Dyna Code Examples for AI

Jason Eisner and Nathaniel W. Filardo (2011). **Dyna: Extending Datalog for modern AI.** In Tim Furche, Georg Gottlob, Giovanni Grasso, Oege de Moor, and Andrew Sellers (eds.), *Datalog 2.0.* Lecture Notes in Computer Science.

AI Algorithms in Conventional Languages

Package	Files	SLOC	Language	Application area
SRILM	285	48967	C++	Language modeling
Charniak parser	266	42464	C++	Parsing
Stanford parser	417	134824	Java	Parsing
cdec	178	21265	C++	Machine translation
Joshua	486	68160	Java	Machine translation
MOSES	351	37703	C++	Machine translation
GIZA++	122	15958	C++	Bilingual alignment
OpenFST	157	20135	C++	Weighted FSAs & FSTs
NLTK	200	46256	Python	NLP education
HTK	111	81596	C	Speech recognition
MALLET	620	77155	Java	Conditional Random Fields
GRMM	90	12926	Java	Graphical model add-on
Factorie	164	12139	Scala	Graphical models

A selection of popular NLP and machine learning systems (top) and toolkits (bottom). Statistics were generated from the most recent stable release as of this writing using SLOCCount [8].

Relax Datalog's restrictions—driven by use cases

- **Drop flatness**: allow terms to encode lists, categorial nonterminals, attribute-value trees, etc.
- **Drop range restriction**: allow default and non-monotonic reasoning, and general function definitions; simplifies certain source-to-source program transformations [4].
- **Drop stratification**: allow non-stratified design patterns
  - dynamic programming (shortest paths, pathsum)
  - recurrent neural networks
  - message passing
  - iterative optimization

As a result,

- execution might not terminate (Turing-complete)
- must reason over non-ground terms (needs mode analysis)
- one program may have multiple models

A General Feed-Forward Neural Network

```
sigmoid(X) = 1 / (1 + exp(-X)).
output(Node) = sigmoid(input(Node)).
input(Node) += output(Child) * weight(Child,Node).
error += (output(Node) - target(Node))**2.
```

Line 1 defines the sigmoid function over all real numbers `X`.
Line 2 applies that function to the *value* of `input(Node)` (evaluated in place).
Line 3 sums over all incoming edges to `Node`. Those edges are simply the `(Child,Node)` pairs for which `weight(Child,Node)` is defined. Additional summands to some of the `input(Node)` items may be supplied at runtime.
Line 4 evaluates error by summing over just those nodes for which `target(Node)` has been defined (i.e., is non-null), presumably the output nodes.

Neural Network Topology for Vision

```
weight(pixel(X+I,Y+J), hidden(X,Y)) = shared_weight(I,J).
```

One layer of a neural network topology for vision, to be used with the previous example. Each hidden node `hidden(X,Y)` is connected to a 5×5 rectangle of input nodes `pixel(X+I,Y+J)` for $\mathtt{I}, \mathtt{J} \in \{-2, -1, 0, 1, 2\}$, using a collection of 25 weights that are reused across spatial positions `(X,Y)`. The `shared_weight(I,J)` items should be defined (non-null) only for $\mathtt{I}, \mathtt{J} \in \{-2, -1, 0, 1, 2\}$. This rule then connects nodes with related names, such as such as `hidden(75,95)` and `pixel(74,97)`.

This rule exploits the fact that the node names are structured objects. By using structured names, we have managed to specify an *infinite* network in a single line (plus 25 weight definitions). Only a finite portion of this network will actually be used by the network above, assuming that the image (the collection of `pixel` items) is finite.

Non-Monotonic Reasoning

```
fly(X) := false.
fly(X) := true if bird(X).
fly(X) := false if penguin(X).
fly(bigbird) := false.
```

An example of non-monotonic reasoning: all birds fly, other than Sesame Street's Big Bird, until such time as they are proved or asserted to be penguins. The `:=` aggregator is sensitive to rule ordering, so that where the later rules apply at all, they override the earlier rules. The first rule is a "default rule" that is not range-restricted: it proves infinitely many items that unify with a pattern (here the very simple pattern `X`).

