Weighted DAG Automata for Semantic Graphs

David Chiang*
University of Notre Dame

Frank Drewes**
University of Umeå

Daniel Gildea†
University of Rochester

Adam Lopez‡
University of Edinburgh

Giorgio Satta§
University of Padua

Graphs have a variety of uses in natural language processing, particularly as representations of linguistic meaning. A deficit in this area of research is a formal framework for creating, combining, and using models involving graphs that parallels the frameworks of finite automata for strings and finite tree automata for trees. A possible starting point for such a framework is the formalism of directed acyclic graph (DAG) automata, defined by Kamimura and Slutzki and extended by Quernheim and Knight. In this article, we study the latter in depth, demonstrating several new results, including a practical recognition algorithm that can be used for inference and learning with models defined on DAG automata. We also propose an extension to graphs with unbounded node degree and show that our results carry over to the extended formalism.

* Department of Computer Science and Engineering, University of Notre Dame, IN 46656, United States. E-mail: dchiang@nd.edu. Some of the work described in this paper was done while Chiang was at the University of Southern California, Information Sciences Institute.
** Department of Computing Science, Umeå University, 90187 Umeå, Sweden. E-mail: drewes@cs.umu.se.
† Department of Computer Science, University of Rochester, Rochester, NY 14627, United States. E-mail: gildea@cs.rochester.edu.
‡ School of Informatics, University of Edinburgh, Edinburgh, EH8 9AB, United Kingdom. E-mail: alopez@inf.ed.ac.uk. Some of the work described in this paper was done while Lopez was at Johns Hopkins University.
§ Department of Information Engineering, University of Padua, I-35131 Padova, Italy. E-mail: satta@dei.unipd.it.

Submission received: 1 August 2016; revised version received: 9 May 2017; accepted for publication: 11 October 2017.

doi:10.1162/COLI_a_00309

1. Introduction

Statistical models of natural language semantics are making rapid progress. At the risk of oversimplifying, work in this area can be divided into two streams. One stream, semantic parsing (Mooney 2007), aims to map from sentences to logical forms that can be executed (for example, to query a knowledge base); work in this stream tends to be on small, narrow-domain data sets like GeoQuery. The other stream aims for broader coverage, and historically tackled shallower, piecemeal tasks, like semantic role labeling (Gildea and Jurafsky 2000), word sense disambiguation (Brown et al. 1991), coreference resolution (Soon, Ng, and Lim 2001), and so on. Correspondingly, resources like OntoNotes (Hovy et al. 2006) provided separate resources for each of these tasks.

This piecemeal situation parallels that of early work on syntactic parsing, which focused on subtasks like part-of-speech tagging (Ratnaparkhi 1996), noun-phrase chunking (Ramshaw and Marcus 1995), prepositional phrase attachment (Collins and Brooks 1995), and so on. As the field matured, these tasks were increasingly synthesized into a single process. This was made possible because of a single representation (phrase structure or dependency trees) that captures all of these phenomena; because of corpora annotated with these representations, like the Penn Treebank (Marcus, Marcinkiewicz, and Santorini 1993); and because of formalisms, like context-free grammars, which can model these representations practically (Charniak 1997; Collins 1997; Petrov et al. 2006).

In a similar way, more recent work in semantic processing consolidates various semantics-related tasks into one. For example, the Abstract Meaning Representation (AMR) Bank (Banarescu et al. 2013) began as an effort to unify the various annotation layers of OntoNotes. It has driven the development of many systems, chiefly string-to-AMR parsers like JAMR (Flanigan et al. 2014) and CAMR (Wang, Xue, and Pradhan 2015a,b), as well as many other systems submitted to the AMR Parsing task at SemEval 2016 (May 2016). AMRs have also been used for generation (Flanigan et al. 2016), summarization (Liu et al. 2015), and entity detection and linking (Li et al. 2015; Pan et al. 2015).

But the AMR Bank is by no means the only resource of its kind. Others include the Prague Dependency Treebank (Böhmová et al. 2003), DeepBank (Oepen and Lønning 2006), and Universal Conceptual Cognitive Annotation (Abend and Rappoport 2013). By and large, these resources are based on, or equivalent to, *graphs*, in which vertices stand for entities and edges stand for semantic relations among them. The Semantic Dependency Parsing task at SemEval 2014 and 2015 (Oepen et al. 2014, 2015) converted several such resources into a unified graph format and invited participants to map from sentences to these semantic graphs.

The unification of various kinds of semantic annotation into a single representation, semantic graphs, and the creation of large, broad-coverage collections of these representations are very positive developments for research in semantic processing. What is still missing—in our view—is a formal framework for creating, combining, and using models involving graphs that parallels those for strings and trees. Finite string automata and transducers served as a framework for investigation of speech recognition and computational phonology/morphology. Similarly, context-free grammars (and pushdown automata) served as a framework for investigation of computational syntax and syntactic parsing. But we lack a similar framework for learning and inferring semantic representations.

Two such formalisms have recently been proposed for NLP: one is hyperedge replacement graph grammars, or HRGs (Bauderon and Courcelle 1987; Habel and Kreowski 1987; Habel 1992; Drewes, Kreowski, and Habel 1997), applied to AMR

parsing by various authors (Chiang et al. 2013; Peng, Song, and Gildea 2015; Björklund, Drewes, and Ericson 2016). The other formalism is directed acyclic graph (DAG) automata, defined by Kamimura and Slutzki (1981) and extended by Quernheim and Knight (2012). In this article, we study DAG automata in depth, with the goal of enabling efficient algorithms for natural language processing applications.

After some background on the use of graph-based representations in natural language processing in Section 2, we define our variant of DAG automata in Section 3. We then show the following properties of our formalism:

- Path languages are regular, as is desirable for a formal model of AMRs (Section 4.1).
- The class of hyperedge-replacement languages is closed under intersection with languages recognized by DAG automata (Section 4.2).
- Emptiness is decidable in polynomial time (Section 4.3).

We then turn to the recognition problem for our formalism, and show the following:

- The recognition problem is NP-complete even for fixed automata (Section 5.1).
- For input graphs of bounded treewidth, there is an efficient algorithm for recognition or summing over computations of an automaton for an input graph (Section 5.2).
- The recognition/summation algorithm can be asymptotically improved using specialized binarization techniques (Section 6).

We expect that nodes of potentially unbounded degree will be important in natural language processing to handle phenomena such as coreference and optional modifiers. We show how to extend our formalism to handle nodes of unbounded degree (Section 7.2), and demonstrate the following additional results:

- All closure and decidability properties mentioned above continue to hold for the extended model, and the path languages stay regular (Section 7.3).
- We provide a practical recognition/summation algorithm for the novel model (Section 7.4).

2. Graphs for Natural Language

Graphs, or representations equivalent to graphs, have been used by many linguistic formalisms and natural-language processing systems to model semantic dependencies. For example, unification-based grammar formalisms use feature structures, like lexical functional grammar f-structures (Kaplan and Bresnan 1982) and head-driven phrase structure grammar synsem objects (Pollard and Sag 1994), that can be drawn as rooted, directed, (usually) acyclic graph structures (Shieber 1986). The Prague Dependency Treebank's tectogrammatical trees (Böhmová et al. 2003) can be turned into graphs using coreference and argument-sharing annotations, and DeepBank's annotations using Minimal Recursion Semantics can be stripped down to Elementary

Dependency Structures, which are graphs (Oepen and Lønning 2006). Universal Conceptual Cognitive Annotation (Abend and Rappoport 2013) uses several annotation layers, which are graphs. AMRs, whose format is derived from the PENMAN generation system, are equivalent to graphs (Banarescu et al. 2013). Several of these graph representations have been the target of the Semantic Dependency Parsing task at SemEval 2014 and 2015 (Oepen et al. 2014, 2015).

In this section, we focus on AMRs, but the formalisms we work with in the remainder of the article could, in principle, be used on any of the other graph representations listed above. Although the standard AMR format somewhat resembles the Penn Treebank's parenthesized representation of trees, AMRs can be thought of as directed graphs. Examples of these two representations, from the AMR Bank (LDC2014T12), are reported in Figures 1 and 2. Nodes are labeled, in order to convey lexical information. Edges are labeled to convey information about semantic roles. Labels at the edges need not be unique, meaning that edges impinging on the same node might have the same label. Furthermore, our DAGs are not ordered, meaning that there is no order relation for the edges impinging at a given node, as is usually the case in standard graph structures. A node can appear in more than one place (for example, in Figure 1, node s2

```
(a / and
      :op1 (a2 / ask-01
            :ARG0 (i / i)
            :ARG1 (t / thing
                  :ARG1-of (t2 / think-01
                        :ARG0 (s2 / she)
                        :ARG2 (l / location
                              :location-of (w / we))))
            :ARG2 s2)
      :op2 (s / say-01
            :ARG0 s2
            :ARG1 (a3 / and
                  :op1 (w2 / want-01 :polarity -
                        :ARG0 s2
                        :ARG1 (t3 / think-01
                              :ARG0 s2
                              :ARG1 l))
                  :op2 (r / recommend-01
                        :ARG0 s2
                        :ARG1 (c / content-01
                              :ARG1 i
                              :ARG2 (e / experience-01
                                    :ARG0 w))
                        :ARG2 i))
            :ARG2 i)
      :op3 c)
```

Figure 1
Example AMR in its standard format, number DF-200-192403-625_0111.7 from the AMR Bank. The sentence is: "I asked her what she thought about where we'd be and she said she doesn't want to think about that, and that I should be happy about the experiences we've had (which I am)."

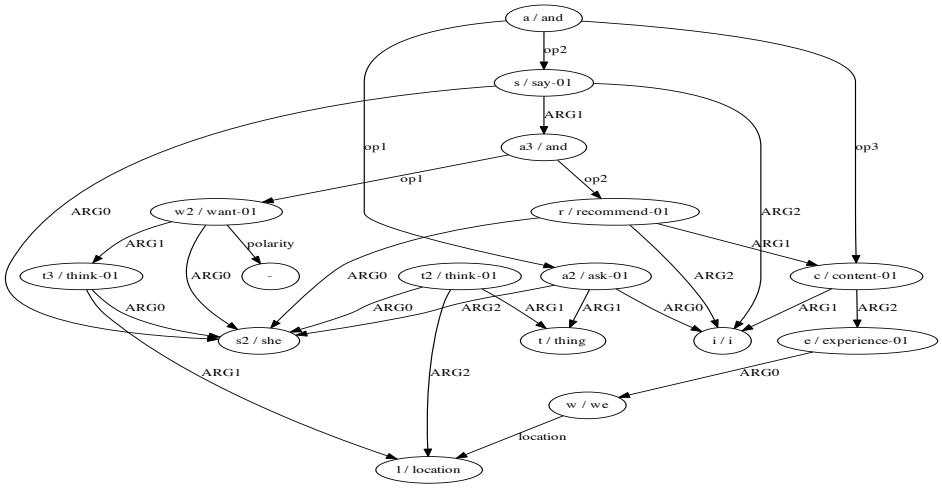


Figure 2
The AMR of Figure 1, presented as a directed graph.

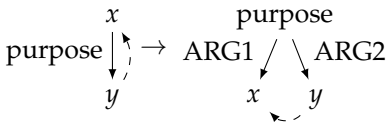


Figure 3
Reification of a role (edge label) can break cycles.

appears six times); we call this a **reentrancy**, analogous to a reentrant feature structure in unification-based grammar formalisms.

Cycles and multiple roots. Although the AMR guidelines[1] describe AMRs as acyclic graphs, the AMR Bank in fact contains some graphs with cycles. The majority of these cyclic graphs involve an edge labeled with an inverse role such as `ARG0-of`, which means that the parent node is the `ARG0` of the child node. The purpose of these inverse roles is to make the graph singly-rooted. If we reverse such edges, most cyclic graphs become acyclic (but multiply-rooted).

Most remaining cycles are caused by a relatively small number of roles. By "reifying" these, that is, changing them into nodes (see Figure 3), these cycles can be eliminated. Table 1 shows some statistics on the December 2014 internal release of the AMR Bank.[2]

The small percentage of graphs that are cyclic is reduced by reversing `*-of` edges, and all but eliminated by reification. The three cyclic graphs that remain (out of 20,628) were clearly annotation mistakes and were subsequently corrected.

1 `http://www.isi.edu/~ulf/amr/help/amr-guidelines.pdf`.
2 The first release is LDC catalog number LDC2014T12; we are grateful to ISI for providing us with an internal release that is somewhat larger than the first release.

Table 1
Statistics on AMR graphs, out of 20,628 total. original = as provided in the corpus; reversed = with all edge labels of the form `*-of` reversed; reified = with certain roles reified as needed to break cycles. A graph with no edges is counted as having zero treewidth.

	original	reversed	reified
cyclic	746	105	3
avg. roots	1.07	2.37	2.37
avg. treewidth	1.55	1.55	1.55
treewidth $= 0$	153	153	153
treewidth $= 1$	10,174	10,174	10,148
treewidth $= 2$	9,092	9,092	9,118
treewidth $= 3$	1,178	1,178	1,178
treewidth $= 4$	31	31	31

The table also shows that the average number of roots more than doubles as a result of these transformations. (The original corpus had a small number of instances that contained more than one sentence, and were annotated as multiple graphs under a `multi-sentence` node; we counted these as multiple roots.)

In summary, we can think of AMRs as singly-rooted, possibly cyclic directed graphs, or as multiply-rooted directed acyclic graphs.

Node degree. The in-degree (out-degree) of a node in a DAG is the number of incoming (outgoing, respectively) edges at that node. AMRs have unbounded in-degree and out-degree. Unbounded in-degree is needed for instance in the semantic representation of sentences with coreference relations, in which some concept is shared among several predicates. Unbounded out-degree allows the attachment to a given predicate a number of optional modifiers that can grow with the length of the sentence. We studied the degree distribution of nodes in the AMR Bank.[3] The maximum degree (in-degree plus out-degree) is 17, and the average is 2.12. The full degree distribution is shown in Figure 4. In practice, AMRs strongly favor nodes of low degree. Nonetheless, the presence of nodes with large degree indicates that practical applications are likely to benefit from algorithms capable of handling potentially unbounded degree, which we develop in Section 7.

Multiple edges. In the standard definition for graphs, also called simple graphs, there can be at most one edge between two nodes. As opposed to simple graphs, multigraphs allow more than one edge between two nodes, called multiple edges. In semantic representations this is very useful. For instance, in the AMR for the sentence "John likes himself," the node for the predicate "like" has its `ARG0` and `ARG1` semantic roles filled by the same argument "John." Accordingly, we use multigraphs to represent AMR. This also simplifies the definition of a recognition model for AMRs, since a check to avoid multiple-edges would in some sense add an external condition, making the theory more difficult to develop.

3 LDC catalog number LDC2014T12.

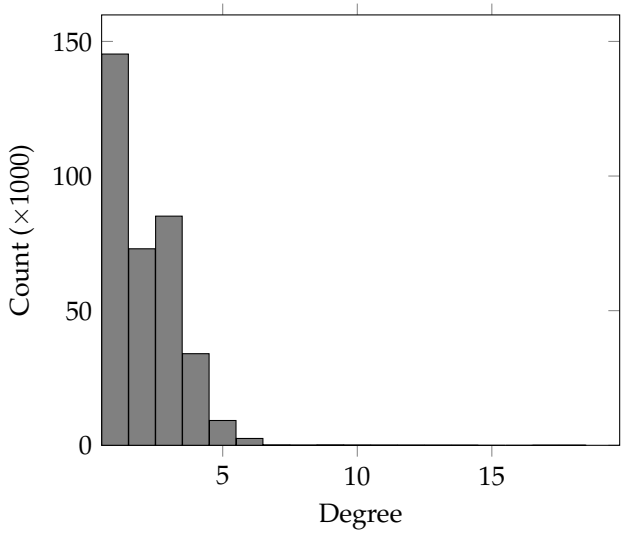


Figure 4
Degree distribution of nodes in the AMR Bank (reversed and reified).

Treewidth. Several of the algorithms presented in this article depend on the graph-theoretical notion of treewidth. The treewidth of a graph G, written $\mathrm{tw}(G)$, is a natural number that formalizes the degree to which G is "tree-like," with trees having treewidth of 1. We will postpone the mathematical definition of $\mathrm{tw}(G)$ to the next section.

For a graph G and a value k given as input, it is NP-complete to determine whether G has treewidth at most k. However, for the semantic graphs we are dealing with, the worst case might not be realized. Using a reimplementation of the QuickBB algorithm (Gogate and Dechter 2004), with only the "simplicial" and "almost-simplicial" heuristics, we found that we could compute the exact treewidth of all the graphs in the AMR Bank in a few seconds. The results (deleting `multi-sentence` nodes) are shown in Table 1: The average treewidth is only about 1.5, and the maximum treewidth is only 4. An example of a graph with treewidth 4 is shown in Figure 2. As we will see, this means that algorithms with an exponential dependence on treewidth can be practical for real world AMRs.

3. DAG Automata

In this section we formally specify the type of DAGs that we use in this article. We then define a family of automata that process languages of these DAGs, under the restriction that nodes have bounded degree. We also briefly discuss the existing literature on DAG automata. The restriction on node degree will later be dropped, in Section 7.

3.1 Preliminaries

We make frequent use of finite multisets. Formally, given a set Q, a **multiset** over Q is a mapping $\mu\colon Q \to \mathbb{N}$. Intuitively, $\mu(q) = n$ means that q occurs n times in μ. The collection of all finite multisets over Q is denoted by $\mathcal{M}(Q)$. We usually specify a multiset $\mu \in \mathcal{M}(Q)$ by listing its elements using a set-like notation such as $\{q_1, \ldots, q_n\}$. Note,

however, that $q_1, \ldots, q_n$ may contain repeated elements, in contrast to ordinary sets. We also use the latter, but the context will always disambiguate the two different meanings. The union of multisets is denoted by the operator $\uplus$ and is defined by pointwise addition: $(\mu \uplus \mu')(q) = \mu(q) + \mu'(q)$ for all $q \in Q$. Thus, if $\mu = \{q_1, \ldots, q_m\}$ and $\mu' = \{q'_1, \ldots, q'_n\}$ then $\mu \uplus \mu' = \{q_1, \ldots, q_m, q'_1, \ldots, q'_n\}$. If $f\colon Q \to P$ is a function, we extend it to a function from $\mathcal{M}(Q)$ to $\mathcal{M}(P)$ in the canonical way: $f(\{q_1, \ldots, q_n\}) = \{f(q_1), \ldots, f(q_n)\}$.

An **alphabet** is a finite set Σ that we are going to use as node labels for our graphs. We consider graphs that are directed and unordered, have nodes labeled by symbols from Σ, and have multiple edges. We do not use edge labels, despite the fact that the AMR structures we want to model have labels at their edges. Our choice is motivated by our goal to simplify the notation. Graphs with labels only at their nodes can easily encode graphs with edge labels by splitting every edge into two, and putting an extra node in the middle, whose label is the label of the edge. We will come back to the discussion of this encoding at the end of this section, after our definition of DAG automata.

Definition 1
A (node-labeled, directed and unordered) **graph** is a tuple $D = (V, E, lab, src, tar)$, where V and E are finite sets of **nodes** and **edges**, respectively, $lab\colon V \to \Sigma$ is a labeling function, and $src, tar\colon E \to V$ are functions that assign to each edge $e \in E$ its **source** node $src(e)$ and its **target** node $tar(e)$, respectively.

Note that our definition does not identify an edge with the pair of nodes that the edge is incident upon. In the terminology of standard graph theory, this means that our graphs are not simple graphs. This allows us to use multiple edges incident upon the same pair of nodes, a feature that is not only natural for AMRs (see the previous section) but will also be used in several of our algorithms.

A graph D as above is a **directed acyclic graph** if it is acyclic. More precisely, there do not exist $e_0, \ldots, e_{k-1} \in E$ with $k > 0$ such that $tar(e_{i-1}) = src(e_{i \bmod k})$ for $1 \leq i \leq k$. In this article, we will only consider directed acyclic graphs that are nonempty and connected. We call them DAGs, for short, and denote the set of all DAGs over Σ by $\mathcal{D}_\Sigma$. Note that a DAG can have multiple roots, that is, there may be more than one node $v \in V$ such that $tar(e) \neq v$ for all $e \in E$. (By acyclicity, there is always at least one root.) For a node $v \in V$ we define the sets of incoming and outgoing edges of v in the obvious way: $in(v) = \{e \in E \mid tar(e) = v\}$ and $out(v) = \{e \in E \mid src(e) = v\}$.

As usual, the graph D is a **tree** if there is a node $r \in V$, the root of D, such that every node $v \in V \setminus \{r\}$ is reachable from r on exactly one directed path, i.e., there is exactly one sequence of edges $e_1, \ldots, e_k$ with $k > 0$ such that $r = src(e_1)$, $tar(e_i) = src(e_{i+1})$ for all $1 \leq i < k$, and $tar(e_k) = v$. We use standard terminology regarding trees. In particular, a node v is a **child** of a node u if $out(u) \cap in(v) \neq \emptyset$.

As mentioned in the previous section, the treewidth of DAGs plays an important role for the algorithms proposed in this article. We now recall the notions of tree decompositions and treewidth, at the same time introducing the specific notation that will be used later in the article.

Definition 2
A **tree decomposition** of a graph $D = (V, E, lab, src, tar)$ is a tree T whose nodes and edges we call **bags** and **arcs**, respectively, and whose node labels are subsets of V. For

the sake of clarity, the label of bag b is denoted by $cont(b)$ rather than by $lab(b)$ and is called the **content** of b. T is required to satisfy the following:

1. For every node $v \in V$, there is a bag b such that $v \in cont(b)$.
2. For every edge $e \in E$, there is a bag b such that $\{src(e), tar(e)\} \subseteq cont(b)$.
3. For every node $v \in V$, the subgraph of T induced by the bags b containing v is connected.

The **width** of T is the maximum of quantity $|cont(b)| - 1$ computed over all bags b of T, and the **treewidth** of D is the minimum of the widths of its tree decompositions.

We note here that, in most definitions in the literature, the edges of a tree decomposition are undirected. In the context of this article, however, it is more convenient to define tree decompositions to be directed trees, because later on we will define algorithms that process our DAGs in an order that is guided by the arc directions in the associated tree decompositions. In order to turn an undirected tree decomposition into a directed one, just choose an arbitrary bag as the root, and establish edge directions accordingly.

Example 1
Consider the DAG D shown in Figure 5(a). A possible tree decomposition T of D is displayed in Figure 5(b), consisting of five bags, each containing a maximum of four nodes from D. It is easy to check that T satisfies the first two conditions in the definition of tree decomposition. Consider now node 5 of D. The bags of T containing this node are the three topmost ones. The subgraph of T induced by these bags is connected (and thus a tree in itself). The same holds true for any other node of D, and this shows that the third condition in the definition of tree decomposition is satisfied as well.

