Time-and-Space-Efficient Weighted Deduction

CENTER FOR LANGUAGE AND SPEECH PROCESSING
JOHNS HOPKINS UNIVERSITY

Jason Eisner, JHU

See algorithm animations and dialogue in the talk video!

Meta-Theorem

- Can *weighted deduction* be made as efficient as *unweighted deduction*?
  - Only a constant factor worse in **time** and **space** …
  - … for every deduction system and every input?
- For acyclic deduction: Yes!
- For cyclic deduction: Almost!
  - Plus time to solve the strongly connected components
  - But you can find those fast, in toposorted order

Example: Parsing — Opedal et al. (ACL 2023)

Regular Earley's

$$\textsc{Pred}: \frac{B \to \rho}{[j,j,B \to \bullet\rho]}\ [i,j,A \to \mu \bullet B\,\nu]$$

$$\textsc{Scan}: \frac{[i,j,A \to \mu \bullet a\,\nu] \quad [j,k,a]}{[i,k,A \to \mu\, a \bullet \nu]}$$

$$\textsc{Comp}: \frac{[i,j,A \to \mu \bullet B\,\nu] \quad [j,k,B \to \rho\bullet]}{[i,k,A \to \mu\, B \bullet \nu]}$$

$O(n^3|\mathcal{G}||\mathcal{R}|)$

Fast Earley's

$$\textsc{Pred1}: \frac{}{[j,j,B \to \bullet\star]}\ [i,j,A \to \mu \bullet B\,\nu]$$

$$\textsc{Pred2}: \frac{B \to \rho}{[j,j,B \to \bullet\rho]}\ [j,j,B \to \bullet\star]$$

$$\textsc{Scan}: \frac{[i,j,A \to \mu \bullet a\,\nu] \quad [j,k,a]}{[i,k,A \to \mu\, a \bullet \nu]}$$

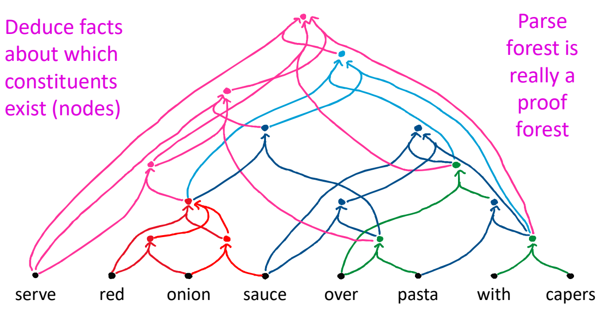
$$\textsc{Comp1}: \frac{[j,k,B \to \rho\bullet]}{[j,k,B \to \star\bullet]}$$

$$\textsc{Comp2}: \frac{[i,j,A \to \mu \bullet B\,\nu] \quad [j,k,B \to \star\bullet]}{[i,k,A \to \mu\, B \bullet \nu]}$$

$O(n^3|\mathcal{G}|)$

But faster for some grammars and sentences, thanks to sparsity. Not obvious how to extend this to probabilistic or weighted parsing, achieving same runtime and space bounds for all classes of inputs.

Parsing as deduction



Axioms are facts about input words and grammar rules

What's a deduction system?

- Set of **rules** that deduce new facts from old
  - They're translated into iterators that can give any node's in-hyperedges and out-hyperedges
  - Rules are usually written in a pattern-matching language like Datalog or Dyna

How about weights?

- Turn the proof forest into a computation graph!
- Each hyperedge is labeled with a function that will be applied to the hyperedge's inputs
- Each node's weight pools the function values from all its in-hyperedges, using that node's aggregation operator, such as + or min (must be associative & commutative)

Useful weight types
- Embeddings
- Counts
- Probabilities
- Beliefs
- Entropies
- Derivations
- Translations

…

CKY parsing written with Dyna rules

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

Applications (see Eisner & Filardo, 2011)

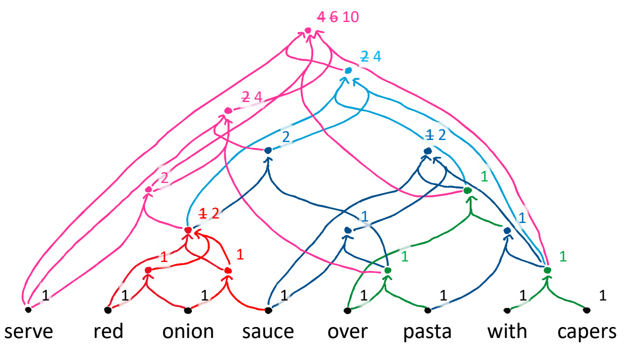
- Nearly all algorithms in formal language theory (parsing, automata, grammar transforms, weighted edit distance, …)
- Systematic search (backtracking with constraint propagation and branch & bound)
- Neural networks (rules specify architecture)
- Iterative methods (loopy belief propagation)
- Reinforcement learning (MDP)
- …

