Linear Time Constituency Parsing with RNNs and Dynamic Programming

Juneki Hong¹ Liang Huang¹,²

¹ Oregon State University
² Baidu Research Silicon Valley AI Lab
A Brief History of Span Parsing

- Cross+Huang 2016 Introduced Span Parsing
  - But with greedy decoding.
- Stern et al. 2017 had Span Parsing with Global Search
  - But was too slow: $O(n^3)$
- Can we get something in between?
  - Something that is both fast and accurate?

![Graph showing accuracy vs. speed with points for different works: Cross + Huang 2016, Stern et al. 2017, Kitaev + Klein 2018, Joshi et al. 2018, and Our Work at New at ACL 2018!].
Both Fast and Accurate!

Baseline Chart Parser (Stern et al. 2017a) 91.79

Our Linear Time Parser 91.97
In this talk, we will discuss:

- Linear Time Constituency Parsing using dynamic programming.
- Going slower in order to go faster: $O(n^3) \rightarrow O(n^4) \rightarrow O(n)$.
- Cube Pruning to speed up Incremental Parsing with Dynamic Programming.
- An improved loss function for Loss-Augmented Decoding.
Span Parsing

- Span differences are taken from an encoder (in our case: a bi-LSTM)

Cross + Huang 2016  Stern et al. 2017  Wang + Chang 2016
Span Parsing

- Span differences are taken from an encoder (in our case: a bi-LSTM)
- A span is scored and labeled by a feed-forward network.

\[ s(i, j, X) \]

\[ (f_j - f_i, b_i - b_j) \]

Cross + Huang 2016  Stern et al. 2017  Wang + Chang 2016
Span Parsing

- Span differences are taken from an encoder (in our case: a bi-LSTM)
- A span is scored and labeled by a feed-forward network.
- The score of a tree is the sum of all the labeled span scores

\[ s_{\text{tree}}(t) = \sum_{(i, j, X) \in t} s(i, j, X) \]

Cross + Huang 2016  Stern et al. 2017  Wang + Chang 2016
### Incremental Span Parsing Example

<table>
<thead>
<tr>
<th>Action</th>
<th>Label</th>
<th>Stack</th>
</tr>
</thead>
</table>

- **0**: Eat
  - **1**: ice
  - **2**: cream
  - **3**: after
  - **4**: lunch

Cross + Huang 2016
Incremental Span Parsing Example

Cross + Huang 2016
Incremental Span Parsing Example

<table>
<thead>
<tr>
<th>Action</th>
<th>Label</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shift</td>
<td>ø</td>
</tr>
<tr>
<td>2</td>
<td>Shift</td>
<td>ø</td>
</tr>
</tbody>
</table>
Incremental Span Parsing Example

<table>
<thead>
<tr>
<th>Action</th>
<th>Label</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shift</td>
<td>ø</td>
</tr>
<tr>
<td>2</td>
<td>Shift</td>
<td>(0, 1, ø)</td>
</tr>
<tr>
<td>3</td>
<td>Shift</td>
<td>(0, 1, ø) (1, 2, ø)</td>
</tr>
</tbody>
</table>

Cross + Huang 2016
# Incremental Span Parsing Example

Below is an example of an incremental span parsing process for the sentence: "Eat ice cream after lunch." The diagram and table outline the steps involved in parsing this sentence.

### Diagram:
- **VB**: Eat
- **NP**: ice cream
- **IN**: after
- **NN**: lunch

### Table:

<table>
<thead>
<tr>
<th>Action</th>
<th>Label</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shift</td>
<td>ø</td>
</tr>
<tr>
<td>2</td>
<td>Shift</td>
<td>ø</td>
</tr>
<tr>
<td>3</td>
<td>Shift</td>
<td>ø</td>
</tr>
<tr>
<td>4</td>
<td>Reduce</td>
<td>NP</td>
</tr>
</tbody>
</table>

The stack evolves as follows:
- **1**: `(0, 1, ø)`
- **2**: `(0, 1, ø) (1, 2, ø)`
- **3**: `(0, 1, ø) (1, 2, ø) (2, 3, ø)`
- **4**: `(0, 1, ø) (1, 3, NP)`

