Time-and-Space-Efficient Weighted Deduction

Jason Eisner
Department of Computer Science
Johns Hopkins University
jason@cs.jhu.edu

Abstract

Many NLP algorithms have been described in terms of deduction systems. Unweighted deduction allows a generic forward-chaining execution strategy. For weighted deduction, however, efficient execution should propagate the weight of each item only after it has converged. This means visiting the items in topologically sorted order (as in dynamic programming). Toposorting is fast on a materialized graph; unfortunately, materializing the graph would take extra space. Is there a generic weighted deduction strategy which, for every acyclic deduction system and every input, uses only a constant factor more time and space than generic unweighted deduction? After reviewing past strategies, we answer this question in the affirmative by combining ideas of Goodman (1999) and Kahn (1962). We also give an extension to cyclic deduction systems, based on Tarjan (1972).

1 Introduction

Many NLP algorithms have been described in terms of deduction systems, starting with the seminal paper "Parsing as Deduction" (Pereira and Warren, 1983). In general, deduction systems are an abstraction of dynamic computation graphs that can be used to describe the structure of many algorithms, including theorem provers, dynamic programming algorithms, and structured neural networks (Sikkel, 1993; Eisner and Filardo, 2011).

Unweighted deduction admits a generic execution strategy known as "forward chaining." Furthermore, the behavior of this strategy is well-understood. Its runtime and space can be easily bounded based on a simple inspection of the deduction system (McAllester, 2002). Static analysis can sometimes find tighter bounds (Vieira et al., 2022).

For *weighted* deduction, however, execution should wait to propagate a derived item's weight until that weight has converged. Ideally it would visit the items in topologically sorted order (i.e., dynamic programming), which is not required for the unweighted case. Fortunately, toposorting is fast. Thus, *is there a generic weighted deduction strategy* which, for every acyclic deduction system and every input, uses only a constant factor more time and space than generic unweighted deduction?

In this paper, we give this missing master strategy, which is surprisingly simple. It establishes a "don't worry, be happy" meta-theorem (McFerrin, 1988): Asymptotic analyses of unweighted acyclic deduction systems do transfer to the weighted case.

Past methods do not achieve this guarantee. For some deduction systems, static analysis may be able to identify a topological ordering. But even then, one is left with increased runtime, either from visiting all possible items (which fails to exploit the sparsity of the set of items actually derived from a particular input) or visiting only the items that have been derived (which exploits sparsity, but requires a priority queue that incurs logarithmic overhead).

The alternative is dynamic analysis: given the input, first identify all the items that can be derived, using unweighted deduction, and then toposort them in order to compute the weights. Goodman (1999) suggested a version of this approach that materializes the graph of dependencies among items, but acknowledged that this generally increases the asymptotic space requirements. In this paper, we show how to avoid the space increase. We give a practical two-pass weighted deduction algorithm, inspired by Kahn (1962), that uses parent counting to efficiently enumerate the derived items in topologically sorted order. It stores the counts of edges but no longer stores the edges themselves.

Also, for the case where the graph may have cycles, we give a practical three-pass algorithm. The first two passes efficiently enumerate the strongly connected components in topologically sorted order (Tarjan, 1972), with the same time and space guarantees as above. The third pass solves for the weights in each cyclic component, which may in-

crease the asymptotic runtime if cycles exist.

As an application of our two-pass acyclic algorithm, consider the extension of Earley parsing (Earley, 1970; Graham et al., 1980) to probabilistic (Stolcke, 1995) or semiring-weighted context-free grammars. Opedal et al. (2023) presents the method as an acyclic deduction system whose unweighted version is easily seen to execute in $\mathcal{O}(n^3|\mathcal{G}|)$ time and $\mathcal{O}(n^2|\mathcal{G}|)$ space on a sentence of length n and a grammar of size $|\mathcal{G}|$. These bounds are tight when every substring of the sentence can be analyzed, in its left context, as every sort of constituent or partial constituent. In practical settings, however, the parse chart is often much sparser than this worst case and execution is correspondingly faster. For instance, Earley (1970) describes "bounded-state grammars" where parsing is guaranteed to require only $\mathcal{O}(n|\mathcal{G}|)$ time and $\mathcal{O}(n|\mathcal{G}|)$ space. Opedal et al. (2023) offered all the same guarantees for weighted parsing, just by invoking the present paper ("don't worry"). In contrast, the various past methods would have been slower by a log factor, or consumed $\Theta(n^3|\mathcal{G}|)$ space for some grammars, or consumed $\Theta(n^3|\mathcal{G}|)$ runtime even for bounded-state grammars.

While the solution seems obvious in retrospect (at least once the problem is framed), it has somehow gone unnoticed in the literature. It has also evaded several hundred graduate and undergraduate students in the author's NLP class over two decades. Students each year are asked to design and implement a Viterbi Earley parser. As extra credit, they are asked if they can achieve $\mathcal{O}(n^3)$ runtime while maintaining correctness. None of them have ever spotted the Kahn-based solution, though some have found the other methods mentioned above.

2 Formal Framework

2.1 Deduction Systems

A **weighted deduction system** $\mathcal{P} = (\mathcal{V}, \mathbb{W}, \mathcal{E}, \oplus)$ consists of

- a possibly infinite set of **items**, $\mathcal{V}$, which may be regarded as representing propositions about the input to the system
- a set of **weights**, $\mathbb{W}$, where the weight associated with an item (if any) might be intended to characterize its truth value, probability, provenance, description, or other properties
- a possibly infinite set of **hyperedges**, $\mathcal{E}$, each of which is written in the form
$$v \overset{f}{\longleftarrow} u_1, \ldots, u_k$$
where $k \geq 1$ is the **in-degree** of the hyperedge,[1] $v, u_1, \ldots, u_k \in \mathcal{V}$ are items, and $f : \mathbb{W}^k \to \mathbb{W}$ is a **combination function**
- a function $\oplus$ that maps each item $v \in \mathcal{V}$ to an associative and commutative binary operator $\oplus_v : \mathbb{W} \times \mathbb{W} \to \mathbb{W}$, called the **aggregator** for item v

In short, $\mathcal{P}$ is an ordered B-hypergraph $(\mathcal{V}, \mathcal{E})$ where each hyperedge is labeled with a combination function and each vertex is labeled with an aggregator.[2] We say that $\mathcal{P}$ is an **acyclic** system if this hypergraph is acyclic. Note that our formalism is not limited to semiring-weighted systems.

2.2 Evaluation

We now explain how to regard $\mathcal{P}$ as an arithmetic circuit—a type of program that can be applied to inputs. An **input** to $\mathcal{P}$ is a pair (V, ω) where

- $V \subseteq \mathcal{V}$ is a set of **axioms**
- $\omega : V \to \mathbb{W}$ assigns a weight to each axiom

Evaluating $\mathcal{P}$ on this input returns an output pair $(\bar{V}, \bar{\omega})$ where

- the **derived items** $\bar{V} \subseteq \mathcal{V}$ are those that are reachable from the axioms V in the hypergraph $(\mathcal{V}, \mathcal{E})$; equivalently, $\bar{V}$ is the smallest set such that
  - $V \subseteq \bar{V}$
  - if $(v \overset{f}{\longleftarrow} u_1, \ldots, u_k) \in \mathcal{E}$ and $u_1, \ldots, u_k \in \bar{V}$, then $v \in \bar{V}$
- for each derived item $v \in \bar{V}$, let $\bar{E}_v \subseteq \mathcal{E}$ be the set of all hyperedges used to derive it:
$$\bar{E}_v = \{(v \overset{f}{\longleftarrow} u_1, \ldots, u_k) \in \mathcal{E} : u_1, \ldots, u_k \in \bar{V}\}$$

[1] The artificial requirement that $k \geq 1$ serves to simplify our pseudocode without loss of generality. A hyperedge $v \overset{f}{\longleftarrow}$ with $k = 0$ would state that v is an axiom of the deduction system, with weight $f()$. It can be replaced by a $k = 1$ hyperedge $v \overset{f'}{\longleftarrow} \texttt{start}$, where f' is the constant function returning $f()$, and `start` is a special item that is always included (with arbitrary weight) in the set of input axioms in §2.2 below.

