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.

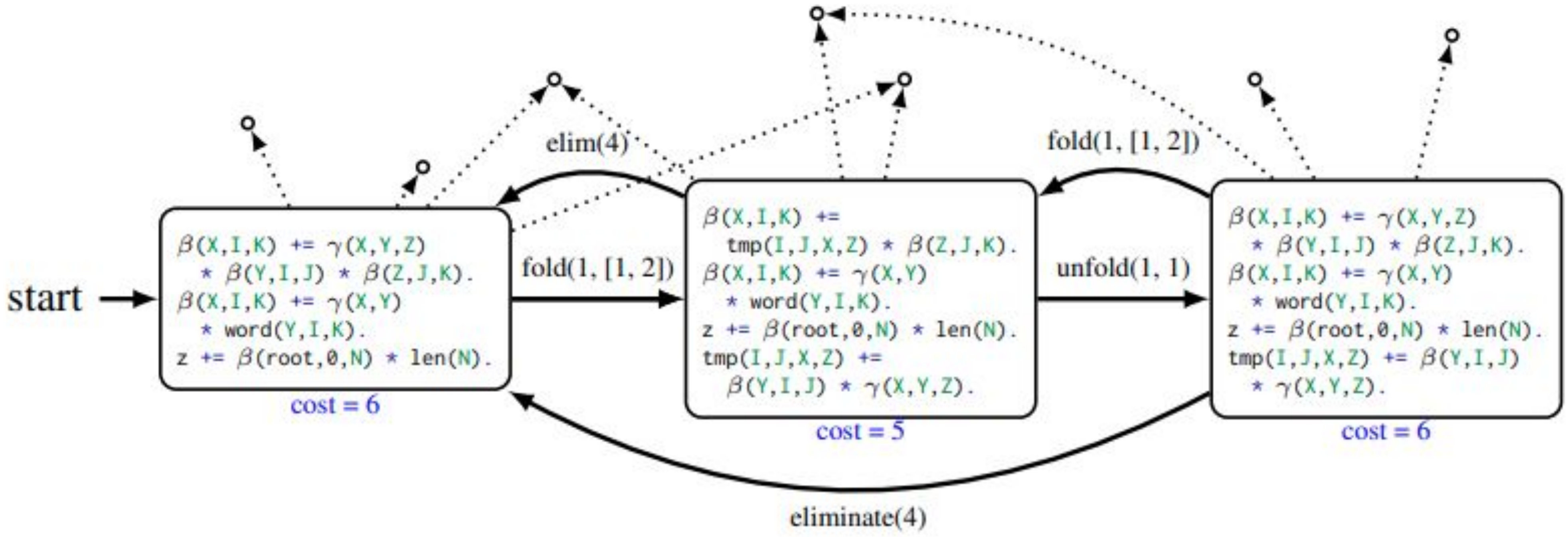
- Bilexical dependency parsing: $O(n^5) \rightarrow O(n^4)$
- Split-head-factored dependency parsing: $O(n^5) \rightarrow O(n^3)$
- Linear index-grammar parsing: $O(n^7) \rightarrow O(n^6)$
- Lexicalized tree adjoining grammar parsing: $O(n^8) \rightarrow O(n^7)$
- Inversion transduction grammar: $O(n^7) \rightarrow O(n^6)$
- 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

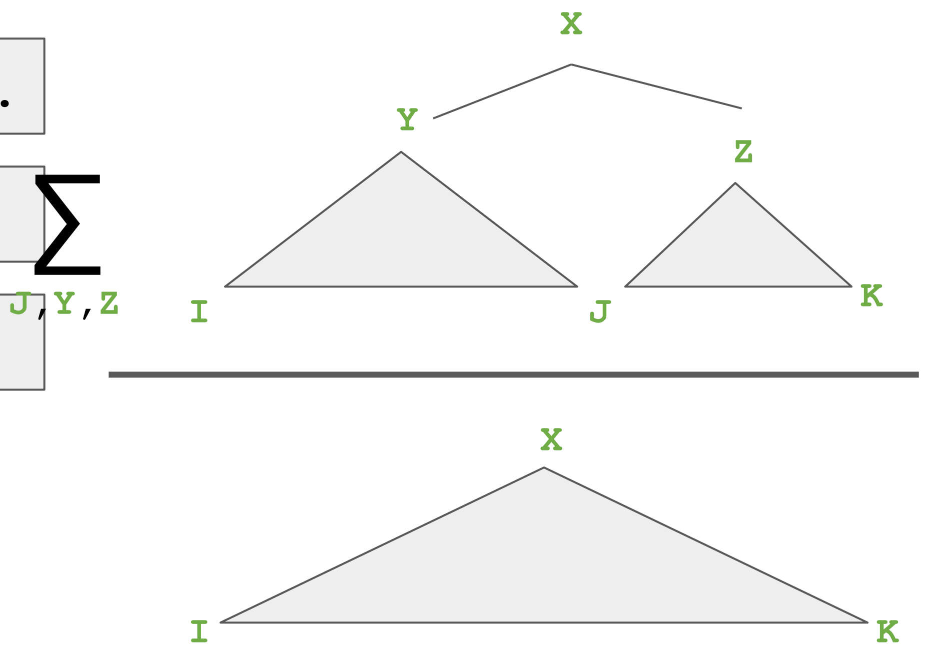


Step 1: Dyna

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

Example (CKY parsing):

```
β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).
β(X,I,K) += γ(X,Y) * word(Y,I,K).
total += β("S",0,N) * len(N).
```