Non-Monotonic Reasoning: Default Arcs in FSAs

```
arc(q,Letter) := r.      % single default: ρ arc from state q to state r
arc(q,"x") := s.         % override default: on input letter "x", go to s instead

arc(q,Letter) := arc(r,Letter).  % inherited defaults: φ arc from q to r
arc(q,"x") := s.         % override default: on input letter "x", go to s instead
```

At a given state of the automaton, one can concisely specify some default transitions, but then override these defaults in selected cases. The above mimics the special ρ and ϕ arcs supported by the OpenFST toolkit [1].

This concise intensional structure can be exploited directly within an algorithm such as the forward-backward algorithm, the Viterbi algorithm, or automaton intersection.

However, nontrivial rearrangements of the standard algorithm can be needed to avoid materializing all transitions from state q. An efficient implementation of Dyna would have to discover these optimizations.

Probabilistic Context-Free Parsing

```
% A single word is a phrase (given an appropriate grammar rule).
phrase(X,I,J) += rewrite(X,W) * word(W,I,J).
% Two adjacent phrases make a wider phrase (given an appropriate rule).
phrase(X,I,J) += rewrite(X,Y,Z) * phrase(Y,I,Mid)
                                * phrase(Z,Mid,J).
% An phrase of the appropriate type covering the whole sentence is a parse.
goal          += phrase(start_nonterminal,0,length).
```

Probabilistic context-free parsing in Dyna (the "inside algorithm") . . .

Probabilistic Context-Free Parsing

Probabilistic context-free parsing in Dyna (the "inside algorithm").
`phrase(X,I,J)` is provable if there might be a constituent of type `X` from position `I` to position `J` of the input sentence. More specifically, the *value* of `phrase(X,I,J)` is the probability that nonterminal symbol `X` would expand into the substring that stretches from `I` to `J`. It is defined using `+=` to sum over all ways of generating that substring (considering choices of `Y`, `Z`, `Mid`). Thus, `goal` is the probability of generating the input sentence, summing over all parses.

The extensional input consists of a sentence and a grammar.
`word("spring",5,6)=1` means that `"spring"` is the sixth word of the sentence; while `length=30` specifies the number of words.
`rewrite("S","NP","VP")=0.9` means that any copy of nonterminal `S` has a priori probability `0.9` of expanding via the binary grammar production `S` $\rightarrow$ `NP VP`; while `start_nonterminal="S"` specifies the start symbol of the grammar.

Modularity: Parsing (a)

```
phrase(X,I,J) += grammar.rewrite(X,W) * input.word(W,I,J).
phrase(X,I,J) += grammar.rewrite(X,Y,Z) * phrase(Y,I,Mid)
                                        * phrase(Z,Mid,J).
goal          += phrase(grammar.start_nonterminal,
                        0,input.length).
```

A parser as earlier, except that its input items are two dynabases (denoted by `grammar` and `input`) rather than many separate numbers (denoted by `rewrite(...)`, `word(...)`, etc.).

Modularity: Parsing (b)

```
% Specialize (a) into an English-specific parser.
english_parser = new $load("parser").   % parser.dyna is given in (a)
english_parser.grammar = $load("english_grammar").   % given in (c)

% Parse a collection of English sentences by providing different inputs.
doc = $load("document").
parse(K) = new english_parser.          % extend the abstract parser ...
parse(K).input = doc.sentence(K).       % ... with some actual input

% The total log-probability of the document, ignoring sentences for which
% no parse was found.
logprob += log(parse(K).goal).
```

An illustration of how to use the above parser ...

Modularity: Parsing (b)

An illustration of how to use the above parser. This declarative "script" does not specify the serial or parallel order in which to parse the sentences, whether to retain or discard the parses, etc. All dynabases `parse(K)` share the same grammar, so the rule probabilities do not have to be recomputed for each sentence. A good grammar will obtain a comparatively high `logprob`; thus, the `logprob` measure can be used for evaluation or training. (Alternative measures that consider the *correct* parses, if known, are almost as easy to compute.)

