Compiling Comp Ling: Practical Weighted Dynamic Programming and the Dyna Language*

Jason Eisner and **Eric Goldlust** and **Noah A. Smith**
Department of Computer Science / Center for Language and Speech Processing
Johns Hopkins University, Baltimore, MD 21218 USA
{jason,goldlust,nasmith}@cs.jhu.edu

Abstract

Weighted deduction with aggregation is a powerful theoretical formalism that encompasses many NLP algorithms. This paper proposes a declarative specification language, Dyna; gives general *agenda-based* algorithms for computing weights and gradients; briefly discusses Dyna-to-Dyna program transformations; and shows that a first implementation of a Dyna-to-C++ compiler produces code that is efficient enough for real NLP research, though still several times slower than hand-crafted code.

1 Introduction

In this paper, we generalize some modern probabilistic parsing techniques to a broader class of weighted deductive algorithms. Our implemented system encapsulates these implementation techniques behind a clean interface—a small high-level specification language, Dyna, which compiles into C++ classes. This system should help the HLT community to experiment more easily with new models and algorithms.

1.1 Dynamic programming as deduction

The "parsing as deduction" framework (Pereira and Warren, 1983) is now over 20 years old. It provides an elegant notation for specifying a variety of parsing algorithms (Shieber et al., 1995), including algorithms for probabilistic or other semiring-weighted parsing (Goodman, 1999). In the parsing community, new algorithms are often stated simply as a set of deductive inference rules (Sikkel, 1997; Eisner and Satta, 1999).

It is also straightforward to specify other NLP algorithms this way. Syntactic MT models, language models, and stack decoders can be easily described using deductive rules. So can operations on finite-state and infinite-state machines.

*We thank Joshua Goodman, David McAllester, and Paul Ruhlen for useful early discussions; pioneer users Markus Dreyer, David Smith, and Roy Tromble for their feedback and input; John Blatz for discussion of program transformations; and several reviewers for useful criticism. This work was supported by NSF ITR grant IIS-0313193, ONR MURI grant N00014-01-1-0685, and a Hertz Foundation fellowship to the third author. The views expressed are not necessarily endorsed by the sponsors.

1.2 The role of toolkits

One might regard deductive inference as merely a helpful perspective for teaching old algorithms and thinking about new ones, linking NLP to logic and classical AI. Real implementations would then be carefully hand-coded in a traditional language.

That was the view ten years ago of finite-state machines—that FSMs were part of the theoretical backbone of CL, linking the field to the theory of computation. Starting in the mid-1990's, however, finite-state methods came to the center of *applied* NLP as researchers at Xerox, AT&T, Groningen and elsewhere improved the expressive power of FSMs by moving from automata to transducers, adding semiring weights, and developing powerful new regular-expression operators and algorithms for these cases. They also developed software. Karttunen et al. (1996) built an FSM toolkit that allowed construction of morphological analyzers for many languages. Mohri et al. (1998) built a *weighted* toolkit that implemented novel algorithms (e.g., weighted minimization, on-the-fly composition) and scaled up to handle large-vocabulary continuous ASR. At the same time, renewed community-wide interest in shallow methods for information extraction, chunking, MT, and dialogue processing meant that such off-the-shelf FS toolkits became the core of diverse systems used in cutting-edge research.

The weakness of FSMs, of course, is that they are only finite-state. One would like something like AT&T's FSM toolkit that also handles the various formalisms now under consideration for lexicalized grammars, non-context-free grammars, and syntax-based MT—and hold the promise of extending to other formalisms and applications not yet imagined.

We believe that deductive inference should play the role of regular expressions and FSMs, providing the theoretical foundation for such an effort. Many engineering ideas in the field can be regarded, we

Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP), pages 281–290, Vancouver, October 2005.

```
1. :- double item=0.                                          % declares that all item values are doubles, default is 0
2. constit(X,I,K) += rewrite(X,W) * word(W,I,K).              % a constituent is either a word ...
3. constit(X,I,K) += rewrite(X,Y,Z) * constit(Y,I,J) * constit(Z,J,K).  % ... or a combination of two adjacent subconstituents
4. goal += constit("s",0,N) whenever ?ends_at(N).             % a parse is any s constituent that covers the input string
```

Figure 1: A probabilistic CKY parser written in Dyna. Axioms are in boldface.

believe, as ideas for how to specify, transform, or compile systems of inference rules.

2 A Language for Deductive Systems

Any toolkit needs an interface. For example, FS toolkits offer a regular expression language. We propose a simple but Turing-complete language, Dyna, for specifying weighted deductive-inference algorithms. We illustrate it here by example; see `http://dyna.org` for more details and a tutorial.

The short Dyna program in Fig. 1 expresses the inside algorithm for PCFGs (i.e., the probabilistic generalization of CKY recognition). Its 3 **inference rules** schematically specify many *equations*, over an arbitrary number of unknowns. This is possible bcause the unknowns (**items**) have *structured names* (**terms**) such as constit("s",0,3). They resemble typed variables in a C program, but we use **variable** instead to refer to the capitalized identifiers X, I, K, ... in lines 2–4. Each rule gives a **consequent** on the left-hand side of the +=, which can be built by combining the **antecedents** on the right-hand side.[1]

Lines 2–4 are equational schemas that specify how to compute the value of items such as constit("s",0,3) from the values of other items. Using the summation operator +=, lines 2–3 say that for any X, I, and K, constit(X,I,K) is defined by summing over the remaining variables, as $\sum_{W}$ rewrite(X,W)*word(W,I,K) + $\sum_{Y,Z,J}$ rewrite(X,Y,Z)*constit(Y,I,J)*constit(Z,J,K). For example, constit("s",0,3) is a sum of quantities such as rewrite("s", "np", "vp")*constit("np",0,1)*constit("vp",1,3). The whenever operator in line 4 specifies a **side condition** that restricts the set of expressions in the sum (i.e., only when N is the sentence length).

To fully define the system of equations, non-default values (in this case, non-zero values) should be **asserted** for some **axioms** at runtime. (Axioms, shown in bold in Fig. 1, are items that never appear as a consequent.) If the PCFG contains a rewrite rule np $\rightarrow$ Mary with probability $p(\text{Mary} \mid \text{np})$=0.005, the user should assert that rewrite("np", "Mary") has value 0.005. If the input is *John loves Mary*, values of 1 should be asserted for word("John",0,1), word("loves",1,2), word("Mary",2,3), and ends_at(3).

