Lecture 14

Properties of Turing machines and their languages

14.1 Equivalent forms of Turing machines

In this section, we will look at various modifications of Turing machines and show that these modifications do not increase their power.

14.1.1 Change of alphabet

First, we will see that the size of the tape alphabet of a Turing machine does not matter, so long as it has at least two distinct symbols. The size of the input alphabet Σ matters in the sense that it affects which types of input strings the TM can take as input, but even the input alphabet can be assumed to be $\Sigma = \{0, 1\}$ if we allow the inputs to be encoded in binary.

To see that the size of the tape alphabet Γ does not matter, we will simulate a TM M with some large finite tape alphabet Γ using a TM M' which has a small tape alphabet Γ'. Let's assume for simplicity that $\Gamma' = \{0, 1, \textvisiblespace\}$. Then we can encode each $a \in \Gamma$ using $k = \lceil \log_2 |\Gamma| \rceil$ bits in $\{0, 1\}$. We will write a sequence of k symbols on the tape of M' to represent a single symbol of M.

Now, each transition of M looks like $\delta(p, a) = (q, b, D)$ for some states p and q of M, some symbols $a, b, \in \Gamma$, and some direction $D \in \{\leftarrow, \rightarrow\}$. To simulate this in M', we will have one state in M' for each state of M, plus many additional helper states. When the TM is at state p of M', we will need it to read the alphabet symbol $a \in \Gamma$ encoded on the tape, which means it should read k symbols of Γ'. To do so, we take the following approach. For each state p of M', we will have one helper state p_w for each string $w \in (\Gamma')^*$ of length at most k. From the non-helper state p, we then take k transitions which all leave the tape unchanged and move right; these transitions will encode the k tape symbols seen into the state p_w by reading one bit at a time. For example, if the first tape symbol seen when at state p is 1, we move to state p_1 and move right; if we then see symbol 0 on the tape, we move to state p_{10} and move right again, and so on.

At the end of this sequence of k steps, we are at the state p_w, where $|w| = k$ and w is the string that was written on the k tape symbols. We can then take an additional k transitions (using additional helper states), moving left each time, to return to the beginning of the k tape symbols, where we began. At this point, we have managed to read k tape symbols, remember them (using the internal states), and return to the same tape position as before. We now know the state p of M we started at, and also the symbol $a \in \Gamma$ that was encoded on the tape. The next task is to overwrite a with b, move to internal state q of M, and then move left or right according to direction D. These

can all be accomplished using more helper states in a tedious and not too insightful manner.

While the argument above is certainly not a rigorous proof, hopefully it convinces you that the main idea can work. The trick is simply that because even a large alphabet Γ is required to be finite, we can encode its symbols using a finite number k of bits, and we can then use a finite number of states to read and memorize chunks of k bits at a time. This allows us to simulate large tape alphabets using small tape alphabets, so the size of the tape alphabet does not affect the power of the machine.

As a side note, while we went through this argument for $\Gamma' = \{0, 1, \textvisiblespace\}$, the same style of argument can even work when Γ' contains only $\textvisiblespace$ plus a single additional symbol.

14.1.2 Multi-tape Turing machines

A multi-tape Turing machine is a new type of machine which has access to some constant number k of distinct tapes. Such a machine still has a finite set Q of internal states, with a single state state q_0, accept state q_{acc}, and reject q_{rej}; it also still has a finite nonempty input alphabet Σ as well as a tape alphabet Γ. The main difference is that the transition function now depends on k different symbols of the tape alphabet, since the head of the machine now reads k different tapes simultaneously. That is, δ acts on tuples $(p, a_1, a_2, \ldots, a_k)$, and outputs tuples $(q, b_1, b_2, \ldots, b_k, D_1, D_2, \ldots, D_k)$ where $a_1, a_2, \ldots, a_k, b_1, b_2, \ldots, b_k \in \Gamma$ and $D_1, D_2, \ldots, D_k \in \{\leftarrow, \rightarrow\}$. Note that a multi-tape Turing machine can move in different directions on different tapes; for example, in a single transition it may move left on its first tape and right on its second tape.

For a multi-tape Turing machine, we will assume that the input is written on the first tape (to the right of the head position) as normal, and that the other tapes are all blank. Of course, it is not hard for a multi-tape Turing machine to copy over the input into its other tapes (remember that each transition can read from all tapes at once, write independent symbols on all tapes, and move left or right on all tapes).

What is the power of multi-tape Turing machines? Perhaps surprisingly, it turns out they are no more powerful than ordinary Turing machines.

