CS 3719 (Theory of Computation and Algorithms) – Lecture 4

Antonina Kolokolova[*]

January 16, 2013

A Turing machine M *recognizes* a language L if it accepts all and only strings in L: that is, $\forall x \in \Sigma^*$, M accepts x iff $x \in L$. As before, we write $\mathcal{L}(M)$ for the language accepted by M.

Definition 13. *A language L is called* Turing-recognizable *(also* recursively enumerable, r.e*, or* semi-decidable*) if $\exists$ a Turing machine M such that $\mathcal{L}(M) = L$.*

A language L is called decidable *(or* recursive*) if $\exists$ a Turing machine M such that $\mathcal{L}(M) = L$, and additionally, M halts on all inputs $x \in \Sigma^*$. That is, on every string M either enters the state q_{accept} or q_{reject} in some point in computation.*

It is possible to define a Turing machine producing an output; in that case, the Turing machine halts in an accepts state, with the tape clear except for the output and head pointing to the first symbol of the output.

Remark 1. One reason that these languages are called "recursively enumerable" is that it is possible to "enumerate" all strings in such a language by a special type of Turing machine, enumerator. This Turing machine starts on blank input and runs forever; as it runs, it outputs (e.g., on some additional tape or part of the main tape) all the strings in the language. It is allowed to print the same string multiple times, as long as it does not print anything not in the language and eventually print every string that is in the language. See Sipser's book for a formal definition and a proof of equivalence.

0.1 Multi-tape Turing machine

Often it is convenient to describe a Turing machine using several tapes, each dedicated to a specific function. For example, if we want to simulate a random-access machine, we can have a dedicated "address tape" where the machine writes the address of the cell it wants to visit next. Let us define a k-tape Turing machine to have k tapes, with a separate head on each

[*]The material in this set of notes came from many sources, in particular "Introduction to Theory of Computation" by Sipser and course notes of U. of Toronto CS 364. Many thanks to Richard Bajona for sharing his LaTeX'ed notes!

tape, and a transition function $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$. That is, now for every state there are k characters the heads are pointing to, and, respectively, k-tuple of characters to overwrite the current ones and k-tuple of directions, one for each tape. Note that there is just one state, not k.

Lemma 15. *Multi-tape Turing machines recognize and decide exactly the same class of languages as the one-tape Turing machines.*

Proof sketch. Here we will show that this definition is equivalent to the traditional one-tape Turing machine. Recall that to show such an equivalence between two models it is enough to show how to construct a description of one given the other, and vice versas. That is, given a description of machine $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\rangle$ of the first type (in our case, a multi-tape Turing machine) we would like to construct a machine $M' = \langle Q', \Sigma, \Gamma', \delta', q'_0, q'_{accept}, q'_{reject}\rangle$ of the second type (in this case, the usual Turing machine), which compute the same languages ($\mathcal{L}(\mathcal{M}) = \mathcal{L}(\mathcal{M}')$).

First, let's look at the easy direction. Suppose M is the usual Turing machine (assume, without loss of generality, that in both cases we talk about two- way infinite tape machines). An equivalent k-tape machine M' would essentially ignore the content of all tapes except for the first tape containing the input. More precisely, we construct $M' = (Q', \Sigma, \Gamma', \delta', q'_0, q'_{accept}, q'_{reject})$ to have $Q' = Q, \Gamma' = \Gamma, q'_0 = q_0, q'_{accept} = q_{accept}, q'_{reject} = q_{reject}$ and δ' defined as follows. For every transition $(q_i, a \to q_j, b, D)$, of M, where $D \in \{L, R\}$ is the direction, we will put in δ' a transition $(q_i, \langle a, \sqcup, \ldots, \sqcup\rangle) \to (q_j, \langle b, \sqcup, \ldots, \sqcup\rangle, \langle D, R, \ldots R\rangle)$ of a k- tape Turing machine M'. Here, the choice of direction where the heads on the other tapes go does not matter: all L, or all D would work just as well. To make it more complete, though, it would be prudent to add such a transition for every tuple $\langle a, a_2, \ldots, a_k\rangle$, $a_2, \ldots, a_k \in \Gamma'$. Note that this will make the size of δ' equal to $|\delta| * |\Gamma|^{k-1}$ (e.g. for a two-tape Turing machine think of its having a copy of a transition from δ for every possible symbol from Γ on the second tape.) Now, the construction is complete. Notice that here, as an added bonus, the running time (number of configurations it goes through on a given input) of M' is exactly the same as the running time of M, and so if M halts on a given input, then so does M'.

