Language and Computation

LING 227 01 / 627 01 / PSYC 327 01
Assignment #3
Due: March 04, 2014

The solutions to the problem set must be handed in on paper at the beginning of the class. It must be typed (printed), no handwritten solution will be accepted. (Learning how to type mathematical formulae and pseudo-code is part of the skills one has to acquire.)

Additionally, you will also send your Python code in an email by 4 pm on the day of the deadline to `tamas.biro@yale.edu`. The file name must contain both your first name and last name.

Each problem set is worth 10 points in total.

Problem 1: Minimal Cost Paths in Weighted Finite State Automaton (4 points)

Probabilistic Finite State Automata, already introduced informally in the lecture of 02/20, are non-deterministic automata with arcs also labelled with probabilities. While probabilities in a probabilistic FSA must sum up to 1 (which probabilities?), the more widespread **Weighted FSAs** relax this requirement: any positive and negative weights can be written on their arcs. A weighted FSA not only reads a string on its (input) tape, but it also sums up the weights along the transition arcs. Therefore, a weighted FSA can be seen as a mapping from Σ^* to the set of real numbers $\mathbb{R}$.[1]

Although slight variations exist, here is a formal definition that will serve us perfectly:

Def 1. *A **Weighted Finite State Automaton** over the set of real numbers $\mathbb{R}$ is a sextuple $(\Sigma, Q, q_0, q_F, \delta, w)$ such that*

1. *Σ is a finite* alphabet *(a.k.a.* label set*),*
2. *Q is a finite* set of states*,*
3. *$q_0 \in Q$ is the* initial state*,*
4. *$q_F \in Q$ is the* final state*,*
5. *$\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is a* transition function *(a.k.a.* transition table*, or a* set of transitions*).*
6. *$w : Q \times \Sigma \times Q \to \mathbb{R}$ is a* weight function*.*

[1] One can also introduce probabilistic/weighted FSTs with arcs that contain character pairs *and* probabilities/weights.

A few notes:

- FSAs, FSTs, and by analogy, WFSAs are usually defined with a set F of final (accepting) states. Now, however, we suppose a WFST has a single final (accepting) state, for the sake of simplicity and analogy with Markov Models. Observe that with the addition of ϵ-transitions, any FSA can be transformed into an FSA with a single final state.
- The transition function δ maps a combination of current state q_1 and character read from the input tape i onto a subset of Q: these are the states to which the WFST can move from q_1, when reading i from the tape. Observe that, unlike in most definitions, we do not allow for ϵ-transitions ($\delta(q_1, \epsilon)$ is not defined), which is a significant restriction made for the sake of simplicity.
- The weight function $w(q_1, i, q_2)$ specifies the **cost** of moving from q_1 to q_2 when reading i from the input tape.[2] Importantly, $w(q_1, i, q_2)$ is defined for all q_1, i and q_2 such that $q_2 \in \delta(q_1, i)$.

A WFSA rejects input strings similarly to FSAs: either there is no transition corresponding to the current state and current character on the input tape, or the end state of the FSA after having read the entire input string is not a final (accepting) state. If, however, the WFSA does accept the input string, then the transition costs sum up to a total cost. If string $\sigma = \sigma_1\sigma_2 \ldots \sigma_n$ is accepted by the WFSA, then there exists a series of $q_i \in \delta(q_{i-1}, i)$, and $q_n = q_F$, while

$$\text{cost}(\sigma) = \sum_{i=1}^{n} w(q_{i-1}, \sigma_i, q_i)$$

The story becomes interesting when the string σ can be accepted in different ways, because the FSA underlying the WFSA is non-deterministic. In this case, the different accepting paths for string σ may have different costs. We are interested in finding the **minimal cost accepting path**.

Remember that the ND-Recognize algorithm stops whenever some accepting path has been found. Now we will have to consider all paths. By modifying the *Viterbi Algorithm*, you can easily solve

Your task: develop an algorithm that finds the **minimal cost accepting path** for a given WFSA conform to our definition above.

You will submit a **pseudo-code** with some explanation. **Explain** how WFSAs relate to Markov Models, and in what respect(s) you had to modify the Viterbi Algorithm in order to adapt it to WFSAs. Moreover, I wrote above "for the sake of simplicity" several times: did you—and if so, where—make use of these simplifying assumptions?

