CSE 105: Introduction to the Theory of Computation Fall 2015

Lecture Notes: Finite State Transducers

Instructor: *Daniele Micciancio* UCSD CSE

This lecture notes are provided as a supplement to the textbook. In the exercises/problems section of Chapter 1, the textbook defines Finite State Transducers (FST) as deterministic automata that at each step read one input symbol $a \in \Sigma$ and write one output symbol $b \in \Gamma$. In these notes we define a more general form of FST that can output arbitrary strings in each state. We also consider non-deterministic FST. Both extensions are often useful in applications.

1 Defining FST

Definition 1 *A Finite State Transducer (FST) is a 5-tuple $T = (Q, \Sigma, \Gamma, \delta, s, \gamma)$ where*

- *Q is a finite set of states,*
- *Σ is a finite set of input symbols,*
- *Γ is a finite set of output symbols,*
- *$\delta: Q \times \Sigma \to Q$ is the transition function,*
- *$s \in Q$ is the start state.*
- *$\gamma: Q \to \Gamma^*$ is the output function.*

Our definition of FST is similar to that of a DFA, with the following differences:

- The FST includes not only an input alphabet Σ, but also an output alphabet Γ. Using different alphabets for input and output may be used to define transducers that convert between different alphabets.
- Instead of a set of accepting states F, an FST as an output function $\gamma: Q \to \Gamma^*$

Just like a DFA, an FST on input $w \in \Sigma^*$ goes through a sequence of states $q_0 = s$, $q_1 = \delta(q_0, w_1)$, ..., $q_n = \delta(q_{n-1}, w_n)$, where n is the length of the input. But instead of producing a single bit answer (as determined by $q_n \in F$ in a DFA), it outpus a string $\gamma(q_0)\gamma(q_1)\ldots\gamma(q_n)$ obtained by concatenating the output strings produced during the computation. So, the behavior of an FST is described by a function $f_T: \Sigma^* \to \Gamma^*$ mapping the input string $w \in \Sigma^*$ to an output string $f_T(w) \in \Gamma^*$. For any FST $T = (Q, \Sigma, \Gamma, \delta, s, \gamma)$, the function $f_T(w)$ is formally defined by induction on the length of the input $w \in \Sigma^*$:

$$\begin{aligned} f_T(\epsilon) &= \gamma(s) \\ f_T(a \cdot w) &= \gamma(s) \cdot f_{T'}(w) \end{aligned}$$

where $a \in \Sigma$, $w \in \Sigma^*$, and $T' = (Q, \Sigma, \Gamma, \delta, \delta(s, a), \gamma)$ is the FST obtained by changing the start state of T to $\delta(s, a)$.

We say that a function $f \colon \Sigma^* \to \Delta^*$ is FST-computable, if it can be computed by an FST, i.e., there is an FST T such that $f(w) = f_T(w)$ for all $w \in \Sigma^*$.

2 FST computations

We can also specify the function computed by an FST by defining a transition system over configurations, similarly to what we did for DFAs and NFAs. As for DFAs and NFAs, we need to define a set of configurations C_T, an initial configuration function $I_T \colon \Sigma^* \to C_T$, a transition relation $R_T \subseteq C_T \times C_T$, set of halting configurations $H_T \subseteq C_T$, and output function $O_T \colon H_T \to \Gamma^*$. Notice how for FST the output function needs to produce a string in Γ^* rather than just e decision bit. The transition system is easily defined:

1. $C_T = Q \times \Sigma^* \times \Gamma^*$. In each configuration (q, α, β), $q \in Q$ represents the current state, $\alpha \in \Sigma^*$ is the part of the input string that needs to be read, and $\beta \in \Gamma^*$ is the string that has already be outputted.

2. The initial configuration on input w is $I_T(w) = (s, w, \gamma(s))$

3. The transition relation R_T is the set of pairs $((q, aw, \beta), (q', w, \beta \cdot \gamma(q')))$ where $q \in Q$, $a \in \Sigma$, $w \in \Sigma^*$, $\beta \in \Gamma^*$, and $q' = \delta(q, a)$.

4. The halting configurations $H_T = \{(q, \epsilon, \beta) \mid q \in Q, \beta \in \Gamma^*\}$ are those where there is no more input to read

5. The output function is $O_T(q, \epsilon, \beta) = \beta$.

We can now apply the same general definitions already used for DFAs and NFAs. We recall that a (finite) computation of T on input $w \in \Sigma^*$ is a sequence of configurations $c_0, c_1, c_2, \ldots$, such that

- $c_0 = I_T(w)$
- $(c_{i-1}, c_i) \in R_T$ for all i, and
- if the computation is finite, then the last configuration is $c_n \in H_T$.

If the computation is finite, then we say that it terminates in n steps with output $O_T(c_n)$. It can be checked that for any T and w, the output of the computation of T on input w always equals $f_T(w)$.

