CMPSCI 250: Introduction to Computation

Lecture #38: Turing Machine Semantics
David Mix Barrington
27 April 2012

Turing Machine Semantics

- Review: A Turing Machine Example
- Turing Recognizable Languages
- Turing Decidable Languages
- The TR/TD Theorem
- Running Two Machines in Parallel
- Multitape Turing Machines
- Nondeterministic Turing Machines

A Turing Machine Example

- Here is a machine that solves a problem that a DFA cannot. When started in configuration i□$w_1w_2...w_n$, it will halt if and only if w is in the language {a^nb^n: n ≥ 0} -- otherwise it will hang.

- With input aabb we get i□aabb, □paabb, □□qabb, □□aqbb, □□abqb, □□abbq□, □□abrb, □□asb, □□sab, □s□ab, □□pab, □□□qb, □□□bq□, □□□rb, □□s□, □□□p□, □□□h□. The string aabb is **accepted**.

```
In i: Move R and go to p.
In p: On □, go to h.  On b, move L and go to z.
      On a, print □, move R, and go to q.
In q: On a or b, move R and stay in q.  On □, move L and go to r.
In r: On a or □, move L and go to z.
      On b, print □, move L, and go to s.
In s: On a or b, move L and stay in s. On □, move R and go to p.
In h: Halt (final state).
In z: Move left and stay in z.
```

Turing Recognizable Languages

- We've seen that a Turing machine, when started on a string, may or may not ever reach a final state. We define the **language of the machine** M, called L(M), to be the set of strings on which M eventually halts.

- If a language X is equal to L(M) for some Turing machine M, we say that X is **Turing recognizable**. The idea is that the machine "recognizes" strings in the language by halting, but gives no output on strings not in the language.

- In the example earlier, the language of our TM is $\{a^n b^n : n \geq 0\}$, a language that we now know is not regular.

- A recognizer for a language is not all that useful in practice, because we would like to know whether *or not* the string is in the language, and the recognizer makes us wait forever on the "no" strings. We can't always recognize that it is in a loop, because it might just use more and more tape.

Turing Decidable Languages

- Suppose, though, that the machine always halts, but halts in an **accepting state** when the input is in the language and in a **rejecting state** when it is not. We redefine L(M) to be the set of strings that are accepted. In this case we say that the language L(M) is **Turing decidable**. Remembering the Church-Turing thesis, a decidable language is one where the decision problem can be solved by an algorithm, as long as we give the algorithm enough time and memory.

- Every Turing decidable language is also Turing recognizable. If we have a decider M for a language X, we can modify M to make a recognizer for X. We replace the rejecting final state of M with a state in which the machine always moves right, like the state "z" of our example. Now this new machine M′ halts on input w if and only if M accepts w, which is true if and only if $w \in X$.

- We'll show next time that there exist TR languages that are not TD.

The TR/TD Theorem

- The **TR/TD Theorem** says that a language X is Turing decidable if and only if both X and its complement X-bar are Turing recognizable.

- One half of the proof is easy. If X is decidable, so is X-bar (by exchanging the accepting and rejecting states, as we did for DFA's). By the proof on the last slide, both X and X-bar must also be recognizable.

- For the other half, we assume that both X and X-bar are recognizable. So we have two Turing machines M and N, with X = L(M) and X-bar = L(N). We need to build a decider for X.

- But if we have an arbitrary string w, we can't run either M or N on it and be confident of getting an answer. Strings in X will cause N to run forever, and strings not in X will cause M to run forever. Connecting M and N in series will not work.

Running Two Machines in Parallel

- The trick is to run both M and N on w **in parallel**. When our decider starts in configuration i□w, the first thing it does is to make a new copy of w to the right of the original one, so that the tape contents become @□w#@□w. The left copy of w represents the initial configuration of M, and the right copy is the initial configuration of N. The two @ symbols mark the head positions.

- We now want the decider to run M and N for one step each on their respective tapes. We make a sweep of the tape. When the machine finds the first @, it changes the next letter and moves the @ left or right, based on what M should do in its current state seeing that letter. Then it continues to the right and does the same thing for the other @, based on what N should do in its current state seeing that letter. It must remember both an M-state and an N-state in its own state.

- One slight complication -- if the character to the right of the first @ is #, the machine must make space for a new blank by shifting everything to the right.

Multitape Turing Machines

- We can use a similar idea to show that a **multitape Turing machine** can be simulated by an ordinary one. A Turing machine with k tapes has a transition function from $Q \times \Gamma^k$ to $(\Gamma \times \{L, R\})^k$. On any step, based on its current state and the letters it sees at the head on each tape, it writes a letter and moves left or right on each tape.

- With an ordinary TM, we use our single tape to store the k different tapes in series, with # symbols between every pair of adjacent tape contents and @ symbols to mark each head position. To simulate a move of the multitape machine, our ordinary machine sweeps the tape left to right, remembers the character after each @, then goes back, implements the write and move operations for each head, and updates its state.

- Other data structures for the Turing machine's memory can also be simulated by single tapes.

Nondeterministic Turing Machines

- A **nondeterministic Turing machine** or **NDTM** is like an ordinary deterministic Turing machine (or DTM) except that for a given state and letter seen, it may have no options, one option, or more than one option as to the character it writes and the direction it moves.

- Nondeterminism did not make NFA's able to decide languages that DFA's could not, though the smallest NFA for a language might have many fewer states than the smallest DFA. In the case of Turing machines, we will see that nondeterminism again does not help qualitatively though it might help quantitatively. Every NDTM has an equivalent DTM with the same language, but the NDTM might take much less time to recognize a given string.

- The **P versus NP problem** asks whether every **polynomial-time** NDTM can be simulated by a polynomial-time DTM. The answer is probably "no", but proving this is the most famous unsolved problem in computer science.

Simulating an NDTM With a DTM

- To simulate an NDTM with a DTM, we first build a DTM with three tapes. The first tape will store the input string w and will never change. The second will be a work tape to exactly simulate a particular computation of the NDTM. The third tape will hold a **choice sequence**, which is a string of symbols telling the NDTM which of its options to take on each of its moves.

- The input w is in the language of the NDTM N if and only if there *exists* a choice sequence that causes N to halt, starting from i□w. So our simulation tests all possible choice sequences, starting with the one of length 0, then all the ones of length 1, then length 2, and so forth. For each choice sequence, the DTM clears its work tape, copies w onto it, then runs N using the sequence. If the simulated N ever halts, the DTM accepts w.

- So if $w \in L(N)$, the DTM will eventually reach a good choice sequence and will accept w. If $w \notin L(N)$, the DTM will run forever because it will keep trying longer and longer choice sequences.