Fast Accurate Dependency Parsing with a Hash Kernel

Dr. Bernd Bohnet
Universitat Pompeu Fabra
Example of a Dependency Tree

- Dependency tree describes the syntax in terms of binary relations between the words of a sentence
- The edges are labeled with the syntactic function
- Dependency parsing is the task to build a (labeled) tree over all words of a sentence
Overview

• Architecture of the Dependency Parser
• Parsing Technique
• Training Technique
• Time Analysis
• Hash Kernel
• Conclusion
Parsing Algorithm: Maximum Spanning Tree (Eisner,96)

- Dynamic Programming Approach:
  - build bottom-up spans
  - Combine adjacent words into spans
  - Combine spans that overlap in one word

The luxury auto maker sold 12,214 cars
Parsing Algorithm: Maximum Spanning Tree (Eisner)

- Combine adjacent words into spans

(The luxury)  (luxury auto)  (auto maker)  (maker sold), ...
Parsing Algorithm: Maximum Spanning Tree (Eisner)

- Combine adjacent spans that overlap in one word

The luxury auto maker sold 12,214 cars

(luxury auto) + (auto maker) → (luxury auto maker)
(The luxury) + (luxury auto maker) → (The luxury auto maker)
Used Syntactic Parsing Techniques

• Decoder: Maximum Spanning Tree
  – 2\textsuperscript{nd} Order (Carreras, 07); disadvantage $O(n^4)$
• Training Technique: Support Vector Machine
  – Perceptron Algorithm
  – Passive-Aggressive-Learning
  – Online learning
• Non-Projective Parsing for English, German, etc.
  – Non-Projective Approximation Algorithm (McDonald, 06)
Long Parsing Times

- High demands on Resources
- Training still takes several days
- Parsing time per sentence up to a minute

Parsing time is very important for many applications such as:
- Dialog systems (have a time slot for parsing a sentence of some 100 milliseconds)
- Machine translation systems (have only milliseconds since up to 10000 alternatives have to be processed)
- ...
Improving the Parsing Speed

• Two possibilities to improve the parsing speed
  • Without accuracy loss: faster algorithms, parallel algorithms, eliminating features, ...
  • With accuracy loss: less computationally expensive algorithm, less features, ...

⇒ In order to create a faster algorithm without any accuracy loss, we analyzed the time usage of our system
## Analysis of the Training Time

### Algorithm 1: Training – base line algorithm

\[
\tau = \{(x_i, y_i)\}_{i=1}^{I} // \text{Training data} \\
\overline{w} = 0, \overline{v} = 0 \\
\gamma = E \times I // \text{passive-aggressive update weight} \\
\text{for } i = 1 \text{ to } I \\
\quad t_{s+e}^{s} : \text{extract-and-store-features}(x_i); t_{s+e}^{e} ; \\
\text{for } n = 1 \text{ to } E // \text{iteration over the training epochs} \\
\quad \text{for } i = 1 \text{ to } I // \text{iteration over the training examples} \\
\quad \quad k \leftarrow (n - 1) \times I + i \\
\quad \quad \gamma = E \times I - k + 2 // \text{passive-aggressive weight} \\
\quad \quad t_{s,k}^{s}, A = \text{read-features-and-calc-arrays}(i, \overline{w}); t_{r,k}^{e} \\
\quad \quad t_{p,k}^{s}, y_p = \text{predicte-projective-parse-tree}(A); t_{p,k}^{e} \\
\quad \quad t_{a,k}^{s}, y_a = \text{non-projective-approx.} (y_p, A); t_{a,k}^{e} \\
\quad \quad \text{update } \overline{w}, \overline{v} \text{ according to } \Delta(y_p, y_i) \text{ and } \gamma \\
w = v/(E \times I) // \text{average}
\]

<table>
<thead>
<tr>
<th></th>
<th>English</th>
<th>German</th>
<th>Chinese</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training time ms/instance:</td>
<td>202 ms</td>
<td>166 ms</td>
<td>1016 ms</td>
</tr>
<tr>
<td>Training time in hours:</td>
<td>38 h</td>
<td>17 h</td>
<td>93 h</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>English</th>
<th>German</th>
<th>Chinese</th>
</tr>
</thead>
<tbody>
<tr>
<td>1509 ms</td>
<td>945 ms</td>
<td>4582 ms</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>140</td>
<td>748</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>8</td>
<td>95</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>18</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>202 ms</td>
<td>166 ms</td>
<td>1016 ms</td>
<td></td>
</tr>
<tr>
<td>38 h</td>
<td>17 h</td>
<td>93 h</td>
<td></td>
</tr>
</tbody>
</table>
The feature extraction and calculation of the weight arrays takes about 100 times longer than by a projective parsing algorithm!

