CIS 511: Theory of Computation **Jan 26, 2017**

Lecture 5: Nondeteminism in Turing Machines

Professor Sampath Kannan *Zach Schutzman*

NB: *These notes are from CIS511 at Penn. The course followed Michael Sipser's Introduction to the Theory of Computation (3ed) text.*

5.1 Nondeterministic Turing Machines

What if our TMs can explore a variety of possible transitions simultaneously.

Definition 5.1 *A **non-deterministic Turing Machine** (NTM) is a Turing machine M where $\delta_M : Q \times \Gamma \to 2^{Q\times\Gamma\times\{L,R\}}$.*

Why do we care?

Claim 5.2 *NTMs are equivalent in power to regular TMs.*

Theorem 5.3 *If M is a NTM recognizing language L, then there exists a deterministic TM D recognizing L.*

Proof:

The idea is to have D simulate the behavior of M. We can think of M's computation as a tree of configurations, branching when it makes non-deterministic choices. We might be tempted to fully explore each branch sequentially, but we have to be careful as there may be some branches that loop forever, and we don't want to get stuck on these.

Our solution will be to explore in a breadth-first manner. This way, if there is some accepting path, we will eventually find it without getting stuck in an infinite loop.

Formally, let D be a 3-tape Turing machine. The first tape will be a read-only input tape. The second will be a work tape that actually simulates M. The third tape will be used to address where we are in the configuration tree. We can upper-bound the number of children a configuration can have by B by observing that at each transition, there are no more than B possible resultant configurations. Tape 3 will store address values $x_1 \dots x_k$, which tell you, from the root, which child to explore. We will skip over invalid addresses.

Begin by copying the initial configuration from Tape 1 to Tape 2. Then look at Tape 3 and get x_1, then process that, then do it for x_2, until we reach x_k. When we reach the end, increment the value on Tape 3 and start over. If we reach q_r, we terminate that branch. If we reach q_a on some branch, we accept.

This runs slowly, but it deterministically simulates M, and therefore recognizes L.

■

Corollary 5.4 *Turing machines can simulate other Turing machines.*

The Turing machine was 'invented' by Alan Turing partly as a result of one of Hilbert's questions of finding solutions to Diophantine equations. Turing wanted to figure out how to show that no possible algorithm exists.

Meanwhile, Alonzo Church created the λ-calculus, which is a functional view of computability that is equivalent to Turing's algorithmic view, proved by Church and Turing. These are both equivalent to something in logic called partial recursive functions.

*Is there a stronger model of computability? The **Church-Turing Thesis** proposes that anything computable by any physical device is computable under the notions of Turing machines or λ-calculus. Presently, there has not been any evidence that there exists a physical device that is more powerful than a Turing machine.*

5.2 Turing Machine Computations = Algorithms

Imagine a Turing machine E hooked up to a printer (write-only output tape)

Definition 5.5 *Such a Turing machine E **enumerates** a language L if for each $x \in L$, E eventually outputs x and for each $y \notin L$, E never outputs y.*

Theorem 5.6 *A language L is recognizable by a TM if and only if there exists some TM that enumerates L.*

Proof:

Suppose L is recognized by a machine by a TM M. We want to build an enumerator E for L. Order the strings in Σ^* by length then by dictionary order (for example). In order, for each $x \in \Sigma*$, run M on x, and if M accepts, print x. We need to be careful about looping forever on some string. We will take a variable k and increment it as E runs. For $k = 1, 2 \ldots, \infty$, run M on each of the first k strings for k steps, printing those that M accepts on.

To see that for every string $y \in L$, we know that y is the i^{th} string in language for some finite i, so for some $k = j$, M will run on y. Since M accepts y in a finite amount of steps, less than or equal to the $\max\{i, j\}$. Since this is finite, E eventually outputs y.

Now suppose L can be enumerated by an enumerator E. We want to build a machine M to recognize L. This is easy. M does the following: given a string x, every time E prints a string, compare it to x. If these values are equal, accept.

E never prints a string not in L, so M loops forever on strings not in L, which is fine.

*The proof is complete and we can say the class of languages recognized by Turing machines is exactly the **recursively enumerable languages**.*

■

Corollary 5.7 *A language is decidable if and only if the enumerator prints the strings in the lexicographic order.*

5.3 Decidable Languages

A language is decidable if there is a Turing machine that recognizes it and halts on all inputs.

Let's think about the problem of finding a minimum spanning tree in a weighted graph. We have an efficient algorithm that always terminates to do this. How do we translate a problem into a language?

Viewing this as a language, we can encode the graph G and tree T as strings and define the language as $L = \langle G, T \rangle$ such that T is the MST of G. Alternatively, we can encode G and an integer K and define the language as $L = \langle G, K \rangle$ such that there exists a MST of G with weight at most K.