Weighted Deduction as an Abstraction Level for AI

Jason Eisner

co-authors on various parts of this work:
Eric Goldlust, Noah A. Smith, John Blatz, Wes Filardo, Wren Thornton

ILP+MLG+SRL (invited talk), July 2009
Q: What do these formalisms have in common?

A1: They all took a lot of sweat to implement 😊

A2: None is perfect (that’s why someone built the next)

Okay, they do have some ideas in common too. (e.g., logic + probability)

But then they should be able to partly share implementation.
This problem is not limited to SRL.

Also elsewhere in AI (and maybe beyond).
Let’s look at natural language processing systems …

Also do inference and learning,
but for other kinds of structured models.

Models: e.g., various kinds of probabilistic grammars.
Algorithms: dynamic programming, beam search, …
## Natural Language Processing (NLP)

Large-scale noisy data, complex models, search approximations, software engineering

<table>
<thead>
<tr>
<th>NLP sys</th>
<th>files</th>
<th>code (lines)</th>
<th>comments</th>
<th>lang (primary)</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRILM</td>
<td>308</td>
<td>49879</td>
<td>14083</td>
<td>C++</td>
<td>LM</td>
</tr>
<tr>
<td>LingPipe</td>
<td>502</td>
<td>49967</td>
<td>47515</td>
<td>Java</td>
<td>LM/IE</td>
</tr>
<tr>
<td>Charniak parser</td>
<td>259</td>
<td>53583</td>
<td>8057</td>
<td>C++</td>
<td>Parsing</td>
</tr>
<tr>
<td>Stanford parser</td>
<td>373</td>
<td>121061</td>
<td>24486</td>
<td>Java</td>
<td>Parsing</td>
</tr>
<tr>
<td>GenPar</td>
<td>986</td>
<td>77922</td>
<td>12757</td>
<td>C++</td>
<td>Parsing/MT</td>
</tr>
<tr>
<td>MOSES</td>
<td>305</td>
<td>42196</td>
<td>6946</td>
<td>Perl, C++,…</td>
<td>MT</td>
</tr>
<tr>
<td>GIZA++</td>
<td>124</td>
<td>16116</td>
<td>2575</td>
<td>C++</td>
<td>MT alignment</td>
</tr>
</tbody>
</table>
NLP systems are big!
Large-scale noisy data, complex models, search approximations, software engineering

Consequences:

- **Barriers to entry**
  - Small number of players
  - Significant investment to be taken seriously
  - Need to know & implement the standard tricks

- **Barriers to experimentation**
  - Too painful to tear up and reengineer your old system, to try a cute idea of unknown payoff

- **Barriers to education and sharing**
  - Hard to study or combine systems
  - Potentially general techniques are described and implemented only one context at a time
Didn’t I just implement something like this last month?

chart management / indexing
cache-conscious data structures
memory layout, file formats, integerization, …
prioritization of partial solutions (best-first, A*)
lazy k-best, forest reranking
parameter management
inside-outside formulas, gradients, …
different algorithms for training and decoding
conjugate gradient, annealing, …
parallelization

I thought computers were supposed to automate drudgery
A few other applied AI systems …

Large-scale noisy data, complex models, search approximations, software engineering

- Maybe a bit smaller outside NLP
- Nonetheless, big and carefully engineered
  - And will get bigger, e.g., as machine vision systems do more scene analysis and compositional object modeling

<table>
<thead>
<tr>
<th>System</th>
<th>files</th>
<th>code</th>
<th>comments</th>
<th>lang</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProbCons</td>
<td>15</td>
<td>4442</td>
<td>693</td>
<td>C++</td>
<td>MSA of amino acid seqs</td>
</tr>
<tr>
<td>MUSTANG</td>
<td>50</td>
<td>7620</td>
<td>3524</td>
<td>C++</td>
<td>MSA of protein structures</td>
</tr>
<tr>
<td>MELISMA</td>
<td>44</td>
<td>7541</td>
<td>1785</td>
<td>C</td>
<td>Music analysis</td>
</tr>
<tr>
<td>Dynagraph</td>
<td>218</td>
<td>20246</td>
<td>4505</td>
<td>C++</td>
<td>Graph layout</td>
</tr>
</tbody>
</table>
Can toolkits help?

<table>
<thead>
<tr>
<th>NLP tool</th>
<th>files</th>
<th>code</th>
<th>comments</th>
<th>lang</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTK</td>
<td>111</td>
<td>88865</td>
<td>14429</td>
<td>C</td>
<td>HMM for ASR</td>
</tr>
<tr>
<td>OpenFST</td>
<td>150</td>
<td>20502</td>
<td>1180</td>
<td>C++</td>
<td>Weighted FSTs</td>
</tr>
<tr>
<td>TIBURON</td>
<td>53</td>
<td>13791</td>
<td>4353</td>
<td>Java</td>
<td>Tree transducers</td>
</tr>
<tr>
<td>AGLIB</td>
<td>163</td>
<td>58475</td>
<td>5853</td>
<td>C++</td>
<td>Annotation of time series</td>
</tr>
<tr>
<td>UIMA</td>
<td>1577</td>
<td>154547</td>
<td>110183</td>
<td>Java</td>
<td>Unstructured-data mgmt</td>
</tr>
<tr>
<td>GATE</td>
<td>1541</td>
<td>79128</td>
<td>42848</td>
<td>Java</td>
<td>Text engineering mgmt</td>
</tr>
<tr>
<td>NLTK</td>
<td>258</td>
<td>60661</td>
<td>9093</td>
<td>Python</td>
<td>NLP algs (educational)</td>
</tr>
<tr>
<td>libbow</td>
<td>122</td>
<td>42061</td>
<td>9198</td>
<td>C</td>
<td>IR, textcat, etc.</td>
</tr>
<tr>
<td>MALLET</td>
<td>559</td>
<td>73859</td>
<td>18525</td>
<td>Java</td>
<td>CRFs and classification</td>
</tr>
<tr>
<td>GRMM</td>
<td>90</td>
<td>12584</td>
<td>3286</td>
<td>Java</td>
<td>Graphical models add-on</td>
</tr>
</tbody>
</table>
Can toolkits help?

- Hmm, there are a lot of toolkits (more alphabet soup).
- The toolkits are big too.
- And no toolkit does everything you want.
  - Which is why people keep writing them.
  - E.g., I love & use OpenFST and have learned lots from its implementation! But sometimes I also want ...

  - automata with > 2 tapes
  - infinite alphabets
  - parameter training
  - A* decoding
  - automatic integerization
  - automata defined “by policy”
  - mixed sparse/dense implementation (per state)
  - parallel execution
  - hybrid models (90% finite-state)

- So what is common across toolkits?
Presumably, we ought to add another layer of abstraction.
- After all, this is CS.

Hope to convince you that a substantive new layer exists.

But what would it look like?
- What’s shared by programs/toolkits/frameworks?
  - **Declaratively**: Weighted logic programming
  - **Procedurally**: Truth maintenance on equations
The Dyna programming language

*Intended as a common infrastructure*

- Most toolkits or declarative languages guide you to model or solve your problem in a particular way.
- That can be a good thing!
- Just the right semantics, operations, and algorithms for that domain and approach.

