CS 3719 (Theory of Computation and Algorithms) – Lecture 17

Antonina Kolokolova*

February 15, 2011

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 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.

Here we will show that this definition is equivalent to the traditional one-tape Turing machine. 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.

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.

*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.

0.2 Non-deterministic Turing machines

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.

Recall that non-deterministic Finite Automata recognize the same class of languages as deterministic ones; however, for pushdown automata there are languages that cannot be recognized without non-determinism. Which of the two cases holds for the Turing machines?

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.