600.465 – Natural Language Processing
Assignment 4: Parsing

Li-Yi Lin, Jinyi Guo

March 2016

1. (a) Some interesting point about Berkeley Parser/Stanford Parser:
   i. There could be multiple (more than 3) subrules correspond to a particular rule. E.g. `VP -> VBD NP PP`.
   ii. Nonterminials are more carefully categorized both vertically and horizontally. E.g. for adjectives, we have `JJ` for normal adjectives, `JJR` for comparative and `JJS` for superlative. As well as `ADJP` for adjective phrase, which includes the cases where adjectives are modified by adverbs or followed by NPs and PPs.
   iii. Word "to" is flexibly combined into many rules like `PP -> TO NP` and `VP -> TO VP` for it has various POS.

   (b) Things got wrong by the parser/hard things for parser to handle:
   i. When parsing a sentence ending with a preposition, things are likely to go wrong. For Berkeley Parser, when parsing the sentence `The very biggest companies are not likely to go under .` it gives a result with an `ADVP` sticks after the period, indicating that there is no rules in the parser that can parse the sentence correctly. In addition, for both parser, the last part of the sentence ("to go under") is parsed as a sentence ("S"), which is also doubtful.
   ii. Numbers with signs like "%", some parser failed to parse them correctly. The quantity phrase "4 % to 5%" was parsed as an `ADJP` by Berkeley Parser.
   iii. For sentences that contain words with rare part of speech or systematic ambiguity in noun sequences, these parser failed to render the "desired" result. For example, in sentence"Fruit flies like a banana .", the word "flies" is parsed as a `VBZ` while it is supposed to be a noun here.

   Some hard things that it manages to get right
   i. The Berkeley parser is able to parse some sentences that have inversion, which Stanford parser often has trouble to deal with.
   ii. Both parse can correctly parse relative clause and attributive clause, which can be represented by notation `SBAR` in the parse tree.

   (c) i. When I entered an inversion sentence with present tense, e.g. "`By exercising more can we get healthier.`" the Sanford parser failed to parse the modal verb, `can`, and classified it as a `NN`. But the Berkeley parser correctly parsed the modal verb, `can`. See the results shown below:

   Stanford parser:

```
(ROOT
  (S
    (PP (IN By)
      (S
        (VP (VBG exercising)
          (NP (JJR more) (NN can)))))
    (NP (PRP we))
    (VP (VB get)
```

```
      (ADJP (RBR healthier)))
    (. .)))
```

But if we changed `can` to `could`, the Stanford parse can correctly parsed `could` as a modal verb.

```
(ROOT
  (S
    (PP (IN By)
      (S
        (VP (VBG exercising)
          (NP
            (NP (JJR more))
            (SBAR
              (S
                (VP (MD could))))))))
    (NP (PRP we))
    (VP (VB get)
      (ADJP (RBR healthier)))
    (. .)))
```

Berkeley parser:

```
(ROOT
  (SINV
    (PP (IN By)
      (S
        (VP (VBG exercising)
          (NP (JJR more)))))
    (VP (MD can))
    (NP (PRP we))
    (VP (VB get)
      (ADVP (RBR healthier)))
    (. .)))
```

We further used sentences from "New York Times" to test the parsers but all the sentences we used are parsed correctly by the parsers.

Additionally, we tested the Turbo Parser with dependency grammar. The highlight of this parser is that it can mark the syntactic dependency between words in the sentence. But sometimes it may produce erroneous result or confusing dependency relations. For example, the word "flies" in sentence "Time flies like an arrow" is parsed as a `NNS` with a dependency `NN` pointing to "time". This might be a rare rule works for sentences like "Fruit flies like an arrow", but definitely it does not work fine for most sentence.

2. (a) To restrict the space complexity to $O(n^2)$, we used several big hash sets or hash tables to store the necessary information for parsing. One of the biggest is the hash set to store the hash of each entry in the parse chart. For each column, whenever we add a new entry to the chart (either by `predict` or `attach`) we have to store the hash of the new entry to the hash set, so that in the future we can check if same dotted rule already exists in the chart when doing `predict` or `attach` operation.

   Therefore, for each column in the chart, there are at most $O(G)$ possible dotted rules for each entry, and $O(n)$ possible starting positions. So the maximum size of the hash set is $O(Gn^2)$. Other data structures are smaller than this because they only store part of the information discussed before.

(b) In order to quickly add an entry to each column, we use two `ArrayList` to store heads and tails of each column. Each entry is a linked-list data structure. So, it takes $O(1)$ time to add an entry to the tail of the linked-list of the target column. By doing so, we only use two `ArrayList` instead of multiple `ArrayList` for each column.

(c) We have a `dottedRule` class to store information of the entry like position of the dot and current best weight. The value of current best weight is updated when we move the dot to the right or a new dotted rule with the same structure but less weight is found (attached).

To keep track of the best parse itself, we assigned two backpoints to each entry (dotted rule). Each time we do attach operation we have the first backpoint of the attached rule pointing to the dotted rule that is used to attach other unfinished dotted rule, and the second backpoint pointing to the dotted rule that is attached. Finally when we finish parsing the entire sentence we can recursively track and print out the entire parse tree from the finished dotted rule with left hand side "ROOT" in the last column.

NOTE: we implemented a third version of the parser that handles the bug mentioned in the handout. Though we made it to produce correct weight and parse, the running time is relatively longer (takes about 11 minutes). The time complexity of our implementation is about $O(n^4)$, but the actual running time seems not that bad.

3. For speed up purpose, we have implemented the first method that keeps track of which categories have already been predicted for the current column. For this function, We use one `hashset` to record categories that have been predicted. The execution time for this method is `4 minutes and 22 seconds`. Since we implemented this method before reading problem 3, so all the parsers we have implemented have adopted this method.

   In addition to the first method, we also the second method that deletes all rules that contains terminals that are not in our sentences. The execution time after adding this method is `3 minutes and 27 second`. This helps speed up the program by 20.9%.