600.406 — Finite-State Methods in NLP, Part II
Assignment 4: Building Finite-State Operators

Prof. J. Eisner — Spring 2001

As discussed in class, this week's exercises involve constructing new finite-state operators from old ones. You will use the FSA Utilities toolkit—the third and last of the finite-state packages that this course has exposed you too. You can review the interface at `http://www.cs.jhu.edu/~jason/405/software.html`.

The FSA Utilities have a powerful Prolog-based macro facility that is particularly convenient for constructing new operators (either algebraically or by manipulating automata). In addition, the integrated graphical interface is useful for debugging. The downside is that if you don't know Prolog well, you may find the interface confusing. We also can't currently compile the Prolog because we haven't licensed the compiler, so your user-defined operators will run slowly.

1. The FSA Utilities define a *same-length cross product* operator `xx`. If A and B are regular languages, then A `xx` B is defined as the regular relation that relates any string in A to any string of the same length in B.

   (a) Restate the above definition mathematically, in the form

   $$A \texttt{ xx } B \stackrel{\text{def}}{=} \{\langle a, b\rangle : \ldots\}$$

   (b) Suppose you have nondeterministic finite-state automata that recognize the sets A and B. Describe how to construct a finite-state transducer that recognizes the relation A `xx` B.

   (c) **[Turned out to be a bad question; see solutions for discussion.]** Under what conditions is this machine sequentiable? (That is, equivalent to a sequential machine, which is one that is deterministic on the input side. If so, A `xx` B is called a sequential relation.)

(d) Suppose you have regular expressions for A and B (call these expressions E and F). Using other standard finite-state operators available in FSA Utilities, write a regular expression for A `xx` B. (You can test your expression using the software, if you want.)

2. In class, we developed a left-to-right, longest-match `replace` operator, following the construction of Gerdemann & Van Noord (1999). (Their implementation in FSA Utilities is available locally at `file:/users/rtfm/rflorian/software/lib/fsa/GerdemannVannoord99/eacl99.pl`.) This question asks you to modify their construction in various ways.

   Recall that `replace`(T, L, R) denotes a transducer that effectively scans the input from left to right, using transducer T to replace substrings as it goes. At a given point in the string, a *match* is any substring that (1) starts at that point, (2) is in the domain of T, (3) is preceded by a substring in L (taking into account any changes already made to the preceding material), and (4) is followed by a substring in R. At each point, the transducer replaces the longest match x (if any) with $T(x)$, and continues at the next available point in the string. The next available point is defined as the next point in the input that falls after any replaced material.[1]

   Also remember that we constructed the transducer `replace`(T, L, R) as a composition of several smaller transducers. The symbols $<_1$, $>_1$, $<_2$, $>_2$ are called *marks* and are assumed to be disjoint from the input and output alphabets of the relation we are defining. The smaller transducers are applied to the input in the following order:

   (A) Insert $>_2$ before every (substring matching) R.

   (B) Insert $<_2$ before every `domain`$(T)>_2$ (ignoring internal marks).

   (C) Nondeterministically replace some nonoverlapping strings y that match $<_2$`domain`$(T)>_2$ (ignoring internal marks) with $<_1 y'>_1$, where y' is y without marks.

   (D) Eliminate outputs that contain substrings of the form $<_1$`domain`$(T)>_2$ (ignoring internal marks) which themselves contain $>_1$. This rules out non-longest matches.

   (E) Replace each substring y' between $<_1$ and $>_1$ with $T(y')$.

[1] "Next" is meant strictly: the transducer does not attempt multiple replacements (e.g., of ϵ) at the same point in the input. So if the transducer did not consume any input at this point—because there were no matches or because the longest match was ϵ—then it leaves the next input character unchanged in the output and looks for matches starting at the following input character.

(F) Eliminate outputs containing $<_1$ not preceded by L (ignoring marks). This ensures that replacement was done only in the appropriate left context (given other replacements).

(G) Eliminate outputs containing $<_2$ preceded by L (ignoring marks). This checks that replacement was done wherever possible.

(H) Delete all marks.

The above is a generate-and-test procedure. (Some of the steps can themselves be performed by generate-and-test: for example, to implement step (A), nondeterministically insert $>_2$ at some positions and then eliminate outputs where $>_2$ appears without R (ignoring marks) or vice-versa.)