- In contrast, Dyna is **domain-independent**.
- Manages **data & computations** that you specify.
  - Doesn’t care what they mean. It’s one level lower than that.
- Languages, toolkits, applications can be built **on top**.
Warning

- Lots more beyond this talk
- See http://dyna.org

read our papers

download an earlier prototype

contact eisner@jhu.edu to

- send feature requests, questions, ideas, etc.
- offer help, recommend great students / postdocs
- get on the announcement list for Dyna 2 release
A Quick Sketch of Dyna
Writing equations in Dyna

- int a.
- a = b * c.
  
  a will be kept up to date if b or c changes.
- b += x.
  b += y.  equivalent to b = x+y.  (almost)
  b is a sum of two variables.  Also kept up to date.
- c += z(1).
- c += z(2).
- c += z(3).
- c += z("four").
- c += z(foo(bar,5)).

  c is a sum of all defined z(...) values.
  At compile time, we don’t know how many!
More interesting use of patterns

- \( a = b \times c. \)
  - scalar multiplication
- \( a(I) = b(I) \times c(I). \)
  - pointwise multiplication
- \( a += b(I) \times c(I). \)
  - dot product; could be sparse
- \( a(I,K) += b(I,J) \times c(J,K). \)
  - matrix multiplication; could be sparse
  - \( J \) is free on the right-hand side, so we sum over it

... + b("yetis") * c("yetis") + b("zebra") * c("zebra")

sparse dot product of query & document

\[ a = \sum_I b(I) \times c(I) \]

\[ \sum_J b(I,J) \times c(J,K) \]
By now you may see what we’re up to!

Prolog has Horn clauses:

\[ a(I,K) :- b(I,J) , c(J,K) . \]

Dyna has “Horn equations”:

\[ a(I,K) += b(I,J) \times c(J,K) . \]

prove a value for it

e.g., a real number,

but could be any term
definition from other values

b*c only has value when b and c do

if no values enter into +=, then a gets no value

Like Prolog:

Allow nested terms
- Syntactic sugar for lists, etc.
- Turing-complete

Unlike Prolog:

Terms can have values
Terms are evaluated in place
Not just backtracking! (+ no cuts)
Type system; static optimizations
Aggregation operators

- Associative/commutative:
  - $b += a(X)$. % number
  - $c \text{ min=} a(X)$.

- E.g., single-source shortest paths:
  - $\text{pathto(start)} \text{ min=} 0$.
  - $\text{pathto(W)} \text{ min=} \text{pathto(V)} + \text{edge(V,W)}$. 
**Aggregation operators**

- **Associative/commutative:**
  - \( b \ += \ a(X). \ % \) number
  - \( c \ \text{min}= \ a(X). \)
  - \( q \ |= \ p(X). \ % \) boolean
  - \( r \ \&= \ p(X). \)
  - ...  

- **Require uniqueness:**
  - \( d = \) \( b+c. \)
  - \( e = \ a(X). \ % \) may fail at runtime

- **Last one wins:**
  - \( \text{fly}(X) := \) true if \( \text{bird}(X). \)
  - \( \text{fly}(X) := \) false if \( \text{penguin}(X). \)
  - \( \text{fly(bigbird)} := \) false.

- **Most specific wins** (syn. sugar):
  - \( \text{fib}(0) => 0. \)
  - \( \text{fib}(1) => 1. \)
  - \( \text{fib(int N)} => \text{fib(N-1)} + \text{fib(N-2)}. \)

- Each ground term has a **single**, type-safe aggregation operator.  
- Some ground terms are willing to accept new aggregands at runtime.  
- (**Note:** Rules define values for ground terms only, using variables.)
Some connections and intellectual debts …

- Deductive parsing schemata (preferably **weighted**)
  - Goodman, Nederhof, Pereira, McAllester, Warren, Shieber, Schabes, Sikkel…

- Deductive databases (preferably with **aggregation**)
  - Ramakrishnan, Zukowski, Freitag, Specht, Ross, Sagiv, …
  - Query optimization
  - Usually limited to decidable fragments, e.g., Datalog

- Theorem proving
  - Theorem provers, term rewriting, etc.
  - Nonmonotonic reasoning

- Programming languages
  - Functional logic programming (Curry, …)
  - Probabilistic programming languages (PRISM, ProbLog, IBAL …)
  - Efficient Prologs (Mercury, XSB, …)
  - Self-adjusting computation, adaptive memoization (Acar et al.)
  - Declarative networking (P2)
  - XML processing languages (XTatic, CDuce)

Increasing interest in resurrecting declarative and logic-based system specifications.
Why is this a good abstraction level?

We’ll see examples soon, but first the big picture …
How you build a system (“big picture” slide)

- cool model
- equations to compute (approx.) results
  \[ \beta_x(i, k) = \sum_{0 \leq j < k \leq n} \beta_y(i, j) \beta_z(j, k) \]
  \[ p(N_x \rightarrow N_y N_z | N_x) \]
  ...
- pseudocode (execution order)
  for width from 2 to n
  for i from 0 to n-width
  k = i+width
  for j from i+1 to k-1
  ...
- tuned C++
  implementation (data structures, etc.)
How you build a system ("big picture" slide)

Cool model

Equations to compute (approx.) results

$$\beta_x(i, k) = \sum_{0 \leq i < j < k \leq n} \beta_y(i, j) \beta_z(j, k) p(N_x \rightarrow N_yN_z | N_x)$$

...  

Dyna language specifies these equations.

Most programs just need to compute some values from other values. Any order is ok.

Feed-forward!
Dynamic programming!
Message passing! (including Gibbs)

Must quickly figure out what influences what.

Compute Markov blanket
Compute transitions in state machine
How you build a system ("big picture" slide)

practical equations

\[ \beta_x(i, k) = \sum_{0 \leq i < j < k \leq n} \beta_y(i, j) \beta_z(j, k) \]

\[ p(N_x \rightarrow N_y N_z \mid N_x) \]

Dyna language specifies these equations.

Most programs just need to compute some values from other values. Any order is ok. May be cyclic.

Some programs also need to update the outputs if the inputs change:
- spreadsheets, makefiles, email readers
- dynamic graph algorithms
- MCMC, WalkSAT: Flip variable & energy changes
- Training: Change params & obj. func. changes
- Cross-val: Remove 1 example & obj. func. changes
How you build a system (“big picture” slide)

cool model

practical equations

\[ \beta_x(i, k) = \sum_{0 \leq i < j < k \leq n} \beta_y(i, j) \beta_z(j, k) \]

\[ p(N_x \rightarrow N_yN_z | N_x) \]

... 

Execution strategies (we’ll come back to this)

pseudocode (execution order)

for width from 2 to n
    for i from 0 to n-width
        k = i+width
        for j from i+1 to k-1
            ...

tuned C++ implementation (data structures, etc.)
Common threads in NLP, SRL, KR&R, …

Dyna hopes to support these

- **Pattern matching** against **structured objects** (e.g., terms)
- **Message passing** among terms (implemented by Horn equations)
  - Implication: “We got proved, so now you’re proved too!”
  - Probabilistic inference: “Proved you another way! Add 0.02.”
  - Arc consistency: “My domain is reduced, so reduce yours.”
  - Belief propagation: “My message is updated, so update yours.”
  - Bounds/box propagation: “My estimate is tighter, so tighten yours.”
  - Gibbs sampling: “My value is updated, so update yours.”
  - Counting: “++count(rule)” “++count(feature)” “++count(subgraph)”
  - Dynamic programming: “Here’s my best solution, so update yours.”
  - Dynamic algorithms: “The world changed, so adjust conclusions.”

