AN INCREMENTAL ALGORITHM FOR TRANSITION-BASED CCG PARSING

B. R. Ambati, T. Deoskar, M. Johnson and M. Steedman

Miyazawa Akira

January 22, 2016

The Graduate University For Advanced Studies / National Institute of Informatics

INCREMEMTAL PARSER

What does "Incremental" mean?

An incremental parser computes the relationship between words as soon as it receives them from the input.

Why is incrementality important?

- Statistical Machine Translation
- Automatic Speech Recognition

BASELINE ALGORITHM

Their baseline algorithm `NonInc` is based on Zhang and Clark (2011). It has four actions.

- Shift
  Push a word from the input buffer to the stack and assign it with a CCG category.
- Reduce Left
  Pop the top two nodes from the stack, combine them into a new node and push it back onto the stack with a new category. The right node is the head and the left node is reduced.
- Reduce Right
  This is similar to the action above except that the left node is the head and the right node is reduced.
- Unary
  Change the category of the top node on the stack.

0

Input buffer

John likes mangoes from India madly

Stack

Dependency graph

1 Shift

Input buffer

likes mangoes from India madly

Stack

NP_{John}

Dependency graph

2 Shift

Input buffer

mangoes from India madly

Stack

NP_{John} $(S\backslash NP)/NP_{likes}$

Dependency graph

3 Shift

Input buffer

from India madly

Stack

NP_{John} $(S\backslash NP)/NP_{likes}$ $NP_{mangoes}$

Dependency graph

4 Shift

Input buffer

India madly

Stack

NP_{John} $(S\backslash NP)/NP_{likes}$ $NP_{mangoes}$ $(NP\backslash NP)/NP_{from}$

Dependency graph

5 Shift

Input buffer

madly

Stack

NP_{John} $(S\backslash NP)/NP_{likes}$ $NP_{mangoes}$ $(NP\backslash NP)/NP_{from}$ NP_{India}

Dependency graph

6 Reduce Right

Input buffer

madly

Stack

NP_{John} $(S\backslash NP)/NP_{likes}$ $NP_{mangoes}$ $NP\backslash NP_{from}$

Dependency graph

$$\text{from} \\ \downarrow \\ \text{India}$$

7 Reduce Right

Input buffer

madly

Stack

NP_{John} $(S\backslash NP)/NP_{likes}$ $NP_{mangoes}$

Dependency graph

$$\text{mangoes} \downarrow \text{from} \downarrow \text{India}$$

8 Reduce Right

Input buffer

madly

Stack

NP_{John} $S\backslash NP_{likes}$

Dependency graph

likes
↓
mangoes
↓
from
↓
India

9 Shift

Input buffer

Stack

NP_{John} $S\backslash NP_{likes}$ $(S\backslash NP)\backslash(S\backslash NP)_{madly}$

Dependency graph

likes
↓
mangoes
↓
from
↓
India

NONINC ALGORITHM

10 Reduce Right

Input buffer

Stack

$$NP_{John} \quad S\backslash NP_{likes}$$

Dependency graph

$$\begin{tikzcd}
\text{likes} \arrow[d] \arrow[dr] & \\
\text{mangoes} \arrow[d] & \text{madly} \\
\text{from} \arrow[d] & \\
\text{India} &
\end{tikzcd}$$

11 Reduce Left

Input buffer

Stack

S_{likes}

Dependency graph

$$\begin{tikzcd}
 & \text{likes} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{John} & \text{mangoes} \arrow[d] & \text{madly} \\
 & \text{from} \arrow[d] & \\
 & \text{India} &
\end{tikzcd}$$

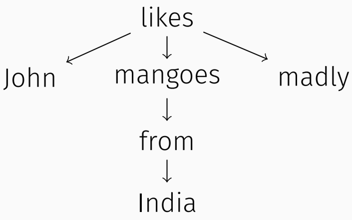
Finish

Input buffer

Stack

S_{likes}

Dependency graph



PROBLEM IN NONINC

The algorithm above is not incremental. The dependncy graph starts to grow only after almost all the words are pushed to the stack.