[2] In a B-hypergraph (Gallo et al., 1993), each hyperedge pairs a set of input vertices with a single output vertex. In an *ordered* B-hypergraph, each hyperedge pairs a *sequence* of input vertices with a single output vertex: $v \overset{f}{\longleftarrow} u_1, u_2$ is not the same as $v \overset{f}{\longleftarrow} u_2, u_1$.

- $\bar{\omega} : \bar{V} \to \mathbb{W}$ satisfies the following constraints: for each $v \in \bar{V} \setminus V$,

$$\bar{\omega}(v) = \bigoplus_{v \xleftarrow{f} u_1, \ldots, u_k \in \bar{E}_v} f(\bar{\omega}(u_1), \ldots, \bar{\omega}(u_k)) \tag{1a}$$

and for each $v \in V$,

$$\bar{\omega}(v) = \omega(v) \oplus_v \bigoplus_{v \xleftarrow{f} u_1, \ldots, u_k \in \bar{E}_v} f(\bar{\omega}(u_1), \ldots, \bar{\omega}(u_k)) \tag{1b}$$

Note that $\bar{\omega}(v)$ always has at least one summand, since $\bar{E}_v = \emptyset$ is not possible in (1a).[3]

$\bar{V}$ is always uniquely determined, as is $\bar{E} \stackrel{\text{def}}{=} \bigcup_{v \in \bar{V}} \bar{E}_v$. How about $\bar{\omega}$? We say that $v \in \bar{V}$ **depends on** u if $\bar{E}_v$ contains a hyperedge of the form $v \xleftarrow{f} \ldots, u, \ldots$. If $(\bar{V}, \bar{E})$ is an acyclic hypergraph, meaning that no $v \in \bar{V}$ depends transitively on itself, then evaluation has a unique solution $(\bar{V}, \bar{\omega})$. This is guaranteed for acyclic $\mathcal{P}$. In general, however, there could be multiple functions $\bar{\omega}$ or no functions $\bar{\omega}$ that satisfy the system of constraints (1).

2.3 Special Cases

An **unweighted deduction system** is very similar to the above, but where f, $\oplus_v$, and ω are omitted. Thus, evaluation returns only $\bar{V}$ and not $\bar{\omega}$. Equivalently, an unweighted deduction system can be regarded as the special case where there is only a single weight: $\mathbb{W} = \{\top\}$. Then $f, \oplus_v, \omega$, and $\bar{\omega}$ are trivial constant functions that only return $\top$.

A **semiring-weighted deduction system** is the special case where $\oplus_v$ and f are fixed throughout $\mathcal{P}$ to be two specific operations $\oplus$ and $\otimes$, respectively, such that $(\mathbb{W}, \oplus, \otimes)$ forms a semiring. This case has proved very useful for NLP algorithms (Goodman, 1999; Eisner and Blatz, 2007; Vieira et al., 2021).

2.4 Example: Weighted CKY Parsing

We give an acyclic weighted deduction system $\mathcal{P} = (\mathcal{V}, \mathbb{W}, \mathcal{E}, \oplus)$ that corresponds to the inside algorithm for probabilistic context-free grammars in Chomsky normal form (Baker, 1979). We take the weights to be probabilities: $\mathbb{W} = [0, 1] \subseteq \mathbb{R}$. The item set $\mathcal{V}$ consists of all objects of the forms

`word(`x`,`i`,`k`)`	`rewrite(`a`,`x`)`
`phrase(`a`,`i`,`k`)`	`rewrite(`a`,`b`,`c`)`

and the hyperedges $\mathcal{E}$ are all objects of the forms

$$\texttt{phrase}(a,i,k) \xleftarrow{\times} \texttt{rewrite}(a,x), \texttt{word}(x,i,k)$$
$$\texttt{phrase}(a,i,k) \xleftarrow{\times} \texttt{rewrite}(a,b,c), \texttt{phrase}(b,i,j), \texttt{phrase}(c,j,k)$$

where x ranges over terminal symbols, a, b, c range over nonterminal symbols, i, j, k range over the natural numbers $\mathbb{N}$, and $\times$ is the ordinary multiplication function on probabilities. To complete the specification of $\mathcal{P}$, define $\oplus_v = +$ for all $v \in \mathcal{V}$.

To run the inside algorithm on a given sentence under a given grammar, one must provide the input (V, ω). Encode the sentence $x_1 x_2 \cdots x_n$ as the n axioms $\{\texttt{word}(x_i, i-1, i) \colon 1 \le i \le n\}$, each having weight 1. Encode the grammar with one `rewrite` axiom per production, whose weight is the probability of that production: for example, the production `S` $\to$ `NP VP` corresponds to the axiom `rewrite(S,NP,VP)`.

After evaluation, $\texttt{phrase}(a,i,k) \in \bar{V}$ iff $a \Rightarrow^* x_{i+1} \ldots x_k$, in which case $\bar{\omega}(\texttt{phrase}(a,i,k))$ gives the total probability of all derivations of that form—that is, the inside probability of nonterminal a over the input span $[i, k]$. We remark that furthermore, $\bar{E}$ is the traditional *packed parse forest*, with the hyperedges in $\bar{E}_v$ being the traditional *backpointers* from item $v \in \bar{V}$.

A variant of this system obtains the same results with greater hardware parallelism by taking the elements of $\mathbb{W}$ to be tensors. For each (i, k) pair, all items $\texttt{phrase}(a,i,k)$ are collapsed into a single item $\texttt{phrase}(i,k)$ whose weight is a 1-dimensional tensor (vector) indexed by nonterminals a. Similarly, all items $\texttt{rewrite}(a,b,c)$ are collapsed into a single item `rewrite_binary` whose weight is a 3-dimensional tensor, and all items $\texttt{rewrite}(a,x)$ into `rewrite_unary` whose weight is a 2-dimensional tensor. The combination functions and aggregators are modified to operate over tensors. It is also possible to make them nonlinear to get a neural parser (Drozdov et al., 2019).

Another variant lets the grammar contain unary nonterminal productions, represented by additional axioms of the form $\texttt{rewrite}(a,x)$, by adding all hyperedges of the form $\texttt{phrase}(a,i,k) \xleftarrow{\times} \texttt{rewrite}(a,b), \texttt{phrase}(b,i,k)$. The resulting $\mathcal{P}$ is cyclic iff the grammar has unary production cycles. Even then, for a probabilistic grammar, it turns out that evaluation always gives a unique $\bar{\omega}$.

[3] When $v \in V$, typically $\omega(v)$ is the *only* summand. However, we have no reason to require that: for generality, our formalism and algorithms do allow an axiom to be derived again from other items, which then contribute to its weight (1b).

2.5 Computational Framework

The definitions above were mathematical. To implement evaluation on a computer, we assume that the set of input axioms V is finite.[4] We also hope that for a given input, the set of derived items $\bar{V}$ will also be finite (even if $\mathcal{V}$ is infinite), since otherwise the algorithms in this paper will not terminate.[5] Since $\mathcal{P}$ itself is typically infinite in order to handle arbitrary inputs, we assume that its components have been specified using code. In particular, $\mathcal{V}$ and $\mathbb{W}$ are datatypes, $\oplus$ is a function, and $\mathcal{E}$ is accessed through iterators described below.

In practice, a deduction system $\mathcal{P}$ is usually specified using finitely many patterns that contain variables, as was done in §2.4. The patterns that specify $\mathcal{V}$ are called **types**, and the patterns that specify $\mathcal{E}$ are called **deductive rules** or **sequents**. Once these patterns are written down, the necessary code can be generated automatically. The most popular formal notation for the unweighted case is the deductive rules of sequent calculus (Sikkel, 1993; Shieber et al., 1995). That formalism was generalized by Goodman (1999) to the semiring-weighted case. Sato (1995) and Eisner et al. (2005) developed notations based on logic programming languages for those special cases, and Eisner and Filardo (2011) then generalized those notations.

Each algorithm in this paper takes $(\mathcal{P}, V, \omega)$ as its input. Each algorithm makes use of a **chart** C, a data structure that maps keys in $\mathcal{V}$ (namely derived items) to values in $\mathbb{W}$. If the algorithm terminates, C then holds the result of evaluating $\mathcal{P}$ on (V, ω): $\bar{V}$ is finite and is given by the keys of C, while $\bar{\omega} : \bar{V} \to \mathbb{W}$ is given by the mapping $v \mapsto C[v]$.