Modularity: Parsing (c)

```
% Define the unnormalized probability of the grammar production X → Y Z
% as a product of feature weights.
urewrite(X,Y,Z) *= left_child_weight(X,Y).
urewrite(X,Y,Z) *= right_child_weight(X,Z).
urewrite(X,Y,Z) *= sibling_weight(Y,Z).
urewrite(X,Y,Y) *= twin_weight.  % when the two siblings are identical
urewrite(X,Y,Z) *= 1.            % default in case no features are defined

% Normalize into probabilities that can be used in PCFG parsing:
% many productions can rewrite X but their probabilities should sum to 1.
urewrite(X) += urewrite(X,Y,Z)
                 whenever nonterminal(Y), nonterminal(Z).
rewrite(X,Y,Z) = urewrite(X,Y,Z) / urewrite(X).
```

Constructing a dense grammar for use by the above programs, with probabilities given by a conditional log-linear model . . .

Modularity: Parsing (c)

Constructing a dense grammar for use by the above programs, with probabilities given by a conditional log-linear model. With k grammar nonterminals, this scheme specifies k^3 rule probabilities with only $O(k^2)$ feature weights to be learned from limited data [2]. Just as for neural nets, these weights may be trained on observed data. For example, maximum likelihood estimation would try to maximize the resulting `logprob`.

Schematic Example of Multiple Dynabases

```
three = 3.
e = { pigs += 100.       % we have 100 adult pigs
      pigs += piglets.   % and any piglets we have are also pigs
    }.

f = new e.          % f is a new pigpen φ that inherits all rules of ε
f.pigs    += 20.    % but has 20 extra adult pigs
f.piglets := three. % and exactly three piglets

g = new e.          % g is another new pigpen γ
offspring = g.pigs / three.    % all pigs have babies
g.piglets := offspring.        % who are piglets

transpose(Matrix) = { element(I,J) = Matrix.element(J,I). }.
```

Monte Carlo Methods: Simple Random Walk

```
point(T) = new point(T-1).                          % copy old point
point(T).x += r.dx(T) if r.flip(T) < 0.5.           % adjust x
point(T).y += r.dy(T) if r.flip(T) >= 0.5.          % or y (but never both)
```

A simple random walk in two dimensions, where `r` is a random dynabase. Exactly one of `x` or `y` changes at each time step. Other derived members may change accordingly.

Arc Consistency

```
% For Var:Val to be possible, Val must be in-domain, and
% also supported by each Var2 that is co-constrained with Var.
% The conjunctive aggregator &= is like universal quantification over Var2.
possible(Var:Val) &= in_domain(Var:Val).
possible(Var:Val) &= supported(Var:Val, Var2).

% Var:Val is supported by Var2 only if it is still possible
% for Var2 to take some value that is compatible with Val.
% The disjunctive aggregator |= is like existential quantification over Val2.
supported(Var:Val, Var2)
    |= compatible(Var:Val, Var2:Val2) & possible(Var2:Val2).

% If consistent ever becomes false, we have detected unsatisfiability:
% some variable has no possible value.
non_empty(Var) |= false.                    % default (if there are no possible values)
non_empty(Var) |= possible(Var:Val).  % Var has a possible value
consistent &= non_empty(Var) whenever is_var(Var).
                                        % each Var in the system has a possible value
```

Arc Consistency

Arc consistency for constraint programming [3]. The goal is to rule out some impossible values for some variables, using a collection of unary constraints (`in_domain`) and binary constraints (`compatible`) that are given by the problem and/or tested during backtracking search. The "natural" forward-chaining execution strategy for this Dyna program corresponds to the classical, asymptotically optimal AC-4 algorithm [5].

