(Weighted) Regular DAG Languages
Properties and Algorithms

WATA 2018

F. Drewes
(joint work with many others: M. Berglund, H. Björklund, J. Blum, D. Chiang, D. Gildea, A. Lopez, G. Satta)

Part 0

Introduction

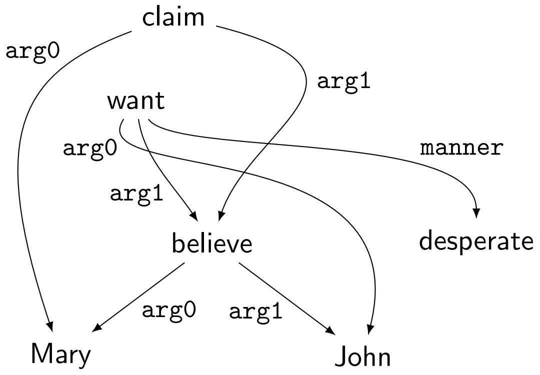
Motivation: Natural Language Semantics

Background Abstract Meaning Representation (AMR, Banarescu et al. 2013) represents sentence meaning as directed (acyclic) graphs.

Goal Develop appropriate types of automata for such structures, generalizing ordinary finite automata and tree automata, with and without weights.

Mindset Do not kling too much to the informal description of AMR. Instead, focus on the essentials to create a theory with good computational and structural properties.

Motivation: Natural Language Semantics



"John desperately wants Mary to believe him. She claims she does."

[Directed acyclic graph (DAG) inspired by AMR]

Existing Approaches

Existing notions of DAG and general graph automata:

- Kamimura & Slutzki 1981
- Thomas 1991
- Charatonik 1999 and Anantharaman et al. 2005
- Priese 2007
- Fujiyoshi 2010
- Quernheim & Knight 2012
- Bailly et al. 2018
- . . . and a few others.

Why Propose Yet Another Approach?

None of the previous approaches seems ideal for handling AMR-like graph languages. In particular, we do not want much power.

A partial wish list:

1. path languages should be regular,
2. Parikh images should be similinear,
3. emptiness and finiteness should be efficiently decidable,
4. there should be efficient membership tests, and
5. the weighted case should be a natural extension.

(In general, we are going to fail at 4.)

The Remainder of this Tutorial

Types of DAG languages covered in the remaining parts:

Parts 1 & 2: Unweighted DAG languages, ordered and of bounded degree.

Parts 3 & 4: Weighted DAG languages, unordered and (eventually) of unbounded degree.

Part 1

DAG automata

The basic case and its properties

Type(s) of DAGs considered:

- Labels are on the nodes.
- For simplicity, edges are unlabelled.
- The outgoing/incoming edges of a node are ordered.
- There are (of course) no directed cycles.

These choices (except the last) are not too important:

- Edge labels can easily be added.
- Unordered DAGs instead of ordered ones can be considered without essential changes.$^{(*)}$

$^{(*)}$ except that deterministic automata do not make sense anymore

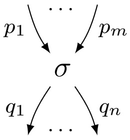
Defining DAG automata

Runs (=computations) assign states to edges.

A rule for a symbol σ, also σ-rule, takes the form

$$\underbrace{p_1 \cdots p_m}_{\substack{\uparrow \\ \text{states on} \\ \text{incoming edges}}} \quad \xrightarrow{\sigma} \quad \underbrace{q_1 \cdots q_n}_{\substack{\uparrow \\ \text{states on} \\ \text{outgoing edges}}} .$$

A run is an assignment of states to edges. It is accepting if it, at each node, coincides with a rule:



The Accepted DAG Language

Regular DAG Language

Automaton A accepts DAG D if D has an accepting run.

The DAG language $L(A)$ of A consists of all nonempty connected DAGs that A accepts.

Such a DAG language is called a regular DAG language.

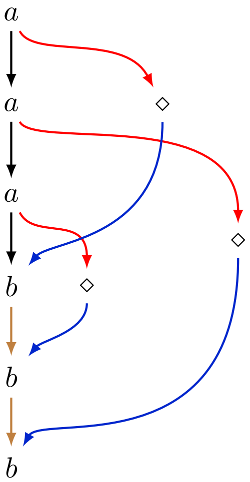
Remark: We may alternatively view A as a reglar DAG grammar that generates DAGs top-down (or bottom-up).

Worthwhile pointing out:

- Rules of the form $\lambda \xrightarrow{\sigma} q_1 \cdots q_n$ and $p_1 \cdots p_m \xrightarrow{\sigma} \lambda$ process roots/leaves (no initial/final states are needed).
- Ordinary tree automata "are" those DAG automata in which $|I| \leq 1$ for all rules $I \xrightarrow{\sigma} O$.
- Regular DAG languages are of bounded node degree.
- We restrict $L(A)$ to nonempty and connected DAGs because A accepts D iff it accepts all connected components of D.
- In particular, the restriction makes it meaningful to talk about emptiness and finiteness of regular DAG languages.
- The automata would work on cyclic graphs as well, but we exclude them.

An Example

Example



$$\emptyset \xrightarrow{a} \{\bullet, {\color{red}\bullet}\}$$

$$\{\bullet\} \xrightarrow{a} \{\bullet, {\color{red}\bullet}\}$$

$$\{{\color{red}\bullet}\} \xrightarrow{\diamond} \{{\color{blue}\bullet}\}$$

$$\{\bullet, {\color{blue}\bullet}\} \xrightarrow{b} \{{\color{brown}\bullet}\}$$

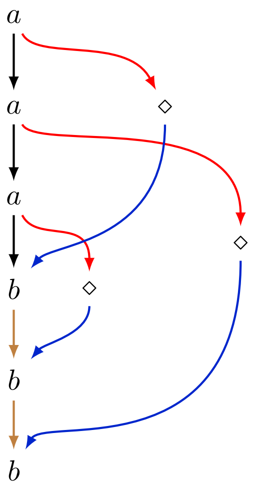
$$\{{\color{brown}\bullet}, {\color{blue}\bullet}\} \xrightarrow{b} \{{\color{brown}\bullet}\}$$

$$\{{\color{brown}\bullet}, {\color{blue}\bullet}\} \xrightarrow{b} \emptyset$$

$$\begin{aligned} &\mathit{paths}(L(A)) \cap \{a, b\}^* \\ &= \{a^n b^n \mid n > 0\} \end{aligned}$$

(likewise for $a^n b^n c^n$ etc)

Example



$$\emptyset \xrightarrow{a} \{\bullet, \textcolor{red}{\bullet}\}$$
$$\{\bullet\} \xrightarrow{a} \{\bullet, \textcolor{red}{\bullet}\}$$

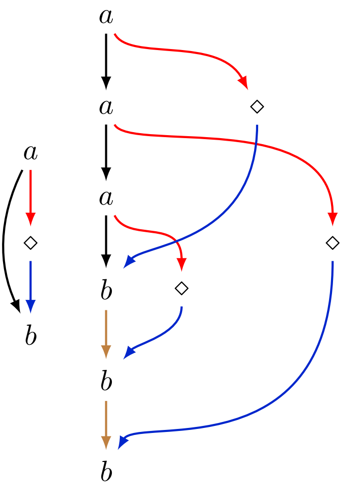
$$\{\textcolor{red}{\bullet}\} \xrightarrow{\diamond} \{\textcolor{blue}{\bullet}\}$$

$$\{\bullet, \textcolor{blue}{\bullet}\} \xrightarrow{b} \{\textcolor{brown}{\bullet}\}$$
$$\{\textcolor{brown}{\bullet}, \textcolor{blue}{\bullet}\} \xrightarrow{b} \{\textcolor{brown}{\bullet}\}$$
$$\{\textcolor{brown}{\bullet}, \textcolor{blue}{\bullet}\} \xrightarrow{b} \emptyset$$