To solve this problem, they introduce a revealing technique (Pareschi and Steedman (1987)).

0

Input buffer

John likes mangoes from India madly

Stack

Dependency graph

1 Shift

Input buffer

likes mangoes from India madly

Stack

NP_{John}

Dependency graph

2 Shift

Input buffer

mangoes from India madly

Stack

NP_{John} $(S\backslash NP)/NP_{likes}$

Dependency graph

3-1 Type-Raising

Input buffer

mangoes from India madly

Stack

$S/(S\backslash NP)_{\text{John}}$ $(S\backslash NP)/NP_{\text{likes}}$

Dependency graph

3-2 Reduce Left

Input buffer

mangoes from India madly

Stack

S/NP_{likes}

Dependency graph

$$\text{likes} \rightarrow \text{John}$$

4 Shift

Input buffer

from India madly

Stack

S/NP_{likes} NP_{mangoes}

Dependency graph

likes

John

REVINC ALGORITHM

5 Reduce Right

Input buffer

from India madly

Stack

S_{likes}

Dependency graph

likes

John mangoes

6 Shift

Input buffer

India madly

Stack

S_{likes} $(NP\backslash NP)/NP_{\text{from}}$

Dependency graph

$$\begin{array}{ccc} & & \text{likes} \\ & \swarrow & \downarrow \\ \text{John} & & \text{mangoes} \end{array}$$

REVINC ALGORITHM

7 Shift

Input buffer

madly

Stack

$S_{\text{likes}} \quad (NP\backslash NP)/NP_{\text{from}} \quad NP_{\text{India}}$

Dependency graph

$$\begin{tikzcd} & \text{likes} \arrow[dl] \arrow[d] \\ \text{John} & \text{mangoes} \end{tikzcd}$$

REVINC ALGORITHM

8 Reduce Right

Input buffer

madly

Stack

S_{likes} $NP\backslash NP_{from}$

Dependency graph

$$
\begin{tikzcd}[row sep=small]
 & \text{likes} \arrow[dl] \arrow[d] \\
\text{John} & \text{mangoes}
\end{tikzcd}
\qquad
\begin{tikzcd}[row sep=small]
\text{from} \arrow[d] \\
\text{India}
\end{tikzcd}
$$

9 Right Reveal

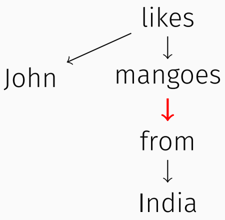
Input buffer

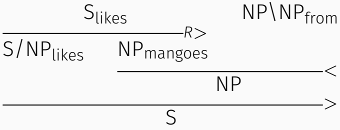
madly

Stack

S_{likes}

Dependency graph





REVINC ALGORITHM

10 Shift

Input buffer

Stack

S_{likes} $(S\backslash NP)\backslash(S\backslash NP)_{madly}$

Dependency graph

$$
\begin{array}{ccc}
 & & \text{likes} \\
 & \swarrow & \downarrow \\
\text{John} & & \text{mangoes} \\
 & & \downarrow \\
 & & \text{from} \\
 & & \downarrow \\
 & & \text{India}
\end{array}
$$

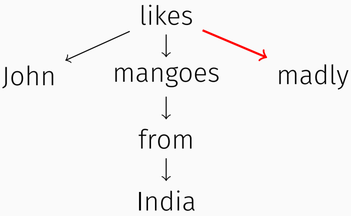
11 Left Reveal

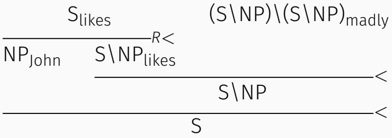
Input buffer

Stack

S_{likes}

Dependency graph





REVINC ALGORITHM

Finish

Input buffer

Stack

S_{likes}

Dependency graph

$$
\begin{tikzcd}
 & \text{likes} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{John} & \text{mangoes} \arrow[d] & \text{madly} \\
 & \text{from} \arrow[d] & \\
 & \text{India} &
\end{tikzcd}
$$