Variables and constraints can be named by arbitrary terms. `Var:Val` is syntactic sugar for an ordered pair, similar to `pair(Var,Val)` (the `:` has been declared as an infix functor). The program determines whether `possible(Var:Val)`. The user should define `is_var(Var)` as `true` for each variable, and `in_domain(Var:Val)` as `true` for each value `Val` that `Var` should consider. To express a binary constraint between the variables `Var` and `Var2`, the user should define `compatible(Var:Val, Var2:Val2)` to be `true` or `false` for each value pair `Val` and `Val2`, according to whether the constraint lets these variables simultaneously take these values. This ensures that `supported(Var:Val,Var2)` will be `true` or `false` (not null) and so will contribute a conjunct to line 2.

Loopy Belief Propagation

```
% Belief at each variable based on the messages it receives from constraints.
belief(Var:Val) *= message(Con, Var:Val).

% Belief at each constraint based on the messages it receives from variables
% and the preferences of the constraint itself.
belief(Con:Asst) = messages_to(Con:Asst) * constraint(Con:Asst).

% To evaluate a possible assignment Asst to several variables, look at messages
% to see how well each variable Var likes its assigned value Asst.Var.
messages_to(Con:Asst) *= message(Var:(Asst.Var), Con).

% Message from a variable Var to a constraint Con.  Var says that it plausibly
% has value Val if Var independently believes in that value (thanks to other
% constraints, with Con's own influence removed via division).
message(Var:Val, Con) := 1.   % initial value, will be overridden
message(Var:Val, Con) := belief(Var:Val) / message(Con, Var:Val).

% Messages from a constraint Con to a variable Var.
% Con says that Var plausibly has value Val if Con independently
% believes in one or more assignments Asst in which this is the case.
message(Con, Var:Val) += belief(Con:Asst) / message(Var:Val, Con)
                                  whenever Asst.Var == Val.
```

Loopy Belief Propagation

Loopy belief propagation on a factor graph [7, 9]. The constraints together define a Markov Random Field joint probability distribution over the variables. We seek to approximate the marginals of that distribution: at each variable `Var` we will deduce a belief about its value, in the form of relative probabilities of the possible values `Val`. Similarly, at each constraint `Con` over a *set* of variables, we will deduce a belief about the correct *joint assignment* of values to *just those* variables, in the form of relative probabilities of the possible assignments `Asst`.

Assignments are slightly complicated because we allow a single constraint to refer to arbitrarily many variables (in contrast to arc consistency, which assumed binary constraints). A specific assignment is a map from variable names (terms such as `color`, `size`) to their values (e.g., `red`, 3). It is convenient to represent this map as a small sub-dynabase, `Asst`, whose elements are accessed by the `.` operator: for example, `Asst.color == red` and `Asst.size == 3`.

Loopy Belief Propagation

As input, the user must define `constraint` so that each constraint ("factor" or "potential function") gives a non-negative value to each assignment, giving larger values to its preferred assignments. Each variable should be subject to at least one constraint, to specify its domain (analogous to `in_domain` in arc consistency).

A *message* to or from a variable specifies a relative probability for each value of that variable. Since messages are proved circularly from one another, we need to initialize some messages to 1 in order to start propagation; but these initial values are overridden thanks to the `:=` aggregator, which selects its "latest" aggregand and hence prefers the aggregand from line 5 (once defined) to the initial aggregand from line 4. *Note:* For simplicity, this version of the program glosses over minor issues of message normalization and division by 0.

Backtracking Search with Constraint Propagation

```
% Freely choose an unassigned variable nextvar, if any exists.
% For each of its values Val that is still possible after arc consistency,
% create a clone of the current dynabase, called child(Val).
nextvar ?= Var whenever unassigned(Var).          % free choice of nextvar
child(Val) = new $self if possible(nextvar:Val).  % create several extensions

% Further constrain each child(Val) via additional extensional input,
% so that it will only permit value Val for nextvar,
% and so that it will choose a new unassigned variable to assign next.
child(Val).possible(nextvar:Val2) &= (Val==Val2)
                                     whenever possible(nextvar:Val).
child(Val).unassigned(nextvar) &= false.   % nextvar has been assigned

% We are satisfiable if arc consistency has not already proved consistent to be false,
% and also at least one of our children (if we have any) is satisfiable.
consistent &= some_child_consistent.
some_child_consistent |= child(Val).consistent.
        % usually is true or false, but is null at a leaf (since nextvar is null)
```

