Searching for More Efficient Dynamic Programs

Tim Vieira, Ryan Cotterell, Jason Eisner
NLP LOVES DYNAMIC PROGRAMMING

It is the primary tool for devising efficient inference algorithms for numerous linguistic formalisms

- finite-state transduction (Mohri, 1997)
- dependency parsing (Eisner, 1996; Koo & Collins, 2010)
- context-free parsing (Stolcke, 1995; Goodman, 1999)
- context-sensitive parsing (Vijay-Shanker & Weir, 1989; Kuhlmann+, 2018)
- machine translation (Wu, 1996; Lopez, 2009)
**Speed-ups**

Designing an algorithm with the best possible running time is challenging.

- Bilexical dependency parsing: $O(n^5) \rightarrow O(n^4)$
- Split-head-factored dependency parsing: $O(n^5) \rightarrow O(n^3)$
- Linear index-grammar parsing: $O(n^7) \rightarrow O(n^6)$
- Lexicalized tree adjoining grammar parsing: $O(n^8) \rightarrow O(n^7)$
- Inversion transduction grammar: $O(n^7) \rightarrow O(n^6)$
- Tomita’s parsing algorithm: $O(G n^{p+1}) \rightarrow O(G n^3)$
- CKY parsing: $O(k^3 n^3) \rightarrow O(k^2 n^3 + k^3 n^2)$

We ask a simple question: **Can we automatically discover these faster algorithms?**
Our Approach

Cast program optimization as a graph search problem
- Nodes are program variations
- Edges are meaning-preserving transformations
- Costs of each node measures its running time
**Step 1: Dyna**

Represent algorithms in Dyna (Eisner et al. 2005), a domain-specific programming language for dynamic programming.

Example (CKY parsing):

\[
\begin{align*}
\beta(X,I,K) &= \gamma(X,Y,Z) \times \beta(Y,I,J) \times \beta(Z,J,K) \\
\beta(X,I,K) &= \gamma(X,Y) \times \text{word}(Y,I,K) \\
\text{total} &= \beta(“S”,0,N) \times \text{len}(N).
\end{align*}
\]
**Step 2: Runtime Bound From Code**

Under some technical conditions, the running time of a Dyna program is proportional to the number of ways to instantiate its rules.

For example,

\[
\beta(x, i, k) \leftarrow \gamma(x, y, z) \times \beta(y, i, j) \times \beta(z, j, k).
\]

\[O(k^3 n^3)\]

We use a simpler analysis \[O(v^6)\] where \(v = \max(n, k)\)

\[\rightarrow \text{degree} = 6\]

Why not run the code? WAY TOO SLOW!
Step 3: Program Transformations

Each program transform maps a Dyna program to another Dyna program with the same meaning and (hopefully) a better running time.

We turn to the playbook: Eisner & Blatz (2007)
\[ \beta(X, I, K) += \gamma(X, Y, Z) \times \beta(Y, I, J) \times \beta(Z, J, K). \]

**O(k^3 n^3) or O(v^6)**

\[ \beta(X, I, K) = \sum_{J, Y, Z} \gamma(X, Y, Z) \times \beta(Y, I, J) \times \beta(Z, J, K). \]

\[ \beta(X, I, K) = \sum_{J, Z} \left( \sum_{Y} \gamma(X, Y, Z) \times \beta(Y, I, J) \right) \times \beta(Z, J, K). \]

\[ = \text{tmp}(X, I, J, Z) \]

\[ \beta(X, I, K) += \text{tmp}(X, I, J, Z) \times \beta(Z, J, K). \]
\[ \text{tmp}(X, I, J, Z) += \gamma(X, Y, Z) \times \beta(Y, I, J). \]

**O(n^3 k^2 + n^2 k^3) or O(v^5)**
**Step 4: Search**

Feed these ingredients to a graph search algorithm

We need search because the best sequence of transformations cannot be found greedily. We experimented with **beam search** and **Monte Carlo tree search**.
## Experiments

### Unit tests

100%

<table>
<thead>
<tr>
<th>benchmark</th>
<th>% optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beam</td>
</tr>
<tr>
<td>bar-hillel</td>
<td>100</td>
</tr>
<tr>
<td>bilexical-labeled</td>
<td>90</td>
</tr>
<tr>
<td>bilexical-unlabeled</td>
<td>100</td>
</tr>
<tr>
<td>chain-10</td>
<td>100</td>
</tr>
<tr>
<td>chain-20</td>
<td>100</td>
</tr>
<tr>
<td>chain-expect</td>
<td>100</td>
</tr>
<tr>
<td>cky+grammar</td>
<td>40</td>
</tr>
<tr>
<td>cky3</td>
<td>90</td>
</tr>
<tr>
<td>cky4</td>
<td>90</td>
</tr>
<tr>
<td>edit</td>
<td>100</td>
</tr>
<tr>
<td>hmm</td>
<td>100</td>
</tr>
<tr>
<td>itg</td>
<td>90</td>
</tr>
<tr>
<td>path</td>
<td>100</td>
</tr>
<tr>
<td>semi-markov</td>
<td>100</td>
</tr>
<tr>
<td>split-head</td>
<td>90</td>
</tr>
</tbody>
</table>

### Stress tests
Summary

- Representing algorithms in a unified language allows us systematize the process of speeding them up.

- We showed how to optimize dynamic programs with graph search on a program transformation graph.

- We found that measuring running time efficiently was essential in order to explore enough of the search graph.
Thanks!
Fold Transform

\[ \beta(X, I, K) += \gamma(X, Y, Z) \times \beta(Y, I, K) \times \beta(Z, J, K). \]

creates an intermediate item \(\text{tmp}(X, I, J, Z)\)

\[ \beta(X, I, K) += \text{tmp}(X, I, J, Z) \times \beta(Z, J, K). \]

\[ \text{tmp}(X, I, J, Z) += \gamma(X, Y, Z) \times \beta(Y, I, K). \]

Generalizes the “hook trick”