Homework Assignment 4

CS 585 Natural Language Processing
Spring Semester, 2019

Dinesh Karamchandani

CWID: A20407484

Problem Description

The aim in this assignment was to experiment and enhance Earley's Algorithm Parser using context free grammars (CFGs) that are present in the 'grammars' folder of the zip provided with the homework instructions. The output to be provided including building a function to showcase the parse structure of the sentence input to the algorithm. As well as building a probabilistic version of the algorithm where we take product(or sum) of the probability (log-probability) associated with each invocation of the rule from the given grammar.

Data

The data for this algorithm would be sentences that we want to parse with the limitation that all the words used in that sentence are present in the grammar.

Evaluation

In this section we will evaluate the algorithm as is given to us and answer the questions detailed in the homework.

1. Evaluate the parser in its current form:
    - What sorts of sentences can it parse?

    Ans: Upon inspecting the simple.gr grammar file and trying different sentences for the algorithm to parse, I found that the parser using the current grammar file can parse sentences of certain type.

    From the example given in the sentences.txt file which is 'I was driving to Chicago' if we modify the above sentence to use other auxiliaries like do, will, does, is then all the sentences parse, for e.g.,

    - I did drive to chicago
    - I will drive to chicago
    - I do drive to chicago

    Also sentences like 'He drives', 'Aliens steal watermelon' and so on are also getting parsed by this grammar.

    Based on the findings from above, I can conclude that statements which generally have the above semantic structure will be parsed by the algorithm. Also sentences having the above sematic structure which have varying terminating literals which would invoke the same non-terminating literals would also get parse as they would satisfy the requisites of the algorithm.

- What types of sentences will it fail to parse?

Ans: Some of the logical and grammatically correct sentences that this algorithm with the 'simple.gr' grammar file cannot parse are:

1. He steals to give watermelon to John
2. In buckaroo john drives
3. He was driven to live on alien planet
4. Earth is alien planet
   And so on…

This is due to below reasons:

a. structure of some of the sentences deviates from the general structure mentioned in the first part of this question,
b. Also, as there aren't any rules in the grammar to handle the deviations of these sentences, this leads to infinite looping often to the recursion
c. Also, there are not enough rules to handle sentences which have determinants or articles (fd): a, these, this, the which leads to sentences not being reduced to match the rules and get us the parse sentence ('s' in the grammar).

- What improvements to the grammar or parser are required to properly parse common failed sentences?

Ans: Some of the improvements that can be done to handle some of the commonly failed sentences are as below.

- Adding rules to improve handling of sentences with non-general semantic structure as seen in the 2nd part of this question.
- Adding additional rules for terminating literals to as to handle more logically and grammatically correct sentences like addition of auxiliaries like am, be, are or pronouns like myself, herself, etc.
- Adding rules to handle punctuation marks in both the parser as well the grammar file would also improve the quality of the parsing

Grammar Improvements

In this section we will discuss about the improvements that could be done to the grammar to handle some of the failed sentences.

2. Look at the grammar. What changes to the grammar alone could improve the parsing (in particular, consider coordinated conjunctions)? Modify the grammar somewhat and evaluate the improvements. What new sentences may be parsed? What invalid sentences are now parsed (if any)?

Ans: In order to find out the changes to be done to the grammar let us first see an instance of failed sentence.

Consider the sentence “earth is a planet”, when we run this in the algorithm we get the below chart details.

Chart:

0 Inactive: (0, 1, 1, 'fname', ('earth',), ())

1 Inactive: (0, 1, 1, 'np', ('fname',), (0,))

2 Active: (0, 1, 1, 'np', ('fname', 'fname'), (0,))

3 Active: (0, 1, 1, 's', ('np', 'vp'), (1,))

4 Active: (0, 1, 1, 's', ('np', 'fis', 'np'), (1,))

5 Inactive: (1, 2, 1, 'fis', ('is',), ())

6 Inactive: (1, 2, 1, 'faux', ('is',), ())

7 Active: (1, 2, 1, 's', ('faux', 'np', 'vp'), (6,))

8 Active: (1, 2, 1, 'vbar', ('faux', 'fv'), (6,))

9 Active: (0, 2, 2, 's', ('np', 'fis', 'np'), (1, 5))

10 Inactive: (2, 3, 1, 'fd', ('a',), ())

11 Active: (2, 3, 1, 'np', ('fd', 'nbar'), (10,))

12 Inactive: (3, 4, 1, 'fname', ('planet',), ())

13 Inactive: (3, 4, 1, 'np', ('fname',), (12,))

As seen here, due to insufficient rules for it to reduce to, we don’t have a parse of this sentence.

If we backtrack, we can see that the rules which were invoked are the below:

fname -> earth

faux-> is

fd -> a

fname->planet

fis-> faux

np -> fname

Since we couldn’t reduce ‘fd fname’ (from ‘a planet’) to some rule, we couldn’t satisfy any of the rules to get S, hence it failed.

In order to do that, we can add the below rule to handle the above issue.

np -> fd fname

For our file it would translate to :

0 np fd fname

On adding this we now get the sentence parsed successfully with the below parse structure.

[s [np [fname earth]] [fis is] [np [fd a] [fname planet]]]

Now using the above, we parse sentences like “buckaroo is a watermelon” (although illogical but grammatically correct”)

Similarly, sentences like “this is chicago”, are also not parsing using the current rules in the simple.gr file. In order to do, we can add below rules to get them parsed.

np-> fd

vbar -> faux fname

For our file these would translate to :

0 np fd

0 vbar faux fname

Also if we have sentences using coordinated conjunctions like the below, then too the parsing fails.

“we steal for watermelons and jetcar”

In order for the grammar to handle sentences with coordinated conjunctions we first need to add the conjunctions as well as rules to help reduce the conjunctions.

cjun -> for

cjun -> and

cjun -> or

cjun -> but

cjun -> so

cjun -> yet

cjun -> also

np-> fn cjun

vbar -> cjun np

vp -> fv vbar

Using the above changes, we can now parse sentences using coordinated conjunctions.

For our file these rules would be translated as below :

0 cjun for

0 cjun and

0 cjun or

0 cjun but

0 cjun so

0 cjun yet

0 cjun also

0 np fn cjun

0 vbar cjun np

0 vp fv vbar

Code Improvements

In this section we will discuss on the code improvements to be done to get the parse tree to be displayed using bracket notation as shown in the example below. We will also showcase part 4 of the assignment where in we implement a probabilistic version of the algorithm which showcases the probability of the parse based on the grammar having set probabilities.

Part 3

3. Modify the code to output the parse tree(s) computed, rather than just the dynamic programming table and whether the sentence parses. You will need to add appropriate back-links into the DP table and write a routine to extract trees from the table. Show the tree(s) by printing the input sentence in a bracketed form, for example:
   S[NP[Det[the] N[man]] VP[V[saw] NP[PN[her]]]]

Ans: To get the above output to be displayed by the code, we need to make the below alterations to the functions printParses() and struct2Str() as well modify some code in getParse() function in the **ChartyPy3.py** file.

The altered version of the above functions are showcased as below:

1. **printParses function – This function was to handle the printing all the parse tree structures obtained from the struct2Str function.**
   def printParses(parses):

```
"""TODO: Prints the parse as bracketed string to the screen."""


 for j,i in enumerate(parses):

   print("Parsing:", j+1, "of", len(parses))

   #added functionality of PROB to print probability of the parse when using probabilistic version
```

```
    if PROB:
        print(i[0])
        #print("With actual probability:", exp(i[1]))
        print("With log probability:", i[1])
    else:
      print(i)
```

