CSCI3390-Lecture 3: Church-Turing Thesis, Multitape Machines

September 11, 2018

1 Summary

- The *Church-Turing thesis* states that a language is Turing-decidable if and only if it can be decided by what we understand to be an algorithmic procedure; in other words, that this precise mathematical definition corresponds to the intuitive notion of what a computation is.
- As support for the Church-Turing thesis, we can prove that a large number of apparently stronger computational models are no more powerful than Turing machines, that is, they recognize exactly the same family of languages. We do this for a variant of Turing machines in which there is more than one tape.
- We give several different proofs (corresponding to several different ways of reasoning about Turing machines) that a language is decidable if and only if both it and its complement are Turing recognizable.

2 The Church-Turing Thesis

Is it true, as Turing claimed, that anything that we might reasonably call a computation (or, as we are more likely to say today, an *algorithm*) can be carried out by a Turing machine? In terms of decision problems, the claim is this:

> There is an algorithm to decide, 'yes' or 'no', if a given string belongs to a set L of strings if and only if L is a Turing-decidable lan-

guage. There is a partial algorithm for L if and only if L is a Turing-recognizable language. Here a 'partial algorithm' means an algorithm that gives an answer of 'yes' for exactly the strings in L, but may fail to give an answer for strings that are not in L.

This claim is called the *Church-Turing thesis.* (Alonso Church formulated a different mathematical model of computation, provably equivalent to Turing's, and made the same claim for his model.) This is something that we cannot really prove or disprove, because it claims to provide a precise mathematical formulation of an intuitive philosophical concept. Turing himself advanced several arguments for the correctness of the thesis. One of these was based on a careful analysis of what the intuitive notion of computation really entails. The other was based on proving its equivalence to other models. For example, one can prove that anything computable by a program in a high-level language like Python (or C, or Java, ...) can be computed by a Turing machine. There is nothing, to our knowledge, that anyone has proposed that we would all agree to call an algorithm, but that cannot be carried out by a Turing machine.

3 Multitape Machines

One potential objection to the Church-Turing thesis is that the Turing machine model appears to be so weak compared to, say, what can be done in a modern programming language. But, as we mentioned above, we really can prove that these apparently stronger models cannot do anything that a Turing machine cannot do. Here we introduce a different, seemingly stronger model of computation, and prove that any problem it solves can be solved by an ordinary Turing machine.

Let's allow our Turing machine to have several tapes. A configuration of a two-tape machine is shown in Figure 1. At each step, the current state and the pair of scanned symbols determines the new state, the symbols to write on each of the tapes, and the direction of movement of each of the two reading heads. Formally, the next-state function has the form:

$$\delta : (Q \backslash \{\text{accept}, \text{reject}\}) \times \Gamma \times \Gamma \to Q \times \Gamma \times \Gamma \times \{L, R, S\} \times \{L, R, S\}.$$

Note that we allow a third 'direction' S, for the head remaining on the same tape cell, rather than moving left or right.

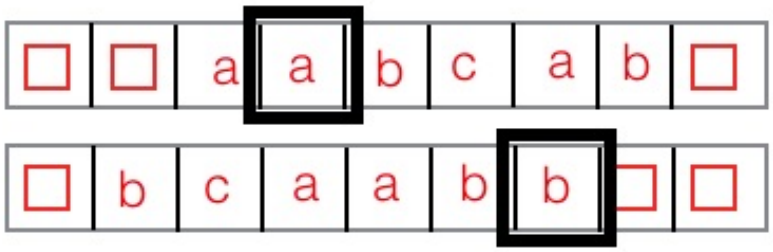

Figure 1: A configuration of a two-tape Turing machine. The state is not shown in this illustration.

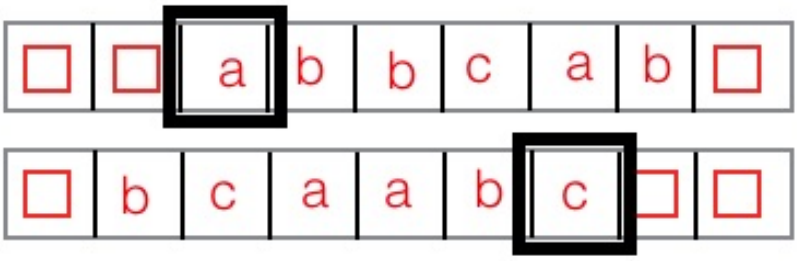

Figure 2: The configuration immediately following the one shown in the preceding figure, after executing a transition $\delta(q, a, b) = (q', b, c, L, S)$.

The two-tape machine begins in the initial state with the input string written on the first tape, and the reading head of the first tape positioned on the first symbol of input. The second tape is initially blank, so it doesn't matter where the reading head is.

