

Homework #7: CMPT-825

Anoop Sarkar – anoop@cs.sfu.ca

Context-Free Grammars:

- (1) Consider the following CFG G with start symbol S that can generate the question: *what does Calvin like* and also the statement: *Calvin likes ice-cream* (among the other sentences it can generate)

$$\begin{aligned} S &\rightarrow WHNP \text{ does } S \\ S &\rightarrow NP VP \\ VP &\rightarrow V \\ VP &\rightarrow V NP \\ WHNP &\rightarrow \textit{what} \\ NP &\rightarrow \textit{Calvin} \mid \textit{ice-cream} \\ V &\rightarrow \textit{like} \mid \textit{likes} \end{aligned}$$

- a. Replace the rule $VP \rightarrow V$ in the CFG G with the two new rules shown below.

$$\begin{aligned} VP/NP &\rightarrow V NP/NP \\ NP/NP &\rightarrow \epsilon \end{aligned}$$

These rules introduce two new non-terminals VP/NP and NP/NP . Introduce new rules to the CFG G so that it can continue to generate the two sentences: *what does Calvin like* and *Calvin likes ice-cream*. Provide the new CFG.

- b. Provide a CFG based on the CFG you provided in Question 1a that can *accept* sentences listed below that are *not* marked with * and *does not* accept sentences listed below that *are* marked with *:

Calvin likes ice-cream and Hobbes

Calvin likes ice-cream and hates beans

what does Calvin like and Hobbes hate

* what does Calvin like and Hobbes hate beans

* what does Calvin like ice-cream and Hobbes hate

Incidentally, this constraint on sentences with conjunctions (e.g. *and*, *or*, and *but*) was discovered by John Ross in his 1967 MIT Linguistics PhD thesis. He called it the Across-the-Board constraint. The particular solution you are providing was first proposed by Gerald Gazdar in his 1981 paper in the journal *Linguistic Inquiry*.

- (2) Consider the sentence *Calvin saw the man with the telescope* which has two meanings, one in which Calvin sees a man carrying a telescope and another in which Calvin using a telescope sees the man. These two meanings can be represented by the derivations possible in the following context-free grammar (CFG) rules. This ambiguity between the *noun-attachment* versus the *verb-attachment* is called *PP-attachment ambiguity*.

$$\begin{aligned}
 S &\rightarrow SBJ VP \\
 VP &\rightarrow V NP \\
 VP &\rightarrow VP PP \\
 NP &\rightarrow NP PP \\
 NP &\rightarrow Det N \\
 PP &\rightarrow P NP \\
 SBJ &\rightarrow Calvin \\
 V &\rightarrow saw \\
 Det &\rightarrow the \\
 N &\rightarrow man \mid telescope \mid hill \\
 P &\rightarrow with \mid on
 \end{aligned}$$

Let S be the start symbol.

- Using the above CFG draw the derivations possible for the input: *Calvin saw the man with the telescope on the hill*.
- The above CFG can be represented as Perl data in the following manner:

```

our (@start, %cfgrule, %cfgpos);

@start = ('S');
$cfgrule{'S'} = [ ['SBJ', 'VP', '#' ] ];
$cfgrule{'VP'} = [ ['V', 'NP', '#'], ['VP', 'PP', '#'] ];
$cfgrule{'NP'} = [ ['NP', 'PP', '#'], ['Det', 'N', '#'] ];
$cfgrule{'PP'} = [ ['P', 'NP', '#'] ];
$cfgpos{'SBJ'} = [ ['Calvin', '#'] ];
$cfgpos{'V'} = [ ['saw', '#'] ];
$cfgpos{'Det'} = [ ['the', '#'] ];
$cfgpos{'N'} = [ ['man', '#'], ['telescope', '#'], ['hill', '#'] ];
$cfgpos{'P'} = [ ['with', '#'], ['on', '#'] ];

```

The `%cfgrule` hash table contains the non-lexical rules (i.e. rules that do not contain any terminal symbols) while the `%cfgpos` hash table contains rules of the form $A \rightarrow a$ where A is a non-terminal and a is a terminal symbol signifying the part of speech tag rules in the grammar.

Using the above Perl representation of a CFG, implement the Earley parsing algorithm `earleyParse` as a Perl program.

Hint: You can represent the *state* data structure described in the algorithm as follows: A state:

$(A \rightarrow B \bullet C, [5, 7])$ can be represented in Perl as

```
$state = [ 'A', [ 'B', 'C', '#' ], 1, 5, 7 ];
```

where `$state->[0]` is the left hand side of the CFG rule, `$state->[1]` is a reference to a list of right hand side symbols for the CFG rule, `$state->[2]` is the location of the dot in the right hand side, in this example, since the dot position is 1 the dot is immediately to the left of 'C' in the right

hand side of the rule, and finally `$state->[3]` and `$state->[4]` are the i, j values for the span of the dotted rule in the input string: 5, 7 in this case. If you represent the state in this manner the following functions might be useful to you:

```
sub incomplete {
    my ($state) = @_;
    return (($state->[1]->[$state->[2]] eq '#') ? 0 : 1);
}
```

```
sub nextCat {
    my ($state) = @_;
    return ($state->[1]->[$state->[2]]);
}
```

