Chapter 1

String Transducers

1.1 Deterministic Finite-state String Transducers

1.1.1 Orientation

This section is about deterministic finite-state transducers for strings. The term *finite-state* means that the memory is bounded by a constant, no matter the size of the input to the machine. The term *deterministic* means there is single course of action the machine follows to compute the output from some input. The term *transducer* means this machine solves *transformation problems*: given an input object x, what object y is x transformed into? The term *string* means we are considering the transformation problem from strings to objects. So x is a string. As we will see, we can easily write transducers where y is a string, natural number, real number, or even a finite stringset! We will also see that DFSAs are a specific case of DFSTs.

However, we first define the output of the transformation to be a string. Then we will generalize it.

1.1.2 Definitions

Definition 1. *A* deterministic finite-state string-to-string transducer (DFST) *is a tuple* $T = (Q, \Sigma, \Delta, q_0, v_0, \delta, F)$ *where*

- Q *is a finite set of states;*
- Σ *is a finite set of symbols (*the input alphabet*);*
- Δ *is a finite set of ouput symbols (*the output alphabet*);*
- $q_0 \in Q$ *is the* initial *state;*
- $v_0 \in \Delta^*$ *is the* initial string*;*
- δ *is a function with domain* $Q \times \Sigma$ *and co-domain* $Q \times \Delta^*$*. It is called the* transition function*. If transition* $(q, a, r, v) \in \delta$ *it means that there is a transition from state* q *to state* r *reading letter* a *and writing string* v*. It will be helpful to refer to the "first" and*

"second" outputs of delta with δ_1 and δ_2 respectively. So for all $(q, a, r, v) \in \delta$, we have $\delta_1(q, a) = r)$ and $\delta_2(q, a) = v)$. These are are the "state" transition and the "output" transitions, respectively.

- *F is a function with domain Q and co-domain Δ^*. Let's call it the* final function.

We also define a new function "process" $\pi : Q \times \Delta^* \times \Sigma^* \to Q \times \Delta^*$ as follows. π processes strings from a given state with a given string on the output and returns the state reached with a new output written.

$$\begin{aligned} \pi(q, v, \lambda) &= (q, v \cdot F(q)) \\ \pi^*(q, v, aw) &= \pi\big((\delta_1(q, a),\ v \cdot \delta_2(q, a),\ w\big) \end{aligned} \tag{1.1}$$

Consider some DFST $T = (Q, \Sigma, q_0, F, \delta)$ and string $u \in \Sigma^*$. Then $T(u) = v$ if and only if there is a state q and string $v \in \Delta^*$ such that $(q, v) = \pi(q_0, v_0, u)$. We say T *transforms* u into v.

Definition 2. *The function defined by T is $\big\{(u, v) \mid \exists q \in Q, v \in \Delta^*$ such that $\pi(q_0, v_0, u) = (q, v)\big\}$. We write $T(u) = v$ iff $(u, v) \in T$.*

Definition 3. *A string-to-string function is called* sequential *if there is a DFST that recognizes it.*

(Sequential functions are also often called subsequential functions. The nomenclature is unfortunate and different people have different opinions about it.) The important thing is that they are *deterministic* on the input and one should always check the definitions and not rely on names.

1.1.3 Exercises

Exercise 1. Let $\Sigma = \Delta = \{a, e, i, o, u, p, t, k, b, d, g, m, n, s, z, l, r\}$.

1. Write a transducer that prefixes *pa* to all words.
2. Write a transducer that suffixes *ing* to all words.
3. Write a transducer that deletes word initial vowels.
4. Write a transducer that voices obstruents which occur immediately after nasals.
5. Write a transducer that deletes word final vowels. So $T(abba) = abb$ and $T(pie) = pi$.
6. Write a transducer that voices obstruents intervocalically.

Note that the transition function and the final function can be partial functions. In this case, the transducer is *incomplete* in the sense it is not defined for all inputs. As before, we will strive to make our sequential transducers describe total functions.

1.2 Some Closure Properties of Sequential functions

Theorem 1 (Closure under composition)**.** *If f, g are sequential functions then so is $f \circ g$.*

Theorem 2 (Closure under intersection)**.** *If f, g are sequential functions then so is $f \cap g$.*

Theorem 3 (Minimal canonical form)**.** *For every sequential string-to-string function f, it is possible to compute a DFST T such that T is equivalent to f and no other DFST T' equivalent to f has fewer states than T.*

The closure theorems follow from proofs which use the product construction. The minimal cananoical form result is due to Choffrut (see his 2003 survey).

Sequential functions are not closed under union. This is because they are functions and so each input has a unique output.