~~$paths(L(A)) \cap \{a, b\}^*$~~
~~$= \{a^n b^n \mid n > 0\}$~~

~~(likewise for $a^n b^n c^n$ etc)~~

Example



$$\emptyset \xrightarrow{a} \{\bullet, \textcolor{red}{\bullet}\}$$

$$\{\bullet\} \xrightarrow{a} \{\bullet, \textcolor{red}{\bullet}\}$$

$$\{\textcolor{red}{\bullet}\} \xrightarrow{\diamond} \{\textcolor{blue}{\bullet}\}$$

$$\{\bullet, \textcolor{blue}{\bullet}\} \xrightarrow{b} \{\textcolor{brown}{\bullet}\}$$

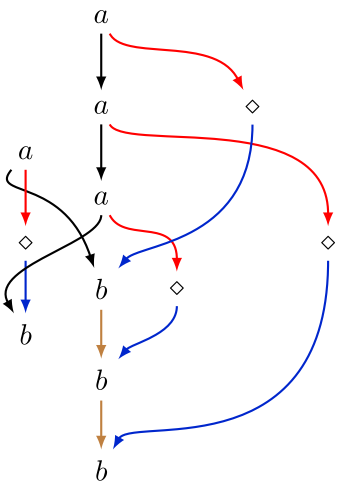
$$\{\textcolor{brown}{\bullet}, \textcolor{blue}{\bullet}\} \xrightarrow{b} \{\textcolor{brown}{\bullet}\}$$

$$\{\textcolor{brown}{\bullet}, \textcolor{blue}{\bullet}\} \xrightarrow{b} \emptyset$$

$\cancel{paths(L(A)) \cap \{a, b\}^*}$

$\cancel{= \{a^n b^n \mid n > 0\}}$

~~(likewise for $a^n b^n c^n$ etc)~~



$$\emptyset \xrightarrow{a} \{\bullet, {\color{red}\bullet}\}$$
$$\{\bullet\} \xrightarrow{a} \{\bullet, {\color{red}\bullet}\}$$

$$\{{\color{red}\bullet}\} \xrightarrow{\diamond} \{{\color{blue}\bullet}\}$$

$$\{\bullet, {\color{blue}\bullet}\} \xrightarrow{b} \{{\color{brown}\bullet}\}$$
$$\{{\color{brown}\bullet}, {\color{blue}\bullet}\} \xrightarrow{b} \{{\color{brown}\bullet}\}$$
$$\{{\color{brown}\bullet}, {\color{blue}\bullet}\} \xrightarrow{b} \emptyset$$

$\cancel{paths(L(A)) \cap \{a, b\}^*}$
$\cancel{= \{a^n b^n \mid n > 0\}}$

~~(likewise for $a^n b^n c^n$ etc)~~

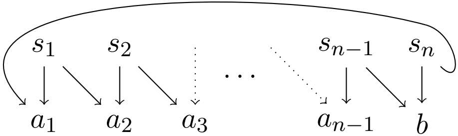
Swapping edges with equal states.
Note that we now have two roots!

Swapping Is a Useful Technique

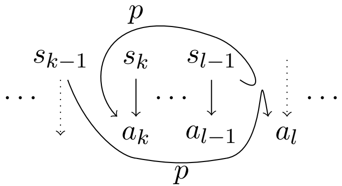
Non-closedness under Complement

Consider binary roots labelled by s and binary leaves labelled by a or b.

The language of DAGs not containing any b is clearly regular. Suppose its complement (DAGs containing at least one b-labelled leaf) is regular:



is in the language. For large n a state p occurs twice. Swapping yields:

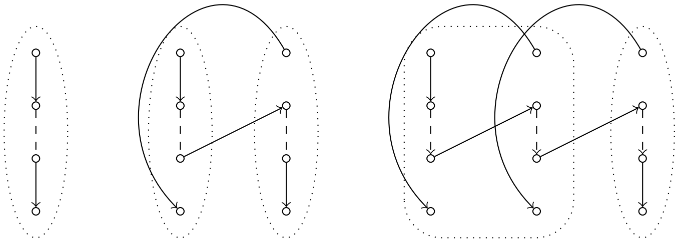


$\Rightarrow$ both connected components are in the language,

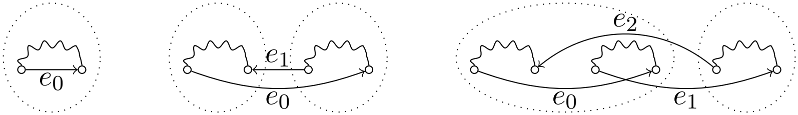
but only one contains a b.

Two Pumping Lemmata Obtained by Swapping

Large DAGs can be pumped by swapping edges between copies:



Undirected cycles always allow to pump:



What a Difference a Root Makes

What a Difference a Root Makes

All (?) earlier notions of DAG automata can restrict the number of roots.

What happens if we add this ability?

	this model	restricted to single root
emptiness	polynomial [3, 2]	decidable [4]
finiteness	polynomial [2]	decidable [1]
path language	regular [3, 2]	not context-free (related to multicounter automata) [1]
unfolding	regular tree lang. [2]	? (but not context-free)
Parikh image	semi-linear [1]	
membership	NP-complete [3]	

From DAGs to Trees to Strings

Unfolding a DAG D from a node v recursively yields a (unique) tree: if v has label σ and outgoing edges to $v_1, \ldots, v_k$ then

$$tree_D(v) = \sigma(tree_D(v_1), \ldots, tree_D(v_k)).$$

Theorem

For every DAG automaton A the tree language

$tree(L(A)) = \{tree_D(v) \mid D \in L(A)$ and v is a root of $D\}$

is regular. Consequently the path language of $L(A)$ is a regular string language.

Proving Regularity of $\mathit{tree}(L(A))$

Proof: Assume that A does not contain useless rules. Turn A into a tree automaton B with the following rules:

$$\begin{array}{rll} \lambda \xrightarrow{\sigma} q_1 \cdots q_n & \text{for every rule } \lambda \xrightarrow{\sigma} q_1 \cdots q_n \text{ of } A \\ (p_i) \xrightarrow{\sigma} q_1 \cdots q_n & \text{for every rule } p_1 \cdots p_m \xrightarrow{\sigma} q_1 \cdots q_n \text{ of } A \\ & \text{and } 1 \leq i \leq m \end{array}$$

Then $\mathit{tree}(L(A)) = L(B)$. The direction $\mathit{tree}(L(A)) \subseteq L(B)$ should be obvious.

Proof sketch of $L(B) \subseteq \mathit{tree}(L(A))$: next slide.

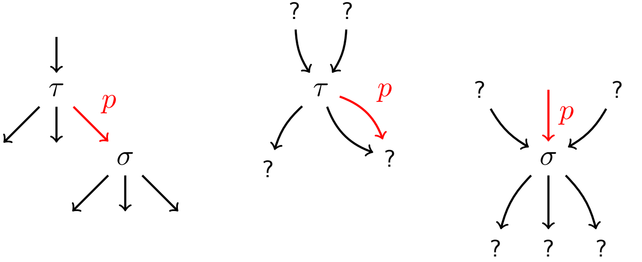
Proving Regularity of $tree(L(A))$

Consider a run of B on a tree t.

- For every node v, if $p_i \xrightarrow{\sigma} q_1 \cdots q_n$ is used at v, choose a run on a DAG D_v using $p_1 \cdots p_m \xrightarrow{\sigma} q_1 \cdots q_n$ at (a copy of) v.
- Similarly, if v is the root and $\lambda \xrightarrow{\sigma} q_1 \cdots q_n$ is used at v, choose a run on a DAG D_v using $\lambda \xrightarrow{\sigma} q_1 \cdots q_n$ at (a copy of) v.
- The disjoint union $D_\cup$ of all D_v is accepted by the union of the runs.
- On D_u, the run uses "the right rule" at u.
- By swapping, we turn $D_\cup$ into a suitable DAG D by redirecting each edge leaving u to the right v in D_v.

