Homework #2: CMPT-413

Anoop Sarkar
http://www.cs.sfu.ca/~anoop

Only submit answers for questions marked with †. For the questions marked with a ††; choose one of them and submit its answer.

Important! To solve this homework you must read the following chapters of the NLTK Book, available online at *http://www.nltk.org/book*

- Chapter 3. Processing Raw Text
- Chapter 4. Writing Structured Programs

(1) The CMU Pronunciation Dictionary (cmudict) contains North American English pronunciations using ARPABET phoneme transcriptions for over 125,000 words. You can view the phoneme inventory and details about the dictionary using the following Python code:

```
from nltk.corpus.reader import cmudict
# print out the documentation string for module cmudict
print cmudict.__doc__
```

The following code explains the basic details of how the cmudict module in nltk can be used to access the pronunciation dictionary:

```
ffrom nltk.corpus import cmudict
for word, pron in cmudict.entries():
    if (pron[-4:] == ['P', 'IH0', 'K', 'S']):
        print word.lower(),
print
transcr = cmudict.dict() # warning: this can be very slow
print transcr['python']
```

Running it produces the output:

```
epic's epics olympic's olympics opic's topics tropics
[['P', 'AY1', 'TH', 'AA0', 'N']]
```

Each word can have multiple pronunciations, for example, the word `ajito` has two pronunciations in cmudict:

```
ajito 1 AH0 JH IY1 T OW0
ajito 2 AH0 HH IY1 T OW0
```

Print out the pronunciation for the word `store` from cmudict. The Speech Accent Archive at *http://accent.gmu.edu/* has speech files from speakers around the world. Listen to an audio clip spoken by a speaker from Boston, Massachusetts, USA. Write down the pronunciation in ARPABET notation of the word `store` as pronounced by a speaker from Boston.

(2) † Read a number *n* from standard input and print out all the entries in cmudict that have *n* or more different pronunciations after you remove the stress markers. Two variant pronunciations for `forecasts`: `F-AO0-R-K-AE1-S-T-S` and `F-AO1-R-K-AE2-S-T-S` are identical once you remove the stress markers. Print out all the variant pronunciations in the following format:

```
terrorists T-EH-R-ER-IH-S-T-S
terrorists T-EH-R-ER-IH-S
terrorists T-EH-R-ER-AH-S-T-S
terrorists T-EH-R-ER-IH-S-S
```

(3) † **FST Recognition**

Implement the FST recognition algorithm. It should print True if the input pair of strings is accepted by the FST, and print False otherwise. You must use the `fst` module provided to you (assume that "`from fst import fst`" will load the module). The FST to use for this question is provided in the file `fst_example.py`. Extend the FST class using inheritance so that you only need to add the recognize function. The file `fst_recognize_stub.py` shows how you can write a new class that inherits from the `fst` class. You have to implement the recognition algorithm rather than call the existing transduce function to infer the recognition output value.

Your program should read the input and output strings for the FST from standard input, one per line. Your program should also show the steps of the recognize algorithm. Look at the testcases directory for examples of the output expected. The FST that is used for all test cases is the one defined in the `fst_example.py` file.

(4) Implement FST composition and add this functionality to your modified FST implementation defined in Q. 3. The two FSTs to be composed are provided in `fst_compose_stub.py` which also specifies the output.

(5) † **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)] for i in range(n) ]
    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.

(6) Print out the valid alignments with the same minimum edit distance. You should print out the first 100 alignments or *N* alignments, where *N* comes from a command line argument -n. This is essential 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:

$$align(n,m) = align(n,m-1) + align(n-1,m-1) + align(n-1,m)$$

Let us assume $n = m$, then:

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

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.

(7) †† **Sino-Korean Number Pronunciation**

The Korean language has two different number systems. The native Korean numbering system is used for counting people, things, etc. while the Sino-Korean numbering system is used for prices, phone numbers, dates, etc. (called Sino-Korean because it was borrowed from the Chinese language).

Write a program that takes a number as input and produces the Sino-Korean number pronunciation appropriate for prices using the table provided below. The program should accept any number from 1 to 999, 999, 999 and produce a single output pronunciation. The commas are used here to make the numbers easier to read, and can be simply deleted from the input.

You should produce the Korean romanized output (Korean written using the Latin script) as shown in the table, rather than the Hangul script (the official script for the Korean language).

1	il	10	sib	19	sib gu	100	baek
2	i	11	sib il	20	i sib	1,000	cheon
3	sam	12	sib i	30	sam sib	10,000	man
4	sa	13	sib sam	40	sa sib	100,000	sib man
5	o	14	sib sa	50	o sib	1,000,000	baek man
6	yuk	15	sib o	60	yuk sib	10,000,000	cheon man
7	chil	16	sib yuk	70	chil sib	100,000,000	eok
8	pal	17	sib chil	80	pal sib		
9	gu	18	sib pal	90	gu sib		

For example, the input 915, 413 should produce the output *gu sib il man o cheon sa baek sib sam*. Note that as shown the table above 1000 is pronounced *cheon* rather than *il cheon*, and so 111, 000 is pronounced as *sib il man cheon*. The testcases directory contains sample inputs and outputs.

An intermediate representation makes this task much easier. For input 915, 413 consider the intermediate representation of `9[10]1[10^4]5[10^3]4[10^2][10]3#`, where `[10]`, `[10^4]` are symbols in an extended alphabet. The mapping from numbers to intermediate representation and the mapping into Korean pronunciations are both easily implemented as finite-state transducers. The intermediate forms for an FST implementation are provided in the `*.hint` files for each input.