Several other tree decompositions can be constructed for D. For instance, a trivial tree decomposition of D is the tree containing a single bag with all the nodes of D. However, it is not difficult to argue that every tree decomposition of D must have a bag that contains at least 4 nodes from D. Thus, the treewidth of D is 3. Informally, the size of the largest bags in a tree decomposition increases with the number of reentrancies that can be found along a path in the DAG.

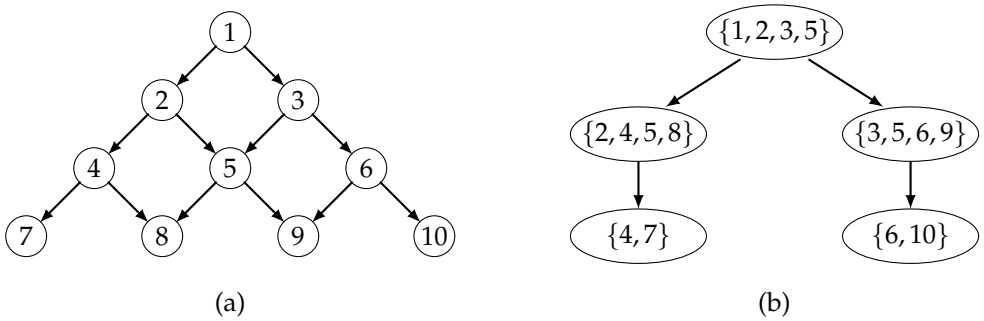


Figure 5
DAG D (a) and a tree decomposition T (b).

3.2 Definition

Let us now embark on the definition of DAG automata. Informally, a DAG automaton consists of a set of nondeterministic transitions that read DAG nodes and associate states with their incoming and outgoing edges. Because we do not only want to recognize DAG languages but, more generally, want to be able to use DAG automata to associate a weight with each DAG, we define a more general version in which the transitions have weights taken from some semiring $\mathbb{K}$. Throughout the entire paper, all semirings are assumed to be commutative—that is, not only the additive but also the multiplicative operator is commutative.

Definition 3
A **weighted DAG automaton** is a tuple $M = (\Sigma, Q, \delta, \mathbb{K})$, where

- Σ is an alphabet of node labels
- Q is a finite set of states
- $(\mathbb{K}, \oplus, \otimes, 0, 1)$ is a semiring of weights (which we identify with its domain $\mathbb{K}$ if there is no danger of confusion)
- $\delta\colon \Theta \to \mathbb{K} \setminus \{0\}$ is a transition function that assigns nonzero weights to a finite set Θ of transitions of the form $t = \langle \{q_1, \ldots, q_m\}, \sigma, \{r_1, \ldots, r_n\} \rangle \in \mathcal{M}(Q) \times \Sigma \times \mathcal{M}(Q)$, where $m, n \geq 0$. If $\delta(t) = w$ we also write this transition in the form

$$\{q_1, \ldots, q_m\} \xrightarrow{\sigma/w} \{r_1, \ldots, r_n\} \tag{1}$$

As already mentioned, a DAG automaton processes an input DAG by assigning states to edges. A transition of the form of Equation (1) gets m states on the incoming edges of a node and puts n states on the outgoing edges. Alternatively, we may read the transition bottom–up, that is, it gets n states on the outgoing edges and puts m states on the incoming edges. As two special cases, note that when $m = 0$ in Equation (1) then the transition processes a root node, and when $n = 0$ it processes a leaf node.

Note that the transition function $\delta\colon \Theta \to \mathbb{K} \setminus \{0\}$ assigns nonzero weights to the transitions of a DAG automaton. Intuitively, the weight of all transitions not in Θ is 0. Reflecting this intuition, we extend δ to the set of all possible transitions $t \in \mathcal{M}(Q) \times \Sigma \times \mathcal{M}(Q)$ by defining $\delta(t) = 0$ for every $t \notin \Theta$. In this way, δ is turned into a total function, which is sometimes convenient.

The use of multisets of states in Equation (1) is needed because, when processing a node v, the same state might be assigned to several of the edges in $in(v)$ or in $out(v)$, and we have to specify the collection of all these state occurrences. As an example, assume $|in(v)| = 3$. Then we should distinguish between the scenario where the assigned states are $\{q, q, q'\}$ and the scenario where the assigned states are $\{q, q', q'\}$.

Let us now formally define the semantics $[\![M]\!]$ of a DAG automaton M as in Definition 3. As may be expected, $[\![M]\!]$ maps every DAG over Σ to its weight. A **run** of M on a DAG D with node set V and edge set E is a mapping $\rho\colon E \to Q$. The transition function δ extends to runs by taking the product of all local transition weights:

$$\delta(\rho) = \bigotimes_{v \in V} \delta(\langle \rho(in(v)), lab(v), \rho(out(v)) \rangle)$$

Now, $[\![M]\!](D)$ is the sum of the weights of all runs on D:

$$[\![M]\!](D) = \bigoplus_{\text{run } \rho \text{ on } D} \delta(\rho)$$

An **unweighted DAG automaton** is the special case of a DAG automaton in which $\mathbb{K}$ is the Boolean semiring. In this case, $[\![M]\!] : \mathcal{D}_\Sigma \to \{true, false\}$ is the characteristic function of a subset of $\mathcal{D}_\Sigma$. We generally identify such a characteristic function with the corresponding set, that is, $[\![M]\!] = \{D \in \mathcal{D}_\Sigma \mid [\![M]\!](D) = true\}$, and call it the **DAG language recognized** by M. DAG languages that can be recognized by unweighted DAG automata are **recognizable DAG languages**. Note that because *false* is the zero element of the Boolean semiring, all transitions appearing in an unweighted DAG automaton are of the form $\{q_1, \ldots, q_m\} \xrightarrow{\sigma/true} \{r_1, \ldots, r_n\}$. So we can simplify the notation of such a transition by writing $\{q_1, \ldots, q_m\} \xrightarrow{\sigma} \{r_1, \ldots, r_n\}$. An **accepting run** of M is a run whose weight is *true*, i.e., which uses only transitions of M.

Example 2
Let us illustrate unweighted DAG automata with a small example, where the label alphabet Σ is given by $\Sigma = \{a, b\}$. In our example, a's have two children and can be roots whereas b's have two parents and can be leaves. We want the automaton to accept all DAGs such that no path contains more than two consecutive a's. To accomplish this, viewing a run as a top–down process, we need to use the states in order to keep track of whether we have recently seen zero, one, or two a's. Consequently, we let $M = (\Sigma, Q, \delta, \mathbb{K})$, where $Q = \{0, 1, 2\}$ and δ is given by the transitions

$$\emptyset \xrightarrow{a} \{1, 1\}$$

$$\{i\} \xrightarrow{a} \{i+1, i+1\} \text{ for } 0 \leq i \leq 1$$

$$\{i, j\} \xrightarrow{b} \{0\} \mid \emptyset \qquad \text{for } 0 \leq i, j \leq 2$$

The notation in the last line, which will also be used later on, abbreviates two transitions, namely $\{i, j\} \xrightarrow{b} \{0\}$ and $\{i, j\} \xrightarrow{b} \emptyset$. An accepting run on a DAG in $[\![M]\!]$ is shown in Figure 6.

It may be instructive to note that the construction of a run of the automaton can be understood as a top–down or a bottom–up process. Under the top–down view, this particular automaton is deterministic: for each node the states on the incoming edges uniquely determine those on the outgoing edges. In contrast, under a bottom–up view, thus essentially reading transitions backwards, the transitions for b create a nondeterministic behavior.

Example 3
A finite automaton for strings, as traditionally defined (Hopcroft and Ullman 1979), is a special case of our DAG automata, where each transition has at most one incoming state and at most one outgoing state. Each DAG in the language recognized by such an

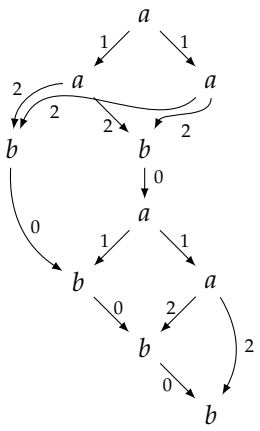
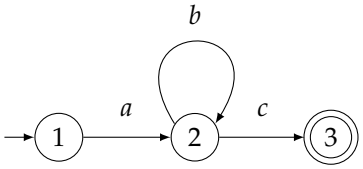


Figure 6
Example run of a DAG automaton.

automaton consists of one long path, and the vertex labels can be interpreted as tokens in a string. For example, the finite automaton



can be represented by a DAG automaton with the transitions:

$$\emptyset \xrightarrow{a} \{q\}$$
$$\{q\} \xrightarrow{b} \{q\}$$
$$\{q\} \xrightarrow{c} \emptyset$$

Note that empty state sets take the place of initial and final states in traditional finite automata.

Similarly, our DAG automata generalize tree automata (Comon et al. 2002), because a DAG automaton with transitions having at most one incoming state and any number of outgoing states will recognize a tree.

Example 4
We now present a linguistic example based on the sentence "John wants Mary to believe him" and its AMR representation D. In Figure 7 we display a fragment of the transitions of a DAG automaton M, along with an accepting run of M on D.

As already mentioned, although the standard AMR representation has labels on both edges and nodes, for simplicity we only consider DAGs with labels on nodes. We represent the edge labels of AMR, such as ARG0 and ARG1, as nodes with one incoming and one outgoing edge.

We observe that our DAG automata could, without any change in the definitions, also be applied to directed acyclic graphs that may be disconnected, or even to graphs over Σ containing cycles if this turns out to be of interest for some application. Of course,

Transitions:

$$\emptyset \xrightarrow{\text{want}} \{q_{\text{want-arg0}}, q_{\text{want-arg1}}\}$$

$$\{q_{\text{want-arg0}}\} \xrightarrow{\text{ARG0}} \{q_{\text{person}}\}$$

$$\{q_{\text{want-arg1}}\} \xrightarrow{\text{ARG1}} \{q_{\text{pred}}\}$$

$$\{q_{\text{pred}}\} \xrightarrow{\text{believe}} \{q_{\text{believe-arg0}}, q_{\text{believe-arg1}}\}$$

$$\{q_{\text{believe-arg0}}\} \xrightarrow{\text{ARG0}} \{q_{\text{person}}\}$$

$$\{q_{\text{believe-arg1}}\} \xrightarrow{\text{ARG1}} \{q_{\text{person}}\}$$

$$\{q_{\text{person}}, q_{\text{person}}\} \xrightarrow{\text{John}} \emptyset$$

$$\{q_{\text{person}}\} \xrightarrow{\text{Mary}} \emptyset$$

Example run:

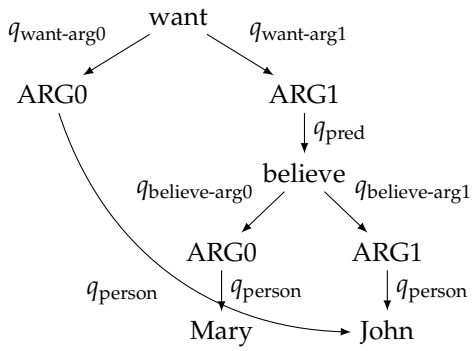


Figure 7
A DAG automaton (top) and an example run (bottom) on the AMR for the sentence "John wants Mary to believe him."

the algorithmic results presented in the following could not necessarily be assumed to hold in such a generalized case anymore.

To conclude the present section, we discuss the theoretical implications of our choice to exclude edge labels for our DAGs. Assume for the moment that we did include edge labels, taken from an alphabet Λ. A generalized transition applying to a node labeled with σ that has incoming edges labeled by $\lambda_1, \ldots, \lambda_m \in \Lambda$ and outgoing edges labeled by $\lambda'_1, \ldots, \lambda'_n \in \Lambda$ would then look like this:

$$\{q_1{:}\lambda_1, \ldots, q_m{:}\lambda_m\} \xrightarrow{\sigma/w} \{r_1{:}\lambda'_1, \ldots, r_n{:}\lambda'_n\}$$

with the obvious semantics. We now show that there is no power added in using edge labels. To this end we generalize the approach of Example 4, where ARG0 and ARG1 are used as node labels rather than edge labels. Each edge e with label λ can be encoded by splitting e into two new unlabeled edges e_1 and e_2, and by adding a fresh node v with label λ in between e_1 and e_2. Using this special encoding, transitions of the form

above can be implemented by an automaton as in Definition 3. For this, we enlarge our node labeling alphabet by adding the labels in Λ to it. Further, the state set Q is replaced by $(\Lambda \times Q) \cup (Q \times \Lambda)$. The automaton contains all transitions $\{(q, \lambda)\} \xrightarrow{\lambda/1} \{(\lambda, q)\}$, for $q \in Q$ and $\lambda \in \Lambda$. Thus, for each of the fresh nodes, the label is carried to the states on its incident edges, and the same state q is assigned to both edges, effectively simulating a single edge labeled with λ and carrying the state q. Now, every generalized transition as above can be turned into the ordinary transition

$$\{(q_1, \lambda_1), \ldots, (q_m, \lambda_m)\} \xrightarrow{\sigma/w} \{(\lambda'_1, r_1), \ldots, (\lambda'_n, r_n)\}$$

It should be clear that this DAG automaton simulates the processing of the edge labels of the generalized DAG automaton in a faithful way.

3.3 Related Formalisms

Other than in the perspective of natural language processing, DAG automata have been investigated in several different domains—for instance, to represent derivations in Chomsky type-0 phrase structure grammars (Kamimura and Slutzki 1981), to solve systems of set constraints (Charatonik 1999), or to process series-parallel graphs in pattern matching applications (Fujiyoshi 2010).

Kamimura and Slutzki (1981) define automata for two classes of DAGs. They primarily consider so-called d-DAGs, a recursively defined type of ordered planar DAG, where ordered means that there is a global total order on the set of nodes of the graph that implicitly orders the incoming and outgoing edges of each node. These d-DAGs are intended to model the derivations of type-0 grammars (equivalent to Turing machines). Accordingly, d-DAGs have bounded node degree and cannot have subgraphs matching certain Z-like patterns that would correspond to the same node being rewritten by two different rules. These restrictions are unsuitable when modeling natural language semantic structures. The authors also briefly consider DAGs without the planarity restriction, but still ordered in the sense mentioned above.

Our definition of DAG automata is based on that of Quernheim and Knight (2012). Also motivated by modeling semantic representations of natural languages, Quernheim and Knight extend the automata of Kamimura and Slutzki (1981) by adding weights and by dropping the planarity restriction as well as the bound on the in-degree. In order to process nodes with unbounded in-degree, Quernheim and Knight exploit some ordering on the incoming edges at each node, and introduce so-called implicit rules that process these edges in several steps. In Section 7 we take a different, simpler approach for processing DAGs with unbounded node degree that can also handle unbounded out-degree. Overall, this article can be viewed as an in-depth exploration of the theoretical properties of a somewhat simplified version of the formalism of Quernheim and Knight.

There are also major notational differences with respect to our proposal: Quernheim and Knight (2012) essentially view computations as top–down rewriting processes, and the rewriting relation is defined via the introduction of specialized DAGs, called incomplete DAGs. In contrast, in our definition of run in Sections 3.1 and 7.2, there is no commitment to a specific rewriting process, which makes the notation somewhat simpler. Quernheim and Knight also show how to obtain weighted DAG-to-tree transducers, which could form the basis of a natural language generation system.

With the goal of modeling ground terms in logical languages, Charatonik (1999) proposes devices that are mainly bottom–up tree automata running on DAGs, and states the external restriction, not implemented through the defined automata, that for these DAGs common substructures should be maximally shared. This maximal sharing condition is quite common in the literature on unification, but is unsuitable when modeling natural language semantic structures: Two copies of the same semantic substructure should be shared only when they refer to the same concept or action. A consequence of the maximal sharing is that, even in a nondeterministic automaton, isomorphic sub-DAGs are assigned the same state (because they are actually identical). This is exploited in the main result of Charatonik, the NP-completeness of the emptiness problem. This is in contrast with the polynomial time result for the same problem for our DAG automata, presented in Section 5.1.

Anantharaman, Narendran, and Rusinowitch (2005) also work under the maximal sharing assumption, and solve in the negative the problem of closure under complementation that had been left as an open question by Charatonik (1999). The authors consider the uniform membership problem for their automata, showing NP-hardness. Here, uniform means that the automaton is considered as part of the input. In our article and relative to our family of automata, we consider the easier problem of deciding membership for a fixed automaton, given only the DAG as input. Despite the more restricted question, we can show NP-hardness. Anantharaman, Narendran, and Rusinowitch also show that universality is undecidable for their automata. Finally, with the motivation of representing sets of terms by means of a single DAG, they also consider DAGs where each node has an additional Boolean label. This representation does not seem to be relevant for modeling of natural language semantic structures.

Fujiyoshi (2010) considers DAG automata that are essentially top–down tree automata. Such an automaton is said to accept a DAG if there exists a spanning tree of the DAG that is accepted by the automaton (viewed as a tree automaton). In particular, whenever a DAG is accepted, every other DAG obtained by adding edges is also accepted. This property does not seem to be desirable for modeling semantic structures. Similarly to our result, Fujiyoshi proves that the non-uniform membership problem is NP-complete, but although he also uses a reduction from SAT, the reduction itself is very different from ours (as expected, due to the differences in the automata models).

Among the types of DAG automata studied in theoretical computer science, the model by Priese (2007) is the one that comes closest to the extended DAG automaton introduced in Section 7, even though Priese uses an algebraic setting to describe it. The major difference is that the DAG automata of Priese are able to check that the multiset of states assigned to the roots and leaves of the input DAG belongs to a given regular set, in the sense of Section 7.1. For example, it is possible to express the condition that recognized DAGs shall have a unique root. At first sight, this may appear to be a minor point, but this is not so. Section 4.1 shows that the path languages of our model are regular, whereas they are not even context-free once it becomes possible to express that a DAG has a unique root (which is also observed by Priese [2007]). We consider this to be an indication that our DAG automata are better suited for studying semantic structures because we expect those to have regular path languages, and in the interest of algorithmic results one should not use unnecessarily powerful models. In the more general setting of Priese, our recognition algorithm does not apply, and our proof of the polynomial decidability of the emptiness problem, and the corresponding result for finiteness of Blum and Drewes (2016), break down. Apart from the mentioned study of path languages, the questions studied by Priese are essentially disjoint with those studied in this article.

Another automaton model for graph processing is the graph acceptor by Thomas (1991, 1996). A graph acceptor consists mainly of a finite set of pairwise non-isomorphic r-tiles that play the role of the rules. Each tile is an r-sphere (i.e., a graph with a center node whose distance to all other nodes is at most r). Each node of such a tile carries a state. A run on an input graph G is then a mapping of states to the nodes of G such that each node is the center of one of the tiles. The definition of the graph acceptor includes an occurrence constraint, a Boolean combination of conditions that restrict the number of occurrences of each tile. A given run is accepting if the occurrence constraint is satisfied. The expressiveness of the model can be characterized by existential monadic second-order logic (Thomas 1996), and it can be extended by weights (Droste and Dück 2015). Similar to our basic (non-extended) model, graph acceptors of this type recognize graph languages of bounded degree. However, because of the overlapping of tiles in runs and the occurrence constraint, they are considerably more powerful than our DAG automata (and thus too powerful for our purposes) unless the tiles are required to have the radius 0 (i.e., they are single nodes). The latter restriction results in too weak a model, because it cannot say anything about the edges in the graph if each tile is just a single node.

More results on the (non-extended) DAG automata invented in this article were proved by Blum (2015) and Blum and Drewes (2016). In particular, an alternative proof of the regularity of path languages was given in Blum (2015) (which is simpler and more constructive, but was conceived after the proof in Section 4.1), and the polynomial decidability of the finiteness problem was proved.

Without going into further detail, we mention here some additional publications by diverse authors on automata recognizing DAGs or graphs: Bossut, Dauchet, and Warin (1988); Kaminski and Pinter (1992); Potthoff, Seibert, and Thomas (1994); Bossut, Dauchet, and Warin (1995). Furthermore, there exists considerable work within the XML community on evaluating tree automata and logical queries on compressed representations of trees, which are DAGs (see, e.g., Frick, Grohe, and Koch 2003; Lohrey and Maneth 2006). This work seems to be only tangentially related to the present article because it is not interested in the DAG as a structure in its own right (and automata that define DAG languages), as we are.

4. Properties

In this section we consider only unweighted DAG automata. We explore three properties of such DAG automata and of the (unweighted) DAG languages recognized by them:

- With multiple roots, the path languages of DAG automata are regular; but not under the constraint of a single root (Section 4.1).
- Hyperedge replacement graph languages are closed under intersection with languages recognized by DAG automata (Section 4.2).
- Testing for emptiness of DAG automata is decidable under our definition, but not under the original definition by Kamimura and Slutzki (Section 4.3).

The results in this section are not required for understanding Sections 5–7 on recognition.

4.1 Path Languages

Reading the labels of nodes on the paths in a DAG D from a root to a leaf yields the **path language** of the DAG, denoted by $paths(D)$. (In the following, all paths are assumed to start at a root; their rootedness will thus no longer be mentioned.) The path language of a set L of DAGs is the union $paths(L) = \bigcup_{D \in L} paths(D)$ of the path languages of its individual DAGs. We now show that the path language of a recognizable DAG language is always a regular string language. Thus, in this respect our DAG automata are similar to those by Kamimura and Slutzki (1981), whose path languages are trivially regular. However, if we restrict recognizable DAG languages to DAGs with only one root, then this no longer holds. In fact, in this case even non-context-free path languages are obtained, as in the case of Priese (2007).

Let us first show that path languages of recognizable DAG languages (without the restriction to unique roots) are regular. To this end, recall that we have defined DAGs as *connected* directed acyclic graphs. Let us now drop the connectedness assumption, and consider *arbitrary* directed acyclic graphs, which we call nc-DAGs. Then any nc-DAG can be written as the finite disjoint union $D_1 + \cdots + D_k$ of (connected) DAGs $D_1, \ldots, D_k$, for $k \geq 1$. Here $D + D'$ is used to denote the disjoint union of DAGs D and D'.

We define as $[\![M]\!]_+$ the language of nc-DAGs recognized by M: In words, each nc-DAG in $[\![M]\!]_+$ is the disjoint union of one or more DAGs from $[\![M]\!]$. We extend our definition of path language of a DAG to nc-DAGs and to languages of nc-DAGs. Let $D_1, \ldots, D_k \in [\![M]\!]$. We have $paths(D_1 + \cdots + D_k) = paths(D_1) \cup \cdots \cup paths(D_k)$. It directly follows that $paths([\![M]\!])$ and $paths([\![M]\!]_+)$ coincide. This observation will be used later to simplify our proof.

