

Homework #1: CMPT-825

Reading: <http://www.research.att.com/~fsmtools/fsm/>

Anoop Sarkar – anoop@cs.sfu.ca

(1) Machine (Back) Transliteration

Languages have different sound inventories. When translating from one language to another, names, technical terms and even some common nouns are routinely transliterated. *Transliteration* means the replacement of a loan word with some approximate phonetic equivalents taken from the sound inventory of the language. These phonetic equivalents are then written in the script of the language.

For example, the word “computer” in English comes out sounding as “konpyutaa” in Japanese, which is written using the katakana syllabic script typically used for loan words. Here are some more examples of transliteration:

Angela Johnson

アンジラ・ジョンソン

(anjira jyonson)

New York Times

ニューヨーク・タイムズ

(nyuuyooku taimuzu)

ice cream

アイスクリーム

(aisukuriimu)

Your task is to back transliterate from Japanese into English. To avoid dealing with encoding issues with the Japanese script, we will assume that the input will be Japanese sound sequences rather than katakana characters. There is a simple mapping between katakana and these sound sequences shown in Fig. 1.

ア (a)	カ (ka)	サ (sa)	タ (ta)	ナ (na)	ハ (ha)	マ (ma)	ラ (ra)
イ (i)	キ (ki)	シ (shi)	チ (chi)	ニ (ni)	ヒ (hi)	ミ (mi)	リ (ri)
ウ (u)	ク (ku)	ス (su)	ツ (tsu)	ヌ (nu)	フ (fu)	ム (mu)	ル (ru)
エ (e)	ケ (ke)	セ (se)	テ (te)	ネ (ne)	ヘ (he)	メ (me)	レ (re)
オ (o)	コ (ko)	ソ (so)	ト (to)	ノ (no)	ホ (ho)	モ (mo)	ロ (ro)
バ (ba)	ガ (ga)	パ (pa)	ザ (za)	ダ (da)	ア (a)	ヤ (ya)	ャ (ya)
ビ (bi)	ギ (gi)	ピ (pi)	ジ (ji)	デ (de)	イ (i)	ヨ (yo)	ョ (yo)
ブ (bu)	グ (gu)	プ (pu)	ズ (zu)	ド (do)	ウ (u)	ユ (yu)	ュ (yu)
ベ (be)	ゲ (ge)	ペ (pe)	ゼ (ze)	ン (n)	エ (e)	ヴ (v)	ッ
ボ (bo)	ゴ (go)	ポ (po)	ゾ (zo)	チ (chi)	オ (o)	ワ (wa)	ー

Figure 1: Mapping from Japanese katakana characters to pronunciations.

In order to generate a Japanese sound sequence for a given English word sequence we take the following intermediate steps:

1. Represent all valid English words that can be used in an English word sequence.
2. Convert each English word to its pronunciation (mapping English words to phonemes). For example, the English pronunciation of the word sequence *golf ball* is G AA L F B AO L.
3. Convert English pronunciations to their equivalent Japanese pronunciations. For example, the sound sequence G AA L F B AO L would be converted to the equivalent Japanese sound sequence g o r u h u b o o r u.

Finally we can invert the above steps so that it takes a Japanese sound sequence as input and produces English word sequences as output.

We can represent each of the above steps as a finite-state transducer. We will do this using the AT&T fsm toolkit. The advantage is that we can use the standard transducer operations which have been implemented in this toolkit. For example, we can use the program `fsmcompose` to compose the finite state transducers described above into a single transducer which computes the conversion from Japanese sound sequences into English words.

In general for this approach to be useful for text-to-text back transliteration we will have to build a mapping from Japanese sound sequences into the katakana script, but we will not do that for this assignment.

You will be using the following data files for each intermediate step in the process of mapping English words to Japanese sound sequences:

- `sample-engwords.txt` – a small set of English words used for this assignment.
- `cmudict` – The CMU pronunciation dictionary. Only use the pronunciations for words in the file `sample-engwords.txt`
- `epron-jpron.map` – Mapping from English pronunciations to Japanese pronunciations (this mapping is given to you but think about how we could infer this mapping from data).
- `jpron-input1.txt` and `jpron-input2.txt` – Sample input files. Contains pronunciations of a small set of Japanese words.