Which steps in the above would you modify, and how, to get each of the following effects? (Answer at the same level of detail.)

(a) Optional longest-match replacement. This is a nondeterministic version of `replace` that, at each point, either does nothing or else replaces the longest match.

(b) Probabilistic longest-match replacement. Same as 2a, but the choice is stochastic: the replacement happens with probability p. The choices at different points are statistically independent.

(c) Shortest-match replacement. That is, at each point replace the shortest match, not the longest. (Be careful!)

(d) $\star$ **[Turned out to be a bad question; see solutions for discussion.]** Probabilistic lengthening-match replacement. The (new!) operator `replace`(T, L, R, p) should yield a conditional stochastic transducer that, for each input, produces several outputs whose probabilities sum to one. The new argument p is a probability.

As usual, the transducer scans the string from left to right. At each point, it may replace some match starting at that point. With probability p it replaces the shortest match (perhaps ϵ) if any; if not, with probability p it replaces the next-shortest match if any; and so on until it has either made a replacement or run out of available matches. Then it continues at the next available point in the string, as usual. All choices are statistically independent.

(e) **[Turned out to be a bad question; see solutions for discussion.]** Probabilistic shortening-match replacement. Same as 2d, but tries longer matches first. (This is like 2b except that the transducer considers shorter matches if it decides not to replace the longest.)

Remarks: There are of course many other possible variants. As discussed in class, the replacement is directed in the sense that it matches R against the input but L against the output, but we could modify the construction to match either L or R against either input or output (and xfst provides a full family of such operators).[2] There are also interesting possibilities for varying probabilistic replacement. For example, it might be possible to define a version that will never turn down its last chance to replace a match, or a version where the probability of replacing a match x is not a constant p but rather depends on the string x and/or the strings that match the context.

3. To solve the following problem, you will make use of a different way of stochasticizing a replacement transducer that was first proposed (though in a more restricted form) by Mohri & Sproat (1996). (You should *not* need to use your answers to the previous problem.)

   Recall that in Assigment 3, you were given a "bigram tag model" that mapped any string of tags to its bigram probability. Briefly describe how to construct such a model in the finite-state calculus, by using the original `replace`(T, L, R) operator. (Assume that you know the probability p_{xy} that tag x will be followed by tag y: $p_{xy} = \Pr(t_i = y \mid t_{i-1} = x)$.) Be careful that your solution handles a string like $xyyyz$ appropriately: it should include the probabilities of starting with x and stopping after z, and it should handle the repeated y's correctly.

4. In class, we looked at Optimality Theory (OT) with directional constraints. In this problem, you will use the FSA Utilities to construct a new "directional constraint" operator.

   This operator is useful in building dialect-specific transducers that map the "deep" phonological representation of a morpheme, word, or sentence to its "surface" representation (which could then be mapped to phonetics with a stochastic transducer).

   Eisner (2000) showed how to carry out the same operation by directly manipulating the states and arcs of finite-state machines, but in this exercise you will get the same result by combining operators of the finite-state algebra. As usual, this gives a cleaner but possibly slower implementation of the new operator. It is analogous to programming in a high-level language instead of assembly language. (Of course the transducer returned by the operator runs fast, if it can be determinized—in fact

[2] However, I don't think there is a way to define 2d or 2e as a stochastic process unless R is matched against the input.

the FSA Utilities can produce C++ code for you; what is slow is constructing that transducer!)

In order to test the operator, you will write an example **candidate generator** and some example **constraints**. It's easiest to explain the operator's behavior in terms of this concrete example. Read the material below carefully before you start!

In the example, you will use an alphabet {a,b,c,d,e} of 3 consonants and two vowels. The deep representation is written as a string of lowercase letters. The surface representation is written as a string of capital letters, with square brackets [] enclosing each syllable. The following table illustrates the input-to-output mapping for some simple cases:

Deep	**Surface**	
da	[DA]	$[CV]$ is a good syllable. (C=consonant, V=vowel)
dab	[DAB]	$[CVC]$ is equally good (at least in this exercise).
edab	[E][DAB]	We also tolerate $[V]$
ecdab	[EC][DAB]	and $[VC]$, but we don't like them as much ...
dabe	[DA][BE]	... which is why we prefer [DA][BE] to [DAB][E]
dabec	[DA][BEC]	and prefer [DA][BEC] to [DAB][EC].

