Problems

1. **Compressed English** [30 points]

   In this problem and the following ones, you will be using the open-source OpenFst package from Google. This package does not have as much of the nice regular expression syntactic sugar as the Xerox `xfst` toolkit we read about, but it does support *weighted* finite-state machines. The weights, in this case probabilities on the arcs, will be important in deciding between competing paths.

   You can either follow instructions for downloading and compiling the tools on your own machine, or you can use the pre-compiled versions on `login.ccs.neu.edu`. To do that, modify your PATH environment as follows:

   ```
   export PATH=/home/dasmith/bin:$PATH
   ```

   You will find it *very* helpful to read through the examples on the OpenFst site.

   The OpenFst toolkit, in addition to C++ libraries, provides command line tools for manipulating finite-state acceptors and transducers. Each tool performs a basic operation, like taking the union of two machines, or inverting a transducers, or the composition of two transducers.

   Two noticeable—perhaps noticeably annoying—features of the toolkit are that you need to compile a text specification of a machine into a binary file format and that for speed purposes, all symbols in the input and output alphabets must be **interned**. Interning is a common trick for speeding up software that deals with language data. (It's also used in compilers.) To avoid constantly comparing strings, which might be of arbitrary length, systems hash all unique strings to integers and then just compare integers in constant time. During composition, for instance, where we need to see if the output on an arc in the first transducer matches in the input on an arc in the second, this trick speeds things up a lot. What this means for you is that you need to specify up-front what the input and output alphabets are. If you don't specify them, you may print out a machine only to see the integers for symbols instead of nice human-readable strings.

   For this problem, you will be working with **n-gram models of characters** rather than of words. This makes the models smaller and your machines easier to write by hand. It also means we can just use the symbol table consisting of the fixed ASCII character set for most of the exercises.

a. Were the Greeks right to invent vowels? What if English, like Hebrew, Aramaic, Arabic, and some other languages, left vowels to be supplied by the reader?

   Write a transducer that removes the vowels `aeiou` (not `y`) from any string of the 26 lowercase letters and space. (Note: By convention, OpenFst uses `<space>`, including those angle brackets, for the space character and uses `<epsilon>` for the empty string.)

   Specify your transducer in text form in a file called `unvowel.txt`. Compile it into binary form with the command:

```
fstcompile --isymbols=ascii.syms --osymbols=ascii.syms --
keep_isymbols --keep_osymbols < unvowel.txt > unvowel.fst
```

   I've compiled a finite-state acceptor for the first 100 or so characters of the Declaration of Independence into `declaration.fst`. You can print it out like so:

```
$ fstprint declaration.fst
0       1       w       w
1       2       h       h
2       3       e       e
3       4       n       n
4       5       <space> <space>
5       6       i       i
6       7       n       n
7       8       <space> <space>
8       9       t       t
9       10      h       h
10      11      e       e
11      12      <space> <space>
...
```

   If you compose `unvowel.txt` with this string, project to the lower language, and remove unnecessary epsilon transitions, you should see this:

```
$ fstcompose declaration.fst unvowel.fst | fstproject --
project_output | fstrmepsilon | fstprint
0       1       w       w
1       2       h       h
2       3       n       n
3       4       <space> <space>
4       5       n       n
5       6       <space> <space>
6       7       t       t
```

```
7       8       h       h
8       9       <space> <space>
...
```

b. The `unvowel` transducer is a deterministic *function* and could easily have been written in your favorite programming language. Where finite-state methods really shine is in the ability to automatically manipulate these machines. You can trivially **invert** your transducer, for instance, turning the output to the input, e.g. `fstinvert unvowel.fst`. This inverted transducer would take unvowelled text and nondeterministically insert vowels into it. Since there are an infinite number of ways to insert vowels, we need some criterion for deciding which re-vowelings are best.

   This is where weighted (character) n-gram language models come in. We've provided models for n = 1, 2, 3, 5, and 7 as `c1.fst`, `c2.fst`, and so on. Compose your inverted unvoweler with each of these language models to produce `revowel1.fst`, `revowel2.fst`, and so on.

   The final pipeline for unvoweling and revoweling with a 7-gram model will look like:

```
fstcompose declaration.fst unvowel.fst | fstproject --
project_output | fstrmepsilon | fstcompose - revowel7.fst | \
    fstshortestpath | fstproject --project_output | fstrmepsilon
| fsttopsort | \
    fstprint
```

   For each of the n-gram levels 1, 2, 3, 5, and 7, how many words have been correctly revoweled? What's wrong with n=1?

c. Perhaps removing all vowels was a bit drastic. Many writing systems will also indicate initial vowels, final vowels, or perhaps all long vowels. (Note also the psycholinguistic research showing that text is still quite legible if the first and last characters of words are intact and other characters are removed.) You should write another transducer, called `squash.txt`, to leave the first and last characters of each word alone, and remove only *internal* vowels. The first three words would thus be `Whn in the`.
d. Construct similar unsquashing transducers for the `squash` transformation at all n-gram levels as above. How many words have been correctly unsquashed?

2. **Word Ladders** [30 points]

Lewis Carroll, a logician most famous as the author of *Alice in Wonderland*, invented a game called "word ladders" or "doublets". Players would be given a pair of words such as *cold* and *warm* or *ape* and *man*. They would need to construct a series of single-letter substitutions to transform one word into the other, such that each intermediate form was also a valid word.

In this problem, you will consider a more open-ended problem: from a given start word, *how many* valid words are reachable in one, two, or three edits. For purposes of this problem, the set of lower-case words with at least four letters from *Alice in Wonderland* will form the lexicon. This lexicon, in the form of a finite-state machine, is in the file `alice4up.fst`.

.

a. Write an FST that performs a single character substitution. Using this FST and the *Alice* lexicon, list all the words one, two, and three steps away in the Word Ladder game from *cold*. Submit the FST, your command pipeline, and these three lists. [Hint: `fstshortestpath --nshortest=?` could be useful in answering this question, but you could also write your own program to list the output words.]
b. Generalize the game to allow a single insertion or deletion at each step in addition to a substituion. Again, list all the valid words one, two, and three steps away from *cold*. Submit the FST, your command pipeline, and these three lists.