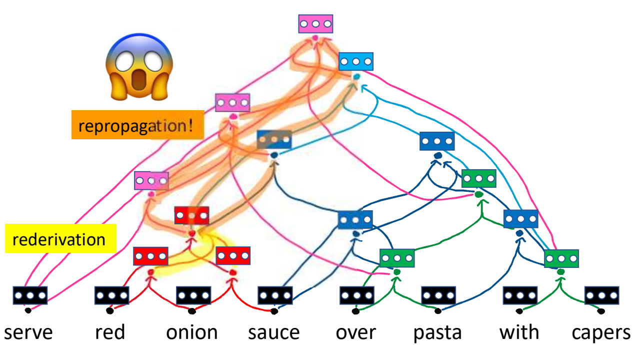
How can we prove facts?

- **Forward chaining**, starting at axioms (Alg 1)
- **Chart** C is set of nodes found so far
  - Reached by following hyperedges that combine other nodes from C
- **Agenda** A is a queue of nodes in C that still have unfollowed out-hyperedges
- At each step, pop a node from A, combine with previously popped nodes (they are in C)
  - Add any resulting new nodes to C and A

How about weights?

- C now maps each node v that has been found to its **weight so far** (the pooled value of its in-hyperedges found so far)
- This pooled value at v is updated …
  1. Each time a new in-hyperedge to v is found
  2. But also, each time an existing hyperedge changes its value because its input weights have been updated!
     - We hope this never happens, as it increases our runtime to process the same node multiple times
     - If the hypergraph is acyclic, we can prevent it by popping nodes from A in topologically sorted order. (But how do we do that???)

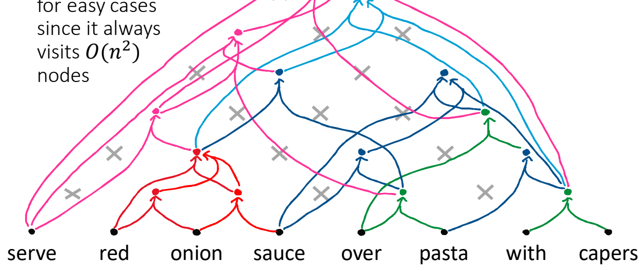




Ideas that don't quite work

- Hopeful forward chaining (Alg 2)
  - No guarantee of topological order
  - So may throw an exception
- Prioritized forward chaining (Goodman 1999)
  - Not generic – must devise a topologically sorting priority function for *each* deduction system
  - *Bucket priority queue*: visits every priority level, may do unnecessary work and break runtime
  - *Heap priority queue*: visits only occupied levels, but log-factor overhead, which breaks runtime
- Dynamic programming tabulation
  - Visits underived nodes, which breaks runtime
- Unweighted forward chaining followed by weighted backward chaining (Algs 1+3)
  - Goodman 1999
  - But backward pass must find in-edges
    - Store them on forward pass (more space)
    - Or recompute them (breaks runtime in pathological cases where in-edges are harder to compute than out-edges)

CKY tabulation can't get $O(n)$ for easy cases since it always visits $O(n^2)$ nodes



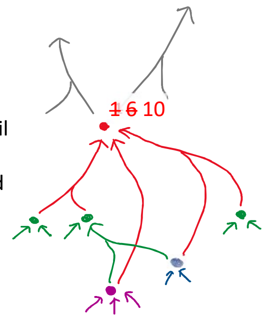
Space $O(|V|)$
Time: $O(|V| + |E|)$

(assuming fast iterators and small weights)

Linear, hooray!

- V = vertices found
- E = hyperedges found

Wait to propagate until value has converged (received all inputs)



Multiple low-space forward passes

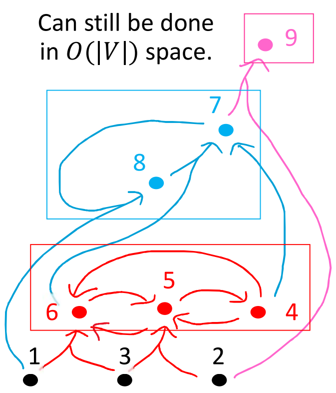
- Unweighted forward chaining followed by weighted forward chaining
  - First pass discovers graph, finding all nodes
  - For space efficiency, don't store the hyperedges, but each node does store its count of in-hyperedges (Alg 4)
  - Second pass decrements this counter as it finds the same hyperedges again (Alg 5)
  - Node is pushed onto the agenda only when its counter reaches 0 (weight converged)
  - Kahn 1962 but on an unmaterialized graph
- Unweighted forward chaining followed by toposorted SCC decomposition
  - Needed for cyclic case
  - First pass as above (without the counting)
  - On second pass, use Tarjan's (1972) algorithm to enumerate all SCCs in (reverse) topologically sorted order (Alg 8)
  - Derive each SCC only from SCCs that have already converged (Alg 6)

Toposorting nodes only works on an acyclic graph …

Tarjan (1972) is like backward chaining (Alg 3) but discovers cycles. It returns SCCs in toposorted order (Alg 7).

