On the internals of disco-dop
How to implement a state-of-the-art LCFRS parser

Kilian Gebhardt

Grundlagen der Programmierung, Fakultät Informatik, TU Dresden

November 16, 2018
Motivation

- LCFRS parsing is hard ($O(n^{m*k})$ where $n$, $m$, and $k$ are sentence length, maximum numbers of nonterminals in a rule, and the fanout of the grammar, respectively.)
- Exact inference with real world LCFRS might feasible up to length 30 (see Angelov and Ljunglöf 2014)?
- We want to parse longer sentences and short sentences faster!
Motivation

- LCFRS parsing is hard ($O(n^{m*k})$ where $n$, $m$, and $k$ are sentence length, maximum numbers of nonterminals in a rule, and the fanout of the grammar, respectively.)
- Exact inference with real world LCFRS might feasible up to length 30 (see Angelov and Ljunglöf 2014)?
Motivation

- LCFRS parsing is hard ($O(n^{m*k})$ where $n$, $m$, and $k$ are sentence length, maximum numbers of nonterminals in a rule, and the fanout of the grammar, respectively.)
- Exact inference with real world LCFRS might feasible up to length 30 (see Angelov and Ljunglöf 2014)?
- We want to parse longer sentences and short sentences faster!
Parsing framework developed by Andreas van Cranenburgh (cf. Cranenburgh, Scha, and Bod 2016)
disco-dop

- Parsing framework developed by Andreas van Cranenburgh (cf. Cranenburgh, Scha, and Bod 2016)
- Uses discontinuous data-oriented model (discontinuous tree-substitution grammar) at its core.
Parsing framework developed by Andreas van Cranenburgh (cf. Cranenburgh, Scha, and Bod 2016)

Uses discontinuous data-oriented model (discontinuous tree-substitution grammar) at its core.

Employs a coarse-to-fine pipeline for parsing:

1. PCFG stage
2. LCFRS stage
3. DOP stage
The coarse-to-fine pipeline (grammars)

▶ The DOP model is equivalent to marginalizing over a latently annotated LCFRS (fine LCFRS) (see Goodman 2003 for continuous case).

1See unknownword6 and unknownword4 in https://github.com/andreasvc/disco-dop/blob/master/discodop/lexicon.py
The coarse-to-fine pipeline (grammars)

- The DOP model is equivalent to marginalizing over a latently annotated LCFRS (fine LCFRS) (see Goodman 2003 for continuous case).
- The original treebank $t_1$ is binarized/Markovized ($= t_2$) and a coarse prob. LCFRS is induced. (Grammar is binarized, simple, ordered, may contain chain rules)

---

$^1$See unknownword6 and unknownword4 in https://github.com/andreasvc/disco-dop/blob/master/discodop/lexicon.py
The coarse-to-fine pipeline (grammars)

- The DOP model is equivalent to marginalizing over a latently annotated LCFRS (fine LCFRS) (see Goodman 2003 for continuous case).
- The original treebank $t_1$ is binarized/Markovized ($= t_2$) and a coarse prob. LCFRS is induced. (Grammar is binarized, simple, ordered, may contain chain rules)
- Discontinuity in $t_2$ is resolved by splitting categories. After binarizing again, we obtain $t_3$ and induce a PCFG. (Grammar is binarized, simple, may contain chain rules.)

\[^1\text{See unknownword6 and unknownword4 in https://github.com/andreasvc/disco-dop/blob/master/discodop/lexicon.py}\]
The coarse-to-fine pipeline (grammars)

- The DOP model is equivalent to marginalizing over a latently annotated LCFRS (fine LCFRS) (see Goodman 2003 for continuous case).
- The original treebank $t_1$ is binarized/Markovized ($= t_2$) and a coarse prob. LCFRS is induced. (Grammar is binarized, simple, ordered, may contain chain rules)
- Discontinuity in $t_2$ is resolved by splitting categories. After binarizing again, we obtain $t_3$ and induce a PCFG. (Grammar is binarized, simple, may contain chain rules.)
- Some preprocessing is applied to lexical rules to handle unknown words. (Stanford signatures\(^1\))

---

\(^1\)See unknownword6 and unknownword4 in https://github.com/andreasvc/disco-dop/blob/master/discodop/lexicon.py
The coarse-to-fine pipeline (application)

- Parse with stage $s$ resulting in chart.
The coarse-to-fine pipeline (application)

- Parse with stage $s$ resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item $i$, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in $k$-best derivations (For PCFG $\rightarrow$ PLCFRS $k = 10,000$ is the default.)
- Next stage $s + 1$ prunes item $i$, if $\text{coarsify}(i)$ is not in whitelist.
- If unsuccessful, stop parsing and greedily/recursively select the largest possible items from chart.
The coarse-to-fine pipeline (application)