Each algorithm also uses an **agenda** A, which is a set of items.[6] Each derived item is added to A ("**pushed**") when it is added as a key to C for the first time (or in the case of Algorithm 5, for the *last* time). Later it is removed again ("**popped**") and new items are derived from it. The chart is complete once the agenda is empty. Algorithm 2 allows a popped item to be pushed again later because its weight has changed; our other algorithms do not.

In our pseudocode, C and A are global but all other variables are local (important for recursion).

The chart class provides iterators specific to $\mathcal{P}$:

- $C.\texttt{in}(v)$ yields all hyperedges in $\mathcal{E}$ of the form $v \overset{f}{\longleftarrow} u_1, \ldots, u_k$ such that $u_1, \ldots, u_k$ have already popped from A.
- $C.\texttt{out}(u)$ yields all hyperedges $v \overset{f}{\longleftarrow} u_1, \ldots, u_k$ in $\mathcal{E}$ such that $u_1, \ldots, u_k$ have already popped from A and $u \in \{u_1, \ldots, u_k\}$. Note that if u appears more than once among $u_1, \ldots, u_k$, then the iterator should still only yield the hyperedge once. This is necessary to ensure that with the hyperedge $v \overset{f}{\longleftarrow} u, u$, Algorithm 2 below will increment $\bar{\omega}(v)$ by $f(\bar{\omega}(u), \bar{\omega}(u))$ only once and not twice.

In practice, these iterators are made efficient via index data structures, as discussed below. Notice that $A.\texttt{pop}(u)$ must notify C to add u to these indexes.

The code that is automatically generated for $\mathcal{P}$ implements the item datatype, the chart class including iterators, and the agenda class.

Some of the algorithms in this paper require only `out()` and not `in()`. We prefer to avoid `in()` for reasons discussed in §5.3 below; see also §8.

In our efficiency analyses, we will sometimes make the following **standard assumptions** (but they are *not* needed for the meta-theorem of §1):

1. There exists a constant K such that all hyperedges in $\mathcal{E}$ have in-degree $k \leq K$.
2. f and $\oplus_v$ can be computed in $\mathcal{O}(1)$ time.
3. Storing an item as a chart key, testing whether it is one, or looking up or modifying its weight in the chart, can be achieved in $\mathcal{O}(1)$ time.
4. The $C.\texttt{out}(v)$ method and (when used) the $C.\texttt{in}(u)$ method create iterators in $\mathcal{O}(1)$ time. Furthermore, calling such an iterator to get the next hyperedge (or learn that there is no next hyperedge) takes $\mathcal{O}(1)$ time. Hence iterating over an item's $m \geq 0$ in- or out-edges can be done in time proportional to $m + 1$.
5. A chart mapping N items to their weights can be stored in $\mathcal{O}(N)$ space, including any private indexes needed to support the iterators.

[4] To keep V finite, arithmetic facts such as `plus(4,1,5)` are generally not included in V (or $\mathcal{V}$). Deductive rules (§2.5) may consult these built-in facts, but they are not treated as items in our framework, only as conditions on hyperedges.

[5] The infinite $\bar{V}$ case does arise when the deduction system $\mathcal{P}$ is designed to derive provable mathematical theorems, valid real-world inferences, reachable configurations of a Turing machine, etc. Computational techniques in this case include timeout, pruning, query-driven evaluation (§8), and lifted inference. Timeout and pruning involve small modifications to the algorithms in this paper. The other strategies can be implemented by automatically transforming $\mathcal{P}$ to derive fewer items (e.g. Beeri and Ramakrishnan, 1991) and leaving our algorithms unchanged.

[6] The agenda A and chart C respectively correspond to the *open set* and *closed set* of search algorithms like A*.

In the author's experience, these assumptions can be made to hold for most deduction systems used in NLP, at least given the Uniform Hashing Assumption[7] and using the Word RAM model of computation. However, a few tricks may be needed:

- To ensure that each axiom can be stored in $\mathcal{O}(1)$ space, it may be necessary to transform the deduction system. E.g., rather than specifying each context-free grammar rule—of any length—by a single axiom, a long rule can be *binarized* so that it is specified by multiple axioms (thus increasing N), each of which requires only $\mathcal{O}(1)$ space to store.
- Even when individual axioms and their weights are bounded in size, the derived items or their weights may be unbounded in size. However, often the space and time requirements above can still be achieved through *structure sharing* and *hash consing*.[8]
- To ensure the given iterator performance, it may be necessary to transform the deduction system, again increasing N. In particular, McAllester (2002) applies a transformation to unweighted deduction systems that are written in his notation; Eisner et al. (2005) apply a very similar approach to weighted systems. In the transformed systems,[9] hyperedges have in-degree ≤ 2. Hence $C.\texttt{out}(u)$ must find all previously popped items u' such that $\mathcal{E}$ contains hyperedges with inputs u, u' or u', u. Moreover, those items can be found with a constant number of hash lookups, by having C maintain a constant number of hash tables (indexes), each of which maps from a possible property of u' to a list of all previously popped items u' with that property. For instance, for the weighted CKY system in §2.4, one such hash table would map each integer j to all items of the form $\texttt{phrase}(b, i, j)$, if any.
- To ensure that `out` does not double-count hyperedges, it may be convenient to transform the deduction system, for example replacing $v \xleftarrow{f} u, u$ with $v \xleftarrow{f} u, u'$ and $u' \xleftarrow{\text{id}} u$.

Further implementation details are given in the papers that were cited throughout the present section. We do not dwell on them as they are orthogonal to the concerns of the present paper.

3 Unweighted Forward Chaining

A basic strategy for unweighted deduction is forward chaining (Algorithm 1), well-known in the Datalog community (Ceri et al., 1990). This is simply a reachability algorithm: it finds all of the items that can be reached from the axioms V in the B-hypergraph $(\mathcal{V}, \mathcal{E})$. If the standard assumptions of §2.5 hold, it runs in time $\mathcal{O}(|\bar{V}|+|\bar{E}|)$ and space $\mathcal{O}(|\bar{V}|)$.

The agenda A is a set that supports $\mathcal{O}(1)$-time `push()` and `pop()` methods. `push()` adds an element to the set (our code is careful to never push duplicates). `pop()` removes and returns some arbitrary element of the set. If `pop()` is implemented to use a LIFO or FIFO policy, then the reachability algorithm will be depth-first or breadth-first, respectively, but any policy ("queue discipline") will do provided that $\bar{V}$ is finite.

Algorithm 1 Unweighted forward chaining

```
C ← ∅; A ← ∅                                  ▷ here C ⊆ V is just a set
for v ∈ V :                                    ▷ axioms
    C.add(v); A.push(v)
while A ≠ ∅ :
    u ← A.pop()                                ▷ remove some element
    for (v ←f u_1, ..., u_k) ∈ C.out(u) :
        if v ∉ C :                             ▷ also implies v ∉ A
            C.add(v); A.push(v)
```

Since this is the unweighted case, the chart C does not need to store weights. It is simply the set of items derived so far, approaching the desired $\bar{V}$.[10] In later algorithms, however, C will be a map from those items to the weights that they have accumulated so far, approaching the desired $\bar{\omega}$.

[7] Alternatively, we can use universal hashing. Then the runtime bounds are bounds on the *expected* runtime, where the expectation is over the choice of hash function.

[8] E.g., for weighted deduction systems written in Dyna (Eisner et al., 2005), items may be unbounded lists or other deeply nested Prolog-style terms. Yet structure sharing allows `out()` or `in()` to build a complex item in $\mathcal{O}(1)$ time, and hash consing finds a pointer in $\mathcal{O}(1)$ time to the copy of that item (if any) that is stored as a chart key alongside the item's weight.

[9] The transformation introduces intermediate items for partially matched rules ("prefix firings"). These new items are analogous to "saved states" in the RETE forward-chaining algorithm for OPS-5 (Forgy, 1979; Perlin, 1990).

[10] To guarantee that C actually *converges to* a possibly infinite $\bar{V}$—in the sense that every item in $\bar{V}$ is eventually added to C—one must use an agenda policy that avoids starvation. For example, if the policy periodically selects the least recently pushed item, then every pushed item is eventually popped.