Another useful observation is the following. Consider a DAG $D = (V, E, lab, src, tar)$ and two edges $e_1, e_2 \in E$. Let $D[e_1 \leftrightarrow e_2]$ denote the graph that is obtained from D by interchanging the targets of e_1 and e_2. More precisely, if v_i ($i = 1, 2$) is the node such that $e_i \in in(v_i)$, then $D[e_1 \leftrightarrow e_2]$ has $e_1 \in in(v_2)$ and $e_2 \in in(v_1)$ but is otherwise identical to D. It is not difficult to see that the edge interchange operator we have just defined might introduce cycles, that is, $D[e_1 \leftrightarrow e_2]$ may no longer be a DAG. However, in what follows we will use this operator in a restricted way, such that the resulting graph is still a DAG.

Suppose that $D \in [\![M]\!]_+$ and let ρ be an accepting run of M on D. If $D = D_1 + D_2$ and, for $i = 1, 2$, e_i is an edge of D_i such that $\rho(e_1) = \rho(e_2)$, then $D[e_1 \leftrightarrow e_2] \in [\![M]\!]_+$. This is true because $D[e_1 \leftrightarrow e_2]$ is still acyclic (because e_1 and e_2 belong to distinct connected components of D) and ρ is an accepting run on $D[e_1 \leftrightarrow e_2]$ as well.

Now, let us turn M into M' by adding a unique leaf symbol ℓ, adding a new state f and the transition $\{f\} \xrightarrow{\ell} \emptyset$, and turning all original transitions of the form $\{q_1, \ldots, q_k\} \xrightarrow{a} \emptyset$ into $\{q_1, \ldots, q_k\} \xrightarrow{a} \{f\}$. Thus, the DAGs recognized by M' are those originally recognized by M, but with additional leaves labeled ℓ added as leaves below the original leaves, and the accepting runs of M' are those of M, extended by labeling the (unique) outgoing edges of the original leaves with f.

For a string $w \in \Sigma^+$, let $\Delta(w)$ denote the set of all states q for which there exists an accepting run ρ of M' on a DAG D such that some path labeled w leads to an edge e with $\rho(e) = q$. Hence, $paths([\![M]\!]) = \{w \mid f \in \Delta(w)\}$. By the Myhill-Nerode theorem, it therefore suffices to show that the equivalence relation $\sim$, given by $w_1 \sim w_2$ if and only if $\Delta(w_1) = \Delta(w_2)$, is a right congruence. In other words, if $\Delta(w_1) = \Delta(w_2)$ and w is any string, then $w_1 w \in paths([\![M]\!])$ if and only if $w_2 w \in paths([\![M]\!])$.

So, assume that $\Delta(w_1) = \Delta(w_2)$ and $w_1 w \in paths([\![M]\!])$. Then there is some $D_1 \in [\![M']\!]$ containing a path p whose node labels are $w_1 w \ell$. Let ρ_1 be a run on D_1 and consider the $|w_1|$-th edge e_1 on p, i.e., the edge between w_1 and w. Then $\rho_1(e_1) \in \Delta(w_1)$.

Choose any $D_2 \in [\![M']\!]$, edge e_2, and accepting run ρ_2 such that some path to e_2 in D_2 is labeled by w_2 and $\rho_2(e_2) = \rho_1(e_1)$. Note that D_2, e_2, and ρ_2 exist because $\Delta(w_1) = \Delta(w_2)$. Now, let $D = D_1 + D_2$. By the observation above the graph $D[e_1 \leftrightarrow e_2]$ is in $[\![M']\!]_+$. Furthermore, it obviously contains the path $w_2 w \ell$, which means that $f \in \Delta(w_2 w)$ and thus $w_2 w \in \mathit{paths}([\![M]\!])$, as required.

We have thus shown that the path language of every recognizable DAG language is a regular string language. The proof of this statement relies crucially on the fact that DAGs in $[\![M]\!]$ may have several roots: We considered the disconnected graph $D = D_1 + D_2 \in [\![M']\!]_+$ and turned it into $D[e_1 \leftrightarrow e_2] \in [\![M']\!]_+$. However, the latter may be connected and may thus, in fact, be an element of $[\![M']\!]$, while containing the roots of both D_1 and D_2.

To end this section, we discuss two examples showing that, indeed, the regularity of path languages (and even its context-freeness) is lost if single-rootedness is imposed on the DAGs (see Priese [2007] for similar arguments). More precisely, let $[\![M]\!]_s = \{D \in [\![M]\!] \mid D \text{ has only one root}\}$. Then $\mathit{paths}([\![M]\!]_s)$ is not necessarily context-free, as the following two examples show.

Example 5
Let $\Sigma = \{a, b, \circ\}$ and consider the DAG automaton with states q, r, r', s and transitions

$$\emptyset \xrightarrow{a} \{q, r\}, \quad \{q\} \xrightarrow{a} \{q, r\}$$
$$\{r\} \xrightarrow{\circ} \{r'\}$$
$$\{q, r'\} \xrightarrow{b} \{s\}, \quad \{s, r'\} \xrightarrow{b} \{s\} \mid \emptyset$$

In an accepting run on a DAG having a single root (labeled by a) every a is related to a uniquely determined b, and vice versa, by paths of the form $a \to \circ \to b$ (where $\rho(e) = r$ and $\rho(e') = r'$ for the incoming and outgoing edge, respectively). Hence, the intersection of the path language of $[\![M]\!]_s$ with a^*b^* is $\{a^n b^n \mid n \geq 1\}$, a strictly context-free language. This means that the path language of $[\![M]\!]_s$ cannot be regular. Note that the construction breaks down if arbitrarily many roots are allowed (i.e., if $[\![M]\!]$ is considered); in this case, no "counting" is possible and we simply get a^+b^+.

Example 6
In a similar way, one may build M such that $\mathit{paths}([\![M]\!]_s)$, intersected with $\{a_1, \ldots, a_k\}^*$, $k \geq 2$, is equal to $\mathit{MIX}(k)$, the language of all strings over the alphabet $\{a_1, \ldots, a_k\}$ that contain the same number of occurrences of each symbol in this alphabet. To simplify the construction, we show how to obtain all strings of the form $\circ w$ such that $w \in \mathit{MIX}(k)$. Let $\Sigma = \{\circ, a_1, \ldots, a_k\}$. We use states $q, r, r_1, \ldots, r_k$ and the following transitions:

$$\emptyset \xrightarrow{\circ} \{q, r\}$$
$$\{r\} \xrightarrow{\circ} \{r, r_1, \ldots, r_k\} \mid \emptyset$$
$$\{q, r_i\} \xrightarrow{a_i} \{q\} \mid \emptyset \quad \text{for } 1 \leq i \leq k$$

An example run of this automaton (for $k = 3$) is illustrated in Figure 8. Similarly to the previous example, taking the intersection of $\mathit{paths}([\![M]\!]_s)$ with the regular language $\{\circ w \mid w \in \{a_1, \ldots, a_k\}^*\}$ yields the intended language. The reader should easily be able to adapt the automaton in such a way that the initial $\circ$ is dropped.

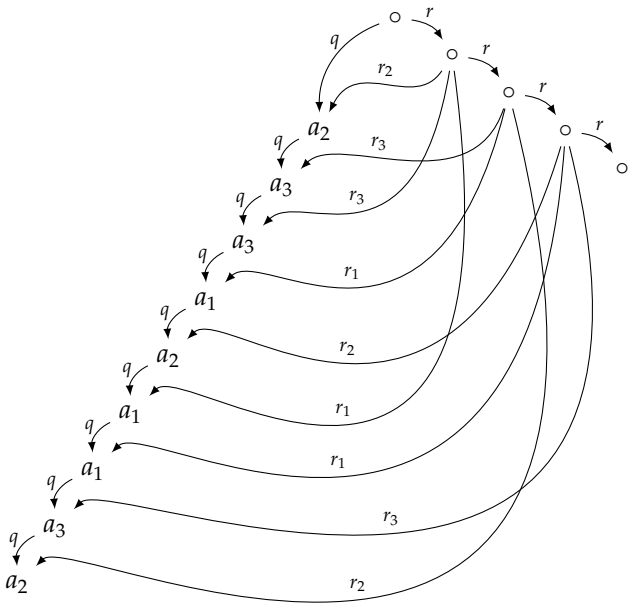


Figure 8
Example run of the DAG automaton that recognizes paths of the form $\circ w$ where $w \in MIX(k)$.

Note that, whereas $MIX(2)$ is well known to be context-free, $MIX(k)$ is not context-free for any $k > 2$. It has been recently shown (Salvati 2014) that $MIX(3)$ can be generated by a Linear Context-Free Rewriting System (Vijay-Shanker, Weir, and Joshi 1987), but it is unknown whether $MIX(k)$, $k > 3$, can be generated by this class.

4.2 Intersection with Hyperedge Replacement Languages

A hyperedge replacement grammar (HRG, see Drewes, Kreowski, and Habel [1997] for an overview) is a context-free type of graph grammar. It can in particular be used to generate DAG languages. Recognition algorithms for HRGs (Lautemann 1990; Chiang et al. 2013) can be thought of as constructions that intersect the graph language $[\![G]\!]$ generated by an HRG G with a single graph. But just as the Cocke-Kasami-Younger algorithm for context-free grammars can be thought of as a special case of intersecting a context-free language with a regular language (Bar-Hillel, Perles, and Shamir 1961), we would more generally like to be able to intersect $[\![G]\!]$ with any recognizable DAG language. In other words, given an unweighted DAG automaton M, we would like to construct an HRG G' such that $[\![G']\!] = [\![G]\!] \cap [\![M]\!]$.

To discuss briefly how this can be done, we need to give a rough introduction to HRGs (adapted to the setting and terminology of the current article). An HRG comes with a ranked alphabet N of nonterminal hyperedge labels, in addition to the alphabet Σ of node labels. Here, saying that N is ranked means that N is specified as a disjoint union $N = \bigcup_k N_k$, where the elements of N_k are said to be the symbols of rank k. A hypergraph H is a graph that may, in addition to the usual elements, contain a finite set of hyperedges. Each hyperedge h has a label $lab(h) \in N_k$ for some $k \in \mathbb{N}$ and a sequence $att(h) \in V^k$ of **attached nodes**. We also view h as having k **tentacles** that connect it to its k attached nodes.

An HR rule $r = (L ::= R)$ consists of a left-hand side L and a right-hand side R. L is a hypergraph that consists of a single hyperedge h labeled by some $X \in N_k$, together with the attached nodes of h, say $u_1, \ldots, u_k$. These nodes should be thought of as being unlabeled as their label is irrelevant. The right-hand side is a hypergraph whose set of nodes also contains $u_1, \ldots, u_k$ (among other nodes). Suppose that a host hypergraph H contains a hyperedge h' labeled with X and attached to nodes $v_1, \ldots, v_k$. Then the rule r can be applied to it, which yields the hypergraph obtained by removing h' from H and inserting the right-hand side of r in its place by identifying each u_i with the corresponding v_i. Figure 9 shows an example of a rule and its application. An HRG G consists of an alphabet N of nonterminals as before, an initial nonterminal of rank 0, and a finite set of HR rules. The generated graph language $[\![G]\!]$, called an HR language, consists of all graphs that can be derived from the initial nonterminal by repeated rule application.

Suppose now that we are given an HRG G that generates graphs over Σ, and an unweighted DAG automaton M that recognizes a DAG language over Σ. We want to construct another HRG G' such that $[\![G']\!] = [\![G]\!] \cap [\![M]\!]$. That this is possible follows from several known results, but most easily using monadic second-order (MSO) logic. Courcelle (1990, Corollary 4.8) shows that the restriction of an HR language by a property expressible in MSO logic yields an HR language (for which a suitable HRG can effectively be constructed). Thus, it suffices to argue that every recognizable DAG language is definable by an MSO formula. Suppose we want to express in MSO logic that a given DAG automaton with state set $Q = \{q_1, \ldots, q_n\}$ accepts a DAG $D = (V, E, lab, src, tar)$. We can do this by constructing an MSO formula that "guesses" an accepting run ρ. The formula states that there exists a partition of E into subsets

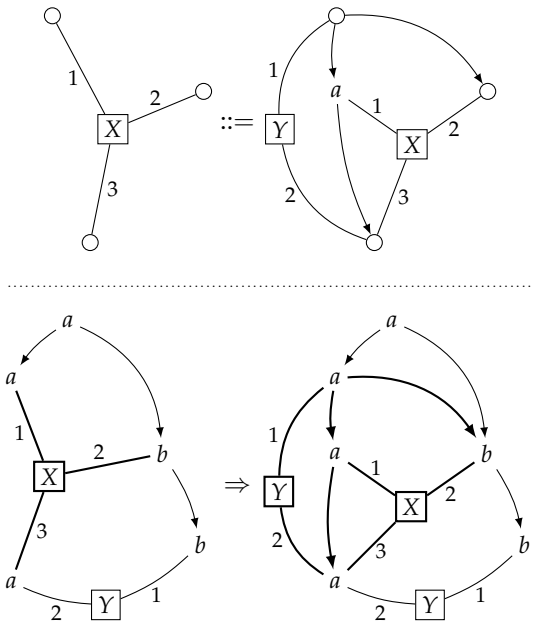


Figure 9
An HR rule (top) and its application to a hyperedge (bottom). For better visibility the replaced hyperedge as well as the portion of the resulting hypergraph that is added by the rule are drawn using thick lines.

$E_1, \ldots, E_n$ such that the following holds: For every node $v \in V$ with $in(v) = \{e_1, \ldots, e_m\}$ and $out(v) = \{e'_1, \ldots, e'_n\}$, there exist $i_1, \ldots, i_m$ and $j_1, \ldots, j_n$ such that

1. $\{q_{i_1}, \ldots, q_{i_m}\} \xrightarrow{lab(v)} \{q_{j_1}, \ldots, q_{j_n}\}$ is a transition of M and
2. $e_r \in E_{i_r}$ for $1 \leq r \leq m$ and $e'_s \in E_{j_s}$ for $1 \leq s \leq n$.

Intuitively, E_i corresponds to the set of all edges e for which $\rho(e) = q_i$.

Let us now sketch a direct construction of G'. Without loss of generality, we may assume that $[\![G]\!]$ is a set of DAGs, because it is well known that the class of HR languages is closed under intersection with the set of all connected acyclic graphs. (This is, in fact, another application of the closedness under intersection with MSO properties.) The idea behind the construction of G' is to use a guess-and-verify strategy to guarantee that only those graphs are generated that have accepting runs in M. To implement this strategy, we augment the nonterminal labels of hyperedges with the guessed information. To understand this, note that every tentacle of a hyperedge intuitively controls a node to which the derivation of this hyperedge will eventually attach a number of incoming and outgoing edges. We have to guess beforehand the multiset of states that an accepting run will assign to these edges. To keep track of this guess, we have to remember two multisets of states for each tentacle, one referring to outgoing edges and one referring to incoming edges that will be generated. Consequently, the new sets N'_k of nonterminal labels of rank k consist of all $(X, \mu_1 \cdots \mu_k, \mu'_1 \cdots \mu'_k)$ such that $X \in N_k$ and $\mu_1, \ldots, \mu_k, \mu'_1, \ldots, \mu'_k$ are multisets of states in Q. The size of these multisets is bounded by the maximum size of multisets in the transitions of M. This makes sure that the set of nonterminals is finite. The initial nonterminal is (X_0, λ, λ), where X_0 is the initial nonterminal of G and λ is the empty sequence.

To define the rules of G', we need a few preparations. Consider a hypergraph H over Σ and N' and a function ρ that maps every (ordinary) edge e of H to a state $\rho(e) \in Q$. In the following, we call ρ a **state assignment** for H. Given such a state assignment and a node v of H, we let $in_\rho(v)$ denote the multiset of states obtained by taking the union of, first, all $\{\rho(e)\}$ with $e \in in(v)$ and, second, all μ_i such that there is a hyperedge h labeled $(X, \mu_1 \cdots \mu_k, \mu'_1 \cdots \mu'_k)$ whose ith tentacle is attached to v. Similarly, $out_\rho(v)$ is the union of all $\{\rho(e)\}$ with $e \in out(v)$ and all μ'_i such that there is a hyperedge h labeled $(X, \mu_1 \cdots \mu_k, \mu'_1 \cdots \mu'_k)$ whose ith tentacle is attached to v.

Now, consider all HR rules $L ::= R$ that can be obtained from rules of G by augmenting each nonterminal label in all possible ways. A rule $L ::= R$ obtained in this way becomes a rule of G' if there exists a state assignment ρ for R such that

1. $in_\epsilon(v) = in_\rho(v)$ and $out_\epsilon(v) = out_\rho(v)$ for all nodes v of L, where ϵ is the unique (empty) state assignment for L, and
2. $in_\rho(v) \xrightarrow{lab(v)} out_\rho(v)$ is a transition of M for every node v of R which is not in L.

By induction on the length of derivations it can be shown that $D \in [\![G']\!]$ if and only if $D \in [\![G]\!]$ and there exists an accepting run of M on D. In other words, $[\![G']\!] = [\![G]\!] \cap [\![M]\!]$, as required.

4.3 Emptiness

The emptiness problem for DAG automata asks, for an unweighted DAG automaton M as input, whether $[\![M]\!] = \emptyset$. As mentioned earlier, the DAG automata of Kamimura and

Slutzki (1981) can encode computations of Turing machines. In particular, this means that their emptiness problem is undecidable. As we shall see next, this sharply distinguishes their DAG automata from ours, whose emptiness problem can be decided in polynomial time as it can be reduced to a particular case of the reachability problem for Petri nets. A similar idea was used by Kaminski and Pinter (1992) to prove the decidability of the emptiness problem for their graph automata. However, in their case it required the use of the general Petri net reachability problem, which leads to an algorithm whose running time is non-elementary. In contrast, we obtain a polynomial algorithm.

Let us first briefly recall Petri nets. A **Petri net** is an unlabeled directed graph $N = (V, E, src, tar)$ such that V consists of disjoint sets T and P of **transitions** and **places**. Edges only point from places to transitions and from transitions to places (i.e., N is a bipartite graph). A **marking** is a mapping $\mu\colon P \to \mathbb{N}$ that assigns to every place p a number $\mu(p)$ of tokens. Intuitively, the idea is that a transition t consumes tokens via edges leading from places to t and it produces tokens via edges leading from t to some places. We make this more precise, as follows. For a place p and a transition t let $input_t(p) = |in(t) \cap out(p)|$ be the number of times p occurs as an input place of t. Similarly, let $output_t(p) = |out(t) \cap in(p)|$ be the number of times p occurs as an output place of t. For a given marking μ, a transition t can **fire** if $\mu(p) \geq input_t(p)$ for each place p, that is, if there are enough tokens on the input places of t. In this case, the firing of t yields the marking μ' given by $\mu'(p) = \mu(p) - input_t(p) + output_t(p)$ for all $p \in P$.

Note that a place p can be an input and output place of t at the same time—that is, we may have $input_t(p) > 0$ and $output_t(p) > 0$. A simple example of a Petri net consisting of one transition together with its input and output places is shown in Figure 10, where the bar represents the transition and the circles represent places. The transition can fire if the topmost place contains at least one token. If it does fire, the token on the topmost place is immediately reproduced. At the same time, four additional tokens are placed on the places at the bottom, namely, two on the place in the middle and one on each of the other two places.

Naturally, a **firing sequence** is a sequence of admissible firings. It transforms an initial marking into a final marking. The **Petri net reachability problem** is the following problem: Given a Petri net and two markings μ, μ', is μ' reachable from μ via some firing sequence? This problem is known to be decidable, but no solution with a primitive recursive running time is known (Reutenauer 1990; Esparza and Nielsen 1994). Fortunately, for our purpose it suffices to consider the case where both μ and μ' are equal to the zero marking $\mathbf{0}$, that is, $\mu(p) = \mu'(p) = 0$ for all places p. If $\mathbf{0}$ is reachable from itself in a Petri net N via a nonempty firing sequence, then we say that N is **structurally cyclic**, because then it holds for all markings μ that μ is reachable from itself. Drewes and Leroux (2015) show that it is decidable in polynomial time whether a Petri net is structurally cyclic.

We can reduce the emptiness problem for DAG automata M to the question of whether a Petri net is structurally cyclic, as follows. Every state of the DAG automaton becomes a place of the Petri net N and every transition $t = (\{q_1, \ldots, q_m\} \xrightarrow{\sigma} \{r_1, \ldots, r_n\})$

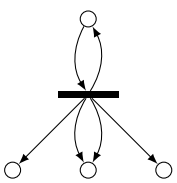


Figure 10
A Petri net with four places and one transition.

of M becomes a transition of N in an obvious way: for $1 \leq i \leq m$, there is an edge pointing from q_i to t, and for $1 \leq j \leq n$, there is one pointing from t to r_j.

To argue for the correctness of the construction, let us consider DAGs that are partial in the sense that, for some edges e, there is no node v with $e \in in(v)$. Such edges are "downward dangling." Now, given a firing sequence starting with the empty marking, we can inductively construct a run on a corresponding partial DAG. The initial empty marking of N corresponds to the empty DAG (with no nodes and zero dangling edges). After some firings the Petri net has reached a marking μ and we have inductively constructed a partial DAG D and a run ρ on D such that for each state q, there are exactly $\mu(q)$ dangling edges e with $\rho(e) = q$. Now suppose that, in N, a transition fires, which was obtained from transition $\{q_1, \ldots, q_m\} \xrightarrow{\sigma} \{r_1, \ldots, r_n\}$ of the DAG automaton M. To reflect the firing of t, we add a node v labeled by σ to D and choose previously dangling edges $e_1, \ldots, e_m$ with $\rho(e_i) = q_i$ as incoming edges of v; n new outgoing dangling edges $e'_1, \ldots, e'_n$ are attached to v, and ρ is extended by defining $\rho(e'_i) = r_i$ for $1 \leq i \leq n$. Clearly, D is a DAG without dangling edges if $\mu = \mathbf{0}$. Thus, $[\![M]\!] \neq \emptyset$ if $\mathbf{0}$ is reachable from itself in N. In a similar way, if M accepts a DAG D, a run of M on D can easily be turned into a nonempty firing sequence of N (under the top–down interpretation of runs) that turns $\mathbf{0}$ into itself. Thus, we have reduced the emptiness problem for DAG automata to the problem of deciding whether a Petri net is structurally cyclic. Clearly, the reduction can be computed in polynomial time (and, in fact, in logarithmic space). Using the main result of Drewes and Leroux (2015) mentioned earlier, we conclude that the emptiness problem for DAG automata is decidable in polynomial time.

5. Recognition

We consider the recognition problem for unweighted DAG automata: For a DAG automaton M and a DAG D, does M accept D? This problem turns out to be NP-complete even in case M is fixed (i.e., instead of both M and D, only D is the input). (In theoretical computer science, the variant where M is part of the input is called the *uniform* membership problem; accordingly, the one where M is fixed is the potentially easier *non-uniform* one.) The situation is similar to that of the recognition problem based on the hyperedge replacement grammar introduced in Section 4.2, which is NP-complete even for a fixed grammar (Aalbersberg, Rozenberg, and Ehrenfeucht 1986; Lange and Welzl 1987). On the positive side, as we shall see in Section 6, recognition by a (fixed) DAG automaton can be done in polynomial time for input graphs of bounded treewidth, which is encouraging in view of Table 1.

