1 Instructions

- This is programming project on machine translation but we assume no prior knowledge in any of the languages involved in the examples (Chinese, French, or Spanish), nor familiarity with any linguistic theory.

- The slides from Liang’s presentation are available online under the homework page. Please refer to them for high-level intuitions and visualizations.

- The submission deadline is Saturday May 6th at midnight. Late submission deadline is Sunday May 7th at 3pm (10% off). You should use the `turnin` program as in homeworks.

- This project is worth 100 points (excluding extra credit problems).

2 The Problem - Syntax-Directed Translation

In this project, you will be building a machine translation system that translates from English to some foreign language, say, French. The translation method we use here is syntax-directed translation, which is originated in compiling, where the source program in a high-level language (say, C or Java) is first parsed into a syntax or expression tree which guides the generation of object program (say, in Assembly or Byte-code). We adapt this technique to the translation of human languages, where the source input is first parsed into a parse-tree and then we recursively convert the tree into a string in the target language.

2.1 An Example

Consider the following English sentence and its Chinese translation (note the reordering in the passive construction):

(1) the gunman was killed by the police.
Figure 1 shows how the translation works. The English sentence (a) is first parsed into the tree in (b)1 which is then recursively converted into the Chinese string in (e) through five steps. First, at the root node, we apply the rule \( r_1 \) which preserves the top-level word-order and translates the English period into its Chinese counterpart:

\[
(r_1) \quad S \left( x_1:NP-C \; x_2:VP \; \text{PUNC} \; (\ldots) \right) \rightarrow x_1 \; x_2 \; .
\]

Then, the rule \( r_2 \) grabs the whole sub-tree for “the gunman” and translates it as a phrase:

\[
(r_2) \quad \text{NP-C ( DT (the) NN (gunman) ) } \rightarrow \text{qiangshou}
\]

Now we get a “partial Chinese, partial English” sentence “qiangshou \( \text{VP .} \)” as shown in Fig. 1 (c). Our recursion goes on to translate the VP sub-tree. Here we use the rule \( r_3 \) for the passive construction:

\[
(r_3) \quad \text{VP \ (VBD \ was \ } x_1: \text{VBN} \; \text{PP \ by \ } x_2: \text{NP-C)} \rightarrow \text{bei } \; x_2 \; x_1
\]

which captures the fact that the agent (NP-C, “the police”) and the verb (VBN, “killed”) are always inverted between English and Chinese in a passive voice. Finally, we apply rules:

\[
(r_4) \quad \text{VBN ( killed ) } \rightarrow \text{jibi}
\]

\[
(r_5) \quad \text{NP-C ( DT (the) NN (police) ) } \rightarrow \text{jingfang}
\]

which perform phrasal translations for the two remaining sub-trees in (d), respectively, and get the completed Chinese string in (e).

You might also have other rules which lead to a different translation, say, replacing \( r_5 \) with another Chinese word for “police”:

\[
(r_5) \quad \text{NP-C ( DT (the) NN (police) ) } \rightarrow \text{jingcha}
\]

In this framework, every rule is associated with a probability, and your job is to search for the best translation (the one with the highest probability) given a parse tree.

---

1The internal nodes in the parse tree are called non-terminal tags or syntactic categories. For example, VP stands for verb phrase and PP prepositional phrase. The tags right above leaf nodes are called preterminals or part-of-speech tags. For example, VBD and VBN correspond to verbs in past tense and past participle, respectively. Our tagset and grammar follows the Penn Treebank Style. For further information, please visit:
http://www.cis.upenn.edu/~treebank
http://bulba.sdsu.edu/jeanette/thesis/PennTags.html
(a) the gunman was killed by the police.

Figure 1: A syntax-directed translation process.
2.2 French/Spanish Example

If you have no idea what the previous example is all about, here is another example, but from English to French and Spanish (it’s not a whole sentence).

(2) my friend’s black cat

le chat noir de mon ami
el gato negro de mi amigo
[the] [cat] [black] [of] [my] [friend]

Again, assume the parse tree for the English phrase is:

```
<table>
<thead>
<tr>
<th>NP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>NP</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PRP</td>
</tr>
</tbody>
</table>
```

We can first have a rule which takes care of the reordering of possessive (’s) construction between English and French/Spanish:

\[
(r'_1) \quad \text{NP} \quad \rightarrow \quad x_1 \quad \text{de} \quad x_0
\]

Then we use a rule to translate the English phrase “my friend”: 

\[
(r'_2) \quad \text{NP} (\text{PRP} (\text{my}) \quad \text{NN} (\text{friend}) ) \quad \rightarrow \quad \text{mon} \quad \text{ami}
\]

\[
(r'_3) \quad \text{NP} (x_0:JJ \quad x_1:NN) \quad \rightarrow \quad x_1 \quad x_0
\]

and a rule for reordering the adjective (black) with the noun (cat):

\[
(r'_4) \quad \text{black} \quad \rightarrow \quad \text{noir}
\]

\[
(r'_4) \quad \text{black} \quad \rightarrow \quad \text{negro}
\]
(r’_5) cat → le chat
               el gato

Of course, you may decompose r’_5 into three smaller rules and get the same translations.