REVEALING ACTIONS

1. **Left Reveal** (LRev)
   Pop the top two nodes in the stack (left, right). Identify the left node's child with a subject dependency. Abstract over this child node and split the category of left node into two categories. Combine the nodes using CCG combinators accodingly.

$$
\begin{array}{lll}
\mathrm{S}_{\text{likes}} & & (\mathrm{S}\backslash\mathrm{NP})\backslash(\mathrm{S}\backslash\mathrm{NP})_{\text{madly}} \\
\hline \text{\scriptsize $R<$} & & \\
\mathrm{NP}_{\text{John}} & \mathrm{S}\backslash\mathrm{NP}_{\text{likes}} & \\
 & \overline{\hspace{10em}}< & \\
 & \mathrm{S}\backslash\mathrm{NP} & \\
\hline & < & \\
 & \mathrm{S} &
\end{array}
$$

2. **Right Reveal** (RRev)
   Pop the top two nodes in the stack (left, right). Check the right periphery of the left node in the dependency graph, extract all the nodes with compatible CCG categories and identify all the possible nodes that the right node can combine with. Abstract over this node, split the category into two categories accordingly and combine the nodes using CCG combinators.

$$
\begin{array}{lll}
S_{likes} & & NP\backslash NP_{from} \\
\hline \text{---------------------} R> & & \\
S/NP_{likes} & NP_{mangoes} & \\
 & \text{------------------------} < & \\
 & NP & \\
\text{-----------------------------------} > & & \\
 & S &
\end{array}
$$

Data: CCGbank

- training: sections 02–21
- development: section 00
- testing: section 23

POS tagger:

C&C POS tagger

Supertagger:

C&C supertagger

FEATURES

- NonInc
  Features based on the top four nodes in the stack and the next four words in the input buffer.

	feature templates
1	S_0wp, S_0c, S_0pc, S_0wc, S_1wp, S_1c, S_1pc, S_1wc, S_2pc, S_2wc, S_3pc, S_3wc,
2	Q_0wp, Q_1wp, Q_2wp, Q_3wp,
3	S_0Lpc, S_0Lwc, S_0Rpc, S_0Rwc, S_0Upc, S_0Uwc, S_1Lpc, S_1Lwc, S_1Rpc, S_1Rwc, S_1Upc, S_1Uwc,
4	S_0wcS_1wc, S_0cS_1w, S_0wS_1c, S_0cS_1c, S_0wcQ_0wp, S_0cQ_0wp, S_0wcQ_0p, S_0cQ_0p, S_1wcQ_0wp, S_1cQ_0wp, S_1wcQ_0p, S_1cQ_0p,
5	S_0wcS_1cQ_0p, S_0cS_1wcQ_0p, S_0cS_1cQ_0wp, S_0cS_1cQ_0p, S_0pS_1pQ_0p, S_0wcQ_0pQ_1p, S_0cQ_0wpQ_1p, S_0cQ_0pQ_1wp, S_0cQ_0pQ_1p, S_0pQ_0pQ_1p, S_0wcS_1cS_2c, S_0cS_1wcS_2c, S_0cS_1cS_2wc, S_0cS_1cS_2, S_0pS_1pS_2,
6	S_0cS_0H cS_0Lc, S_0cS_0HcS_0Rc, S_1cS_1HcS_1Rc, S_0cS_0RcQ_0p, S_0cS_0RcQ_0w, S_0cS_0LcS_1c, S_0cS_0LcS_1w, S_0cS_1cS_1Rc, S_0wS_1cS_1Rc,

- RevInc
  ↑ + B_1c and B_1cS_0c, where B_1 is the bottom most node in the right periphery.

Measures of incrementality:

- **Connectedness**
  the average number of nodes in the stack before shifting
- **Waiting time**
  the number of nodes that need to be shifted to the stack before a dependency between any two nodes in the stack is resolved

Table 1: Connectedness and waiting time.

Algorithm	Connectedness	Waiting Time
NonInc	4.62	2.98
RevInc	**2.15**	**0.69**

Table 2: Performance on the development data[1].