- **Aggregation** of messages from multiple sources

- **Default reasoning**
  - Lifting, program transfs: Reasoning with non-ground terms
  - Nonmonotonicity: Exceptions to the rule, using := or =>

- **Inspection of proof forests** (derivation forests)
- **Automatic differentiation** for training free parameters
Pattern matching against structured objects (e.g., terms)

Message passing among terms (implemented by Horn equations)

Implication: “We got proved, so now you’re proved too!”

Probabilistic inference: “Proved you another way! Add 0.02.”

Arc consistency: “My domain is reduced, so reduce yours.”

Belief propagation: “My message is updated, so update yours.”

Bounds/box propagation: “My estimate is tighter, so tighten yours.”

Gibbs sampling: “My value is updated, so update yours.”

Counting: “++count(rule)”, “++count(feature)”, “++count(subgraph)”

Dynamic programming: “Here’s my best solution, so update yours.”

Dynamic algorithms: “The world changed, so adjust conclusions.”

Aggregation of messages from multiple sources

Default reasoning

Lifting, program transfs: Just reasoning with non-ground terms

Nonmonotonicity: Exceptions to the rule, using := or =>

Inspection of proof forests

Automatic differentiation for training free parameters

Note: Semantics of these messages may differ widely.

E.g., consider some common uses of real numbers:

✓ probability, unnormalized probability, log-probability
✓ approximate probability (e.g., in belief propagation)
✓ strict upper or lower bound on probability
✓ A* heuristic; inadmissible best-first heuristic
✓ feature weight or other parameter of model or of var. approx.
✓ count, count ratio, distance, scan statistic, …
✓ mean, variance, degree … (suff. statistic for Gibbs sampling)
✓ activation in neural net; similarity according to kernel
✓ utility, reward, loss, rank, preference
✓ expectation (e.g., expected count; risk = expected loss)
✓ entropy, regularization term, …
✓ partial derivative

Common threads in NLP, SRL, KR&R, …
Common implementation issues

Dyna hopes to support these

- **Efficient storage**
  - Your favorite data structures
    - (BDDs? tries? arrays? hashes? Bloom filters?)

- **Efficient computation of new messages**
  - **Unification** of queries against clause heads or memos
  - **Indexing** of facts, clauses, and memo table
  - **Query planning** for unindexed queries (e.g., joins)

- **Deciding which messages to send, and when**
  - **Forward chaining** (eager, breadth-first)
    - Priority queue order – this can matter!
  - **Backward chaining** (lazy, depth-first)
    - Memoization, a.k.a. tabling
    - Updating and flushing memos
  - **Magic templates** (lazy, breadth-first)
  - Hybrid strategies
  - Avoiding useless messages (e.g., convergence, watched variables)

- **Code as data** (static analysis, program transformation)

- **Parallelization**
Example:
CKY and Variations
The CKY inside algorithm in Dyna

\[
\text{phrase}(X,I,J) \;+\; \text{rewrite}(X,W) \times \text{word}(W,I,J).
\]

\[
\text{phrase}(X,I,J) \;+\; \text{rewrite}(X,Y,Z) \times \text{phrase}(Y,I,Mid) \times \text{phrase}(Z,Mid,J).
\]

\[
\text{goal} \;+\; \text{phrase}(\text{“s”},0,\text{sentence_length}).
\]
The CKY inside algorithm in Dyna

\[
\text{phrase}(X,I,J) \; +\; \text{rewrite}(X,Y,Z) \; \cdot \text{phrase}(Y,I,Mid) \; \cdot \text{phrase}(Z,Mid,J).
\]
The CKY inside algorithm in Dyna

using namespace cky;
chart c;

phrase(X,I,J) += rewrite(X,W) * word(W,I,J).
phrase(X,I,J) += rewrite(X,Y,Z) * phrase(Y,I,Mid) * phrase(Z,Mid,J).
goal += phrase(“s”,0,sentence_length).

put in axioms (values not defined by the above program)

using namespace cky;
chart c;

c[rewrite(“s”,“np”,“vp”)] = 0.7;
c[word(“Pierre”,0,1)] = 1;
c[sentence_length] = 30;
cin >> c; // get more axioms from stdin

cout << c[goal]; // print total weight of all parses

(C++ API for older prototype version)
Visual debugger: Browse the proof forest

- desired theorem
- ambiguity
- dead end
- shared substructure (dynamic programming)
- axioms
Visual debugger: Browse the proof forest

- ambiguity
- constit(s,0,5)
- length(5)

- constit(vp,2,5)
- constit(vp,1,5)
- constit(s/vp,3,5)
- constit(s/vp,0,2)
- constit(s/vp)

- tit(vp/advp,2,3)
- constit(vp/advp,1,2)
- constit(advp,2,5)
- constit(np,0,2)
- rewrites(s,np)

- rewrites(vp,v,advp)
- constit(advp(np,2,3)
- constit(np,3,5)
- constit(n,1,2)
- constit(np)

- shared substructure (dynamic programming)

- dead end
Parameterization ...

- `phrase(X,I,J) += rewrite(X,W) * word(W,I,J).`
- `phrase(X,I,J) += rewrite(X,Y,Z) * phrase(Y,I,Mid) * phrase(Z,Mid,J).`
- `goal += phrase("s",0,sentence_length).`

- **rewrite(X,Y,Z)** doesn’t have to be an atomic parameter:
  - `urewrite(X,Y,Z) *= weight1(X,Y).`
  - `urewrite(X,Y,Z) *= weight2(X,Z).`
  - `urewrite(X,Y,Z) *= weight3(Y,Z).`
  - `urewrite(X,Same,Same) *= weight4.`

  - `urewrite(X) += urewrite(X,Y,Z). % normalizing constant`
  - `rewrite(X,Y,Z) = urewrite(X,Y,Z) / urewrite(X). % normalize`
Related algorithms in Dyna?

- Viterbi parsing?
- Logarithmic domain?
- Lattice parsing?
- Incremental (left-to-right) parsing?
- Log-linear parsing?
- Lexicalized or synchronous parsing?
- Binarized CKY?
- Earley’s algorithm?
Related algorithms in Dyna?

- Viterbi parsing?
- Logarithmic domain?
- Lattice parsing?
- Incremental (left-to-right) parsing?
- Log-linear parsing?
- Lexicalized or synchronous parsing?
- Binarized CKY?
- Earley’s algorithm?
### Related algorithms in Dyna?

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>phrase(X,I,J)</code></td>
<td>( \log + ) rewrite(X,W) + word(W,I,J).</td>
</tr>
<tr>
<td><code>phrase(X,I,J)</code></td>
<td>( \log + ) rewrite(X,Y,Z) + phrase(Y,I,Mid) + phrase(Z,Mid,J).</td>
</tr>
<tr>
<td><code>goal</code></td>
<td>( \log + ) phrase(&quot;s&quot;,0,sentence_length).</td>
</tr>
</tbody>
</table>

- Viterbi parsing?
- **Logarithmic domain?**
- Lattice parsing?
- Incremental (left-to-right) parsing?
- Log-linear parsing?
- Lexicalized or synchronous parsing?
- Binarized CKY?
- Earley’s algorithm?
Related algorithms in Dyna?