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,

```
β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).
```

$O(k^3 n^3)$

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

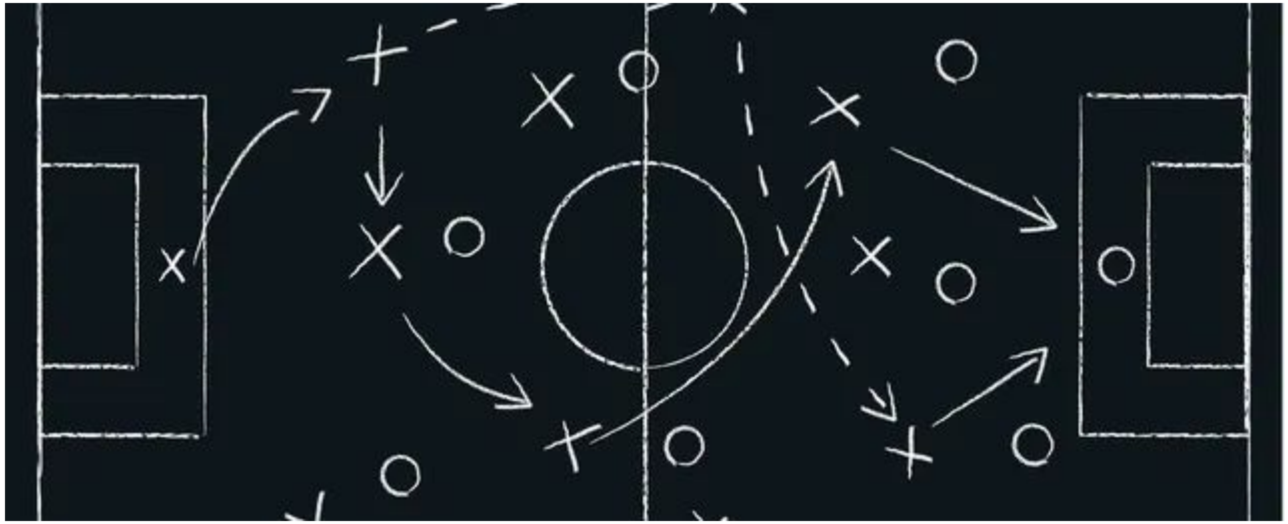
→ degree = 6

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)



Fold Transform

```
β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * β(Z,J,K).
```

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

$$\beta(X,I,K) = \sum_{J,Y,Z} \gamma(X,Y,Z) * \beta(Y,I,J) * \beta(Z,J,K).$$

$$\beta(X,I,K) = \sum_{J,Z} \Big(\underbrace{\sum_{Y} \gamma(X,Y,Z) * \beta(Y,I,J)}_{= \, tmp(X,I,J,Z)} \Big) * \beta(Z,J,K).$$