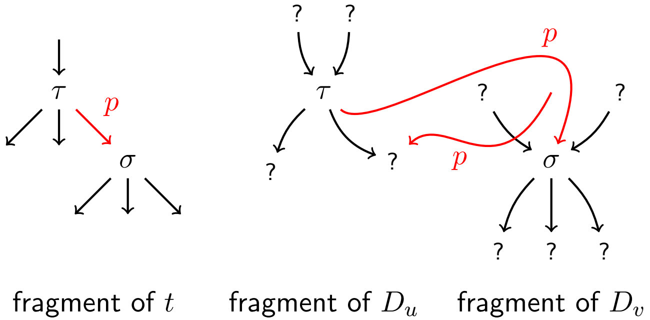
Proving Regularity of $tree(L(A))$

Example:



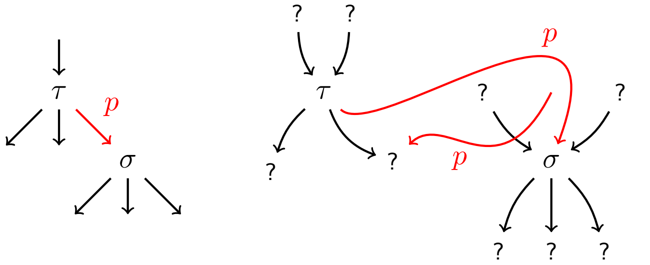
fragment of t fragment of D_u fragment of D_v

Example:


τ
p
σ
fragment of t
?
τ
p
σ
fragment of D_u
fragment of D_v

Proving Regularity of $tree(L(A))$

Example:



fragment of t fragment of D_u fragment of D_v

(Note that the other 5 edges leaving the nodes are treated similarly.)

Part 2

Deterministic DAG Automata

Determinism

Definition

For a rule $u \xrightarrow{\sigma} v$ let u be the head and v the tail.
A DAG automation is

- top-down deterministic if no two σ-rules for any σ have pairwise distinct heads, and
- bottom-up deterministic if no two σ-rules for any σ have pairwise distinct tails.

Observation

$L(A)^R = L(A^R)$, and A is top-down deterministic iff A^R is bottom-up deterministic, where -^{R} reverses edge directions in DAGs and interchanges heads and tails in automata.

Determinism Is a (Serious) Restriction

Observations

1. The well-known tree language

$$L = \{f(a,b), f(b,a)\}$$

(viewed as a DAG language) is not top-down deterministic, and so L^R is not bottom-up deterministic.

2. Consequently, $L \cup L^R$ is not deterministic at all.
3. Thus, there is no general determinization procedure.

Minimization

Distinguishable States for Top-Down Determinism

Definition

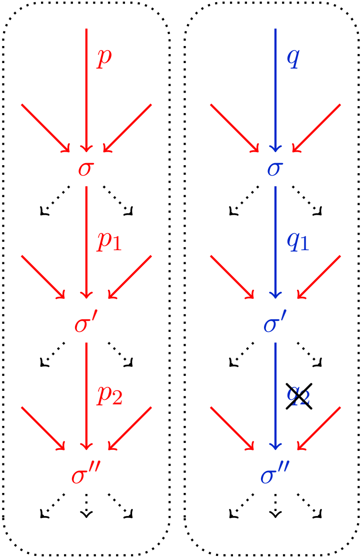
States p, p' are distinguishable if there are $\alpha, \beta \in Q^*$ and σ s.t.

- there is a σ-rule with head $\alpha p \beta$ but none with head $\alpha p' \beta$, or
- both σ-rules

  $$\begin{aligned} \alpha p \beta &\xrightarrow{\sigma} q_1 \cdots q_n \\ \alpha p' \beta &\xrightarrow{\sigma} q'_1 \cdots q'_n \end{aligned}$$

  exist and q_i and q'_i are distinguishable for some i.

Indistinguishable states are equivalent.



Minimization

Theorem: Minimal top-down deterministic DAG automata

Given a deterministic DAG automaton A, an equivalent minimal deterministic DAG automaton $A_{\min}$ can be constructed in polynomial time. Minimal deterministic DAG automata are unique up to state renaming.

Proof parts:

1. State equivalence is an equivalence relation.
2. Useless rules (not only in deterministic DAG automata) can be detected and removed in polynomial time.
3. Replace every state by its equivalence class.
4. This affects neither determinism nor the language.
5. Prove minimality and uniqueness (next slides).

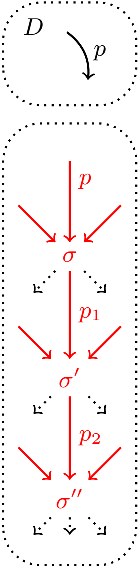
Proof of Minimality

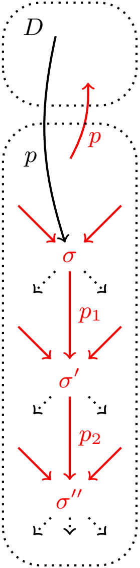
Suppose A' has fewer states than $A_{\min}$.

$\Rightarrow$ there are accepted DAGs D, D' with edges e, e' such that

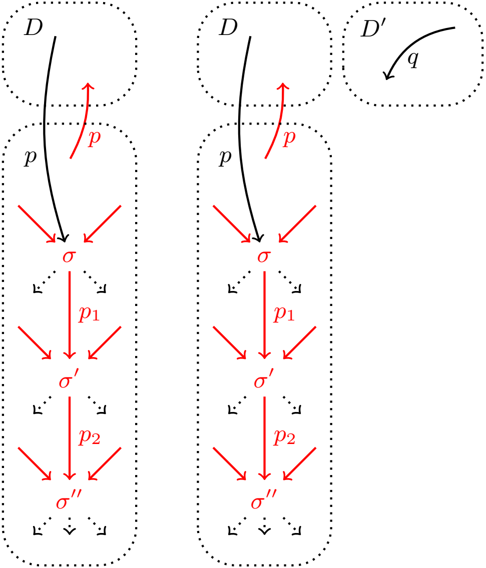
1. $A_{\min}$ assigns states p and q, $p \neq q$, to e and e',
2. A' assigns the same state to e and e'.

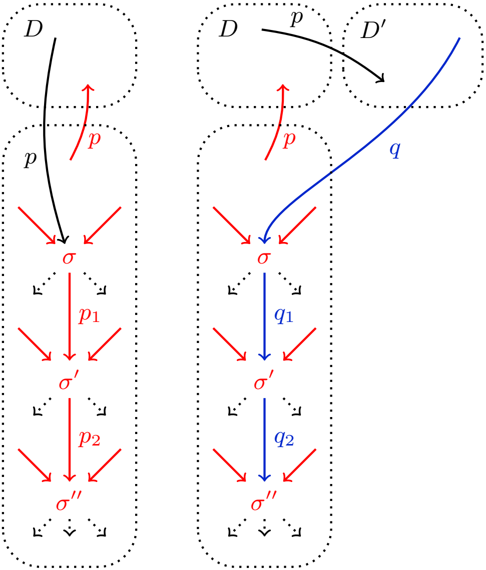
Since $p \neq q$, they are distinguishable in $A_{\min}$.