Given the axioms as base cases, the equations in Fig. 1 enable deduction of values for other items. The value of the **theorem** constit("s",0,3) will be the inside probability $\beta_{\mathsf{s}}(0,3)$,[2] and the value of goal will be the total probability of all parses.

If one replaces += by max= throughout, then constit("s",0,3) will accumulate the maximum rather than the sum of these quantities, and goal will accumulate the probability of the *best* parse.

With different input, the same program carries out lattice parsing. Simply assert axioms that correspond to (weighted) lattice arcs, such as word("John",17,50), where 17 and 50 are arbitrary terms denoting states in the lattice. It is also quite straightforward to lexicalize the nonterminals or extend to synchronous grammars.

A related context-free parsing strategy, shown in Fig. 2, is Earley's algorithm. These equations illustrate nested terms such as lists. The side condition in line 2 prevents building any constituent until one has built a left context that calls for it.

3 Relation to Previous Work

There is a large relevant literature. Some of the well-known CL papers, notably Goodman (1999), were already mentioned in section 1.1. Our project has three main points of difference from these.

First, we provide an efficient, scalable, open-source implementation, in the form of a compiler from Dyna to C++ classes. (Related work is in §7.2.) The C++ classes are efficient and easy to use, with statements such as c[rewrite("np",2,3)]=0.005 to assert axiom values into a chart named c (i.e., a deduc-

[1] Much of our notation and terminology comes from logic programming: term, variable, inference rule, antecedent/consequent, assert/retract, axiom/theorem.

[2] That is, the probability that s would stochastically rewrite to the first three words of the input. If this can happen in more than one way, the probability sums over multiple derivations.

```
1. need("s",0) = 1.                                                    % begin by looking for an s that starts at position 0
2. constit(Nonterm/Needed,I,I) += rewrite(Nonterm,Needed) whenever ?need(Nonterm, I).   % traditional predict step
3. constit(Nonterm/Needed,I,K) += constit(Nonterm/cons(W,Needed),I,J) * word(W,J,K).    % traditional scan step
4. constit(Nonterm/Needed,I,K) += constit(Nonterm,cons(X,Needed),I,J) * constit(X/nil,J,K).   % traditional complete step
5. goal += constit("s"/nil,0,N) whenever ?ends_at(N).                  % we want a complete s constituent covering the sentence
6. need(Nonterm,J) += constit(_/cons(Nonterm, _), _,J).                % Note: underscore matches anything (anonymous wildcard)
```

Figure 2: An Earley parser that recovers inside probabilities (Earley, 1970; Stolcke, 1995). The rule np → det n should be encoded as the axiom rewrite("np",cons("det",cons("n",nil))), a nested term. "np"/Needed is the label of a partial np constituent that is still missing the *list* of subconstituents in Needed. need("np",3) is derived if some partial constituent seeks an np subconstituent starting at position 3. As in Fig. 1, lattice parsing comes for free, as does training.

tive database) and expressions like c[goal] to extract the values of the resulting theorems, which are computed as needed. The C++ classes also give access to the proof forest (e.g., the forest of parse trees), and integrate with parameter optimization code.

Second, we fully generalize the agenda-based strategy of Shieber et al. (1995) to the weighted case—in particular supporting a *prioritized* agenda. That allows probabilities to guide the search for the best parse(s), a crucial technique in state-of-the-art context-free parsers.[3] We also give a "reverse" agenda algorithm to compute gradients or outside probabilities for parameter estimation.

Third, regarding weights, the Dyna language is designed to express systems of arbitrary, heterogeneous equations over item values. In previous work such as (Goodman, 1999; Nederhof, 2003), one only specifies the inference rules as unweighted Horn clauses, and then weights are added automatically in a standard way: all values have the same type $\mathcal{W}$, and all rules transform to equations of the form $c \oplus\!= a_1 \otimes a_2 \otimes \cdots \otimes a_k$, where $\oplus$ and $\otimes$ give $\mathcal{W}$ the structure of a semiring.[4] In Dyna one writes these equations explicitly in place of Horn clauses (Fig. 1). Accordingly, *heterogeneous* Dyna programs, to be supported soon by our compiler, will allow items of different types to have values of different types, computed by different aggregation operations over arbitrary right-hand-side expressions. This allows specification of a wider class of algorithms from NLP and elsewhere (e.g., minimum expected loss decoding, smoothing formulas, neural networks, game tree analysis, and constraint programming). Although §4 and §5 have space to present only techniques for the semiring case, these can be generalized.

Our approach may be most closely related to deductive databases, which even in their heyday were apparently ignored by the CL community (except for Minnen, 1996). Deductive database systems permit inference rules that can derive new database facts from old ones.[5] They are essentially declarative logic programming languages (with restrictions or extensions) that are—or could be—implemented using efficient database techniques. Some implemented deductive databases such as CORAL (Ramakrishnan et al., 1994) and LOLA (Zukowski and Freitag, 1997) support aggregation (as in Dyna's +=, log+=, max=, ...), although only "stratified" forms of it that exclude unary CFG rule cycles.[6] Ross and Sagiv (1992) (and in a more restricted way, Kifer and Subrahmanian, 1992) come closest to our notion of attaching aggregable values to terms.

Among deductive or other database systems, Dyna is perhaps unusual in that its goal is not to support transactional databases or *ad hoc* queries, but rather to serve as an abstract layer for specifying an algorithm, such as a dynamic programming (DP) algorithm. Thus, the Dyna program already implicitly or explicitly specifies all queries that will be needed. This allows compilation into a hard-coded C++ implementation. The compiler's job is to support these queries by laying out and indexing the database re-

[3] Previous treatments of weighted deduction have used an agenda only for an unweighted parsing phase (Goodman, 1999) or for finding the single best parse (Nederhof, 2003). Our algorithm works in arbitrary semirings, including non-idempotent ones, taking care to avoid double-counting of weights and to handle side conditions.