- Parse with stage $s$ resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item $i$, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in $k$-best derivations

Next stage $s + 1$ prunes item $i$, if $\text{coarsify}(i)$ is not in whitelist.
If unsuccessful, stop parsing and greedily/recursively select the largest possible items from chart as fallback strategy.
The coarse-to-fine pipeline (application)

- Parse with stage $s$ resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item $i$, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in $k$-best derivations
    (For PCFG $\rightarrow$ PLCFRS $k = 10^4$ is the default.)
- Next stage $s + 1$ prunes item $i$, if $\text{coarsify}(i)$ is not in whitelist.
- If unsuccessful, stop parsing and greedily/recursively select the largest possible items from chart as fallback strategy.
The coarse-to-fine pipeline (application)

- Parse with stage $s$ resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item $i$, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in $k$-best derivations

- Next stage $s + 1$ prunes item $i$, if $\text{coarsify}(i)$ is not in whitelist.
- If unsuccessful, stop parsing and greedily/recursively select the largest possible items from chart as fallback strategy.
The coarse-to-fine pipeline (application)

- Parse with stage $s$ resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item $i$, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in $k$-best derivations

(For PCFG $\rightarrow$ PLCFRS $k = 10,000$ is the default.)
The coarse-to-fine pipeline (application)

- Parse with stage $s$ resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item $i$, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in $k$-best derivations
  (For PCFG \(\rightarrow\) PLCFRS $k = 10,000$ is the default.)
- Next stage $s + 1$ prunes item $i$, if $\text{coarsify}(i)$ is not in whitelist.
The coarse-to-fine pipeline (application)

- Parse with stage $s$ resulting in chart.
- If successful, obtain a whitelist of items from chart:
  - $k = 0$: select all items that are part of successful derivation
  - $0 < k < 1$: select each item $i$, where $\alpha(i) \cdot \beta(i) \geq k$
  - $k \geq 1$: select all items that occur in $k$-best derivations

(For PCFG $\rightarrow$ PLCFRS $k = 10,000$ is the default.)
- Next stage $s + 1$ prunes item $i$, if coarsify$(i)$ is not in whitelist.
- If unsuccessful, stop parsing and greedily/recursively select the largest possible items from chart as fallback strategy.
Representation of LCFRS rules I

\[ A \rightarrow \langle x_1^{(1)} x_1^{(2)} x_2^{(1)}, x_2^{(2)} x_3^{(1)} x_4^{(1)} \rangle (B, C) \]
Representation of LCFRS rules I

\[ A \rightarrow \langle x_1^{(1)} x_1^{(2)} x_2^{(1)} x_2^{(2)} x_3^{(1)} x_4^{(1)} \rangle (B, C) \]

\[
\begin{array}{cccccc}
0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 \\
\end{array}
\]

\( i - 1 \) if \( x_j^{(i)} \)

1 if end of component
Representation of LCFRS rules I

\[ A \rightarrow \langle x_1^{(1)}, x_1^{(2)}, x_2^{(1)}, x_2^{(2)}, x_3^{(1)}, x_4^{(1)} \rangle (B, C) \]

\[
\begin{array}{cccc}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{array}
\]

\( i - 1 \) if \( x_j^{(i)} \)

1 if end of component

```c
struct ProbRule {
    // total: 32 bytes.
    double prob; // 8 bytes
    uint32_t lhs; // 4 bytes
    uint32_t rhs1; // 4 bytes
    uint32_t rhs2; // 4 bytes
    uint32_t args; // 4 bytes => 32 max vars per rule
    uint32_t lengths; // 4 bytes => same
    uint32_t no; // 4 bytes
};
```
e.g. \( \text{args} = 0b001010 \) and \( \text{lengths} = 0b100100 \).
2. \( A \rightarrow \langle x_1^{(1)}, x_2^{(1)} x_3^{(1)} \rangle (B) \) (same, with \( rhs2 = 0 \))
2. $A \rightarrow \langle x_1^{(1)}, x_2^{(1)}, x_3^{(1)} \rangle (B)$ (same, with $\text{rhs2} = 0$)
3. $A \rightarrow \langle \alpha \rangle$

stored via a map $\Sigma \rightarrow \text{vector<} \text{uint32_t}>$ and a 
\text{vector<} \text{LexicalRule}>$ where:

```cpp
struct LexicalRule {
    double prob;
    uint32_t lhs;
};
```
PCFG parsing I

bottom-up chart parsing (based on Bodenstab 2009's fast grammar loop)

