Lecture 19: Dovetailing and non-deterministic Turing machines

6 April 2010

This lecture covers dovetailing, a method for running a gradually expanding set of simulations in parallel. We use it to demonstrate that non-deterministic TMs can be simulated by deterministic TMs.

1 Dovetailing

1.1 Interleaving

We have seen that you can run two Turing machines in parallel, to compute some function of their outputs, e.g. recognize the union of their languages.

Suppose that we had Turing machines $M_1, \ldots, M_k$ recognizing languages $L_1, \ldots, L_k$, respectively. Then, we can build a TM M which recognizes $\bigcup_{i=1}^{k} L_i$. To do this, we assume that M has k simulation tapes, plus input and working tapes. The TM M cycles through the k simulations in turn, advancing each one by a single step. If any of the simulations halts and accepts, then M halts and accepts.

We could use this same method to run a single TM M on a set of k input strings $w_1, \ldots, w_k$; that is, accept the input list if M accepts any of the strings $w_1, \ldots, w_k$.

The limitation of this approach is that the number of tapes is finite and fixed for any particular Turing machine.

1.2 Interleaving on one tape

Suppose that TM M recognizes language L and consider the language

$$\widehat{L} = \left\{ w_1 \# w_2 \# \ldots \# w_k \;\middle|\; M \text{ accepts } w_k \text{ for some } k \right\}.$$

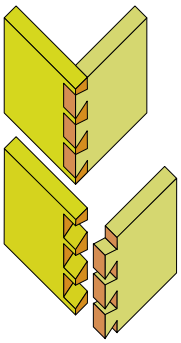
The language $\widehat{L}$ is recognizable, but we have to be careful how we construct its recognizer $\widehat{M}$. Because M is not necessarily a decider, we can not process the input strings one after another, because one of them might get stuck in an infinite loop. Instead, we need to run all k simulations in parallel. But k is different for different inputs to $\widehat{M}$, so we can not just give k tapes to $\widehat{M}$.

Instead, we can store all the simulations on a single tape $\textcircled{\scriptsize \#}_T$. Divide up

$$\textcircled{\scriptsize \#}_T$$

into k sections, one for each simulation. If a simulation runs out of space in its section, push everything over to the right to make more room.

1.3 Dovetailing



Dovetailing (in carpentry) is a way of connecting two pieces of wood by interleaving them, see picture on the right.

Dovetailing is an interleaving technique for simulating many (in fact, infinite number of) TM together. Here, we would like to interleave an infinite number of simulations, so that if any of them stops, our simulation of all of them would also stop.

Consider the language:

$$J = \left\{ \langle M \rangle \;\middle|\; M \text{ accepts at least one string in } \Sigma^* \right\}.$$

It is tempting to design our recognizer for J as follows.

```
algBuggyRecog(⟨M⟩)
  x = ε
  while True do
        simulated M on x (using U_TM)
        if M accepts then
              halt and accept
        x ← next string in lexicographic order
```

Unfortunately, if M never halts on one of the strings, this process will get stuck before it even reaches the string that M does accept. So we need to run our simulations in parallel. Since we can not start up an infinite number of simulations all at once, we use the following idea.

Dovetailing is the idea of running k simulations in parallel, but keep dynamically increasing k. So, for our example, suppose that we store all our simulations on tape $\textcircled{\scriptsize 3}_T$ and x lives on some other tape. Then our code might look like:

```
algDovetailingRecog(⟨M⟩)
  x = ε
  while True do
        On ③_T, start up the simulation of M on x
        Advance all simulations on ③_T by one step.
        if any simulation on ③_T accepted then
              halt and accept
        x ← Next(x)
```

Here $\textbf{Next}(x)$ yields the next string in the lexicographic ordering.

In each iteration through the loop, we only start up one new simulation. So, at any time, we are only running a finite number of simulations. However, the set of simulations keeps expanding. So, for any string $w \in \Sigma^*$, we will eventually start up a simulation on w.

1.3.1 Increasing resource bounds

The effect of dovetailing can also be achieved by running simulations with a resource bound and gradually increasing it. For example, the following code can also recognize J.

Increasing resource bound

for $i = 0, 1, \ldots$

(1) Generate the first i strings (in lexicographic order) in Σ^*

(2) On tape T, start up simulations of M on these i input strings

(3) Run the set of simulations for i steps.

(4) If any simulation has accepted, halt and accept

(5) Otherwise increment i and repeat the loop

Each iteration of the loop does only a finite amount of work: i steps for each of i simulations. However, because i increases without bound, the loop will eventually consider every string in Σ^* and will run each simulation for more and more steps. So if there is some string w which is accepted by M, our procedure will eventually simulate M on w for enough steps to see it halt.

2 Nondeterministic Turing machines

A ***non-deterministic Turing machine*** (NTM) is just like a normal TM, except that the transition function generates a set of possible moves (not just a single one) for a given state and character being read. That is, the output of the transition function is a set of triples (r, c, D) where r is the new state, c is the character to write onto the tape, and D is either L or R. That is

$$\delta(q, \mathtt{c}) = \left\{ (r, \mathtt{d}, D) \;\middle|\; \text{for some } r \in Q, \mathtt{d} \in \Gamma \text{ and } D \in \{\mathtt{L}, \mathtt{R}\} \right\}.$$

An NTM M accepts an input w if there is some possible run of M on w which reaches the accept state. Otherwise, M does not accept w.