- Viterbi parsing?
- Logarithmic domain?
- **Lattice parsing?**
- Incremental (left-to-right) parsing?
- Log-linear parsing?
- Lexicalized or synchronous parsing?
- Binarized CKY?
- Earley’s algorithm?

```
phrase(X,I,J) += rewrite(X,W) * word(W,I,J).
phrase(X,I,J) += rewrite(X,Y,Z) * phrase(Y,I,Mid) * phrase(Z,Mid,J).
goal += phrase(“s”,0,sentence_length).
```

- c[ word(“Pierre”, state(5), state(9) ) ] = 0.2
Related algorithms in Dyna?

- Viterbi parsing?
- Logarithmic domain?
- Lattice parsing?
- **Incremental (left-to-right) parsing?**
- Log-linear parsing?
- Lexicalized or synchronous parsing?
- Binarized CKY?
- Earley’s algorithm?

\[
\begin{align*}
\text{phrase}(X,I,J) & \quad +\quad \text{rewrite}(X,W) \quad * \quad \text{word}(W,I,J). \\
\text{phrase}(X,I,J) & \quad +\quad \text{rewrite}(X,Y,Z) \quad * \quad \text{phrase}(Y,I,Mid) \quad * \quad \text{phrase}(Z,Mid,J). \\
\text{goal} & \quad +\quad \text{phrase}(“s”,0,\text{sentence_length}).
\end{align*}
\]

Just add words one at a time to the chart.
Check at any time what can be derived from words so far.
Similarly, dynamic grammars.
Related algorithms in Dyna?

- Viterbi parsing?
- Logarithmic domain?
- Lattice parsing?
- Incremental (left-to-right) parsing?
- **Log-linear parsing?** Again, no change to the Dyna program
- Lexicalized or synchronous parsing?
- Binarized CKY?
- Earley’s algorithm?
Related algorithms in Dyna?

- Viterbi parsing?
- Logarithmic domain?
- Lattice parsing?
- Incremental (left-to-right) parsing?
- Log-linear parsing?
- **Lexicalized or synchronous parsing?**
- Binarized CKY?
- Earley’s algorithm?

### Phrase Rewriting

\[
\text{phrase}(X, I, J) \; \text{+=} \; \text{rewrite}(X, W) \; \text{*} \; \text{word}(W, I, J).
\]

\[
\text{phrase}(X, I, J) \; \text{+=} \; \text{rewrite}(X, Y, Z) \; \text{*} \; \text{phrase}(Y, I, \text{Mid}) \; \text{*} \; \text{phrase}(Z, \text{Mid}, J).
\]

\[
\text{goal} \; \text{+=} \; \text{phrase}(“s”, 0, \text{sentence_length}).
\]

Basically, just add extra arguments to the terms above.
Related algorithms in Dyna?

1. Viterbi parsing?
2. Logarithmic domain?
3. Lattice parsing?
4. Incremental (left-to-right) parsing?
5. Log-linear parsing?
6. Lexicalized or synchronous parsing?
7. Binarized CKY?
8. Earley’s algorithm?

$$\text{phrase}(X,I,J) \; += \; \text{rewrite}(X,W) \; * \; \text{word}(W,I,J).$$

$$\text{phrase}(X,I,J) \; += \; \text{rewrite}(X,Y,Z) \; * \; \text{phrase}(Y,I,Mid) \; * \; \text{phrase}(Z,Mid,J).$$

$$\text{goal} \; += \; \text{phrase}(“s” ,0, \text{sentence\_length}).$$
Rule binarization

\[
\text{phrase}(X, I, J) \ \Rightarrow \ \text{phrase}(Y, I, \text{Mid}) \times \text{phrase}(Z, \text{Mid}, J) \times \text{rewrite}(X, Y, Z).
\]

\[
\text{temp}(X\backslash Y, \text{Mid}, J) \ \Rightarrow \ \text{phrase}(Z, \text{Mid}, J) \times \text{rewrite}(X, Y, Z).
\]

\[
\text{phrase}(X, I, J) \ \Rightarrow \ \text{phrase}(Y, I, \text{Mid}) \times \text{temp}(X\backslash Y, \text{Mid}, J).
\]
Rule binarization

\[\text{phrase}(X,I,J) \; += \; \text{phrase}(Y,I,Mid) \times \text{phrase}(Z,Mid,J) \times \text{rewrite}(X,Y,Z).\]

\[\text{temp}(X \setminus Y,Mid,J) \; += \; \text{phrase}(Z,Mid,J) \times \text{rewrite}(X,Y,Z).\]

\[\text{phrase}(X,I,J) \; += \; \text{phrase}(Y,I,Mid) \times \text{temp}(X \setminus Y,Mid,J).\]

\[\sum_{Y,Z,Mid} \text{phrase}(Y,I,Mid) \times \text{phrase}(Z,Mid,J) \times \text{rewrite}(X,Y,Z)\]

\[\sum_{Y,Mid} \text{phrase}(Y,I,Mid) \times \sum_{Z} \text{phrase}(Z,Mid,J) \times \text{rewrite}(X,Y,Z)\]

folding transformation: asymp. speedup!

graphical models

constraint programming

multi-way database join
Program transformations

cool model

practical equations

\[ \beta_x(i, k) = \sum_{0 \leq i < j < k \leq n} \beta_y(i, j) \beta_z(j, k) \]

Eisner & Blatz (FG 2007):

Lots of equivalent ways to write a system of equations!

Transforming from one to another may improve efficiency.

Many parsing “tricks” can be generalized into automatic transformations that help other programs, too!
Related algorithms in Dyna?

- Viterbi parsing?
- Logarithmic domain?
- Lattice parsing?
- Incremental (left-to-right) parsing?
- Log-linear parsing?
- Lexicalized or synchronous parsing?
- Binarized CKY?
- Earley’s algorithm?
Earley’s algorithm in Dyna

\[
\text{phrase}(X,I,J) \;+\; \text{rewrite}(X,W) \;* \;\text{word}(W,I,J).
\]

\[
\text{phrase}(X,I,J) \;+\; \text{rewrite}(X,Y,Z) \;* \;\text{phrase}(Y,I,\text{Mid}) \;* \;\text{phrase}(Z,\text{Mid},J).
\]

\[
\text{goal} \;+\; \text{phrase}(“s”,0,\text{sentence\_length}).
\]

\text{need(“s”,0) = true.}

\text{need(Nonterm,J) :- \text{phrase(}_/\text{[Nonterm\_\_],_},J).}

\text{phrase(Nonterm/Needed,I,I)}
\quad +\; \text{need(Nonterm,I), rewrite(Nonterm,Needed).}

\text{phrase(Nonterm/Needed,I,K)}
\quad +\; \text{phrase(Nonterm/[W|Needed],I,J) \;* \;\text{word}(W,J,K).}

\text{phrase(Nonterm/Needed,I,K)}
\quad +\; \text{phrase(Nonterm/[X|Needed],I,J) \;* \;\text{phrase}(X/[],J,K).}

\text{goal} \;+\; \text{phrase(“s”/[],0,\text{sentence\_length}).}

\text{magic templates transformation}
\quad \text{(as noted by Minnen 1996)}
Related algorithms in Dyna?