1.3 Generalizing sequential functions with monoids

A *monoid* is a mathematical term which means any set which is closed under some associative binary operation with an identity. So if $(S, *, 1)$ is a monoid then for all $x, y \in S$:

1. is the case that $x * y$ is in S too (Closure under $*$)
2. $(x * y) * z = x * (y * z)$ (Associativity)
3. $1 * x = x * 1 = x$ (1 is the identity)

It is typical to refer to $*$ as "times", "multiplication" or as a product. It is also typical to refer to 1 as the "identity", "unit" or "one".

Σ^* is closed under the binary operation of concatenation. Also the empty string behaves like the identity with respect to concatenation. So $(\Sigma^*, \ \cdot \ , \lambda)$ is a monoid. As we processed the input string, we moved from state to state and updated the ouput value by concatenating strings along the output transitions. We can do the same thing and update the output value using some other product from another monoid.

Example 1. Here are some examples.

1. $(\{\texttt{True}, \texttt{False}\}, \wedge, \texttt{True})$. Boolean values and conjunction. This monoid shows the membership problem is a special case of the transformation problem.
2. $(\mathbb{N}, +, 0)$. Natural numbers and addition. Useful for counting!
3. $([0, 1], \times, 1)$. The real unit interval and multiplication. Useful for probabilities!
4. $(\mathbb{R}, \times, 1)$. All real numbers and multiplication.
5. (FIN, $\cdot$, $\{\lambda\}$) where FIN is the class of finite stringsets and $(\cdot)$ is concatenation of stringsets.

    - FIN $= \{S \mid \exists n \in \mathbb{N}$ with $|S| = n\}$

- $S_1 \cdot S_2 = \{u \cdot v \mid u \in S_1, v \in S_2\}$.

6. There are many others!

This means we can generalize DFSTs to transducers which output elements from any monoid.

Definition 4. *A* generalized sequential transducer (GST) *is a tuple* $T = (Q, \Sigma, \mu, q_0, v_0, \delta, F)$ *where*

- Q *is a finite set of states;*
- Σ *is a finite set of symbols (*the input alphabet*);*
- $(\mu, *, 1)$ *is a monoid*
- $q_0 \in Q$ *is the* initial *state;*
- $v_0 \in \mu$ *is the* initial *value;*
- δ *is a function with domain* $Q \times \Sigma$ *and co-domain* $Q \times \mu$*. It is called the* transition function*. As before, from this we derive* $\delta_1 : Q \times \Sigma \to Q$ *and* $\delta_2 : Q \times \Sigma \to \mu$ *to be the state and output transition functions.*
- F *is a function with domain* Q *and co-domain* μ*. Let's call it the* final function*.*

The process function π is the same except we replace concatenation of the outputs with the monoid operator $*$.

$$\begin{aligned} \pi(q, v, \lambda) &= (q, v * F(q)) \\ \pi^*(q, v, aw) &= \pi\big((\delta_1(q, a),\ v * \delta_2(q, a),\ w\big) \end{aligned} \tag{1.2}$$

Then, the definition of the function computed by the transducer is identical to what was written formerly.

That's it!

1.3.1 Exercises

Exercise 2. Let $\Sigma = \mu = \{a, e, i, o, u, p, t, k, b, d, g, m, n, s, z, l, r\}$.

1. Write a transducer that counts how many NC (nasal-consonant) sequences occurs in the input word.
2. Write a transducer that optionally voices obstruents which occur immediately after nasals. So for in input like *anta* the output should be the set {*anda, anta*}. (Hint: use the FIN monoid).

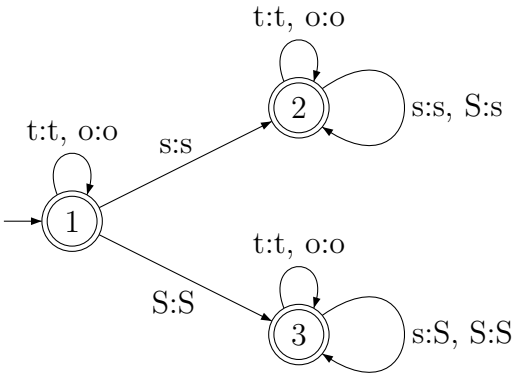
1.4 Learning more

Unfortunately, the material on deterministic transducers has yet to make its way into standard textbooks. Standard textbooks in computer science discuss non-deterministic finite-state transducers if they discuss transducers at all. Within computational linguistics where transducers are widely used, non-determinstic ones are the norm. See, for instance Roche and Schabes (1997); Beesley and Kartunnen (2003); Roark and Sproat (2007) and Jurafsky and Martin (2008). A notable exception is work by Mehyar Mohri (1997; 2005). The most textbook-like discussion of sequential functions I am aware of comes from Lothaire (2005, chapter 1).