The implementation `earlyParse` returns the chart after parsing of the input has finished. We can use the chart to decide whether the input string was accepted by the CFG or not. The following code is an example of how we can check the chart to see if a start symbol successfully spans the entire input string. Note that this code assumes that states are represented as described above. It also assumes that the chart is implemented as a reflist of reflists containing the states.

```
my @input = qw(Calvin saw the man with the telescope);
my $chart = earleyParse(@input);
my $length = $#input+1;
my $yes = 0;
for my $start (@start) {
    for my $finalState (@{$chart->[$length]}) {
        $yes = 1 if (parseSuccess($finalState, $start, 0, $length));
    }
}
print (($yes) ? "yes" : "no", "\n");
```

```
sub parseSuccess {
    my ($finalState, $start, $begin, $end) = @_;
    return (($finalState->[0] eq $start) and
            (! incomplete($finalState)) and
            ($finalState->[3] == $begin) and
            ($finalState->[4] == $end));
}
```

A sample final state for the input *Calvin saw the man with the telescope* which would succeed in this test (given the CFG above) is the state ($S \rightarrow SBJ VP\bullet, [0, 7]$) or in the Perl equivalent:

```
['S', ['SBJ', 'VP', '#'], 2, 0, 7]
```

Provide the Perl code for your implementation of the Earley parsing algorithm. Also provide a trace of your recognition algorithm for 3 sentences accepted by your program, and 3 sentences that are not accepted.

The Earley algorithm was proposed by J. Earley in his 1968 CMU CS PhD thesis as a more elegant form of the Cocke-Younger-Kasami parsing algorithm which was the first polynomial time parsing algorithm for CFGs. The Cocke-Younger-Kasami algorithm was independently discovered by the three people mentioned in its name and is often referred to as the CKY or the CYK algorithm. Both the Earley algorithm and the CKY algorithm take worst case time: $O(n^3)$ where n is the length of the input string.

- (3) This question will introduce you to the basics of statistical parsing. The task will be to modify an existing parser written in Perl in a sequence of steps listed below.

A recurring theme in statistical parsing is the need to limit the search space of the parser. Many productions that a hand-constructed CFG would rule out as impossible, a learned PCFG might permit with low but non-zero probability. Techniques to limit the search space might still guarantee that the optimal parse is found (such as A* search). Or they might do more aggressive pruning for even more speed, but sometimes fail to find the most probable parse. The Beam Search Thresholding you will implement is an example of this second, more aggressive pruning.

You will be taking an existing probabilistic context-free CKY parser and adding beam search thresholding to improve its running time. The CKY algorithm without pruning is exhaustive, guaranteed to find every possible parse of a sentence. Beam search thresholding is a particular form of pruning which ignores items stored in the parser chart that have a very low probability, relative to other items in the same cell. Pruning these items is like making a bet that these items are so unlikely, they will not be in the most probable parse, even if they combine with other high-probability items. The ratio in probabilities from the most probable item in a cell to the lowest probability permitted for a "competing" item is the Beam Width, and this parameter can be tuned to trade off speed (a narrower beam can prune more items) vs. accuracy (pruning more items risks pruning the most probable parse).

Notes on the grammar: The grammar was extracted from the Penn Treebank of WSJ articles, sections 01-09, with no smoothing of the count statistics. The test sentences were drawn from section 0. To reduce the size of the grammar and parses, the words (lexical symbols) were removed, so that the parse tree "leaves" are part-of-speech (POS) tags.

The grammar was converted into a form that is (almost) in Chomsky Normal Form. The implementation has some additional code to handle unary rules (rules of the form: $A \rightarrow B$), since the grammar used to annotate the treebank include many nodes with a single non-terminal as a (Right-Hand Side) RHS.

The hand-annotated "gold" reference parses, and the output of the provided parser, use the standard nested parentheses to represent the parse structure. The evaluation of the quality of parses uses the evalb tool written by Satoshi Sekine and Michael Collins, which reports the precision and recall of the brackets (the parse tree nodes and their spans) relative to the reference parse.

Here's an example of what the evaluation is:

```
candidate: (S (A (P this) (Q is)) (A (R a) (T test)))
gold:      (S (A (P this)) (B (Q is) (A (R a) (T test))))
```

In order to evaluate this, we list all the labeled brackets with their spans, skipping the brackets of the form (A a), i.e. those brackets that have a span of 1 (note that evaluation of spans of length 1 would be equivalent to evaluating the part of speech tagging accuracy):

Candidate	Gold
(0,4,S)	(0,4,S)
(0,2,A)	(0,1,A)
(2,4,A)	(1,4,B)
	(2,4,A)

Precision is defined as $\frac{\#correct}{\#proposed} = \frac{2}{3}$ and recall as $\frac{\#correct}{\#in\ gold} = \frac{2}{4}$. You can use evalb program to check this for yourself:

```
/cs/natlang-a/packages/EVALB/evalb
```

The files for this question are in the directory /cs/natlang-a/packages/parselab/:

- Grammar: out.gram

- Input sentences: `input.small`
- "Gold" reference parses for the input sentences: `gold.raw`
- Pretty printing of parse trees: `indentrees.pl`
- Plotting parsing time comparison: `plot_times.pl`
- Testing parser output against references: `test_output`

Optionally copy the above files to your directory. Copy the file `pcfg-cky-parser-incomplete.pl` to your directory as the file `pcfg-cky-parser.pl`. You will modify this Perl file to complete the homework. Run `perl pcfg-cky-parser.pl -h` to check the options that are built in to this program.