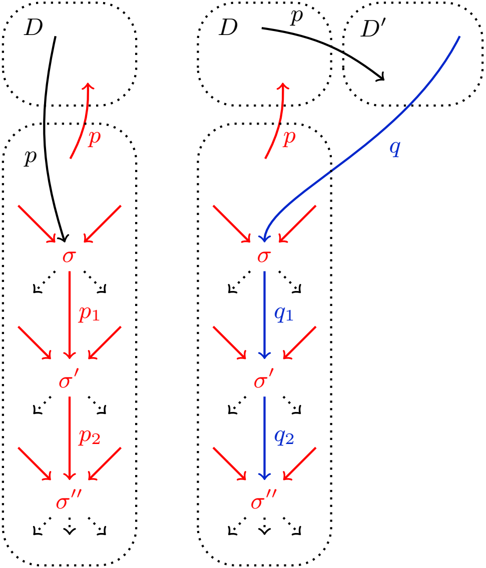



D
p
p
σ
p_1
σ'
p_2
σ''

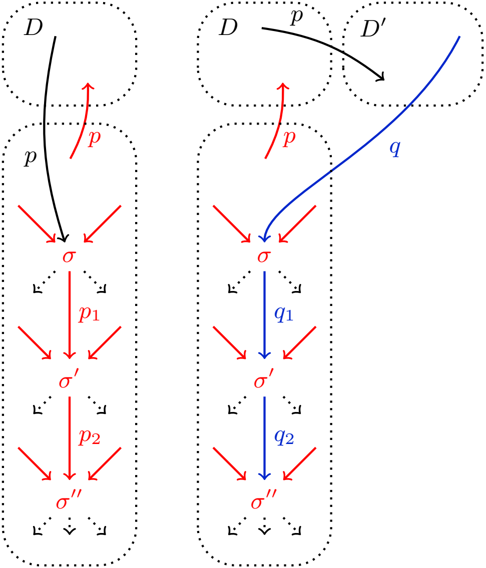
Minimality



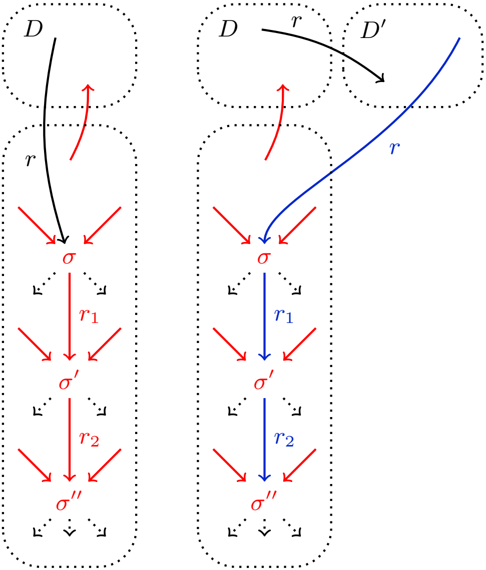

D
p
p
σ
p_1
σ'
p_2
σ''
D
p
D'
p
q
σ
q_1
σ'
q_2
σ''



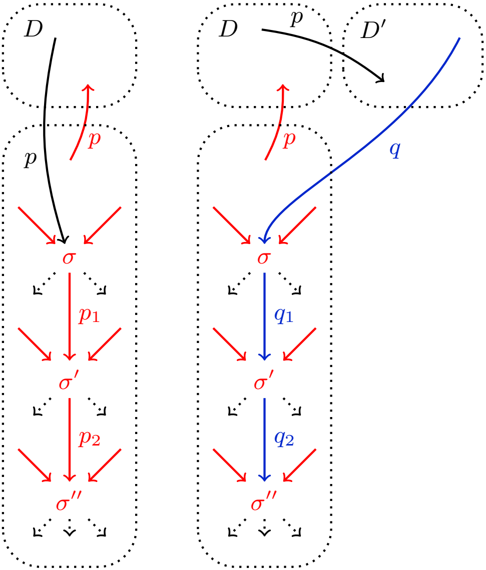
1. $A_{\min}$ accepts the left DAG (by swapping) but rejects the right one. (The bottom rule does not exist, by distinuishability.)



1. $A_{\min}$ accepts the left DAG (by swapping) but rejects the right one. (The bottom rule does not exist, by distinuishability.)
2. A' also accepts the left one (by equivalence).



1. A_{min} accepts the left DAG (by swapping) but rejects the right one. (The bottom rule does not exist, by distinuishability.)
2. A' also accepts the left one (by equivalence).
3. However, then A' accepts the right one as well (by swapping, since e, e' carry the same state r).



1. $A_{\min}$ accepts the left DAG (by swapping) but rejects the right one. (The bottom rule does not exist, by distinuishability.)
2. A' also accepts the left one (by equivalence).
3. However, then A' accepts the right one as well (by swapping, since e, e' carry the same state r).
4. Hence, $L(A_{\min}) \neq L(A')$.

Uniqueness

Proof of Uniqueness

Assume A' has the same number of states as $A_{\min}$, but there is no bijection between the state sets that turns $A_{\min}$ into A'.

$\Rightarrow$ again, there are $D, D' \in L(A_{\min})$ with edges e, e' such that

1. $A_{\min}$ assigns different states to e and e' in D and D', resp.,
2. A' assigns the same state to both.

Proof of Uniqueness

Assume A' has the same number of states as $A_{\min}$, but there is no bijection between the state sets that turns $A_{\min}$ into A'.

$\Rightarrow$ again, there are $D, D' \in L(A_{\min})$ with edges e, e' such that

1. $A_{\min}$ assigns different states to e and e' in D and D', resp.,
2. A' assigns the same state to both.

As we just saw, this implies $L(A') \neq L(A_{\min})$.

Equivalence Testing

The Equivalence Test

Equivalence of top-down deterministic A och B can be tested as usual:

1. Detect and remove useless rules.
2. Minimize both automata.
3. Check whether $A_{\min}$ and $B_{\min}$ are isomorphic.

Each of these steps takes at most polynomial time.

1. Reject right away if A' has more rules than A.
2. Initialize f as the empty partial mapping from Q to Q'.
3. Repeat as long as there are unprocessed rules left:
   1. Choose a rule $r = (\alpha \xrightarrow{\sigma} \beta)$ of A such that f is defined on all states in β.
   2. Check if B has a σ-rule $\alpha' \xrightarrow{\sigma} \beta'$ with $\alpha' = f(\alpha)$, and that f can be extended so that $f(\beta) = \beta'$.
   3. If so, extend f, remove r and repeat; otherwise reject.
4. When no rule is left, accept.

Part 3

Weighted DAG Automata

Unordered DAGs

1. Following Chiang et al. [3] we now consider unordered DAGs.
2. Unordered means that there is no order on the incoming and outgoing edges of nodes.
3. This reflects the NLP motivation slightly better, but makes little formal difference except when being interested in
   - determinism or
   - dropping the restriction to bounded degree (last part).

Weighted DAG Automata

Let $(\mathbb{S}, \oplus, \otimes, 0, 1)$ be a commutative semiring.

1. Heads and tails of a rule $I \xrightarrow{\sigma} O$ are now finite multisets of states.
2. A weight function δ assigns a non-zero weight to each rule in the set of rules.
3. As usual, the weight of a run is the $\otimes$-product of the weights of its rules and the weight of a DAG is the $\oplus$-sum of the weights of its runs.
4. The resulting mapping of DAGs to weights is a weighted DAG language.

More formally

$A = (\Sigma, Q, R, \delta)$ consists of

1. sets Σ and Q of node labels and states,
2. a finite set R of rules $I \xrightarrow{\sigma} O$ with $I, O \in \mathbb{N}^Q$ and $\sigma \in \Sigma$, and
3. a weight function $\delta\colon R \to \mathbb{S} \setminus \{0\}$.

A run ρ on DAG D maps every node v to a rule $\rho(v)$:

$$\begin{tikzcd} \cdots \arrow[d, bend right, "e_1"'] \arrow[d, bend left, "e_m"] \\ \sigma \arrow[d, bend right, "f_1"'] \arrow[d, bend left, "f_n"] \\ \cdots \end{tikzcd} \quad \mapsto \quad \{\rho(e_1), \ldots, \rho(e_m)\} \xrightarrow{\sigma} \{\rho(f_1), \ldots, \rho(f_n)\}$$

$$A(D) = \bigoplus_{\text{run } \rho} \bigotimes_{\text{node } v} \delta(\rho(v)) \text{ is the weight of } D.$$

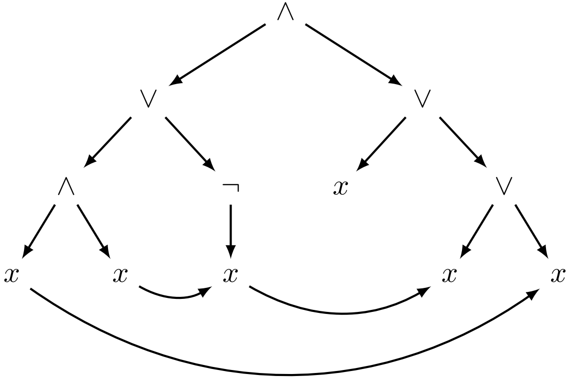
Weight Computation

Weight Computation is Difficult

Even in the Boolean case, the computation of weights (i.e., the membership problem) is difficult.