1. populate_pos(chart, grammar, sentence)

2. for span in range(2, n+1):
   3.   for left in range(1, n + 1 - span):
   4.     right = left + span
   5.     for lhs in grammar.nonts:
   6.       for rule in grammar.rules[lhs]:
   7.         for mid in range(left + 1, right):
   8.           p1 = chart.getprob(left, mid, rule.rhs1)
   9.           p2 = chart.getprob(mid, right, rule.rhs2)
   10.          p_new = rule.prob + p1 + p2
   11.          if chart.updateprob(left, right, p_new):
   12.             chart.add_edge(...)
   13.          applyunary(left, right, chart, grammar)
PCFG parsing II

beam search (based on Zhang et al. 2010)

▶ local beam search by beam thresholding with parameters
\( \eta = 10^{-4}, \delta = 40 \)
PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$

- If $\text{span} \leq \delta$ and $p_{\text{new}} < \eta \cdot p_{\text{best4cell}}$, then prune.
beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$
- If $\text{span} \leq \delta$ and $p_{\text{new}} < \eta \cdot p_{\text{best4cell}}$, then prune.
- Only applied to binary rules.
beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters \( \eta = 10^{-4}, \delta = 40 \)
- If \( \text{span} \leq \delta \) and \( p_{\text{new}} < \eta \cdot p_{\text{best4cell}} \), then prune.
- Only applied to binary rules.

chart datastructures
- items are densely enumerated (cellidx(start, stop, nonterminal))
- saves log-probabilities in vector (indexed by cellidx)
- saves incoming edges for each item (chart.parseforest)
- best derivation (or k-best derivations) retrieved afterwards by recursively selecting best edge
PCFG parsing II

beam search (based on Zhang et al. 2010)
  ▶ local beam search by beam thresholding with parameters
    \( \eta = 10^{-4}, \delta = 40 \)
  ▶ If \( \text{span} \leq \delta \) and \( p_{\text{new}} < \eta \cdot p_{\text{best4cell}} \), then prune.
  ▶ Only applied to binary rules.

chart datastructures
  ▶ items are densely enumerated
    (cellid(start, stop, nonterminal))
beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters
  \( \eta = 10^{-4}, \delta = 40 \)

- If \( \text{span} \leq \delta \) and \( p_{\text{new}} < \eta \cdot p_{\text{best4cell}} \), then prune.

- Only applied to binary rules.

chart datastructures

- items are densely enumerated
  \( \text{cellidx(start, stop, nonterminal)} \)

- saves log-probabilities in vector (indexed by \text{cellidx})
beam search (based on Zhang et al. 2010)
- local beam search by beam thresholding with parameters $\eta = 10^{-4}$, $\delta = 40$
- If $\text{span} \leq \delta$ and $p_{\text{new}} < \eta \cdot p_{\text{best4cell}}$, then prune.
- Only applied to binary rules.

chart datastructures
- items are densely enumerated
  $(\text{cellidx}($start, stop, nonterminal$))$
- saves log-probabilities in vector (indexed by cellidx)
- saves incoming edges for each item (chart.parseforest)
PCFG parsing II

beam search (based on Zhang et al. 2010)

- local beam search by beam thresholding with parameters $\eta = 10^{-4}, \delta = 40$
- If $\text{span} \leq \delta$ and $p_{\text{new}} < \eta \cdot p_{\text{best4cell}}$, then prune.
- Only applied to binary rules.

chart datastructures

- items are densely enumerated
  ($\text{cellidx}(\text{start, stop, nonterminal})$)
- saves log-probabilities in vector (indexed by $\text{cellidx}$)
- saves incoming edges for each item ($\text{chart.parseforest}$)
- best derivation (or k-best derivations) retrieved afterwards by recursively selecting best edge
mid filter = auxiliary data structure (size: $4 \cdot |N| \cdot n$) with entries

minleft($A, j$) = $\max\{ i \mid [A, i, j] \in \text{chart} \}$
maxleft($A, j$) = $\min\{ i \mid [A, i, j] \in \text{chart} \}$
minright($A, j$) = $\min\{ j \mid [A, i, j] \in \text{chart} \}$
maxright($A, j$) = $\max\{ j \mid [A, i, j] \in \text{chart} \}$

replace “for mid in range(left + 1, right)” by

for mid in range(
    max(minright(B, left), maxleft(C, right)),
    min(maxright(B, left), minleft(C, right)))
LCFRS parsing

agenda driven LCFRS parser (with filter)
LCFRS parsing

agenda driven LCFRS parser (with filter)
1 populate_pos(...)