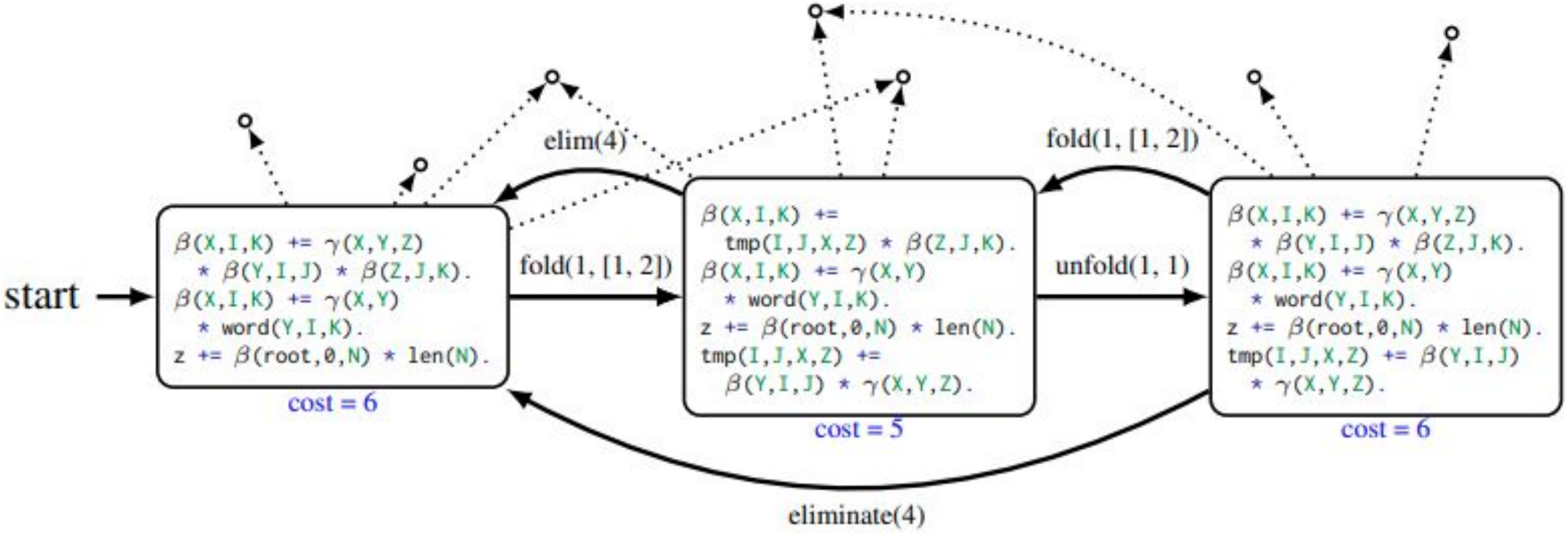
```
β(X,I,K) += tmp(X,I,J,Z) * β(Z,J,K).
tmp(X,I,J,Z) += γ(X,Y,Z) * β(Y,I,J).
```

$O(n^3 k^2 + n^2 k^3)$ or $O(v^5)$

Unfold Transform

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%

benchmark	% optimal beam	% optimal mcts
bar-hillel	100	100
bilexical-labeled	90	100
bilexical-unlabeled	100	90
chain-10	100	100
chain-20	100	100
chain-expect	100	100
cky+grammar	40	40
cky3	90	90
cky4	90	80
edit	100	90
hmm	100	100
itg	90	60
path	100	100
semi-markov	100	100
split-head	90	90

Stress tests

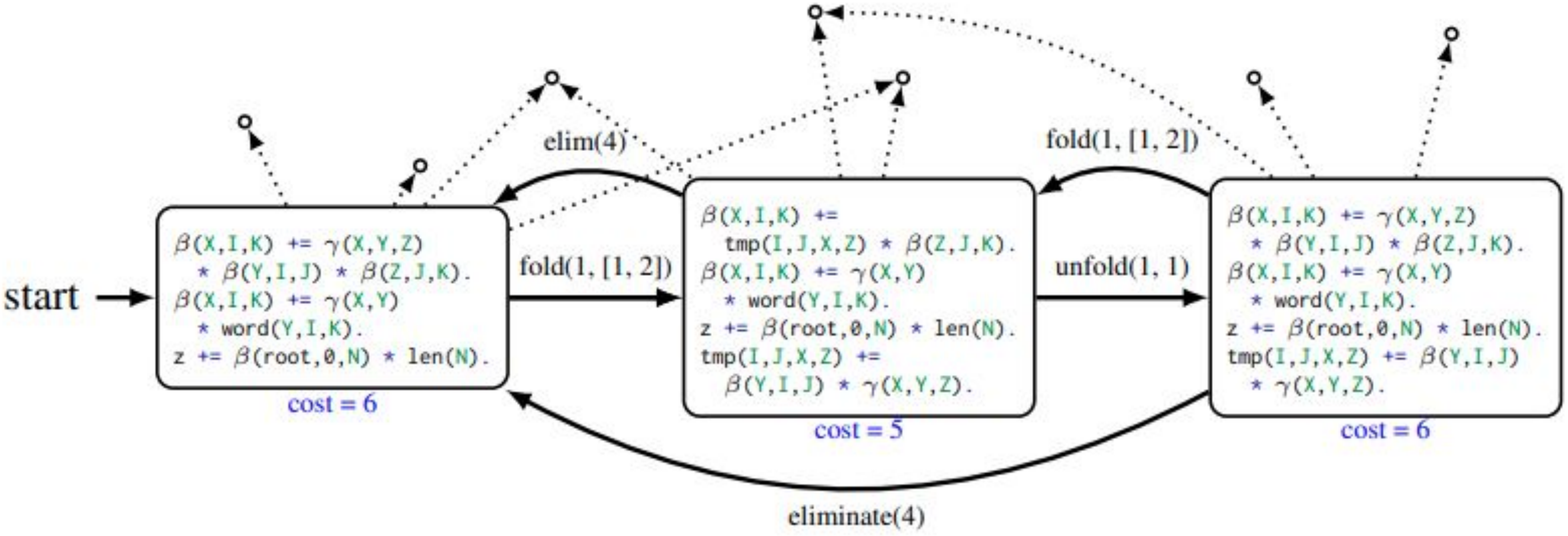
Summary

- 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.