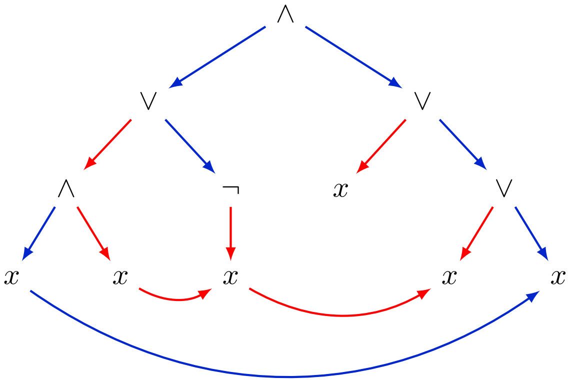
NP-Completeness

Even non-uniform membership (i.e., for a fixed unweighted DAG automaton) is easily shown to be NP-complete:



$$((x_1 \wedge x_2) \vee \neg x_2) \wedge (x_3 \vee (x_2 \vee x_1))$$

Even non-uniform membership (i.e., for a fixed unweighted DAG automaton) is easily shown to be NP-complete:



blue = true

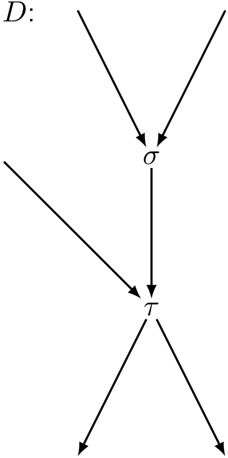
red = false

$$
\begin{aligned}
\{{\color{red}\bullet},{\color{red}\bullet}\} &\xrightarrow{x} \{{\color{red}\bullet}\},\\
\{{\color{blue}\bullet},{\color{blue}\bullet}\} &\xrightarrow{x} \{{\color{blue}\bullet}\},\\
&\vdots\\
\{{\color{blue}\bullet}\} &\xrightarrow{\wedge} \{{\color{blue}\bullet},{\color{blue}\bullet}\},\\
\{{\color{red}\bullet}\} &\xrightarrow{\wedge} \{{\color{red}\bullet},{\color{blue}\bullet}\},\\
&\vdots
\end{aligned}
$$

$$((x_1 \wedge x_2) \vee \neg x_2) \wedge (x_3 \vee (x_2 \vee x_1))$$

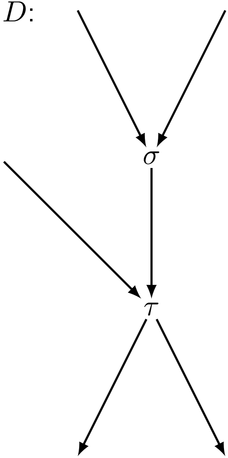
However, let's do it anyway...

Edge contraction algorithm for an input DAG D:



Edge contraction algorithm for an input DAG D:

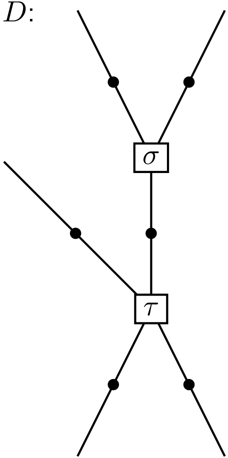
1. Turn D into its linegraph (nodes turn into hyperedges, edges into nodes).



A Weight Computation Algorithm

Edge contraction algorithm for an input DAG D:

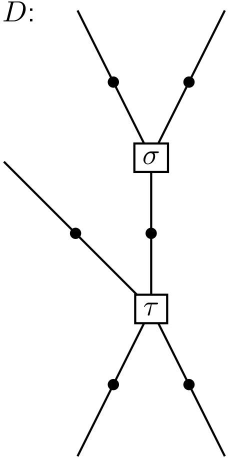
1. Turn D into its linegraph (nodes turn into hyperedges, edges into nodes).



A Weight Computation Algorithm

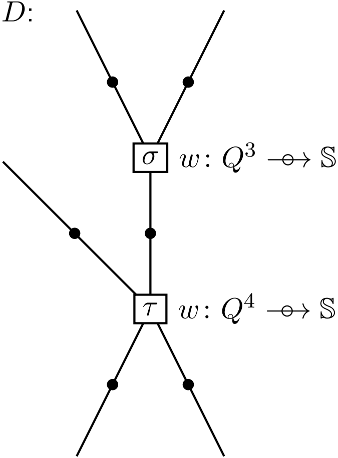
Edge contraction algorithm for an input DAG D:

1. Turn D into its linegraph (nodes turn into hyperedges, edges into nodes).
2. Annotate each hyperedge with all valid state assignments and their respective weights.



Edge contraction algorithm for an input DAG D:

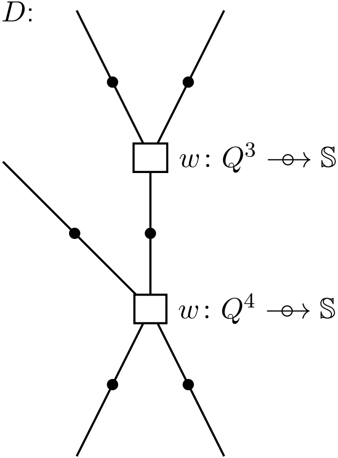
1. Turn D into its linegraph (nodes turn into hyperedges, edges into nodes).
2. Annotate each hyperedge with all valid state assignments and their respective weights.



A Weight Computation Algorithm

Edge contraction algorithm for an input DAG D:

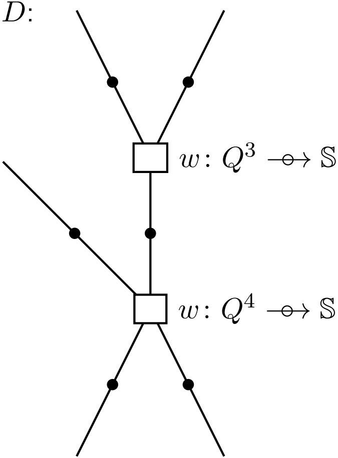
1. Turn D into its linegraph (nodes turn into hyperedges, edges into nodes).
2. Annotate each hyperedge with all valid state assignments and their respective weights.



A Weight Computation Algorithm

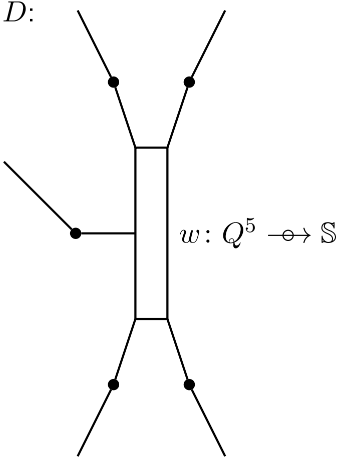
Edge contraction algorithm for an input DAG D:

1. Turn D into its linegraph (nodes turn into hyperedges, edges into nodes).
2. Annotate each hyperedge with all valid state assignments and their respective weights.
3. Repeatedly contract 2 neighboring hyperedes, multiplying weights of assignments which agree on the contracted "arms", and summing up.