- Viterbi parsing?
- Logarithmic domain?
- Lattice parsing?
- Incremental (left-to-right) parsing?
- Log-linear parsing?
- Lexicalized or synchronous parsing?
- Binarized CKY?
- Earley’s algorithm?
- Epsilon symbols?

\[
\begin{align*}
\text{phrase}(X,I,J) & \quad \text{+= rewrite}(X,W) \times \text{word}(W,I,J). \\
\text{phrase}(X,I,J) & \quad \text{+= rewrite}(X,Y,Z) \times \text{phrase}(Y,I,Mid) \times \text{phrase}(Z,Mid,J). \\
\text{goal} & \quad \text{+= phrase(“s”,0,sentence_length).}
\end{align*}
\]

\[
\text{word(epsilon,I,I)} = 1. \\
(\text{i.e., epsilons are freely available everywhere})
\]
Some examples from my lab (as of 2006, w/prototype)...

- **Parsing using …**
  - factored dependency models (Dreyer, Smith, & Smith CONLL’06)
  - with annealed risk minimization (Smith and Eisner EMNLP’06)
  - constraints on dependency length (Eisner & Smith IWPT’05)
  - *unsupervised learning of deep transformations* (see Eisner EMNLP’02)
  - *lexicalized algorithms* (see Eisner EMNLP’02, ACL’99, etc.)

- **Grammar induction**
  - partial supervision (Dreyer & Eisner EMNLP’06)
  - structural annealing (Smith & Eisner ACL’06)
  - contrastive estimation (Smith & Eisner GIA’05)
  - deterministic annealing (Smith & Eisner ACL’04)

- **Machine translation**
  - Very large neighborhood search of permutations (Eisner & Tromble, NAACL-W’06)
  - Loosely syntax-based MT (Smith & Eisner in prep.)
  - Synchronous cross-lingual parsing (Smith & Smith EMNLP’04 - see also Eisner ACL’03)

- **Finite-state methods for morphology, phonology, IE, even syntax …**
  - Unsupervised cognate discovery (Schafer & Yarowsky ’05, ’06)
  - Unsupervised log-linear models via contrastive estimation (Smith & Eisner ACL’05)
  - Context-based morph. disambiguation (Smith, Smith & Tromble EMNLP’05)
  - *Trainable (in)finite-state machines* (see Eisner ACL’02, EMNLP’02, …)
  - *Finite-state machines with very large alphabets* (see Eisner ACL’97)
  - *Finite-state machines over weird semirings* (see Eisner ACL’02, EMNLP’03)

- **Teaching** (Eisner JHU’05-06; Smith & Tromble JHU’04)

---

Easy to try stuff out!

Programs are very short & easy to change!
A few more language details

So you’ll understand the examples …
Terms (generalized from Prolog)

- These are the “Objects” of the language
- Primitives:
  - 3, 3.14159, “myUnicodeString”
  - user-defined primitive types
- Variables:
  - X
  - int X [type-restricted variable; types are tree automata]
- Compound terms:
  - atom
  - atom(subterm1, subterm2, …) e.g., f(g(h(3),X,Y), Y)
  - Adding support for keyword arguments (similar to R, but must support unification)
A Dyna program is a finite rule set that defines a partial function (“dynabase”).

- Dynabase only defines values for ground terms
  - Variables (X,Y,…) let us define values for infinitely many ground terms
  - Compute values that satisfy the equations in the program
  - Not guaranteed to halt (Dyna is Turing-complete, unlike Datalog)
  - Not guaranteed to be unique
Fixpoint semantics

- A Dyna program is a finite rule set that defines a partial function ("dynabase")

- Dynabase only defines values for ground terms
- Dynabase remembers relationships
  - Runtime input
  - Adjustments to input (dynamic algorithms)
  - Retraction (remove input), detachment (forget input but preserve output)
“Object-oriented” features

- Dynabases are terms, i.e., first-class objects

- Dynabases can appear as subterms or as values
  - Useful for encapsulating data and passing it around:
    - \( \text{fst3} = \text{compose(fst1, fst2)} \). % value of \( \text{fst3} \) is a dynabase
    - \( \text{forest} = \text{parse(sentence)}. \)
  - Typed by their public interface:
    - \( \text{fst4} \rightarrow \text{edge(Q,R)} \) += \( \text{fst3} \rightarrow \text{edge(R,Q)}. \)

- Dynabases can be files or web services
  - Human-readable format (looks like a Dyna program)
  - Binary format (mimics in-memory layout)
Creating dynabases

immutable dynabase literal

mygraph(int N) =
{ edge("a", "b") += 3.
  edge("b", "c") = edge("a", "b")*N.
  color("b") := purple. }

So if it’s immutable, how are the deductive rules still live? How can we modify inputs and see how outputs change?

mygraph(6) → edge("a", "b") has value 3.
mygraph(6) → edge("b", "c") has value 18.
Creating dynabases

- mygraph(int N) :=
  {
    edge("a", "b") += 3.
    edge("b", "c") = edge("a", "b")*N.
    color("b") := purple.
  }

immutable dynabase literal

cloning

define how this clone differs

- mygraph(6) → edge("a", "b") += 2.

mygraph(6) → edge("b", "c") has value 18.
Creating dynabases

- mygraph(int N) :=
  ```
  { edge("a", "b") += 3.
    edge("b", "c") = edge("a", "b")*N.
    color("b") := purple. }
  ```

define how this clone differs

- mygraph(6) → edge("a", "b") += 2.
- mygraph(N) → color(S) :=
  ```
  coloring(load("yourgraph.dyna")) → color(S).
  ```

these dynabases are also immutable (by us) since fully defined elsewhere

mygraph(6) → edge("b", "c") has value 18.
Functional features: Auto-evaluation

- Terms can have values.
- So by default, subterms are evaluated in place.
  - Arranged by a simple desugaring transformation:
    
    ```
    foo( X ) += 3*bar(X).
    => foo( X ) += B is bar(X), Result is 3*B, Result.
    ```

  - Possible to suppress evaluation \( &f(x) \) or force it \( *f(x) \)
    - Some contexts also suppress evaluation.
    - Variables are replaced with their bindings but not otherwise evaluated.
Functional features: Auto-evaluation

- Terms can have values.
- So by default, subterms are evaluated in place.
  - Arranged by a simple desugaring transformation:
    \[ \text{foo}(f(X)) += 3*\text{bar}(g(X)). \]
    \[ \Rightarrow \text{foo}( F ) += F \text{ is } f(X), \ G \text{ is } g(X), \ B \text{ is } \text{bar}(G), \text{ Result is } 3*B, \text{ Result.} \]

- Possible to suppress evaluation \&f(x) or force it *f(x)
  - Some contexts also suppress evaluation.
  - Variables are replaced with their bindings but not otherwise evaluated.
Other handy features

- \( \text{fact}(0) = 1. \)
- \( \text{fact}(\text{int } N) = N > 0, N \times \text{fact}(N-1). \)

Restricts applicability of this rule.

(Note: There’s a strong type system, but it’s optional. Use it as desired for safety and efficiency, and to control the implementation.)

Guard condition on a rule:
If \( X \) is true, then \( X,Y \) has value \( Y \).
Otherwise \( X,Y \) is not provable.