Notice that the transition system associated to an FST is deterministic: for any configuration $c \in C_T$

- if $c \in H_T$, then there is no $c' \in C_T$ such that $(c, c') \in R_T$, i.e., the computation must necessarily terminate
- if $c \notin H_T$, then there is exacly one $c' \in C_T$ such that $(c, c') \in R_T$.

3 Combining an FST with a DFA

Cosider the result of running an FST T on an input string w, and then passing the result of a DFA M. It is natural to ask if these two computations can be combined together into a single DFA M' that takes w as input, and outputs the same result as $M(f_T(w))$. Notice that the set of strings accepted by this computation is

$$f_T^{-1}(\mathcal{L}(M)) = \{w \mid f_T(w) \in \mathcal{L}(M)\}.$$

So, the problem of combining T and M into a DFA can be phrased as a closure property of regular languages: prove that if L is regular, then $f_T^{-1}(L)$ is also regular.

The question of combining an FST and DFA together is non-trivial, because the obvious approach of first computing $f_T(w)$ and then running M on it cannot be directly implemented as a DFA because the strings w and $f_T(w)$ can be arbitrarily long, and will exceed the memory capacity of any DFA. In order to combine T and M together into a single DFA one needs to run T and M in parallel.

The following theorem proves this closure property of regular languages.

Theorem 1 *For any FST-computable function $f_T\colon \Sigma^* \to \Gamma^*$ and any regular language $B \subseteq \Gamma^*$, the langue*

$$A = f_T^{-1}(B) = \{w \in \Sigma^* \mid f(w) \in B\}$$

is also regular.

Proof Let $M = (Q, \Gamma, \delta, s, F)$ be a DFA such that $\mathcal{L}(M) = B$, and let $T = (Q_T, \Sigma, \Gamma, \delta_T, s_T, \gamma_T)$ be an FST. We combine M and T into a DFA $M' = (Q', \Sigma, \delta', s', F')$ where

- $Q' = Q \times Q_T$
- $\delta'((q, q_T), a) = (\delta^*(q, w), q'_T)$ for $q'_T = \delta_T(q_T, a)$ and $w = \gamma_T(q'_T)$.
- $s' = (\delta^*(s, \gamma_T(s_T)), s_T)$
- $F' = F \times Q_T$.

This automaton M' works by running T on the input string w, and feeding the result into M. So, $M'(w)$ accepts if and only if $M(f_T(w))$ accepts. It follows that the language $f_T^{-1}(B)$ is regular because it is the language of the DFA M' $\square$

4 Composing FSTs

Any two functions $f\colon \Sigma^* \to \Gamma^*$ and $g\colon \Gamma^* \to \Delta^*$ can be combined using the standard function composition operation $(g \circ f)\colon \Sigma^* \to \Delta^*$ defined as $(g \circ f)(w) = g(f(w))$.

Just as we were able to combine an FST with a DFA, you may ask if one can combine two FSTs together into a single FST that computes their function composition. In other words, if two functions are FST-computable, is their composition also FST-computable? As before, this is not a completely trivial question: given two FST-computable functions f_{T_1} and f_{T_2} (say, computed by FSTs T_1 and T_2), evaluating $f_{T_2}(f_{T_1}(w))$ requires the computation of an intermediate result $f_{T_1}(w)$ which may be just too long to be stored by an FST. So, we cannot simply apply the two functions in sequence as you would do using a general purpose programming language. In order for $f_{T_2} \circ f_{T_1}$ to be FST-computable, we need to be able to run T_1 and T_2 at the same time, and process the input string w in a streaming fashion. The following theorem shows how to do that.

Theorem 2 *For any FST $T_1 = (Q_1, \Sigma, \Gamma, \delta_1, s_1, \gamma_1)$ and $T_2 = (Q_2, \Gamma, \Delta, \delta_2, s_2, gamma_2)$, there is an FST $M = T_2 \circ T_1$ such that $f_M = f_{T_2} \circ f_{T_1}$.*

Proof Let $T = (Q, \Sigma, \Delta, \delta, s, \gamma)$ where s is a new state, $Q = Q_1 \times Q_2 \cup \{s\}$, and $\delta\colon Q \times \Sigma \to Q$ $\gamma\colon Q \to \Gamma^*$ are defined as follows. For the transition function we define

$$\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2^*(q_2, \gamma_1(q_1)))$$

$$\delta(s), a) = (\delta_1(s_1, a), \delta_2^*(s_2, \gamma_1(s_1))).$$

Notice that the start state s is treated just like (s_1, s_2). The reason we introduce a new start state is that the output is defined differently. For the start state we set

$$\gamma(s) = f_{T_2}(\gamma_1(s_1)),$$

i.e., we take the output of the second FST when given the string produces by the initial state of the first FST. For all other states the definition is the same

$$\gamma((q_1, q_2), a) = f_{T_2'}(\gamma_1(q_1))$$

but with a slighly modified version $T_2' = (Q_2 \cup \{s'\}, \Gamma, \Delta, \delta_2', s', \gamma_2')$ of the second FST that starts in a new state s', similar to q_2 but with no output. Formally, the modified T_2 has a new start state s' with output $\gamma_2'(s') = \epsilon$, and transitions $\delta'(s', a) = \delta_2(q_2, a)$ as q_2.

