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

Solution Set

Prof. J. Eisner — Fall 2000

1. (a) Assume $a, b \in K$ are both identities for $\oplus$. Then $a \oplus b = a$ because b is an identity, and $a \oplus b = b$ because a is an identity, so $a = b$. The proof for $\otimes$ is similar.

   (b) The interpretation of $(\{false, true\}, \wedge, \wedge)$: not one but *all* paths that read a string would have to reach a final state for the string to be accepted.

   As a special case, a string with no paths that read it is accepted by all the paths that read it, and therefore would be accepted by the machine! (If this strikes you as odd, notice that the total weight of no paths is always $\underline{0}$, and $\underline{0} = true$ here.)

   But $(\{false, true\}, \wedge, \wedge)$ is not a semiring because it violates the last axiom that $(\forall x \in K) x \otimes \underline{0} = \underline{0} = \underline{0} \otimes x$. Specifically, take $x = false$ and observe that $false \wedge true \neq true$. It does satisfy all the other axioms.

   Remark: Of course one could define a different kind of machine—let's call it a co-automaton—that accepts a string iif all paths that read that string accept. There are two ways to see that the languages accepted by co-automata are regular:

   - Given a co-automaton, we can make it complete and deterministic via the usual subset construction: the only change is that a stateset is final iff *all* of its component states are final. Then we can simply interpret it as an ordinary automaton—which certainly defines a regular language. Why? Because a complete deterministic machine will define the same language (function to $\{false, true\}$) whether it's interpreted as a co-automaton over $(\{false, true\}, \wedge, \wedge)$ or an ordinary automaton over $(\{false, true\}, \vee, \wedge)$. This is because in complete deterministic machines, $\oplus$ and $\underline{0}$ are not used at all, since there is exactly one path reading each string to sum.

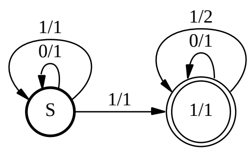
- Given a co-automaton accepting L, we can change its non-final states to final ones and vice-versa to get an ordinary automaton over $(\{false, true\}, \vee, \wedge)$ that accepts the complement $\neg L$. So $\neg L$ is regular and therefore L is too. Of course, flipping the finality of all states is the usual way to take the complement of an automaton. It's precisely because it changes the $\oplus$ operation that this construction is ordinarily applied only to complete deterministic machines, where the change in $\oplus$ is irrelevant as discussed above.

Similar arguments show that co-automata accept *all* regular languages, so they accept exactly the regular languages, just like ordinary automata.

2. (a) It returns the number of 1's in the input string, plus 1. (The "plus 1" is because the start state is not final.)

   (b) It returns the length of the input string, plus 1. (The "plus 1" is because the stopping weight is 1 at both final states.)

   (c) $\star$



Alternatively, make the start state final but give it stopping weight 0.

(We could drop the formal notion of final vs. non-final states; non-final states are just those that happen to have stopping weight $\underline{0}$. It is nonetheless conventional (and helpful) to draw the two kinds of states differently in diagrams.)

3. (a) Let's review the pumping lemma: any regular language is closed under "pumping" within a sufficiently long prefix. **Pumping** the substring v of $uvw \in L$ yields the strings $uw, uvw, uvvw, uv^3w \ldots = \{uv^iw : i \geq 0\}$. v is called **pumpable** if all these strings are also in L and $v \neq \epsilon$. The pumping lemma states that if L is regular, $\exists k(L) > 0$ such that every string $z \in L$ with at least $k(L)$ characters has a (non-empty) pumpable substring within its first $k(L)$ characters.[1]

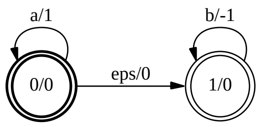
[1]Proof sketch: Take $k(L)$ to be the number of states in some FSA for L. When this FSA reads z, the accepting path must cycle back on itself within the first $k(L)$ characters. The substring read by this cycle can be pumped.

Suppose L were regular. Then some substring of a's in the first $k(L)$ characters of $a^{k(L)}b^{k(L)}$ would be pumpable; but that would mean *inter alia* that removing this substring from $a^{k(L)}b^{k(L)}$ would give another string of L, which is false.

(Alternate proof: No substring anywhere in $a^n a^n$ is ever pumpable. Such a substring would have to have the form $a^i b^i$ ($i > 0$) so that pumping it would give equal numbers of each symbol, but pumping it once would give $a^{n-i}a^i b^i a^i b^i b^{n-1} \notin L$.)

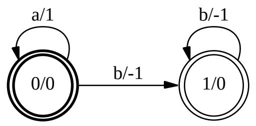
(b) *Erratum:* I meant to say that strings in the language (and in the Dyck language below) should be accepted with weight $\underline{1}$, not $\underline{0}$.