Algorithm	UP	UR	UF	LP	LR	LF	Cat Acc.
NonInc (beam=1)	**92.57**	82.60	87.30	**85.12**	75.96	80.28	91.10
RevInc (beam=1)	91.62	**85.94**	**88.69**	83.42	**78.25**	**80.75**	90.87
NonInc (beam=16)	92.71	89.66	91.16	85.78	82.96	84.35	92.51
Z&C (beam=16)	-	-	-	87.15	82.95	85.00	92.77

- NonInc gets higher precision because it can use more context while making a decision.
- RevInc achieves higher recall because information on nodes is available even after they are reduced.

[1]'U' stands for unlabaled and 'L' stands for labeles. 'P', 'R' and 'F' are precision, recall and F-score respectively.

RESULTS AND ANALYSIS

Table 3: Label-wise F-score of RevInc and NonInc parsers.

Category	RevInc	NonInc
(NP\NP)/**NP**	81.36	**83.21**
(NP**NP**)/NP	78.66	**82.94**
((S\NP)\(S\NP))/**NP**	65.09	**66.98**
((S\NP)**(S\NP)**)/NP	62.69	**65.89**
(S[dcl]\NP)/**NP**	**78.96**	78.29
(S[dcl]**NP**)/NP	**76.71**	75.22
(S\NP)**(S\NP)**	**80.49**	76.90

- NonInc performs better in labels corresponding to PP due to the availability of more context.
- RevInc has advantage in the case of verbal arguments and verbal modifiers as the effect of “reveal” actions.

Parsing speed:

- NonInc parses 110 sentences/sec.
- RevInc parses 125 sentences/sec.

Significant amount of parsing time is spent on the feature extraction step. But in RevInc, usually only two nodes have their feature extracted because connectedness = 2.15, while all four nodes have to be processed in NonInc (connectedness = 4.62).

Complex actions, `LRev` and `RRev` are rarely used (5%).

Table 4: Performance on the test data.

Algorithm	UP	UR	UF	LP	LR	LF	Cat Acc.
NonInc (beam=1)	**92.45**	82.16	87.00	**85.59**	76.06	80.55	91.39
RevInc (beam=1)	91.83	**86.35**	**89.00**	84.02	**79.00**	**81.43**	91.17
NonInc (beam=16)	92.68	89.57	91.10	86.20	83.32	84.74	92.70
Z&C (beam=16)	-	-	-	87.43	83.61	85.48	93.12
Hassan et al. 09	-	-	86.31	-	-	-	-

F-scores are improved compared to NonInc in both unlabaled and labeled cases.

- Use information about lexical category probabilities (Auli and Lopez (2011)).
- Explore the limited use of a beam to handle lexical ambiguity.
- Use a dynamic oracle strategy (Xu et al. (2014)).
- Apply the method to SMT and ASR.

CONCLUSION

- They designed and implemented an incremental CCG parser by introducing a technique called revealing.
- The parser got high scores in two measures of incrementality: connectedness and waiting time.
- It performs better in parsing in the view of recall and F-score. (labeled: +0.88%, unlabeld: +2.0%)
- It parses sentences faster.

Ambati, R. B., Deoskar, T., Johnson, M., and Steedman, M. (2015). An incremental algorithm for transition-based CCG parsing. In *Proceedings of the 2015 Conference of NAACL*, pages 53–63.

Auli, M. and Lopez, A. (2011). A comparison of loopy belief propagation and dual decomposition for integrated CCG supertagging and parsing. In *Proceedings of the 49th Annual Meeting of ACL*, pages 470–480.

Pareschi, R. and Steedman, M. (1987). A lazy way to chart-parse with categorial grammars. In *Proceedings of the 25th Annual Meeting of ACL*.

Xu, W., Clark, S., and Zhang, Y. (2014). Shift-reduce CCG parsing with a dependency model. In *Proceedings of the 52nd Annual Meeting of ACL*, pages 218–227.

Zhang, Y. and Clark, S. (2011). Shift-reduce CCG parsing. In *Proceedings of the 49th Annual Meeting of ACL*, pages 683–692.