The other direction is, understandably, a bit more complicated: intuitively, multi-tape Turing machines promise more power by the virtue of having those extra tapes. There are several ways to do this simulation, similar to two ways of simulating a two-way tape Turing machine by a one-way tape TM. In the first case, we write strings corresponding to (non-blank part of) the tapes one after another, with a delimiter between strings on different tapes. Then, whenever a new symbol needs to be added, the rest of the tapes is shifted (just like putting a new symbol to the left of the used part of the tape in simulating two-way tape TM by one-way). Here, we use special symbols ("marked" versions of symbols of Γ) to represent a symbol with head pointing to it. This is the way it is presented in Sipser's book, as well.

Another, a more interesting way is to expand the alphabet to make a symbol of each k-tuple (for k tapes) of symbols of the original alphabet (also with marked versions of symbols of Γ). This is similar to the second way of simulating two-way tape TM, where we used new symbols for the pairs of symbols from Γ; again, we need markers for the head positions.

In both cases, the Turing machine works by scanning all tape from the beginning to the end noting head positions and symbols under them. On the way back, it makes the changes according to the transition table.

Let's look at the first construction in a little more detail. There are several parts of the simulation. First, M' would have an extra preparatory stage where it adds special symbols to its tape to encode the other tapes. So as opposed to the other direction, our Γ' will have a number of new symbols: $\Gamma' = \Gamma \cup \{\#\} \cup \{$set of all symbols from Γ with a dot over them $\}$ plus a few extra symbols that will come up in the construction. For example, if the original $\Gamma = \{0, 1, \sqcup\}$, then $\Gamma' = \{0, 1, \sqcup, \#, \dot{0}, \dot{1}, \dot{\sqcup}\}$ together with a few extra ones.

Then, we need to describe what M does to simulate a single transition of M. Also, it is useful to have a "subroutine" for "shifting" the content of the tape to the left when we access a previously untouched blank cell of the tape.

So the first thing M does is to prepare the tape. It starts with the tape with just the input; it wants to put in delimiters between (and, say, to the ends) of the tapes, and put a special "dotted" symbol into each tape representing that this is where the head points. So from q_0' rather than doing the first transition of M, say M' first overwrites the symbol it points to with a "the head is here" version (that is, over binary alphabet, if the first symbol is 0 then the first transition executed is $(q_0, 0) \to (q_{p_1}, \dot{0}, L)$ where q_{p_1} is the first state of the preparation stage. Then change the blank before the input to #; this can be accomplished with transition $(q_{p_1}, \sqcup) \to (q_{p_2}, \#, R)$. Then go to the right in state q_{p_2} until hitting a blank; overwrite the blank with #, go to q_{p_3}, write $\dot{\sqcup}$, go to q_{p_4}, wrtite # and keep going like this until there are $k+1$ symbols # on the tape. Finally, switch to $q_{p_{back}}$ and go to the beginning until hitting a blank, then right. Now it is ready to start simulating the first transition of M. All those q_{p_i} will be put in Q', and the corresponding transitions in δ'. Exercise: how many states exactly are needed for this stage? And how many transitions in δ'?

The hardest part is simulating one transition of M, $(q_i, \langle a_1, \dots, a_k\rangle) \to (q_j, \langle b_1, \dots, b_k\rangle, \langle D_1, \dots, D_k\rangle)$. The intuition is as follows. M' will scan the input starting from the beginning, 'slowly learning' the k-tuple of symbols with dots over them (that is, the symbols being read). It remembers this information by using states: suppose it is reading the 3rd tape, and just read over a symbol a_3 with a dot there. Then it will switch from a state q_{i,a_1,a_2} to q_{i,a_1,a_2,a_3}. When it finally hits the blank (end of tape), it will be in some state $q_{i,a_1,\dots,a_k}$ encoding the fact that it is simulating a transition from $(q_i, \langle a_1, \dots, a_l\rangle)$ in δ. Now it will go backwards (sometimes with one extra step to the right), updating every tape (and using a number of additional states to do that). When it is done updating all tapes and gone to the beginning of the tape, it can switch to a state q_j. Note that although there is a large number of states introduced, this number depends only on the number of tapes and number of symbols in Γ', both of which are finite, so our new Q' and δ' are still finite. But we could not, for example, use states to remember the position of the head on a tape.