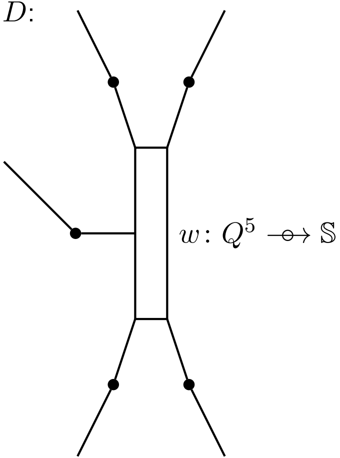
Edge contraction algorithm for an input DAG D:

1. Turn D into its linegraph (nodes turn into hyperedges, edges into nodes).
2. Annotate each hyperedge with all valid state assignments and their respective weights.
3. Repeatedly contract 2 neighboring hyperedes, multiplying weights of assignments which agree on the contracted "arms", and summing up.



Edge contraction algorithm for an input DAG D:

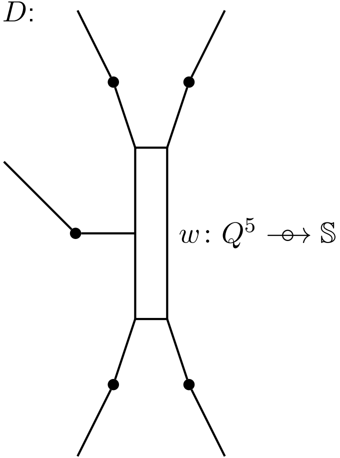
1. Turn D into its linegraph (nodes turn into hyperedges, edges into nodes).
2. Annotate each hyperedge with all valid state assignments and their respective weights.
3. Repeatedly contract 2 neighboring hyperedes, multiplying weights of assignments which agree on the contracted "arms", and summing up.
4. Stop when only one hyperedge is left, return $w()$ if defined, zero otherwise.



A Weight Computation Algorithm

Edge contraction algorithm for an input DAG D:

1. Turn D into its linegraph (nodes turn into hyperedges, edges into nodes).
2. Annotate each hyperedge with all valid state assignments and their respective weights.
3. Repeatedly contract 2 neighboring hyperedes, multiplying weights of assignments which agree on the contracted "arms", and summing up.
4. Stop when only one hyperedge is left, return $w()$ if defined, zero otherwise.



Optimal contraction order yields a running time exponential in the treewidth of the linegraph of D.

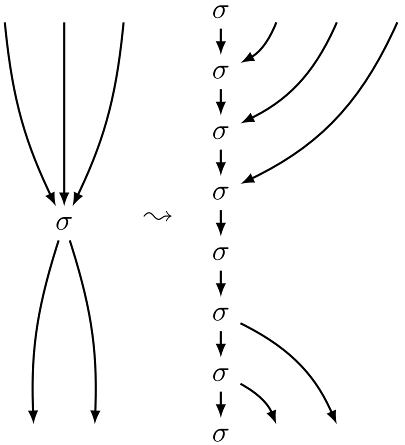
The treewidth of the line graph is at least the node degree of D.

Is there a way to make the node degree smaller?

Binarization

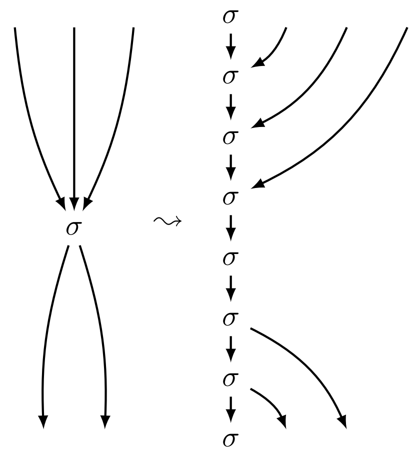
The Basic Idea of Binarization

- Similar to the first-child next-sibling encoding.



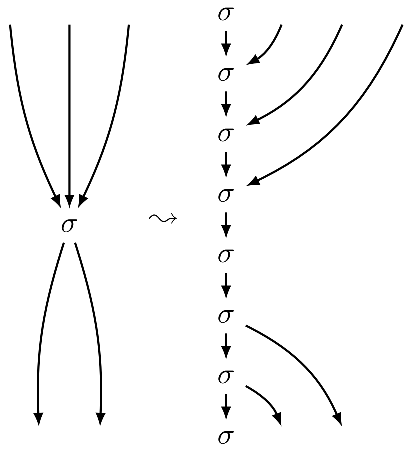
The Basic Idea of Binarization

- Similar to the first-child next-sibling encoding.
- In-/outdegree becomes as most 2, overall degree at most 3.



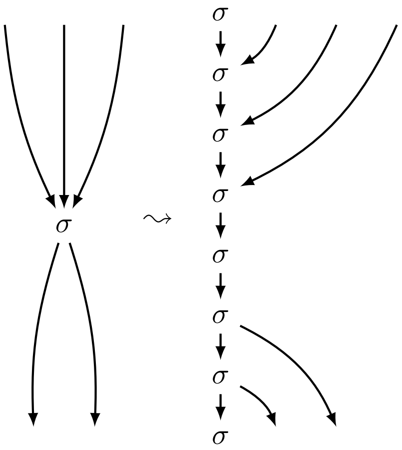
The Basic Idea of Binarization

- Similar to the first-child next-sibling encoding.
- In-/outdegree becomes as most 2, overall degree at most 3.
- Adapting the original DAG automaton is straightforward.



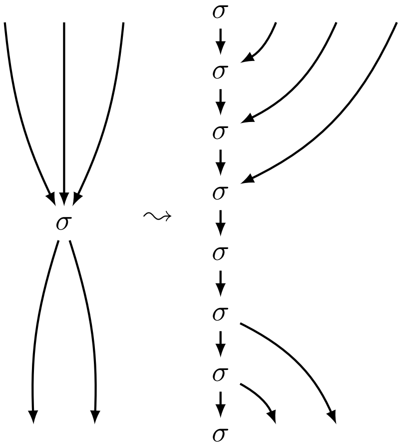
The Basic Idea of Binarization

- Similar to the first-child next-sibling encoding.
- In-/outdegree becomes as most 2, overall degree at most 3.
- Adapting the original DAG automaton is straightforward.
- It will then accept the image of the original DAG language after binarization.



The Basic Idea of Binarization

- Similar to the first-child next-sibling encoding.
- In-/outdegree becomes as most 2, overall degree at most 3.
- Adapting the original DAG automaton is straightforward.
- It will then accept the image of the original DAG language after binarization.



Now the node degree is 3!
(But there are exponentially many states.)