- \( 0! = 1. \) user-defined syntactic sugar
- \( (\text{int } N)! = N \times (N-1)! \text{ if } N \geq 1. \) Unicode

Degenerate aggregator.
Like \( += \), but it’s an error if it tries to aggregate more than one value.
Frozen variables

- Dynabase semantics concerns ground terms.
- But want to be able to reason about non-ground terms, too.
  - Manipulate Dyna rules (which are non-ground terms)
  - Work with classes of ground terms (specified by non-ground terms)
    - Queries, memoized queries …
    - Memoization, updating, prioritization of updates, …

- So, allow ground terms that contain “frozen variables”
  - Treatment under unification is beyond scope of this talk

- $\text{priority}(f(X)) = \text{peek}(f(X))$. % each ground term’s priority is its own curr. val.
- $\text{priority} (#f(X)) = \text{infinity}$. % but non-ground term f(X) will get immed. updates
Other features in the works

- Gensyms (several uses)
- Type system (type = “simple” subset of all terms)
- Modes (for query plans, foreign functions, storage)
- Declarations about storage
  (require static analysis of modes & finer-grained types)
- Declarations about execution
Some More Examples

- Shortest paths
- Neural nets
- Vector-space IR
- FSA intersection
- Generalized A* parsing
- n-gram smoothing
- Arc consistency
- Game trees
- Edit distance
Path-finding in Prolog

- `pathto(1).` % the start of all paths
- `pathto(V) :- edge(U,V), pathto(U).`

When is the query `pathto(14)` really inefficient?

- What’s wrong with this swapped version?
  `pathto(V) :- pathto(U), edge(U,V).`
Shortest paths in Dyna

- **Single source:**
  - \( \text{path}(\text{start}) \) \( \text{min} = 0. \)
  - \( \text{path}(W) \) \( \text{min} = \text{path}(V) + \text{edge}(V,W). \)

- **All pairs:**
  - \( \text{path}(U,U) \) \( \text{min} = 0. \)
  - \( \text{path}(U,W) \) \( \text{min} = \text{path}(U,V) + \text{edge}(V,W). \)

  - This hint gives Dijkstra’s algorithm (pqueue):
    - \( \text{priority}(\text{path}(V) \text{min} = \text{Delta}) = \text{Delt} + \text{heuristic}(V). \)
    - Must also declare that \( \text{path}(V) \) has converged as soon as it pops off the priority queue; this is true if heuristic is admissible.

A* can change \( \text{min} = \) to \( += \) to sum over paths (e.g., PageRank)
Neural networks in Dyna

- \( \text{out(Node)} = \text{sigmoid(in(Node))}. \)
- \( \text{sigmoid(X)} = \frac{1}{1 + e^{-X}}. \)
- \( \text{in(Node)} += \text{weight(Node,Child)} \times \text{out(Child)}. \)
- \( \text{in(Node)} += \text{input(Node)}. \)
- \( \text{error} += (\text{out(Node)} - \text{target(Node)})^2. \)

value of \( \text{out(y)} \) is not a sum over all its proofs (not distribution semantics)

Backprop is built-in; recurrent neural net is ok
Vector-space IR in Dyna

- \text{bestscore}(\text{Query}) \text{ max } = \text{ score}(\text{Query}, \text{Doc}).
- \text{score}(\text{Query}, \text{Doc}) + = \text{ tf}(\text{Query}, \text{Word}) \times \text{tf}(\text{Doc}, \text{Word}) \times \text{idf}(\text{Word}).
- \text{idf}(\text{Word}) = 1 / \log(\text{df}(\text{Word})).
- \text{df}(\text{Word}) + = 1 \text{ whenever } \text{tf}(\text{Doc}, \text{Word}) > 0.
Intersection of weighted finite-state automata (epsilon-free case)

Here ‘o’ and ‘x’ are infix functors. A and B are dynabases representing FSAs. Define a new FSA called A o B, with states like Q x R.

- (A o B)\rightarrow start = A\rightarrow start x B\rightarrow start.
- (A o B)\rightarrow stop(Q x R) |= A\rightarrow stop(Q) & B\rightarrow stop(R).
- (A o B)\rightarrow arc(Q1 x R1, Q2 x R2, Letter) 
  += A\rightarrow arc(Q1, Q2, Letter) * B\rightarrow arc(R1, R2, Letter).

Computes full cross-product. But easy to fix so it builds only reachable states (magic templates transform). Composition of finite-state transducers is very similar.
n-gram smoothing in Dyna

- These values all update automatically during leave-one-out cross-validation.
- \[ mle\_prob(X,Y,Z) = \frac{\text{count}(X,Y,Z)}{\text{count}(X,Y)}. \]
- \[ \text{smoothed\_prob}(X,Y,Z) = \lambda \times mle\_prob(X,Y,Z) + (1-\lambda) \times mle\_prob(Y,Z). \]
  for arbitrary-length contexts, could use lists

- \[ \text{count\_of\_count}(X,Y,\text{count}(X,Y,Z)) += 1. \]
  - Used for Good-Turing and Kneser-Ney smoothing.
    - E.g., \[ \text{count\_of\_count}(\text{“the”}, \text{“big”}, 1) \] is number of word types that appeared exactly once after “the big.”
Arc consistency (= 2-consistency)

Agenda algorithm ...

Note: These steps can occur in somewhat arbitrary order

X:3 has no support in Y, so kill it off
Y:1 has no support in X, so kill it off
Z:1 just lost its only support in Y, so kill it off

X, Y, Z, T :: 1..3
X #< Y
Y #= Z
T #< Z
X #< T

slide thanks to Rina Dechter (modified)
Arc consistency in Dyna (*AC-4 algorithm*):

- **Axioms** (*alternatively, could define them by rule*):
  - `indomain(Var:Val) := … % define some values true`
  - `consistent(Var:Val, Var2:Val2) := …`
    - Define to be true or false if Var, Var2 are co-constrained.
    - Otherwise, leave undefined (or define as true).
  - For Var:Val to be kept, Val must be in-domain and also not ruled out by any Var2 that cares:
    - `possible(Var:Val) & indomain(Var:Val).`
    - `possible(Var:Val) & supported(Var:Val, Var2).`
  - Var2 cares if it’s co-constrained with Var:Val:
    - `supported(Var:Val, Var2) |= consistent(Var:Val, Var2:Val2) & possible(Var2:Val2).`
Propagating bounds consistency in Dyna

- E.g., suppose we have a constraint $A \#\leq B$ (as well as other constraints on $A$). Then
  - $\text{maxval}(a) \text{ min} = \text{maxval}(b)$. 
    % if B’s max is reduced, then A’s should be too
  - $\text{minval}(b) \text{ max} = \text{minval}(a)$. % by symmetry

- Similarly, if $C+D \#\neq 10$, then
  - $\text{maxval}(c) \text{ min} = 10-\text{minval}(d)$.
  - $\text{maxval}(d) \text{ min} = 10-\text{minval}(c)$.
  - $\text{minval}(c) \text{ max} = 10-\text{maxval}(d)$.
  - $\text{minval}(d) \text{ max} = 10-\text{maxval}(c)$. 
Game-tree analysis

All values represent total advantage to player 1 starting at this board.

- % how good is Board for player 1, if it’s player 1’s move?
  - best(Board) max = stop(player1, Board).
  - best(Board) max = move(player1, Board, NewBoard) + worst(NewBoard).

- % how good is Board for player 1, if it’s player 2’s move?
  - (player 2 is trying to make player 1 lose: zero-sum game)
  - worst(Board) min = stop(player2, Board).
  - worst(Board) min = move(player2, Board, NewBoard) + best(NewBoard).