However, some more complicated cases have different surface forms in different languages or dialects. Many languages rule out syllables like $[CCV]$ or $[CVCC]$ that contain two consonants in a row, perhaps because such "clusters" are hard to pronounce or understand.

Deep	**Surface**	
abcde	[AB][CDE]	Language L_1 allows the bad syllable.
	[ABC][DE]	Language L_2 allows a different bad syllable.
	[AB]C[DE]	Language L_3 doesn't syllabify everything.
	[AB][CE]	Language L_4 deletes extra consonants.
	[AC][DE]	Language L_5 is the same, but prefers to delete early.
	[AB][CV][DE]	Language L_6 inserts extra vowels. ($V \in \{\texttt{A}, \texttt{E}\}$)
	[A][BVC][DE]	Language L_7 is the same, but prefers to insert early.

Notice that multiple surface forms are allowed in some cases: [AB][CV][DE] stands for the set {[AB][CA][DE],[AB][CE][DE]}. So the mapping from deep to surface is in general a nondeterministic transduction.

An OT grammar consists of a generating transducer gen and a sequence of constraint transducers $C_1, C_2, \ldots C_n$. The above languages can be described by almost exactly the same grammar—the only difference is the order of the constraints.

`gen` maps the deep string to infinitely many possible **candidates**. We will define `gen` so that it inserts surface symbols arbitrarily into the deep string. For example, `gen` maps `abcde` to many candidates such as `[aAbB][cCA][dDeE]`, each of which contains a copy of the input `abcde`. A series of constraints then applies, and each constraint may remove ("prune") some of the candidates in a manner to be described later.

Let T_0 denote `gen`: it maps `abcde` to many candidates. Pruning T_0 by C_1 yields a transducer T_1 that maps `abcde` to a subset of those candidates. Pruning T_1 by C_2 yields a transducer T_2 that maps `abcde` to a subset of *those* candidates, and so on until we get the full grammar, a transducer T_n. The full grammar for our language L_6 maps `abcde` to only two candidates: `[aAbB][cCA][dDeE]` and `[aAbB][cCE][dDeE]`. All of the other infinitely many candidates were pruned away by one constraint or another. Note that in the surviving candidates, every surface (capital) letter—except for the inserted vowel—can be seen to correspond to a deep (lowercase) letter.

To find the surface form, we compose T_n with a transducer that removes the deep symbols (lowercase letters), leaving `[AB][CA][DE]` and `[AB][CE][DE]` as shown in the table above. We had to preserve the deep symbols at intermediate stages, for consideration by **correspondence constraints** that considered how well the deep symbols match the surface symbols. The other constraints are **surface constraints** that ignore the deep symbols.

Now, what is a constraint and how does pruning work? There are a few possible answers that correspond to different pruning operators:

hard constraints A constraint C_i is a regular language. It prunes any candidates that are not in the language. For this case we can use the simple transducer composition $T_i = T_{i-1} \circ C_i$. Of course this is an operator we already have.

binary constraints A constraint C_i is a regular language. It prunes any candidates that are not in the language, *unless* this would prune *all* the remaining candidates for this input, in which case it prunes nothing! For this case we use the notation $T_i = T_{i-1}$ `oo` C_i. Formally, T_i is defined to be the relation that maps input x to the set $T_{i-1}(x) \cap C_i$ if that set is non-empty and to $T_{i-1}(x)$ otherwise.

The operator `oo` was introduced by Karttunen (1998); it is reminiscent of composition and stands for "optimality operator." In the OT spirit, it makes C_i **violable**. That is, it refrains from pruning the best candidates available, even if they violate C_i. At least one candidate always survives.

counting constraints A constraint C_i is a transducer that inserts 0 or more stars `*` into a candidate—one star at every place the constraint doesn't like. (Remember star means "bad" in linguistics. For example, if the constraint doesn't like syllables, then it might map `[aAbB][cCE][dDeE]` to `[*aAbB][*cCE][*dDeE]`, which has one star per syllable.)