2.3 Translation Algorithm

Given a fixed parse-tree τ*, we are to search for the best derivation with the highest probability. This can be done by a simple top-down traversal (or depth-first search) from the root of τ*: at each node η in τ*, try each possible rule r whose English-side pattern t(r) matches the subtree τη* rooted at η, and recursively visit each descendant node η_i in τη* that corresponds to a variable in t(r). We then collect the resulting target-language strings and plug them into the Chinese-side s(r) of rule r, getting a translation for the subtree τη*. We finally take the best of all translations.

With the extended LHS of our transducer, there may be many different rules applicable at one tree node. As a result, the number of derivations is exponential in the size of the tree, since there are exponentially many decompositions of the tree for a given set of rules. This problem can be solved by memoization: we cache each subtree that has been visited before, so that every tree node is visited at most once. This results in a dynamic programming algorithm that is guaranteed to run in O(npq) time where n is the size of the parse tree, p is the maximum number of rules applicable to one tree node, and q is the maximum size of an applicable rule. For a given rule-set, this algorithm runs in time linear to the length of the input sentence, since p and q are considered grammar constants, and n is proportional to the input length. The full pseudo-code is worked out in Algorithm 1.

Algorithm 1 Top-down Memoized Recursion

```
1: function TRANSLATE(η)  
2:     if cache[η] defined then ▷ this sub-tree visited before?
3:         return cache[η]
4:     best ← 0
5:     for r ∈ R do ▷ try each rule r
6:         matched, sublist ← PATTERNMATCH(t(r), η) ▷ tree pattern matching
7:         if matched then ▷ if matched, sublist contains a list of matched subtrees
8:             prob ← Pr(r) ▷ the probability of rule r
9:             for η_i ∈ sublist do
10:                p_i, s_i ← TRANSLATE(η_i) ▷ recursively solve each sub-problem
11:                prob ← prob · p_i
12:         if prob > best then ▷ plug in the results
13:             best ← prob
14:             str ← [x_i → s_i]s(r)
15:     cache[η] ← best, str ▷ caching the best solution for future use
16: return cache[η] ▷ returns the best string with its prob.
```
3 I/O Specifications and Samples

There are two input files:

- **input.txt** - English parse trees, which contains several lines, with each line representing one parse tree.

- **rules.txt** - rules, one line for each rule in the format of

  \[
  lhs \rightarrow rhs \text{ ### prob=} p
  \]

  where \(lhs\) is the left-hand-side (English) tree pattern (trees with variables), and \(rhs\) is the right-hand-side (Foreign) string (with the same set of variables, possibly permutated). The variables will be in the form of \(x_0, x_1, \) and so on. The real number \(p\) is the probability of this rule.

Sample input files:

**input.txt:**

NP(DP(NP(PRP("my") NN("friend")) POS("'s")) NP(JJ("black") NN("cat")))
NP(DP(NP(PRP("my") NN("friend")) POS("'s")) NP(JJ("white") NN("cat")))

**rules.txt:**

NP(DP(x0:NP POS("'s")) x1:NP) \rightarrow x1 "de" x0 ### prob=1.0
NP(PRP("my") NN("friend")) \rightarrow "mon" "ami" ### prob=0.51
NP(PRP("my") NN("friend")) \rightarrow "mon" "amie" ### prob=0.49
NP(x0:JJ x1:NN) \rightarrow x1 x0 ### prob=0.7
NP(x0:JJ x1:NN) \rightarrow x0 x1 ### prob=0.3
JJ("black") \rightarrow "noir" ### prob=0.6
JJ("black") \rightarrow "noire" ### prob=0.4
NN("cat") \rightarrow "le" "chat" ### prob=1.0
NP(x0:JJ NN("cat")) \rightarrow "le" "chat" x0 ### prob=1
NP(JJ("black") x0:NN) \rightarrow x0 "noir" ### prob=0.55
NP(JJ("black") x0:NN) \rightarrow x0 "noire" ### prob=0.45

To run the program, type in the terminal

```
cat input.txt | ./translate -r rules.txt
```

and you will get the output

```
my friend 's black cat -> le chat noir de mon ami ### prob=0.306
my friend 's white cat -> *** failed ***
```

The output contains one line for each English tree in **input.txt**. As shown in the above example, the format is quite similar to the rules file, except there are no quotes, and the probability of the best translation is printed to the 3rd decimal digit. To do this, you might wanna use Python’s string formatting (borrowed from C’s `printf()`):
print "prob=\%.3f" % bestprob

or, for the whole line:

print "%s -> %s ### prob=\%.3f" % (lhs_str, besttrans, bestprob)

Note:

1. the `-r` switch is mandatory: if it is missing, simply print a “mini-usage” and exit, as in HW3 problem 1.

2. if the search fails, simply print *** failed *** in place of the best translation. (see the above example).

3. each individual word in the input (e.g., "my") is surrounded by a pair of double quotes, and we assume that white spaces, double quotes, or parentheses are not part of any word.