You all got the answer I intended anyway, which accepts L with weight $\underline{1} = 0$ in the semiring $(\mathbb{R} \cup \{\infty\}, \min, +)$:



Warning: We have defined recognition in a funny way. This machine does not recognize $a^n b^n$ in the same sense in which ordinary FSAs do. In particular, you could write weighted machines to recognize $a^n b^n c^*$ and $a^* b^n c^n$, but you couldn't intersect them to get a machine for $a^n b^n c^n$.

(c)



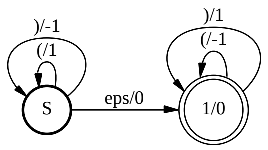
Note that the start state must be final so that ϵ is accepted.

(d) $(a/1)^*(b/-1)^*$

(e) $\star$ It's hard to recognize the Dyck language using a *deterministic* automaton like the one given above. The idea of having left and right parentheses add 1 and -1 to the weight of the path still makes sense, but the path must somehow crash if its weight ever goes negative.

Unfortunately the path doesn't "know" its own weight, i.e., the availability of arcs cannot depend on the current path weight but only on the state.

Amazingly, we *can* recognize the Dyck language using nondeterminism (Cortes & Mohri, forthcoming). The following automaton over $(\mathbb{R} \cup \{\infty\}, \min, +)$ assigns weight 0 ($= \underline{1}$) to exactly the strings of the Dyck language D:



Claim that if $w \in D$, then it has a 0-weight path and no negative-weight paths (so $\min = 0$), while if $w \notin D$, then it has a negative-weight path (so $\min < 0$). These three claims establish correctness of the automaton, and they have very short proofs.[2]

If we insist on a deterministic machine, as most of you tried to do, then we have to arrange by semiring addition rules that a bad path (one that has read more right than left parentheses) can never recover (get back to weight $\underline{1}$ by reading more symbols).

What's hard is to do this while satisfying the semiring axioms, such as associativity of $\otimes$. In particular, a string in the Dyck language may have many substrings that are not in the Dyck language, such as **)** and **))))((**.

The most straightforward approach is to let the weights be strings of parentheses. The $\otimes$ operation should be able to repeatedly delete substrings of the form **()**: so we want **))(((** $\otimes$ **))(** $=$ **))((**. In fact, with this kind of automatic cancellation, every path weight will be a string of the form $\mathbf{)}^i\mathbf{(}^j$. It is clear that paths with weight $\underline{1} = \epsilon$ are exactly those that read strings of the Dyck language.[3]

One might prefer to reprsent the weight $\mathbf{)}^i\mathbf{(}^j$ more concisely as just the ordered pair $\langle i, j\rangle$. So the monoid $(K, \cdot)$ we have just defined on strings is isomorphic to $(\mathbb{N}^2, \otimes)$ where

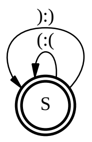
$$\langle i, j\rangle \otimes \langle k, \ell\rangle = \begin{cases} \langle i + (k - j), \ell\rangle & \text{if } k \geq j \\ \langle i, (j - k) + \ell\rangle & \text{otherwise} \end{cases}$$

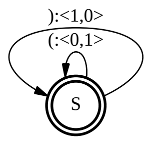
So here are two drawings of the deterministic machine to recognize the Dyck language: one uses the string notation, the other uses the ordered-pair nota-

[2]Try it! Use the fact that $w \in D$ iff, as one reads successive characters of w, the excess of left over right parentheses stays ≥ 0 and ends up at 0. Also take advantage of symmetries of the language and the automaton.

[3]Mathematically speaking, we are defining a monoid $(K, \cdot)$ as the quotient of $(\Sigma^*, \cdot)$ by the equation **()** $= \epsilon$. This is a monoid whose elements are equivalence classes of Σ^* under the relation $u\mathbf{()}v \equiv uv$ for any u, v. It is then convenient to denote an equivalence class by its unique member of the form $\mathbf{)}^i\mathbf{(}^j$. Note that **(** has a right inverse **)** in this monoid, whereas it does not in Σ^*.

tion. Strings of the language are assigned the weight $\underline{1}$, which is ϵ or $\langle 0, 0\rangle$ in the respective notations.

