CS 544, Natural Language Processing, Spring 2010
HW 6: Grammar and Parsing

Liang Huang

due: **electronic version only** at lhuang@isi.edu, **Tuesday March 23, 11:59pm**

In this assignment you will build a simple CKY-style parser given a probabilistic context-free grammar (PCFG) trained from the Penn Treebank (training is already done for you). You will need `test.txt`, `toy.pcfg` and `grammar.pcfg` from the course website.

1 CKY Parser (70 pts)

We represent a Probabilistic Context-Free Grammar (PCFG) in the following format (see `toy.pcfg`):

```
S
S -> NP VP # 1
NP -> DT NN # 0.6
NP -> DT NNS # 0.4
DT -> the # 0.7
DT -> a # 0.3
NN -> boy # 0.5
NN -> girl # 0.5
NNS -> boys # 1
VP -> VB NP # 1
VB -> saw # 1
```

where the first line (`S`) is the start nonterminal of the grammar. We call the above grammar the *toy grammar*.

Now write a CKY parser that takes a PCFG and a test file as input, and outputs, for each sentence in the test file, the best parse tree in the grammar, along with its **negative log probability**. For example, if the input sentence is

```
the boy saw a girl
```

the output line (with the toy grammar) should be:

```
(S (NP (DT the) (NN boy)) (VP (VB saw) (NP (DT a) (NN girl)))) # 3.969
```

Note the tree format is `(X ...)` where `X` is a nonterminal and "..." are the list of subtrees, with a whitespace separating consecutive subtrees. The negative log probability, i.e., the cost should be rounded to the nearest thousandth.

Q1.a (45 pts) Your CKY code should have the following input-output protocol:

```
cat test.txt | ./<your_cky_code> grammar.pcfg > test.parsed 2> test.log
```

where `test.log` is a log file in which each line (except the last one) contains the following fields separated by tabs:

i *len* *time* *nodes* *hedges*

where i is the sentence index (1-based), *len* is the sentence length (number of tokens including punctuations), *time* is the wall-clock time spent on this sentence (in seconds, rounded to the nearest **thousandth**), and *nodes* and *hedges* are the number of nodes and hyperedges explored, respectively. The last line should be (again, separated by tabs)

`total=`n `avglen=`*avglen* `avgtime=`*avgtime* `avgnodes=`*avgnodes* `avghedges=`*avghedges*

Average numbers should all round to the nearest tenth, **except for time, which rounds to thousandth**. Include your CKY code and the resulting `test.parsed` and `test.log` files in your submission. It is very important to observe the IO protocol, as your code will be tested against many other testcases, and we will `diff` your output with the reference output. If you use C/C++ (not recommended), be sure to include a Linux-executable file. Note: as mentioned in class, there are many (slightly) different ways of implementing CKY; 15 pts will be based on the *cleverness* (i.e., efficiency) of your implementation. Code clarity also matters (5 pts).

Q1.b (5 pts) How many sentences failed to parse? Your CKY code should output (to stdout)

```
NONE # 0
```

for these cases (so that the number of lines in `test.parsed` should be equal to that of `test.txt`. Note that the cost is 0 for those sentences, while parsing time should still be measured in the log file, so that overall averages include those sentences). Briefly explain why there are parsing failures.

Q1.c (20 pts) Based on `test.log`, make a *scatter plot* of parsing time for `test.txt` where the x axis is sentence length, and the y axis is time in seconds, with each point representing one sentence. What can you tell from this plot? Then use log-log scale, in which $y = x^k$ appears as a line of slope k. What is the value of k? Why is it so? Now do the same (scatter plot and slope) for number of nodes and number of hyperedges. Submit the three (3) log-log scatter plots. Parsing time (or speed) correlates with which one better, number of nodes or hyperedges? Why?