5.1 NP-completeness

It is easy to see that recognition is in NP even if the automaton is part of the input: We can nondeterministically "guess" an assignment of states to the edges of D and check in linear time whether it constitutes an accepting run of M. Next, we show that recognition is NP-complete. Like Fujiyoshi (2010), we do this by reduction from SAT, but the reduction is different (because our DAG automata differ essentially from his).

Because we want to prove NP-completeness of the non-uniform membership problem, that is, for a fixed DAG automaton, we construct a single DAG automaton M and a reduction that turns any propositional formula ϕ into a DAG D_ϕ that is accepted by M if and only if ϕ is satisfiable. We first define D_ϕ. Thus, assume that we are given a propositional formula ϕ (which we do not require to be in conjunctive normal form). We use the alphabet $\Sigma = \{true, \wedge, \vee, \neg, x\}$. First, we construct in the obvious way the tree

T_ϕ corresponding to ϕ (where every occurrence of a variable x_i is represented by a node labeled x). We then add a special root node labeled *true* on top of the tree. Intuitively, the root node represents the claim that ϕ evaluates to *true* under an appropriate assignment. Finally, for every variable x_i, if there are $n+1$ nodes $u_0, \ldots, u_n$ in T_ϕ that represent the occurrences of x_i in ϕ from left to right, we add edges from u_{j-1} to u_j for $j = 1, \ldots, n$. Thus, all nodes representing the same variable are linked together in a chain.

Example 7
For $\phi = ((x_1 \vee x_2) \vee \neg x_3) \wedge (\neg x_2 \vee (x_4 \vee x_1))$, the resulting DAG D_ϕ is shown in Figure 11, where we have added indices to the x-labeled nodes in order to illustrate the correspondence with the formula ϕ.

We can easily construct a DAG automaton M that, for every formula ϕ, accepts DAG D_ϕ if and only if ϕ is satisfiable. The automaton has just two states, t and f, to compute a truth value for each node in a consistent way by means of a guess-and-verify technique. The only transition for *true* is $\emptyset \xrightarrow{true} \{t\}$.

The transitions for processing conjunctions, disjunctions, and negations are the expected ones:

$$\{t\} \xrightarrow{\wedge} \{t,t\} \qquad \{f\} \xrightarrow{\wedge} \{t,f\} \qquad \{f\} \xrightarrow{\wedge} \{f,f\}$$

$$\{t\} \xrightarrow{\vee} \{t,t\} \qquad \{t\} \xrightarrow{\vee} \{t,f\} \qquad \{f\} \xrightarrow{\vee} \{f,f\}$$

$$\{t\} \xrightarrow{\neg} \{f\} \qquad \{f\} \xrightarrow{\neg} \{t\}$$

Then for the nodes corresponding to the variables, we need the following transitions, for $b \in \{t,f\}$:

$$\{b,b\} \xrightarrow{x} \{b\}, \quad \{b\} \xrightarrow{x} \{b\}, \quad \{b,b\} \xrightarrow{x} \emptyset, \quad \text{and} \quad \{b\} \xrightarrow{x} \emptyset$$

These ensure that multiple occurrences of the same variable (i.e., occurrences of x that are "chained together") are assigned the same truth value. It should be clear that D_ϕ is accepted by this DAG automaton if and only if ϕ is satisfiable.

Note that, no matter whether we construct runs top–down or bottom–up, there is always nondeterminism involved. Under the top–down view, the transitions for $\wedge$ and $\vee$ are nondeterministic (reflecting the fact that $\wedge$ and $\vee$ are not injective) whereas

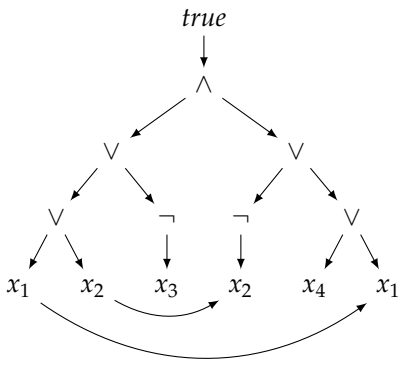


Figure 11
Example instance in the reduction of 3-SAT to DAG automata recognition. The 3-SAT instance is $\phi = ((x_1 \vee x_2) \vee \neg x_3) \wedge (\neg x_2 \vee (x_4 \vee x_1))$. We have added indices to the x-labeled nodes merely to illustrate the correspondence with ϕ.

those for x are deterministic. Conversely, under the bottom–up view, the transitions for x become nondeterministic whereas those for $\wedge$ and $\vee$ become deterministic (because $\wedge$ and $\vee$ are functions). Intuitively, the top–down process corresponds to guessing the values of subtrees and verifying consistency. In contrast, the bottom–up process guesses an assignment of truth values and computes the resulting truth value of ϕ deterministically in order to check that it results in *true*. In both cases, the outlined computational difficulty is preserved.

5.2 Algorithm

We provide an algorithm for a more general problem than the recognition problem for unweighted DAG automata: Given a weighted DAG automaton M and a DAG D, what is the total weight (in the semiring $\mathbb{K}$) of all runs of M for D? This includes in particular the recognition problem, because unweighted DAG automata are a special case of general DAG automata, as explained at the end of Section 3.1. We also obtain an analogue of the Viterbi algorithm if we define $\otimes$ and $\oplus$ to be multiplication and maximum. In Section 5.3 we will also discuss how to use this algorithm for learning transition weights from data.

We have already discussed in Example 3 how our DAG automata generalize finite automata for strings. In order to introduce our algorithm for DAG automata, we therefore consider the analogous problem for finite automata: Given an input string w, find the total weight of all runs of a nondeterministic weighted finite automaton M on w. Let Q be the state set of M. A naïve algorithm for this problem would consider all possible assignments of states in Q to the $|w| + 1$ inter-symbol positions of w, under the restriction that the first position is assigned the unique starting state for M. For each such assignment, we then check against M's transitions that it corresponds to a run of M and, if this is the case, we add in the weight of that run. The number of possible assignments is $|Q|^{|w|}$ and each assignment can clearly be checked in time $\mathcal{O}(|w|)$. If we assume that the semiring operations can be computed in constant time, the algorithm runs in time $\mathcal{O}\left(|Q|^{|w|}|w|\right)$.

A better algorithm, the forward algorithm (Baum 1972), uses dynamic programming to run in polynomial time in the size of both w and M. This is reported in Algorithm 1. We view w as a sequence of tokens w_i from the alphabet of M. Symbols s and F denote the initial state and the final state set, respectively, of M. Symbol δ denotes the transition function, mapping a pair of states and an input symbol from M to a weight. For instance, $\delta(q, w_i, r)$ is the weight of the transition that takes M from state q to state r upon reading token w_i.

Algorithm 1 (Forward algorithm) Sum the weights of all computations of a finite automaton on a single string.

$n = |w|$
$\alpha[0, s] \leftarrow 1$
for $i \leftarrow 1, \ldots n$ **do**
  for $r \in Q$ **do**
    $\alpha[i, r] = \bigoplus_{q \in Q} \alpha[i-1, q] \otimes \delta(q, w_i, r)$
return $\bigoplus_{f \in F} \alpha[n, f]$

The algorithm processes w from left to right, computing the weights of larger and larger prefixes of w. More precisely, for each prefix $w_1 w_2 \cdots w_i$ of w and for each state $r \in Q$, we compute the sum of the weights of all runs of M that start in s, read $w_1 w_2 \cdots w_i$, and end up in r. This quantity is then stored in a chart entry $\alpha[i, r]$, for future reuse. In fact, the basic idea underlying Algorithm 1 is that $\alpha[i, r]$ can be computed as a function of all quantities $\alpha[i-1, q]$, $q \in Q$, combined with all possible transitions of M over token w_i, using a recursive relation. We call each chart entry $\alpha[i, r]$ a **partial analysis** of w. Observe that each partial analysis of w is uniquely identified by the inter-symbol position i we have reached on w, and by the state r we have reached on M.

The complexity analysis of Algorithm 1 is rather straightforward. Considering the two embedded for-loops and the summation performed at the inner loop, we get a running time of $\mathcal{O}\left(|Q|^2 |w|\right)$.

We are now in a position to discuss the same problem for DAG automata. Let D be an input DAG and let M be our DAG automaton with state set Q. In order to strengthen the similarity with the string case, we view the nodes of D like the tokens of w and the edges of D like the inter-symbol positions of w. A naïve algorithm, similar to the one for finite automata, can be developed for computing the total weight for all runs of M on D. We iterate over all possible assignments of states from Q to edges in E, that is, over all runs, and sum up their weights. The total number of runs is $|Q|^{|E|}$, and the weight of each run can be checked in time $\mathcal{O}(|E|)$. We thus conclude that the algorithm runs in time $\mathcal{O}\left(|Q|^{|E|} |E|\right)$.

Once again, we can do much better by using dynamic programming. The main difference with respect to the string case is that the tokens of D are now organized in some partial order, so we can no longer parse the input from left to right. To deal with this, our algorithm assumes a total ordering of the edges of D, which is provided along with D, and parses D accordingly, as we explain now.

Informally, our parsing algorithm consists of the following two phases.

- First, we make a partial analysis for each node v of D. Each partial analysis records what states the incoming edges might be in and what states the outgoing edges might be in, together with a weight.
- Second, we merge partial analyses into larger and larger partial analyses. For each edge e (following the total ordering of edges provided as input), we contract it, replacing its source node $src(e)$ and target node $tar(e)$ with a new node z. We then retrieve partial analyses associated with $src(e)$ and $tar(e)$ and merge them into new partial analyses associated with z. This process is repeated, ending when all of D has been contracted to a single node with a single analysis. The weight of this analysis is the weight of all the runs on D.

The merging of partial analyses in the second phase requires some additional discussion. If p_1 and p_2 are partial analyses associated with $src(e)$ and $tar(e)$, respectively, the partial analysis p, associated with z, inherits its state assignments from p_1 and p_2. Because the edge e is shared between p_1 and p_2, the merging of p_1 and p_2 can be carried out only if they assign the same state to e. Moreover, if several merges result in several analyses for z with the same state assignments, their weights are summed.

In order to gain a better understanding of these ideas, we discuss a simple example, before providing a precise specification of the algorithm itself.

Example 8
The evolution of the structure of a DAG over a run of our DAG parsing algorithm is shown in Figure 12. We start with DAG D in (a) with node set $\{v_1, v_2, v_3, v_4, v_5\}$. To keep the example simple, we only display one possible assignment of a hypothetical automaton at each node; for instance, at node v_2 we display the partial analysis representing the transition in which q_1 is assigned to the incoming edge, and q_2, q_3 are assigned to the outgoing edges. We then contract the edge from v_2 to v_3, resulting in the new DAG displayed in (b), where node (v_2, v_3) represents the merge of nodes v_2 and v_3. Observe that, after edge contraction, the remaining incoming and outgoing edges at v_2 and v_3 are inherited at (v_2, v_3). All possible partial analyses at v_2 and v_3 are pairwise merged at (v_2, v_3) (again, only one such analysis is displayed). We proceed by contracting the edge from v_1 to (v_2, v_3), the edge from v_4 to v_5, and finally the multiple edges from (v_1, v_2, v_3) to (v_4, v_5), ending up with the final DAG in (d) consisting of a single node $(v_1, v_2, v_3, v_4, v_5)$. In general, whenever we contract an edge e we also contract all parallel edges along with it to avoid creating loops.

Just as DAG automata generalize traditional finite automata defined on strings, our DAG parsing algorithm generalizes Algorithm 1. To see this, imagine applying our

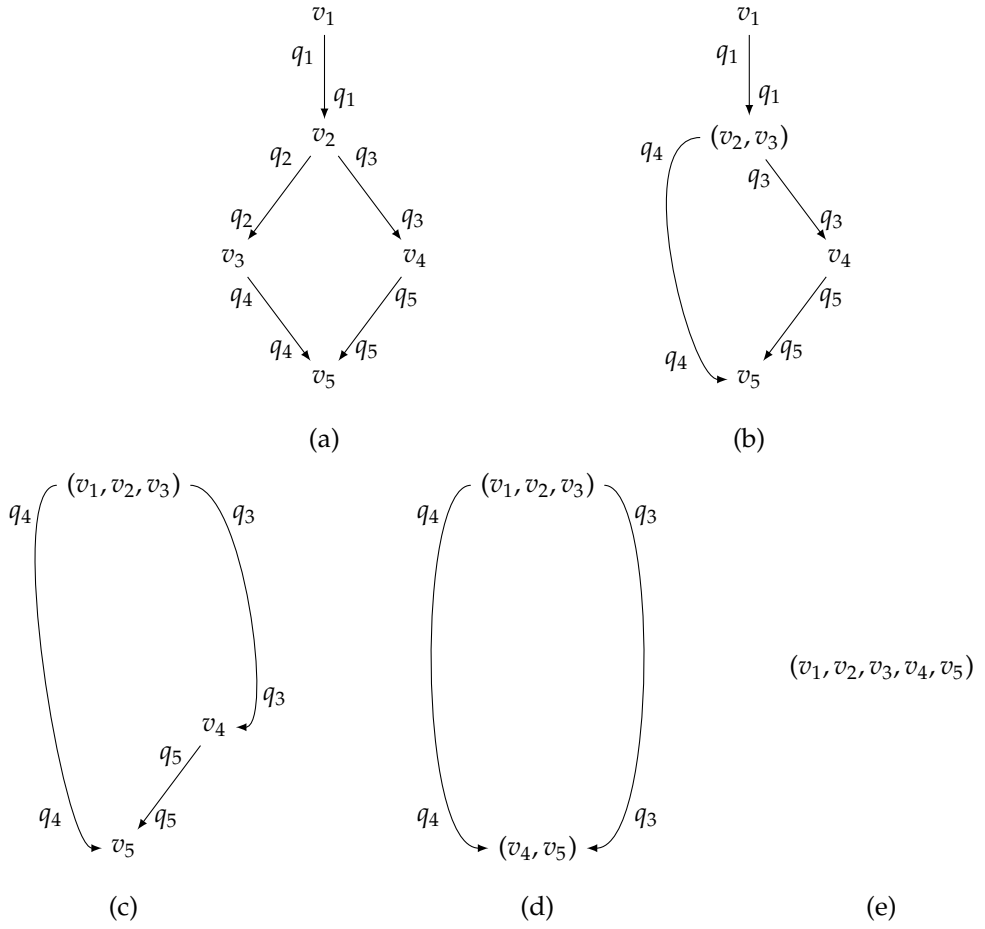


Figure 12
Example run of the DAG parsing algorithm: (a) the starting DAG D; (b), (c), (d) intermediate DAGs obtained after individual edge contraction; (e) final DAG consisting of a single node.

DAG parsing algorithm to a DAG consisting of a single long chain of edges. If the edges are contracted in order from left to right, our DAG parsing algorithm performs the same computation as Algorithm 1, building partial analyses for longer and longer prefixes of the chain. Of course, under some other ordering of the edges, a partial analysis may correspond to a sub-chain of D that is not a prefix. As we will see later, the choice of ordering does affect the overall computational complexity of the algorithm.

We note that the problem of summing over state assignments is an instance of the general problem of weighted constraint satisfaction, where each edge in our input DAG is a variable whose values are states of M, and each node in our DAG is a weighted constraint, with weights specified by the transitions in the automaton. We can solve this problem using general techniques for graphical models (Jensen, Lauritzen, and Olesen 1990; Shafer and Shenoy 1990); the algorithm here is an adaptation of the variable elimination algorithm to our setting.

The pseudocode of our recognition algorithm for DAG automata is reported in Algorithm 2. It uses some additional notation, which we define here. For a node v of D, let $star(v) = in(v) \cup out(v)$. In words, $star(v)$ is the set of edges connecting v to its neighbor nodes. In order to assign states to these edges, we use functions $f\colon star(v) \to Q$. For an edge set $I \subseteq star(v)$, we also write $f|_I$ to denote f restricted to I, and $f[I]$ to denote the multiset of all $f(e)$ such that $e \in I$, i.e., if $I = \{e_1, \ldots, e_n\}$ then $f[I] = \{f(e_1), \ldots, f(e_n)\}$.

Algorithm 2 Compute $[\![M]\!](D)$ by summing up the weights of all runs of M on D.

for each node v **do**
  for all $f\colon star(v) \to Q$ **do**
    $\alpha[v,f] \leftarrow \delta(\langle f[in(v)], lab(v), f[out(v)]\rangle)$
for each edge e in order, s.t. e has not been deleted **do**
  $(u,v) \leftarrow (src(e), tar(e))$
  $I \leftarrow star(u) \cap star(v)$
  create new node z
  $in(z) \leftarrow in(u) \cup in(v) \setminus I$
  $out(z) \leftarrow out(u) \cup out(v) \setminus I$
  for all $h\colon star(z) \to Q$ **do**
    $\alpha[z,h] \leftarrow 0$
  for all $f\colon star(u) \to Q$ **do**
    for all $g\colon star(v) \to Q$ s.t. $f|_I = g|_I$ **do**
      $h = f \cup g \setminus f|_I$
      $\alpha[z,h] \leftarrow \alpha[z,h] \oplus \alpha[u,f] \otimes \alpha[v,g]$
  delete u, v, and all edges in I
one node v remains
return $\alpha[v, \emptyset]$

The complexity of Algorithm 2 depends both on the structure of the input DAG and the order in which we contract its edges. More precisely, the complexity of the optimal edge ordering is determined by the treewidth (see Definition 2 in Section 3.1) of the line graph of D.

Definition 4
The **line graph** of a graph D is the hypergraph $\mathcal{LG}(D)$ obtained as follows: Each edge of D becomes a node of $\mathcal{LG}(D)$; conversely, each node of D with incident edges $e_1, \ldots, e_n$

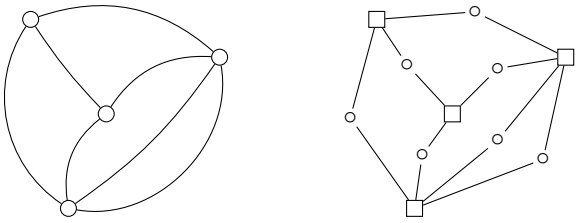


Figure 13
A graph D (left) and its line graph $\mathcal{LG}(D)$ (right); as in Section 4.2 hyperedges of $\mathcal{LG}(D)$ are drawn as squares connected by lines to their attached nodes (which are the edges of D).

becomes a hyperedge of $\mathcal{LG}(D)$ attached to $e_1, \ldots, e_n$ (in any order, as the order will not affect any of the following).

Example 9
A simple example of a graph with four nodes and its corresponding line graph is shown in Figure 13. Note that labels, edge directions, order of attached nodes of hyperedges, and labels are irrelevant and therefore not shown.

Because we want to make use of the treewidth of a line graph, and line graphs are hypergraphs (see Section 4.2), we extend the notion of tree decompositions to hypergraphs in the obvious way: For every hyperedge e, there must be a bag of the tree decomposition that contains all of the attached nodes of e. Note that the bags of the tree decomposition of $\mathcal{LG}(D)$ contain nodes of $\mathcal{LG}(D)$, which correspond to edges of D.

To obtain an optimal edge ordering, first find an optimal tree decomposition, that is, a tree decomposition with minimal width, which we call k. This takes time $\mathcal{O}(|E|^{k+2})$ using the algorithm of Arnborg, Corneil, and Proskurowski (1987). We can also take advantage of the various heuristics and approximation algorithms that are available for treewidth (Gogate and Dechter 2004; Feige, Hajiaghayi, and Lee 2005); as mentioned in Section 2, these heuristics work extremely well on AMR.

Second, visit the bags bottom–up. For each bag b, contract the edges that are in b but not in the parent of b. It can be shown (Rose 1970; Arnborg, Corneil, and Proskurowski 1987) that the maximum degree of any node created by an edge contraction is k. This means that there are at most $(k+1)$ edges in $star(u) \cup star(v)$, and at most $|Q|^{k+1}$ possible state assignments to those edges in the innermost loop of the algorithm.

Then, because there are $|E_D|$ edges to contract, the overall running time of the algorithm is

$$\mathcal{O}\left(|E_D| \cdot |Q|^{\mathrm{tw}(\mathcal{LG}(D))+1}\right) \tag{2}$$

Thus, recognition is polynomial in the number of states but exponential in the treewidth of the line graph of the input graph. Holding these factors constant, recognition is linear in the size of the input graph.

5.3 Learning

We briefly discuss here the problem of learning the weights of our DAG automata, though this in itself is a broad topic worthy of further research. Throughout this section, we assume that our semiring of weights $\mathbb{K}$ is the semiring of real numbers, with the usual addition and multiplication operations.