Notice that since items are seen by the iterators only after they have popped, the hyperedge $v \xleftarrow{f} u_1, \ldots, u_k$ is considered only once—when the last of its input items (not necessarily u_k) pops. This "considered only once" property makes Algorithm 1 more efficient, but is not required for correctness. For the weighted algorithm below, however, it is required to prevent double-counting of hyperedges.

4 Weighted Forward Chaining

Algorithm 2 is a first attempt to generalize to the weighted case, where C becomes a map.

The pseudocode uses the following conventions: If v is not a key of C, then $C[v]$ returns a special value $\bot \notin \mathbb{W}$. We define $\bot \oplus_v \omega = \omega$ for all ω. Changing $C[v]$ from $\bot$ to a value in $\mathbb{W}$ adds v as a new key to the chart C.

Algorithm 2 Weighted forward chaining

```
1:  C ← ∅                                   ▷ map with keys in 𝒱, values in 𝕎
2:  A ← ∅
3:  for v ∈ V :                             ▷ axioms
4:      C[v] ⊕_v= ω(v)
5:      A.push(v)
6:  while A ≠ ∅ :
7:      u ← A.pop()                         ▷ remove some element
8:      for (v ←f u_1, ..., u_k) ∈ C.out(u) :
9:          C[v] ⊕_v= f(C[u_1], ..., C[u_k])
10:         if C[v] changed :
11:             if v has already popped from A : error
12:             if v ∉ A : A.push(v)
```

If $C[v]$ has changed at line 10, we must have it on the agenda to ensure that its new value is propagated forward through the hypergraph. The error at line 11 occurs when we would propagate *both* old and new values, which in general leads to incorrect results. The *major challenge of this paper* is to efficiently avoid this error condition.

In an acyclic system, the "obvious" way to avoid the error condition is to pop items from the agenda in topologically sorted (toposorted) order—do not pop v until every item that v depends on has already popped. If that can be arranged, then v will only pop after its value has converged; $C[v]$ will never change again, so v will never be pushed again.

We remark that toposorting is not the only option. Line 11 can simply be omitted in the special case where $\mathbb{W} = \mathbb{R}$, $\oplus_v = \min$ for all $v \in \mathcal{V}$, and all functions f are monotonic on all arguments. Then $C[v]$ can only decrease at line 9, and propagating its new (smaller) value will in effect replace the old value.[11] Separately, there exists a slightly different family of algorithms which, each time u pops from the agenda, propagates the *difference* between $C[u]$ and the old value of $C[u]$ from the last time it popped. Such "difference propagation algorithms" exist for semiring-weighted deduction systems (Eisner et al., 2005, Fig. 3) and also exist for weighted deduction systems in which each operation $\oplus_v$ has a corresponding operation $\ominus_v$ that can be used to compute differences of values.

The approaches in the previous paragraph are correct in the special cases where they apply. They even work for cyclic deduction systems (provided that they converge). However, they are an inefficient choice for the acyclic case—the same derived item in $\bar{V}$ could pop more than once (in fact far more), which we refer to as **repropagation**,[12] but in the toposorted solution, it will pop exactly once. We therefore focus on toposorting approaches.

5 Previous Toposorted Strategies

Assume that $\mathcal{P}$ is acyclic and that unweighted forward chaining terminates. We discuss previously proposed strategies that visit the finitely many derived items $\bar{V}$ in a toposorted order, and explain why these strategies do not satisfy our goals.

In addition to methods based on Algorithms 1 and 2, we consider algorithms that **relax** the items in $\bar{V}$ in a topologically sorted order. $C.\texttt{relax}(v)$ recomputes $C[v]$ based on the current state of the chart C as well as the input (V, ω):

```
1:  C[v] ← if v ∈ V then ω(v) else ⊥
2:  for (v ←f u_1, ..., u_k) ∈ C.in(v) :
3:      C[v] ⊕_v= f(C[u_1], ..., C[u_k])
```

[11] This special case also applies when $\mathbb{W}$ is a lattice, $\oplus_v$ is the meet operation, and f preserves the lattice's partial order.

[12] Although for the special case with $\oplus_v = \min$ that was just mentioned at the start of the previous paragraph, there is a well-known way to avoid repropagation even in the cyclic case, inspired by Dijkstra (1959)'s shortest-path algorithm: prioritize the agenda so that v with minimum $C[v]$ pops first (Knuth, 1977; Nederhof, 2003). This suffices provided that each f is **superior**, meaning that f is not only monotonic (i.e., monotonically non-decreasing in all arguments) but also its output is $\geq$ all of its inputs. Under the standard assumptions of §2.5, a runtime of $O(|\bar{E}| + |\bar{V}| \log |\bar{V}|)$ can be achieved by implementing the agenda as a Fibonacci heap (Fredman and Tarjan, 1987). Note that the prioritization methods in §5.1 below are quite different: they prioritize based only on v, not $C[v]$, so they do not depend on any properties of $\mathbb{W}$, $\oplus_v$, or f, and do not require the `decrease_key` method.

If we relax every $v \in \bar{V}$ once, *after* relaxing all of the items that it depends on, then C will hold the correct result of evaluation from (1). Note that it is harmless to relax $v \notin \bar{V}$: this will simply leave $C[v] = \bot$, so it will not be added to the chart.

5.1 Prioritization

One approach to evaluation is to run Algorithm 2 where the agenda A is a priority queue that pops the derived items in a *known toposorted order*. This requires specifying a **priority function** π such that if v depends on u, then u pops first because $\pi(u) \prec \pi(v)$, where $\prec$ is a total ordering of the priorities returned by π. π could depend on the input V. It must be manually designed for a given $\mathcal{P}$ (or perhaps in some cases, it could be automatically constructed via some sort of static analysis of $\mathcal{P}$). We will assume in our discussion that π and $\prec$ are constant-time functions.

Now the agenda push and pop methods are simply the `insert` and `delete_min` operations of a priority queue. Each takes $\mathcal{O}(\log |\bar{V}|)$ time when the priority queue is implemented as a binary heap with at most $|\bar{V}|$ entries.[13] Unfortunately this approach adds $\mathcal{O}(|\bar{V}| \log |\bar{V}|)$ overall to our runtime bound, which can make it asymptotically slower than unweighted forward chaining (§3).

If the priorities are integers in the range $[0, N)$ for some $N > 0$, then this extra runtime can be reduced to $\mathcal{O}(N)$ using a bucket priority queue (Dial, 1969). A **bucket priority queue** maintains an array that maps each $p \in [0, N)$ to a **bucket** (set) that contains all agenda items with priority p. The bucket queue also maintains p_{last}, the priority of the most recently popped item (initially 0). $A.\mathsf{push}(v)$ simply adds v to bucket $\pi(v)$. $A.\mathsf{pop}()$ increments p_{last} until it reaches the first non-empty bucket, then removes any item from that bucket. This works because our usage of the priority queue is **monotone** (Thorup, 2000)—that is, the minimum priority never decreases, so p_{last} never needs to decrease. Overall, Algorithm 2 will scan forward once through all the empty and non-empty buckets, taking $\mathcal{O}(N)$ time. When there is a risk of large N, we observe that an alternative is to store only the non-empty buckets on the integer priority queue of Thorup (2000), so the next non-empty bucket can be found in $\mathcal{O}(\log \log B)$ time where B is the current number of non-empty buckets (assuming the Word RAM model of computation and assuming that priorities fit in a single word). The monotone property ensures that each bucket is added and emptied only once, so the extra runtime is now $\mathcal{O}(N' \log \log N')$ where N' bounds the number of distinct priority levels (buckets) ever used, e.g., $N' = \min(|\bar{V}|, N)$. This is an asymptotic improvement if $N' \log \log N' = o(N)$.

An example permitting small N is the deduction system for CKY parsing (§2.4). Take $N = n$, where n is the length of the input sentence, and define $\pi(\texttt{phrase(}a\texttt{,}i\texttt{,}k\texttt{)}) = k - i - 1 \in [0, N)$. Many items will have the same integer priority.

Consider also the extension that allows acyclic unary productions (§2.4). Here we can identify each nonterminal a with an integer in $[0, m)$ such that $b < a$ if $\texttt{rewrite(}a\texttt{,}b\texttt{)} \in V$. Take $N = nm$, $\pi(\texttt{phrase(}a\texttt{,}i\texttt{,}k\texttt{)}) = (k-i-1)\cdot m + a \in [0, N)$.