In order to do back transliteration from Japanese sound sequences into English words, you will have to convert the above data into the AT&T fsm toolkit format. You will need to create a text file with the finite state machine/transducer and an additional file with the mapping between the input/output symbols used and a unique number for each symbol (the `.syms` file). *An important thing to keep in mind is that the same token must get the same number in the .syms file across all these finite-state transducers.* This is how the output alphabet of one transducer is recognized to be the same as the input alphabet of another transducer.

This conversion is not entirely trivial. For example, consider the fragment of the pronunciation dictionary given below:

```
GOLF  G AA1 L F
BALL  B AO1 L
```

The numbers 0, 1, 2 are to be removed from this input (they represent stress markers which are not needed for this task). The first column is the English word and the remaining columns contain the pronunciation for that word.

Once you write your code to convert the above format into the AT&T fsm toolkit, the transducer text files should look like this (in the symbols file the number 0 is reserved for the empty string):

tiny.txt:	tiny.syms:
0 2 <eps> G	<eps> 0
2 3 <eps> AA	PAUSE 1
3 4 <eps> L	G 2
4 5 <eps> F	AA 3
5 1 GOLF <eps>	L 4
0 7 <eps> B	F 5
7 8 <eps> AO	GOLF 6
8 9 <eps> L	B 7
9 1 BALL <eps>	AO 8
1 0 <eps> <eps>	BALL 9
1 0 <eps> PAUSE	
1	

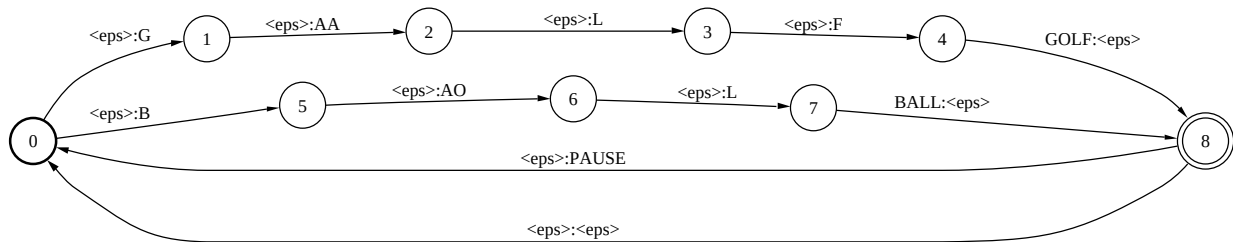


Figure 2: FST drawn using: `fsmdraw -i tiny.syms -o tiny.syms tiny.fsm | dot -Tps > tiny.ps`

Once you have these two files you can compile it to the binary format used by the AT&T fsm toolkit by using the program `fsmcompile`.

```
fsmcompile -t -i tiny.syms -o tiny.syms tiny.stxt > tiny.fsm
```

The `-t` indicates that the input file is a transducer and the `-i` and `-o` options indicate the symbols for the input and output alphabet respectively. The transducer is shown in Fig. 2.

Once you have compiled all the individual finite state transducers for each model, you should use the command `fsmcompose` to combine them together into one big model which maps Japanese sounds into English words. Once this is done you can then compose this combined fsm file with a particular input Japanese sound sequence using `fsmcompose` once again. To extract the English word sequence you will need to use the following programs from the fsm toolkit (they are implementations of standard transducer algorithms):

- `fsmrmepsilon fstfile`: removes epsilon transitions from `fstfile`
- `fsmprint -i symsfile -o symsfile fstfile`: prints out the compiled `fstfile` as text
- `fsmproject -1 fstfile`: projects the transducer `fstfile` to a finite-state machine over the input language
- `fsmproject -2 fstfile`: projects the transducer `fstfile` to a finite-state machine over the output language

Produce the English output for the example input Japanese sound sequences `anjirajyonson` and `goruhubooru` (use the files: `jpron-input1.txt`, `jpron-input2.txt` and their syms file: `jpron-syms.txt`).