3 while not agenda.empty():
   4 item, prob = agenda.pop()
   5 chart.updateprob(item, prob)

7 if item == goal and not exhaustive:
   8 break

10 applyunaryrules(item, grammar, chart, agenda)
11 for rule in lbinary[item.nont]:
   12 for item2 in chart.items[rule.rhs2]:
       13 process(rule, item, item2, chart, agenda, whitelist)
14 for rule in rbinary[item.nont]:
   15 for item2 in chart.items[rule.rhs1]:
       16 process(rule, item2, item, chart, agenda, whitelist)
LCFRS parsing (heuristics)

- SX, SXlrgaps, etc. (Klein and Manning 2003 and Kallmeyer and Maier 2013)
- score += length * MAX_LOGPROB, i.e., smaller items are processed before larger items
LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length \( \leq 64 \))

```cpp
cdef cppclass SmallChartItem:
    uint32_t label
    uint64_t vec
```

- LCFRS Item (for sentences with length \( > 64 \))

```cpp
cdef cppclass FatChartItem:
    uint32_t label
    uint64_t vec[SLOTS]
```

- Combination of items based on algorithm in rparse's FastYFComposer

- Items are indexed in the order they are found. Index is stored in a B-Tree map. Items are ordered by label (primary) and vec (secondary).

- Probabilities are stored in a vector, indexed by item index.

- Incoming edges are stored in a `vector[vector[Edge]]`, indexed by item index.
LCFRS parse items

Use bitvector representation of spanned sentence positions:

▶ LCFRS Item (for sentences with length ≤ 64)
  cdef cppclass SmallChartItem:
    uint32_t label
    uint64_t vec

▶ LCFRS Item (for sentences with length > 64)
  cdef cppclass FatChartItem:
    uint32_t label
    uint64_t vec[SLOTS]
LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length \( \leq 64 \))
  
  ```c
cdef cppclass SmallChartItem:
    uint32_t label
    uint64_t vec
  ```

- LCFRS Item (for sentences with length \( > 64 \))
  
  ```c
cdef cppclass FatChartItem:
    uint32_t label
    uint64_t vec[SLOTS]
  ```

- Combination of items based on algorithm in rparse's FastYFComposer
LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length \( \leq 64 \))
  ```python
  cdef cppclass SmallChartItem:
    uint32_t label
    uint64_t vec
  ```

- LCFRS Item (for sentences with length \( > 64 \))
  ```python
  cdef cppclass FatChartItem:
    uint32_t label
    uint64_t vec[SLOTS]
  ```

- Combination of items based on algorithm in rparse’s FastYFComposer

- Items are indexed in the order they are found. Index is stored in a B-Tree map. Items are ordered by label (primary) and vec (secondary).
LCFRS parse items

Use bitvector representation of spanned sentence positions:

► LCFRS Item (for sentences with length \( \leq 64 \))
  cdef cppclass SmallChartItem:
    uint32_t label
    uint64_t vec

► LCFRS Item (for sentences with length > 64)
  cdef cppclass FatChartItem:
    uint32_t label
    uint64_t vec[SLOTS]

► Combination of items based on algorithm in rparse’s FastYFComposer

► Items are indexed in the order they are found. Index is stored in a B-Tree map. Items are ordered by label (primary) and vec (secondary).

► Probabilities are stored in a vector, indexed by item index.
LCFRS parse items

Use bitvector representation of spanned sentence positions:

- LCFRS Item (for sentences with length ≤ 64)
  ```cpp
cdef cppclass SmallChartItem:
    uint32_t label
    uint64_t vec
  ```
- LCFRS Item (for sentences with length > 64)
  ```cpp
cdef cppclass FatChartItem:
    uint32_t label
    uint64_t vec[SLOTS]
  ```
- Combination of items based on algorithm in rparse’s FastYFComposer

- Items are indexed in the order they are found. Index is stored in a B-Tree map. Items are ordered by label (primary) and vec (secondary).
- Probabilities are stored in a vector, indexed by item index.
- Incoming edges are stored in a vector[vector[Edge]], indexed by item index.
Agenda

▶ combines heap of (item, prob) and map: item → best probability
Agenda

- combines heap of (item, prob) and map: item → best probability
- while popping: check that best (item, prob) in heap satisfies map(item) = prob, otherwise pop next
Agenda

▶ combines heap of (item, prob) and map: item $\rightarrow$ best probability

▶ while popping: check that best (item, prob) in heap satisfies $\text{map}(\text{item}) = \text{prob}$, otherwise pop next

▶ on adding (item, prob): check that item $\notin \text{map}$ or $\text{map}(\text{item}) < \text{prob}$, otherwise discard


References III