[4] E.g., the inside algorithm in Fig. 1 falls into Goodman's framework, with $\langle \mathcal{W}, \oplus, \otimes \rangle = \langle \mathbb{R}_{\geq 0}, +, * \rangle$—the PLUSTIMES semiring. Because $\otimes$ distributes over $\oplus$ in a semiring, computing goal is equivalent to an aggregation over many separate parse trees. That is not the case for heterogeneous programs.

[5] Often they use some variant of the *unweighted* agenda-based algorithm, which is known in that community as "semi-naive bottom-up evaluation."

[6] An unweighted parser was implemented in an earlier version of LOLA (Specht and Freitag, 1995).

lations in memory[7] in a way that resembles hand-designed data structures for the algorithm in question. The compiler has many choices to make here; we ultimately hope to implement feedback-directed optimization, using profiled sample runs on typical data. For example, a sparse grammar should lead to different strategies than a dense one.

4 Computing Theorem Values

Fig. 1 specifies a set of equations but not how to solve them. Any declarative specification language must be backed up by a solver for the class of specifiable problems. In our continuing work to develop a range of compiler strategies for arbitrary Dyna programs, we have been inspired by the CL community's experience in building efficient parsers.

In this paper and in our current implementation, we give only the algorithms for what we call *weighted dynamic programs*, in which all axioms and theorems are variable-free. This means that a consequent may only contain variables that already appear elsewhere in the rule. We further restrict to semiring-weighted programs as in (Goodman, 1999). But with a few more tricks not given here, the algorithms can be generalized to a wider class of heterogeneous weighted logic programs.[8]

4.1 Desired properties

Computation is triggered when the user requests the value of one or more particular items, such as goal. Our algorithm must have several properties in order to substitute for manually written code.

Soundness. The algorithm cannot be guaranteed to terminate (since it is possible to write arbitrary Turing machines in Dyna). However, if it does terminate, it should return values from a valid model of the program, i.e., values that simultaneously satisfy all the equations expressed by the program.

Reasonable completeness. The computation should indeed terminate for programs of interest to the NLP community, such as parsing under a probabilistic grammar—even if the grammar has left recursion, unary rule cycles, or ϵ-productions. This appears to rule out pure top-down ("backward-chaining") approaches.

1. **for** each axiom a, set *agenda*$[a]$:= value of axiom a
2. **while** there is an item a with *agenda*$[a] \neq \mathbf{0}$
3. (* *remove an item from the agenda and move its value to the chart* *)
4. choose such an a
5. Δ := *agenda*$[a]$; *agenda*$[a] := \mathbf{0}$
6. *old* := *chart*$[a]$; *chart*$[a]$:= *chart*$[a] \oplus \Delta$
7. **if** *chart*$[a] \neq$ *old* (* *only propagate actual changes* *)
8. (* *compute new resulting updates and place them on the agenda* *)
9. **for** each inference rule "$c \oplus= a_1 \otimes a_2 \otimes \cdots \otimes a_k$"
10. **for** i from 1 to k
11. **for** each way of instantiating the rule's variables such that $a_i = a$
12. *agenda*$[c] \oplus= \bigotimes_{j=1}^{k} \begin{cases} \textit{old} & \text{if } j < i \text{ and } a_j = a \\ \Delta & \text{if } j = i \\ \textit{chart}[a_j] & \text{otherwise} \end{cases}$
(* *can skip this line if any multiplicand is* $\mathbf{0}$ *)*

Figure 3: Weighted agenda-based deduction in a semiring, without side conditions (see text).

Efficiency. Returning the value of goal should do only as much computation as necessary. To return goal, one may not need to compute the values of *all* items.[9] In particular, finding the best parse should not require finding all parses (in contrast to Goodman (1999) and Zhou and Sato (2003)). Approximation techniques such as pruning and best-first search must also be supported for practicality.

4.2 The agenda algorithm

Our basic algorithm (Fig. 3) is a weighted agenda-based algorithm that works only with rules of the form $c \oplus= a_1 \otimes a_2 \otimes \cdots \otimes a_k$. $\otimes$ must distribute over $\oplus$. Further, the default value for items (line 1 of Fig. 1) must be the semiring's zero element, denoted $\mathbf{0}$.[10]

Agenda-based deduction maintains two indexed data structures: the **agenda** and the **chart**. *chart*$[a]$ stores the current value of item a. The agenda holds future work that arises from assertions or from previous changes to the chart: *agenda*$[a]$ stores an incremental update to be added (using $\oplus$) to *chart*$[a]$ in future. If *chart*$[a]$ or *agenda*$[a]$ is not stored, it is

[7] Some relations might be left unmaterialized and computed on demand, with optional memoization and flushing of memos.

[8] Heterogeneous programs may propagate non-additive updates, which arbitrarily modify one of the inputs to an aggregation. Non-dynamic programs require non-ground items in the chart, complicating both storage and queries against the chart.

[9] This also affects completeness, as it sometimes enables the computation of goal to terminate even if the program as a whole contains some irrelevant non-terminating computation. Even in practical cases, the runtime of computing all items is often prohibitive, e.g., proportional to n^6 or worse for a dense tree-adjoining grammar or synchronous grammar.

[10] It satisfies $x \oplus \mathbf{0} = x, x \otimes \mathbf{0} = \mathbf{0}$ for all x. Also, this algorithm requires $\otimes$ to distribute over $\oplus$. Dyna's semantics requires $\oplus$ to be associative and commutative.

taken to be the default $\mathbf{0}$.

When item a is removed from the agenda, its chart weight is updated by the increment value. This change is then propagated to other items c, via rules of the form $c \oplus\!= \cdots$ with a on the right-hand-side. The resulting changes to c are placed back on the agenda and carried out only later.

The unweighted agenda-based algorithm (Shieber et al., 1995) may be regarded as the case where $\langle \mathcal{W}, \oplus, \otimes \rangle = \langle \{T, F\}, \vee, \wedge \rangle$. It has previously been generalized (Nederhof, 2003) to the case $\langle \mathcal{W}, \oplus, \otimes \rangle = \langle \mathbb{R}_{\geq 0}, \max, + \rangle$. In Fig. 3, we make the natural further generalization to any semiring.