(2) Minimum Edit Distance

The following Python code computes the minimum number of edits: insertions, deletions, or substitutions that can convert an input source string to an input target string. Using the cost of 1, 1 and 2 for insertion, deletion and replacement is traditionally called Levenshtein distance.

```
def distance(target, source, insertcost, deletecost, replacecost):
    n = len(target)+1
    m = len(source)+1
    # set up dist and initialize values
    dist = [ [0 for j in range(m+1)] for i in range(n+1) ]
    for i in range(1,n):
        dist[i][0] = dist[i-1][0] + insertcost
    for j in range(1,m):
        dist[0][j] = dist[0][j-1] + deletecost
    # align source and target strings
    for j in range(1,m):
        for i in range(1,n):
```

```

        inscost = insertcost + dist[i-1][j]
        delcost = deletecost + dist[i][j-1]
        if (source[j-1] == target[i-1]): add = 0
        else: add = replacecost
        substcost = add + dist[i-1][j-1]
        dist[i][j] = min(inscost, delcost, substcost)
    # return min edit distance
    return dist[n-1][m-1]

if __name__=="__main__":
    from sys import argv
    if len(argv) > 2:
        print "levenshtein distance =", distance(argv[1], argv[2], 1, 1, 2)

```

Let's assume we save this program to the file `distance.py`, then:

```

$ python2.5 distance.py gamble gumbo
levenshtein distance = 5

```

Your task is to produce the following visual display of the best (minimum distance) alignment:

```

$ python2.5 view_distance.py gamble gumbo
levenshtein distance = 5
g a m b l e
|   |   |
g u m b _ o

$ python2.5 view_distance.py "recognize speech" "wreck a nice beach"
levenshtein distance = 14
_ r e c _ _ o g n i z e   s p e e c h
| | |           | |   | |       |   | |
w r e c k   a   n i c e   _ b e a c h

$ python2.5 view_distance.py execution intention
levenshtein distance = 8
_ e x e c u t i o n
      |       | | | |
i n t e _ n t i o n

```

The 1st line of the visual display shows the *target* word and the 3rd line shows the *source* word. An insertion in the target word is represented as an underscore in the 3rd line aligned with the inserted letter in the 1st line. Deletion from the source word is represented as an underscore '_' in the 1st line aligned with the corresponding deleted character in the source on the 3rd line. Finally, if a letter is unchanged between target and source then a vertical bar (the pipe symbol '|') is printed aligned with the letter in the 2nd line.

You can produce this visual alignment using two different methods:

- Memorize which of the different options: insert, delete or substitute was taken as the entries in the table are computed; or
- Trace back your steps in the table starting from the final distance score by comparing the scores from the predecessor of each table entry and picking the minimum each time.

There can be many different alignments that have exactly the same minimum edit distance. Therefore, for the above examples producing a visual display with a different alignment but which has the same edit distance is also correct.

- (3) Print out all the valid alignments with the same minimum edit distance. For longer input strings, you should print out only the first N alignments (where N has to be set by the user) because the number of possible alignments is exponential in the size of the input strings.

We can see this by considering a recursive function that prints out all alignments (instead of using the dynamic programming approach). Let us call this function *align*. Let us assume that the two input strings are of length n, m . Then, the number of recursive calls can be written as a recurrence:

$$\text{align}(n, m) = \text{align}(n, m - 1) + \text{align}(n - 1, m - 1) + \text{align}(n - 1, m)$$

Let us assume $n = m$, then:

$$\begin{aligned} \text{align}(n, n) &= \text{align}(n, n - 1) + \text{align}(n - 1, n - 1) + \text{align}(n - 1, n) \\ &= 2 \cdot \text{align}(n, n - 1) + \text{align}(n - 1, n - 1) \\ &= 2 [\text{align}(n, n - 2) + \text{align}(n - 1, n - 2) + \text{align}(n - 1, n - 1)] + \text{align}(n - 1, n - 1) \\ &> 3 \cdot \text{align}(n - 1, n - 1) \end{aligned}$$

Thus, each call to the function *align*(n, n) results in three new recursive calls. The number of times the align function will be called is 3^n which is a bound on the total number of distinct alignments.

- (4) Use the AT&T fsm toolkit to implement finding the minimum edit distance for the examples provided in Q 2.