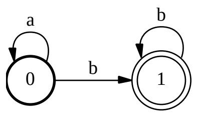


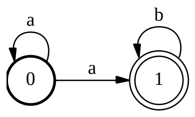


Because the machines are deterministic, the definition of $\oplus$ is irrelevant (see the discussion of problem (1b) above): there is always just one path to sum over. However, we do need to establish that there is some $\oplus$ such that the semiring axioms are satisfied. An $\oplus$ that always works is set union: if the multiplicative monoid we want is $(K, \otimes)$, then use the semiring $(\mathcal{P}(K), \cup, \otimes')$ where $A \otimes' B \stackrel{\text{def}}{=} \{a \otimes b : a \in A, b \in B\}$. The usual semiring for string-to-string transducers is lifted from the monoid $(\Sigma^*, \cdot)$ in exactly this way.

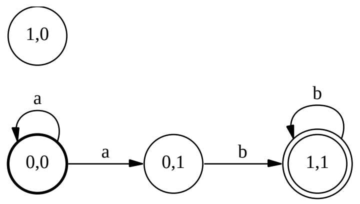
Specialized idiosyncratic semirings like this one can have expensive $\oplus$ and $\otimes$ operations (so be careful). They are also of limited use, since machines over different semirings can't be composed or intersected with one another. However, the quasi-determinization and minimization algorithms do apply. A machine like this one is also useful for illustrating that a computation can be performed with little memory (here, a finite state and two unbounded integers) and—when the machine is deterministic—bounded lookahead.

4. (a)



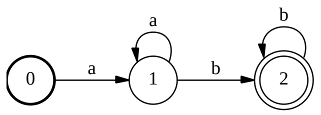


(b)



(c) The minimization has 3 states. Note that is not just a matter of removing the unreachable state above (which you can do with `fsmconnect` in the

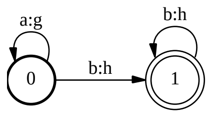
FSM package), since the machine above is nondeterministic. Determinization yields the following (and minimization does not change it further):

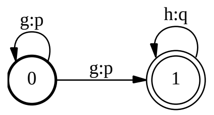


(d) The regexps $\Sigma^* a \Sigma^*$ and $\Sigma^* b \Sigma^*$ can be realized by 2-state machines. The intersected language consists of all strings containing both a and b, in either order. This requires $2 \cdot 2$ states to remember whether a has been seen yet and also whether b has been seen yet.

Another nice answer is that $(a^n)^* \cap (a^m)^* = (a^{nm})^*$ if n and m happen to be relatively prime. The minimal automata for these languages are simple cycles with n, m, and nm states respectively.

5. (a)





(b)

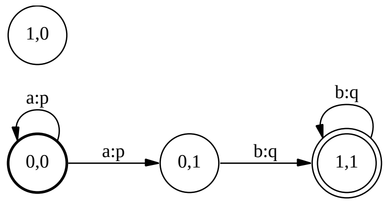
$$\begin{aligned}
L(R_1) &= \{(b,h),(ab,gh),(aab,ggh),\ldots,(bb,hh),(abb,ghh),(aabb,gghh),\ldots\} \\
&= \{(a^i b^j, g^i h^j) : i \geq 0, j > 0\} \\
L(R_2) &= \{(g,p),(gh,pq),(ghh,pqq),\ldots,(gg,pp),(ggh,ppq),(gghh,ppqq),\ldots\} \\
&= \{(g^i h^j, p^i q^j) : i > 0, j \geq 0\} \\
&= \{(g,p),(gg,pp),\ldots,(gh,pq),(ggh,ppq),\ldots,(ghh,pqq),(gghh,ppqq),\ldots\}
\end{aligned}$$

(Notice that the latter ordering for $L(R_2)$ matches the ordering for $L(R_1)$ better.)

(c)

$$\begin{aligned}
L(R_1 \circ R_2) &= \{(ab,pq),(aab,ppq),\ldots,(abb,pqq),(aabb,ppqq),\ldots\} \\
&= \{(a^i b^j, p^i q^j) : i \geq 0, j \geq 0\}
\end{aligned}$$

(d)



6. (a) Finite-state machines are not very good at moving or copying substrings around, because they need special states to remember them.One could extend the regular expression language, or the automaton formalism, with registers that can save substrings for later emission. If there are finitely many registers of finite size, the result is still finite-state. The following regexp seems likely to be broadly useful for solving such problems:

```
define Pair a a | b b | c c | d d | e e | f f | g g | h h ...;
```

This centralizes the pain of listing cases. It's now easy to get funky:

```
define Triple Pair ? & ? Pair;                       # aaa,bbb,...
define PairX [Pair .o. [[..] -> X || ? _ ?]].l;      # aXa,bXb,...
define Palindrome3 [PairX .o. [X->?]].l;             # a?a,b?b,...
define Palindrome5 [PairX .o. [X->Palindrome3]].l;# radar, etc.
define Redup4 Palindrome3 ? & ? Palindrome3;         # mama, etc.
```

`Pair` starts you off with one state per letter; then the automata operations let you build bigger memories.