This works just like non-determinism for the simpler automata. That is, you can either imagine searching through all possible runs, or you can imagine that the NTM magically makes the right guess for what option to take in each transition.

For regular languages, the deterministic and non-deteterministic machines do the same thing. For context-free languages, they do different things. We claim that non-deterministic Turing machines can recognize the same languages as ordinary Turing machines.

2.1 NTMs recognize the same languages

Theorem 2.1 *NTMs recognize the same languages as normal TMs.*

Suppose that a language L is recognized by an NTM M. We need to construct a deterministic TM that recognizes L. We can do this using dovetailing to search all possible choices that the NTM could make for its moves.

Our simulation will use two simulation tapes S and T in a similar way. In this case, flicker isn't a big problem. But the double buffering makes the algorithm slightly easier to understand.

simulate NTM M on input w

(1) put the start configuration q_0w onto tape S

(2) for each configuration C on S
   - generate all options $D_1, D_2, \ldots D_k$ for the next configuration
   - if any of $D_1, D_2, \ldots D_k$ is an accept configuration, halt and accept
   - otherwise, erase all the D_i that have halted and rejected
   - copy the rest onto the end of T

(3) if tape T is empty, halt and reject

(4) copy the contents of T to S, overwriting what was there

(5) erase tape T.

(6) goto step 2

You can think about the set of possible configurations as a tree. The root is the start configuration. Its children are the configurations that could be reached in one step. Its grandchildren are the configurations that could be reached in two steps. And so forth. Our simulator is then doing a breadth-first search of this tree of possibilities, looking for a branch that ends in acceptance.

2.2 Halting and deciders

Like a regular TM, an NTM can cleanly reject an input string w or it can implicitly reject it by never halting. An NTM halts if all possible runs eventually halt. Once it halts, the NTM accepts the input if some run ended in the accept state, and rejects the input if all runs ended in the reject state.

A NTM is a ***decider*** if it halts on all possible inputs on all branches. Formally, if you think about all possible configurations that a TM might generate for a specific input as a tree (i.e., a branch represents a non-deterministic choice) then an NTM is a decider if and only if this tree is finite, for all inputs.

The simulation we used above has the property that the simulation halts exactly if the NTM would have halted. So we have also shown that

Theorem 2.2 *NTMs decide the same languages as normal TMs.*

2.3 Enumerators

A language can be ***enumerated***, if there exists a TM with an output tape (in addition to its working tape), that the TM prints out on this tape all the words in the language (assume that between two printed words we place a special separator character). Note, that the output tape is a write only tape.

Definition 2.3 (Lexicographical ordering.) For two strings s_1 and s_2, we have that $s_1 < s_2$ in lexicographical ordering if $|s_1| < |s_2|$ or $|s_1| = |s_2|$ then s_1 appears before s_2 in the dictionary ordering.

(That is, lexicographical ordering is a dictionary ordering for strings of the same length, and shortest strings appear before longer strings.)

Claim 2.4 *A language* L *is TM recognizable iff it can be enumerated.*

Proof: Let T the TM recognizer for L, and we need to build an enumerator for this language. Using dovetailing, we "run" T on all the strings in $\Sigma^* = \{w_1, w_2, \ldots\}$ (say in lexicographical ordering). Whenever one of this executions stops and accepts on a string w_i, we print this string w_i to the enumerator output string. Clearly, all the words of L would be sooner or later printed by this enumerated. As such, this language can be enumerated.

As for the other direction, assume that we are given an enumerate T_{enum} for L. Given a word $x \in \Sigma^*$, we can recognize if it is in L, by running the enumerator and reading the strings it prints out one by one. If one of these strings is x, then we stop and accept. Otherwise, this TM would continue running. Clearly, if $x \in$ L sooner or later the enumerator would output x and our TM would stop and accept it. ■

Claim 2.5 *A language* L *is decidable iff it can be enumerated in lexicographic order.*

Proof: If L is finite the claim trivially hold, so we assume that L is infinite.

If L is decidable, then there is a decider **deciderForL** for it. Generates the words of Σ^* in lexicographic ordering, as $w_1, w_2, \ldots$. In the ith stage, check if $w_i \in$ L by calling **deciderForL** on w_i. If **deciderForL** accepts w_i then we print it to the output tape. Clearly, this procedure never get stuck since **deciderForL** always stop. More importantly, the output is sorted in lexicographical ordering. Note, that if $x \in$ L, then there exists an i such that $w_i = x$. Thus, in the ith stage, the program would output x. Thus, **deciderForL** indeed prints all the words in L.

Similarly, assume we are given a lexicographical enumerator T_{enum} for L. Consider a word $x \in \Sigma^*$. Clearly, the number of words in Σ^* that appear before x in the lexicographical ordering is finite. As such, if x is the ith word in Σ^* in the lexicographical ordering, then if T_{enum} outputs i words and none of them is x, then we know it would never x, and as such x is not in the language. As such, we stop and reject. Similarly, if x is output in the first i words of the output of T_{enum} then we stop and accept. Clearly, since T_{enum} enumerates an infinite language, it continuously spits out new strings. As such, sooner or later it would output i words of L, and at this point our procedure would stop. Namely, this procedures accepts the language L, and it always stop; namely, it is a decider for L. ■