- a. Run the existing parser on the input file:

```
perl pcfg-cky-parser.pl out.gram < input.small > parser.out
```

This runs the CKY parser without any pruning and hence is extremely slow. It will take about 15-20 mins depending on your machine speed and memory. It creates the output file `parser.out`. You can see the parse trees produced in `parser.out` and the gold reference parses in `gold.raw` using the program `indentrees.pl`:

```
cat parser.out | perl indentrees.pl | more
cat gold.raw | perl indentrees.pl | more
```

Check the accuracy of the parser output:

```
sh test_output parser.out gold.raw > evalb.nopruning
```

Provide this file `evalb.nopruning`. The precision/recall figures in `evalb.nopruning` are the best the CKY parser can do with this training data. Next we will try to maintain this accuracy while at the same time speeding up the parse times.

- b. Modify the function `prune_items` in the file `pcfg-cky-parser.pl`. Remove all items in each cell i, j whose probability is less than that of the most probable item in the same cell multiplied by β . The value of β is stored in the variable `$beam` which can be assigned the value `log(value)` using the option `-b`, e.g. `perl pcfg-cky-parser.pl -b value`.

For example, let us assume that the value of the beam, $\beta = 0.0001$. Let us assume that the itemlist for the span $2, 4$ is ("NP", "VP", "PP") which is stored as a refflist in the variable `$A[2][3]`. The probabilities for each item for the span is stored in the variable `$P`, e.g.

$P(\text{"NP"}, 2, 4) = \$P[2][4][\text{"NP"}]$.

In this example, let us take $P(\text{"NP"}, 2, 4) = 0.12$, $P(\text{"VP"}, 2, 4) = 0.03$, $P(\text{"PP"}, 2, 4) = 0.000011$.

Beam thresholding would imply that we prune away $P(\text{"PP"}, 2, 4)$ since it is less than the maximum probability for the same span times the beam value: $P(\text{"NP"}, 2, 4) \times \beta$.

$P(\text{"VP"}, 2, 4)$ is not pruned since it sneaks by under the beam threshold.

Important: Remember that unlike the above example the values in `$P` and the variable `$beam` are stored as log probabilities.

Measure accuracy (using `test_output` as above) on the file `input.small` with beam thresholding using several different beam widths. Include the beam values 0.001 and 0.01. It's important to use the `-p` option which enables the call to the `prune_items` function.

```
perl pcfg-cky-parser.pl -p -b 0.001 out.gram <input.small \
  >parser.out.pruning_0.001
perl pcfg-cky-parser.pl -p -b 0.01 out.gram <input.small \
  >parser.out.pruning_0.01
```

- c. A problem with pruning using beam thresholding as we have done in the previous step is that some non-terminals can get the highest probability for a span, but when combined with other non-terminals

in the right-hand side of a CFG rule, it might lead to a very improbable derivation. For example, $P(\text{"FRAG"}, 2, 4)$ might be very high but the rule $VP \rightarrow \text{FRAG PP}$ might be very improbable leading to search errors.

A solution to this problem is to use a rough estimate as to how likely a subtree will lead to full parse tree. This rough estimate can be as simple as the probability of a non-terminal independent of any context. This is the *prior* probability of a non-terminal. In the example above $P_{\text{prior}}(\text{"FRAG"})$ would be multiplied with the probability $P(\text{"FRAG"}, 2, 4)$ before we do beam thresholding as in the previous step.

Multiply the prior probability for each non-terminal in the pruning process. The function `get_prior($nt)` returns the prior probability for a non-terminal string `$nt`.

Provide the accuracy of the parsing with prior probabilities and a beam value of 0.001:

```
perl pcfg-cky-parser.pl -p -r -b 0.001 out.gram <input.small \  
> parser.out.prior.pruning_0.001
```

- d. Plot the change in parsing time, comparing the parser without pruning and the parser with pruning at the various beam widths. Use the following script to generate the output graph:

```
perl plot_times.pl parser.out parser.out.pruning_0.001 parser.out.pruning_0.01 ...
```

Check the name of the output file (for some arbitrary number `xx`): `timing-xx.eps` and view it using `gv`:

```
gv timing-xx.eps
```

Provide this graph which should show that your pruning strategy succeeds in speeding up parse times as you tighten the beam threshold value.

Your output should look like the following graph:

