11-722 Grammar Formalisms

Parsing Tree-Adjoining Grammars

Alon Lavie

February 27, 2006

References:
Aravind Joshi & Yves Schabes, “Tree-Adjoining Grammars”
Anthony Kroch & Aravind Joshi, “The Linguistic Relevance of TAG”
XTAG Online Tutorial: http://www.cis.upenn.edu/ xtag/tutorial.html
**Parsing TAGs**

- **Goal:** define practical parsers for TAGs that can be easily modified to handle extensions to basic TAGs

- There is a CYK-like algorithm for parsing TAGs
  - Worst-case time complexity: $O(n^6)$
  - Best-case time complexity: $O(n^6)$
  - Purely bottom-up (no predictive information)
  - Requires binary elementary trees (0 or 2 branches at each node)

- For CFGs, parsing algorithms that use top-down filtering information are faster and more efficient in practice (but same worst-case complexity)

- **Goal:** design an efficient bottom-up TAG parser that scans the input left to right and uses top-down information

- The solution is not obvious since adjunction allows the splicing of one tree (string) into the middle of another

- **Approach:** Adapt Earley’s parsing algorithm for CFGs: its average-case time complexity depends on the grammar and tends to be better than the worst-case
Dotted Trees

- Use a **dotted tree** data structure similar to dotted CFG rules
  - At any time there is only one dot in a tree
  - The “dot” represents “where we are”: what portion of the tree has been matched with the input, and what remains to be matched
  - Allows us to explicitly define parser “configurations” - how the parser moves the dot through a tree

- The *dot* can be in one of four positions with respect to a node in a tree: left and above (*la*), left and below (*lb*), right and below (*rb*), right and above (*ra*)

- A tree traversal starts with the dot at the *la* position of the root and ends when the dot reaches the *ra* position of the root

- Two dot positions are equivalent if no node is crossed between them
A Dotted Tree

10.1. Example of a tree traversal.
The Chart

- The algorithm collects items in a chart rather than constructing sets of items
- A chart item is defined by an 8-tuple $s = [\alpha, dot, pos, i, j, k, l, sat?]
  - $\alpha$ is an elementary tree
  - $dot$ is the address of the dot in $\alpha$
  - $pos \in \{la, lb, rb, ra\}$ is the position of the dot
  - $i, j, k, l \in \{0, \ldots, n\} \cup \{-\}$ are indices of positions in the input string.
    * $i \leq j \leq k \leq l$.
    * $-$ means the index is unbound.
  - $i$ and $l$ are endpoints of the input segment covered by the item (always bound).
  - $j$ and $k$ are the endpoints of the input segment covered by a tree attached to the foot node of $\alpha$
  - $sat? \in \{true, nil\}$ is a boolean indicating whether an adjunction has been recognized on the node at address $dot$ in $\alpha$
Parsing Operations

- There are five main chart operations:
  1. **SCAN**
  2. **PREDICT**
  3. **COMPLETE**
  4. **ADJOIN**
  5. **SUBSTITUTE**
The Parsing Operations

Fig. 10.7. The four operations of the parser.
Parsing Operations: SCAN

- SCAN is a bottom-up operation that scans the input string
- The SCAN operation has two cases:
  1. The dot is left and above a non-empty terminal that matches the next item in the input → Consume the input item
     
     \[
     [\alpha, \text{dot}, l_a, i, j, k, l, \text{nil}] \rightarrow [\alpha, \text{dot}, r_a, i, j, k, l + 1, \text{nil}]
     \]
  2. The dot is left and above an empty symbol → Move the dot without consuming any input item
     
     \[
     [\alpha, \text{dot}, l_a, i, j, k, l, \text{nil}] \rightarrow [\alpha, \text{dot}, r_a, i, j, k, l, \text{nil}]
     \]
The **SCAN** Operation

\[ a = a \downarrow + 1 \]

\[ \varepsilon \]

\[ [i,j,k,l,nil] \]

\[ [i,j,k,l,nil] \]

\[ [i,j,k,l,nil] \]

\[ [i,j,k,l,nil] \]

\[ [i,j,k,l,nil] \]

\[ [i,j,k,l,nil] \]

\[ [i,j,k,l,nil] \]
Parsing Operations: PREDICT

- **PREDICT** is a top-down operation that predicts new items based on what has already been seen

- The **PREDICT** operation has three cases for adjunction and one for substitution
  
  1. The dot is left and above a non-terminal that allows adjoining
     → Predict all auxiliary trees adjoinable at the dotted node
  2. The dot is left and above a non-terminal that allows adjoining
     → Predict that no adjoining takes place at the dotted node
  3. The dot is left and below the foot node of an auxiliary tree
     → Try to recognize the subtree below any node to which the auxiliary tree could have been adjoined
  4. The dot is left and above a non-terminal marked for substitution
     → Predict all initial trees that could be substituted at the dotted node
The PREDICT Operation

(1) \[
\begin{array}{c}
\text{A} \\
[i,j,k,l,nil]
\end{array}
\rightarrow
\begin{array}{c}
\text{A} \\
[l,\cdot,\cdot,l,nil]
\end{array}
\]

(2) \[
\begin{array}{c}
\text{A} \\
[i,j,k,l,nil]
\end{array}
\rightarrow
\begin{array}{c}
\text{A} \\
[l,\cdot,\cdot,l,nil]
\end{array}
\]

(3) \[
\begin{array}{c}
\text{A} \\
[l,\cdot,\cdot,l,nil]
\end{array}
\rightarrow
\begin{array}{c}
\text{A} \\
[l,\cdot,\cdot,l,nil]
\end{array}
\]

PREDICT
Parsing Operations: COMPLETE

- COMPLETE is a bottom-up operation that combines two items to form a new item that spans a larger portion of the input.

- The COMPLETE operation has two cases:
  1. The dot is right and below a non-terminal that allows adjoining:
     → Assume that the next input token comes from the part right of the foot node of an auxiliary tree adjoined on the dotted node.
  2. The dot is right and below a non-terminal:
     → Try to further recognize the same tree by combining two items from the tree.
The **COMPLETE Operation**

(1) 

![Diagram 1](image1)

\[ [i,j,k,l,nil] \]

\[ [i,-,-,i,nil] \]

\[ [i,i,l,l,nil] \]

(2.1) 

![Diagram 2](image2)

\[ [i,j,k,l,sat?] \]

\[ [h,-,-,i,nil] \]

\[ [h,j,k,l,nil] \]

(2.2) 

![Diagram 3](image3)

\[ [i,-,-,l,sat?] \]

\[ [h,j,k,i,nil] \]

\[ [h,j,k,l,nil] \]
 Parsing Operations: ADJOIN

- ADJOIN is a bottom-up operation that combines two items by adjunction to form a new item that spans a larger portion of the input.
- The ADJOIN operation has a single case:
  1. The dot is right above a non-terminal that allows adjoining
     → Adjoin a completed auxiliary tree at the dotted node.
The ADJOIN Operation
Parsing Operations: SUBSTITUTE

- SUBSTITUTE is a bottom-up operation that combines two items by substitution to form a new item that spans a larger portion of the input.

- The SUBSTITUTE operation has a single case:
  1. The dot is left and above a non-terminal marked for substitution
     → Substitute a completed initial tree at the dotted node.
TAG Parsing Algorithm

- The algorithm imposes no conditions on the grammar
  - Elementary trees are not required to be binary
  - The empty string may appear on the frontier of elementary trees
- The algorithm is off-line (it needs to know the length of the input before starting) but can be modified to run on-line
- The algorithm is an agenda driven parser
- An input string is accepted if a tree that is derived from an initial tree and spans the input is found (i.e. if an item of the form \([\alpha, 0, ra, 0, -, -, n, nil]\) is in the chart)
The Recognizer Algorithm

Let $G = (\Sigma, NT, I, A, S)$ be a TAG.
Let $a_1 \cdots a_n$ be the input string.

program recognizer
begin
$C = \{ [\alpha, o, la, 0, -, -, 0, nil] | \alpha \in I, \alpha(0) = S \}$

Apply the following operations on each item in the chart $C$
until no more items can be added to the chart $C$:

1. $[\alpha, \text{dot}, la, i, j, k, l, nil] \quad \alpha(\text{dot}) \in \Sigma, \alpha(\text{dot}) = a_{i+1}$
2. $[\alpha, \text{dot}, ra, i, j, k, l + 1, nil] \quad \alpha(\text{dot}) \in \Sigma, \alpha(\text{dot}) = \epsilon$
3. $[\alpha, \text{dot}, la, i, j, k, l, nil] \quad \alpha(\text{dot}) \in NT, \beta \in \text{Adj}(\alpha, \text{dot})$
4. $[\alpha, \text{dot}, lb, i, j, k, l, nil] \quad \alpha(\text{dot}) \in NT, OA(\alpha, \text{dot}) = \text{false}$
5. $[\alpha, \text{dot}, lb, i, j, k, l, nil] \quad \text{dot} = \text{Foot}(\alpha), \alpha \in \text{Adj}(\alpha, \text{dot})$
6. $[\beta, \text{dot'}, lb, i, j, k, l, nil] \quad \begin{align*}
\beta(\text{dot'}) & = \text{Foot}(\beta), \\
\beta & \in \text{Adj}(\beta, \alpha, \text{dot})
\end{align*}$
7. $[\alpha, \text{dot}, ra, i, j, k, l, \text{true}] \quad \alpha(\text{dot}) \in NT$
8. $[\beta, 0, ra, i, j, k, l, \text{true}] \quad \beta \in \text{Adj}(\beta, \alpha, \text{dot})$

If there is an item of the form $[\alpha, 0, ra, 0, -, -, n, nil]$ in $C$ with $\alpha \in I$
and $\alpha(0) = S$ then return acceptance, otherwise return rejection.

end.

Fig. 10.8. Pseudo-code for the recognizer.
Example: The Grammar

Fig. 10.9. TAG generating $L = \{a^n b^n e c^n d^n | n \geq 0\}$
## Example: Recognition

<table>
<thead>
<tr>
<th>Input read</th>
<th>Items in the chart</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. [a, dot : 0, la, 0, , , , 0, nil] init</td>
<td></td>
</tr>
<tr>
<td>2. [β, dot : 0, la, 0, , , , 0, nil] pred(1)</td>
<td></td>
</tr>
<tr>
<td>3. [α, dot : 1, la, 0, , , , 0, nil] pred(1)</td>
<td></td>
</tr>
<tr>
<td>4. [β, dot : 1, la, 0, , , , 0, nil] pred(2)</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>5. [β, dot : 2, la, 0, , , , 1, nil] sc(4)</td>
</tr>
<tr>
<td>a</td>
<td>6. [β, dot : 0, la, 1, , , , 1, nil] pred(5)</td>
</tr>
<tr>
<td>a</td>
<td>7. [β, dot : 2, la, 1, , , , 1, nil] pred(5)</td>
</tr>
<tr>
<td>a</td>
<td>8. [β, dot : 1, la, 1, , , , 1, nil] pred(6)</td>
</tr>
<tr>
<td>aa</td>
<td>9. [β, dot : 2, la, 1, , , , 2, nil] sc(8)</td>
</tr>
<tr>
<td>aa</td>
<td>10. [β, dot : 0, la, 2, , , , 2, nil] pred(9)</td>
</tr>
<tr>
<td>aa</td>
<td>11. [β, dot : 2, la, 2, , , , 2, nil] pred(9)</td>
</tr>
<tr>
<td>aa</td>
<td>12. [β, dot : 1, la, 2, , , , 2, nil] pred(10)</td>
</tr>
<tr>
<td>aab</td>
<td>13. [β, dot : 2, la, 3, , , , 3, nil] sc(11)</td>
</tr>
<tr>
<td>aab</td>
<td>14. [β, dot : 2, lb, 3, , , , 3, nil] pred(11)</td>
</tr>
<tr>
<td>aab</td>
<td>15. [β, dot : 2, la, 3, , , , 3, nil] pred(14)</td>
</tr>
<tr>
<td>aab</td>
<td>16. [α, dot : 1, la, 3, , , , 3, nil] pred(14)</td>
</tr>
<tr>
<td>aab</td>
<td>17. [β, dot : 2, lb, 3, , , , 4, nil] sc(15)</td>
</tr>
<tr>
<td>aab</td>
<td>18. [β, dot : 2, lb, 4, , , , 4, nil] pred(17)</td>
</tr>
<tr>
<td>aab</td>
<td>19. [β, dot : 2, la, 4, , , , 4, nil] pred(18)</td>
</tr>
<tr>
<td>aab</td>
<td>20. [α, dot : 1, la, 4, , , , 4, nil] pred(18)</td>
</tr>
<tr>
<td>aabbe</td>
<td>21. [α, dot : 0, rb, 4, , , , 5, nil] sc(20)</td>
</tr>
<tr>
<td>aabbe</td>
<td>22. [β, dot : 2, rb, 4, 4, 5, nil] comp(21+18)</td>
</tr>
<tr>
<td>aabbe</td>
<td>23. [β, dot : 2, lb, 3, 4, 5, nil] comp(22+15)</td>
</tr>
<tr>
<td>aabbec</td>
<td>24. [β, dot : 2, rb, 3, 4, 5, nil] sc(23)</td>
</tr>
<tr>
<td>aabbec</td>
<td>25. [β, dot : 2, rb, 3, 3, 6, 6, nil] comp(24+14)</td>
</tr>
<tr>
<td>aabbec</td>
<td>26. [β, dot : 2, rb, 3, 2, 3, 6, 6, nil] comp(25+13)</td>
</tr>
<tr>
<td>aabbec</td>
<td>27. [β, dot : 2, rb, 2, 3, 6, 7, nil] sc(26)</td>
</tr>
<tr>
<td>aabbec</td>
<td>28. [β, dot : 3, la, 1, 3, 6, 7, nil] comp(26+9)</td>
</tr>
<tr>
<td>aabbec</td>
<td>29. [β, dot : 0, rb, 1, 3, 6, 8, nil] sc(28)</td>
</tr>
<tr>
<td>aabbec</td>
<td>30. [β, dot : 0, rb, 1, 3, 6, 8, nil] comp(28+6)</td>
</tr>
<tr>
<td>aabbec</td>
<td>31. [β, dot : 2, rb, 1, 4, 5, 8, 6] adj(30+24)</td>
</tr>
<tr>
<td>aabbec</td>
<td>32. [β, dot : 3, la, 0, 4, 5, 8, 6] comp(31+5)</td>
</tr>
<tr>
<td>aabbec</td>
<td>33. [β, dot : 0, rb, 0, 4, 5, 9, nil] sc(32)</td>
</tr>
<tr>
<td>aabbec</td>
<td>34. [β, dot : 0, ra, 0, 4, 5, 9, nil] comp(33+2)</td>
</tr>
<tr>
<td>aabbec</td>
<td>35. [α, dot : 0, rb, 0, , , , 9, nil] adj(34+21)</td>
</tr>
<tr>
<td>aabbec</td>
<td>36. [α, dot : 0, ra, 0, , , , 9, nil] comp(35+1)</td>
</tr>
</tbody>
</table>

**Fig. 10.10.** Items constituting the chart for the input: \( a @ a b c d e f g h i \)
The worst-case time complexity of the general algorithm is $O(|A||A \cup I|Nn^6)$
- $A$ = number of auxiliary trees
- $I$ = number of initial trees
- $N$ = maximum number of nodes in an elementary tree

The ADJOIN operation may be called at most $|A||A \cup I|Nn^6$ times
- There are at most $n^6$ instances of the indices $(i, j, k, l, p, q)$
- There are at most $|A| \times |A \cup I|N$ pairs of dotted trees to combine

Unambiguous TAGs can be parsed in $O(|A||A \cup I|Nn^4)$ time

A large class of unambiguous TAGs can be parsed in linear time