Let M be a multi-tape Turing machine, and let k be the number of tapes it has. We will define a single-tape Turing machine M' which simulates the behavior of M.

The tape alphabet of M' will have one symbol a for each symbol $a \in \Gamma$, an additional new symbol $\overline{a}$ for each symbol $a \in \Gamma$, and also an additional new symbol $\#$. We will store all k tapes of M on the single tape of M', using $\#$ as a separator symbol between the tapes, and using $\overline{a}$ to mark the position of the head of M on each of its k tapes. In other words, if the configuration of M on each of its k tapes is $u_1(p, a_1)v_1, u_2(p, a_2)v_2, \ldots u_k(p, a_k)v_k$, then we will ensure that the tape of the simulating machine M' will have the string $\#u_1\overline{a_1}v_1\#u_2\overline{a_1}v_2\# \ldots \#u_k\overline{a_k}v_k\#$ written on its tape.

It is not hard for M' to initialize the tape to this configuration: it starts in configuration $(q_0, \textvisiblespace)w$ where w is the input string, and all it needs to do is write $\overline{\textvisiblespace}$ in its starting position (this is a blank symbol with an overline on top, to mark that the head position is currently at this blank symbol), move one left to write $\#$, and then move right until it sees a blank symbol and write a sequence of symbols $\#\overline{\textvisiblespace}\#\textvisiblespace\# \ldots \#$ which repeats a constant number of times (depending on k). The machine M' can then have one state for each state of M, plus additional helper states which help it store the k contents of all head positions; the machine M can sweep across its input tape each time (counting $k + 1$ total occurrences of the separator symbol $\#$, including the outer ones), and use the helper internal states to memorize the symbols $a_1, a_2, \ldots, a_k$ corresponding to the k head positions of M. It can then have transitions that match the transition function of M, except that each time it needs to overwrite $a_1, a_2, \ldots, a_k$ with the symbols $b_1, b_2, \ldots, b_k$, M' will do another sweep from start to finish, looking for the k head positions and replacing the tape contents accordingly.

One remaining issue is how to deal with the moving head positions. It is not hard for M' to do a sweep and move each head position left or right as specified by the transition rule of M that it is trying to implement. But what should M' do if this causes a head position to hit the end of the tape (that is, to go on a # symbol)? The answer is that when this happens, M' needs to expand the length of the tape in that position. M' can do this by going to the last separator symbol #, moving it further to the right, and then iteratively going back and moving every preceding symbol to the right by one step, until it reaches the problematic marked symbol that landed on #. At this point, M' has effectively expanded the i-th tape if the i-th tape ran out of space, and it can put a blank symbol in the newly created position (it should also mark that blank symbol since the head position of M should now be there).

Once again, the above is just a sketch, since formally constructing such simulations is quite tedious (and does not provide much additional insight). Hopefully this convinces you that with sufficient effort, it is possible to simulate any multi-tape Turing machine using a regular single-tape Turing machine. Also note that although our construction increased the tape alphabet Γ, we can use the previous construction to reduce the tape alphabet Γ down to something small.

14.2 Everything a computer can do can be done by a Turing machine

Next, we will argue (informally) that anything that can be done by an ordinary computer can also be done by a Turing machine. In particular, everything that you can program using your favorite programming language can be "programmed" using a Turing machine as well.

To see why, recall that all programming languages eventually get converted into assembly code. This assembly code has a finite list of commands, and uses a small constant number of registers. Each command reads a block of memory into a register, writes to a block of memory from a register, or performs some simple operations on registers such as addition, multiplication, or checking for equality.

We can implement this behavior using a Turing machine. We will use a 3-tape Turing machine for convenience. The first tape will represent the memory of the computer (surrounded by a special symbol #); the second tape will represent the registers, which we also separate by the special symbol #; and the third tape will be a helper "work" tape. Each line of assembly code will correspond to one state of the Turing machine, plus a set of helper states.

Now, consider a line of assembly code that says "go to the memory position specified by register 3, and copy its contents into register 5". We need to implement this action in the Turing machine, using the state corresponding to this line of code together with its set of helper states. The TM will do so as follows. First, it will copy over register 3 into its work tape. Second, it will go to the beginning of the memory tape, and start counting towards the right memory position, as specified by the work tape; to do this, is repeatedly subtracts 1 from the number written on the work tape, and moves one the right on the memory tape by the appropriate length (the size of one "word", in assembly), until the number on the work tape is 0 (subtracting 1 from the binary representation of a number is not hard to do on a Turing machine). Finally, the TM copies over the memory cell into register 5. All of this is implemented by the TM state corresponding to that assembly line of code together with its helper states; after this is finished, the TM transitions into the state corresponding to the next line of assembly code.

Implementing other assembly commands works similarly. Note that by repeatedly subtracting 1 from one number and adding 1 to another (until the former reaches 0), a Turing machine can add two numbers. Similarly, by repeatedly adding, a Turing machine can multiply two numbers. It can write

to memory in a similar way to reading from memory. Finally, the GOTO command is particularly easy: a Turing machine implements this by simply transitioning to the state corresponding to the new assembly line of code.

From now on, we will often describe Turing machines simply by informally sketching an algorithm for a given task. The understanding is that if you can translate the informal description into computer code, then it is also possible to translate it into a Turing machine.

14.3 Deciding regular and context-free languages

Recall that a language A is decidable if there is a TM M that accepts each string in A and rejects each string not in A.

14.3.1 Every regular language is decidable

Let A be a regular language. Then there is some DFA M which recognizes it. It is now simple to construct a Turing machine M' that decides A. The Turing machine M' will have one state for each state of the DFA M, as well as separate start, accept, and reject states. From its start state, M' will always move one step right and transition to the start state of M. Then it will follow all transitions of M, using the tape symbol it reads as the input, and moving right each time. That is, of M has $\delta(p, a) = q$, then M' will have transition $\delta'(p, a) = (q, a, \rightarrow)$. When M' sees a blank symbol (meaning the input has ended), it transitions to q_{acc} if its current state is an accept state of M, and otherwise it transitions to q_{rej}. That is, $\delta'(p, \textvisiblespace) = (q_{\text{acc}}, \textvisiblespace, \rightarrow)$ for each accept state p of M and $\delta'(p, \textvisiblespace) = (q_{\text{rej}}, \textvisiblespace, \rightarrow)$ for each non-accept state of M.

It is easy to see that the above construction gives a Turing machine M' which decides the language of M, though we will not formally go through the proof of this.

14.3.2 Every context-free language is decidable

We could try to use the same approach to simulate a PDA using a Turing machine. One issue, however, is that PDAs are non-deterministic, while Turing machines are deterministic. This means it will not be so simple to simulate a PDA and know when to accept and reject each string.

Instead, to show that every context-free language is decidable, we will use the fact that each context-free language has a context-free grammar in Chomsky normal form. Let A be a context-free language, and let G be a CFG for A in Chomsky normal form. We will now construct a TM M which decides A; that is, for each input string $w \in \Sigma^*$, the machine M must accept if G can generate w and must reject if G cannot generate w.

We will describe a Turing machine informally, by sketching the algorithm it will follow. First, the TM will check if w is empty. If G has the rule $S \rightarrow \epsilon$, we program the TM to accept the empty string, but otherwise we code it to reject the empty string.

Next, the TM will compute the length n of w, and then expand out all ways of starting from the start variable S of G and following rules that lead to a string of length n. Enumerating all these possibilities is not hard to do: recall that each rule of the CFG must look like either $X \rightarrow YZ$ or $X \rightarrow a$, where X, Y, Z are variables and a is an alphabet symbol (the empty string ϵ cannot occur except for the special rule $S \rightarrow \epsilon$, which we've already dealt with). Hence each rule of the form $X \rightarrow YZ$ extends the final length of the generated string by exactly 1. Starting from S, we follow $n - 1$ such rules followed by n rules of the form $X \rightarrow a$. Since G has only a finite set of rules, there are only finitely many choices of rules at each step, and a Turing machine can go through all of them separately (in some fixed order). Then, for each generated string x, the TM will check if

$x = w$; if this happens, it will accept, while if this never happens (after checking all finitely many expansions of the rules), the TM will reject.

Of course, this is a very informal construction, but it should give you an idea of how a Turing machine can decide a context-free language, even though PDAs are nondeterministic.

14.4 Closure properties

14.4.1 Behavior under complement

We now examine some closure properties of decidable and recognizable languages. Recall once again that a language A is *decidable* if there is a TM M which accepts strings in A and rejects strings not in A, while a language is *recognizable* if there is a TM M which accepts strings in A and *either rejects or loops* on each string not in A. This difference is very important. We of course have that every decidable language is recognizable, by definition. However, a recognizable language does not need to be decidable; if we have a TM M that accepts strings in A but may loop on strings not in A, it is not clear how to convert it into a TM M' that *rejects* the strings not in A rather than looping on them.

On the other hand, we claim that if a language A and its complement $\overline{A}$ are both recognizable, then A is decidable. To see this, note that if A and $\overline{A}$ are recognizable, then we have Turing machines M and M' that (respectively) recognize them. We can now construct a Turing machine M'' which decides A. This machine M'' will work as follows: on input w, it will simulate both M and M' on w in parallel; a simple way to do this is to use two tapes in M'', one for M and one for M', and then have each state of M'' correspond to a pair of states (one from M and one from M'). This way, each step of M'' can implement one step of M and one step of M'.

On input w, the machine M'' then runs M and M' in parallel, both on w. If M ever enters its accept state, M'' accepts (since this means that $w \in A$). Similarly, if M rejects, M'' rejects.

On the other hand, if M' ever enters its accept state, M'' rejects (since M' accepts if and only if $w \in \overline{A}$, which means $w \notin A$). Similarly, if M' rejects, M'' accepts.

It is clear that M'' only accepts strings in A and only rejects strings not in A. To show that M'' decides A, we also need to show that M'' always halts. Now, we know that M halts (indeed, accepts) on all strings in A, and also that M' halts on all strings in $\overline{A}$. Since each string is either in A or in $\overline{A}$, it follows that on each string w, either M halts or M' halts (or both). Hence M'' always halts, which means it decides A.

Note also that if A is decidable, $\overline{A}$ is decidable. This is because if M decides A, we can create a Turing machine M' which acts just like M except with accept and reject states flipped. This Turing machine decides $\overline{A}$.

On the other hand, if A is recognizable, it does not necessarily mean that $\overline{A}$ is recognizable. This is because flipping the accept and reject states does not flip the looping behavior; that is, if M loops on a string $w \notin A$, then flipping the accept and reject states does not prevent the machine from looping on w, so the new machine does not accept each string in $\overline{A}$ (and hence does not recognize A).

This motivates a new definition.

Definition 14.1. *Let A be a lanugage. If $\overline{A}$ is recognizable, we call A* co-recognizable.

Note that the complement of a recognizable language is a co-recognizable language, and the complement of a co-recognizable language is a recognizable language. We have also seen that if a language is both recognizable and co-recognizable, then it is decidable.

14.4.2 Other closure properties

We claim that the recognizable languages are closed under union. To see this, let A and B be two recognizable languages, and let M_A and M_B be Turing machines recognizing them. We construct a Turing machine M for the language $A \cup B$. On input w, M will simply run both M_A and M_B on w in parallel, and then accept as soon as one of them accepts. Note that if $w \in A \cup B$, then either $w \in A$ or $w \in B$, which means that either $M_A(w)$ or $M_B(w)$ accepts; hence M accepts w. Conversely, if M accepts w, then either M_A or M_B must accept w, which means $w \in A \cup B$. We conclude that $L(M) = A \cup B$, so $A \cup B$ is recognizable.

The decidable languages are also closed under union. To see this, consider same construction above. If M_A and M_B decide A and B respectively, then they both halt on all strings. We can then modify M so that it rejects as soon as both M_A and M_B reject. Then M will halt on all strings, and it will accept a string w if and only if either M_A or M_B accepts w, meaning that M decides $A \cup B$.

Next, the recognizable languages are closed under intersection. To see this, let A and B be recognizable, and let M_A and M_B recognize them. We construct a Turing machine M for $A \cap B$. On input w, this machine M will simulate M_A and M_B on w in parallel. It will reject if one of them rejects, and accept if they both accept (it can also loop, for example if M_A and M_B never halt on w). It is not hard to check that $L(M) = A \cap B$, since M accepts a string if and only if it is accepted by both M_A and M_B.

The decidable languages are also closed under the intersection. The same construction above works: note that if M_A and M_B decide A and B (rather than merely recognizing them), then they always halt on all strings, which means that on all input w, eventually either M_A and M_B both accept or one of them rejects. This means M always halts on all w, and hence it decides $A \cap B$.

14.5 Turing machines with a fixed tape size

What happens if we limit the tape size of a Turing machine to some fixed length k (which is fixed in advance, so cannot depend on the input size)? Well, first of all, if we do this then there is no longer enough space on the tape to write down the input string w if $|w| > k$. To fix this, we could consider giving the bits of the input to the Turing machine one at a time, so that each transition depends on the next input symbol (just like in a DFA). The Turing machine could then write down those symbols on its tape, if it wishes to do so.

This new model turns out to simply be equivalent to DFAs. Why? Because there are only finitely many configurations of a finite tape (there are $|\Gamma|^k$ of them, to be precise).