You can use the FST code from Q. 3 and Q. 4 or the more scalable OpenFST toolkit (installed in ~anoop/cmpt413/sw/).

(8) †† **Machine (Back) Transliteration**

Languages have different sound inventories. Full machine translation is a complex task, but a special case occurs when one wants to translate names, technical terms and even some recently introduced common nouns. *Transliteration* is the replacement of a loan word from a source language for names, technical terms, etc. with the approximate phonetic equivalents taken from the sound inventory of the target language. These phonetic equivalents are then written in the script of the language.

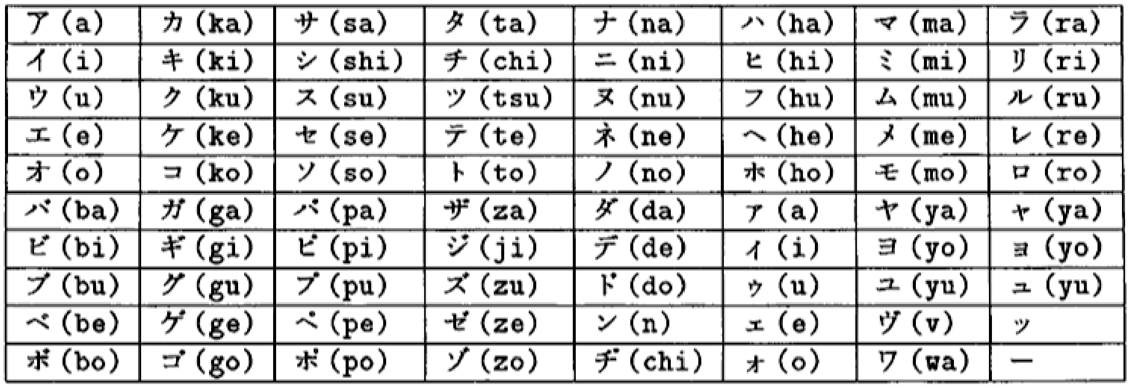
For example, the noun phrase “New York Times” in English which would sound like “niyu yoku taimuzu” in Japanese using the sound system native to the Japanese language, which is then written down using the Katakana syllabic script typically used for loan words as: ニューヨークタイムズ.

Table 1 provides the mapping from Katakana to pronunciations. Use the file `katakana-jpron.map` for your implementation in order to convert the Japanese Katakana input in UTF-8 into Japanese pronunciation symbols in ASCII. Dealing with UTF-8 can be tricky. The file `katakana_jpron_stub.py` contains example Python code for dealing with UTF-8 input.

Your task is to back transliterate from Japanese into English. It is called back transliteration since the original word was borrowed from English into Japanese.

The following resources can be used to solve this task:

1. The file `katakana-jpron.map` maps Katakana input to Japanese pronunciation symbols.



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 (hu)	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)	ァ (a)	ヤ (ya)	ャ (ya)
ビ (bi)	ギ (gi)	ピ (pi)	ジ (ji)	デ (de)	ィ (i)	ヨ (yo)	ョ (yo)
ブ (bu)	グ (gu)	プ (pu)	ズ (zu)	ド (do)	ゥ (u)	ユ (yu)	ュ (yu)
ベ (be)	ゲ (ge)	ペ (pe)	ゼ (ze)	ン (n)	ェ (e)	ヴ (v)	ッ
ボ (bo)	ゴ (go)	ポ (po)	ゾ (zo)	ヂ (chi)	ォ (o)	ワ (wa)	ー

Figure 1: Mapping from Japanese katakana characters to pronunciations.

2. The file `epron-jpron.map` maps English pronunciations to Japanese pronunciations (this mapping is given to you but think about how we could produce this mapping automatically). By convention the English pronunciations are uppercase and the Japanese pronunciations are lowercase. This can be used to convert Japanese pronunciations to their equivalent English pronunciations. For example, the Japanese pronunciation `n i y u` can be converted to the English `N Y UW`.
3. cmudict (using the nltk interface) can be used to convert English pronunciations to English words. Given input `N Y UW` produce the output `new`. Remember to remove stress markers.

The ASCII - symbol in the map files refers to an empty string.

Important: Since cmudict is quite large, your program must read a file called `cmudict_words.txt` from the current directory. This file will contain the English words that you should extract from cmudict. We can add or delete words from this file to increase or decrease the set of target English words.

In addition to the testcases provided, Google Translate (*translate.google.com*) can help you produce additional Japanese Katakana equivalents for simple English proper nouns or common technical terms.

The mapping between pronunciations and from English pronunciations to valid English words are both easily implemented as finite-state transducers. You can use the FST code from Q. 3 and Q. 4 or the more scalable OpenFST toolkit (installed in ~anoop/cmpt413/sw/).

(9) **Re-write Rules**

The Thrax toolkit implements context-dependent rewrite rules: $i : o/\lambda\_\ \rho$ where $i : o$ is a transducer, and *lambda* is the left context and ρ is the right context. The rewrite rules get compiled into finite-state transducers (FSTs) which are compatible with the openfst software. In addition, one can specify if the rewrite rule should apply left to right, right to left or simultaneously. Also, one can specify if the rule application is obligatory or optional. Implement the following rewrite rules using Thrax:

$$\begin{array}{lcl} iN & \rightarrow & im/\_\ (p|b|m) \\ iN & \rightarrow & in/\_ \end{array}$$

where *N* is an abstract nasal phoneme.