We define a log-linear model on runs of M on some input DAG D as follows. Let $\Phi : \Theta \to \mathbb{R}^d$ be a mapping from transitions to feature vectors. This extends naturally to runs by summing over the transitions in the run:

$$\Phi(\rho) \stackrel{\text{def}}{=} \sum_{v \in V_D} \Phi(\langle \rho(in(v)), lab(v), \rho(out(v)) \rangle)$$

Let $\mathbf{w} \in \mathbb{R}^d$ be a vector of feature weights, which are the parameters to be estimated. Then we can parameterize δ in terms of the features and feature weights:

$$\delta(t) = \exp \mathbf{w} \cdot \Phi(t)$$

so that

$$\delta(\rho) = \exp \mathbf{w} \cdot \Phi(\rho)$$
$$[\![M]\!](D) = \sum_{\text{run } \rho \text{ on } D} \exp \mathbf{w} \cdot \Phi(\rho)$$

To obtain a probability model of runs of M on D, we simply renormalize the run weights:

$$p_M(\rho \mid D) = \frac{\delta(\rho)}{[\![M]\!](D)}$$

Assume a set of training examples $\{(D_i, \rho_i) \mid 1 \leq i \leq N\}$, where each example consists of a DAG D_i and an associated run ρ_i. We can train the model by analogy with conditional random fields (CRFs), which are log-linear models on finite automata (Johnson et al. 1999; Lafferty, McCallum, and Pereira 2001). The training procedure is essentially gradient ascent on the log-likelihood, which is

$$LL = \sum_{i=1}^{N} \log p_M(\rho_i \mid D_i)$$

$$= \sum_{i=1}^{N} \left(\log \delta(\rho_i) - \log [\![M]\!](D_i) \right)$$

The gradient of LL is:

$$\begin{aligned}
\frac{\partial LL}{\partial \mathbf{w}} &= \sum_{i=1}^{N} \left(\frac{\partial}{\partial \mathbf{w}} \log \delta(\rho_i) - \frac{\partial}{\partial \mathbf{w}} \log [\![M]\!](D_i) \right) \\
&= \sum_{i=1}^{N} \left(\frac{1}{\delta(\rho_i)} \frac{\partial}{\partial \mathbf{w}} \delta(\rho_i) - \frac{1}{[\![M]\!](D_i)} \frac{\partial}{\partial \mathbf{w}} [\![M]\!](D_i) \right) \\
&= \sum_{i=1}^{N} \left(\frac{1}{\delta(\rho_i)} \frac{\partial}{\partial \mathbf{w}} \delta(\rho_i) - \frac{1}{[\![M]\!](D_i)} \sum_{\rho \text{ on } D_i} \frac{\partial}{\partial \mathbf{w}} \delta(\rho) \right) \\
&= \sum_{i=1}^{N} \left(\Phi(\rho_i) - \sum_{\rho \text{ on } D_i} \frac{\delta(\rho)}{[\![M]\!](D_i)} \Phi(\rho) \right) \qquad \text{since } \frac{\partial \delta(\rho)}{\partial \mathbf{w}} = \delta(\rho)\, \Phi(\rho) \\
&= \sum_{i=1}^{N} \left(\Phi(\rho_i) - E_{\rho \mid D_i}[\Phi(\rho)] \right) \qquad (3)
\end{aligned}$$

Unfortunately, we cannot derive a closed-form solution for the zeros of Equation (3). We therefore use gradient ascent. In CRF training for finite automata, the expectation in Equation (3) is computed efficiently using the forward-backward algorithm; for DAG automata, the expectation can be computed analogously. Algorithm 2 provides the bottom–up procedure for computing a chart of inside weights. If we compute weights in the *derivation forest semiring* (Goodman 1999), in which $\otimes$ creates an "and" node and $\oplus$ creates an "or" node, the resulting and/or graph has the same structure as a CFG parse forest generated by CKY, so we can simply run the inside-outside algorithm (Lari and Young 1990) on it to obtain the desired expectations. Alternatively, we could compute weights in the expectation semiring (Eisner 2002; Chiang 2012). Because the log-likelihood LL is concave (Boyd and Vandenberghe 2004), gradient ascent is guaranteed to converge to the unique global maximum.

We may also wish to learn a distribution over the DAGs themselves—for example, in order to provide a prior over semantic structures. A natural choice would be to adopt a similar log-linear framework:

$$p_M(D, \rho) = \frac{\delta(\rho)}{\sum_{D'} [\![M]\!](D')}$$

where $\delta(\rho)$ is a log-linear combination of weights and per-transition features as before. Here, the normalization ranges over all possible DAGs. For some values of the weight vector, this sum may diverge, as in weighted context-free grammars (CFGs; Chi 1999), meaning that the corresponding probability distribution is not defined. More importantly, estimating the normalization constant is computationally difficult, whereas in the case of weighted CFGs it can be estimated relatively easily with an iterative numerical algorithm (Abney, McAllester, and Pereira 1999; Smith and Johnson 2007). A similar problem arises in Exponential Random Graph Models (Frank and Strauss 1986); the most common solution is to use Markov chain Monte Carlo (MCMC) methods (Snijders 2002). To train a model over DAGs, we can perform gradient ascent on the log likelihood:

$$LL = \sum_i \log p_M(\rho_i, D_i)$$

$$\frac{\partial LL}{\partial \mathbf{w}} = \sum_i \Phi(\rho_i) - E_{D',\rho}[\Phi(\rho)]$$

by using MCMC to estimate the second expectation.

Finally, we may wish to learn a distribution over DAGs by learning the states in an unsupervised manner, either because it is not practical to annotate states by hand, or because we wish to automatically find the set of states that best predicts the observed DAGs. This corresponds to a latent variable CRF model (Quattoni, Collins, and Darrell 2004) with states as the hidden variables:

$$p_M(D) = \frac{\sum_{\text{run } \rho \text{ on } D} \delta(\rho)}{\sum_{D'} [\![M]\!](D')}$$

$$LL = \sum_i \log p_M(D_i)$$

$$\frac{\partial LL}{\partial \mathbf{w}} = \sum_i \left(E_{\rho|D_i}[\Phi(\rho)] - E_{D',\rho}[\Phi(\rho)] \right)$$

Here, the second expectation is again over all possible DAGs. We can use the derivation forest semiring to compute the first expectation as with Equation (3), and we can use MCMC methods to estimate the second expectation. Although gradient ascent methods are often used with latent variable CRF models, it is important to note that the log likelihood is not concave—meaning that local maxima are possible.

6. Binarization

Let M be a DAG automaton with set of states Q and let D be an input DAG with set of edges E_D. As we have seen in Section 5.2, the time complexity of Algorithm 2 is $\mathcal{O}\left(|E_D| \cdot |Q|^{\text{tw}(\mathcal{LG}(D))+1}\right)$, where $\text{tw}(\mathcal{LG}(D))$ is the treewidth of the line graph $\mathcal{LG}(D)$. By definition, $\text{tw}(\mathcal{LG}(D))$ is at least the degree of nodes of D minus one, because every node of degree k is turned into a hyperedge of size k that must be covered by some bag. The treewidth of $\mathcal{LG}(D)$ can therefore be quite large. We can improve Algorithm 2 by binarizing both the input DAG and, accordingly, the transitions of our DAG automaton. In this section we develop specialized techniques for the binarization of DAGs and for the binarization of transitions of DAG automata, and prove some relevant properties. Our techniques will further be developed in Section 7 to process DAG languages with unbounded node degree.

6.1 General Idea

A binary DAG is one in which each node has at most two incoming edges and one outgoing edge, or else one incoming edge and two outgoing edges. In order to produce a binary DAG D' from a source DAG D, we introduce a construction that replaces every node of D with a treelet consisting of fresh nodes, and connects the edges of D to these fresh nodes in such a way that the resulting DAG D' is binary. Furthermore, D' preserves all of the information in D, in a way that will be specified later.

Our technique is a generalization of what is known from the theory of tree automata, in particular unranked tree automata, where nodes of any rank are encoded by subtrees entirely consisting of binary nodes; see Comon et al. (2002, Section 8.3) for details. We introduce the idea underlying our DAG binarization technique by discussing a simple example.

Example 10
Consider the DAG D shown in Figure 14(a). From D we construct a new binary DAG D', shown in Figure 14(b), using the following procedure. Let v be a node of D with label σ and with node degree n. Node v is replaced in D' by a binary treelet T_v with exactly n leaf nodes that are labeled by σ. All of the remaining nodes of T_v are labeled by σ': these are internal nodes with one or two children. For instance, if v is the root node of D labeled a, then T_v is the treelet at the top of D' consisting of two binary nodes with label a' and three leaves with label a.

Because the leaves of T_v correspond one-to-one to the edges of D incident on v, they can be used as "docking places" for the original edges. More precisely, each edge e in D such that $src(e) = v$ and $tar(e) = v'$ is used in D' to connect some leaf of T_v to some leaf of $T_{v'}$. Note that, according to this construction, the edge set of D' can be partitioned into the set of fresh edges coming from the treelets and the set of edges coming from the source DAG D; the latter are exactly those edges whose source nodes carry a label $\sigma \in \Sigma$, and are drawn with thick lines in Figure 14(b).

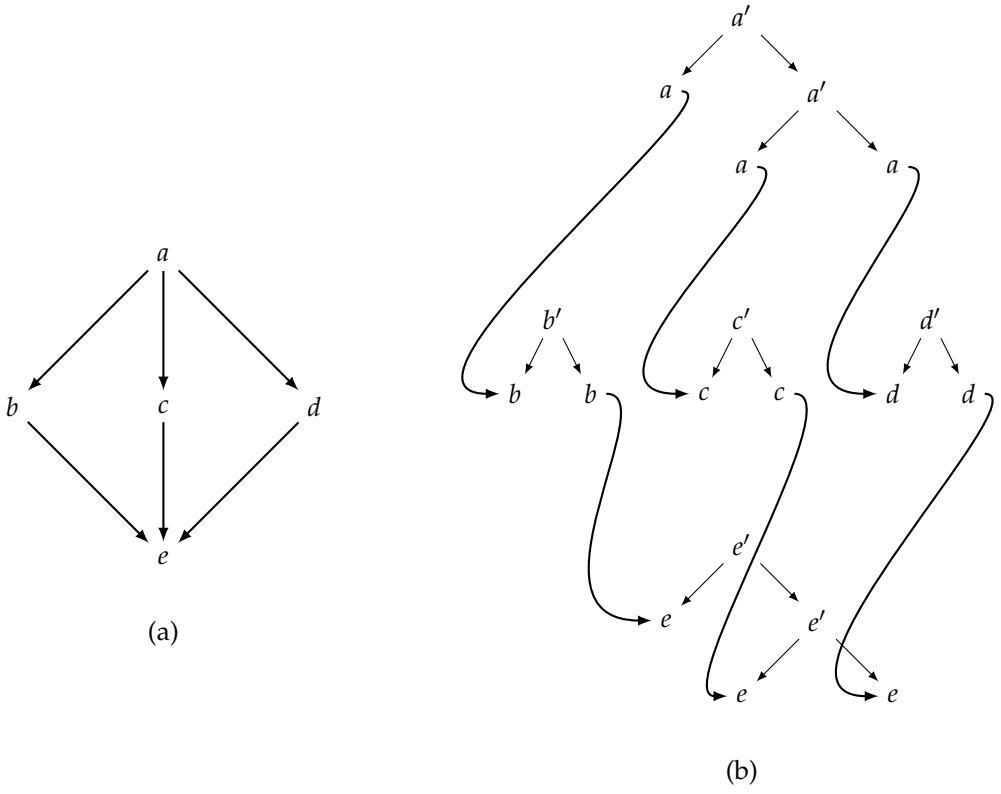


Figure 14
(a) Source DAG D and (b) binarized DAG D'. The edges in D and their counterparts in D' are drawn using thick lines.

In the following, the specific topology of each treelet T_v will be obtained from a tree decomposition of D. Because the leaves of T_v have only one parent and no child, the construction yields a binary DAG.

Along with DAG binarization, we must also replace each transition t of the DAG automaton with a set of "binary" transitions that process the nodes of the binarized graph. The binary transitions have at most two states in the left-hand side and one state in the right-hand side, or else at most one state in the left-hand side and two states in the right-hand side. Again, we demonstrate the intuitive idea underlying the construction by means of a simple example.

Example 11

Consider an unweighted transition $t\colon \{p, q\} \xrightarrow{a} \{r, s\}$ applied to a node v with label a in a DAG D, as shown in the snapshot in Figure 15(a). Consider also the snapshot of the binary DAG D' in Figure 15(b), representing the treelet T_v obtained from v. We discuss how to binarize t such that the resulting transitions can process T_v.

For the binary transitions we use the states p, q, r, s appearing in t, along with some new states of the form (I, O), where I is a subset of the multiset in the left-hand side of t and O is a subset of the multiset in the right-hand side of t. States p, q, r, s will be assigned to the edges of D' that were also present in D, drawn with thick lines in Figure 15(b). States of the form (I, O) will be assigned to the fresh edges of the treelet T_v.

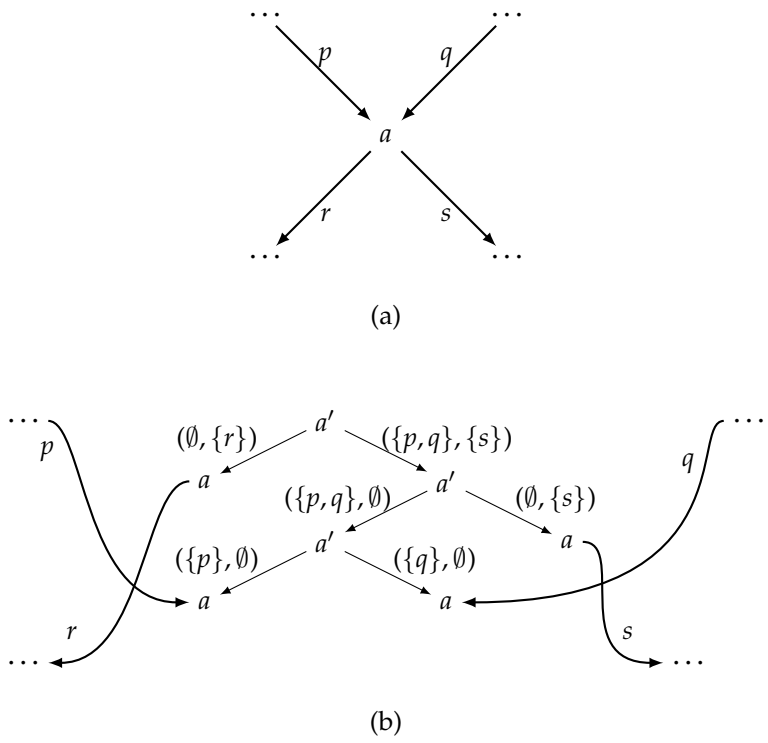


Figure 15
(a) Snapshot of a node labeled a in some DAG D, with two incoming edges assigned states p and q and two outgoing edges assigned states r and s. (b) Snapshot of the binary DAG D' obtained from D, showing the treelet associated with node in (a). As in Figure 14, we use thick lines for edges of D and for their counterparts in D', and we use thin lines for fresh edges in D'.

Consider an edge e of T_v. Let T be the subtree of T_v whose root node is the target node of e. Let also $\mathcal{S}_T$ be the set of edges of D' that are connected to the leaves of T, not including edges internal to T. When viewed as edges from D, the edges in $\mathcal{S}_T$ are a subset of the edges incident on v. Informally, the meaning of a state (I, O) being assigned to edge e is that we expect to find the states in I on the edges within $\mathcal{S}_T$ that are incoming at v, and likewise we expect to find the states in O on the edges within $\mathcal{S}_T$ that are outgoing at v.

Let us discuss three among the several binary transitions obtained from t. Consider the run represented in Figure 15(a). To support intuition, we view this run as a top–down process. The transition

$$t_1 \colon \emptyset \xrightarrow{a'} \{(\emptyset, \{r\}), (\{p, q\}, \{s\})\}$$

is one of those that apply at the (binary) roots of treelets labeled with a', implementing the "guess" that the left subtree will provide the required outgoing edge that is assigned the state r, and the right subtree will provide the required incoming edges that are assigned the states p and q, and the required outgoing edge that is assigned the state s.

A second example is the transition

$$t_2 \colon \{(\{p, q\}, \{s\})\} \xrightarrow{a'} \{(\{p, q\}, \emptyset), (\emptyset, \{s\})\}$$

which processes a node with an incoming edge that has been assigned the state $\{(\{p,q\},\{s\})\}$. Transition t_2 makes the guess that the expected incoming states $\{p,q\}$ are both realized at the left subtree, and that the expected outgoing state in $\{s\}$ is realized at the right subtree.

Finally, our third example is the transition

$$t_3\colon \{(\{q\},\emptyset),q\} \xrightarrow{a} \emptyset$$

which processes two incoming edges and zero outgoing edges. This transition matches the expectation, indicated by $\{q\}$, that there is an incoming edge with state q, and the state actually encountered on the other incoming edge.

Now that we have seen an intuitive description of the procedures for binarizing a DAG and for binarizing a DAG automaton, we can outline the improved version of Algorithm 2:

1. For each transition in the input DAG automaton M, construct the corresponding set of binary transitions to form a new automaton M'.
2. Binarize the input DAG D into DAG D'.
3. Run Algorithm 2 on the binarized DAG D' with automaton M'.

Step 1 is independent of the remaining steps, and can therefore be carried out offline. In the remainder of this section, we discuss at length the process of binarizing a DAG and that of binarizing a DAG automaton, and we present a computational analysis of the improved algorithm.

6.2 DAG Binarization

Let D be some input DAG and let D' be a binarized DAG derived from D. We have already discussed in Section 2 how AMR structures have very small treewidth. For this reason, in the following discussion we use as a term of comparison quantity $\mathrm{tw}(D)$, the treewidth of D.

When processing D', the running time of Algorithm 2 depends on $\mathrm{tw}(\mathcal{LG}(D'))$, the treewidth of the line graph of D', which in turn depends on the choice of the binarization of D. There are several ways in which we can binarize D, resulting in different values of $\mathrm{tw}(\mathcal{LG}(D'))$. However, a bad choice of binarization for D may result in $\mathrm{tw}(\mathcal{LG}(D'))$ much larger than $\mathrm{tw}(D)$. Our objective should therefore be to binarize D in such a way that $\mathrm{tw}(\mathcal{LG}(D'))$ is not much larger than $\mathrm{tw}(D)$. We provide an algorithm for constructing D' from a tree decomposition of D, and we show that $\mathrm{tw}(D') \leq \mathrm{tw}(D)+1$ and $\mathrm{tw}(\mathcal{LG}(D')) \leq 2(\mathrm{tw}(D)+1)$.

In what follows, we exclude from our DAG automata transitions of the form $\emptyset \xrightarrow{a} \emptyset$, which only accept DAGs consisting of a single isolated node. Clearly, this is an uncritical assumption because $\mathcal{D}_\Sigma$ contains only $|\Sigma|$ of these DAGs. This assumption is similar to the exclusion of the empty string from context-free languages when parsing with the CKY algorithm that uses context-free grammars in Chomsky normal form (Aho and Ullman 1972).

We will have to refer to the components of different graphs and tree decompositions. To disambiguate the notation, we will index the components of such an object by the object in question. For example, the edge set of a DAG D will be referred to as

E_D, the source of an edge $e \in E_D$ by $src_D(e)$, and the set of bags of a tree decomposition T by V_T.

In this section we consider tree decompositions of DAGs that are in a special form that we call binary. This has the advantage of greatly simplifying the binarization construction. A tree decomposition T of a DAG D is **binary** if both of the following conditions are met:

- every bag of T has at most two children;
- each edge $e \in E_D$ is explicitly assigned to a unique leaf b of T.

More precisely, every leaf b of T is assigned an edge $edg_T(b) \in E_D$ such that the content of b consists of the two nodes this edge is incident upon. Formally, we have $cont_T(b) = \{src_D(edg_T(b)), tar_D(edg_T(b))\}$. Furthermore, we require that the mapping edg_T is a bijection between the leaves of T and the edges of D. In other words, every edge of D is introduced by a unique leaf b of T. Note that, because edg_T is a bijection, for every edge $e \in E_D$ we have that $edg_T^{-1}(e)$ yields the unique leaf b of T such that $edg_T(b) = e$. In Appendix A, Theorem 8, we show that if a graph has a tree decomposition of width k, then there exists a binary tree decomposition of the same graph also having width k.

Our method of binarization is illustrated in Figure 16 and explained in the following.[4] Let $D \in \mathcal{D}_\Sigma$. For every symbol $\sigma \in \Sigma$, we let σ' be a fresh copy of σ. In the binarized DAG, every node $v \in V_D$ will be represented by a treelet, each of whose nodes is labeled by σ or σ'. To define the binarized version of D, consider a binary tree decomposition T of D. By the definition of tree decomposition, the subtree of T induced by $\{b \in V_T \mid v \in cont_T(b)\}$ forms itself a (binary) tree. Let us denote this treelet by T_v. To distinguish between the copies of $b \in V_T$ appearing in the different treelets T_v such that $v \in cont_T(b)$, we let $[v, b]$ denote the copy of b in T_v. In DAG D of Figure 16, its nodes x, y, u, v are shown instead of their node labels. In the tree decomposition T, the bags b are identified with their Gorn addresses and the boxes show their contents.

Binarization replaces each node $v \in V_D$ by T_v. Formally, D_T is the DAG obtained from the union of all T_v, for $v \in V_D$, by labeling the nodes and inserting the edges of D as follows:

- For every node $[v, b]$ of T_v, $lab_{D_T}([v, b]) = lab_D(v)$ if b is a leaf of T and $lab_{D_T}([v, b]) = lab_D(v)'$ otherwise.
- Let $e \in E_D$ with $(src_D(e), tar_D(e)) = (x, y)$ and $b = edg_T^{-1}(e)$. Then T_x and T_y contain the leaves $[x, b]$ and $[y, b]$, respectively, and we set $src_{D_T}(e) = [x, b]$ and $tar_{D_T}(e) = [y, b]$.

To avoid confusion, we note that, according to the second item above, the edges of D are "reused" in D_T (though of course with other source and target nodes) rather than taking copies, as one would probably do in an implementation.

The construction is illustrated in the middle of Figure 16. (The figure does not show node labels, however. For example, in the sub-DAG resulting from T_x, if $lab_D(x) = \sigma$ then the label of $[x, \epsilon]$, $[x, 1]$, and $[x, 2]$ is σ' and the label of $[x, 1.2]$ and $[x, 2.2]$ is σ.)