How is this a further generalization? Since $\oplus$ (unlike $\vee$ and max) might not be idempotent, we must take care to avoid erroneous double-counting if the antecedent a combines with, or produces, another copy of itself.[11] For instance, if the input contains ϵ words, line 2 of Fig. 1 may get instantiated as constit("np",5,5) += rewrite("np","np","np") * constit("np",5,5) * constit("np",5,5). This is why we save the old values of *agenda*$[a]$ and *chart*$[a]$ as Δ and *old*, and why line 12 is complex.

4.3 Side conditions

We now extend Fig. 3 to handle Dyna's side conditions, i.e., rules of the form $c \oplus\!=$ *expression* whenever *boolean-expression*. We discuss only the simple side conditions treated in previous literature, which we write as $c \oplus\!= a_1 \otimes a_2 \otimes \cdots \otimes a_{k'}$ whenever $?b_{k'+1}$ & $\cdots$ & $?b_k$. Here, $?b_j$ is true or false according to whether there exists an *unweighted* proof of b_j.

Again, what is new here? Nederhof (2003) considers only max= with a uniform-cost agenda discipline (see §4.5), which guarantees that no item will be removed more than once from the agenda. We wish to support other cases, so we must take care that a second update to a_i will not retrigger rules of which a_i is a side condition.

For simplicity, let us reformulate the above rule as $c \oplus\!= a_1 \otimes a_2 \otimes \cdots \otimes a_{k'} \otimes ?b_{k'+1} \otimes \cdots \otimes ?b_k$, where $?b_i$ is now treated as having value $\mathbf{0}$ or $\mathbf{1}$ (the identity for $\otimes$) rather than false or true respectively.

We may now use Fig. 3, but now any a_j might have the form $?b_j$. Then in line 12, *chart*$[a_j]$ will be *chart*$[?b_j]$, which is defined as $\mathbf{1}$ or $\mathbf{0}$ according to whether *chart*$[b_j]$ is stored (i.e., whether b_j has been derived). Also, if $a_i = ?a$ at line 11 (rather than $a_i = a$), then Δ in line 12 is replaced by $\Delta?$, where we have set $\Delta? :=$ *chart*$[?a]$ at line 5.

4.4 Convergence

Whether the agenda algorithm halts depends on the Dyna program and the input. Like any other Turing-complete language, Dyna gives you enough freedom to write undesirable programs.

Most NLP algorithms do terminate, of course, and this remains true under the agenda algorithm. For typical algorithms, only finitely many different items (theorems) can be derived from a given finite input (set of axioms).[12] This ensures termination if one is doing unweighted deduction with $\langle \mathcal{W}, \oplus, \otimes \rangle = \langle \{T, F\}, \vee, \wedge \rangle$, since the test at line 7 ensures that no item is processed more than once.[13]

The same test ensures termination if one is searching for the best proof or parse with (say) $\langle \mathcal{W}, \oplus, \otimes \rangle = \langle \mathbb{R}_{\geq 0}, \min, + \rangle$, where values are negated log probabilities. Positive-weight cycles will not affect the $\min$. (Negative-weight cycles, however, would correctly cause the computation to diverge; these do not arise with probabilities.)

If one is using $\langle \mathcal{W}, \oplus, \otimes \rangle = \langle \mathbb{R}_{\geq 0}, +, * \rangle$ to compute the total weight of all proofs or parses, as in the inside algorithm, then Dyna must solve a system of nonlinear equations. The agenda algorithm does this by iterative approximation (propagating updates around any cycles in the proof graph until numerical convergence), essentially as suggested by Stolcke (1995) for the case of Earley's algorithm.[14] Again, the computation may diverge.

[11] An agenda update that increases x by 0.3 will increase $r * x * x$ by $r * (0.6x + 0.09)$. Hence, the rule x += $r * x * x$ must propagate a new increase of that size to x, via the agenda.

[12] This holds for all Datalog programs, for instance.

[13] This argument does not hold if Dyna is used to express programs outside the semiring. In particular, one can write instances of SAT and other NP-hard constraint satisfaction problems by using cyclic rules *with negation* over finitely many boolean-valued items (Niemelä, 1998). Here the agenda algorithm can end up flipping values forever between false and true; a more general solver would have to be called in order to find a stable model of a SAT problem's equations.

[14] Still assuming the number of items is finite, one could in principle materialize the system of equations and call a dedicated numerical solver. In some special cases only a linear solver is needed: e.g., for unary rule cycles (Stolcke, 1995), or ϵ-cycles in FSMs (Eisner, 2002).

One can declare the conditions under which items of a particular type (constit or goal) should be treated as having converged. Then asking for the value of goal will run the agenda algorithm not until the agenda is empty, but only until *chart*[goal] has converged by this criterion.

4.5 Prioritization

The order in which items are chosen at line 4 does not affect the soundness of the agenda algorithm, but can greatly affect its speed. We implement the agenda as a priority queue whose priority function may be specified by the user.[15]

Charniak et al. (1998) and Caraballo and Charniak (1998) showed that, when seeking the best parse (using min= or max=), *best-first* parsing can be extremely effective. Klein and Manning (2003a) went on to describe admissible heuristics and an A* framework for parsing. For A* in our general framework, the priority of item a should be an estimate of the value of the best proof of goal that uses a. (This non-standard formulation is carefully chosen.[16]) If so, goal is guaranteed to converge the very first time it is selected from the priority-queue agenda.

Prioritizing "good" items first can also be useful in other circumstances. The inside-outside training algorithm requires one to find all parses, but finding the high-probability parses first allows one to ignore the rest by "early stopping."

In all these schemes (even A*), processing promising items as soon as possible risks having to reprocess them if their values change later. Thus, this strategy should be balanced against the "topological sort" strategy of waiting to process an item until its value has (probably) converged.[17] Ultimately we hope to *learn* priority functions that effectively balance these two strategies (especially in the context of early stopping).