While prioritization is often useful both theoretically and practically, it does not solve our main problem. First, it assumes we know a priority function for $\mathcal{P}$. Second, the runtime is not guaranteed to be proportional to that of Algorithm 1, even with integer priorities. In the last example above, N may not be $\mathcal{O}(|\bar{E}| + |\bar{V}|)$: consider a pathological class of inputs where the grammar has very long unary production chains (so m is large) that are not actually reached from the input sentences. The alternative $N' \log \log N'$ may also fail to be $\mathcal{O}(|\bar{E}| + |\bar{V}|)$, e.g., on a class of inputs where $|\bar{E}| = \mathcal{O}(|\bar{V}|)$.

5.2 Dynamic Programming Tabulation

Again by analyzing $\mathcal{P}$ manually (or potentially automatically), it may be possible to devise a method which, given the axioms V, iterates in topologically sorted order over some finite "worst-case" set $X \subseteq \mathcal{V}$ that is guaranteed to be a superset of $\bar{V}$. By relaxing all $v \in X$ in this order, we get the correct result.

This is called **tabulation**—systematically filling in a table (namely the chart) that records $\bar{\omega}(v) \in \mathbb{W} \cup \{\bot\}$ for all $v \in X$. An example is CKY parsing (Kasami, 1965; Baker, 1979). Define $\mathcal{V}$ as in §2.4. Given V that specifies a sentence of length n and a context-free grammar in Chomsky Normal Form, let $X \stackrel{\text{def}}{=} V \cup \{\texttt{phrase(}x\texttt{,}i\texttt{,}k\texttt{)}\colon\ 0 \le i < k \le n\}$. The CKY algorithm relaxes all items in X in increasing order of their widths $k - i$, where `rewrite` items are said to have width 0.

This strategy does away with the $\mathsf{out}()$ iterator.

[13] In contrast to shortest-path problems (footnote 12), there is no asymptotic advantage here to using a more complicated Fibonacci heap, since the `decrease_key` operation is not needed: the priority (i.e., key) of an item u is fixed in advance at $\pi(u)$ and hence never changes after it is pushed.

The trouble is that it calls $C.\texttt{in}(v)$ on all items $v \in X$—and X may be much larger than $\bar{V}$. That ruins our goal of making weighted evaluation on every input as fast as unweighted evaluation (up to a constant factor). For example, in CKY parsing, $|X|$ is $\Theta(n^2)$, yet parsing a particular sentence with a particular grammar may lead to a very sparse $\bar{V}$. Algorithm 1 will be fast on this input whereas CKY remains slow ($\Omega(n^2)$). For a more precisely analyzed example, consider the deduction system corresponding to Earley's algorithm, as already mentioned in §1. A bounded-state grammar is guaranteed to give a sparse $\bar{V}$ on every input sentence. Goodman (1999, §5) points out that on such a grammar, Algorithm 1 runs in $\mathcal{O}(n)$ time, but again, tabulation takes $\Omega(n^2)$ time since $|X| = \Theta(n^2)$.[14]

5.3 The Two-Pass Trick

To avoid this problem, Goodman (1999, §5) proposes to (1) compute $\bar{V}$ using unweighted forward chaining (Algorithm 1), and then (2) relax just the items in $\bar{V}$, in effect choosing $X = \bar{V}$. The question now—in our acyclic case—is how pass (2) can visit the items in $\bar{V}$ in a topologically sorted order. Using a priority queue to do so would have the same overhead as using a priority queue with the simpler one-pass method (§5.1).

Instead, we can use **backward chaining** during the second pass (Algorithm 3). This is a simple recursive algorithm that essentially interleaves relaxation with depth-first search. It indeed relaxes the vertices in a toposorted order. The call stack acts like a LIFO agenda (made explicit in Algorithm 7).

Algorithm 3 Weight computation by backward chaining

```
▷ Algorithm 1 has already been run to compute V̄
C ← ∅                                   ▷ now C is a map with keys in V̄
for v ∈ V̄ : COMPUTE(v)                  ▷ V̄ is the old set C
procedure COMPUTE(v)
    if C[v] = ⊥ :                       ▷ first visit
        ▷ this iterator requires u_1, ..., u_k to have
          been popped from Algorithm 1's agenda
        for (v ←f u_1, ..., u_k) ∈ C.in(v) :
            for i ← 1 to k : COMPUTE(u_i)
        C.relax(v)
        assert C[v] ≠ ⊥                 ▷ because v ∈ V̄
```

Note that $C.\texttt{in}(v)$ is only called on $v \in \bar{V}$, and it only iterates over the hyperedges in $\bar{E}_v$, which are supported by items in $\bar{V}$. This is why a first pass to discover $\bar{V}$ is helpful. By contrast, a pure one-pass backward chaining method would have to identify a finite X and iterate over all of $\mathcal{E}$'s hyperedges to items $v \in X$, which would visit many items and hyperedges that are not in $\bar{V}$ and $\bar{E}$.

At first glance, Algorithm 3 appears to be exactly what we need: under our standard assumptions (§2.5), it runs in time $\mathcal{O}(|\bar{V}| + |\bar{E}|)$ and space $\mathcal{O}(|\bar{V}|)$, just like Algorithm 1. The question is whether the standard assumptions still hold. Algorithm 3 requires not only $C.\texttt{out}(u)$ on the forward pass (like Algorithm 1) but also $C.\texttt{in}(v)$ on the backward pass. Supporting the second type of iterator will generally require maintaining additional indexes in the chart C. For some deduction systems, that is only a mild annoyance and a constant-factor overhead. Unfortunately, there do exist pathological cases where the second iterator is asymptotically more expensive. This implies that Algorithm 3 cannot serve as a generic master algorithm to establish our meta-theorem (§1).

For such a case, consider a deduction system with hyperedges of the form $\texttt{t}(p \cdot q) \xleftarrow{f} \texttt{r}(p), \texttt{s}(q)$ where p and q are large primes and $p \cdot q$ is their product $\pmod m$.[15] Suppose there are n r items and n s items. On the forward pass, this requires all n^2 combinations. On the backward pass, since factoring $p \cdot q$ is computationally hard, the $\texttt{out}()$ iterator cannot easily work backwards from a given t item—the best strategy is probably to iterate over *all* (r,s) pairs that have popped from the agenda to find out which ones yield the given t item. Since this would be done for *each* t item, the overall runtime of the backward pass could be as bad as $\Theta(mn^2)$, even though there are only n^2 edges and the forward pass runs in time $\mathcal{O}(n^2)$.

What Goodman (1999, §5) actually proposed was to materialize the hypergraph of derived items during forward chaining, allowing the use of classical toposorting algorithms, which run on materialized graphs. In particular, if each derived item v stores its incoming hyperedges $\bar{E}_v$, then it is easy to make $C.\texttt{in}(v)$ satisfy the standard assumptions—it simply iterates over the stored list. Then Algorithm 3 achieves the desired runtime. Unfortunately, as Goodman acknowledged, materialization

[14] In principle, this might be fixed by making X a tighter upper bound on $\bar{V}$. Given V, evaluation would have to analyze the grammar and construct a special smaller X for the sentence with $|X| = \mathcal{O}(n)$, then use that X. However, §5.3 is easier.

[15] Or more dramatically, p and q are strings and $p \cdot q$ denotes a cryptographic hash of their concatenation into range $[0, m)$.