□

Note that for some problems multi-tape Turing machine is faster than single-tape. For example, the Palindrome problem can be solved in linear time on a multi-tape Turing machine by copying the string from input tape to the second tape, then moving one of the heads to the beginning of the tape, and finally moving the heads in opposite direction on two tapes comparing the numbers. It is possible to show, though, that this is about as bad as it gets: the simulation above runs in roughly O of the square of the time of the original Turing machine.

0.2 Non-deterministic Turing machines

Recall from the previous lecture that in non-deterministic computation, the transition function δ, rather than giving exactly one choice for the next step, gives a (possibly empty, but finite) set of choices. So the non-deterministic computation is not a sequence, but rather a tree, and it is successful (accepting) if at least one leaf is accepting. We call one branch of this tree a computational path. Each such path can be described by a sequence of choices the Turing machine made. The arity d of the tree is the maximum over all $q \in Q, a \in \Gamma$ of $|\delta(q,a)|$. So each node in the tree can be described by a sequence of numbers each from 1 to d saying which transition was chosen from a set (in, say, lexicographic order). Of course, some sequences do not correspond to any node, but it is OK as long as any node has a unique sequence describing it.

Example 1. Consider the following non-deterministic Turing machine which tries to determine if its input contains a symbol 1. It will scan the input from left to right; when it sees a 1, it would either accept or continue scanning as if nothing has happened.

State q	*Symbol s*	*Action $\delta(q,s)$*
q_0	0	$(q_0, 0, R)$
q_0	1	$(q_0, 1, R)$
q_0	1	$(q_{accept}, \not b, R)$
q_0	$\not b$	$(q_{reject}, \not b, R)$

Now, on input 1101, there are four possible computation paths:

1) $q_0 1101 \to 1 q_a 101$
2) $q_0 1101 \to 1 q_0 101 \to 11 q_a 01$
3) $q_0 1101 \to 1 q_0 101 \to 11 q_0 01 \to 110 q_0 1 \to 1101 q_a$
4) $q_0 1101 \to 1 q_0 101 \to 11 q_0 01 \to 110 q_0 1 \to 1101 q_0 \to 1101 q_r$

Since at least one of them (three, actually) is accepting, this Turing machine accepts 1101.

Example 2. Consider a Turing machine for testing if a number is composite (not prime) that works as follows. It goes to the end of the input, puts some separator there, and starts writing out a number: in the corresponding state q, on every step, it can either write a 1 and move right keeping the state q, or write a 0 and move right also remaining in q, or write a blank, change the state to some q' and start moving backward. At this point, there are two numbers written on the tape, and the Turing machine checks if the new number is ≥ 2, not equal to the input and divides the input; if so, it accepts, otherwise rejects. That is, the machine "guesses" a number, and checks if this is a correct divisor of the input. Now, if the number is composite, then there would be some number that divides it; the corresponding computation branch will end in Accept, and therefore, the non-deterministic Turing machine accepts. If the number is prime, then no branch will accept. Note that in this case there are branches that are infinite: imagine a sequence of choices that never chooses the third possibility.

We will show here that it is possible to simulate a non-deterministic Turing machine by a deterministic one, although this simulation is not efficient. This efficiency issue is one of the main open problems in theoretical computer science, the famous P vs. NP question, whether efficient non-deterministic Turing machine computation can always be simulated by an efficient deterministic one. We will define precisely what we mean by efficient in a few lectures.

In order to simulate non-deterministic computation by deterministic we need to find if there is an accepting leaf in its computation tree. Since it is a Turing machine, there may be infinite branches in the tree (corresponding to infinite loops of the Turing machine), so we cannot do a depth-first search of this tree. However, we can do a breadth-first search. If we don't care at all about efficiency, we can do the simulation by just remembering which node we were in last, and restarting the computation from the beginning, making non-deterministic choices according to the path to the next node.

For simplicity, let's use the multi-tape Turing machine we just defined. Suppose our TM has three tapes: an input tape on which it will not write, a work tape, and a "address" tape, which in this case will keep track of the current computation branch. The address of a node here is a the sequence of choices that led to this node. The machine will work as follows: at every stage of the computation, starting with writing 0 on the address tape, it will run the computation from the start to the next non-deterministic choice following the sequence of choices written on the address tape. When it reaches a choice not on the tape, it will increment the address tape to the next node in the breadth-first order, and restart the computation. If one of the choices leads to a non-existing transition or reject state, abort the computation, increment the node and restart. If q_{accept} is reached, accept.

Since this Turing machine will eventually get to all nodes in every level, if there is an accepting leaf it will be found. Otherwise it will run forever. A check can be added to see if all nodes on the last level aborted and then reject.