Backtracking Search with Constraint Propagation

Determining the satisfiability of a set of constraints, using backtracking search interleaved with arc consistency. These rules extend the program of arc consistency—which rules out some impossible values for some variables, and which sometimes detects unsatisfiability by proving that `consistent` is `false`. Here, we strengthen `consistent` with additional conjuncts so that it fully checks for satisfiability. Lines 1–2 choose a single variable `nextvar` (using the "free-choice" aggregator ?=) and guess different values for it in child dynabases. We place constraints into the child at lines 3–4 and read back the result (whether that child is satisfiable) at line 6.

Backtracking Search with Constraint Propagation

Branch and bound: Find a *maximum-scoring* joint assignment to the variables, subject to the constraints. The score of a given assignment is found by summing the `subscore` values (as specified by the user) of the several `Var:Val` pairs in the assignment.

Above, replace `consistent` (a boolean item aggregated by `&=`) by `score` (a real-valued item aggregated by `min=`). Just as `consistent` computes a boolean upper bound on satisfiability, `score` computes a numeric upper bound on the best achievable score:

```
subscore(Var) max= -∞.
subscore(Var) max= subscore(Var:Val) whenever possible(Var:Val).
upper_bound   +=   subscore(Var) whenever is_var(Var).
score         min= upper_bound.
```

Then above, `score` is reduced to the best score actually achieved by any child:

```
score            min= best_child_score.
best_child_score max= child(nextvar:Val).score.
```

Markov Decision Processes

```
% The optimal value function V.
value(State)       max= value(State,Action).

% The optimal action-value function Q.  Note: The value of p(s,a,s')
% is a conditional transition probability, P(s' | s,a).
value(State,Action) += reward(State,Action).
value(State,Action) += γ * p(State,Action,NewState)
                                 * value(NewState).

% The optimal policy function π.  The free-choice aggregator ?= is used
% merely to break ties.
best_action(State)  ?= Action whenever
                         value(State) == value(State,Action).
```

Markov Decision Processes

Finding the optimal policy in an infinite-horizon Markov decision process, using value iteration. The reward and transition probability functions can be sensitive to properties of the states, or to their structured names as earlier. The optimal value of a `State` is the expected total reward that an agent will earn if it follows the optimal policy from that `State` (where the reward at t steps in the future is discounted by a factor of γ^t). The optimal value of a `(State,Action)` pair is the expected total reward that the agent will earn by first taking the given `Action`—thereby earning a specified reward and stochastically transitioning to a new state—and thereafter following the optimal policy to earn further reward.

The mutual recurrence between V and Q interleaves two different aggregators: `max=` treats optimization by the agent, while `+=` computes an expectation to treat randomness in the environment. This "expectimax" strategy is appropriate for acting in a random environment, in contrast to the "minimax" strategy using `max=` and `min=` that is appropriate when acting against an adversarial opponent. The final line with `?=` merely extracts the optimal policy once its value is known.

Weighted Edit Distance

```
% Base case: distance between two empty strings.
dist([],[]) = 0.

% Recursive cases.
dist([X|Xs],    Ys ) min= delete_cost(X)    + dist(Xs,Ys).
dist(   Xs, [Y|Ys]) min= insert_cost(Y)    + dist(Xs,Ys).
dist([X|Xs],[Y|Ys]) min=  subst_cost(X,Y) + dist(Xs,Ys).

% Part of the cost function.
substcost(L,L) = 0.    % cost of 0 to align any letter to itself
```

Weighted Edit Distance

Weighted edit distance between two strings. This example illustrates items whose names are arbitrarily deep terms: each `dist` name encodes two strings, each being an list of letters. As in Prolog, the syntactic sugar `[X|Xs]` denotes a list of length > 0 that is composed of a first element `X` and a remainder list `Xs`.