Here is how a two-tape machine would solve the problem of determining whether the input string (written on the first tape) contains equal numbers of a's and b's: The machine initially marks an X on a cell of the second tape. Then in each step, it moves one cell to the right on the first tape. If the scanned symbol on the first tape is a, the head of the second tape moves right, and if the scanned symbol on the first tape is b, the head moves left on the second tape. When the first head reaches a blank, we check to see if the scanned symbol on the second tape is X or blank: this tells us whether the numbers of a's and b's are equal.

This computation requires something like $n + 2$ steps for an input of length

n, involving $2n$ separate head movements. In contrast, as we saw, the one-tape machine for the same problem requires time proportional to n^2, so this way of solving the problem is much faster.

Similarly, think of how we would use a 2-tape machine to solve the problem of adding two integers in binary. We write the first summand on the first tape, and the second on the second tape. The answer will be on the second tape when the machine halts: In the initial phase, both heads scan to the right, until the rightmost bit of each input is found. Since the two inputs might not be of the same length, if the end of, say, the first input is found before the other, then the read-write head on the first tape will remain stationary until the second one catches up. The heads now scan from right to left, adding in the normal way, and overwriting the bits on the second tape with the corresponding bits of the sum. The state is used to remember whether there was a carry out of the previously-visited column. If each summand has no more than n bits, then the whole process requires about $2n$ steps. The same problem on a one-tape machine, will require a number of steps proportional to n^2, because of all the shuttling back and forth it has to do.

Nonetheless, anything the two-tape machine (or k-tape machine) can do, the one-tape machine can do as well. Furthermore, the slowdown is not worse than quadratic–*i.e.,* the running time approximately squares, just as in the examples above.

Theorem 1 *A language $L \subseteq \Sigma^*$ is Turing-recognizable (respectively, decidable) if and only if it is recognizable (resp. decidable) by a two-tape Turing machine.*

Here is an outline of the proof: One direction is trivial. If the language is Turing-recognizable (which means recognizable by a one-tape machine), then we can certainly recognize it with a two-tape machine that never uses any information about its second tape. So the real content of the theorem is the converse direction. We will prove that for any two-tape machine $\mathcal{M}$ there is a one-tape machine $\mathcal{M}'$ such that for any input string w, $\mathcal{M}$ accepts w if and only if $\mathcal{M}'$ accepts w, and likewise for rejection. Thus the one-tape machine $\mathcal{M}'$ has to simulate the two-tape machine $\mathcal{M}$. Figure 3 illustrates how the simulation works: The contents of the two tapes are interleaved in alternating cells of the single tape. For each tape symbol of the two-tape machine, there are two alternate versions of the symbol (indicated by letters with a prime and a double prime in the figure) to show the location of the reading head on each tape. Thus the tape alphabet of $\mathcal{M}'$ is three times the size of the tape alphabet of $\mathcal{M}$.

In the initial phase of the simulation, the one-tape machine transforms its

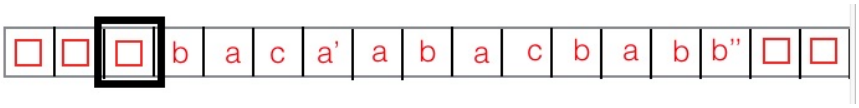

Figure 3: The configuration of the one-tape machine simulating the configuration of the two-tape machine shown in Figure 1. The two tapes are interleaved, and the primed letters represent new tape symbols indicating where the reading heads of two-tape machine are located.

input—let us say it's

$$aababa$$

into

$$a'\square''a\square b\square a\square b\square a\square.$$

(Noted the primed symbols a' and $\square''$ to indicate the reading head location.) In your homework you will be asked to show how a one-tape Turing machine can execute this transformation.

Now the simulation proper begins. Each single transition of the two-state machine is simulated by a sequence of transitions of the one-tape machine. At the beginning of the sequence, the head of the one-tape machine will be at the left of start of the tape contents. It then scans right, looking for both a cell associated with tape 1 containing a primed symbol, and a cell associated with tape 2 containing symbol with a double prime. This requires a substantial increase in the number of states: We must record in the state of $\mathcal{M}'$ what the corresponding state of $\mathcal{M}$ is, whether it has found the alternate symbol associated with tape 1, or tape 2, or both, and which symbols these are.

Once $\mathcal{M}'$ has located the second alternate symbol, it modifies the cell and neighboring cells in accordance with the transitions of $\mathcal{M}$, then heads left to find again the first alternate symbol, and again modifes it. It then enters a state corresponding to the next state in $\mathcal{M}$, and continues left until it finds two consecutive blank cells. Now it is ready for the next transition.

When $\mathcal{M}'$ enters a state corresponding to the accepting state of $\mathcal{M}$, it halts and accepts (and likewise for the reject state).

Writing out all the details of the simulation of a single transition is a bit unpleasant (so we won't do it!) but it involves no really new ideas. The behavior of the one-tape machine on the rightward scan is very much like that of the machine we saw earlier that checks for equal numbers of a's and b's. The states of the

one-tape machine correspond to 'states of mind' like

> I am in state q of $\mathcal{M}$. I have found a double-primed symbol a'' and am now looking for a single-primed symbol.

much like the state 'I've found an a and now I am looking for a b' that we saw earlier.