Clearly, D_T can be turned into D by contracting each sub-DAG T_v into a single node (with the appropriate label). D_T is binary because the T_v are treelets and each leaf $[v, b]$

4 The DAG D used in Figure 16 already happens to be binary, but this does not affect the construction.

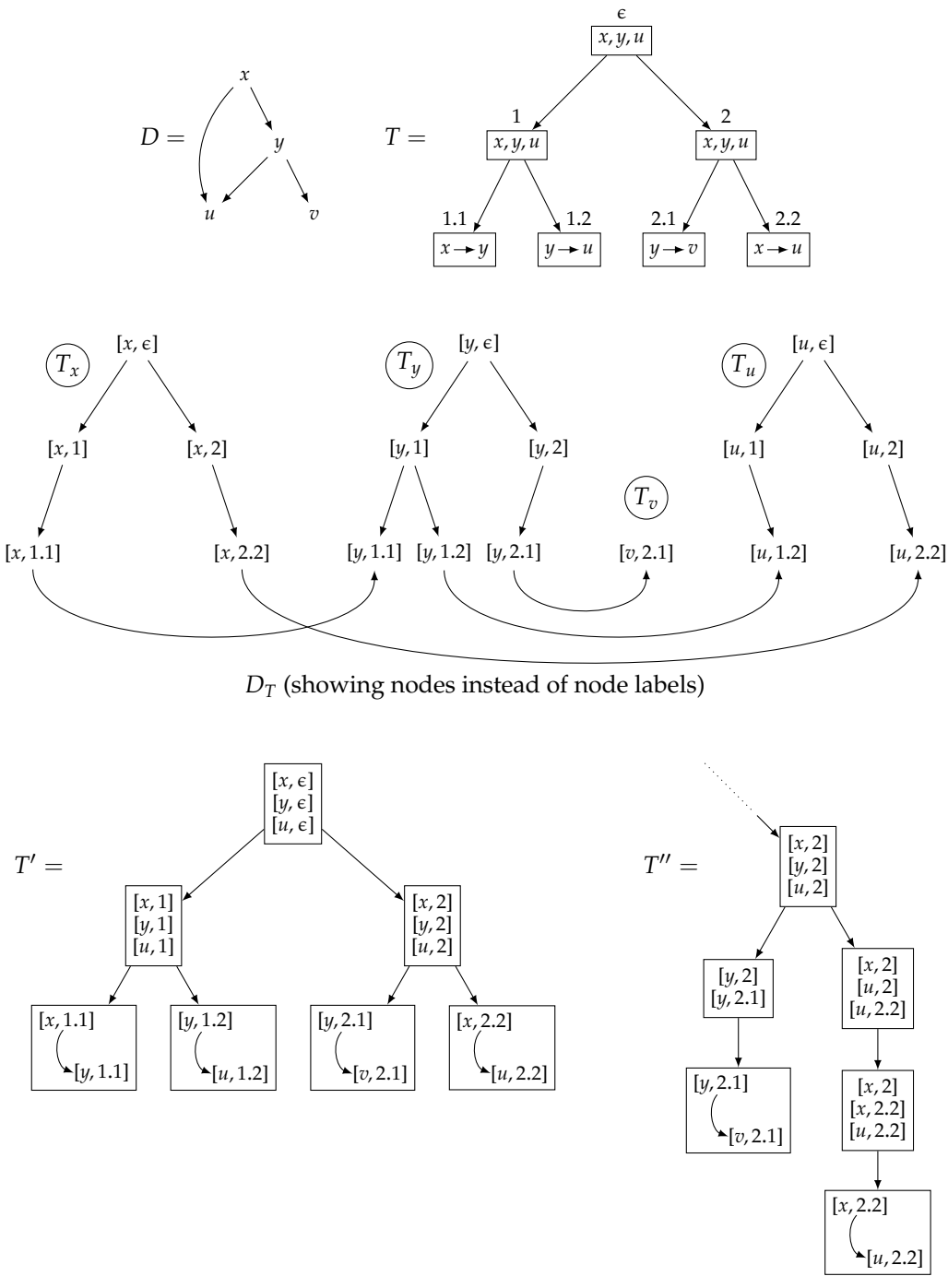


Figure 16
Binarization of a DAG D along a tree decomposition T, yielding D_T. The bottom part illustrates the construction of a tree decomposition of D_T in the proof of Theorem 1.

is attached to exactly one of the original edges $e \in E_D$, namely, to $edg_T(b)$. In particular, D_T does not have cycles.[5]

Note that, as D_T depends on T, one of the effects of binarization is that DAGs representing the same semantic information are not necessarily isomorphic anymore. However, this is not a severe disadvantage because binarization is only a technical tool that allows us to derive efficient algorithms and, in Section 7, transfer results from the ranked case to the unranked one.

Theorem 1
For every DAG D and every binary tree decomposition T of D of width $k \geq 1$, $\mathrm{tw}(D_T) \leq k + 1$.

Proof. As a first step, consider the tree T' that is identical to T, but with the content of bag $b \in V_T = V_{T'}$ being given by $cont_{T'}(b) = \{[v, b] \mid v \in cont_T(b)\}$. Intuitively, (the content of bags of) T' is obtained by overlaying the different copies T_v of T; see again Figure 16. With this definition, T' is not a tree decomposition of D_T yet, but we note that $|cont_{T'}(b)| = |cont_T(b)|$ for all $b \in V_T$ and that every edge $e \in E_D$ can be assigned to the bag $b = edg_{T'}^{-1}(e)$ because, in D_T, e connects the two nodes in $cont_{T'}(b)$. Furthermore, every node $[v, b]$ of D_T occurs in precisely one bag: $[v, b] \in cont_{T'}(b)$. Hence, T' is a tree decomposition of width k except for the fact that those edges of D_T which are arcs of the treelets T_v are not covered by any bags. For this, we shall add intermediate bags to T' in order to construct a valid tree decomposition T'' of D_T.

Consider an arc $e \in E_T$ with $src_T(e) = b$ and $tar_T(e) = c$. A treelet T_v contains a copy e_v of e if $v \in cont_T(b) \cap cont_T(c)$, and in this case we have $src_{T_v}(e_v) = [v, b]$ and $tar_{T_v}(e_v) = [v, c]$. Thus, $\{v_1, \ldots, v_\ell\} = cont_T(b) \cap cont_T(c)$ is the set of nodes v such that the edges e_v exist. To make sure that each e_{v_i} is covered by a bag of T'' we insert, between c and b in T', a rising chain of ℓ bags $b_1, \ldots, b_\ell$. In other words, $b_0 = c$ becomes the child of b_1, which becomes the child of b_2, …to b_ℓ, which becomes the child of $b_{\ell+1} = b$. For $i \in \{1, \ldots, \ell\}$ define

$$cont_{T''}(b_i) = cont_{T'}(c) \setminus \{[v_1, c], \ldots, [v_{i-1}, c]\} \cup \{[v_1, b], \ldots, [v_i, b]\}$$

Intuitively, viewing $b_1, \ldots, b_\ell$ bottom up, the nodes $[v_i, b]$ are introduced while the $[v_i, c]$ are forgotten, but with a delay of one. In Figure 16, this is illustrated for the right subtree of T'. Now, b_i covers e_{v_i}, and $|cont_{T''}(b_i)| = |cont_{T'}(c)| + 1 = |cont_T(c)| + 1 \leq k + 2$. For $b \in V_{T'}$, we let $cont_{T''}(b) = cont_{T'}(b)$. By construction, for a given bag $b \in V_T$, the bags of T'' that contain nodes of the form $[v, b]$ form a connected subgraph of T''. This completes the proof. □

Theorem 2
For every DAG D and every binary tree decomposition T of D of width $k \geq 1$, $\mathrm{tw}(\mathcal{LG}(D_T)) \leq 2(k + 1)$.

Proof. For a node $[u, b]$ of D_T (where $u \in V_D$ and $b \in V_T$) which is not the root of treelet T_u, let $e(u, b)$ denote the arc of T_u (which is an edge of D_T) whose target is $[u, b]$. As an illustration, Figure 17 (top) shows the line graph of the binarized DAG D_T from

5 D_T would not even have cycles if D did, because every cycle would have to enter some T_v through one leaf and exit it through another, which is impossible because T_v is a directed tree.

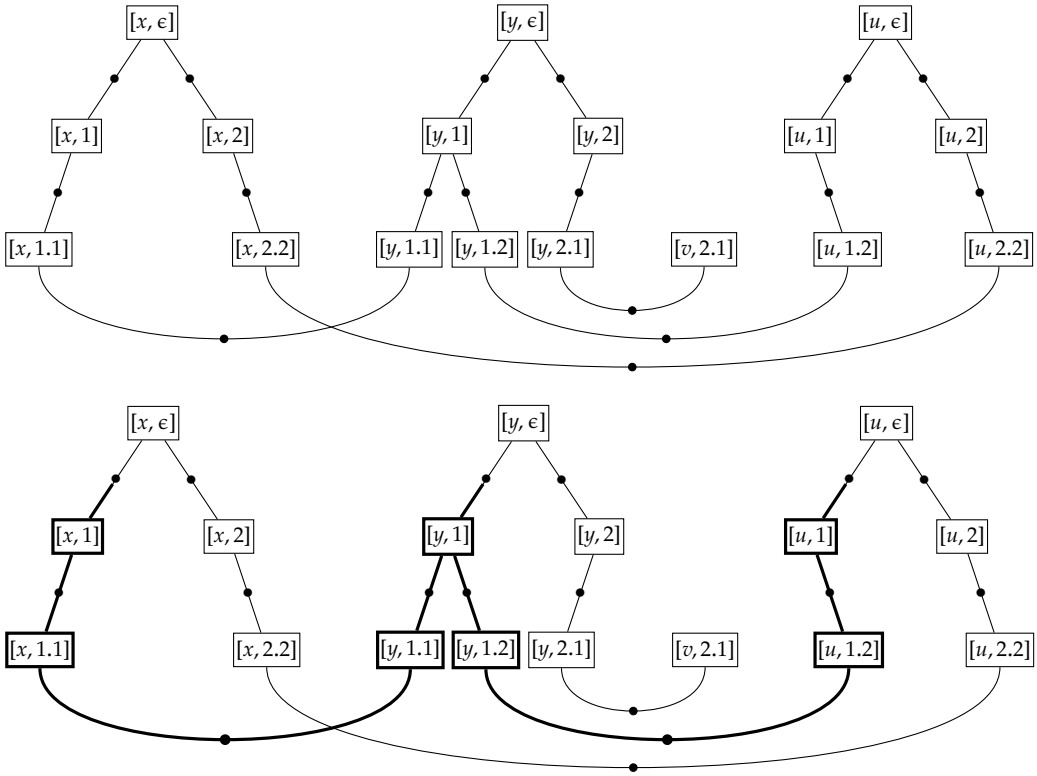


Figure 17
The line graph of the binarized DAG D_T of Figure 16.

Figure 16. The nodes of D_T have become hyperedges drawn as boxes and the edges have become nodes drawn as bullets. The node $e(x, 1.1)$, for instance, is the one on top of the hyperedge $[x, 1.1]$.

Consider a bag b of T and let $D_T(b)$ be the sub-DAG of D_T induced by the treelet nodes $[v, c]$ such that c is a descendant of b (or b itself) and $v \in cont_T(c)$. Let, furthermore, $\mathcal{LG}(D_T, b)$ be the subgraph of $\mathcal{LG}(D_T)$ having as nodes the edges of $D_T(b)$ as well as all edges $e(v, b)$ with $v \in cont_T(b)$, and as hyperedges the nodes of $D_T(b)$. As an example, Figure 17 (middle) indicates $\mathcal{LG}(D_T, 1)$ by means of thick lines.

In a bottom–up manner, starting at the leaves b of T, we construct a tree decomposition T_b of $\mathcal{LG}(D_T, b)$ of width at most $2(k+1)$ such that the root bag of T_b contains $\{e(v, b) \mid v \in cont_T(b)\}$. (Recall that the bags of T_b should contain the edges of D_T, because they are the nodes of $\mathcal{LG}(D_T)$.)

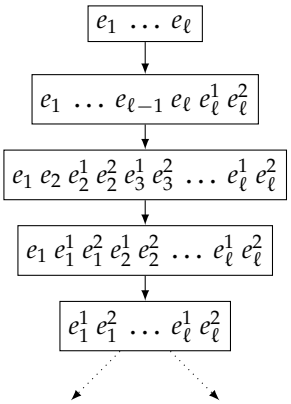
If b is a leaf and $cont_T(b) = \{v_1, v_2\}$, then $\mathcal{LG}(D_T, b)$ contains the edge e of D covered by b (i.e., the one incident on v_1 and v_2), as well as $e(v_1, b)$ and $e(v_2, b)$. Thus, we let T_b consist of a leaf containing $\{e, e(v_1, b), e(v_2, b)\}$ and a root containing $\{e(v_1, b), e(v_2, b)\}$. (If $[v_i, b]$ is the root of T_{v_i}, such as $[v, 2.1]$ in Figure 16, $e(v_i, b)$ does not exist and is omitted from the bags.) Clearly, T_b is as claimed because $k \geq 1$.

Now suppose that b is not a leaf and assume that it has two children c_1, c_2, because this is the interesting case. Combine the inductively computed tree decompositions T_{c_1} and T_{c_2} into a single tree T_0 by adding a root bag b_0 whose contents are the union of the contents of the root bags of T_{c_1} and T_{c_2}. Then T_0 is a tree decomposition of the union G

of $\mathcal{LG}(D_T, c_1)$ and $\mathcal{LG}(D_T, c_2)$ of width $2(k+1)$ whose root b_0 contains the edges $e(v, c_i)$ for all $v \in cont_T(c_i)$ and $i = 1, 2$. If b is the root of T, this completes the construction since then T_0 is also a tree decomposition of $\mathcal{LG}(D_T)$. This is because G is $\mathcal{LG}(D_T)$ without the hyperedges $[v, b]$, which are covered by b_0 as $[v, b]$ is only connected to $[v, c_1]$ and $[v, c_2]$ in T_v (provided that the latter exist). For example, in Figure 17, these are the hyperedges $[x, \epsilon]$, $[y, \epsilon]$, and $[u, \epsilon]$, which are covered by $\{e(x, 1), e(x, 2), e(y, 1), e(y, 2), e(u, 1), e(u, 2)\}$.

Thus, assume finally that b is not the root of T. If $cont_T(b) = \{v_1, \ldots, v_\ell\}$ then b_0 contains the (at most) two outgoing edges $e_i^1 = e(v_i, c_1)$ and $e_i^2 = e(v_i, c_2)$ of $[v_i, b]$, for $i = 1, \ldots, \ell$. (Again, if $v_i \notin cont_T(c_j)$ then e_i^j does not exist and can be disregarded.) The edges e_i^1 and e_i^2 are connected to $e_i = e(v_i, b)$ by the ternary hyperedge $[v_i, b]$ in $\mathcal{LG}(D_T, b)$ (or by a binary hyperedge if only one of e_i^1, e_i^2 exists). This hyperedge must be covered by a bag. As an example, consider $[y, 1]$ in Figure 17 (bottom). Its outgoing edges are $e(y, 1.1)$ and $e(y, 1.2)$, and $[y, 1]$ is the hyperedge that connects them to $e(y, 1)$. The situation for $[x, 1]$ and $[u, 1]$ is similar, even though these have only one outgoing edge each, and are thus binary hyperedges in $\mathcal{LG}(D_T)$.

The bag b_0 contains all of $e_1^1, e_1^2, \ldots, e_\ell^1, e_\ell^2$ that exist. Hence, to cover $[v_1, b], \ldots, [v_\ell, b]$, we can proceed in a way similar to the completion of the tree decomposition T'' in the preceding proof by adding, on top of b_0, a chain of bags as follows:



This completes the construction of T_b. Since $\ell \leq k+1$, the largest bag added contains $2\ell + 1 \leq 2(k+1) + 1$ edges of $\mathcal{LG}(D_T, b)$ (i.e., the width of T_b is at most $2(k+1)$) and the root contains $e_1, \ldots, e_\ell$, as claimed. □

6.3 Transition Binarization

We now describe how to construct the binarized DAG automaton M'. Let M be the source DAG automaton, with state set Q. The set of states of M' is defined as $Q' = Q \cup Q_{\text{io}}$, where each state in Q_{io} is an ordered pair (I, O) of multisets over Q. These states will be assigned to the edges which, in a binarized DAG, D_T, stem from the treelets T_v, as opposed to the original edges of D. The interpretation of assigning (I, O) to an edge e belonging to T_v (i.e., an edge whose source node carries a label of the form σ') is that we are in the process of simulating some transition M applied to v, and that the subtree of T_v rooted at $tar_{D_T}(e)$ collects those incoming and outgoing edges of the original node v that need to be assigned the states in I and O, respectively.

Following this intuition, the transitions of M' are specified as follows. Consider a transition $I \xrightarrow{\sigma} O$ of the original DAG automaton M. The roots of D_T with label σ' are handled by the following transitions of M':

1. $\emptyset \xrightarrow{\sigma'} \{(I, O)\}$ (these transitions process unary roots of treelets).
2. $\emptyset \xrightarrow{\sigma'} \{(I_1, O_1), (I_2, O_2)\}$ for all I_1, I_2, O_1, O_2 such that $I_1 \uplus I_2 = I$ and $O_1 \uplus O_2 = O$ (these transitions process binary roots of treelets).

Note that the unions $I_1 \uplus I_2$ and $O_1 \uplus O_2$ in item 2 (and similarly in item 4) are multiset unions. Thus, no states are "lost" if I_1 and I_2 or O_1 and O_2 overlap. The transitions for binary roots can be omitted if we change the treelets by adding a unary root above every binary root.

Second, for nodes labeled σ' that are not roots, and all $I' \subseteq I$ and $O' \subseteq O$, we use the following transitions:

3. $\{(I', O')\} \xrightarrow{\sigma'} \{(I', O')\}$ (these transitions simply skip unary nodes).
4. $\{(I', O')\} \xrightarrow{\sigma'} \{(I_1, O_1), (I_2, O_2)\}$ for all I_1, I_2, O_1, O_2 such that $I_1 \uplus I_2 = I'$ and $O_1 \uplus O_2 = O'$ (these transitions split I' and O' at binary nodes).

Finally, we let M' contain the transitions:

5. $\{p, (\{p\}, \emptyset)\} \xrightarrow{\sigma} \emptyset$ and $\{(\emptyset, \{q\})\} \xrightarrow{\sigma} \{q\}$ for all $p \in I$ and all $q \in O$ (these transitions process leaf bags of a treelet, matching the individual state at the edge of D incoming or outgoing at the leaf bag).
6. $\{p\} \xrightarrow{\sigma} \emptyset$ if $I = \{p\}$ and $O = \emptyset$ and $\emptyset \xrightarrow{\sigma} \{q\}$ if $I = \emptyset$ and $O = \{q\}$ (these transitions handle the special case of treelets consisting of a single node).

As a slight optimization, the reader may have noticed that the state $(\emptyset, \emptyset)$ is actually useless and can therefore be discarded, together with all transitions in which it appears.

To see that M' works as intended, consider a DAG $D \in \mathcal{D}_\Sigma$, a binary tree decomposition T of D, and the binarized DAG D_T. Given an accepting run ρ of M on D, we can build an accepting run ρ' of M' on D_T, as follows. For all $e \in E_D$, we let $\rho'(e) = \rho(e)$. It remains to assign appropriate states to the edges of the treelets T_v for $v \in V_D$. Consider such a node v and let $\{p_1, \ldots, p_m\} \xrightarrow{\sigma} \{q_1, \ldots, q_n\}$ be the transition of M applied to v, i.e., $(\{p_1, \ldots, p_m\}, \{q_1, \ldots, q_n\}) = (\rho(in_D(v)), \rho(out_D(v)))$. For each edge e' of T_v such that $tar_{T_v}(e')$ is a leaf u of T_v, consider the unique edge $e \in E_D$ which is incident on u in D_T. Let $\rho'(e') = (\{\rho(e)\}, \emptyset)$ if $tar_{D_T}(e) = u$ (which means that $e \in in_D(v)$) and $\rho'(e') = (\emptyset, \{\rho(e)\})$ otherwise. It follows that ρ' assigns $(\{p_1\}, \emptyset), \ldots, (\{p_m\}, \emptyset), (\emptyset, \{q_1\}), \ldots, (\emptyset, \{q_n\})$ to the $m + n$ edges of T_v that target the leaves of T_v, and that the corresponding transitions $\{p_i, (\{p_i\}, \emptyset)\} \xrightarrow{\sigma} \emptyset$ and $\{(\emptyset, \{q_j\})\} \xrightarrow{\sigma} \{q_j\}$ exist in M'. Every other edge e' of T_v points to a σ'-labeled unary or binary node of T_v. If $out_{T_v}(tar_{D_T}(e')) = \{e_1\}$, let $\rho'(e') = \rho'(e_1)$. If $out_{T_v}(tar_{D_T}(e')) = \{e_1, e_2\}$ with $\rho'(e_i) = (I_i, O_i)$ for $i = 1, 2$, we set $\rho'(e') = (I_1 \uplus I_2, O_1 \uplus O_2)$. By items 1–4 in the construction of M', the corresponding transitions are in M', which means that ρ is accepting.

Conversely, suppose that ρ' is a run of M' on D_T and consider one of its treelets T_v whose root is labeled σ'. By the transitions in items 1–4, together with the fact that only transitions of the form $\{p, (\{p\}, \emptyset)\} \xrightarrow{\sigma} \emptyset$ or $\{(\emptyset, \{q\})\} \xrightarrow{\sigma} \{q\}$ apply to the leaves of T_v, it follows that there is a transition $\{p_1, \ldots, p_m\} \xrightarrow{\sigma} \{q_1, \ldots, q_n\}$ in M such that ρ' assigns the states $(\{p_1\}, \emptyset), \ldots, (\{p_m\}, \emptyset), (\emptyset, \{q_1\}), \ldots, (\emptyset, \{q_n\})$ to the $m + n$ edges of T_v that target the leaves of T_v. In turn, this means that $\rho'(in_D(v)) = \{p_1, \ldots, p_m\}$ and $\rho'(out_D(v)) = \{q_1, \ldots, q_n\}$, because the edges in $in_D(v)$ and $out_D(v)$ are those whose targets and sources, respectively, are the leaves of T_v in D_T. Consequently, the restriction of ρ' to E_D is a run of M on D.

This argument yields the following theorem.

Theorem 3
For every DAG $D \in \mathcal{D}_\Sigma$ and every binary tree decomposition of D, M' accepts D_T if and only if M accepts D.

6.4 Computational Analysis

We derive here an upper bound on the running time of the improved version of Algorithm 2. Recall that the binarized automaton M' has state set $Q' = Q \cup Q_{\text{io}}$, where Q is the state set of the source automaton M, and Q_{io} is the set of new states of the form (I, O) added by the binarization construction. We start by deriving an upper bound on $|Q_{\text{io}}|$.

As already discussed, each state $(I, O) \in Q_{\text{io}}$ refers to some transition t of M, with multisets I and O representing instances of states from t that still need to be assigned. Let us focus for now on I, and let us assume that m is the maximum size of the left-hand side of a transition of M. We can represent I by providing a count, for each state $q \in Q$, of the occurrences of q in I. In this way, a number between 0 and m needs to be stored for each q. Because the left-hand side of t contains at most $|Q|$ distinguishable states, the total number of possible values for I is the total number of possible choices *with repetition* of m elements from set Q. This number is usually written as $\left(\!\binom{|Q|}{m}\!\right)$ and amounts to $\binom{|Q|+m-1}{m}$. To simplify our formulas, we bound $\binom{|Q|+m-1}{m}$ from above by $(m+1)^{|Q|}$. We observe that this is not a tight bound, because the worst case of m and $|Q|$ cannot be both realized at the same time. We will discuss a tighter bound later.

Similarly to the case of multiset I, if n is the maximum size of of the right-hand side of a transition of M, we can derive an upper bound of $(n+1)^{|Q|}$ for the total number of possible values for O. Putting everything together we have

$$|Q_{\text{io}}| \leq (m+1)^{|Q|}(n+1)^{|Q|} \tag{4}$$

Let D be some source DAG, and let D' be the binarized DAG obtained from D through our construction in Section 6.2. Recall that Algorithm 2, when run on D, has a time complexity of

$$\mathcal{O}\left(|E_D| \cdot |Q|^{\text{tw}(\mathcal{LG}(D))+1}\right) \tag{5}$$

where $\text{tw}(\mathcal{LG}(D))$ is the treewidth of the line graph $\mathcal{LG}(D)$. From Theorem 2 we know that $\text{tw}(\mathcal{LG}(D')) \leq 2(\text{tw}(D) + 1)$. Combining these facts and our upper bound on $|Q_{\text{io}}|$

we have that, when given as input the DAG D' and the binarized automaton M', Algorithm 2 runs in time

$$\mathcal{O}\left(|E_{D'}|(|Q| + (m+1)^{|Q|}(n+1)^{|Q|})^{2\text{tw}(D)+3}\right) \tag{6}$$

In what follows, we compare the two running times in (5) and (6). We start our analysis by looking into the input DAG automaton M. We have already remarked that $\text{tw}(\mathcal{LG}(D)) + 1$ is at least the degree of nodes of D because every node of degree k in D is turned into a hyperedge of size k that must be covered by some bag. Thus we have

$$\text{tw}(\mathcal{LG}(D)) + 1 \geq m + n$$

Whereas the original algorithm was exponential in the number of edges participating in M's transitions, the binarized algorithm is only polynomial in this number.

