600.405 — Finite-State Methods in NLP
Assignment 2: Semirings etc.

Prof. J. Eisner — Fall 2000
Handed out: Sat., Nov. 18, 2000
Due: At the start of the Tue., Nov. 28 lecture

1. Recall from class that a **semiring** is a set K equipped with binary operations $\oplus$ ("collect") and $\otimes$ ("extend") that satisfy the following axioms:

   - $(K, \otimes)$ is a *monoid*. That means that:
     - $\otimes$ is a function from $K \times K \to K$.
     - $\otimes$ is associative: $(\forall x, y, z \in K).(x \otimes y) \otimes z = x \otimes (y \otimes z)$
     - K has a two-sided identity: Some element $\underline{1} \in K$ satisfies $(\forall x \in K)\underline{1} \otimes x = x = x \otimes \underline{1}$.
   - $(K, \oplus)$ is also a *monoid*, whose identity is denoted by $\underline{0}$. Furthermore it is a *commutative monoid*: $(\forall x, y \in K).x \oplus y = y \oplus x$.
   - $\otimes$ distributes over $\oplus$ from both directions: that is, $(\forall x, y, z \in K)x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z)$ and also $(y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x)$. (Both statements are necessary because $\otimes$ might be some non-commutative operation like concatenation.)
   - $\underline{0}$ (the identity for $\oplus$) also has a property with respect to $\otimes$: $(\forall x \in K)x \otimes \underline{0} = \underline{0} = \underline{0} \otimes x$.

   Simple semirings $(K, \oplus, \otimes)$ include $(\{false, true\}, \vee, \wedge)$, $(\mathbf{R}, +, \times)$, and $(\mathbf{R}, \min, +)$.

   (a) Show that if $(K, \oplus, \otimes)$ are specified, then $\underline{0}$ and $\underline{1}$ can be deduced. That is, explain why there cannot be two different elements of K that could be the identity for $\oplus$ (or $\otimes$), provided that the other semiring axioms also hold.

(b) An interesting semiring might be $(\{false, true\}, \wedge, \wedge)$. Describe how a weighted automaton would be interpreted under this potential semiring. Then check whether it is a semiring: which axioms does it satisfy?

2. (a) If the following weighted machine is interpreted using the semiring $(K, \oplus, \otimes) = (\mathbf{R}, +, \times)$, it computes a simple function on strings in $0, 1^*$. What function is it? Note: Arc labels in this diagram have the form "input symbol / weight." State labels have the form "state number / stopping weight." The state number is arbitrary.

   (b) If you interpret the same machine using the semiring $(\mathbf{R}, \min, +)$, what function does it compute then?

   (c) $\star$ Using $(\mathbf{R}, +, \times)$, can you change the machine's weights so that it computes the function that maps any string in $0, 1^*$ to the integer it denotes in binary? For example, it should assign the input string 0100101 a total weight of 37.

3. Let L denote the language $a^n b^n : n \geq 0$.

   (a) Using the pumping lemma, prove that L cannot be recognized by any unweighted finite-state automaton.

   (b) Show that L can be recognized with a *weighted* finite-state automaton, in the following sense: the function $f : a, b^* \to K$ computed by the automaton returns $\underline{0}$ on exactly the elements of L. That is, $f(w) = \underline{0}$ if and only if $w \in L$. You should give the automaton (or regexp) and specify the semiring $(K, \oplus, \otimes)$ that you are assuming.

   (c) Can you give a *deterministic* version of your automaton from (b)?

   (d) Write a weighted regular expression for your automaton.

   (e) $\star$ The balanced parenthesis language D (also called the Dyck language) is generated by this context-free grammar:

   - $S \to \epsilon$
   - $S \to (S)$
   - $S \to SS$

For example, $()()(()(()))\in D$, but $())(()\notin D$ because its parentheses are not balanced.

Could you "recognize" D with an automaton similar to the one you gave above, over the same semiring? If so, give the automaton. If not, would it help to use a different semiring?

4. (a) Draw automata for the regexps a^*b^+ and a^+b^*. Each automaton should have only 2 states; name the states 0 and 1.

   (b) Use the intersection construction to draw the intersection of these automata. Hint: The states will have names (0,0), (0,1), (1,0), and (1,1).

   (c) How many states are in the minimization of the above automaton?

   (d) Give two small automata, with $n \geq 2$ and $m \geq 2$ states respectively, whose intersection requires fully nm states even when minimized.

5. (a) Draw transducers for the regexps $R_1 = (a : g)^*(b : h)^+$ and $R_2 = (g : p)^+(h : q)^*$. Each automaton should have only 2 states; name the states 0 and 1.

   (b) The regexp R_1 describes a relation $L(R_1)$, i.e., a set of pairs of strings. List the elements of that set (there are infinitely many, so use "…" once the pattern is clear!). Similarly, list the elements of $L(R_2)$.

   (c) The composition $R_1 \circ R_2$ describes the relation $L(R_1 \circ R_2) = \{(x, z) : (\exists y)(x, y) \in R_1 \wedge (y, z) \in R_2\}$. List the elements of $L(R_1 \circ R_2)$.

   (d) Draw the composition of the automata from 5a. The construction is essentially the same as in intersection: again the states will have names (0,0), (0,1), (1,0), and (1,1).

6. (a) Pig Latin is a simple coded version of English used by children. Follow the assignment at `http://www.cis.upenn.edu/~cis639/assign/assign1.html` (thanks to Lauri Karttunen and others), which defines the code and instructs you to build a transducer that translates from English to Pig Latin. Hand in a printout of your xfst script, which should contain enough comments to explain how you solved the problem. (Note: You may use a tool other than xfst if you prefer.)

   (b) The transducer can be run backwards to translate Pig Latin to English. Give examples of Pig Latin strings that have 0, 1, and 2 English translations.

   (c) You were told to translate "this street" as "histay treetsay." But really, most Pig Latin speakers would say "isthay eetstray," moving entire consonant clusters "th" and "str." Briefly, explain how you could modify your transducer to do this. That is, what would be the easiest trick so that if a word starts with multiple consonants (up to 3), *all* of them will be moved to the end before "ay" is added?

(d) What if a word can start with an unlimited number of consonants (not just up to 3)?