- % How good for player 1 is the starting board?
  - goal = best(Board) if start(Board).
Edit distance between two strings

4 edits
- clara
- caca

3 edits
- clara
- caca

2
- clara
- caca

3
- clara
- caca

9
- clara
- caca
Edit distance in Dyna on input lists

- \( \text{dist}([], []) = 0. \)
- \( \text{dist}([X|Xs], Ys) \) min = \( \text{dist}(Xs, Ys) + \text{delcost}(X) \).
- \( \text{dist}(Xs, [Y|Ys]) \) min = \( \text{dist}(Xs, Ys) + \text{inscost}(Y) \).
- \( \text{dist}([X|Xs], [Y|Ys]) \) min = \( \text{dist}(Xs, Ys) + \text{substcost}(X, Y) \).
- \( \text{substcost}(L, L) = 0. \)

- \( \text{result} = \text{align}([“c”, “l”, “a”, “r”, “a”], [“c”, “a”, “c”, “a”]). \)
Edit distance in Dyna on input lattices

- dist(S,T) min = dist(S,T,Q,R) + S→final(Q) + T→final(R).

- dist(S,T, S→start, T→start) min = 0.

- dist(S,T, I2, J) min = dist(S,T, I, J) + S→arc(I,I2,X) + delcost(X).
  - dist(S,T, I, J2) min = dist(S,T, I, J) + T→arc(J,J2,Y) + inscost(Y).
  - dist(S,T, I2,J2) min = dist(S,T, I, J) + S→arc(I,I2,X) + S→arc(J,J2,Y) + substcost(X,Y).

- substcost(L,L) = 0.

- result = dist(lattice1, lattice2).
- lattice1 = { start=state(0).
  arc(state(0),state(1),“c”)=0.3.
  arc(state(1),state(2),“l”)=0. ... final(state(5)). }
Generalized A* parsing (CKY)

- % Get Viterbi outside probabilities.
- % Isomorphic to automatic differentiation (reverse mode).
- outside(goal) = 1.
- outside(Body) max= outside(Head)
  whenever $rule(Head max= Body)$.
- outside(phrase B) max= (*phrase A) * outside(&(A*B)).
- outside(phrase A) max= outside(&(A*B)) * (*phrase B).

- % Prioritize by outside estimates from coarsened grammar.
- $priority(phrase P) = (*P) * outside(coarsen(P))$.
- $priority(phrase P) = 1$ if P==coarsen(P).
  % can't coarsen any further
Generalized A* parsing (CKY)

- % coarsen nonterminals.
  - coa("PluralNoun") = "Noun".
  - coa("Noun") = "Anything".
  - coa("Anything") = "Anything". ... 

- % coarsen phrases.
  - coarsen(&phrase(X,I,J)) = &phrase(coa(X),I,J).

- % make successively coarser grammars
  - % each is an admissible estimate for the next-finer one.
    - coarsen(rewrite(X,Y,Z)) = rewrite(coa(X),coa(Y),coa(Z)).
    - coarsen(rewrite(X,Word)) = rewrite(coa(X),Word).
    - *coarsen(Rule) max= Rule.
      - i.e., Coarse max= Rule whenever Coarse=coarsen(Rule).
Iterative update (EM, Gibbs, BP, …)

- $a := \text{init}_a$.
- $a := \text{updated}_a(b)$. % will override once $b$ is proved
- $b := \text{updated}_b(a)$.
Lightweight information interchange?

- Easy for Dyna terms to represent:
  - XML data (Dyna types are analogous to DTDs)
  - RDF triples (semantic web)
  - Annotated corpora
  - Ontologies
  - Graphs, automata, social networks

- Also provides facilities missing from semantic web:
  - Queries against these data
  - State generalizations (rules, defaults) using variables
  - Aggregate data and draw conclusions
    - Keep track of provenance (backpointers)
    - Keep track of confidence (weights)

- Dynabase = deductive database in a box
  - Like a spreadsheet, but more powerful, safer to maintain, and can communicate with outside world
One Execution Strategy
(forward chaining)
How you build a system ("big picture" slide)

Propagate updates from right-to-left through the equations. a.k.a. "agenda algorithm" "forward chaining" "bottom-up inference" "semi-naive bottom-up"

\[ \beta_x(i,k) = \sum_{0 \leq i < j < k \leq n} \beta_y(i,j) \beta_z(j,k) \]
\[ p(N_x \rightarrow N_y N_z | N_x) \]

... for width from 2 to n
for i from 0 to n
for j from i+1 to k-1
...
Bottom-up inference

agenda of pending updates

rules of program

chart of derived items with current values

we updated np(3,5); what else must therefore change?

If np(3,5) hadn’t been in the chart already, we would have added it.
How you build a system ("big picture" slide)

cool model

practical equations

$$\beta_x(i, k) = \sum_{0 \leq i < j \leq k \leq n} \beta_y(i, j) \beta_z(j, k) \quad p(N_x \rightarrow N_y)$$

... 

pseudocode (execution order)

for width from 2 to n
for i from 0 to n-width
  k = i+width
  for j from i+1 to k-1
    ...

What's going on under the hood?

tuned C++ implementation (data structures, etc.)
Compiler provides …

**agenda of pending updates**

**efficient priority queue**

- `s(I,K) += np(I,J) * vp(J,K)`
- `np(3,5) += 0.3`

**rules of program**

- hard-coded pattern matching

**efficient storage of terms**

(given static type info)

- implicit storage,
- “symbiotic” storage,
- various data structures,
- support for indices,
- stack vs. heap, …)

**automatic indexing**

for O(1) lookup

- chart of derived items with current values

**copy, compare, & hash**

terms fast, via integerization (interning)
Beware double-counting!

agenda of pending updates

to make another copy of itself

comparing with itself

rules of program

chart of derived items with current values

\[ n(I,K) + n(5,5) + n(5,5) \times n(J,K) + n(5,5) = n(5,5) \]

epsilon constituent
Issues in implementing forward chaining

- Handling non-distributive updates
  - Replacement
    - $p_{\text{max}} = q(X)$. what if $q(0)$ is reduced and it’s the curr max?
  - Retraction
    - $p_{\text{max}} = q(X)$. what if $q(0)$ becomes unprovable (no value)?
  - Non-distributive rules
    - $p_{\text{max}} = q(X)$. adding $\Delta$ to $q(0)$ doesn’t simply add to $p$

- Backpointers (hyperedges in the derivation forest)
  - Efficient storage, or on-demand recomputation

- Information flow between $f(3)$, $f(\text{int } X)$, $f(\text{X})$
Issues in implementing forward chaining

- User-defined priorities
  - $\text{priority}(\text{phrase}(X,I,J)) = -(J-I)$. CKY (narrow to wide)
  - $\text{priority}(\text{phrase}(X,I,J)) = \text{phrase}(X,I,J)$. uniform-cost
  - $\text{priority}(\text{phrase}(X,I,J)) = \text{phrase}(X,I,J)$. $\text{A}^*$
    
    Can we learn a good priority function? (can be dynamic)

- User-defined parallelization
  - $\text{host}(\text{phrase}(X,I,J)) = J$.
    
    Can we learn a host choosing function? (can be dynamic)

- User-defined convergence tests
More issues in implementing inference

- Time-space tradeoffs
  - When to consolidate or coarsen updates?
  - When to maintain special data structures to speed updates?
  - Which queries against the memo table should be indexed?