The Cross + Huang 2016 algorithm is used for this parsing process.
Incremental Span Parsing Example

<table>
<thead>
<tr>
<th>Action</th>
<th>Label</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shift</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>Shift</td>
<td>Ø</td>
</tr>
<tr>
<td>3</td>
<td>Shift</td>
<td>Ø</td>
</tr>
<tr>
<td>4</td>
<td>Reduce</td>
<td>NP</td>
</tr>
<tr>
<td>5</td>
<td>Reduce</td>
<td>Ø</td>
</tr>
</tbody>
</table>

Cross + Huang 2016
Incremental Span Parsing Example

<table>
<thead>
<tr>
<th>Action</th>
<th>Label</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shift</td>
<td>ø</td>
</tr>
<tr>
<td>2</td>
<td>Shift</td>
<td>ø (0, 1, ø) (1, 2, ø)</td>
</tr>
<tr>
<td>3</td>
<td>Shift</td>
<td>ø (0, 1, ø) (1, 2, ø) (2, 3, ø)</td>
</tr>
<tr>
<td>4</td>
<td>Reduce</td>
<td>NP (0, 1, ø) (1, 3, NP)</td>
</tr>
<tr>
<td>5</td>
<td>Reduce</td>
<td>ø (0, 3, ø)</td>
</tr>
<tr>
<td>6</td>
<td>Shift</td>
<td>ø (0, 3, ø) (3, 4, ø)</td>
</tr>
</tbody>
</table>

Cross + Huang 2016
Incremental Span Parsing Example

<table>
<thead>
<tr>
<th>Action</th>
<th>Label</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shift</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>Shift</td>
<td>Ø</td>
</tr>
<tr>
<td>3</td>
<td>Shift</td>
<td>Ø</td>
</tr>
<tr>
<td>4</td>
<td>Reduce</td>
<td>NP</td>
</tr>
<tr>
<td>5</td>
<td>Reduce</td>
<td>Ø</td>
</tr>
<tr>
<td>6</td>
<td>Shift</td>
<td>Ø</td>
</tr>
<tr>
<td>7</td>
<td>Shift</td>
<td>NP</td>
</tr>
</tbody>
</table>

Cross + Huang 2016
Incremental Span Parsing Example

Cross + Huang 2016
**Incremental Span Parsing Example**

<table>
<thead>
<tr>
<th>Action</th>
<th>Label</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Shift</td>
<td>(\emptyset) ((0, 1, \emptyset))</td>
</tr>
<tr>
<td>2</td>
<td>Shift</td>
<td>(\emptyset) ((0, 1, \emptyset) (1, 2, \emptyset))</td>
</tr>
<tr>
<td>3</td>
<td>Shift</td>
<td>(\emptyset) ((0, 1, \emptyset) (1, 2, \emptyset) (2, 3, \emptyset))</td>
</tr>
<tr>
<td>4</td>
<td>Reduce</td>
<td>NP ((0, 1, \emptyset) (1, 3, NP))</td>
</tr>
<tr>
<td>5</td>
<td>Reduce</td>
<td>(\emptyset) ((0, 3, \emptyset))</td>
</tr>
<tr>
<td>6</td>
<td>Shift</td>
<td>(\emptyset) ((0, 3, \emptyset) (3, 4, \emptyset))</td>
</tr>
<tr>
<td>7</td>
<td>Shift</td>
<td>NP ((0, 3, \emptyset) (3, 4, \emptyset) (4, 5, NP))</td>
</tr>
<tr>
<td>8</td>
<td>Reduce</td>
<td>PP ((0, 3, \emptyset) (3, 5, PP))</td>
</tr>
<tr>
<td>9</td>
<td>Reduce</td>
<td>S-VP ((0, 5, S-VP))</td>
</tr>
</tbody>
</table>

Cross + Huang 2016
Using a Graph Structured Stack

- This parsing procedure requires a stack of spans.
- We can use a Graph Structured Stack
  - To keep track of the next span on the stack
- And only use the top span \((i, j)\) as our parsing state.
Graph Structured Stack Example

(Note: Spans are independently labeled)

(So we can worry about just the spans themselves)

Gold: Shift (0,1)
GSS Graph Structured Stack Example

Gold:

<table>
<thead>
<tr>
<th></th>
<th>Shift (0,1)</th>
<th>Shift (1,2)</th>
</tr>
</thead>
</table>

Left Pointers

Gold Parse
Graph Structured Stack Example

Gold: Shift (0,1) Shift (1,2) Shift (2, 3)
Graph Structured Stack Example

Gold:

- Shift (0,1)
- Shift (1,2)
- Shift (2,3)
- Reduce (1,3)
Graph Structured Stack Example

Gold:

<table>
<thead>
<tr>
<th>Shift</th>
<th>Shift</th>
<th>Shift</th>
<th>Reduce</th>
<th>Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>(1,2)</td>
<td>(2,3)</td>
<td>(1,3)</td>
<td>(0,3)</td>
</tr>
</tbody>
</table>
Graph Structured Stack Example

Gold Parse

Left Pointers

<table>
<thead>
<tr>
<th>Gold:</th>
<th>Shift (0, 1)</th>
<th>Shift (1, 2)</th>
<th>Shift (2, 3)</th>
<th>Reduce (1, 3)</th>
<th>Reduce (0, 3)</th>
<th>Shift (3, 4)</th>
</tr>
</thead>
</table>

Transactions:
- \( \epsilon \) → (0,1)
- Shift (0,1) → (1,2)
- Shift (1,2) → (2,3)
- Shift (2,3) → (3,4)
- Shift (3,4) → (4,5)
- Reduce (4,5) → (3,5)
- Shift (3,5) → (2,4)
- Reduce (2,4) → (1,4)
- Reduce (1,4) → (0,3)
- Shift (0,3) → (3,4)
Graph Structured Stack Example

Gold:

<table>
<thead>
<tr>
<th></th>
<th>Shift (0,1)</th>
<th>Shift (1,2)</th>
<th>Shift (2, 3)</th>
<th>Reduce (1, 3)</th>
<th>Reduce (0, 3)</th>
<th>Shift (3, 4)</th>
<th>Shift (4, 5)</th>
<th>Reduce (3, 5)</th>
<th>Reduce (0, 5)</th>
</tr>
</thead>
</table>

Shifting and reducing operations are illustrated in the graph, where nodes represent positions in the stack and edges indicate transitions between states.
Runtime Analysis: $O(n^4)$

Huang+Sagae 2010
Runtime Analysis: $O(n^4)$

#steps: $2n - 1 = O(n)$

Huang+Sagae 2010
Runtime Analysis: $O(n^4)$

#states per step: $O(n^2)$

#steps: $2n - 1 = O(n)$

Huang+Sagae 2010
Runtime Analysis: \(O(n^4)\)

\[ 2n - 1 = O(n) \]

\(O(n^3)\) states

\[ (i, j) \]

\#states per step: \(O(n^2)\)

\#steps: \(2n - 1 = O(n)\)

Huang+Sagae 2010
Runtime Analysis: $O(n^4)$

#left pointers per state: $O(n)$

#states per step: $O(n^2)$

#steps: $2n - 1 = O(n)$

Check out the paper for Deng’s Theorem:

$$\ell' = \ell - 2(j - i) + 1$$

$O(n^3)$ states

Huang+Sagae 2010
Runtime Analysis: $O(n^4)$

Check out the paper for Deng's Theorem:

$$\ell' = \ell - 2(j - i) + 1$$

#left pointers per state: $O(n)$

#states per step: $O(n^2)$

#steps: $2n - 1 = O(n)$

$O(n^3)$ states with $O(n)$ reduce actions: $O(n^4)$ runtime

Huang+Sagae 2010
Going slower to go faster

- Our Action-Synchronous algorithm has a slower runtime than CKY!
Going slower to go faster

- Our Action-Synchronous algorithm has a slower runtime than CKY!
- However, it also becomes straightforward to prune using beam search.

```
ε → sh → (0,1) → sh → (1,2) → sh → (2,3) → sh → (3,4) → sh → (4,5) → r → (3,5) → r → (2,5) → r → (1,5) → r → (0,5)
```

```
ε → sh → (0,1) → sh → (1,2) → sh → (2,3) → sh → (3,4) → sh → (4,5) → r → (3,5) → r → (2,5) → r → (1,5) → r → (0,5)
```

```
(0,2) → sh → (2,3) → r → (0,3) → sh → (3,4) → r → (0,4) → sh → (4,5)
```

```
(0,2) → sh → (2,3) → r → (0,3) → sh → (3,4) → r → (0,4) → sh → (4,5)
```
Going slower to go faster

- Our Action-Synchronous algorithm has a slower runtime than CKY!
- However, it also becomes straightforward to prune using beam search.
- So we can achieve a linear runtime in the end.
Now our runtime is $O(n)$.
But the $O(n)$ is hiding a constant.
But the $O(n)$ is hiding a constant.