4.6 Matching, indexing, and interning

The crucial work in Fig. 3 occurs in the iteration over instantiated rules at lines 9–11. In practice, we restructure this triply nested loop as follows, where each line retains the variable bindings that result from the unification in the previous line:

9. **for** each antecedent pattern a_i that appears in some program rule r and unifies with a
10. **for** each way of simultaneously unifying r's remaining antecedent patterns $a_1, \ldots a_{i-1}, a_{i+1}, \ldots a_k$ with items that may have non-**0** value in the chart
11. construct r's consequent c *(* all vars are bound *)*

Our implementation of line 9 tests a against all of the antecedent patterns at once, using a tree of simple "if" tests (generated by the Dyna-to-C++ compiler) to share work across patterns. As an example, $a =$ constit("np",3,8) will match two antecedents at line 3 of Fig. 1, but will fail to match in line 4. Because a is variable-free (for DPs), a full unification algorithm is not necessary, even though an antecedent pattern can contain repeated variables and nested subterms.

Line 10 rapidly looks up the rule's other antecedents using *indices* that are automatically maintained on the chart. For example, once constit("np",4,8) has matched antecedent 2 of line 3 of Fig. 1, the compiled code consults a maintained list of the chart constituents that start at position 8 (i.e., items of the form constit(Z,8,K) that have already been derived). Suppose one of these is constit("vp",8,15): then the code finds the rule's remaining antecedent by consulting a list of items of the form rewrite(X,"np","vp"). That leads it to construct consequents such as constit("s",4,15) at line 11.

By default, equal terms are represented by equal pointers. While this means terms must be "interned" when constructed (requiring hash lookup), it enforces structure-sharing and allows any term to be rapidly copied, hashed, or equality-tested without dereferencing the pointer.[18]

Each of the above paragraphs conceals many decisions that affect runtime. This presents future opportunities for feedback-directed optimization, where profiled runs on typical data influence the compiler.

[15] At present by writing a C++ function; ultimately within Dyna, by defining items such as priority(constit("s",0,3)).

[16] It is correct for proofs that incorporate two copies of a's value, or—more important—no copies of a's value because a is a side condition. Thus, it recognizes that a low-probability item must have *high* priority if it could be used as a side condition in a higher-probability parse (though this cannot happen for the side conditions derived by the magic templates transformation (§6)). Note also that a's own value (Nederhof, 2003) might not be an optimistic estimate, if negative weights are present.

[17] In parsing, for example, one often processes narrower constituents before wider ones. But such strategies do not always exist, or break down in the presence of unary rule cycles, or cannot be automatically found. Goodman's (1999) strategy of building all items and sorting them before computing any weights is wise only if one genuinely wants to build all items.

[18] The compiled code provides garbage collection on the terms; this is important when running over large datasets.

5 Computing Gradients

The value of goal is a *function* of the axioms' values. If the function is differentiable, we may want to get its gradient with respect to its parameters (the axiom values), to aid in numerically optimizing it.

5.1 Gradients by symbolic differentiation

The gradient computation can be derived from the original by a program transformation. For each item a in the original program—in particular, for each axiom—the new program will also compute a new item $\mathsf{g}(a)$, whose value is $\partial \mathsf{goal}/\partial a$.

Thus, given weighted axioms, the new program computes both goal and ∇goal. An optimization algorithm such as conjugate gradient can use this information to tune the axiom weights to maximize goal. An alternative is the EM algorithm (Dempster et al., 1977) for probabilistic generative models such as PCFGs. Luckily the same program serves, since for such models, the E count (expected count) of an item a can be found as $a \cdot \mathsf{g}(a)/\mathsf{goal}$. In other words, the inside-outside algorithm has the same structure as computing the function and its gradient.

The GRADIENT transformation is simple. For example,[19] given a rule c += $a_1 * a_2 * \cdots * a_{k'}$ whenever $?b_{k'+1}$ & $\cdots$ & $?b_k$, we add a new rule $\mathsf{g}(a_i)$ += $\mathsf{g}(c) * a_1 * \cdots * a_{i-1} * a_{i+1} * \cdots * a_{k'}$ whenever $?a_i$, for each $i = 1, 2, ..., k'$. (The original rule remains, since we need inside values to compute outside values.) This strategy for computing the gradient $\partial \mathsf{goal}/\partial a$ via the chain rule is an example of automatic differentiation in the reverse mode (Griewank and Corliss, 1991), known in the neural network community as back-propagation.

5.2 Gradients by back-propagation

However, what if goal might be computed only approximately, by early stopping before convergence (§4.5)? To avoid confusing the optimizer, we want the *exact* gradient of the *approximate* function.

To do this, we "unwind" the computation of goal, *undoing* the value updates while building up the gradient values. The idea is to differentiate an "unrolled" version of the original computation (Williams and Zipser, 1989), in which an item at time t is considered to be a different variable (possibly with different value) than the same item at time $t+1$. The reverse pass must recover earlier values. Our somewhat tricky algorithm is shown in Fig. 4.

1. **for** each a, *gchart*$[a] := 0$ and *gagenda*$[a] := 0$
   (* *respectively hold* $\partial \mathsf{goal}/\partial$*chart*$[a]$ *and* $\partial \mathsf{goal}/\partial$*agenda*$[a]$ **)*
2. *gchart*[goal] := 1
3. **for** each $\langle a, \Delta, old \rangle$ triple that was considered at line 8 of Fig. 3, but in the *reverse order* (* Δ *is agenda*$[a]$ **)*
4. $\Gamma :=$ *gchart*$[a]$ (* *will accumulate gagenda[a] here *)*
5. **for** each inference rule "c += $a_1 * a_2 * \cdots * a_k$"
6. **for** i from 1 to k
7. **for** each way of instantiating the rule's variables such that $a_i = a$
8. **for** h from 1 to k such that a_h is not a side cond.
   (* *find* $\partial \mathsf{goal}/\partial$*agenda*$[c] \cdot \partial$*agenda*$[c]/\partial(a_h$ *factor) *)*
9. $\gamma := \prod_{j=1}^{k} \begin{cases} \textit{gagenda}[c] & \text{if } j = h \\ \textit{old} & \text{if } j \neq h \text{ and } j < i \text{ and } a_j = a \\ \Delta & \text{if } j \neq h \text{ and } j = i \\ \textit{chart}[a_j] & \text{otherwise} \end{cases}$
10. **if** $h \neq i$ **then** *gchart*$[a_h]$ += γ
11. **if** $h \leq i$ and $a_h = a$ **then** Γ += γ
12. *gagenda*$[a] := \Gamma$
13. *chart*$[a] :=$ *old*
14. **return** *gagenda*[a] for each axiom a

Figure 4: An efficient algorithm for computing ∇goal (even when goal is an early-stopping approximation), specialized to the case $\langle \mathcal{W}, \oplus, \otimes \rangle = \langle \mathbb{R}, +, * \rangle$. The proof is suppressed for lack of space.