- On-demand computation (backward chaining)
  - Very wasteful to forward-chain everything!
  - Query planning for on-demand queries that arise
  - Selective or temporary memoization
  - Mix forward- and backward-chaining (a bit tricky)

*Can we choose good mixed strategies & good policies?*
Parameter training

- Maximize some objective function.
- Use Dyna to compute the function.
- **Then how do you differentiate it?**
  - ... for gradient ascent, conjugate gradient, etc.
  - ... gradient of log-partition function also tells us the expected counts for EM

- Two approaches supported:
  - Tape algorithm – remember agenda order and run it “backwards.”
  - Program transformation – **automatically** derive the “outside” formulas.
Automatic differentiation via the gradient transform

- \[ a += b \times c. \quad \Rightarrow \quad g(b) += g(a) \times c. \]
- \[ g(c) += b \times g(a). \]

Now \( g(x) \) denotes \( \frac{\partial f}{\partial x} \), \( f \) being the objective function.

- **Examples:**
  - Backprop for neural networks
  - Backward algorithm for HMMs and CRFs
  - Outside algorithm for PCFGs

- Can also get expectations, 2\textsuperscript{nd} derivs, etc.

Dyna implementation also supports “tape”-based differentiation.
How fast was the prototype version?

- It used “one size fits all” strategies
- Asymptotically optimal, but:
  - 4 times slower than Mark Johnson’s inside-outside
  - 4-11 times slower than Klein & Manning’s Viterbi parser
  - 5-6x speedup not too hard to get
Are you going to make it faster? (yup!)

- Static analysis
- Mixed storage strategies
  - store X in an array
  - store Y in a hash
- Mixed inference strategies
  - don’t store Z (compute on demand)
- Choose strategies by
  - User declarations
  - Automatically by execution profiling
More on Program Transformations
Program transformations

- An optimizing compiler would like the freedom to radically rearrange your code.

- Easier in a declarative language than in C.
  - Don’t need to \textit{reconstruct} the source program’s intended semantics.
  - Also, source program is much shorter.

- \textbf{Search problem (open):} Find a good sequence of transformations (helpful on a given workload).
Variable elimination via a folding transform

- Undirected graphical model:

- goal max = f1(A,B) * f2(A,C) * f3(A,D) * f4(C,E) * f5(D,E).

- tempE(C,D) max = f4(C,E) * f5(D,E).

- tempE(C,D) max = f4(C,E) * f5(D,E).

   to eliminate E, join constraints mentioning E, and project E out

figure thanks to Rina Dechter
Variable elimination via a **folding** transform

- Undirected graphical model:

  ![Undirected Graphical Model](image)

  - goal max = \( f_1(A,B) \times f_2(A,C) \times f_3(A,D) \times \text{tempE}(C,D) \).
  - tempD(A,C) max = \( f_3(A,D) \times \text{tempE}(C,D) \).
  - \( \text{tempD}(A,C) \) max = \( f_3(A,D) \times \text{tempE}(C,D) \). to eliminate D, join constraints mentioning D, and project D out
  - \( \text{tempE}(C,D) \) max = \( f_4(C,E) \times f_5(D,E) \).
Variable elimination via a folding transform

- Undirected graphical model:

- goal: \( \max = f_1(A,B) \times f_2(A,C) \times \text{tempD}(A,C). \)
  \[ \text{tempC}(A) \]

- \( \text{tempC}(A) \max = f_2(A,C) \times \text{tempD}(A,C). \)

- \( \text{tempD}(A,C) \max = f_3(A,D) \times \text{tempE}(C,D). \)

- \( \text{tempE}(D,C) \max = f_4(C,E) \times f_5(D,E). \)

figure thanks to Rina Dechter
Variable elimination via a folding transform

- Undirected graphical model:

- goal max = tempC(A) * f1(A,B).
- tempB(A) max = f1(A,B).
- tempC(A) max = f2(A,C) * tempD(A,C).
- tempD(A,C) max = f3(A,D) * tempE(C,D).
- tempE(C,D) max = f4(C,E) * f5(D,E).

figure thanks to Rina Dechter
Variable elimination via a folding transform

- Undirected graphical model:

  - goal max = tempC(A) * tempB(A).
  - tempB(A) max = f1(A,B).
  - tempC(A) max = f2(A,C) * tempD(A,C).
  - tempD(A,C) max = f3(A,D) * tempE(C,D).
  - tempE(C,D) max = f4(C,E) * f5(D,E).

  could replace max = with + = throughout, to compute partition function Z instead of MAP

figure thanks to Rina Dechter
Grammar specialization as an unfolding transform

- \( \text{phrase}(X,I,J) \ += \text{rewrite}(X,Y,Z) \times \text{phrase}(Y,I,Mid) \times \text{phrase}(Z,Mid,J) \).

- \( \text{rewrite}("s","np","vp") \ += 0.7 \). unfolding
  \( \text{phrase}("s",I,J) \ += 0.7 \times \text{phrase}("np",I,Mid) \times \text{phrase}("vp",Mid,J) \).

  \( s(I,J) \ += 0.7 \times np(I,Mid) \times vp(Mid,J) \).

  term flattening (actually handled implicitly by subtype storage declarations)
On-demand computation via a “magic templates” transform

- a :- b, c. \(\Rightarrow\) a :- magic(a), b, c.
  - magic(b) :- magic(a).
  - magic(c) :- magic(a), b.

- Examples:
  - Earley’s algorithm for parsing
  - Left-corner filter for parsing
  - On-the-fly composition of FSTs

- The weighted generalization turns out to be the “generalized A*” algorithm (coarse-to-fine search).
Speculation transformation

*(generalization of folding)*

- Perform some portion of computation speculatively, before we have all the inputs we need; a kind of lifting
  - Fill those inputs in later

- Examples from parsing:
  - Gap passing in categorial grammar
    - Build an S/NP (a sentence missing its direct object NP)
  - Transform a parser so that it preprocesses the grammar
    - E.g., unary rule closure or epsilon closure
    - Build phrase("np", I, K) from a phrase("s", I, K) we don’t have yet (so we haven’t yet chosen a particular I, K)
  - Transform lexical context-free parsing from $O(n^5) \rightarrow O(n^3)$
    - Add left children to a constituent we don’t have yet (without committing to how many right children it will have)
    - Derive Eisner & Satta (1999) algorithm
Summary

- AI systems are too hard to write and modify.
  - Need a new layer of abstraction.

- Dyna is a language for computation (no I/O)
  - Simple, powerful idea:
    Define values from other values by weighted logic programming.

- Compiler supports many implementation strategies
  - Tries to abstract and generalize many tricks
  - Fitting a strategy to the workload is a great opportunity for learning!
  - Natural fit to fine-grained parallelization
  - Natural fit to web services
Dyna contributors!

- **Prototype** (available):
  - Eric Goldlust (core compiler), Noah A. Smith (parameter training), Markus Dreyer (front-end processing), David A. Smith, Roy Tromble, Asheesh Laroia

- **All-new version** (under design/development):
  - Nathaniel Filardo (core compiler), Wren Ng Thornton (type system), Jay Van Der Wall (source language parser), John Blatz (transformations and inference), Johnny Graettinger (early design), Eric Northup (early design)

- **Dynasty hypergraph browser** (usable):
  - Michael Kornbluh (initial version), Gordon Woodhull (graph layout), Samuel Huang (latest version), George Shafer, Raymond Buse, Constantinos Michael