Usually, the complexity of the parsing algorithm is blamed for the high time consumption, but it is not the major factor due to our analysis.
What causes the high time consumption?

The creation of features?

We map the attributes to their minimal description length

e.g., Feature-Patter: F-type + Label + P-pos + D-form + DIST(P,D)

\[ \log_2(72 \text{ labels}) = 6.2 \Rightarrow 7 \text{ bit}; \ 54 \text{ pos} \Rightarrow 6 \text{ bit}; \text{ form:} 16 \text{ bit}; \text{ dist:} 4 \text{ bit}, \]

\[
F5 \quad \text{NMOD} \quad \text{NN} \quad \text{red} \quad 1 \\
0000101 \ 0000011 \ 000101 \ 0001000100100100 \ 000100..000 \\
1234567 \ 1234567 \ 123456 \ 1234567890123456 \ 1234 \\
1 \quad 8 \quad 15 \quad 21 \quad \ldots \quad 37 \quad \ldots \quad 64
\]

We remove everything else \( \Rightarrow \) 88 ms/instance

No, the extraction is fast.

What does consume the rest of the time (1166ms)?
The Feature-Index Mapping

- Further analysis shows that the feature-index mapping uses about 90% of the parsing time!
- The Feature-Index Mapping has two tasks:
  - Filter out features that never occur in the training set
  - Map the features to the index of the weight in the weight vector
- There are 10 !! times more “negative features” (= features which never occur in the training set)
Simple Bottleneck Causes a Lot of Trouble

⇒ It is a bad idea to use a hash table to filter out negative features
⇒ Hash tables are designed to find values and not to filter out
⇒ Large hash tables cause a lot of random access to the main memory which is the worst in modern CPUs
⇒ Because of this bottleneck, it is hard to gain an improvement with parallel algorithms since feature extraction is memory-width-bound
Solution: Hash Kernel

- We use a hash function \( h: J \rightarrow \{1,..,n\} \) to index the features \( \varphi \)
- \( x \) is a sentence and \( y \) its parse tree
- \( \varphi(x,y) \) is a numeric feature representation indexed by \( j \)
- \( \varphi_j(x,y) = \varphi'_k(x,y) \), where \( h(j)=k \)
- The process of parsing a sentence \( x_i \) is to find a parse tree \( y_p \) that maximizes a scoring function \( F(x_i, y_p) \)
- The learning problem is to fit the function such that the errors of the predicted parse tree \( y \) are as low as possible
- The scoring function of the Hash Kernel is \( F(x,y) = w * \varphi'_k(x,y) \), where \( w \) is the weight vector and \( n = |w| \)
Pros and Cons

+ We save the memory for the feature-index mapping
+ We save a lot of time for the access of the hash table
+/‐ We use negative features as weights too
    ⇒ accuracy improvement because of additional (negative) features

- We get hash misses for the access of the values of the weight vector
    ⇒ two or more features might share a value in the weight vector
- We need larger weight vectors to avoid the misses
### Algorithm: Parsing

\[
\tau = \{x_i\}_{i=1}^I \quad // \text{Test set} \\
\overline{w} = \text{read-weight-vector} \\
\text{for } i = 1 \text{ to } I \quad // \text{iteration over the test set} \\
\quad A = \text{extr.-features-and-calc-weight-arrays}(x_i, \overline{w}) \\
\quad y_p = \text{predicte-projective-parse-tree}(A) \\
\quad y_\alpha = \text{non-projective-approx.}(y_p, A)
\]