Algorithm 4 Unweighted forward chaining with parent counting (compare Algorithm 1)

$C \leftarrow \emptyset$; $A \leftarrow \emptyset$ ▷ *here $C \subseteq \mathcal{V}$ is just a set*
for $v \in V$: ▷ *axioms*
  $C.\mathsf{add}(v)$; $A.\mathsf{push}(v)$
  $waiting\_edges[v]$ += 1 ▷ *# summands needed*
while $A \neq \emptyset$:
  $u \leftarrow A.\mathsf{pop}()$ ▷ *remove some element*
  $waiting\_items$ += 1 ▷ *# items popped*
  for $(v \xleftarrow{f} u_1, \ldots, u_k) \in C.\mathsf{out}(u)$:
    if $waiting\_edges[v] = 0$: ▷ *$v \notin C$*
      $C.\mathsf{add}(v)$; $A.\mathsf{push}(v)$
    $waiting\_edges[v]$ += 1

Algorithm 5 Weighted forward chaining with parent counting (compare Algorithm 2)

▷ *Algorithm 4 has already computed $waiting\_edges[v]$ and $waiting\_items$.*
$C \leftarrow \emptyset$; $A \leftarrow \emptyset$ ▷ *now C is a map with keys in $\bar{V}$*
for $v \in V$: ▷ *axioms*
  CONTRIBUTE($v, \omega(v)$)
while $A \neq \emptyset$:
  $u \leftarrow A.\mathsf{pop}()$ ▷ *remove some element*
  $waiting\_items$ −= 1 ▷ *# items unpopped*
  for $(v \xleftarrow{f} u_1, \ldots, u_k) \in C.\mathsf{out}(u)$:
    CONTRIBUTE(v, $f(C[u_1], \ldots, C[u_k])$)
if $waiting\_items \neq 0$: **error** ▷ *cycle detected*
procedure CONTRIBUTE(v,w)
  $C[v] \oplus_v = w$
  $waiting\_edges[v]$ −= 1 ▷ *# summands needed*
  if $waiting\_edges[v] = 0$:
    $A.\mathsf{push}(v)$ ▷ *delayed push: item is ready*

drives the *space* requirement up from $\Theta(|\bar{V}|)$ to $\Theta(|\bar{V}| + |\bar{E}|)$. In the example of the previous paragraph, the space goes from $\Theta(m+n)$ to $\Theta(m+n^2)$.

5.4 Discussion

In summary, none of the previous strategies for acyclic weighted deduction are guaranteed to achieve the same time and space performance as unweighted deduction (Algorithm 1). In addition, some of them require constructing priority functions or efficient `in()` iterators. While they are quite practical for some systems, it has not previously been possible for a paper that presents an unweighted deduction system to simply assert that its asymptotic analysis—for example, via McAllester (2002)—will carry through to weighted deduction.

6 Two-Pass Weighted Deduction via Parent Counting

We now present our simple solution to the problem, in the acyclic case. As already mentioned, it allows acyclic weighted deduction systems such as the inside algorithm (§2.4) and weighted Earley's algorithm (§1) to benefit from sparsity in the chart just as much as their unweighted versions do, without any asymptotic time or space overhead.

The idea is to use a version of topological sorting (Kahn, 1962) that only requires following out-hyperedges, as in forward chaining. We first modify our first pass (Algorithm 4). An item $v \in C$ does not store its incoming hyperedges $\bar{E}_v$, but just stores the *count* of those incoming hyperedges in $waiting\_edges[v]$ (initially 0).

Now, the second pass (Algorithm 5) computes the weights, waiting to push an item v onto the agenda until all contributions to its total weight have been aggregated into $C[v]$. $waiting\_edges[v]$ holds the count of contributions that have *not yet* arrived; when this is reduced to 0, the item can finally be pushed.[16] The agenda in the second pass plays the role of the "ready list" of Kahn (1962).

All of the same work of deriving items will be done again on the second pass, with no memory of the first pass other than the counts. Both passes use forward chaining and derive the same items—but the second pass may derive them in a different order, as it deliberately imposes ordering constraints.

Our pseudocode also maintains an extra counter, $waiting\_items$, used only to check the requirement that the input graph must be acyclic.

For any input, both passes consume only as much runtime and space as Algorithm 1, up to a constant factor.[17] This constant-factor result does not require all of the standard assumptions of §2.5. It needs only the assumptions that f and $\oplus_v$ can be computed in $\mathcal{O}(1)$ time, and that an item's weight can be stored in space proportional to the space required by the item itself. If even these assumptions fail, we can still say that the additional runtime of the weighted algorithm is only the total cost of running f and $\oplus_v$ for every hyperedge $v \xleftarrow{f} \cdots$ in

[16] Just as a garbage collector can destroy an object when its reference count is reduced to 0.

[17] More precisely, only as much as the *maximum* runtime and space of Algorithm 1 on that input, where the max is over possible pop orders. The pop order is left unspecified in Algorithm 1 but might affect the actual runtime of the iterators.

$|\bar{E}|$, and the additional space is only what is needed to store the weights of the $|\bar{V}|$ items. When all the standard assumptions apply, the method achieves $\mathcal{O}(|\bar{V}| + |\bar{E}|)$ time and $\mathcal{O}(|\bar{V}|)$ space, just like §3.

7 Weighted Deduction With Cycles

Goodman (1999) also considers the case where the derived hypergraph $(\bar{V}, \bar{E})$ may be cyclic (while remaining finite). There, we should partition it into strongly connected components (SCCs), which we should visit and solve in toposorted order.

7.1 Solving the SCCs

Let $S \subseteq \bar{V}$ be an SCC. We assume a procedure SOLVESCC(S) that sets $C[v] = \bar{\omega}(v)$ for all $v \in S$, where the $\bar{\omega}(v)$ values jointly satisfy the $|S|$ equations (1). The procedure will be called only after previous SCCs have been solved. Thus, $C[u] = \bar{\omega}(u)$ already whenever $v \in S$ depends on $u \notin S$.

Solution is commonly attempted by relaxing the items $v \in S$ repeatedly until there is no further change.[18] Since $C.\texttt{relax}(v)$ (from §5) requires the $\texttt{in}()$ iterator, we can optionally reorganize the computation to use only $\texttt{out}()$: see Algorithm 6.[19]

Repeated relaxation is not guaranteed to terminate, nor even converge in the limit to a solution $\bar{\omega}$, even when one exists. However, under some deduction systems, S may have special structure that enables a finite-time solution algorithm. For example, footnote 12 noted that Knuth (1977) solves certain weighted deduction systems in which $\oplus_v = \min$. As another example, the equations (1) may become linear once the weights $\bar{\omega}(u)$ from previous SCCs have been replaced with constants. If the $\oplus$ and $\otimes$ operations of this linear system of equations form a semiring that also has a **closure operator** for summing a geometric series in finite time, then the Kleene-Floyd-Warshall algorithm (Lehmann, 1977) solves the SCC S in $\mathcal{O}(|S|^3)$ operations. This may be improved to sub-cubic (e.g., Strassen, 1969) if an $\ominus$ operator is also available.

[18] Much like the Bellman-Ford shortest-paths algorithm (Bellman, 1958), although that algorithm also includes a test that halts with an error if convergence is impossible.

[19] Algorithm 6 relaxes all $v \in S$ in parallel, which unfortunately can cause oscillation in some cases where serial relaxation would converge. A serial variant, which requires more space, would temporarily materialize the out-edges from all $u \in S$: the out-edges to $v \in S$ are stored at v to support serial relaxation within S, while the out-edges to $v \notin S$ may be saved in a separate list and used at the end to propagate to later SCCs. Alternatively, when a difference propagation algorithm exists (§4), one could run it on S to convergence.

Algorithm 6 Solving an SCC using only $\texttt{out}()$

procedure SOLVESCC(S)
  ▷ *This procedure assumes that $C[v]$ already aggregates the* non-cyclic *contributions to $\bar{\omega}(v)$, from axioms and earlier SCCs.*
  ▷ *$C_{\text{prev}}, C_{\text{new}}$ are local maps with keys in S.*
  for $v \in S$:
    if $v \in V : C[v] \oplus_v= \omega(v)$ ▷ *axiom?*
    $C_{\text{prev}}[v] \leftarrow C[v]$ ▷ *all acylic contributions*
  while true : ▷ *update until convergence*
    $C_{\text{new}} \leftarrow C_{\text{prev}}$ ▷ *deep copy*
    for $u \in S$: ▷ *see footnote 23*
      for $(v \xleftarrow{f} u_1, \ldots, u_k) \in C.\texttt{out}(u)$:
        if $v \in S$: ▷ *cyclic hyperedge back to S*
          $C_{\text{new}}[v] \oplus_v= f(C[u_1], \ldots, C[u_k])$
    if $C_{\text{new}} = C$: **break** ▷ *deep equality test*
    $C \leftarrow C_{\text{new}}$
  ▷ *Finally, propagate the solution to later SCCs, in case they too are solved with Algorithm 6.*
  for $u \in S$: ▷ *see footnote 23*
    for $(v \xleftarrow{f} u_1, \ldots, u_k) \in C.\texttt{out}(u)$:
      if $v \notin S$: ▷ *hyperedge to later SCC*
        $C[v] \oplus_v= f(C[u_1], \ldots, C[u_k])$

However the SCCs are solved, the point is that solving n SCCs separately (if $n > 1$) is generally faster than applying the same algorithm to solve the entire graph (Purdom, 1970). For example, Tarjan (1981) noted that since Kleene-Floyd-Warshall runs in cubic time on the number of items, the runtimes of these two strategies are proportional to $|\bar{E}| + \sum_i |S_i|^3 < |\bar{E}| + |\bar{V}|^3$, respectively, when the SCCs have sizes $|S_1| + \cdots + |S_n| = |\bar{V}|$. Similarly, Nederhof (2003) suggested running Knuth's algorithm on each SCC separately, giving runtimes proportional to $\sum_i |\bar{E}_i| \log(1+|S_i|) < |\bar{E}| \log(1+|\bar{V}|)$. Applying the relaxation method to the SCCs of an *acyclic* hypergraph recovers the fast Algorithm 3, since each SCC S_i has $|S_i| = 1$ and can be solved to convergence with a single relaxation.