Pruning with C_i keeps the candidates with the fewest stars and prunes the rest. For this case we use the notation $T_i = T_{i-1}$ `o+` C_i. Formally, let $N(z)$ denote the number of stars in a string z. Then T_i is defined to be the relation that maps x to the set $\{y \in T_{i-1}(x) : (\forall y' \in T_{i-1}(x))N(C_i(y)) \leq N(C_i(y'))\}$.

This `o+` is the traditional pruning operator in OT. Unfortunately, as we saw in class (and Frank & Satta (1998) first showed), it cannot be implemented in the finite-state calculus. Even if T_{i-1} and C_i are rational (i.e., finite-state), T_{i-1} `o+` C_i might not be rational.[3]

directional left-to-right constraints A constraint C_i is a transducer that inserts 0 or more stars `*` into a candidate, as described above. It prunes candidates that have "unnecessarily early" stars relative to the deep symbols. For this case we use the notation $T_i = T_{i-1}$ `o>` C_i. To judge how early the stars are, we delete all the surface symbols:

`[*aAbB][*cCE][*dDeE]`	$\rightarrow$ `*ab*c*de`	$\rightarrow \langle 1,0,1,1,0,0\rangle$
`[*aAb*B][**cCE][dDeE]*`	$\rightarrow$ `*ab***cde*`	$\rightarrow \langle 1,0,3,0,0,1\rangle$
`[aAb*B][**cCE][dDeE]*`	$\rightarrow$ `ab***cde*`	$\rightarrow \langle 0,0,3,0,0,1\rangle$

If z is a string in the first column, let $S(z)$ denote the **star pattern** in the second column. These strings can be ordered alphabetically, taking `*` to be the last symbol in the alphabet. (Equivalently, let $S(z)$ denote the vector in the third column, which says how many stars fall at each position in `abcde` in the star pattern, and order these vectors lexicographically.) $S(z)$ is greatest in the second row and smallest in the third row. For a given input, pruning keeps only the candidates for which $S(z)$ is smallest: that is, T_i maps x to the set $\{y \in T_{i-1}(x) : (\forall y' \in T_{i-1}(x))S(C_i(y)) \leq S(C_i(y'))\}$.

directional right-to-left constraints The mirror image of the previous case: it prunes candidates that have "unnecessarily late" stars. For this case we use the notation $T_i = T_{i-1}$ `<o` C_i.

[3]However, Ellison (1994) (as reformulated by Eisner (1997)) pointed out that for any input x, $T_i(x)$ is regular, and can be found from $T_{i-1}(x)$ and C_i by a best-paths operation. Thus we have a way to find the set of surviving surface candidates $T_n(x)$ for any deep x. The trouble is that the relation T_n is not a transducer, so we can't invert it or compose it with others.

Now for the problems!

(a) To warm up, let's deal with binary constraints.

i. Explain why a binary constraint is a kind of counting constraint.

ii. ⋆ Frank and Satta (1998) showed that the binary constraint optimality operator `oo` can be implemented in the finite-state calculus although the counting version `o+` cannot be. Give an expression in the language of the FSA Utilities for T `oo` C in terms of the relation T and the language C. (Hint: First give an expression in terms of relations Q and R for the **priority union** of Q and R, which is the relation $Q \cup \{\langle x, y\rangle \in R : Q(x) = \emptyset\}$ that maps x to the set $Q(x)$ if that set is non-empty and to $R(x)$ otherwise. This formulation is due to Karttunen (1998). If you really get stuck, check out the latter at `http://www.xrce.xerox.com/publis/mltt/pto/bilkent.html`.)

iii. Suppose C is a "bounded" counting constraint that can only count up to 3. Show that you can express T `o+` C in terms of `oo`.

(b) Now we'll do directional constraints. Download a copy of the file `http://www.cs.jhu.edu/~jason/405/otdir.plg`. This is a full solution to the problem, but some of the definitions have been removed. Follow the directions (read carefully!) to fill in the missing definitions, and test that they work as expected. Hand in just the definitions you filled in:

i. `constraint(Lif,Lthen,Rif,Rthen)`

ii. ⋆ `surfconstraint(Lif,Lthen,Rif,Rthen)`

iii. `noins`

iv. `onset`

v. `nocomplex`

vi. `singlenuc`

vii. `worsen`

viii. ⋆ `prune_lr(TC)`

ix. `T do C` (the program's notation for `T <o C`)

x. `lang_one`

xi. `lang_two`

xii. `lang_three`

xiii. `lang_four`

xiv. `lang_five`

xv. `lang_seven`

(c) What does your `lang_three` transducer do on the input `abccccde`? Why? Could you have given a different grammar that also predicted all the above data for L_3 but made a different prediction on `abcccddde`?

(d) Describe how to add a constraint to your grammar for L_6 so that the inserted vowel will always be `A`, not `E`. So the modified grammar maps `abcde` only to `[AB][CA][DE]`, not `[AB][CE][DE]`. (Just explain how to modify the grammar; you don't have to try it out.)

(e) Explain why `ignore` is used in the definitions of `addstarwhere` and `delstarwhere`.

(f) We only intend to apply the grammar to strings that match `deep*`. Would the implementation remain correct (on such strings) if we removed `deep* o` from the definition of `gen`? If not, why not? If so, are there other reasons not to remove it? (Feel free to try this out!)

(g) Examine the transducer for `syllabify`. It is rather complex. But you can probably imagine a simple 2-state transducer that does the job of starring any surface segment that is not in a syllable. (The two states would stand for "currently in a syllable" and "currently not in a syllable.") Let's try some optimizations in order to get a better machine.

i. How many states in `t_minimize(syllabify)`? (Use the "Count FA" button at the bottom of the GUI window.)

ii. How many states in `t_minimize(syllabify)`?

iii. We can assume that a constraint will only apply to a star-free string with matched brackets. Use the trick of question 4f to modify the definition of `constraint()` accordingly. (Hint: use `range(gen)`.) Now how many states in `t_minimize(syllabify)`?

iv. Explain what the arc labels in `t_minimize(syllabify)` mean. (The top of `otdir.pl` explains how to read them.) Why are they so much messier than the arc labels in `syllabify`?

(h) A transducer like `lang_five` is rather large and messy. It appears to have arcs that deal with individual letters like `B`, `C`, and `D` separately. So we should be concerned that it will get larger and messier with a bigger alphabet. To check this, we can ask whether it would get *smaller* with a *smaller* alphabet.

Write an expression `lang_five_small` in terms of `lang_five` that does the same thing as `lang_five` but only on the smaller symbol set {`a`, `b`, `A`, `B`, `[`, `]`},

so that it allows only one consonant and one vowel. How many states are in each of `t_minimize(lang_five)` and `t_minimize(lang_five_small)`? Notice that the latter machine is reasonably easy to understand by inspection. (The function `t_minimize` minimizes a transducer in FSA Utilities.)

(i) ⋆ Ideally, we would have a machine as small as `t_minimize(lang_five_small)` even with a large alphabet. This might be possible if we allowed the output letters to be lowercase. Then instead of separate arcs for `b:B`, `c:C`, and so on, we could have just one arc `?` that says "copy a symbol," or one arc `b..d` that says "copy a consonant."[4]

i. Briefly, how would you implement this strategy? (Hint: To get a small machine, you will not be able to use anything like `corrpair` (nor a lowercase equivalent) when defining `noins` and `nodel`. That would result in deleting one symbol and inserting it again later, not copying it using `?`. So you will actually have to change the way candidates are encoded as strings.)

ii. The determinization algorithm (available as `t_determinize`) may postpone an arc's output to a later arc. For example, when we determinize E, an arc labeled `b:B` might become `b:`ϵ and a later arc such as `c:BC` would output the `B`. (This postponement is necessary if we need to look ahead farther in the input to tell whether it is the `b:B` arc or some other `b:`$\cdots$ arc that we need to follow if we are to be able to read rest of the input after `b`.)

It is also possible to postpone the output of an arc labeled `b`, since that is equivalent to `b:b`. But what if the arc is labeled `?` or `b..d`, so that the output is unified with the input? Can the output still be deferred? Does this prevent us from determinizing the transducer—hence also from minimizing it?

Experiment with some simple expressions in FSA Utilities to see what happens in this case. Briefly describe what you did and what you concluded from it.

[4] Remember that a language L is coerced to the identity relation $\langle w, w\rangle : w \in L$, which copies any string in L from input to the output. So an arc labeled `b` is considered to copy `b` (it is equivalent to `b:b`), and an arc labeled `?` copies any symbol.