Auto-Sentence Structure Parser

1. Description of the project:

Language processing is one of the major fields of artificial intelligence technology. Its goal is to make machines be able to read human language comprehensively and use human language to communicate with human beings. To achieve this goal, the first step is to parse the sentence structure correctly. Auto-Sentence Structure Parser (ASSP) is a program that automatically parses the input, which is a sentence, and displays the sentence structure as an output based on corpora of grammar and lexicon. It is quite generic so that there is not many specifications when use this program. Also, it has a lot of feasibility to add other features in. Section 2 will describe how program works and show the sample runs. Section 3 will be an analysis to the program algorithm and performance, and Section 4 will talk about other features that can be extended from this program.

2. Methodology and Work Flow:

ASSP uses a top-down parsing strategy. It searches for a parse tree by trying to build from the root node “S” down to the leaves. (Speech And Language Processing, Page 429) I choose this method because it is pretty straight forward. The program is coded in C# under a .NET environment. The only outside source that ASSP requires is the corpora of grammar and lexicon, which is saved in an xml file. Without this file, ASSP does not have the corpora to reference to. Figure 1 shows an example of grammar and lexicon.

ASSP basically handles four tasks:

1. Read xml and put the grammar and lexicon in a dataset
2. Parse input sentence into a list of words

3. Parse out the sentence structure given a list of words

4. Generate output

2.1 Read xml

The first challenge from this project is creating a class that parses the xml into a dataset. To store the xml content, I create an empty dataset initially. Then I use XmlTextReader class which is from System.Xml to loop through the xml file. Inside loop, grammar, element_group, word, and lexicon are processed separately. In a result, this class will come up with a dataset that includes two tables: “Grammar” and “Lexicon”. In Figure 2, it shows an example of the result dataset.

2.2 Parse the sentence in to word list

ASSP takes one input from the input textbox. Then it parses the input sentence by tracking the space (“ “) in the input sentence. From my point of view, this is not a key part of this project because it does not refer to any grammar or lexicon. Therefore I did not spend too much time to create a super clever parser to parse any sentence into a word list. Right now, ASSP cannot parse any sentence that has punctuation inside sentence except the period in the end. However, this part can always be improved by writing a good “sentence to word” parser and replace the old one the ASSP using.

2.3 Parse the word list into sentence structure

The most important part of this project is to parse the word list into a sentence structure correctly. To simplify coding and improve the readability of the code, I create three classes to represent the three level structures: ob_Sentence,(Sentence) ob_Node(Node), and ob_Edge(Edge). Sentence is the highest level. It has a list of nodes and a list of edges. It also holds a list of
complete edges to facilitate output generation. Node is the second level. Node starts from 0 to the
count of the word in the list and each node holds a word. It also has a list of roles that is requesting
(my_role), a list of roles that this word can actually act (word_type), a list of edges that start at this
node and a list of edges that end at this node (edge list stored in node is not used in ASSP but
possibly used in other program). Edge is the lowest level. It stores some basic information such as
“edge from”, “edge to”, “constituent”, “current category” and “next category” that forms the
constituent. Since I use the top-down parsing strategy, “S” is added to the node0’s “my_role”
initially. Then the code starts the highest level loop which goes through all the nodes. Inside that
loop, there are another two loops. The first loop is to populate current node’s “my_role” and
“word_type”. The second loop is to check if any previous edge is completed by this node and
populate next node’s “my_role”. If the node completes previous edge, it will recursively check all
the edges that related with the complete edge until no more edge can be completed. During this
recursion, program gathers all the new roles for the next node. The loop structure is showed in
Figure 3. Sample run is showed in Figure 4.

2.4 Generate output

ASSP simply display all the complete edges as output.