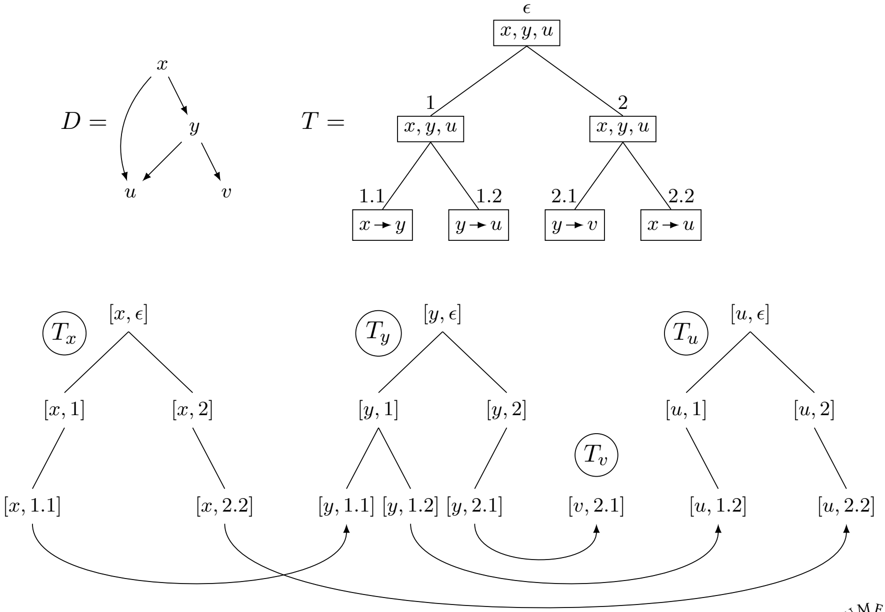
Binarization Along a Tree Decomposition

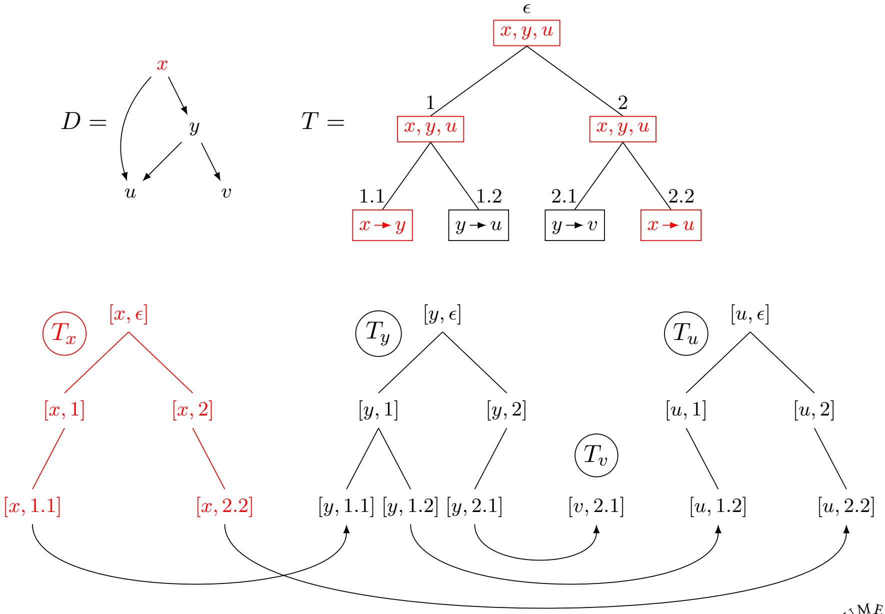
Can binarization speed up recognition?

Aim: Get rid of the potentially large treewidth of the linegraph.

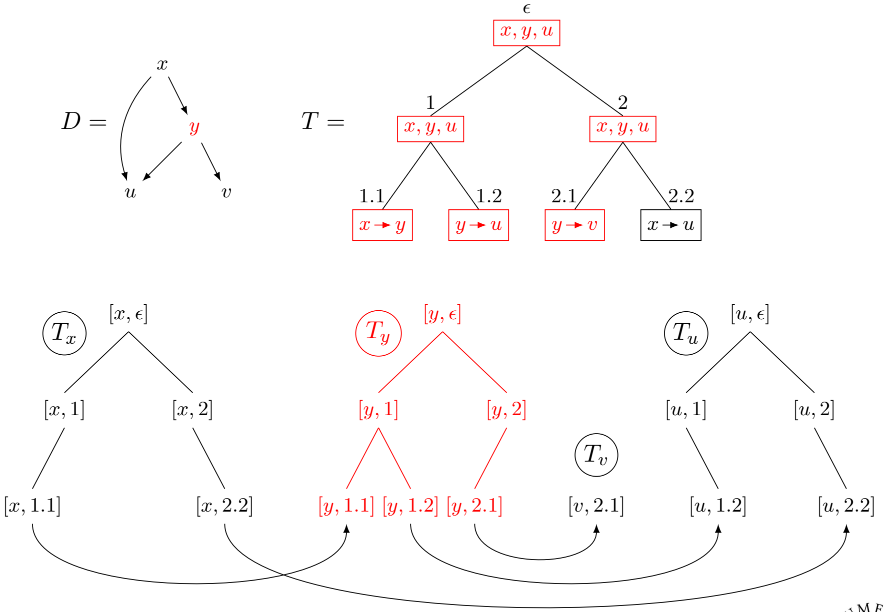
Intuition:

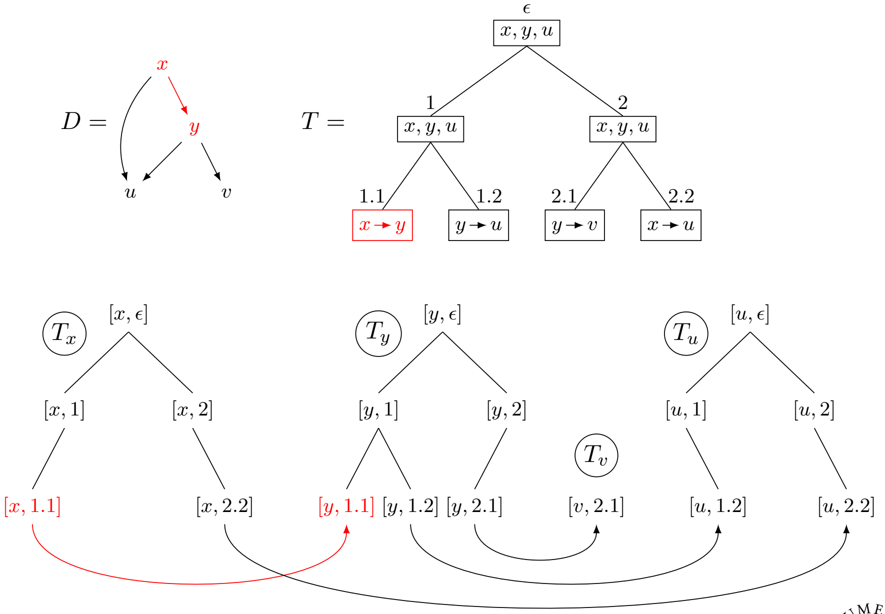
If we replace each node in D not by a "spine" but by a subtree of a (binary) tree decomposition of D, the tree decomposition of the linegraph is only twice that of D.


$D =$
x
y
u
v
$T =$
ϵ
x, y, u
1
2
x, y, u
x, y, u
1.1
1.2
2.1
2.2
$x \to y$
$y \to u$
$y \to v$
$x \to u$
T_x
$[x, \epsilon]$
$[x, 1]$
$[x, 2]$
$[x, 1.1]$
$[x, 2.2]$
T_y
$[y, \epsilon]$
$[y, 1]$
$[y, 2]$
$[y, 1.1]$
$[y, 1.2]$
$[y, 2.1]$
T_v
$[v, 2.1]$
T_u
$[u, \epsilon]$
$[u, 1]$
$[u, 2]$
$[u, 1.2]$
$[u, 2.2]$



D =
x
y
u
v
T =
ϵ
x, y, u
1
x, y, u
2
x, y, u
1.1
x → y
1.2
y → u
2.1
y → v
2.2
x → u
T_x
[x, ϵ]
[x, 1]
[x, 2]
[x, 1.1]
[x, 2.2]
T_y
[y, ϵ]
[y, 1]
[y, 2]
[y, 1.1]
[y, 1.2]
[y, 2.1]
T_v
[v, 2.1]
T_u
[u, ϵ]
[u, 1]
[u, 2]
[u, 1.2]
[u, 2.2]


D =
x
y
u
v
T =
ϵ
x, y, u
1
x, y, u
2
x, y, u
1.1
x → y
1.2
y → u
2.1
y → v
2.2
x → u
T_x
[x, ϵ]
[x, 1]
[x, 2]
[x, 1.1]
[x, 2.2]
T_y
[y, ϵ]
[y, 1]
[y, 2]
[y, 1.1]
[y, 1.2]
[y, 2.1]
T_v
[v, 2.1]
T_u
[u, ϵ]
[u, 1]
[u, 2]
[u, 1.2]
[u, 2.2]


$D =$
x
y
u
v
$T =$
ϵ
x, y, u
1
x, y, u
2
x, y, u
1.1
$x \to y$
1.2
$y \to u$
2.1
$y \to v$
2.2
$x \to u$
T_x
$[x, \epsilon]$
$[x, 1]$
$[x, 2]$
$[x, 1.1]$
$[x, 2.2]$
T_y
$[y, \epsilon]$
$[y, 1]$
$[y, 2]$
$[y, 1.1]$
$[y, 1.2]$
$[y, 2.1]$
T_v
$[v, 2.1]$
T_u
$[u, \epsilon]$
$[u, 1]$
$[u, 2]$
$[u, 1.2]$
$[u, 2.2]$

No Free Lunch

Advantages and disadvantages for recognition

– Binarization increases the size of the DAG automaton exponentially in the node degree.