$O(b)\, \text{left pointers per state}$

$O(nb^2)\, \text{runtime}$
We can apply cube-pruning to make $O(nb \log b)$
• We can apply cube-pruning to make $O(nb \log b)$

• By pushing all states and their left pointers into a heap

Chiang 2007
Huang+Chiang 2007
Cube-Pruning

- We can apply cube-pruning to make $O(nb \log b)$

- By pushing all states and their left pointers into a heap
- And popping the top $b$ unique subsequent states

Chiang 2007
Huang+Chiang 2007
Cube-Pruning

- We can apply cube-pruning to make $O(nb \log b)$
  - By pushing all states and their left pointers into a heap
  - And popping the top $b$ unique subsequent states
  - First time Cube-Pruning has been applied to Incremental Parsing

Chiang 2007
Huang+Chiang 2007
Runtime on PTB and Discourse

- Chart Parsing: $O(n^{2.26})$
- Beam 20 No Cube-Pruning: $O(n^{1.26})$
- Beam 20 Cube Pruned: $O(n^{1.08})$
- Beam 5 Cube Pruned: $O(n^{0.97})$
Training
Training

- Structured SVM approach:

- Goal: Score the gold tree higher than all others by a margin:

\[
\forall t, s(t^*) - s(t) \geq \Delta(t, t^*)
\]
Training

• Structured SVM approach:
  • Goal: Score the gold tree higher than all others by a margin:
    \[ \forall t, s(t^*) - s(t) \geq \Delta(t, t^*) \]

• Loss Augmented Decoding:
  • During Training: Return tree with highest augmented score:
    \[ \hat{t} = \arg \max_t \left( s(t) + \Delta(t, t^*) \right) \]
Training

- **Structured SVM approach:**
  - **Goal:** Score the gold tree higher than all others by a margin:
    \[
    \forall t, s(t^*) - s(t) \geq \Delta(t, t^*)
    \]

- **Loss Augmented Decoding:**
  - **During Training:** Return tree with highest augmented score:
    \[
    \hat{t} = \arg \max_t \left( s(t) + \Delta(t, t^*) \right)
    \]

- **Minimize Loss:**
  \[
  (s(\hat{t}) + \Delta(\hat{t}, t^*)) - s(t^*)
  \]
Before training:

- $t_1$: Eat ice cream after lunch
- $t_2$: Eat ice cream after lunch
- $t^*$: Eat ice cream after lunch

Score vs. Accuracy graph showing the transition from $t_1$ to $t_2$ to $t^*$.
Ideally after training:

\[ \Delta(t_1, t^*) \]

\[ \Delta(t_2, t^*) \]
Ideally after training:

*Y Axis not drawn to scale
Delta Margins

- Counts the incorrectly labeled spans in the tree.
- Happens to be decomposable, so can even be used to compare partial trees.

$$\Delta(t, t^*) = \sum_{(i,j,X) \in t} 1\left(X \neq t^*_{(i,j)}\right)$$
Cross-Span Loss

- We observe that the null label $\emptyset$ is used in two different ways:
We observe that the null label $\emptyset$ is used in two different ways:

- To facilitate ternary and n-ary branching trees.

$$t^*(i, j) = \emptyset$$
We observe that the null label $\emptyset$ is used in two different ways:

- To facilitate ternary and n-ary branching trees.
- As a default label for incorrect spans that violate other gold spans.
Cross-Span Loss

• We modify the loss to account for incorrect spans in the tree.

\[ \Delta(t, t^*) = \sum_{(i, j, X) \in t} 1 \left( X \neq t^*_{(i,j)} \right) \]
We modify the loss to account for incorrect spans in the tree.