We now hold the automaton M fixed, and analyze the running time of the two algorithms in terms of the properties of the input DAG D. In this way, the running time for the original algorithm is $\mathcal{O}\left(|E_D| c_1^{\text{tw}(\mathcal{LG}(D))+1}\right)$, and the running time for the binarized algorithm is $\mathcal{O}\left(|E_{D'}| c_2^{2\text{tw}(D)+3}\right)$, for some constants c_1 and c_2.

We start by comparing quantities $|E_D|$ and $|E_{D'}|$. The binarized DAG D' can be preprocessed to remove any unary nodes in a treelet T_v in linear time. This leaves $2(n-1)$ internal edges in each treelet T_v derived from a vertex v in D having degree n. This implies that $|E_{D'}| < 5|E_D|$, because for each edge $(u, v) \in E_D$, $E_{D'}$ contains a copy of (u, v), two edges internal to T_u, and two edges internal to T_v. Thus,

$$|E_{D'}| = \mathcal{O}\left(|E_D|\right)$$

We are now left with the comparison of the two terms $c_1^{\text{tw}(\mathcal{LG}(D))+1}$ and $c_2^{2\text{tw}(D)+3}$. The binarized algorithm depends on $\text{tw}(D)$ rather than $\text{tw}(\mathcal{LG}(D))$. In general, $\text{tw}(\mathcal{LG}(D))$ may be larger than $\text{tw}(D)$ by an arbitrary amount. To see this, consider a star graph with one central node attached to n leaves. Whereas the treewidth is 1, the treewidth of the line graph is n.

In the other direction, we can derive a lower bound on $\text{tw}(\mathcal{LG}(D))$ in terms of $\text{tw}(D)$ as follows. A tree decomposition of D can be produced from a tree decomposition T of $\mathcal{LG}(D)$ by replacing each node of $\mathcal{LG}(D)$ in T with the corresponding two nodes of D. This leads to the relation

$$\text{tw}(\mathcal{LG}(D)) + 1 \geq \frac{1}{2}(\text{tw}(D) + 1)$$

Thus, although the exponent in the running time of the binarized algorithm may be larger than the exponent in the original algorithm, it must be within a constant factor.

As already observed, our upper bound on $|Q_{\text{io}}|$ is not very tight, because our worst case hypotheses cannot be realized all at the same time. Consider then the maximum number of distinguishable states appearing in the left-hand side or in the right-hand side of a transition of M, which we denote as m_Q. Let also μ be the maximum number of occurrences of an individual state appearing in the left-hand side or in the right-hand side of a transition of M. While we have $m_Q \leq |Q|$ and $\mu \leq \max\{m, n\}$, we cannot have $m_Q = |Q|$ and $\mu = \max\{m, n\}$ both at the same time. Using these quantities, we can rewrite our upper bound on Q_{io} as $(\mu + 1)^{2m_Q}$.

To summarize, the binarized algorithm is particularly beneficial for automata having transitions with large degree and small numbers of states, or else for automata with small values of m_Q and μ. In these cases, the time savings over the unbinarized algorithm can be exponentially large.

7. Extended DAG Automata

In a natural language setting, we may want to model DAGs in which the incoming degree of a node is not bounded by any constant. This is useful in the semantic representation of sentences with coreference relations, in which some concept is shared among several predicates. Similarly, the outgoing degree of our DAGs should not be bounded by a constant, allowing us to add to a given predicate a number of optional modifiers that can grow with the length of the sentence.

As already discussed in Section 3.3, Quernheim and Knight (2012) exploit some ordering on the incoming edges at each node and introduce *implicit rules* that process these edges in several steps, making it possible to accept nodes of unbounded in-degree. This approach allows the incoming edges of a node to have states that form any semilinear set—for example, an equal number of edges in state q and in state r. This does not seem to be motivated in the perspective of natural language semantics.

As an alternative, we propose a milder extension of the DAG automata in Definition 3 that is analogous to the definition of extended CFGs, also called regular right part grammars (LaLonde 1977). In extended CFGs, the right-hand side of each production is a regular expression denoting a set of strings of nonterminal and terminal symbols. Similarly, in our extended DAG automata the left-hand side and the right-hand side of a transition can be a regular expression of a restricted type.

7.1 Regular and Recognizable Languages of Multisets

Let Q be the state set used by some DAG automaton that we do not further specify yet. Subsequently, we view Q as the input alphabet of a device that is used to recognize the collections of incoming or outgoing states of a transition of our DAG automaton. Because these collections are multisets rather than strings, we must first introduce some machinery for the denotation or recognition of regular languages of multisets.

7.1.1 Multisets and Languages of Multisets. Recall from Section 3.1 that $\mathcal{M}(Q)$ denotes the collection of all (finite) multisets over Q. If $\mu_1, \mu_2 \in \mathcal{M}(Q)$, we write $\mu_1 \uplus \mu_2$ for the multiset union of μ_1 and μ_2, which just adds the counts from μ_1 and the counts from μ_2.

A language L of multisets is a subset of $\mathcal{M}(Q)$. If $\langle \mathbb{K}, \oplus, \otimes, 0, 1 \rangle$ is a (commutative) semiring, a $\mathbb{K}$-weighted (or simply weighted) language of multisets additionally assigns a weight from $\mathbb{K}$ to each multiset in the language. More formally, in this case L is a function $L\colon \mathcal{M}(Q) \to \mathbb{K}$ that maps every multiset $\mu \in \mathcal{M}(Q)$ to its weight $L(\mu) \in \mathbb{K}$. The weight $L(\mu) = 0$ indicates that μ is not in the language at all.

The union of two languages $L_1 \cup L_2$ is defined as usual; in the weighted case, if μ is in both languages, its weights are added. The concatenation of two languages is $L_1 L_2 = \{\mu_1 \uplus \mu_2 \mid \mu_1 \in L_1, \mu_2 \in L_2\}$; in the weighted case, $L = L_1 L_2$ is given by

$$L(\mu) = \bigoplus_{\mu = \mu_1 \uplus \mu_2} L_1(\mu_1) \otimes L_2(\mu_2)$$

For a (weighted) language L of multisets and an integer n, we let $L^n = \{\emptyset\}$ if $n = 0$ and $L^n = LL^{n-1}$ if $n > 0$. Finally, the Kleene star is defined as $L^* = \bigcup_{i \geq 0} L^i$.

Define a **unary** language to be a language that only uses one symbol; that is, a language $L \subseteq \mathcal{M}(\{q\})$ for some symbol $q \in Q$. Next, we give two equivalent characterizations of a class of regular (or recognizable) languages of multisets, analogous to regular expressions and finite automata for languages of strings.

7.1.2 C-regular Expressions. The set of **c-regular expressions** α for multisets over alphabet Q (cf. Ochmański 1985) is defined inductively as follows, together with the semantics $[\![\alpha]\!]$ of these expressions:

1. ϵ is a c-regular expression, and $[\![\epsilon]\!] = \{\emptyset\}$.
2. If $q \in Q$, then q is a c-regular expression, and $[\![q]\!] = \{\{q\}\}$.
3. If α_1 and α_2 are c-regular expressions, then $\alpha_1 \cup \alpha_2$ is a c-regular expression, and $[\![\alpha_1 \cup \alpha_2]\!] = [\![\alpha_1]\!] \cup [\![\alpha_2]\!]$.
4. If α_1 and α_2 are c-regular expressions, then $\alpha_1\alpha_2$ is a c-regular expression, and $[\![\alpha_1\alpha_2]\!] = [\![\alpha_1]\!]\,[\![\alpha_2]\!]$.
5. If α is a c-regular expression such that $[\![\alpha]\!]$ is unary, then α^* is a c-regular expression, and $[\![\alpha^*]\!] = [\![\alpha]\!]^*$.
6. No expressions but those which can be constructed according to the previous items are c-regular expressions.

Sometimes we write q^n in place of the c-regular expression $q \cdots q$, where q is repeated n times for some integer n.

To mention some examples, let $q, r \in Q$.

- qr is a c-regular expression and $[\![qr]\!] = \{\{q, r\}\}$.
- $q(qq)^*$ is a c-regular expression and $[\![q(qq)^*]\!]$ is the language of all multisets consisting of an odd number of q's.
- The set $[\![qr]\!]^* = \bigcup_{i \geq 0} [\![qr]\!]^i$ is the language of all multisets with an equal number of q's and r's. We emphasize that $(qr)^*$ is *not* a valid c-regular expression for this language, because the starred subexpression involves mentions of more than one state. It should be clear that the language cannot be expressed by means of a c-regular expression, because such an expression would have to contain at least two starred subexpressions, one containing only qs and one containing only rs, which necessarily allows for multisets containing different numbers of qs and rs.

The definition of c-regular expressions can be extended to weighted c-regular expressions (Droste and Gastin 1999; Allauzen and Mohri 2006). The semantics of a weighted c-regular expression α over Q with weights in $\mathbb{K}$ is then a function $[\![\alpha]\!] : \mathcal{M}(Q) \to \mathbb{K}$. We have already specified how to combine the weights for the union

Table 2
Terminology and notational conventions used for multiset automata and DAG automata.

DAG automaton	Multiset automaton	
state	m-state	
transition	m-transition	
M	A	(automaton)
Q	Ξ	(state set)
Σ	Q	(input alphabet)
Δ	τ	(transitions)

and the concatenation operators. Then it suffices to add the following conditions, which newly introduce weights into a c-regular expression:[6]

1. The weight of $\emptyset$ in $[\![\epsilon]\!] = \{\emptyset\}$ and that of $\{q\}$ in $[\![qr]\!] = \{\{q\}\}$ is 1. In more formal functional notation, $[\![\epsilon]\!](\emptyset) = [\![q]\!](\{q\}) = 1$, and $[\![\epsilon]\!](\mu) = [\![q]\!](\mu') = 0$ for all $\mu \neq \emptyset$ and $\mu' \neq \{q\}$.
2. If α is a (weighted) c-regular expression and $k \in \mathbb{K}$, then $k\alpha$ is a weighted c-regular expression. The weighted language $[\![k\alpha]\!]$ is just $[\![\alpha]\!]$ with all of its weights multiplied by k: $[\![k\alpha]\!](\mu) = k \otimes [\![\alpha]\!](\mu)$ for all $\mu \in \mathcal{M}(Q)$. (When writing regular expressions that are not fully parenthesized, this operation has the same precedence as concatenation.)

We note that expressions of the form $k\alpha$ are not the only ones that create weights other than 1 (depending, of course, on the definition of the operations of the semiring K). For example, $[\![q \cup q]\!]$ assigns the weight $1 \oplus 1$ to $\{q\}$ (and 0 to every other multiset).

Multiset languages generated by c-regular expressions will be called **regular** multiset languages, and similarly for the weighted case.

7.1.3 Multiset Finite Automata. In this section we introduce weighted finite automata that recognize multisets. Later on, the use of finite automata for multisets might create several clashes with our notion of (extended) DAG automata. To avoid this, we introduce in Table 2 our naming and notational conventions used to distinguish between multiset automata and DAG automata. When referring to multiset automata, for instance, we always use the terms m-state and m-transition, whereas the terms state and transition are used for DAG automata. Note also that we are overloading symbol Q, which is used to denote the state set of a DAG automaton as well as the input alphabet of a multiset automaton. As already explained, this is because we use multiset automata to recognize the collections of incoming or outgoing states of a transition of the DAG automaton.

In the following, we assume that a multiset $\mu \in \mathcal{M}(Q)$ is represented as a sequence of all the elements of μ, with repetitions, in any possible order. In this way we can use standard string automata to process multisets. A weighted finite automaton has the same form as a weighted finite automaton for strings (Fülöp and Kuich 2009).

6 Droste and Gastin (1999) talk about weighted *mc*-rational languages, where the *m* stands for an additional constraint needed in their more general case. In our case, c-regular and mc-regular are equivalent.

The difference is that the order in which the elements of the multiset are read by the automaton must not affect the computed weight.

Definition 5
A **weighted finite automaton for multisets**, or **m-automaton** for short, is defined as a tuple $A = (\Xi, Q, \tau, s, \rho)$, where

1. Ξ is a set of m-states,
2. Q is a finite input alphabet,
3. $\tau : \Xi \times Q \times \Xi \to \mathbb{K}$ is the m-transition function, satisfying the following condition: for all m-states $i, k \in \Xi$ and for all alphabet symbols $q, r \in Q$

$$\bigoplus_{j \in \Xi} \tau(i, q, j)\tau(j, r, k) = \bigoplus_{j \in \Xi} \tau(i, r, j)\tau(j, q, k) \tag{7}$$

4. $s \in \Xi$ is the initial m-state, and
5. $\rho : \Xi \to \mathbb{K}$ maps m-states to final weights (where a state $j \in \Xi$ is called final if $\rho(j) \neq 0$).

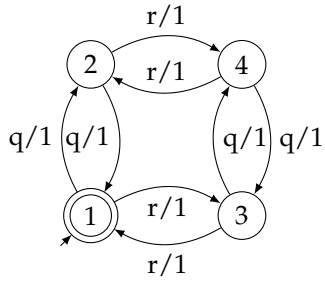
As a notational convention, in what follows we always assume $\Xi = \{1, \ldots, d\}$, for some $d \geq 1$. Condition (7) in Definition 5 states that, when we move from i to j by processing symbols q and r, the resulting total weight does not depend on the order in which q and r are read. It should be clear that Condition (7) is a sufficient condition for the desired property that an m-automaton assigns the same weight to all possible permutations of a given sequence (see below for a precise definition of the weight of a sequence). However, Condition (7) is not a necessary condition for this property; it is not difficult to provide a counter-example to show this; however, we do not further pursue this issue here.

Let us now formally define the semantics $[\![A]\!]$ of an m-automaton as a mapping from multisets to weights in $\mathbb{K}$. Let μ be a multiset over Q with n elements, $n \geq 0$. We arrange the elements of μ into a string $c = q_1 q_2 \cdots q_n$, choosing the order of symbols arbitrarily. Now, we define $[\![A]\!](\mu)$ to be $[\![A]\!](c)$, where $[\![A]\!](c)$ is given as usual for weighted finite automata on strings. To make this explicit, let a **run** of A on c be any sequence $\rho_c = i_0 i_1 \cdots i_n$ of m-states in Ξ such that $i_0 = s$. We extend the m-transition function τ to runs by viewing ρ_c as a sequence of n transitions of A on c, and by taking the product of weights of these transitions and the final weight for i_n:

$$[\![A]\!](\rho_c) = \left(\bigotimes_{h=1}^{n} \tau(i_{h-1}, q_h, i_h) \right) \otimes \rho(i_n)$$

Accordingly, the weight of c under A is defined as $[\![A]\!](c) = \bigoplus_{\rho_c} [\![A]\!](\rho_c)$, where ρ_c ranges over all runs of A on c. Finally, as mentioned, set $[\![A]\!](\mu) = [\![A]\!](c)$. Using Condition (7) in Definition 5, it is not difficult to show that $[\![A]\!](\mu)$ is unique, that is, this quantity does not depend on the specific order of the symbols in μ used to create c.

Example 12
As an example, consider the language represented by the weighted c-regular expression $(qq)^*(rr)^*$. This is the language of all multisets containing an even number of q's and an even number of r's, where each multiset has the weight 1. The corresponding m-automaton A is shown below. Here, an edge from i to j labeled by q/w means that $\tau(i,q,j) = w$, and a missing edge indicates that $\tau(i,q,j) = 0$. The final weight of all the states of A is 0 except for state 1, whose final weight is 1.



It is easy to verify that Condition (7) holds for this m-automaton. Consider for instance the multiset $\mu = \{q,q,r,r\}$. We have that, for any ordering c of the elements of μ, one run relative to c has weight 1 and all remaining runs have weight of 0. We thus have $[\![A]\!](\mu) = 1$.

Multiset (weighted) languages generated by (weighted) m-automata will be called **recognizable** multiset (weighted) languages.

7.1.4 Equivalence of C-regular Expressions and M-automata. The relationship between (restrictions of) regular expressions and finite automata on trace monoids was investigated by Ochmański (1985) and extended by Droste and Gastin (1999) to weighted regular expressions and finite automata. These results, applied to multisets (the simplest example of trace monoids), imply that weighted c-regular expressions and weighted m-automata are equivalent. Because the equivalence proof is much easier for this case, we include it here for completeness. For this, let us make a short detour to recall the technique for turning an ordinary regular expression (i.e., on strings) into a finite automaton originally proposed by McNaughton and Yamada (1960). (For regular expressions and finite automata on strings, we use the same notation as introduced above for the multiset case, only changing their semantics in the obvious way.)

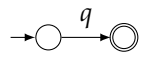
Theorem 4 (McNaughton-Yamada)
Every string-based regular expression α can be converted into an equivalent finite automaton A such that:

1. A has no ϵ transitions, and
2. the initial state of A has no incoming transitions.

Proof. The construction proceeds by induction on the structure of the regular expression:

(a) If $\alpha = \epsilon$ then A consists of a single initial and final state.

(b) If $\alpha = q, q \in Q$, then A consists of the following two states (initial and final, respectively):



(c) If $\alpha = \beta \cup \gamma$, then convert β and γ to automata A_1 and A_2, respectively. Let s_1 and s_2 be the initial states of A_1 and A_2, respectively. Then merge s_1 and s_2 into a single new initial state, which is a final state if either s_1 or s_2 was.

(d) If $\alpha = \beta\gamma$, then convert β and γ to automata A_1 and A_2, respectively. Then for each final state f of A_1 and for each transition $s_2 \xrightarrow{q} j$, where s_2 is the initial state of A_2, add a transition $f \xrightarrow{q} j$. State f continues to be a final state if and only if s_2 was. Then remove s_2.

(e) If $\alpha = \beta^*$, then convert β to an automaton A_1. Then for each final state f of A_1 and for each transition $s_1 \xrightarrow{q} j$, where s_1 is the initial state of A_1, add a transition $f \xrightarrow{q} j$. Finally, add s_1 to the set of final states. □

Still discussing the string case, the preceding theorem can easily be extended to weighted regular expressions and weighted finite automata by augmenting the cases of the construction above as follows:

(a) The unique state of A gets the final weight 1.

(b) The transition of A gets the weight 1 and so does the final state, while the initial state gets the weight 0.

(c) When merging s_1 and s_2 into a single state in the construction of A for $\beta \cup \gamma$, their final weights are summed up. This is correct because these states have no incoming transitions.[7]

(d) Every newly added transition $f \xrightarrow{q} j$ created in A gets the product of the final weight of f in A_1 and the weight of $s_2 \xrightarrow{q} j$ in A_2. The final weight of f is the product of the final weights of f and s_2.

(e) Similarly to the previous case, the weight of $f \xrightarrow{q} j$ is the product of the final weight of f and the weight of $s_1 \xrightarrow{q} j$ in A_1. The final weight of s_1 becomes 1.

(f) Finally, to convert an expression $\alpha = k\beta$ ($k \in \mathbb{K}$), just take the automaton obtained for β and multiply all final weights by k.

Let us now return to the case of c-regular expressions and m-automata. We will need the notion of **size** for (weighted) c-regular expressions and m-automata. The size $|\alpha|$ of a c-regular expression α is the number of occurrences of nullary symbols (ϵs and alphabet symbols) in it. The **size** $|A|$ of an m-automaton A is its number of states.

For the equivalence of weighted c-regular expressions and weighted m-automata, note first that if we restrict attention to unary languages, then there is no relevant

7 Both here and in the remaining items all weights and final weights not explicitly mentioned carry over from A_1 and A_2, respectively.

difference between weighted c-regular expressions and weighted regular expressions on strings, or between weighted m-automata and weighted finite automata on strings. This is because commuting symbols in a string over a unary alphabet does not change anything. Hence weighted c-regular expressions and weighted m-automata are clearly equivalent in the unary case.

In particular, it is possible to convert a unary weighted c-regular expression α to an equivalent weighted m-automaton $A(\alpha)$, using the McNaughton-Yamada construction recalled earlier. By an easy induction following the construction in Theorem 4, the size of $A(\alpha)$ will then be at most $|\alpha| + 1$.

For treating the general case, the property in Theorem 4 that the initial state has no incoming transitions is quite useful. We will call weighted m-automata satisfying this requirement **non-reentrant**.

We will make use of the following lemma.

Lemma 1
Let L_1 and L_2 be multiset languages recognizable by non-reentrant weighted m-automata.

1. $L_1 \cup L_2$ is recognizable by a non-reentrant weighted m-automaton.
2. If L_1, L_2 use disjoint sets of symbols, then $L_1 L_2$ is recognizable by a non-reentrant weighted m-automaton.

Proof. For the first statement, if A_1 and A_2 recognize L_1 and L_2, respectively, just use the McNaughton-Yamada construction for the union operator. The resulting weighted m-automaton satisfies the commutativity requirement (Condition (7)) because each of the individual automata does.

For the second statement, now let $A_1 = (S_1, Q_1, \tau_1, s_1, \rho_1)$ and $A_2 = (S_2, Q_2, \tau_2, s_2, \rho_2)$ recognize L_1 and L_2, respectively, where $Q_1 \cap Q_2 = \emptyset$. Then the shuffle product (Hopcroft and Ullman 1979, p. 142) of A_1 and A_2 is the automaton that simulates A_1 and A_2 together, feeding each input symbol to either machine but not both. More formally, $A = (S_1 \times S_2, Q_1 \cup Q_2, \tau, (s_1, s_2), \rho)$, where:

$$\tau((i_1, i_2), q, (j_1, i_2)) = \tau_1(i_1, q, j_1) \qquad q \in Q_1$$
$$\tau((i_1, i_2), q, (i_1, j_2)) = \tau_2(i_2, q, j_2) \qquad q \in Q_2$$

and $\rho(i_1, i_2) = \rho_1(i_1)\rho_2(i_2)$ for all $(i_1, i_2) \in S_1 \times S_2$. Clearly, A recognizes the multisets of $L_1 L_2$ in any order, and it is non-reentrant if both A_1 and A_2 are. □

Theorem 5
Every weighted c-regular expression α with $|\alpha| \leq n$ can be converted into an equivalent non-reentrant weighted m-automaton $A(\alpha)$ with $|A(\alpha)| \leq 2^n$.