We pay some cost for aligning the first 0 or 1 letters from one string with the first 0 or 1 letters from the other string, and then recurse to find the total cost of aligning what is left of the two strings. The choice of how many initial letters to align is at lines 2–4: the program tries all three choices and picks the one with the minimum cost. Reuse of recursive subproblems keeps the runtime quadratic. For example, if all costs not shown are 1, then `dist([a,b,c,d], [s,b,c,t,d])` has value 2. This is obtained by optimally choosing the line with `subst_cost(a,s)` at the first recursive step, then `subst_cost(b,b)`, `subst_cost(c,c)`, `insert_cost(t)`, `subst_cost(d,d)`, for a total cost of 1+0+0+1+0.

Conditional Probability Estimation

```
count(X,Y) += 0 whenever is_event(X), is_event(Y).   % default
count(X)   += count(X,Y).
count      += count(X).

% Maximum likelihood estimates
mle_prob(X)   = count(X)   / count.
mle_prob(X,Y) = count(X,Y) / count(Y).

% Good-Turing smoothed estimates [6]
gt_prob(X)   = total_mle_prob(count(X)+1)   / n(count(X)).
gt_prob(X,Y) = total_mle_prob(count(X)+1,Y) / n(count(X),Y).

% Used by Good-Turing: How many events X occurred R times, or
% cooccurred R times with Y, and what is their total probability?
n(R)   += 0.                 n(R)   += 1 whenever R==count(X).
n(R,Y) += 0.                 n(R,Y) += 1 whenever R==count(X,Y).
total_mle_prob(R)   += mle_prob(X)   whenever R==count(X).
total_mle_prob(R,Y) += mle_prob(X,Y) whenever R==count(X,Y).
```

Conditional Probability Estimation

Estimating conditional probabilities $p(x)$ and $p(x \mid y)$, based on counts of x with y. The user can simply increment `count(`x`,`y`)` whenever x is observed together with y, and the probability estimates will update.

The user should also set `is_event(`x`)` to `true` for each possible event x, to ensure that even never-observed events will have a defined count (of 0) and will be allocated some probability; `n(0)` counts the number of never-observed events. The final four lines could be written more concisely; e.g., the first of them as `n(count(X)) += 1.` The final two lines should be optimized [4], e.g., the first of them is equivalent to `total_mle_prob(R) = (R/count)*n(R).`

[1] C. Allauzen, M. Riley, J. Schalkwyk, W. Skut, and M. Mohri.

OpenFST: A general and efficient weighted finite-state transducer library.

In *Proc. of the 12th International Conference on Implementation and Application of Automata*, pages 11–23. Springer-Verlag, 2007.

[2] Taylor Berg-Kirkpatrick, Alexandre Bouchard-Côté, John DeNero, and Dan Klein.

Painless unsupervised learning with features.

In *Proc. of NAACL*, pages 582–590, Los Angeles, California, June 2010. ACL.

[3] Rina Dechter.

Constraint Processing.

Morgan Kaufmann, 2003.

[4] Jason Eisner and John Blatz.

Program transformations for optimization of parsing algorithms and other weighted logic programs.

In Shuly Wintner, editor, *Proc. of FG 2006: The 11th Conference on Formal Grammar*, pages 45–85. CSLI Publications, 2007.

[5] R. Mohr and T.C. Henderson.

Arc and path consistency revised.

Artificial Intelligence, 28:225–233, 1986.

[6] Arthur Nádas.

On Turing's formula for word probabilities.

IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-33(6):1414–1416, December 1985.

[7] Judea Pearl.

Probabilistic reasoning in intelligent systems: networks of plausible inference.

Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[8] David Wheeler.

SLOCcount.

[9] J. S. Yedidia, W.T. Freeman, and Y. Weiss.

Understanding belief propagation and its generalizations.

In *Exploring Artificial Intelligence in the New Millennium*, chapter 8. Science & Technology Books, 2003.