At line 3, a stack is needed to remember the sequence of $\langle a, \textit{old}, \Delta \rangle$ triples from the original computation.[20] It is a more efficient version of the "tape" usually used in automatic differentiation. For example, it uses $O(n^2)$ rather than $O(n^3)$ space for the CKY algorithm. The trick is that Fig. 3 does not record all its computations, but only its sequence of items. Fig. 4 then re-runs the inference rules to reconstruct the computations in an acceptable order.

This method is a generalization of Eisner's (2001) prioritized forward-backward algorithm for infinite-state machines. As Eisner (2001) pointed out, the tape created on the first forward pass can also be used to speed up later passes (i.e., after the numerical optimizer has adjusted the axiom weights).[21]

[19] More generally, $\mathsf{g}(a_i) = \partial \mathsf{goal}/\partial a_i = \sum_c \partial \mathsf{goal}/\partial c \cdot \partial c/\partial a_i = \sum_c \mathsf{g}(c) \cdot \partial c/\partial a_i$ by the chain rule.

[20] If one is willing to risk floating-point error, then one can store only $\langle a, \textit{old} \rangle$ on the stack and recover Δ as *chart*$[a] -$ *old*. Also, *agenda*$[a]$ and *gagenda*$[a]$ can be stored in the same location, as they are only used during the forward and the backward pass, respectively.

[21] In brief, a later forward pass that chooses a at Fig. 3, line 4 according to the recorded tape order (1) is faster than using a priority queue, (2) avoids ordering-related discontinuities in the objective function as the axiom weights change, (3) can prune by skipping useless updates a that scarcely affected goal (e.g.,

5.3 Parameter estimation

To support parameter training using these gradients, our implementation of Dyna includes a training module, DynaMITE. DynaMITE supports the EM algorithm (and many variants), supervised and unsupervised training of log-linear ("maximum entropy") models using quasi-Newton methods, and smoothing-parameter tuning on development data. As an object-oriented C++ library, it also facilitates rapid implementation of new estimation techniques (Smith and Eisner, 2004; Smith and Eisner, 2005).

6 Program Transformations

Another interest of Dyna is that its high-level specifications can be manipulated by mechanical source-to-source program transformations. This makes it possible to derive new algorithms from old ones. §5.1 already sketched the *gradient* transformation for finding ∇goal. We note a few other examples.

Bounding transformations generate a new program that computes upper or lower bounds on goal, via generic bounding techniques (Prieditis, 1993; Culberson and Schaeffer, 1998). The A* heuristics explored by Klein and Manning (2003a) can be seen as resulting from bounding transformations.

With John Blatz, we are also exploring transformations that can result in asymptotically more efficient computations of goal. Their unweighted versions are well-known in the logic programming community (Tamaki and Sato, 1984; Ramakrishnan, 1991). *Folding* introduces new intermediate items, perhaps exploiting the distributive law; applications include parsing speedups such as (Eisner and Satta, 1999), as well as well-known techniques for speeding up multi-way database joins, constraint programming, or marginalization of graphical models. *Unfolding* eliminates items; it can be used to specialize a parser to a particular grammar and then to eliminate unary rules. *Magic templates* introduce top-down filtering into the search strategy and can be used to derive Earley's algorithm (Minnen, 1996), to introduce left-corner filters, and to restrict FSM constructions to build only accessible states.

Finally, there are low-level optimizations. *Term* transformations restructure terms to change their layout in memory. We are also exploring the introduction of declarations that control which items use the agenda or are memoized in the chart. This can be used to support lazy or "on-the-fly" computation (Mohri et al., 1998) and asymptotic space-saving tricks (Binder et al., 1997).

constituents not in any good parse) by consulting *gagenda*$[a]$ values that the previous backward pass can have written onto the tape (overwriting Δ or *old*).

7 Usefulness of the Implementation

7.1 Applications

The current Dyna compiler has proved indispensable in our own recent projects, in the sense that we would not have attempted many of them without it.

In some cases, we were experimenting with genuinely new algorithms not supported by any existing tool, as in our work on dependency-length-limited parsing (Eisner and Smith, 2005b) and loosely syntax-based machine translation (Eisner and D. Smith, 2005). (Dyna would have been equally helpful in the first author's earlier work on new algorithms for lexicalized and CCG parsing, syntactic MT, transformational syntax, trainable parameterized FSMs, and finite-state phonology.)

In other cases (Smith and Eisner, 2004; Smith and Smith, 2004; Smith et al., 2005), Dyna let us quickly replicate, tweak, and combine useful techniques from the literature. These techniques included unweighted FS morphology, conditional random fields (Lafferty et al., 2001), synchronous parsers (Wu, 1997; Melamed, 2003), lexicalized parsers (Eisner and Satta, 1999),[22] partially supervised training à la (Pereira and Schabes, 1992),[23] and grammar induction (Klein and Manning, 2002). These replications were easy to write and extend, and to train via §5.2.

7.2 Experiments

We compared the current Dyna compiler to hand-built systems on a variety of parsing tasks. These problems were chosen not for their novelty or interesting structure, but for the availability of existing well-tuned implementations.

Best parse. We compared a Dyna CFG parser to the Java parser of Klein and Manning (2003b),[24]

[22] Markus Dreyer's reimplementation of the complex Collins (1999) parser uses under 30 lines of Dyna.

[23] For example, lines 2–3 of Fig. 1 can be extended with whenever permitted(X,I,K).

[24] Neither uses heuristics from Klein and Manning (2003a).