2. struct2Str function – This function contained the actual recursive implementation of getting the various symbols from the chart to be printed in the form tree using the bracket notation.

```
def struct2Str(edge, chart, grammar):
  """TODO: Returns a string representation of the parse with
    labeled brackets.

    Parameters:
    edge - the index of edge in the chart
    chart - the current chart (list of edges)
  """
  temporaryString = ""
  #print("Chart:", chart)
  #print("Edge:", edge)
  enums = chart[edge][5]
  temporaryString = "".join((temporaryString, "[", grammar.id2s(chart[edge][3])))
  for symbol in chart[edge][4]:
    if grammar.isTerminal(symbol): #symbols are terminals i.e. they are words of sentence
      temporaryString = " ".join( (temporaryString, grammar.id2s(symbol)) )
    else:                  # symbols are not terminals i.e. not words
      temporaryString = " ".join( (temporaryString, struct2Str(enums[0], chart, grammar)) )
      enums = enums[1:]
    #print("temporaryString:", temporaryString)
  temporaryString = "".join( (temporaryString, " ]") )
```

```
    return temporaryString
```

3. **getParse function: This function would be called to obtain the different parses of the sentence input.**

```
def getParse(inputlength, chart, grammar):
  """TODO: Returns a list of all parses in bracketing notation."""
  parses = []
  for i in range(len(chart)):
    if not isActive(chart[i]):
      if chart[i][0] == 0 and chart[i][1] == inputlength: # got spanning edge
        print("Successfully parsed!")
        #parses.append(struct2Str(chart[i],chart,grammar))
        #print("Chart=",chart)
        #print("edge = ",i)
        #changed to use PROB if we are using parameter -pr, --probability to get probabilistic version of
        #the algorithm
        #also using i instead of chart[i], since i would give location of the edge in the chart which already
        #has the information so no need to save chart[i] structure
        if PROB:
          #adding getProb function call to get probability of the parse as well
          parses.append([struct2Str(i,chart,grammar),getProb(i,chart,grammar)])
        else:
          #only calling the parse since this is without using the probabilistic version
          parses.append(struct2Str(i,chart,grammar))
  return parses
```

Part 4

4. Modify the code to implement a version of probabilistic parsing for the Earley algorithm using a similar method as that discussed in class for the CYK algorithm. Each dotted rule

will have a probability (log-probability) that is the product (sum) of the rule's probability (log-probability) with the probabilities (log-probabilities) of all the completed children it covers so far. Test the system using a probabilized version of simple.gr (see "prob-simple.gr"). Test the system on ambiguous sentences to see if the highest probability parses are indeed the most "reasonable" parses. Write a couple of ideas of how you might improve the system yet further.

Ans: In order to get the above working, we need to first save the probabilities associated with the different rules as well save these rules and probabilities in a structure that would be easily available to us.

So, we first alter PSGParse3.py file and do the below alterations.

In __init__() function we need to add the below lines , to get a dictionary structure to hold our rules and its probability just before the 'self.load(filename)' line.

```
#adding a dictionary to hold the grammar rules containing both the lhs and rhs symbols

self.myrules      = {}
```

In the load() function, we need to add after line# 92 the below lines of code,

```
#getting the probability of the rules from the regex

prob = res.group('num')

# adding to the myrules dictionary the ruletuple containing lhs and rhs symbols and assigning
#them their probability

self.myrules[ruletuple] = prob
```

With that done, we now need to modify the ChartyPy3.py file to calculate as well as call the probability calculation function.

In order to get probability of the parses we would add a new function called getProb(), as shown below.

