delta, s, H)$ is a Turing machine and $w \in (\Sigma - \{\sqcup, \rhd\})^*$ then the* initial configuration of M on input *w is $(s, \rhd\underline{\sqcup}w)$.*

Definition 1.2 (4.2.1 modified, Acceptance, rejection, halting) *Let $M = (K, \Sigma, \delta, s, H)$ be a Turing machine such that $H = \{y, n\}$.*

- *Then any halting configuration whose state is y is called an* accepting configuration *and a halting configuration whose state is n is called a* rejecting configuration*.*
- *We say M* accepts *an input $w \in (\Sigma - \{\sqcup, \rhd\})^*$ if there is an accepting configuration C such that $(s, \rhd\underline{\sqcup}w) \vdash_M^* C$ and M rejects w if there is a rejecting configuration C such that $(s, \rhd\underline{\sqcup}w) \vdash_M^* C$.*
- *Also, M* halts *on input $w \in (\Sigma - \{\sqcup, \rhd\})^*$ if there is a halting configuration C such that $(s, \rhd\underline{\sqcup}w) \vdash_M^* C$.*

Note that it is also possible for the Turing machine to *loop*, that is, to continue computing forever.

Let $\Sigma_0 \subseteq \Sigma - \{\sqcup, \rhd\}$ be an alphabet called the *input alphabet* of M.

Definition 1.3 (4.2.1, Decidability) *A Turing machine M* decides *a language $L \subseteq \Sigma_0^*$ if for any $w \in \Sigma_0^*$,*

- *if $w \in L$ then M accepts w and*
- *if $w \notin L$ then M rejects w.*

Note that M never loops for inputs in Σ_0^*. Also, L is recursive (decidable) *if there is a Turing machine M such that M decides L.*

For this definition, it is not necessary that we know which Turing machine decides L. If there is one, then L is decidable, and similarly for semidecidable.

Definition 1.4 (4.2.4, Semidecidability)

- M semidecides (partially decides) $L \subseteq \Sigma_0^*$ *if for any string* $w \in \Sigma_0^*$, $w \in L$ *if and only if* M *halts on input* w.
- *A language* L *is* recursively enumerable (partially decidable, semidecidable) *iff there is a Turing machine* M *that semidecides* L.

Here is an equivalent definition of semidecides that may be more intuitive.

Definition 1.5 (Semidecidability, equivalent definition) *Suppose Turing machine* M *has two halting states* $\{y, n\}$. *Then* M *semidecides* $L \subseteq \Sigma_0^*$ *if for any* $w \in \Sigma_0^*$,

- *if* $w \in L$ *then* M *accepts* w *and*
- *if* $w \notin L$ *then either* M *loops on input* w *or* M *rejects* w.

It can be shown that this definition is equivalent to the preceding one.

- This definition allows for the Turing machine to halt on some inputs that are not in L.
- This means that the language is partially decidable if when the answer is "yes," the machine always answers correctly.
- When the answer is "no," the machine may sometimes answer "no" and may sometimes fail to answer at all (by looping).

This definition makes it clear that any language that is decidable is also partially decidable.

Interesting facts:

- If L is recursive then it is also recursively enumerable.
- If L is recursive then $\overline{L}$ (that is, $\Sigma_0^* - L$) is also recursive.
- L is recursive if and only if both L and $\overline{L}$ are recursively enumerable.
- There are recursively enumerable languages that are not recursive.
- There are recursively enumerable languages L such that $\overline{L}$ is not recursively enumerable.

We have the following hierarchy:

$$\text{FINITE} \subset \text{REGULAR} \subset \text{CFL} \subset \text{DECIDABLE} \subset \text{SEMIDECIDABLE}$$

1.1 Problems and Languages

We need to relate languages to problems. A language L is a subset of Σ^* for some finite alphabet Σ. A problem Q is something like, "Given an integer n, is n prime?"

In general, a problem Q consists of a set S (like the integers) and a property P of elements of that set (such as primality). Then one seeks a method which, given any element x of S, such as an integer n, will decide if the property P of x is true (is n prime). This method is then a solution to Q.

A problem Q can be expressed as a language L_Q in the following way.

- An alphabet Σ is chosen, and elements of S are expressed as elements of Σ^* by some encoding function $\mathit{encode}(x)$.
- Then the language L_Q corresponding to the problem Q is

$$\{\mathit{encode}(x) : x \in S, x \text{ has property } P\}.$$

- So a solution to the problem Q is a method which, given an element $\mathit{encode}(x)$ of Σ^*, can determine if $\mathit{encode}(x) \in L_Q$.

1.1.1 Examples

Let Q be the problem, "Given nonnegative integer x, is x prime?" Let $\Sigma = \{0, 1\}$ and let $\mathit{encode}(x)$ represent x in binary. Then L_Q is $\{\mathit{encode}(x) : x \text{ is prime}\}$ which is $\{10, 11, 101, 111, 1011, \ldots\}$.

Let Q be the problem, "Given nonnegative integer x, is x even?" Let $\Sigma = \{0, 1\}$ and let $\mathit{encode}(x)$ represent x in binary. Then L_Q is $\{\mathit{encode}(x) : x \text{ is even}\}$ which is $\{0, 10, 100, 110, 1000, \ldots\}$.

Let Q be the problem, "Will UNC have a winning basketball season next year?" Then S is just the set containing UNC, and P is the property of having a winning season. Encode UNC as UNC (a single symbol) so that *encode* maps UNC onto UNC. Thus $\Sigma = \{UNC\}$. If UNC will have a winning season then $L_Q = \{UNC\}$ else $L_Q = \{\}$.

1.1.2 Decidability of problems

- We say the problem Q is *decidable* if the language L_Q is decidable (by a Turing machine). This depends on the encoding, but we assume a reasonable encoding.
- We say the problem Q is *partially decidable* if the language L_Q is partially decidable. This also depends on the encoding
- Finite problems are always decidable, even if we don't know the Turing machine M that can decide them. So the problem Q of whether UNC will have a winning basketball season next year is decidable, because in either case there is a Turing machine that decides the language L_Q.
- Likewise given any single mathematical conjecture A, the problem Q of whether A is provable, is decidable, even if we don't know which Turing machine decides L_Q.

1.2 Computing Functions

Turing machines can also compute functions such as addition and substraction. The basic idea is that the Turing machine accepts as input, the input to the function, and when it halts, leaves the value of the function on the tape.

Definition 1.6 (Output of a Turing machine) *Let $M = (K, \Sigma, \delta, s, \{h\})$ be a Turing machine. Let $\Sigma_0 \subseteq \Sigma - \{\sqcup, \rhd\}$ be an alphabet, and let $w \in \Sigma_0^*$. Suppose M halts on input w and $(s, \rhd\underline{\sqcup}w) \vdash_M^* (h, \rhd\underline{\sqcup}y)$ for some $y \in \Sigma_0^*$. Then y is called the* output of M on input w*, denoted by $M(w)$.*

Note that $M(w)$ is defined only if M halts on input w and on a configuration of the stated form. Thus M can be a partial function, seen in this sense.

Definition 1.7 (Turing machine computes a function) *Let f be any total function from Σ_0^* to Σ_0^*. We say that M* computes f *if for all $w \in \Sigma_0^*$, $M(w) = f(w)$. Thus M halts on all such w and outputs $f(w)$ given input w.*

Definition 1.8 (Recursive function) *A function f is called* recursive *(*computable*) if there is a Turing machine M that computes f.*

This definition only specifies functions from strings to strings. To get functions on other objects, such as integers or sets, they have to be encoded as strings.

Multiple inputs can be separated by some delimiter, such as a semicolon. So, for example, a Turing machine to compute addition might have an input of the form $101;11$ and output 1000. One can also represent graphs or arrays or even Turing machines as strings, if necessary. We are free to use any encoding, but the definitions will not be very meaninngful unless a reasonable encoding is chosen.

When one represents integers as binary, and uses other reasonable encodings, then it turns out that all the usual functions, such as addition, subtraction, multiplication, and so on are computable.

1.3 The Busy Beaver Function

There are several definitions of various versions of the Busy Beaver function, a very fast growing uncomputable function. In fact, the Busy Beaver function is not upper bounded by any computable function, and grows much faster than any computable function. The text on page 253 defines it this way:

Definition 1.9 (Busy Beaver Function) *The* busy beaver function $\beta : N \mapsto N$ *has the property that* $\beta(n)$ *is the largest number* m *such that there is a Turing machine with alphabet* $\{\triangleright, \sqcup, a, b\}$ *and with exactly* n *states which, when started with the blank tape, eventually halts at configuration* $(h, \triangleright\sqcup a^m)$.

Other definitions do not require the one's in the output to be consecutive, but just count the total number of one's written. Other definitions count the total number of moves made, not the number of one's written. Some definitions allow a two-way infinite tape and allow Turing machines that can both write on the tape and move left or right on the same move.

Rado's Σ function is defined with a two-way infinite tape and a machine that can both write and move at the same time, and counts the number of one's on the tape when the machine halts. Also, the tape is assumed to have two symbols, one of which is regarded as a blank and the other as a one. Also, $S(n)$ is the maximum number of steps used by such a machine that eventually halts.

Known values:

n	$\Sigma(n)$	$S(n)$
0	0	0
1	1	1
2	4	6
3	6	21
4	13	107
5	≥ 4098	$\geq 47{,}176{,}870$
6	$\geq 10^{18267}$	$\geq 10^{36534}$

- The problem with trying to compute this function is that there are some very long running Turing machines that we cannot tell whether they will ever halt.

- In fact, this will always be so, no matter how advanced our mathematics becomes, because the Busy Beaver Function is uncomputable, meaning that no Turing machine can compute it.

- Some machines that loop, do so in very clever ways that our mathematics will never be able to understand. For example, there are some long-running 5-state Turing machines for which people still don't know whether they will ever halt.

It's a wonder that people were even able to compute how long some of the long running machines ran, because they certainly couldn't simulate them!

Some simulations of the Busy Beaver Machines are linked from the course web page.

Why does this matter? One might think that it would be possible just to run Turing machines for a small number of steps, and if they hadn't halted by then, they never would. The Busy Beaver function shows dramatically why this approach does not work.

1.4 Importance of the Topic

Why discuss decidability and partial decidability at all?

- We can show that some problems are not decidable, and that some functions are not computable, as you will see.

- This helps us, because if we know that a problem is not decidable, we don't have to waste time trying to find a way to decide it.

- Example: Ambiguity of context-free grammars is undecidable.
- Then we can look for special cases that can be solved, or possibly approximate a solution by heuristics.

Even if a problem is not decidable, it may be partially decidable. Then we can still get some useful information about it by a partial decision procedure. An example where this is true is theorem proving in first-order and other logics.

Some problems are known to be not even partially decidable. This is helpful because it saves us the effort of trying to find a partial decision procedure for them.

In this way we can concentrate our effort on what is achievable, and not waste our time and effort on impossibilities.