4. The sample case is available online as input1.txt, rules1.txt, and output1.txt under the homework page, where you can also find a more complicated example.

4 Modules

Note: This is just a recommended design. You are encouraged to implement it from scratch.

4.1 Tree class

A separate class for trees is highly recommended. It should work for both parse-trees and tree patterns (trees with variables).

4.2 Parsing the trees in the input

This might be the most difficult part of this project. You may want to write a “pretty-print” function inside the Tree class to help you debug this part.

4.3 Tree Pattern Matching

The function patternmatch(tree, pattern) takes a parse tree and a tree pattern as input and returns a pair (matched, sublist), where matched is a boolean (whether the matching succeeds or not), and sublist is a list of matched subtrees in case of success. See Algorithm 1.

4.4 Rule class

This is optional. In fact, a tuple of (tree_pattern, rhs, prob) would be enough.
4.5 search

This module implements the memoized depth-first search (Algorithm 1).

5 Extensions (Extra Credit Problems)

5.1 Outputing the derivation (10 points, Easy)

What if we want to look into the derivation (see Figure 1)? How is the English sentence translated to the foreign sentence and what rules are used?

We add another option `-d` to the command line for outputing the derivation of the best translation (if any). For example,

```
cat input1.txt | ./translate.py -r rules1.txt -d
```

which should output the following:

```
my friend ’s black cat -> le chat noir de mon ami ### prob=0.306
NP (DP (x0:NP POS (’s)) x1:NP) -> x1 de x0 ### prob=1.000
   | x0: NP (PRP (my) NN (friend)) -> mon ami ### prob=0.510
   | x1: NP (x0:JJ NN (cat)) -> le chat x0 ### prob=1.000
   | | x0: JJ (black) -> noir ### prob=0.600
   | black cat -> le chat noir ### prob=0.600
my friend ’s black cat -> noir ### prob=0.600
my friend ’s white cat -> *** failed ***
```

This output is available online as `output1-d.txt`. A derivation basically outputs all the rules involved in a tree-like format. For each subtree, we first output the top-level rule, e.g.

```
NP (DP (x0:NP POS (’s)) x1:NP) -> x1 de x0 ### prob=1.000
```

and then recursively output the sub-derivations for the subtrees corresponding to the two variables (x0 and x1) with one more level of indentation. Finally, we output the best translation for this subtree in the standard format.

There is a more complicated example online.

5.2 Rule Indexing (5 points, Easy)

In line 5 of Algorithm 1, we enumerate each rule and check if its LHS tree-pattern matches the current tree. This is rather inefficient in general when we have a very large rule set. However, you can index the rules using a dictionary, where the keys are the top-level productions of the tree pattern. For example, for the rule

```
NP (DP (x0:NP POS (’s)) x1:NP) -> x1 de x0 ### prob=1.000
```

the top-level production is

```
NP => DP NP
```
Now when you are translating a tree $T$ whose top-level production is $\gamma$, you only need to look at rules indexed under $\gamma$ (because other rules are certainly incompatible with $T$).

You can further group rules with same LHS tree pattern together, as a second-level of indexing, which will save some time by reducing redundant calls of `PatternMatch()`.

### 5.3 Branch-and-Bound Search (15 points, Intermediate)

The efficiency of Algorithm 1 can be significantly improved by branch-and-bound. Consider if we already have one translation $t$ with probability 0.6 for a tree, and we are now searching for other possible (better) translations. If we proceed with a rule that has probability 0.5, there is clearly no hope that this will end up with a better translation. Or, if we use another rule $r'$ with probability 0.8, then there is still hope to beat the current translation $t$, although we only have $0.6/0.8 = 0.75$ for the rest of the translation. In other words, the product of sub translations should be larger than 0.75 in order to beat $t$. Suppose $r'$ has two variables $x_0$ and $x_1$, and we know (by recursion) the best translation for the $x_0$ subtree has probability 0.5, then there is no need to search for $x_1$ either.

This trick is also inspired from the $\alpha - \beta$ pruning in Game AI (see Steve’s slides for adversarial search).

### 5.4 Alternative translations (only for fun and/or research, Very Challenging)

Clearly, there are many possible translations given a ruleset and an input tree. Can you also output the 2nd-best, 3rd-best, or up to $k$th-best translations? We add another option `-k` to the command line to request “top-k” translations. For example,

```
cat input1.txt | ./translate.py -r rules1.txt -k 3
```

will output

```
my friend’s black cat -> le chat noir de mon ami ### prob=0.306
my friend’s black cat -> le chat noir de mon amie ### prob=0.294
my friend’s black cat -> le chat noire de mon ami ### prob=0.204
my friend’s white cat -> *** failed ***
```

Note: do not print duplicate translations (say, there might be another way of deriving the best translated string). Be sure to include in `debrief.txt` your algorithm or thoughts if you attempted this question.

### 6 debrief.txt

1. How many hours did you spend on this project?

2. If you did any extra credit problem, describe your algorithms or methods.