Note (important): nodes are like `S,0,5` and hyperedges are like `S,0,5 -> NP,0,2 VP,2,5`. `DT,0,1` is a node but the word "the" is not. There shold be 9 nodes and 9 hyperedges for the example sentence with `toy.pcfg` (see Extra Credit A), but in general there should be many more hyperedges than nodes (due to ambiguity). Also, only count nodes and hyperedges with non-zero probabilities.

2 Most Probable Tree out of a PCFG (30 pts)

As discussed in class, the problem of "highest probabilitiy tree out of a PCFG" can be solved by the Knuth 1977 Algorithm (see slides), which is a best-first dynamic programming algorithm extending Dijkstra's famous shortest-path algorithm from graphs to hypergraphs. The key observation is that the "non-negative edge costs" requirement in Dijkstra is naturally satisfied by the fact that probabilities are always less than or equal to 1; in other words, as you go along, your accumulated cost always gets *worse and worse*, which is required for the correctness of best-first algorithms.

Note that greedy algorithm does not work, because using a suboptimal rule locally might lead a better tree globally. For example, the **best tree out of the toy grammar above should be**

```
(S (NP (DT the) (NNS boys)) (VP (VB saw) (DT the) (NNS boys))) # 2.546
```

which uses the `NP -> DT NNS` rule instead of the higher-probability rule `NP -> DT NN`.

Your Knuth code should have the following input-output protocol:

```
cat grammar.pcfg | ./<your_knuth_code>  > best_tree
```

Submit your Knuth code and the line (tree and cost) in file `best_tree`. Your code will be tested against other testcases so make sure it observes the IO protocol.

Extra Credit A (15 pts)

Now that you have your Knuth 1977 implemented, it is fairly easy to put together a best-first CKY parser without really writing one. You can simply create an *intersected PCFG* (though no longer normalized) and feed it into your Knuth code. The IO protocol should follow Problem 1, and make sure they output the same trees and costs. Submit your code, your new `test.bcky_log`, and three (3) overlaid scatter plots for time, nodes, and hyperedges, resp., showing both CKY (from Problem 1) and best-first CKY on each plot. You can use either normal or log-log scale, whichever is better for the contrasts. Which one is faster in time, and which one explores less hyperedges or less nodes? Why? What did you learn from this experiment, and what are possible ways to improve best-first CKY?

Note: for example, the intersected PCFG for the example sentence and `toy.pcfg` should be:

```
S,0,5
S,0,5 -> NP,0,2 VP,2,5 # 1
NP,0,2 -> DT,0,1 NN,1,2 # 0.6
NP,3,5 -> DT,3,4 NN,4,5 # 0.6
DT,0,1 -> the # 0.7
DT,3,4 -> a # 0.3
NN,1,2 -> boy # 0.5
NN,4,5 -> girl # 0.5
VP,2,5 -> VB,2,3 NP,3,5 # 1
VB,2,3 -> saw # 1
```

Extra Credit B (35 pts)

Implement an Earley parser (see slides), which does left-to-right parsing with top-down filtering via prediction step. The IO protocol should follow Problem 1, and make sure they output the same trees and costs. Submit your code, your new `test.earley_log`, and three (3) overlaid scatter plots for time, nodes, and hyperedges, resp., showing both CKY (from Problem 1) and Earley on each plot. You can use either normal or log-log scale, whichever is better for the contrasts. Which one is faster in time, and which one explores less hyperedges or less nodes? Why? (If you also did Extra Credit A, which one is the fastest, and why?)

Debriefing

Please answer these questions in **debrief.txt** and submit it along with your work. **Note:** You get 5 points off for not responding to this part.

1. How many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Are the lectures too fast, too slow, or just in the right pace?
4. Any other comments (to the course or to the instructors)?

Additional Resources

1. Python tutorial: `http://docs.python.org/tutorial/`.
   Python course: `http://www.cis.upenn.edu/~lhuang3/cse399-python/`.
2. gnuplot tutorial (best tool for plotting): `http://t16web.lanl.gov/Kawano/gnuplot/index-e.html`
3. original Knuth 77 paper: `http://www.cis.upenn.edu/~lhuang3/knuth77.pdf`.