<table>
<thead>
<tr>
<th>Language</th>
<th>English</th>
<th>German</th>
<th>Chinese</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>319</td>
<td>109</td>
<td>1555</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>6</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>11</td>
<td>37</td>
</tr>
</tbody>
</table>

Parsing time in ms/sentence: 354 ms 126 ms 1617 ms

⇒ The parser is about 3.5 times faster in average for English (1256 => 354) and German, and about 2.2 for Chinese
# Labeled Attachment Scores of the Hash Kernel

<table>
<thead>
<tr>
<th>Language</th>
<th>Catalan</th>
<th>Chinese</th>
<th>Czech</th>
<th>English</th>
<th>German</th>
<th>Japanese</th>
<th>Spanish</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top CoNLL</td>
<td>87.86</td>
<td>79.19</td>
<td>80.38</td>
<td>89.88</td>
<td>87.48</td>
<td>92.57</td>
<td>87.64</td>
</tr>
<tr>
<td>Base Line</td>
<td>86.35</td>
<td>76.51</td>
<td>80.11</td>
<td>89.88</td>
<td>87.48</td>
<td>92.21</td>
<td>87.19</td>
</tr>
<tr>
<td>Hash Kernel</td>
<td>87.45</td>
<td>76.99</td>
<td>80.96</td>
<td>90.33</td>
<td>88.06</td>
<td>92.47</td>
<td>88.13</td>
</tr>
<tr>
<td>2 (+1)</td>
<td>2</td>
<td>1 (+1)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1 (+1)</td>
<td></td>
</tr>
</tbody>
</table>

The scores represent the attachment accuracy for different languages, with the Hash Kernel approach showing improvements over the Base Line in several languages.
LAS vs. Weight Vector Size

- Base line parser uses 8.7 Million features
- A hash function might map different indices to the same hash value
  ⇒ Hash misses / collisions of weights
- Larger weight vectors avoid collisions

![Graph showing LAS vs. Training time (hours) with different weight vector sizes.]
Higher Accuracy with Less Training Data

• Overproportional accuracy gain for training with less data
  ⇒ higher accuracy for languages with less training data, e.g. Spanish
Parallel Training and Parsing Algorithm

<table>
<thead>
<tr>
<th>Cores</th>
<th>$t_e$</th>
<th>$t_p$</th>
<th>$t_a$</th>
<th>rest</th>
<th>total</th>
<th>pars.</th>
<th>train.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>379</td>
<td>21.3</td>
<td>18.2</td>
<td>1.5</td>
<td>420</td>
<td>354</td>
<td>45.8h</td>
</tr>
<tr>
<td>2</td>
<td>196</td>
<td>11.7</td>
<td>9.2</td>
<td>2.1</td>
<td>219</td>
<td>187</td>
<td>23.9h</td>
</tr>
<tr>
<td>3</td>
<td>138</td>
<td>8.9</td>
<td>6.5</td>
<td>1.6</td>
<td>155</td>
<td>126</td>
<td>16.6h</td>
</tr>
<tr>
<td>4</td>
<td>106</td>
<td>8.2</td>
<td>5.2</td>
<td>1.6</td>
<td>121</td>
<td>105</td>
<td>13.2h</td>
</tr>
<tr>
<td>4+4h</td>
<td>73.3</td>
<td>8.8</td>
<td>4.8</td>
<td>1.3</td>
<td>88.2</td>
<td>77</td>
<td>9.6h</td>
</tr>
</tbody>
</table>

- The parallel algorithm is about 1.9 faster on two cores and 3.4 times faster on 4 cores
- With hyper threading it is 4.6 times faster
Hey

... look at your training technique
  does it use a lot of features?
  would you like to execute it faster?
  does it filter out negative features?
    \[ \Rightarrow \text{possibility to improve your results!} \]

Consider a hash version of your training technique!
Conclusion and Further Work

- We developed a dependency parser which has very high accuracy
- The Hash Kernel provides 3.5 times better parsing time
- The parallel parser with the Hash Kernel is 16 times faster
- We get an overproportional accuracy gain for languages with less training data
- We are convinced that the Hash Kernel is applicable also for transition based dependency parsing, phrase based dependency parsing and many other NLP applications