7.2 Finding the SCCs

To solve the SCCs one at a time, we need to enumerate them in toposorted order. Accomplishing this in $\mathcal{O}(|\bar{V}|+|\bar{E}|)$ time is a job for Tarjan's algorithm (Tarjan, 1972). We give two solutions.[20]

[20] Goodman (1999) offers three solutions, but §5.4 rejected them as too expensive even in the acyclic case. In particular, Goodman's solution based on SCC decomposition could employ Tarjan's algorithm, but it materializes the hyperedges first. We use iterators instead to save space.

If efficient `in()` iterators for $\mathcal{P}$ are available, then we can use Algorithm 7, a variant of Algorithm 3 that has been upgraded to use Tarjan's algorithm rather than ordinary depth-first search. Again, this is a two-pass algorithm: (1) compute $\bar{V}$ using unweighted forward chaining (Algorithm 1), and then (2) run a Tarjan-enhanced backward pass to find (and solve) the SCCs in toposorted order.[21] The SCCs that are closest to the axioms are found first, so each SCC can be solved as soon as it is found.

But what if we do not have `in()` iterators that are as efficient as the `out()` iterators (see §5.3)? Fortunately, it is also possible to use pure forward chaining, since the forward graph has the same set of SCCs as the backward graph. This is a three-pass algorithm (Algorithm 8): (1) run Algorithm 1 to find $\bar{V}$, (2) run Tarjan's algorithm on the forward graph to enumerate the SCCs in reverse order, pushing them onto a stack, (3) pop the SCCs off the stack to visit them in the desired order, solving each one when it is popped. For any K, *this method consumes only as much time and space as Algorithm 1*[17]—excluding the extra time it uses for the SOLVESCC calls, which is unavoidable—up to a constant factor, over all inputs and all deduction systems whose hyperedges have in-degree $\leq K$. The constant factor depends only on K (see footnote 22 for why it does).

One might hope to combine passes (1) and (2). That would indeed work if we were dealing with an ordinary graph. However, hypergraphs are trickier. It is crucial for Tarjan's algorithm that when it visits a node u, it *immediately* explores all paths from u, so as to discover any cycle containing u while u is still on the stack. But $C.\texttt{out}(u)$ will not enumerate a hyperedge $v \xleftarrow{f} u, u'$ if u' is not yet known to be in $\bar{V}$, so the necessary exploration from u to v will be blocked. Pass (1) solves this by precomputing the vertices $\bar{V}$. Now calling $C.\texttt{out}(u)$ during pass (2) will not be blocked but will be able to find the hyperedge to v as desired, perhaps looping back through u (or u'). In effect, pass (2) runs Tarjan's algorithm on the implicit *dependency graph* that has an edge $u \to v$ whenever v was derived by combining u with other items during pass (1). Pass (2) can enumerate the dependency edges from u at any time. To enable this, at Algorithm 8 line 13, the $C.\texttt{out}$ iterator must retain state from pass (1) and only require $u_1, \ldots, u_k$ to have been popped previously—not necessarily during pass (2): that is, $u_1, \ldots, u_k \in \bar{V}$.[22] The backward methods also used this more relaxed test (Algorithm 3 line 7 and Algorithm 7 line 11), but Algorithm 5 line 8 did not.

Algorithm 7 Weighted cyclic backward chaining (compare Algorithm 3)

```
1:  ▷ Algorithm 1 has already been run to compute V̄
2:  ▷ Here A is a stack of distinct items. A.index[v] records the current stack position of v (with the bottom and top elements at 0 and |A| − 1 respectively), or takes a special value if v ∉ A.
3:  C ← ∅; A ← ∅ ▷ now C is a map with keys in V̄
4:  for v ∈ V̄ : COMPUTE(v)  ▷ V̄ is the old set C
5:  function COMPUTE(v)
6:      ▷ Set C[v] = ω̄(v), unless any of v's own SCC is on the stack A. In that case, add v and its remaining SCC ancestors to A, setting their C values as required by Algorithm 6 line 2. Returns the # of items at the bottom of the stack that were not detected to be in v's SCC.
7:      low ← |A|  ▷ will become return value
8:      if C[v] = ⊥ :  ▷ first visit
9:          A.push(v)  ▷ we may undo this at line 22
10:         if v ∈ V : C[v] ← ω(v)  ▷ axiom?
11:         for (v ←f u_1, ..., u_k) ∈ C.in(v) :
12:             r ← true  ▷ acyclic hyperedge?
13:             for i ← 1 to k :
14:                 if u_i ∈ A :  ▷ cycle detected
15:                     r ← false; low min= A.index[u_i]
16:                 else low min= COMPUTE(u_i)
17:             if r : C[v] ⊕_v= f(C[u_1], ..., C[u_k])
18:         assert C[v] ≠ ⊥  ▷ because v ∈ V̄
19:         if low = A.index[v] :  ▷ low is unchanged
20:             ▷ nothing beneath v is in v's SCC
21:             S ← ∅  ▷ pop v's entire SCC into this set
22:             while |A| > low : S.add(A.pop())
23:             SOLVESCC(S)
24:     return low
```

[21] Algorithm 7 uses a somewhat nonstandard presentation of Tarjan's that exposes a helpful contract at line 6. Note that A no longer serves as an agenda of items to explore in future, but as a stack of items that are currently being explored. An item waits on the stack because we are backward chaining and its inputs are not yet all known. It always depends transitively on all items above it on the stack. This is because each item v on the stack depends on the item immediately above v either directly, if COMPUTE(v) is still in progress (hence that item is some u_i), or else transitively, via some item below v (then called $A[low]$) that kept v from popping when COMPUTE(v) returned.

[22] As a result, this hyperedge will be visited k times, but only the first time will have an effect, due to the test at line 11.

[23] Each time we arrive at line 9 or 16, the loop must enumerate each hyperedge from S exactly once. Under our iterator design (§2.5), this can be arranged if we iterate through $u \in S$

Algorithm 8 Weighted cyclic forward chaining (compare Algorithm 7)

▷ *Algorithm 1 has already been run to compute* $\bar{V}$
▷ *Again* A *is a stack of distinct items.*
$C \leftarrow \emptyset$; $A \leftarrow \emptyset$ ▷ *here* $C \subseteq \bar{V}$ *is a set*
$\mathcal{T} \leftarrow \emptyset$ ▷ *toposorted stack of SCCs*
for $v \in V$: FINDNEWSCCS(v) ▷ *pass (2)*
$C \leftarrow \omega$ ▷ *now* C *is a map with keys in* $\bar{V}$
while $\mathcal{T} \neq \emptyset$: SOLVESCC($\mathcal{T}$.pop()) ▷ *pass (3)*
function FINDNEWSCCS(u)
  ▷ *Push onto* $\mathcal{T}$ *all SCCs that are reachable from* u *and not yet in* $\mathcal{T}$*, unless any of* u*'s own SCC is on the stack* A*. In that case, just add* u *and its remaining SCC descendants to* A*. Returns the # of items at the bottom of the stack that were* not *detected to be in* u*'s SCC.*
  $low \leftarrow |A|$ ▷ *will become return value*
  if $u \notin C$: ▷ *first visit*
    C.add(u); A.push(u) ▷ *we may undo push*
    for $(v \xleftarrow{f} u_1, \ldots, u_k) \in C$.out($u$) :
      if $v \in A$: low min= A.index[v]
      else low min= FINDNEWSCCS(v)
    if $low = A$.index[u] : ▷ *low is unchanged*
      ▷ *nothing beneath* u *is in* u*'s SCC*
      $S \leftarrow \emptyset$ ▷ *pop* u*'s entire SCC into this set*
      while $|A| > low$: S.add(A.pop())
      $\mathcal{T}$.push(S)
  return low

8 Limitations

We have not discussed parallel execution beyond tensorization of a deduction system (§2.4). But in forward chaining, one could pop multiple items from the agenda and then process them simultaneously in multiple threads, taking care to avoid write conflicts.[24] Indeed, *semi-naive bottom-up evaluation*, a classical strategy for evaluating unweighted deduction systems (Ceri et al., 1990), is equivalent to processing *all* items on the agenda in parallel at each step. For backward chaining, Matei (2016) reviews concurrent versions of Tarjan's algorithm.