3. Performance analysis

The program complexity depends on two things: size of corpora of grammar and lexicon,
complexity of grammar. The bigger the corpora size is, the longer it takes to check through the
corpora. The more complicate the grammar is, the more roles a node can act and the more edges
will be created. Because the corpora are from the outside, it is really hard to come up a
performance model. However compare to the size of corpora, the complexity will be the key
factor that affects the ASSP performance because the number of computation is likely to increase exponentially. One of the drawbacks for ASSP is that it has to read the grammar and lexicon every time it runs. Assume the corpora is huge, this step will kill the performance. If it can link to a database, it will save some time on loading data and generating queries. Another possible way to improve the performance is parallel computing. Since ASSP is basically a large number of querying, it should be easy to implement in parallel.

4. Possible features based on ASSP

First, features and unifications can be added to the corpora so that “love” and “loves” do not have to be saved in two records. It will help when dealing with different tenses for the regular verb, different amount for countable noun and so on. Second, program that works on semantics can be built based on successfully parsing the sentence. It could use the output of ASSP as a input so that it is easy when representing the meaning of a complicate sentence.

5. Conclusion

The biggest challenge for ASSP is how to translate the top-down parsing method into computer language. In this project, I only test on the small simple corpora that I built. However, in real world, the grammar structure is much more complicate. There are a lot of challenges in language parsing. It is not hard to make a program that parses a regular sentence because basically the code just needs to follow rules. The hardest part in language parsing, I believe, should be how to summarize all grammar and lexicon used in real world into corpora correctly.

Bibliography & Resources

Speech And Language Processing, by Daniel Jurafsky & James H.Martin, Second Edition
Microsoft Visual Studio .Net 2005

Wikipedia

Appendix

Figure 1:

```xml
<rule>
  <all_grammar>
    <grammar>
      <title>S</title>
      <element_group>
        <element>NP</element>
        <element>VP</element>
      </element_group>
    </grammar>
  </all_grammar>
  <all_lexicon>
    <lexicon>
      <type>Det</type>
      <word_group>
        <word>the</word>
        <word>a</word>
        <word>an</word>
      </word_group>
    </lexicon>
  </all_lexicon>
</rule>
```

Figure 2:

DataSet Structure

Table “Grammar”

<table>
<thead>
<tr>
<th>Col_Title</th>
<th>Col_Count</th>
<th>Col_Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>2</td>
<td>NP-VP-</td>
</tr>
<tr>
<td>NP</td>
<td>2</td>
<td>Det-N-</td>
</tr>
<tr>
<td>NP</td>
<td>1</td>
<td>N-</td>
</tr>
<tr>
<td>VP</td>
<td>2</td>
<td>V-NP-</td>
</tr>
<tr>
<td>V</td>
<td>1</td>
<td>V-</td>
</tr>
</tbody>
</table>
Table “Lexicon”

<table>
<thead>
<tr>
<th>Col_Type</th>
<th>Col_Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Cat</td>
</tr>
<tr>
<td>N</td>
<td>Dog</td>
</tr>
<tr>
<td>Det</td>
<td>The</td>
</tr>
<tr>
<td>V</td>
<td>Love</td>
</tr>
<tr>
<td>V</td>
<td>Chase</td>
</tr>
</tbody>
</table>

**Figure 3:**

```csharp
for ( Node, from 0 to N)
{
    while (Node.myrole has new roles)
    {
        for (Role, from 0 to Count of new roles)
        {
            Search Role in dataset
            foreach (DataRow in GrammarSearchResult)
            {
                Process DataRow
            }
        }
    }
}

foreach(WordType in Node.wordtype)
{
    Process wordtype
    foreach (EdgeA in Node.ownEdges)
    {
        // ownEdge means edge from and edge to is same
        Find matching edge
        Check_Previous_Edges(EdgeA.FromNode.ownEdges);
        Check_Previous_Edges(EdgeA.FromNode.endEdges);
        // endEdge means all edge end at this node
    }
    foreach (ob_Edge edgeB in Node.endEdges)
    {
        Find matching edge
        Check_Previous_Edges(EdgeB.FromNode.ownEdges);
        Check_Previous_Edges(EdgeB.FromNode.endEdges);
    }
}
```
Figure 4:

Type in Sentence in textbox and click “Parse”

(Initial State)

Only the complete edges will be displayed

(After Parsing)