The Pig Latin problem could be solved as follows, starting with `Pair` and `Letter` only:

```
define PairSkip Pair ./. ?;                         # a?*a,b?*b,...
define MoveChar ?* 0:? .o. PairSkip .o. ?:0 ?*; # 1st char to end
define Vowel a|e|i|o|u;
define Cons Letter - Vowel;
define PigWord Letter+ .o. [Cons ?* .o. MoveChar 0:{ay}]
                            | Vowel ?*];
define NonWord \Letter+;
define Latinize (NonWord) [PigWord NonWord]* (PigWord);
```

Remark: One would really like to simplify the last 3 lines to this:

```
define PigWord Cons Letter+ .o. MoveChar 0:{ay};
define Latinize @PigWord || EdgeOfWord _ EdgeOfWord;
```

where the last line is intended to generalize `A @-> B || L _ R` (directed replacement). It is meant to denote a transducer that applies `PigWord` as often as possible throughout its input, using greedy left-to-right longest-match to pick out substrings of the input that are in the domain (upper language) of `PigWord` and are flanked by `EdgeOfWord`. This construct is not available in xfst, but could be built up as a macro in the FSA utilities (anyone want to try?).

For the record, everyone in the class wrote machines that explicitly copy a particular consonant from the start of a word to the end: for example,

```
[..] -> bay || EdgeOfWord b Letters _ EdgeOfWord
```

or

```
b Letters -> ... bay || EdgeOfWord _ EdgeOfWord
```

where the capitalized symbols are the names of simple regexps. You then composed these together and composed the result with a transducer that deleted word-initial consonants.

As a matter of convenience, it was not necessary to compose the replacement rules pairwise as many of you did. The `compose` command will compose an entire stack of machines. As one of you noted, you could also have written all the replacements in parallel, like this:

```
b Letters -> ... bay, c Letters -> ... day|| Edge _ Edge
```

Also, the directed replacement operator `->@` carries out longest-match replacement, so it would have saved you from specifying `EdgeOfWord` as the right context.

(b)	`bcdefg`	$\leftarrowtail$	{}	($C \ldots$ not ay)
	`quay`	$\leftarrowtail$	{}	($C \ldots Vay$)
	`abcdefg`	$\leftarrowtail$	{`abcdefg`}	($V \ldots$ not ay)
	`aquay`	$\leftarrowtail$	{`aquay`}	($V \ldots Vay$)
	`belay`	$\leftarrowtail$	{`lba`}	($C \ldots Cay$)
	`ebay`	$\leftarrowtail$	{`be`,`ebay`}	($V \ldots Cay$)

(c) Compose three copies of your movement transducer. Each copy moves an initial consonant (if present) to the end of the word.

This is a little tricky, since you need to append a *single* copy of `ay` if 1, 2, or 3 (but not 0) consonants were moved to the end.

A clean solution: Start out by appending `ay` to all words that start with a consonant. Write the movement transducer to replace in the context `EdgeOfWord _ a y EdgeOfWord`.

Other approaches: There are many solutions that start by introducing special symbols, known as **marks**, which are later deleted. For example, one might mark consonant-initial or vowel-initial words and make the subsequent transducers sensitive to these marks. Another way to use marks: have each movement transducer append a mark `Y` after every moved consonant, and then you can fix things up after all the movements, replacing final `Y`'s with `ay` and deleting the others.

(d) No finite-state machine can remember an arbitrarily long string of consonants; it has only finitely many states with which to remember things!

There is a small escape hatch *if* the rest of the word is guaranteed to be short. We are trying to swap the initial consonant string, u, with the rest of the word, v. We have seen that bounded u can be swapped with unbounded v. The converse is also true—at least in a nondeterministic machine: (i) Guess a bounded string v_0 and write it on the output. (ii) Remember what we wrote by going to a state associated with v_0. (iii) Read u and copy it to the output. (iv) Read the real v and crash if $v \neq v_0$.

What FSTs can't do (but Perl regexps can) is to swap unbounded strings. For example, you can argue in the fashion of the pumping lemma that no finite-state transducer can transduce $a^i b^j \mapsto b^j a^i$ for all $i, j \in \mathbb{N}$. Try it!

A couple of you wanted to compose unboundedly many copies of your movement FST. But that does not yield another FST: it is much more powerful. A Turing machine can be represented by a simple FST on strings such as $abbab\mathbf{3}aab$ (representing tape $abbabaab$ in state 3 with the tape head at the position of the **3**). The FST is designed to carry out a single move of the machine (deterministically or not). The composition of unboundedly many copies (if it existed) would compute the function described by the machine, but this is not necessarily a rational function— it may not even halt!