[2] Alternative definitions introduce *finite set of transitions* as $E \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \mathbb{R} \times Q$. In this approach, several weights can be associated to the same (q_1, i, q_2) triplet, but for our purposes, we can safely ignore the weights that are not minimal. Moreover, some replace $\mathbb{R}$ with any semiring $\mathbb{K}$. Some also introduce an *initial weight* and a *final weight*.

Problem 2: Text Normalization (6 points)

J&M section 8.1 describes the complexity of *text normalization*. Your task is to write a (Python) program that transforms sentences into a sequence of phones.

For *dictionary lookup*, you will use the *CMU Pronouncing Dictionary.* It is freely available and comes with the *NLTK* module in Python. It is described by section 8.2.1 of your textbook, and you will also find a chart with the ARPAbet symbols on the closing inner cover of the book.[3]

The CMU Pronouncing Dictionary (the word 'dictionary' used in its everyday sense) can be accessed in different formats, a Python `dictionary` being one of them. Here is an example of how it works:

```
>>> from nltk.corpus import cmudict
>>> cmudict.dict()["apple"]
[['AE1', 'P', 'AH0', 'L']]
>>> D=cmudict.dict()
>>> D["apple"]
[['AE1', 'P', 'AH0', 'L']]
```

Remember that all words looked up in the dictionary, as a dictionary key, must be in lower case.

Your task: Write a Python program that converts a text (such as `apple`) into a series of phonemes, as returned by `cmudict` (such as `['AE1', 'P', 'AH0', 'L']`). The input is read from the `standard input`, containing, for instance, one sentence per line. The output will be written to the `standard output`, one word per line.

Here is an example of the expected format:

```
tamas@tamas-laptop:~/course/LC-assignment-3\$ python test.py
26 letters from A to Z
['T', 'W', 'EH1', 'N', 'T', 'IY0']
['S', 'IH1', 'K', 'S']
['L', 'EH1', 'T', 'ER0', 'Z']
['F', 'R', 'AH1', 'M']
['EY1']
['T', 'UW1']
['Z', 'IY1']
tamas@tamas-laptop:~/course/LC-assignment-3\$ cat testset | python test.py
['T', 'W', 'EH1', 'N', 'IY0']
['S', 'IH1', 'K', 'S']
etc.
```

[3]For further information: `http://www.nltk.org/_modules/nltk/corpus/reader.html` and `http://www.nltk.org/_modules/nltk/corpus/reader/cmudict.html` More on the Carnegie Mellon Pronouncing Dictionary [cmudict.0.6]: `ftp://ftp.cs.cmu.edu/project/speech/dict/`, and especially `http://www.nltk.org/book/ch02.html`, "A pronouncing dictionary".

In order for us to easily check the output of your program, please keep the brackets and the quotation marks. Remember also to print each word in a new line.

Exactly one pronunciation should be provided for each word, even if CMU-Dict offers several of them. How you choose between them is something for you to find out. If the input contains a word that CMUDict cannot pronounce, then return `<>`, still a better solution than an exception.

You will write as much comment as possible into your program code.

On paper, you will submit your program code (including comments), and some additional explanation: How is your program built up, which problems in text normalization have you considered, what challenges have you encountered? If there were problems that you considered but decided not to tackle, this is also the place for you to mention them. (You may decide that some problems are too complicated to be solved within this problem set. Then, although you might lose points for performance, you can still gain points here.) This part of your solution is worth **2 points**.

Then, you will also email me your code, which is worth another **2 points**. Please make sure you let us know if you have used Python 2 or Python 3. We will check whether your code is written in a good programing style, and whether it runs faultlessly.

Finally, the performance of your program will be checked on a testset, not to be revealed before the deadline (in order to simulate real life situations and challenges in computational linguistics). This testset will contain examples similar to the difficulties discussed in section 8.1 of your textbook (see also the example above, which includes a number and letter names), without being extremely nasty to you. Your performance on this testset—your program's ability to solve most of these complexitities—is worth yet another **2 points**.

For those of you new to Python, here is the skeleton of a program that reads from the `standard input` as long as something is provided:

```
while (True):
    l=raw_input("? ")   # read a line from the standard input
    if not l :          # if line is empty,
        print("bye")    # then, be polite and
        break           # leave. Otherwise
    # do something else, for example,
    print(l)
```