We have not discussed dynamic algorithms, where the axioms or their weights can be updated after or during evaluation, and one wishes to update the derived items and their weights without recomputing them from scratch (Eisner et al., 2005; Filardo and Eisner, 2012).

Finally, §2.2 defined evaluation to return all of $\bar{V}$. We did not discuss **query-driven evaluation**, which returns only $\bar{V} \cap Q$ (and corresponding weights) for a given set of **query items** $Q \subseteq \mathcal{V}$. This can allow termination even in some cases with infinite $\bar{V}$. Intuitively, we hope to backward-chain from Q while also forward-chaining from V. This can be achieved by a technique such as **magic sets** (Beeri and Ramakrishnan, 1991), which transforms the deduction system $\mathcal{P}$ to (1) accept additional input axioms that specify the queries Q, (2) derive additional items corresponding to recursive queries, (3) augment the original hyperedges so that they will derive an original item v only when it answers a query. Evaluation of the new system $\mathcal{P}'$ requires only forward chaining via out() (e.g., our Algorithms 4–6 and 8)—but forward chaining for (2) simulates backward chaining under $\mathcal{P}$. As a result, the out() iterators of $\mathcal{P}'$ must do additional work corresponding to what the in() iterators of $\mathcal{P}$ did.

9 Conclusions

Often an NLP algorithm can be expressed as a weighted deduction system (§2) that defines how various quantities should be computed from one another. Such a system is usually presented as a small collection of rules that combine to generate a large or even infinite computation graph (really a hypergraph). The rules specify concisely what is to be computed, leaving the details of scheduling and data structures to generic strategies that are well-understood, reliable, and easy to analyze.

We adopted a general formalism for weighted deduction systems (going beyond the usual semiring-weighted case) and surveyed a range of generic strategies that can be interpreted or compiled for a given system when seeking a practical CPU implementation. All of the strategies were presented uniformly in terms of a chart C and agenda A.

Our main novel contribution was to exhibit a generic strategy for acyclic *weighted* deduction that is guaranteed to be as time- and space-efficient *on every system and input* as the corresponding strategy for *unweighted* deduction, up to a constant factor that is independent of the system and input. We also exhibited a strategy with related guarantees for the more general case of cyclic weighted deduction.

by pushing S's items onto an agenda and then popping them off, provided that we first roll back the state of the C.out iterators as if S's items had never previously popped.

[24] Caveat: The out iterator must avoid duplicates when finding the hyperedges out of the just-popped set. E.g., a hyperedge like $v \xleftarrow{f} u, u', u, u''$ should be found only once if u' and u'' are jointly popped after u, just as it should be found only once if u is popped alone after u' and u'' (see §2.5).

Acknowledgements

Tim Vieira, Andreas Opedal, and Matthew Francis-Landau provided useful discussion. In particular, Tim suggested discussing bucket queues in §5.1. The anonymous reviewers contributed several useful suggestions and questions about the presentation, which I have tried to address. Any remaining errors are my own.

References

J. K. Baker. 1979. Trainable grammars for speech recognition. In *Speech Communication Papers Presented at the 97th Meeting of the Acoustical Society of America*.

Catriel Beeri and Raghu Ramakrishnan. 1991. On the power of magic. *Journal of Logic Programming*, 10(3):255–299.

Richard Bellman. 1958. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90.

Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1990. *Logic Programming and Databases*. Springer.

Robert B. Dial. 1969. Algorithm 360: Shortest-path forest with topological ordering. *Communications of the ACM*, 12(11):632–633.

Edsger W. Dijkstra. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Andrew Drozdov, Patrick Verga, Mohit Yadav, Mohit Iyyer, and Andrew McCallum. 2019. Unsupervised latent tree induction with deep inside-outside recursive auto-encoders. In *Proceedings of NAACL: HLT*, pages 1129–1141.

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

Jason Eisner and John Blatz. 2007. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proceedings of FG 2006: The 11th Conference on Formal Grammar*, pages 45–85. CSLI Publications.

Jason Eisner and Nathaniel W. Filardo. 2011. Dyna: Extending Datalog for modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 181–220. Springer.

Jason Eisner, Eric Goldlust, and Noah A. Smith. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 281–290.

Nathaniel Wesley Filardo and Jason Eisner. 2012. A flexible solver for finite arithmetic circuits. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP)*, volume 17 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 425–438.

C. L. Forgy. 1979. *On the Efficient Implementation of Production Systems*. Ph.D. thesis, Carnegie-Mellon University Department of Computer Science.

Michael L. Fredman and Robert E. Tarjan. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615. Announced at FOCS'84.

Giorgio Gallo, Giustino Longo, Stefano Pallottino, and Sang Nguyen. 1993. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42:177–201.

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics*, 25(4):573–605.

Susan L. Graham, Michael A. Harrison, and Walter L. Ruzzo. 1980. An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462.

Arthur B. Kahn. 1962. Topological sorting of large networks. *Commmunications of the ACM*, 5(11):558–562.

Tadao Kasami. 1965. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

Donald E. Knuth. 1977. A generalization of Dijkstra's algorithm. *Information Processing Letters*, 6(1):1–5.

Daniel J. Lehmann. 1977. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76.

Vera Matei. 2016. Parallel algorithms for detecting strongly connected components. Master's thesis, Vrije Universiteit Amsterdam.

David McAllester. 2002. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4):512–537.

Bobby McFerrin. 1988. Don't worry, be happy. In *Simple Pleasures*. Manhattan Records.

Mark-Jan Nederhof. 2003. Weighted deductive parsing and Knuth's algorithm. *Computational Linguistics*, 29(1):135–143.

Andreas Opedal, Ran Zmigrod, Tim Vieira, Ryan Cotterell, and Jason Eisner. 2023. Efficient semiring-weighted Earley parsing. In *Proceedings of ACL*.

Fernando C. N. Pereira and David H. D. Warren. 1983. Parsing as deduction. In *Proceedings of ACL*, pages 137–144.

Mark Perlin. 1990. Topologically traversing the RETE network. *Applied Artificial Intelligence*, 4(3):155–177.

Paul Purdom, Jr. 1970. A transitive closure algorithm. *BIT Numerical Mathematics*, 10(1):76–94.

Taisuke Sato. 1995. A statistical learning method for logic programs with distribution semantics. In *Proceedings of ICLP*, pages 715–729.

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

Klaas Sikkel. 1993. *Parsing Schemata*. Ph.D. thesis, University of Twente, Enschede, The Netherlands. Published as a book by Springer-Verlag in 1997.

Andreas Stolcke. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201.

Volker Strassen. 1969. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356.

Robert E. Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1(2):146–160.

Robert Endre Tarjan. 1981. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614.

Mikkel Thorup. 2000. On RAM priority queues. *SIAM Journal on Computing*, 30(1):86–109.

Tim Vieira, Ryan Cotterell, and Jason Eisner. 2021. Searching for more efficient dynamic programs. In *Findings of EMNLP'21*, pages 3812–3830.

Tim Vieira, Ryan Cotterell, and Jason Eisner. 2022. Automating the analysis of parsing algorithms (and other dynamic programs). *Transactions of the Association for Computational Linguistics*. Accepted for publication.