\[ \Delta(t, t^*) = \sum_{(i, j, X) \in t} 1 \left( X \neq t^*_{(i, j)} \lor \text{cross}(i, j, t^*) \right) \]
Cross-Span Loss

- We modify the loss to account for incorrect spans in the tree.

\[
cross(i, j, t^*)
\]

- Indicates whether \((i, j)\) is crossing a span in the gold tree

\[
\Delta(t, t^*) = \sum_{(i, j, X) \in t} 1 \left( X \neq t^*_{(i,j)} \lor \text{cross}(i, j, t^*) \right)
\]
Cross-Span Loss

- We modify the loss to account for incorrect spans in the tree.

\[
\text{cross}(i, j, t^*)
\]

- Indicates whether \((i, j)\) is crossing a span in the gold tree

\[
\Delta(t, t^*) = \sum_{(i, j, X) \in t} 1 \left( X \neq t^*_{(i, j)} \lor \text{cross}(i, j, t^*) \right)
\]

- Still decomposable over spans, so can be used to compare partial trees.
Max-Violation Updates

- Take the largest augmented loss value across all time steps.
- This is the Max-Violation, that we use to train.

Huang et al. 2012
### Experiments: PTB Test

<table>
<thead>
<tr>
<th>Model</th>
<th>Note</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stern et al. (2017a)</td>
<td>Baseline Method (Chart Parser)</td>
<td>91.79</td>
</tr>
<tr>
<td>Stern et al. (2017a) + cross-span</td>
<td>+our improved loss</td>
<td>91.81</td>
</tr>
<tr>
<td>Stern et al. (2017b)</td>
<td>Github Code</td>
<td>91.86</td>
</tr>
<tr>
<td>GSS Beam 15</td>
<td>Our Work</td>
<td>91.84</td>
</tr>
<tr>
<td>GSS Beam 20</td>
<td>Our Work</td>
<td>91.97</td>
</tr>
<tr>
<td>Model</td>
<td>Note</td>
<td>F1 Score</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>Durett + Klein 2015</td>
<td></td>
<td>91.1</td>
</tr>
<tr>
<td>Cross + Huang 2016</td>
<td>Original Span Parser</td>
<td>91.3</td>
</tr>
<tr>
<td>Liu + Zhang 2016</td>
<td></td>
<td>91.7</td>
</tr>
<tr>
<td>Dyer et al. 2016</td>
<td>Discriminative</td>
<td>91.7</td>
</tr>
<tr>
<td>Stern 2017a</td>
<td>Baseline Chart Parser</td>
<td>91.79</td>
</tr>
<tr>
<td>Stern 2017c</td>
<td>Separate Decoding</td>
<td>92.56</td>
</tr>
<tr>
<td>Our Work</td>
<td>Beam 20</td>
<td>91.97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Note</th>
<th>F1 Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vinyals et al. 2015</td>
<td>Ensemble</td>
<td>90.5</td>
</tr>
<tr>
<td>Dyer et al. 2016</td>
<td>Generative Reranking</td>
<td>93.3</td>
</tr>
<tr>
<td>Choe + Charniak 2016</td>
<td>Reranking</td>
<td>93.8</td>
</tr>
<tr>
<td>Fried et al. 2017</td>
<td>Ensemble Reranking</td>
<td>94.25</td>
</tr>
</tbody>
</table>

Comparison to other parsers
PTB only, Single Model, End-to-End

Reranking, Ensemble, Extra Data
Conclusions:

- Linear Time Span-Based Constituency Parsing with Dynamic Programming.
- Cube-Pruning to speedup Incremental Parsing with Dynamic Programming.
- Cross-Span Loss extension for improving Loss-Augmented Decoding.
- Result: Faster and more accurate than cubic-time Chart Parsing.
Caveats:

- 2nd highest accuracy for single-model end-to-end systems trained on PTB only.
- Stern et al. 2017c is more accurate, but with separate decoding, and is much slower.
- After this ACL, definitely no longer true. (e.g. Joshi et al. 2018, Kitaev+Klein 2018)
- But both are Span-Based Parsers and can be linearized in the same way!
Questions?

Thank You