getProb() – This function returns a log probability of the parse by summing the log probabilities of all rules that are invoked during the parsing of the sentence.

```
def getProb(edge, chart, grammar):

    """TODO: Returns a probability of the parse
```

```
    Parameters:
    edges - the index of the edge in the chart
    chart - the current chart (list of edges)
    """
    symbollist = []
    total_prob = 0.0
    #total_prob = 1.0
    #print("Chart:", chart)
    #print("Edge:", edge)
    edgenums = chart[edge][5]
    #append to rhs the list of symbols
    symbollist.append(grammar.id2s(chart[edge][3]))
    for x in chart[edge][4]:
      if grammar.isTerminal(x):
        #get terminal so the element preceding this is the lhs of this rule
        rhsterminal = symbollist.pop()
        lhsterminal = grammar.id2s(x)
        #get probability from the myrules dictionary
        prob = float(grammar.myrules[rhsterminal,(lhsterminal,)])
        if prob > 0.0:
          total_prob += log(prob)
          #total_prob *= prob
        #string1 = grammar.myrules[rhsterminal,(lhsterminal,)]
        #print("RHS:{} LHS:{} prob:{}
total_prob:{}".format(rhsterminal,lhsterminal,prob,total_prob))
      else:
        #append the remaining elements from the chart's rhs list to our local symbol list
```

```
        symbollist.append(grammar.id2s(x))
        #recursive call for the x element
        total_prob += getProb(edgenums[0], chart, grammar)
        #total_prob *= getProb(edgenums[0], chart, grammar)

        #get remaining elements from our chart rhs list
        edgenums = edgenums[1:]
    #all the elements of lhs and rhs are now gathered so the first element in this list is our rhs and remaining are lhs
    #print("Before prob:", total_prob)
    if symbollist: #if any remaining elements are present go inside
        #get probability of the remaining symbols
        prob_int = float(grammar.myrules[symbollist[0],tuple(symbollist[1:])])
        total_prob += log(prob_int)
        #total_prob *= prob_int
        #print("symbollist: {}, After Prob:{}".format(symbollist,total_prob))
    #print("symbollist:", symbollist)
    return total_prob
```

Now that we have the functionality we also need to be able to call it, for which I have implemented it using the command line option '-pr' or '--prob'. To get this functionality as well to get the remaining to work gracefully with this, I introduced a global variable PROB, which would only print and invoke the function call to getProb function when we use -pr in the command line arguments. The code modifications to handle this, was introduced in functions printParses and getParse and are already mentioned in part 3 of this assignment.

We also modified the main function in ChartyPy3.py file to add below logic just before the line 'args = parser.parse_args()' on line#246 in the original unmodified file.

```
parser.add_argument("-pr", "--prob",
        action="store_true", dest="PROB", default=False,
        help="don't print the probability content  [default False]")
```

Finally, in order to run the probabilistic version, we can do it by passing the below arguments on the command line interface.

python ChartyPy3.py -g ../grammars/prob-simple.gr -i "john drives to chicago" -pr

or

python ChartyPy3.py -g ../grammars/prob-simple.gr -i "john drives to chicago" –prob

Ambiguous sentences:

On running the above code for the ambiguous sentence “john drives a jetcar in a watermelon” we get the below 3 parses for it, along with their log probabilities.

Parsing: 1 of 3

[s [np [fname john]] [vp [vbar [vbar [fv drives]] [np [fd a] [nbar [fn jetcar]]]] [vbar-mods [pp [fp in] [np [fd a] [nbar [fn watermelon]]]]]]]

With log probability: -22.754145812011814

Parsing: 2 of 3

[s [np [fname john]] [vp [vbar [vbar [fv drives]] [np [fd a] [nbar [fn jetcar]]] [pp [fp in] [np [fd a] [nbar [fn watermelon]]]]]]]

With log probability: -23.005460240292717

Parsing: 3 of 3

[s [np [fname john]] [vp [vbar [vbar [fv drives]] [np [fd a] [nbar [fn jetcar] [nbar-mods [pp [fp in] [np [fd a] [nbar [fn watermelon]]]]]]]]]]

With log probability: -21.61916587917283

On examining them we can see that the last parse (3) with the highest log probability does have the most “reasonable” parse.

Further Code Improvements

We can further improve the above code by implementing some of the below ideas.

- Introducing an exception check for handling sentences with words not present in the grammar
- Adding functionality to get sentences from files and check against them batchwise and store the output in a file.

- Inorder to better explain the functionality of parsing using the given grammar, a function to generate visual aids like building a graph like structure using python packages like NetworkX could also be done.
- Adding a checking mechanism, when using probabilistic version of the algorithm, to ensure that the grammar file being used does indeed has assigned some probability greater than 0 to the rules in the grammar file.