How long does this simulation take? More precisely, how much *slower* is this one-tape machine than the two-tape machine? Let us suppose that the input has n symbols, and the computation of $\mathcal{M}$ on w requires t steps. After these t steps, the combined lengths of the contents of the two tapes cannot be larger than $n + 2t$, because we will add at most one symbol to each tape in each step.

The initial phase of transforming the input will require roughly n^2 steps of $\mathcal{M}'$. (This will be part of a homework problem.) The simulation of a transition will make two scans of the input, one to the right and one to the left, and use no more than 4 steps to update the tape contents. So this will require no more than $2(n + 2t) + 4$ steps. As a result, the total number of steps executed by $\mathcal{M}'$ is no more than $2nt+4t^2+4t$. In most cases of interest, we will have $n \leq t$, because the machine will at least have to look at each cell of its input, so this gives an upper bound of $6t^2 + 4t$ steps. So the quadratic slowdown we observed earlier holds in general.

The analogous argument works for 3 or more tapes. We *still* have a quadratic slowdown, but with larger coefficients.

4 How to reason about Turing machines

There is a tradeoff between the simplicity of the Turing machine model and the complexity of the the specifications of Turing machines to carry out specific tasks. When we want to prove that there is a Turing machine that can recognize a particular language, or compute a particular function, there are essentially three different ways that we can argue.

The first is to give a *low-level* description, where we carefully exhibit all the states and transitions of the machine. This is what we did in showing that the set of strings containing equal numbers of a's and b's is decidable—we wrote down every detail of the Turing machine that decided it.

As we grow more accustomed to the sort of basic bookkeeping tasks that Turing machines are capable of, we can content ourselves with *higher-level* descriptions. Here we don't give detailed lists of states and transitions, and instead use

language like, 'move right until you have seen three occurrences of b and change the last one to c'. This is what we did when we proved above that a two-tape TM can be simulated by a one-tape TM. You have to be careful in this kind of informal argument not to assert too much–it should always be clear how to go about converting this into a low-level description, even if the actual details are quite onerous.

The third is to not talk about tapes or states or reading heads at all, but to implicitly assume the Church-Turing thesis: As long as we know that there is a pencil-and-paper algorithm (even if it takes a very long time) for transforming the input to the output, then we can safely suppose that there is a Turing machine that does the same thing.

As an example of the different approaches, we'll give a couple of different proofs of an important fact:

Theorem 2 *A language $L \subseteq \Sigma^*$ is decidable if and only if both L and $\Sigma^* \backslash L$ (the complement of L) are Turing-recognizable.*

This is an 'if and only if' statement, so there are two things to prove. First let's do the easy direction. If L is decidable, then it is by definition Turing-recognizable. Further, if we just reverse the accept and reject states of a machine deciding L, we see that $\Sigma^* \backslash L$ is decidable, so it too is Turing-recognizable

The hard direction is the converse. We have to show that if there is a TM $\mathcal{M}_1$ that recognizes L, and a TM $\mathcal{M}_2$ that recognizes the complement of L, then there is a TM $\mathcal{M}$ that decides L.

A low-level proof of this would take the specifications of $\mathcal{M}_1$ and $\mathcal{M}_2$ and use them to produce a specification of $\mathcal{M}$. We won't carry this out.

Here is a higher-level proof: Since we have shown that any language decided by a two-tape Turing machine is decidable in the original sense, it is sufficient to describe a two-tape machine that decide L. The machine works as follows: In its initial phase, it copies its input on the first tape to the second tape. It then essentially runs $\mathcal{M}_1$ and $\mathcal{M}_2$ in parallel: Each state of the 2-tape machine is a pair consisting of a state of $\mathcal{M}_1$ and a state of $\mathcal{M}_2$. At each step, the machine executes the applicable transition of $\mathcal{M}_1$ on the first tape, and the transition of $\mathcal{M}_2$ on the second tape. The machine halts if either of the component machines enters a halted state.

Note that we are guaranteed to enter a halted state on one of the machines, because any $w \in \Sigma^*$ will be accepted by $\mathcal{M}_1$ or accepted by $\mathcal{M}_2$, but not both. Thus we know by the result whether the input string is in L or not, and we can use

this information to switch $\mathcal{M}$ into the accept or reject state. (It is possible that, say, $\mathcal{M}_2$, will enter the *reject* state: if this happens, then we know the input is in$L$, so we can switch $\mathcal{M}$ into the accept state.

And here is a very high-level version of the same argument: We have an algorithm for recognizing L, which answers 'yes' if the input is in L, but may not answer for inputs not in L. And we have a second algorithm that answers 'yes' if the input *isn't* in L. We make two separate copies of the input and run a step of one algorithm in the first copy, then a step of the second in the other copy, alternating in this way until one of the algorithms gives an answer. Since we are guaranteed an answer in one of them, this algorithm decides whether or not the input is in L.