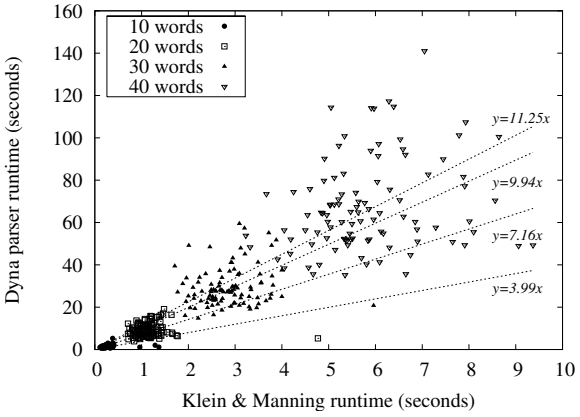

Figure 5: Dyna CKY parser vs. Klein & Manning hand-built parser, comparing runtime.

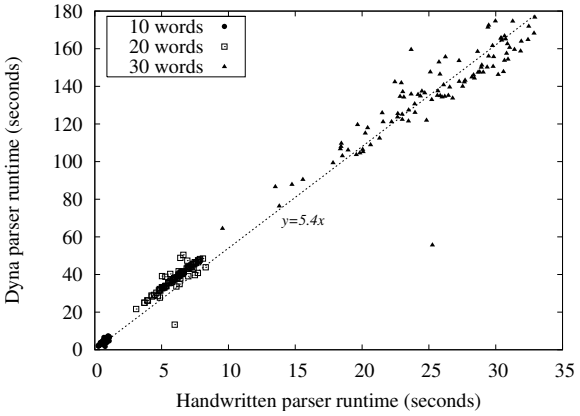

Figure 6: Dyna CKY parser vs. C++PARSE, a similar hand-built parser. The implementation differences amount to storage and indexing and give a consistent 5-fold speedup.

on the same grammar. Fig. 5 shows the results. Dyna's disadvantage is greater on longer sentences—probably because its greater memory consumption results in worse cache behavior.[25]

We also compared a Dyna CKY parser to our own hand-built implementation, C++PARSE. C++PARSE is designed like the Dyna parser but includes a few storage and indexing optimizations that Dyna does not yet have. Fig. 6 shows the 5-fold speedup from these optimizations on binarized-Treebank parsing with a large 119K-rule grammar. The sharp diagonal indicates that C++PARSE is simply a better-tuned version of the Dyna parser.

These optimizations and others are now being incorporated into the Dyna compiler, and are expected

[25]Unlike Java, Dyna does not yet decide automatically when to perform garbage collection. In our experiment, garbage collection was called explicitly after each sentence and counted as part of the runtime (typically 0.25 seconds for 10-word sentences, 5 seconds for 40-word sentences).

	99%		99.99%	
uniform	89.3	(4.5)	90.3	(4.6)
after 1 EM iteration	82.9	(6.8)	85.2	(6.9)
after 2 EM iterations	77.1	(8.4)	79.1	(8.3)
after 3 EM iterations	71.6	(9.4)	73.7	(9.5)
after 4 EM iterations	66.8	(10.0)	68.8	(10.2)
after 5 iterations	62.9	(10.3)	65.0	(10.5)

Table 1: Early stopping. Each row describes a PCFG at a different stage of training; later PCFGs are sharper. The table shows the percentage of agenda runtime (mean across 1409 sentences, and standard deviation) required to get within 99% or 99.99% of the true value of goal.

to provide similar speedups, putting Dyna's parser in the ballpark of the Klein & Manning parser. Importantly, these improvements will speed up existing Dyna programs through recompilation.

Inside parsing. Johnson (2000) provides a C implementation of the inside-outside algorithm for EM training of PCFGs. We ran five iterations of EM on the WSJ10 corpus[26] using the Treebank grammar from that corpus. Dyna took 4.1 times longer.

Early stopping. An advantage of the weighted agenda discipline (§4.2) is that, with a reasonable priority function such as an item's inside probability, the inside algorithm can be stopped early with an estimate of goal's value. To measure the goodness of this early estimate, we tracked the progression of goal's value as each sentence was being parsed. In most instances, and especially after more EM iterations, the estimate was very tight long before all the weight had been accumulated (Table 1). This suggests that early stopping is a useful training speedup.

PRISM. The implemented tool most similar to Dyna that we have found is PRISM (Zhou and Sato, 2003), a probabilistic Prolog with efficient tabling and compilation. PRISM inherits expressive power from Prolog but handles only probabilities, not general semirings (or even side conditions).[27] In CKY parsing tests, PRISM was able to handle only a small fraction of the Penn Treebank ruleset (2,400 high-probability rules) and tended to crash on long sentences. Dyna is designed for real-world use: it consistently parses over 10× faster than PRISM and scales to full-sized problems.

IBAL (Pfeffer, 2001) is an elegant and powerful language for probabilistic modeling; it generalizes Bayesian networks in interesting ways.[28] Since

[26]Sentences with ≤10 words, stripping punctuation.

[27]Thus it can handle a subset of the cases described by Goodman (1999), again by building the whole parse forest.

[28]It might be possible to implement IBAL in Dyna (Pfeffer,

PCFGs and marginalization can be succinctly expressed in IBAL, we attempted a performance comparison on the task of the inside algorithm (Fig. 1). Unfortunately, IBAL's algorithm appears not to terminate if the PCFG contains any kind of recursion reachable from the start symbol.

8 Conclusions

Weighted deduction is a powerful theoretical formalism that encompasses many NLP algorithms (Goodman, 1999). We have given a bottom-up "inside" algorithm for general semiring-weighted deduction, based on a prioritized agenda, and a general "outside" algorithm that correctly computes weight gradients even when the inside algorithm is pruned.

We have also proposed a declarative language, Dyna, that replaces Prolog's Horn clauses with "Horn equations" over terms with values. Dyna can express more than the semiring-weighted dynamic programs treated in this paper. Our ongoing work concerns the full Dyna language, program transformations, and feedback-directed optimization.

Finally, we evaluated our first implementation of a Dyna-to-C++ compiler (download and documentation at `http://dyna.org`). We hope it will facilitate EMNLP research, just as FS toolkits have done for the FS case. It produces code that is slower than hand-crafted code but acceptably fast for our NLP research, where it has been extremely helpful.

References

J. Binder, K. Murphy, and S. Russell. 1997. Space-efficient inference in dynamic probabilistic networks. In *Proc. of IJCAI*.