+ The treewidth of the linegraph is only twice that of D.

What is better in practice remains to be seen.

Binarization will, however, turn out to be useful for handling unbounded degree.

Part 4

Removing the Bound on the Degree

Considerations

How can we handle unbounded degree?

1. An infinite number of rules $I \xrightarrow{\sigma} O$ must be described.
2. Obvious idea: use regular expressions α, β (over states) to specify those I and O which are valid.
3. Thus, the rules will be schemata of the form $\alpha \xrightarrow{\sigma} \beta$.
4. But α and β should
    1. specify languages of multisets of states and
    2. be weighted (to give each instance of a rule its individual weight).

We use a weighted version of Ochmański's c-regular expressions [6] or, equivalently, weighted multiset automata.

Weighted c-regular Expression

Defined like ordinary regular expressions, but:

1. Kleene star is restricted to expressions over unary alphabets.
2. Concatenation is interpreted as multiset union.
3. Expression kE multiplies weights by k.

Weighted Multiset Automaton

A weighted automaton such that the order of input symbols does not matter: For all states i, j and input symbols p, q:

$$\bigoplus_{\text{states } k} w(i,p,k) \otimes w(k,q,j) = \bigoplus_{\text{states } k} w(i,q,k) \otimes w(k,p,j).$$

Conversion between Expressions and Automata

Special case of general results by Droste & Gastin 1999 [5].

From Expressions to Automata

1. Can use ordinary McNaughton-Yamada for expressions E^*, because they are over unary alphabets.
2. Construction for EE' uses shuffle product of automata.

Note: size may become exponential because of the latter.

From Automata to Expressions

1. Consider the automaton as a string automaton and intersect with $q_1^* \cdots q_k^*$.
2. This yields an automaton which is mainly a sequence of k automata over unary alphabets $\{q_i\}$.
3. Construct $E_1 \cdots E_k$ by converting the automata individually.

Weighted Extended DAG Automaton

Weighted Extended DAG Automaton

In a weighted extended DAG automaton, each rule is of the form $\alpha \xrightarrow{\sigma} \beta$, where α, β are weighed c-regular expressions.

1. For a given run, the local weight of a σ-node with incoming and outgoing edges carrying state multisets I, O is

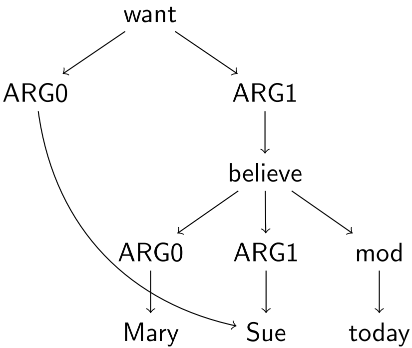
$$\bigoplus_{\text{rule } \alpha \xrightarrow{\sigma} \beta} [\![\alpha]\!](I) \otimes [\![\beta]\!](O).$$

2. As usual, multiply all local weights to obtain the weight of a run; sum up the weights of all runs to obtain the weight of the input DAG.

Example

$$
\begin{aligned}
\epsilon &\xrightarrow{\text{want}} q_{\text{arg0}} q_{\text{arg1}} q^*_{\text{mod}} \\
q_{\text{arg0}} &\xrightarrow{\text{ARG0}} q_{\text{person}} \\
q_{\text{arg1}} &\xrightarrow{\text{ARG1}} q_{\text{pred}} \\
q_{\text{arg1}} &\xrightarrow{\text{ARG1}} q_{\text{person}}
\end{aligned}
$$

$$
\begin{aligned}
q_{\text{pred}} &\xrightarrow{\text{want}} q_{\text{arg0}} q_{\text{arg1}} q^*_{\text{mod}} \\
q_{\text{pred}} &\xrightarrow{\text{believe}} q_{\text{arg0}} q_{\text{arg1}} q^*_{\text{mod}} \\
q_{\text{person}} q^*_{\text{person}} &\xrightarrow{\textit{propper name}} \epsilon \\
q_{\text{mod}} &\xrightarrow{\text{mod}} q_{\text{today}}
\end{aligned}
$$

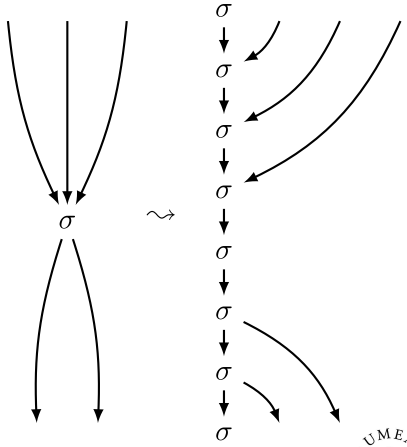


Properties of the Boolean (=Unweighted) Case

Recall Basic Binarization

Binarization makes it easy to carry over results:

- The subgraph can be processed by the multiset automata. $\Rightarrow$ blow-up exponential or linear, depending on input representation.
- Emptiness and finiteness are preserved.
- Path languages are related by an FST.



Theorem

For extended DAG automata over the Boolean semiring

1. emptiness and finiteness are decidable (in polynomial or exponential time, depending on the input representation), and
2. the path languages are regular.

Computing Weights

Weight computation by means of binarization:

1. Binarize the input DAG along a tree decomposition as before.
2. Similarly, transform A into a non-extended DAG automaton A'. (Turn the multiset automata of A' into DAG automata rules.)
3. Run the earlier algorithm on D using A'.

Running Time

The running time of this procedure is

$$O(|E_D|(|Q| + m^2|\Sigma|)^{2\mathrm{tw}(D)+3}).$$

A slightly "faster" algorithm avoiding binarization runs in time

$$O(|E_D|(|Q|m^{2(\mathrm{tw}(D)+2)} + m^{3(\mathrm{tw}(D)+1)}).$$

Some Questions to Work on

Questions

1. Decidability of decision problems such as equivalence in the basic (but nondeterministic) case. (Unbounded degree case should follow by binarization.)
2. Study more general notions of determinism/non-ambiguity.
3. All questions of this kind for the weighted case.
4. n-best algorithms for weighted regular DAG languages.
5. Find useful cases in which recognition/weight computation can be done efficiently.
6. Learning and training algorithms.
7. Practical evaluation (e.g., apply to AMR bank).

Thank you!

Some Papers I

Martin Berglund, Henrik Björklund, and Frank Drewes.
Single-rooted DAGs in regular dag languages: Parikh image and path languages.
In *13th Intl. Workshop on Tree-Adjoining Grammar and Related Formalisms (TAG+13)*, pages 94–101. Association for Computational Linguistics, 2017.

Johannes Blum and Frank Drewes.
Language theoretic properties of regular DAG languages.
Information and Computation, 2018.
To appear.

David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta.
Weighted DAG automata for semantic graphs.
Computational Linguistics, 44:119–186, 2018.

Frank Drewes.
On DAG languages and DAG transducers.
Bulletin of the European Association for Theoretical Computer Science, 121:142–163, 2017.

Some Papers II

Manfred Droste and Paul Gastin.

The Kleene-Schützenberger theorem for formal power series in partially commuting variables.

Information and Computation, 153:47–80, 1999.

Edward Ochmański.

Regular behaviour of concurrent systems.

Bulletin of the European Association for Theoretical Computer Science, 27:56–67, 1985.