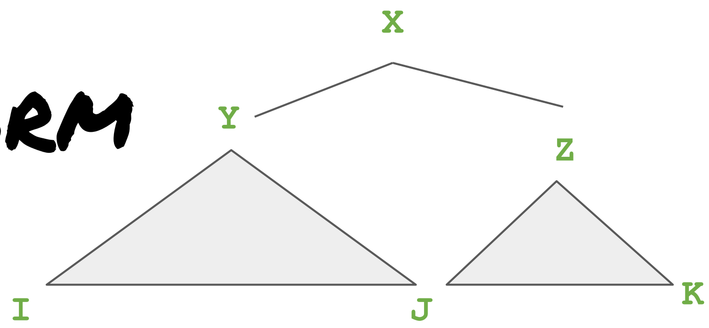
Thanks!



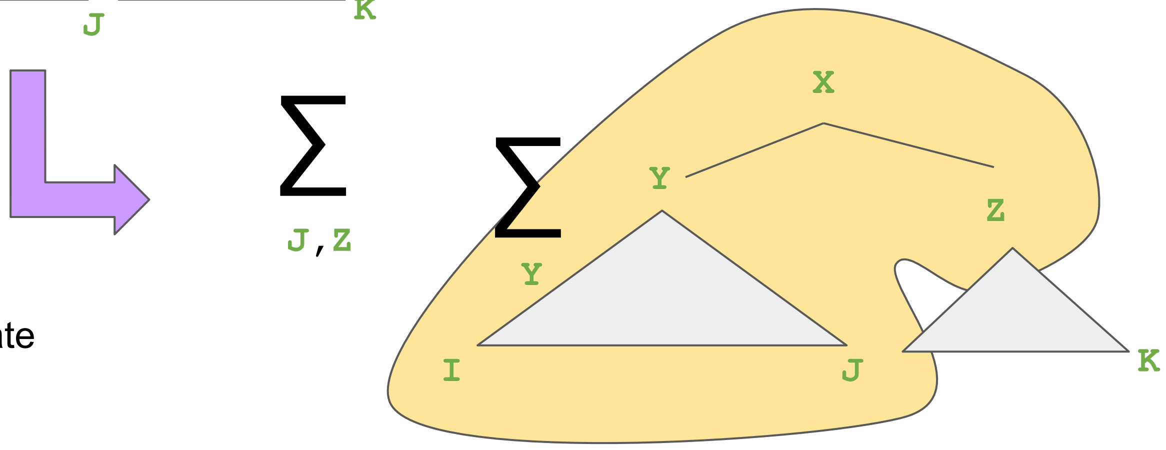
https://arxiv.org/abs/2109.06966

https://twitter.com/xtimv/status/1438611768868225026

Fold Transform

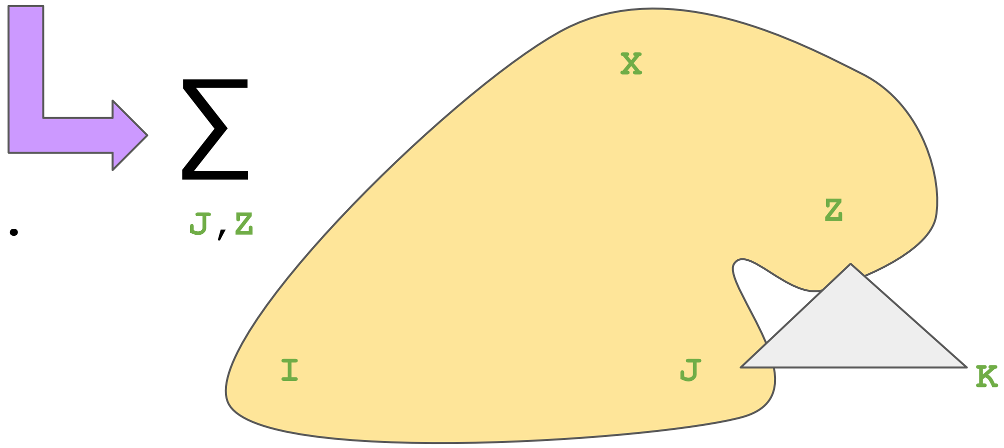



```
β(X,I,K) += γ(X,Y,Z)
   * β(Y,I,K) * β(Z,J,K).
```



creates an intermediate item **tmp**(X,I,J,Z)

```
β(X,I,K) +=
   tmp(X,I,J,Z) * β(Z,J,K).
tmp(X,I,J,Z) +=
   γ(X,Y,Z) * β(Y,I,K).
```



Generalizes the “hook trick”