To avoid expensive in-hyperedges, we can run it on reversed graph (same SCCs).

Can still be done in $O(|V|)$ space.



Key references

Unweighted deduction
Prolog (Colmerauer & Roussel 1972), Datalog (Ceri et al. 1990)
Parsing as Deduction (Pereira & Warren 1983; Sikkel 1993; Shieber, Schabes, & Pereira 1995)
Transformations of deduction systems (e.g., Beeri & Ramakrishnan 1991)
Static analysis of deduction systems (McAllester, 2002; Vieira et al. 2021, 2022)
Weighted deduction
Min-weighted deduction (Nederhof 2003)
Probability-weighted deduction (Sato 1995)
Semiring-weighted deduction (Goodman 1999; Eisner et al. 2005)
Generalized weighted deduction (Filardo & Eisner 2011)
Transformations of deduction systems (Eisner & Blatz 2007)
Graph algorithms
Topological sorting (Kahn 1962)
Discovery & toposorting of strongly connected components (Tarjan 1972)
Solving strongly connected components (e.g., Lehmann 1977)

Algorithm 1 Unweighted forward chaining

1: $C \leftarrow \emptyset$; $A \leftarrow \emptyset$ ▷ *here $C \subseteq \bar{V}$ is just a set*
2: **for** $v \in V$: ▷ *axioms*
3: C.add(v); A.push(v)
4: **while** $A \neq \emptyset$:
5: $u \leftarrow A$.pop() ▷ *remove some element*
6: **for** $(v \xleftarrow{f} u_1, \ldots, u_k) \in C$.out($u$) :
7: **if** $v \notin C$: ▷ *also implies $v \notin A$*
8: C.add(v); A.push(v)

Algorithm 2 Weighted forward chaining

1: $C \leftarrow \emptyset$ ▷ *map with keys in $\bar{V}$, values in $\mathbb{W}$*
2: $A \leftarrow \emptyset$
3: **for** $v \in V$: ▷ *axioms*
4: $C[v] \oplus_v\!= \omega(v)$
5: A.push(v)
6: **while** $A \neq \emptyset$:
7: $u \leftarrow A$.pop() ▷ *remove some element*
8: **for** $(v \xleftarrow{f} u_1, \ldots, u_k) \in C$.out($u$) :
9: $C[v] \oplus_v\!= f(C[u_1], \ldots, C[u_k])$
10: **if** $C[v]$ changed :
11: **if** v has already popped from A : **error**
12: **if** $v \notin A$: A.push(v)

Algorithm 3 Weight computation by backward chaining

1: ▷ *Algorithm 1 has already been run to compute $\bar{V}$*
2: $C \leftarrow \emptyset$ ▷ *now C is a map with keys in $\bar{V}$*
3: **for** $v \in \bar{V}$: COMPUTE(v) ▷ *$\bar{V}$ is the old set C*
4: **procedure** COMPUTE(v)
5: **if** $C[v] = \bot$: ▷ *first visit*
6: ▷ *this iterator requires $u_1, \ldots, u_k$ to have been popped from Algorithm 1's agenda*
7: **for** $(v \xleftarrow{f} u_1, \ldots, u_k) \in C$.in($v$) :
8: **for** $i \leftarrow 1$ **to** k : COMPUTE(u_i)
9: C.relax(v)
10: **assert** $C[v] \neq \bot$ ▷ *because $v \in \bar{V}$*

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

1: $C \leftarrow \emptyset$; $A \leftarrow \emptyset$ ▷ *here $C \subseteq \bar{V}$ is just a set*
2: **for** $v \in V$: ▷ *axioms*
3: C.add(v); A.push(v)
4: *waiting_edges*$[v]$ += 1 ▷ *# summands needed*
5: **while** $A \neq \emptyset$:
6: $u \leftarrow A$.pop() ▷ *remove some element*
7: *waiting_items* += 1 ▷ *# items popped*
8: **for** $(v \xleftarrow{f} u_1, \ldots, u_k) \in C$.out($u$) :
9: **if** *waiting_edges*$[v] = 0$: ▷ *$v \notin C$*
10: C.add(v); A.push(v)
11: *waiting_edges*$[v]$ += 1

