11-711 Algorithms for NLP

GLR Parsing

Reading:

Tomita,

“An Efficient Augmented Context-Free Parsing Algorithm”

Computational Linguistics 13(1-2), pp. 31–46
GLR Parsing

Goal:

- Extend LR Parsing to handle general CFGs - including ambiguous grammars
- Make the LR Parsing techniques applicable to parsing NL
- Preserve the efficiency of the LR approach as much as possible

General Method:

- Extend the LR parsing algorithm to handle conflicting actions
- Use the constructed LR tables as is - usually SLR table
- Pursue in a “pseudo-parallel” fashion all possible parsing actions
- The “closer” the grammar is to being LR - the fewer conflicts - the closer to linear the performance
The Graph Structured Stack

- Compact representation of a set of LR stacks
- Contains state nodes and symbol nodes (like an LR stack)
- Initial state of parser is at the “bottom” of the GSS.
- Each path from the “bottom” of the GSS to a leaf node corresponds to an LR stack
- GSS contains a set of active state nodes

Splitting the Stack:
- When the parsing table specifies multiple actions, the GSS is split into separate branches, each of which pursues one of the actions

Merging Branches:
- When separate branches of the GSS end in identical active states, they are merged back together
- Merging allows efficient pursuit of further actions of the parser that are common to the branches
The Graph Structured Stack

Examples of Splitting and Merging of the GSS:

```
0  a  1  b  2
○  □  ○  □  ●
```

```
0  a  1  A  3
○  □  ○  □  ●
```

```
0  a  1  B  4
○  □  ○  □  ●
```

```
0  a  1  A  3  c  5
○  □  ○  □  ○  □  ○  ●
```

11-711 Algorithms for NLP
The GLR Parsing Algorithm

- The Algorithm operates in Stages - each processing a single input word
- Each stage consists of three Phases:
  - **Reduce Phase**: Perform all Reduce actions defined on the set of active state nodes
  - **Shift Phase**: Perform all Shift actions defined on the set of active state nodes
  - **Merge Phase**: Merge active state nodes that have same state
The GLR Parsing Algorithm

The Algorithm in Pseudo-code:

\[
\begin{align*}
ip &= 1 \\
active &= \{s_0\} \\
\text{while (ip} &\leq (n+1)) \text{ do} \\
&\quad a = \text{the input word pointed by ip} \\
&\quad \text{if active is empty then return with ‘‘Error’’} \\
&\quad \text{if ip} = (n+1) \text{ then} \\
&\quad \quad \text{If for every s in active action}[s,a] = \text{Accept} \\
&\quad \quad \quad \text{then return with ‘‘Accept’’} \\
&\quad \quad \text{else return with ‘‘Reject’’}
\end{align*}
\]

/* Reduce Phase: */

\[
\begin{align*}
&\quad \text{for every s in active:} \\
&\quad \quad \text{if action}[s,a] = \text{Reduce by (rule-i: A} \rightarrow \text{Beta)} \text{ then:}
\end{align*}
\]
find all paths p in the GSS of length 2*|Beta|
for each path p:
    Pop the 2*|Beta| elements from the path
    and reveal state s’
    Push A on top of s’ and then the state
    s’’ = goto[s’,A]
    Add s’’ to active

/* Shift Phase: */
for every s in active:
    if action[s,a] = Shift s’ then:
        Push a on top of s and then the state s’
        Add s’ to active

/* Merge Phase: */
For every pair of state nodes s and s’ in active
    if state s = state s’ then merge them

ip = ip+1
Efficient Ambiguity Representation

- **Sub-tree Sharing:**
  - Sharing of common sub-trees between full parse trees
  - Symbols in the GSS contain *pointers* to parse tree constituents
  - Whenever a constituent participates in more than one reduction, the same pointer is used

- **Local Ambiguity Packing:**
  - A *Local Ambiguity* - substring analyzable as a constituent $A$ in more than one way
  - Represented in the GSS as two similarly labeled symbol nodes sharing the same state nodes on both sides
  - A procedure can detect these situations and pack the nodes by:
    1. Creating a single parse node corresponding to the constituent
    2. Collapsing the two GSS paths into a single path
Efficient Ambiguity Representation

Examples:

Figure 3.1. Unpacked shared forest.

Figure 3.2. Packed shared forest.
GLR Parsing - Example

The Grammar:

(1) \( S \rightarrow NP VP \)
(2) \( S \rightarrow S PP \)
(3) \( NP \rightarrow n \)
(4) \( NP \rightarrow det n \)
(5) \( NP \rightarrow NP PP \)
(6) \( PP \rightarrow prep NP \)
(7) \( VP \rightarrow v NP \)

The original input: “\( x = I \) saw a man in the park with a telescope”
POS assigned input: “\( x = n \ v \ det \ n \ prep \ det \ n \ prep \ det \ n \)”
Parser input: “\( x = n \ v \ det \ n \ prep \ det \ n \ prep \ det \ n \$”
GLR Parsing - Example

Constructed Parsing Table for the Grammar:

<table>
<thead>
<tr>
<th>State</th>
<th>*det</th>
<th>*n</th>
<th>*v</th>
<th>*prep</th>
<th>$</th>
<th>NP</th>
<th>PP</th>
<th>VP</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sh3</td>
<td>sh4</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>sh6</td>
<td>acc</td>
<td>5</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>sh7</td>
<td>sh6</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>sh10</td>
<td></td>
<td>re3</td>
<td>re3</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>re2</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>sh3</td>
<td>sh4</td>
<td>re3</td>
<td>re3</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>re2</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>sh3</td>
<td>sh4</td>
<td>re3</td>
<td>re3</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>re5</td>
<td>re5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td>re4</td>
<td>re4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td>re6</td>
<td>re6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td>re6,sh6</td>
<td>re6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>re7,sh6</td>
<td>re7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2. LR parsing table with multiple entries.
GLR Parsing - Example

Parser input: “$x = n v \text{det } n \text{ prep } \text{det } n \text{ prep } \text{det } n$”
Handling Multi POS Words

- When we have a POS ambiguity for a word - handle as a “shift-shift” table conflict: perform multiple shift actions
- Top-down prediction constraints that are implicit in the parser states restrict the set of possible POS at any point in the parsing of an input
- Paths that pursue a POS that is grammatically inconsistent with later portions of the input are eliminated during parsing
- Similar in principle to the way this can be done with Earley
- See example in Tomita article
- Unknown words can be handled similarly - assume the word can have any possible POS (allowed by the current state)
Time Complexity of the GLR Parser

- Time complexity analysis is not straightforward
- Main factor is the number of reduce operations per stage
- Worst-case time bound for general CFGs is $O(n^{p+1})$ where $p$ is the length of the longest grammar rule
- Inferior to Earley and Chart Parser
- Modified version of GLR can do recognition in $O(n^3)$ time
- In practice - GLR time performance is slightly worse than linear with practical large NL grammars
- In comparative evaluations - outperformed both Earley and Chart parsers
- Thus - using GLR is worthwhile: NL grammars are “close enough” to being LR