My interest in sequential transducers stems from three interrelated facts. First, they appear sufficient to decribe morpho-phonologial generalizations in natural language (Jardine, 2016) and subsequent discussions. So the extra power that comes with non-deteminsitic transducers appears unnecessary in this domain. Second, sequential transducers have canonical forms (Choffrut's theorem), but non-deterministic ones do not. Third, the class of sequential transducers can be learned from examples, unlike the class of non-deterministic transducers (de la Higuera, 2010, chapter 18).

1.5 Left and Right Sequential Transducers

Below is a sequential transducer for progressive sibilant harmony. This means later sibilants agree in anteriority with the first sibilant in the word. The alphabet here is simply {s,S,t,o} with {s,S} signifying the two classes of sibilants and {t} other consonants, and {o} the vowels. We assume the initial value is λ and for all q the final function maps q to λ.



Many examples of sibilant harmony in natural language are not progressive. They are in fact regressive. For instance here are some wonderfully long words in Samala (Chumash) (Applegate 1972). The underlying form is on the left and the surface form is on the right. There are some alternations in the vowels which we ignore here.

/ha-s-xintila-waʃ/	[haʃxintilawaʃ]	'his former Indian name'
/k-su-kili-mekeken-ʃ/	[kʃuk'ilimekeket͡ʃ]	'I straigten myself up'
/k-su-al-puj-ʊn-ʃaʃi/	[kʃalpujat͡ʃɨʃi]	'I get myself wet'
/s-taja-nowon-waʃ/	[ʃtojowonowaʃ]	'it stood upright'

Exercise 3. Explain why regressive sibilant harmony *cannot* be modeled with sequential transducers.

There is a straightforward way to address this issue. The transducers we are using process strings from left-to-right. However, transducers could also process strings from right-to-left. If we use the transducer above to process the Samala strings from right to left, we can model regressive sibilant harmony. The example below shows /stototooS/↦[StototooS].

		s		t		o		t		o		t		o		o		S						
←	3	←	3	←	3	←	3	←	3	←	3	←	3	←	3	←	3	←	3	←	3	←	1	←
		S		t		o		t		o		t		o		o		S						

If a transducer processes the string left-to-right, it is called a *left sequential* transducer. If it processes it right-to-left it is called a *right sequential* transducer.

Theorem 4. *Left sequential functions and right sequential functions are incomparable; that is, there are functions that are both left and right sequential; neither left nor right sequential; left but not right sequential; and right but not left sequential.*

Progressive sibilant harmony is a case in point. It is left sequential but not right sequential. On the other hand, regressive sibilant harmony is right sequential, but not left sequential. The identity function is both left and right sequential. Can you think of a a function which is neither left nor right sequential?

The recursive data structure we are using for strings is inherently left-to-right because the outermost element in the list structure is on the left. If we were to define lists so that the outermost element of the list structure was on the right, it would become natural to process strings right-to-left.

An easy way to simulate a right sequential transducer with the recursive data structures we have is to do the following.

1. Implement left sequential transducers as we have done.
2. Before processing a string w, reverse it.
3. Then reverse the output v of the transducer.

In other words, `t.transduce(w)` will process the string w left-to-right. However, `reverse(t.transduce((reverse(w))))` will simulate t processing w right-to-left.

Bibliography

Applegate, R.B. 1972. Ineseño Chumash grammar. Doctoral dissertation, University of California, Berkeley.

Beesley, Kenneth, and Lauri Kartunnen. 2003. *Finite State Morphology*. CSLI Publications.

Choffrut, Christian. 2003. Minimizing subsequential transducers: a survey. *Theoretical Computer Science* 292:131 – 143.

de la Higuera, Colin. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press.

Jardine, Adam. 2016. Computationally, tone is different. *Phonology* 32:247–283.

Jurafsky, Daniel, and James Martin. 2008. *Speech and Language Processing: An Introduction to Natural Language Processing, Speech Recognition, and Computational Linguistics*. 2nd ed. Upper Saddle River, NJ: Prentice-Hall.

Lothaire, M., ed. 2005. *Applied Combinatorics on Words*. 2nd ed. Cmbridge University Press.

Mohri, Mehryar. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics* 23:269–311.

Mohri, Mehryar. 2005. Statistical natural language processing. In *Applied Combinatorics on Words*, edited by M. Lothaire. Cambridge University Press.

Roark, Brian, and Richard Sproat. 2007. *Computational Approaches to Morphology and Syntax*. Oxford: Oxford University Press.

Roche, Emmanuel, and Yves Schabes. 1997. *Finite-State Language Processing*. MIT Press.