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

1: ▷ *Algorithm 4 has already computed waiting_edges[v] and waiting_items.*
2: $C \leftarrow \emptyset$; $A \leftarrow \emptyset$ ▷ *now C is a map with keys in $\bar{V}$*
3: **for** $v \in V$: ▷ *axioms*
4: CONTRIBUTE(v, $\omega(v)$)
5: **while** $A \neq \emptyset$:
6: $u \leftarrow A$.pop() ▷ *remove some element*
7: *waiting_items* −= 1 ▷ *# items unpopped*
8: **for** $(v \xleftarrow{f} u_1, \ldots, u_k) \in C$.out($u$) :
9: CONTRIBUTE(v, $f(C[u_1], \ldots, C[u_k])$)
10: **if** *waiting_items* $\neq 0$: **error** ▷ *cycle detected*
11: **procedure** CONTRIBUTE(v,w)
12: $C[v] \oplus_v\!= w$
13: *waiting_edges*$[v]$ −= 1 ▷ *# summands needed*
14: **if** *waiting_edges*$[v] = 0$:
15: A.push(v) ▷ *delayed push: item is ready*

Algorithm 6 Solving an SCC using only out()

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

Algorithm 7 Weighted cyclic backward chaining (compare Algorithm 3)

1: ▷ *Algorithm 1 has already been run to compute $\bar{V}$*
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 $v \notin A$.*
3: $C \leftarrow \emptyset$; $A \leftarrow \emptyset$ ▷ *now C is a map with keys in $\bar{V}$*
4: **for** $v \in \bar{V}$: COMPUTE(v) ▷ *$\bar{V}$ is the old set C*
5: **function** COMPUTE(v)
6: ▷ *Set $C[v] = \bar{\omega}(v)$, 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* $\leftarrow |A|$ ▷ *will become return value*
8: **if** $C[v] = \bot$: ▷ *first visit*
9: A.push(v) ▷ *we may undo this at line 22*
10: **if** $v \in V : C[v] \leftarrow \omega(v)$ ▷ *axiom?*
11: **for** $(v \xleftarrow{f} u_1, \ldots, u_k) \in C$.in($v$) :
12: $r \leftarrow$ true ▷ *acyclic hyperedge?*
13: **for** $i \leftarrow 1$ **to** k :
14: **if** $u_i \in A$: ▷ *cycle detected*
15: $r \leftarrow$ false; *low* min= A.index$[u_i]$
16: **else** *low* min= COMPUTE(u_i)
17: **if** $r : C[v] \oplus_v\!= f(C[u_1], \ldots, C[u_k])$
18: **assert** $C[v] \neq \bot$ ▷ *because $v \in \bar{V}$*
19: **if** *low* $= A$.index$[v]$: ▷ *low is unchanged*
20: ▷ *nothing beneath v is in v's SCC*
21: $S \leftarrow \emptyset$ ▷ *pop v's entire SCC into this set*
22: **while** $|A| >$ *low* : S.add(A.pop())
23: SOLVESCC(S)
24: **return** *low*

Algorithm 8 Weighted cyclic forward chaining (compare Algorithm 7)

1: ▷ *Algorithm 1 has already been run to compute $\bar{V}$*
2: ▷ *Again A is a stack of distinct items.*
3: $C \leftarrow \emptyset$; $A \leftarrow \emptyset$ ▷ *here $C \subseteq \bar{V}$ is a set*
4: $\mathcal{T} \leftarrow \emptyset$ ▷ *toposorted stack of SCCs*
5: **for** $v \in V$: FINDNEWSCCS(v) ▷ *pass (2)*
6: $C \leftarrow \omega$ ▷ *now C is a map with keys in $\bar{V}$*
7: **while** $\mathcal{T} \neq \emptyset$: SOLVESCC($\mathcal{T}$.pop()) ▷ *pass (3)*
8: **function** FINDNEWSCCS(u)
9: ▷ *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.*
10: *low* $\leftarrow |A|$ ▷ *will become return value*
11: **if** $u \notin C$: ▷ *first visit*
12: C.add(u); A.push(u) ▷ *we may undo push*
13: **for** $(v \xleftarrow{f} u_1, \ldots, u_k) \in C$.out($u$) :
14: **if** $v \in A$: *low* min= A.index$[v]$
15: **else** *low* min= FINDNEWSCCS(v)
16: **if** *low* $= A$.index$[u]$: ▷ *low is unchanged*
17: ▷ *nothing beneath u is in u's SCC*
18: $S \leftarrow \emptyset$ ▷ *pop u's entire SCC into this set*
19: **while** $|A| >$ *low* : S.add(A.pop())
20: $\mathcal{T}$.push(S)
21: **return** *low*