It can be easily verified by induction that $f_T = f_{T_2} \circ f_{T_1}$. $\square$

We remark that in order to compose two FSTs $T_2 \circ T_1$, the output alphabet of T_1 must match the input alphabet of T_2. The intuition behind the above construction is the following. The FST $T_2 \circ T_1$ works by running T_1 on the input string $w \in \Sigma^*$ to obtain some intermediate result $u \in \Gamma^*$. As T_1 outputs u, the composed automaton $T_2 \circ T_1$ runs the second FST on u to obtain the final output string v. Since finite automata (and FST in particular) do not have enough memory to store the intermediate result of the computation u, the two component automata T_1, T_2 are run at the same time, and the output of T_1 is fed to T_2 while it is being produced. In order to run the two automata at the same time, we use the cartesian product $Q_1 \times Q_2$ as the set of states of the composite automaton. Each state $(q_1, q_2) \in Q$ records the current state of T_1 and the current state of T_2.

5 Non-deterministic FST (NFST)

In applications it is sometime useful to consider nondeterministic transducers to model underspecified systems, user interaction, concurrency, etc. We define non-deterministic FST (NFST) by extending NFAs with an output alphabet Γ and replacing the set of accepting states F with an output function $\gamma\colon \Gamma^*$. In other words, NFST are finite state transducers that may take multiple transitions on the same input, or follow ϵ-transitions without reading any input at all.

Definition 2 *A Nondeterministic Finite State Transducer (NFST) is defined by a 6-tuple* $M = (Q, \Sigma, \Gamma, \delta, s, \gamma)$ *where*

- Q *is a finite set of states,*
- Σ *is a finite set of input symbols,*
- Γ *is a finite set of output symbols,*
- $\delta\colon Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ *is the (non-deterministic) transition function,*
- $s \in Q$ *is the start state, and*
- $\gamma\colon Q \to \Gamma^*$ *is the output function.*

As a reminder, $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ is the extended input alphabet, which includes all alphabet symbols in Σ, and a special element ϵ denoting the empty string. The definition extends that of deterministic FSTs in a way similar to how NFAs extend DFAs: the transition function δ outputs not just a single state $q \in Q$, but a (possibly emtpy) set of possible states $\delta(q, a) \subseteq Q$. Also, the transition function can be applied to the empty string $\delta(q, \epsilon)$, allowing the automaton to take a step without reading any input. At each step, the NFST selects (nondeterministically) one of the possible steps $q \in Q \times \Gamma^*$ from the output of the transition function, and executes it by transitioning to state q.

Just like NFAs, an NFST can perform several different computations on a given input, and some of these computations may abort before the input has been completely processed. When a computation is aborted, its partial output accumulated during the computation is also discarded.

The behavior of an NFST is described by a function $f_M\colon \Sigma^* \to \mathcal{P}(\Gamma^*)$ mapping the input string $w \in \Sigma^*$ to a set $f_M(w) \subseteq \Gamma^*$ of possible output strings. (This set can be empty if all computation branches abort.)

The function computer by an NFST is formally specified by defining a transition system. The definition is essentially the same as the one for FSTs:

1. The set of configurations is $C_T = Q \times \Sigma^* \times \Gamma^*$
2. The initial configuration on input w is $I_T(w) = (s, w, \gamma(s))$

3. The transition relation R_T is the set of pairs $((q, aw, \beta), (q', w, \beta \cdot \gamma(q')))$ where $q \in Q$, $a \in \Sigma_\epsilon$, $w \in \Sigma^*$, $\beta \in \Gamma^*$, and $q' \in \delta(q, a)$.
4. The halting configurations are $H_T = \{(q, \epsilon, \beta) \mid q \in Q, \beta \in \Gamma^*\}$
5. The output function is $O_T(q, \epsilon, \beta) = \beta$.

We see that all defininitions are identical to those already given for FSTs, except for the transition relation R_T which is now non-deterministic because it allows possibly empty inputs $a \in \Sigma_\epsilon$, and multiple transitions $q' \in \delta(q, a)$.

The definition of computations of an NFST T on input w, and their output, are defined identically to FSTs DFAs, the only difference being that now there are several possible (and potentially infinite) computations for $T(w)$. As a result, given an NFST T and an input w, we don't get just a single output string, but a set $f_T(w) \subseteq \Gamma^*$ of possible outputs.

As you may expect, given an NFST T and a DFA or NFA M, it is possible to combine the two together into a single NFA for the language

$$\mathcal{L}(M \circ T) = \{w \mid \exists u \in f_T(w).M(u) = \text{accept}\}$$

Similarly, given two NFST T_1, T_2, one can build an NFST T such that

$$f_T(w) = \bigcup_{u \in f_{T_1}(w)} f_{T_2}(u).$$

Proving both properties is left as an excercise.