S. A. Caraballo and E. Charniak. 1998. New figures of merit for best-first probabilistic chart parsing. *CL*, 24(2):275–298.

E. Charniak, S. Goldwater, and M. Johnson. 1998. Edge-based best-first chart parsing. In *Proc. of COLING-ACL*.

M. J. Collins. 1999. *Head-Driven Statistical Models for Natural Language Parsing*. Ph.D. thesis, U. of Pennsylvania.

J. C. Culberson and J. Schaeffer. 1998. Pattern databases. *Computational Intelligence*.

A. Dempster, N. Laird, and D. Rubin. 1977. Maximum likelihood estimation from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society B*, 39:1–38.

J. Earley. 1970. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102.

J. Eisner and G. Satta. 1999. Efficient parsing for bilexical CFGs and head-automaton grammars. In *Proc. of ACL*.

J. Eisner and D. A. Smith. 2005a. Quasi-synchronous grammars: Alignment by soft projection of syntactic dependencies. Technical report, Johns Hopkins U.

J. Eisner and N. A. Smith. 2005b. Parsing with soft and hard constraints on dependency length. In *Proc. of IWPT*.

p.c.). Dyna is a lower-level language that itself knows nothing about the semantics of probability models, but whose inference rules could be used to implement any kind of message passing.

J. Eisner, E. Goldlust, and N. A. Smith. 2004. Dyna: A declarative language for implementing dynamic programs. In *Proc. of ACL* (companion vol.).

J. Eisner. 2001. *Smoothing a Probabilistic Lexicon via Syntactic Transformations*. Ph.D. thesis, U. of Pennsylvania.

J. Eisner. 2002. Parameter estimation for probabilistic FS transducers. In *Proc. of ACL*.

J. Goodman. 1999. Semiring parsing. *CL*, 25(4):573–605.

A. Griewank and G. Corliss, editors. 1991. *Automatic Differentiation of Algorithms*. SIAM.

M. Johnson. 2000. Inside-outside (computer program). `http://www.cog.brown.edu/~mj/Software.htm`.

L. Karttunen, J.-P. Chanod, G. Grefenstette, and A. Schiller. 1996. Regular expressions for language engineering. *JNLE*, 2(4):305–328.

M. Kifer and V. S. Subrahmanian. 1992. Theory of generalized annotated logic programming and its applications. *Journal of Logic Programming*, 12(4):335–368.

D. Klein and C. D. Manning. 2002. A generative constituent-context model for grammar induction. In *Proc. of ACL*.

D. Klein and C. D. Manning. 2003a. A* parsing: Fast exact Viterbi parse selection. In *Proc. of HLT-NAACL*.

D. Klein and C. D. Manning. 2003b. Accurate unlexicalized parsing. In *Proc. of ACL*.

J. Lafferty, A. McCallum, and F. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. of ICML*.

I. D. Melamed. 2003. Multitext grammars and synchronous parsers. In *Proc. HLT-NAACL*.

G. Minnen. 1996. Magic for filter optimization in dynamic bottom-up processing. In *Proc. of ACL*.

M. Mohri, F. Pereira, and M. Riley. 1998. A rational design for a weighted FST library. *LNCS*, 1436.

M.-J. Nederhof. 2003. Weighted deductive parsing and Knuth's algorithm. *CL*, 29(1):135–143.

I. Niemelä. 1998. Logic programs with stable model semantics as a constraint programming paradigm. In *Proc. Workshop on Computational Aspects of Nonmonotonic Reasoning*.

F. Pereira and Y. Schabes. 1992. Inside-outside reestimation from partially bracketed corpora. In *Proc. of ACL*.

F. Pereira and D. H. D. Warren. 1983. Parsing as deduction. In *Proc. of ACL*.

A. Pfeffer. 2001. IBAL: An integrated Bayesian agent language. In *Proc. of IJCAI*.

A. Prieditis. 1993. Machine discovery of effective admissible heuristics. *Machine Learning*, 12:117–41.

R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. 1994. The CORAL deductive system. *The VLDB Journal*, 3(2):161–210.

R. Ramakrishnan. 1991. Magic templates: a spellbinding approach to logic programs. *J. Log. Program.*, 11(3-4):189–216.

K. A. Ross and Y. Sagiv. 1992. Monotonic aggregation in deductive databases. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*.

S. M. Shieber, Y. Schabes, and F. Pereira. 1995. Principles and implementation of deductive parsing. *Journal of Logic Programming*, 24(1–2):3–36.

K. Sikkel. 1997. *Parsing Schemata: A Framework for Specification and Analysis of Parsing Algorithms*. Texts in Theoretical Computer Science. Springer.

N. A. Smith and J. Eisner. 2004. Annealing techniques for unsupervised statistical language learning. In *Proc. of ACL*.

N. A. Smith and J. Eisner. 2005. Contrastive estimation: Training log-linear models on unlabeled data. In *Proc. of ACL*.

D. A. Smith and N. A. Smith. 2004. Bilingual parsing with factored estimation: Using English to parse Korean. In *Proc. of EMNLP*.

N. A. Smith, D. A. Smith, and R. W. Tromble. 2005. Context-based morphological disambiguation with random fields. In *Proc. of HLT-EMNLP*.

G. Specht and B. Freitag. 1995. AMOS: A NL parser implemented as a deductive database in LOLA. In *Applications of Logic Databases*. Kluwer.

A. Stolcke. 1995. An efficient probabilistic CF parsing algorithm that computes prefix probabilities. *CL*, 21(2):165–201.

H. Tamaki and T. Sato. 1984. Unfold/fold transformation of logic programs. In S. Å. Tärnlund, editor, *Proceedings Second International Conference on Logic Programming*, pages 127–138, Uppsala University.

R. J. Williams and D. Zipser. 1989. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280.

D. Wu. 1997. Stochastic inversion transduction grammars and bilingual parsing of parallel corpora. *CL*, 23(3):377–404.

N.-F. Zhou and T. Sato. 2003. Toward a high-performance system for symbolic and statistical modeling. In *Proc. of Workshop on Learning Statistical Models from Relational Data*.

U. Zukowski and B. Freitag. 1997. The deductive database system *LOLA*. In *Logic Programming and Nonmonotonic Reasoning*, LNAI 1265. Springer.