CSE 498 Natural Language Processing
Project 3, Fall 2017
Sachin Joshi

Exercises

• How can you modify the production frequencies so that longer sentences can be generated? Explain how and why.

Increasing the production frequencies, results in a higher probability thereby ensuring that the production rule can be used again and again. Let's say we increase the production frequency of the rule NP -> NP PP. This results in a comparatively higher probability for that production rule, which ensures that this production rule now has a higher chance (probability) of being selected. If this production rule occurs more number of times, then it must be expanded recursively until a terminal symbol is found. This will in turn result in higher number of terminal symbols and will eventually result in longer sentences.

• Is every sentence generated by the grammar can be parsed by the CYK algorithm using the same grammar? Give your intuition.

If a sentence is generated by the grammar, then there exist some production rules that can generate this sentence. Since CKY algorithm enumerates all possible derivations of the sentence given the production rules in the grammar, and returns the highest probable parse tree using dynamic programming, we can say that it's possible to parse every sentence generated by the grammar.

Generator.java

This class is used to generate a sentence and output a parse tree to the file sentence.txt given the grammar grammar.gr. The idea is to start from the ROOT production rule and add the non-terminal S to the linkedlist. S is further expanded as S->NP VP and the non-terminals VP and NP are added at the starting of the linkedlist. NP positioned at index 0 is further expanded and the resulting non-terminals are stored at the current index. This step is recursive until a terminal symbol is found. We then move on to the next index to expand the non-terminal at that particular index until it results in a terminal symbol. The process is repeated until there are no non-terminals or pre-terminals left in the linkedlist to be expanded. The linkedlist here behaves like a stack.

A separate linkedlist is used to store the parse tree generated. Whenever a non-terminal or a pre-terminal is expanded, the LHS and the RHS of the particular production rule is added to the last index of the LHS in the linkedlist. We start building a parse tree by adding the rule ROOT->S. to the linkedlist in the desired format. Then the last index of non-terminal S is recorded, the non-

terminal S is expanded and the RHS corresponding to the rule S->NP VP is stored at that particular index. The same is done for the rest of the non-terminals, resulting in the final parse tree.

Parser.java

When implementing the CKY algorithm, we don't use triple arrays. Instead we define a data structure Chart.java, which holds all of a table cell's information and then create a double array of those cells. The CKY algorithm assumes a data structure called a chart which is two-dimensional array of size (n+1) * (n+1) where n is length of the sentence. Each cell [i, j] may contain multiple chart items of the form A[i, j] where A is a non-terminal symbol labeling the span between two indices 0 <= i <= j <= n in the sentence. We store a tree in a chart by storing its labeled spans in the relevant chart cells and then recording the back pointers to the labeled symbols that created this span.

CYK builds a table containing a cell for each substring. The cell for a substring x contains a list of variables V from which we can derive x (in one or more steps). The bottom row contains the variables that can derive each substring of length 1. Now we fill the table row-by-row, moving upwards. To fill in the cell for a 2-word substring x, we look at the labels in the cells for its two constituent words and see what rules could derive this pair of labels. For each longer substring x, we have to consider all the ways to divide x into two shorter substrings. The algorithm also keeps a track of the back pointers to the combined cells of the most probable chart item. Once the chart is completely filled, we find the start symbol and traverse the back-pointers indicating the most probable chart items that construct the tree rooted in the symbol.

```
ganymede:~% cd /home/saj415/saj415_p3
ganymede:~/saj415_p3% ./build_java.sh
ganymede:~/saj415_p3% ./run_java.sh
the
fine
delicious
staff
on
a
president
wanted
the
staff
under
the
perplexed
sandwich
!
(ROOT (S (NP (NP (Det the)
                 (Noun (Adj fine)
                       (Noun (Adj delicious)
                             (Noun staff))))
             (PP (Prep on)
                 (NP (Det a)
                     (Noun president))))
         (VP (Verb wanted)
             (NP (NP (Det the)
                     (Noun staff))
                 (PP (Prep under)
                     (NP (Det the)
                         (Noun (Adj perplexed)
                               (Noun sandwich)))))))
      !)
(ROOT(S(NP(NP(Det the)(Noun(Adj fine)(Noun(Adj delicious)(Noun staff))))(PP(Prep
 on)(NP(Det a)(Noun president))))(VP(Verb wanted)(NP(NP(Det the)(Noun staff))(PP
(Prep under)(NP(Det the)(Noun(Adj perplexed)(Noun sandwich)))))))!)
ganymede:~/saj415_p3%
```