Proof. First, we show that any weighted c-regular expression α can be rewritten in the form

$$\alpha' = \bigcup_{i \in I} \prod_{q \in Q} \alpha_{iq}$$

for a suitable index set I, where each α_{iq} only uses the alphabet $\{q\}$. We show this by induction on the structure of α. Throughout the proof, we write $\alpha \equiv \alpha'$ if $[\![\alpha]\!] = [\![\alpha']\!]$.

If $\alpha = \epsilon$, $\alpha = q$ or $\alpha = \beta^*$, then α is trivially in the desired form. Now, assume as induction hypothesis that β and γ are in the desired form.

- If $\alpha = \beta \cup \gamma$, then

$$\alpha = \bigcup_{i \in I} \prod_{q \in Q} \beta_{iq} \cup \bigcup_{j \in J} \prod_{q \in Q} \gamma_{jq}$$

  which is in the desired form.

- If $\alpha = \beta\gamma$, then we can rewrite α to the desired form because

$$\begin{aligned}
\alpha &= \left(\bigcup_{i \in I} \prod_{q \in Q} \beta_{iq} \right) \left(\bigcup_{j \in J} \prod_{q \in Q} \gamma_{jq} \right) \\
&\equiv \bigcup_{i \in I} \bigcup_{j \in J} \prod_{q \in Q} \beta_{iq} \prod_{q \in Q} \gamma_{jq} && \text{(by distributivity of concatenation over union)} \\
&\equiv \bigcup_{i \in I} \bigcup_{j \in J} \prod_{q \in Q} \beta_{iq}\gamma_{jq} && \text{(by commutativity of concatenation)}
\end{aligned}$$

  which is in the desired form.

- If $\alpha = k\beta$ for a $k \in \mathbb{K}$, choose a state $q_0 \in Q$. Then

$$\begin{aligned}
\alpha &= k \left(\bigcup_{i \in I} \prod_{q \in Q} \beta_{iq} \right) \\
&\equiv \bigcup_{i \in I} k \left(\prod_{q \in Q} \beta_{iq} \right) && \text{(by distributivity of multiplication over addition)} \\
&\equiv \bigcup_{i \in I} (k\beta_{iq_0}) \prod_{q \in Q \setminus \{q_0\}} \beta_{iq}
\end{aligned}$$

Second, we convert an expression in this form to a non-reentrant weighted m-automaton as follows:

- For each α_{iq}, convert it into an automaton $A(\alpha_{iq})$ using the McNaughton-Yamada construction.
- For each i, form the shuffle product of the $A(\alpha_{iq})$, according to the second statement of Lemma 1.
- Finally, use the first statement of Lemma 1 to obtain an automaton $A(\alpha)$.

The size bound $|A(\alpha)| \leq 2^{|\alpha|}$ can be shown by induction on $|\alpha|$.

- If $\alpha = \epsilon$, because we use the McNaughton-Yamada construction, we have $|A(\alpha)| = 1 < 2^{|\alpha|}$.
- If $\alpha = q$, because we use the McNaughton-Yamada construction, we have $|A(\alpha)| = 2 \leq 2^{|\alpha|}$.
- If $\alpha = \beta^*$, then $|A(\alpha)| = |A(\beta)| \leq 2^{|\beta|} = 2^{|\alpha|}$, again by the McNaughton-Yamada construction.
- If $\alpha = k\beta$ for $k \in \mathbb{K}$, then $|A(\alpha)| = |A(\beta)| \leq 2^{|\beta|} = 2^{|\alpha|}$, by our extension of the McNaughton-Yamada construction to the weighted case.
- If $\alpha = \beta \cup \gamma$, then

$$|A(\alpha)| = |A(\beta)| + |A(\gamma)| - 1 < 2^{|\beta|} + 2^{|\gamma|} \leq 2^{|\beta|} \cdot 2^{|\gamma|} = 2^{|\alpha|}$$

- If $\alpha = \beta\gamma$, and $\beta \equiv \bigcup_i \prod_q \beta_{iq}$, and $\gamma \equiv \bigcup_j \prod_q \gamma_{jq}$, then

$$\begin{aligned}
|A(\alpha)| &= \sum_i \sum_j \prod_q (|A(\beta_{iq})| + |A(\gamma_{jq})|) \\
&\leq \sum_i \sum_j \prod_q |A(\beta_{iq})| \cdot |A(\gamma_{jq})| \\
&= \left(\sum_i \prod_q |A(\beta_{iq})| \right) \left(\sum_j \prod_q |A(\gamma_{jq})| \right) \\
&= |A(\beta)| \cdot |A(\gamma)| \\
&\leq 2^{|\beta|} 2^{|\gamma|} = 2^{|\alpha|}
\end{aligned}$$

□

It is also possible to convert a weighted m-automaton to an equivalent weighted c-regular expression. Even though we do not use this result here, we mention it briefly for completeness.

Theorem 6
Any weighted m-automaton can be converted into an equivalent weighted c-regular expression.

Proof. Given a weighted m-automaton A, view it as a weighted *string* automaton and intersect it with an ordinary string automaton recognizing the language $q_1^* \cdots q_{|Q|}^*$. The resulting automaton A' is a weighted string automaton such that

$$[\![A']\!](u) = \begin{cases} [\![A]\!](u) & \text{if } u \in q_1^* \cdots q_{|Q|}^* \\ 0 & \text{otherwise} \end{cases}$$

Now use the standard state elimination algorithm (for the weighted case) to convert A' into a regular expression (which will then necessarily be a weighted c-regular expression). □

7.2 Extended Weighted DAG Automata

In this section we introduce a definition that extends the DAG automata of Section 3. We start with an overview of the basic idea. In our extended DAG automata, the left-hand side and the right-hand side of a transition are weighted c-regular expressions α and β defining (the weights of) acceptable combinations of states at the incoming and at the outgoing edges, respectively, of the node to be processed. These c-regular expressions are therefore defined over the alphabet Q, that is, the state set of the extended DAG automaton.

Note that a transition in an extended DAG automaton has a potentially infinite set of instantiations $\{q_1, \ldots, q_m\} \xrightarrow{\sigma} \{r_1, \ldots, r_n\}$, where a **transition instantiation** is defined as a transition of the DAG automata of Section 3. The weight of such a transition instantiation is defined as the product of the weights assigned by α and β to $\{q_1, \ldots, q_m\}$ and $\{r_1, \ldots, r_n\}$, respectively. Using the definition of run for a DAG D that we introduced in Section 3.1, the weight of a run on D is the product of the weights of all instantiated transitions of the run. In the unweighted case, this means that D is accepted by the extended DAG automaton if and only if there exists an assignment of states to the edges of D such that, for each node v of D with label σ, there is some extended transition $\alpha \xrightarrow{\sigma} \beta$ such that the multiset of states at the incoming edges of v matches α and, similarly, the multiset of states at the outgoing edges matches β.

Definition 6
An **extended weighted DAG automaton** is a tuple $M = (\Sigma, Q, \Delta, \mathbb{K})$, where

1. Σ, Q, and $\mathbb{K}$ are defined as in the case of weighted DAG automata
2. Δ is a transition relation consisting of a finite set of triples of the form $t = \langle \alpha, \sigma, \beta \rangle$, where $\sigma \in \Sigma$ and α, β are $\mathbb{K}$-weighted c-regular expressions over Q. We also write t in the form $\alpha \xrightarrow{\sigma} \beta$ and call it an **extended transition**.

For a precise definition of the semantics of extended DAG automata, recall that a run has been defined in Section 3.1 as an assignment of states of the automaton to the edges of a DAG. Consider a run ρ on a DAG $D = (V, E, lab, src, tar)$. Let $v \in V$ with $lab(v) = \sigma$, and let l and r be the multisets of states on its incoming and outgoing edges, respectively, under ρ. This gives rise to an unweighted transition instance $t = (l \xrightarrow{\sigma} r)$. Every extended transition $\alpha \xrightarrow{\sigma} \beta$ in Δ contributes $[\![\alpha]\!](l) \otimes [\![\beta]\!](r)$ to the weight of t. We denote the resulting weight of t by $w_\rho(v)$, namely,

$$w_\rho(v) = \bigoplus_{(\alpha \xrightarrow{\sigma} \beta) \in \Delta} [\![\alpha]\!](l) \otimes [\![\beta]\!](r)$$

Now, as mentioned before, the weight of ρ is obtained by taking the product of all the $w_\rho(v)$, over all $v \in V$, and the total weight of D is the sum of the weights of all runs on D:

$$[\![M]\!](D) = \bigoplus_{\text{run } \rho \text{ on } D} \bigotimes_{v \in V} w_\rho(v)$$

We now argue that, if the support of $[\![\alpha]\!]$ and $[\![\beta]\!]$ is finite for all transitions $\langle \alpha, \sigma, \beta \rangle$ of an extended DAG automaton, then the automaton is equivalent to an ordinary DAG automaton. (The support of a weighted c-regular expression α over Q is the set of all $\mu \in \mathcal{M}(Q)$ such that $[\![\alpha]\!](\mu) \neq 0$.) To turn a transition $l \xrightarrow{\sigma/w} r$ of an ordinary DAG automaton into an equivalent extended transition $\alpha \xrightarrow{\sigma} \beta$, let α' be a sequence with all the elements of multiset l, in any order, and define $\alpha = w\alpha'$. Furthermore, let β be defined as a sequence with all the elements of multiset r, in any order. In this way we have that, for all multisets $m \in \mathcal{M}(Q)$,

$$[\![\alpha]\!](m) = \begin{cases} w \text{ if } m = l \\ 0 \text{ otherwise} \end{cases} \qquad \text{and} \qquad [\![\beta]\!](m) = \begin{cases} 1 \text{ if } m = r \\ 0 \text{ otherwise.} \end{cases}$$

Conversely, for every extended DAG automaton M there exists a non-extended DAG automaton M' over Σ such that $[\![M']\!](D) = [\![M]\!](D)$ for every DAG D. For this, just define the transition function δ of M' similarly to w_ρ: for every symbol $\sigma \in \Sigma$ and all multisets $l, r \in \mathcal{M}(Q)$,

$$\delta(l, \sigma, r) = \bigoplus_{(\alpha \xrightarrow{\sigma} \beta) \in \Delta} [\![\alpha]\!](l) \otimes [\![\beta]\!](r)$$

Because any c-regular expression appearing in an extended transition of Δ has finite support, function δ is finite, as desired.

Example 13
An extended DAG automaton that models AMR structures with unbounded node degree is specified in Figure 18. The extended transitions are based on the (non-extended) transitions in Figure 7, and make use of the Kleene star operator of c-regular expressions. More specifically, each predicate can take zero or more modifiers, labeled *mod*, allowing sentences such as "John wants Mary to believe Sue," "John wants Mary to believe Sue today," and so forth. Similarly, entities including *John*, *Mary*, and *Sue* can be generated from one or more states labeled q_{person}, allowing an arbitrary number of instances of coreference.

7.3 Properties

We now extend the properties studied for unweighted non-extended DAG automata to the (also unweighted) extended case. Thus, from this point onwards up until the start of Section 7.4, all DAG automata and extended DAG automata are assumed to be unweighted. Consequently, the m-automata $A = (\Xi, Q, \tau, s, \rho)$ appearing in this section are also unweighted. We therefore view the transition function τ as a set of transitions $\langle \xi, q, \xi' \rangle$ rather than as a mapping from all possible such transitions to the domain $\{true, false\}$ of the Boolean semiring.

As a basis for most of the results of this section, we use a binarization approach similar to Section 6, though somewhat simpler because we do not want to optimize parsing and thus do not need to use tree decompositions.

Let $M = (\Sigma, Q, \Delta, \mathbb{K})$ be an extended DAG automaton. We binarize (unranked) DAGs over Σ as follows. Let $\Sigma' = \{\sigma_{m,n} \mid \sigma \in \Sigma\}$ for $m, n \in \{0, 1, 2\}$. The idea

Transitions:

$$\epsilon \xrightarrow{\text{want}} q_{\text{want-arg0}} q_{\text{want-arg1}} q^*_{\text{want-mod}}$$

$$q_{\text{want-arg0}} \xrightarrow{\text{ARG0}} q_{\text{person}}$$

$$q_{\text{want-arg1}} \xrightarrow{\text{ARG1}} q_{\text{pred}}$$

$$q_{\text{pred}} \xrightarrow{\text{believe}} q_{\text{believe-arg0}} q_{\text{believe-arg1}} q^*_{\text{believe-mod}}$$

$$q_{\text{believe-arg0}} \xrightarrow{\text{ARG0}} q_{\text{person}}$$

$$q_{\text{believe-arg1}} \xrightarrow{\text{ARG1}} q_{\text{person}}$$

$$q_{\text{person}} q^*_{\text{person}} \xrightarrow{\text{John}} \epsilon$$

$$q_{\text{person}} q^*_{\text{person}} \xrightarrow{\text{Mary}} \epsilon$$

$$q_{\text{person}} q^*_{\text{person}} \xrightarrow{\text{Sue}} \epsilon$$

Some of the accepted AMRs:

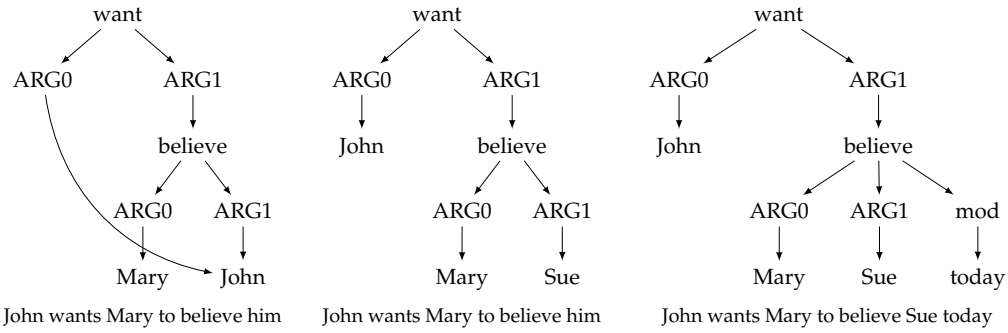


Figure 18
Extended rules for AMRs. The Kleene star in the expressions for input and output state multisets means that a state can occur zero or more times.

underlying the binarization of a DAG D is to replace every σ-labeled node v of in-degree m and out-degree n by a "vertical" chain of $m + n + 3$ copies of σ of the form

$$\sigma_{0,1} \underbrace{\sigma_{2,1} \cdots \sigma_{2,1}}_{m \text{ times}} \sigma_{1,1} \underbrace{\sigma_{1,2} \cdots \sigma_{1,2}}_{n \text{ times}} \sigma_{1,0},$$

where each $\sigma_{2,1}$ is assigned one of the incoming edges of v, and vice versa for $\sigma_{1,2}$ with respect to the outgoing edges of v. Figure 19 shows an example where $(m, n) = (3, 2)$. Note that nodes of in-degree 0 are turned into chains that start with $\sigma_{0,1}\sigma_{1,1}$ at the top. Similarly, nodes of out-degree 0 are turned into chains that end in $\sigma_{1,1}\sigma_{1,0}$ at the bottom. Different orderings of the incoming and outgoing edges yield potentially different binarizations. Thus, every DAG D over Σ gives rise to a finite set $B(D)$ of binarized DAGs over Σ'. It is now straightforward to turn M into a non-extended DAG automaton M' such that $[\![M']\!] = \bigcup_{D \in [\![M]\!]} B(D)$. For this, note that the sub-DAGs of the form shown in Figure 19 contain $m + n + 2$ additional edges—those on the vertical spine—and $m + n$ edges stemming from the original DAG. In a run of M', the latter are assigned states

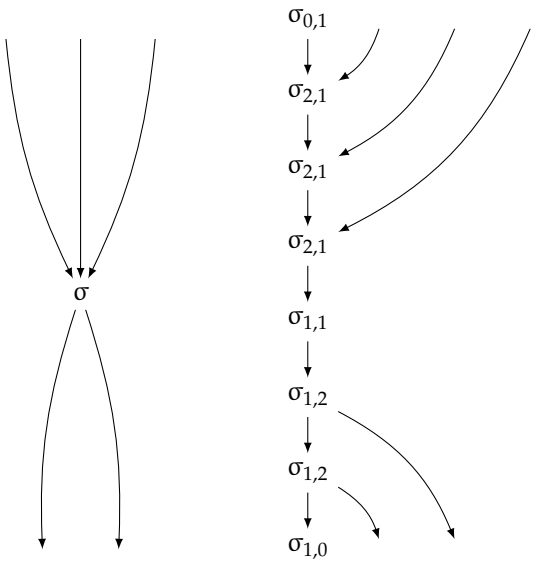


Figure 19
A node of in-degree 3 and out-degree 2, and the corresponding sub-DAG created by binarization.

in Q, whereas the former are assigned states of the m-automata that implement the transitions of M.

Consider such a transition $\langle \alpha, \sigma, \beta \rangle$ and let $A = (\Xi, Q, \tau, s, \rho)$ and $A' = (\Xi', Q, \tau', s', \rho')$ with $\Xi \cap \Xi' = \emptyset$ be m-automata equivalent to α and β, respectively. Then M' contains the transitions

- $\emptyset \xrightarrow{\sigma_{0,1}} s$ (this assigns the initial state of A to the top-most edge of the spine),
- $\{\xi, q\} \xrightarrow{\sigma_{2,1}} \{\xi'\}$ for all $\xi \in \Xi$, $q \in Q$, and $\langle \xi, q, \xi' \rangle \in \tau$ (this "reads" q on an incoming edge in state ξ, assigning the resulting state ξ' to the next edge on the spine),
- $\{\xi\} \xrightarrow{\sigma_{1,1}} \{s'\}$ for all final states ξ of A (this allows A' to "cross" the middle of the spine and continue to work on the outgoing edges) and, similarly to the preceding items,
- $\{\xi\} \xrightarrow{\sigma_{1,2}} \{\xi', q\}$ for all $\xi \in \Xi'$, $q \in Q$, and $\langle \xi, q, \xi' \rangle \in \tau'$ (similarly to the second item, but "reading" q on an outgoing edge) and
- $\{\xi\} \xrightarrow{\sigma_{1,0}} \emptyset$ for all final states ξ of A' (similarly to the first item).

It should be clear that, indeed, $[\![M']\!] = \bigcup_{D \in [\![M]\!]} B(D)$. As a consequence, we can extend three results from non-extended DAG automata to extended ones: The emptiness and finiteness problems are decidable and the path languages are regular. (The decidability of the finiteness problem was shown in Blum and Drewes [2016] for the non-extended case.)

These results are summarized in the next theorem.

Theorem 7
For unweighted extended DAG automata

1. the emptiness problem is decidable,
2. the finiteness problem is decidable, and
3. the path language is regular.

If the transitions of the input automata of the emptiness and finiteness problems are specified by means of m-automata (rather than c-regular expressions), then the decision algorithms run in polynomial time.

Proof. The first two statements follow directly from the fact that a DAG language is empty (finite) if and only if its binarized counterpart is empty (finite, respectively). Furthermore, if the m-automata that specify the transitions of M are given, then the DAG automaton M' discussed earlier can be obtained from M in polynomial time.

To see that the path languages are regular, consider some $D \in [\![M]\!]$ and $D' \in B(D)$. Intuitively, every path in D is represented by one in D' such that, whenever the original path passes a node, the corresponding path in D' enters the chain in Figure 19 through one of the edges coming from the right and leaves it through one of the outgoing edges going to the right. However, in D' a path can start (and end) at any such chain, since each contains a root (and a leaf), even if the node it represents is an internal node of D. Fortunately, the desired paths can easily be singled out, because a chain represents a root if it starts with $\sigma_{0,1}\sigma_{1,1}$ and a leaf if it ends with $\sigma_{1,1}\sigma_{1,0}$.

This amounts to saying that a string $w \in \Sigma^*$ is a path in D if and only if there is a path w' in D' that satisfies the following:

(a) w' has a prefix of the form $\sigma_{0,1}\sigma_{1,1}$,

(b) w' has a suffix of the form $\sigma_{1,1}\sigma_{1,0}$, and

(c) w is obtained from w' by applying the homomorphism that replaces each $\sigma_{1,1}$ by σ and erases all other symbols.

Thus, because the path language of $[\![M']\!]$ is regular, so is that of $[\![M]\!]$, because it is obtained from the former by intersection with two regular languages and applying a homomorphism. □

Let us now consider the intersection of a hyperedge replacement language with $[\![M]\!]$. Assume for simplicity that the given HRG G is "normalized" in the sense that every right-hand side either contains no terminal edges at all, or consists only of the nodes in the left-hand side and terminal edges. We sketch briefly how G can be turned into a HRG G' that generates $[\![G]\!] \cap [\![M]\!]$, hoping that the interested reader will be able to work out the details by herself.

Similarly to Section 4.2, the idea is to use a Bar-Hillel-like construction. However, the construction of Section 4.2 has to be generalized slightly because now the degree of nodes in the graphs in $[\![M]\!]$ is not a priori bounded anymore.

Recall that in the previous case we annotated each tentacle of a nonterminal hyperedge with two multisets of states. Intuitively, if v is the node the tentacle points to, then this annotation "guesses" the states on incoming and outgoing edges that this nonterminal will attach to v. In the extended version, suppose the label of v is σ and

there is a transition $\langle \alpha, \sigma, \beta \rangle$, where $A = (\Xi, Q, \tau, s, \rho)$ and $A' = (\Xi', Q, \tau', s', \rho')$ are m-automata that implement α and β, respectively. Then the annotation of the tentacle will consist of two pairs of states, $((\xi_1, \xi_2), (\xi'_1, \xi'_2)) \in \Xi^2 \times \Xi'^2$, representing the "guess" that the derivation of this nonterminal hyperedge will eventually attach incoming and outgoing edges to the node that can be assigned states from Q which take A and A' from ξ_1 to ξ_2 and from ξ'_1 to ξ'_2, respectively.

To see how this can be done, consider first a nonterminal rule of the original HRG, and assume that it replaces the nonterminal hyperedge in such a way that, in the right-hand side of the rule, two new nonterminal hyperedges have tentacles to the corresponding node. Then the resulting HRG will contain a version of the rule in which these tentacles carry annotations $((\xi_1, \xi), (\xi'_1, \xi'))$ and $((\xi, \xi_2), (\xi', \xi'_2))$, for all possible choices of $\xi \in \Xi$ and $\xi' \in \Xi'$. Similarly, if the right-hand side contains a node that is not in the left-hand side, and that node is attached to, say, a single tentacle, then this tentacle would be annotated with some $((s, \xi), (s', \xi'))$ such that ξ and ξ' are final states of A and A', respectively.

Finally, the terminal rules verify the consistency of the nondeterministic guesses. Suppose the original HRG contains a terminal rule $L ::= R$ for the nonterminal in question. For each annotated version L' of L, the modified HRG contains the rule $L' ::= R$ if there exists an assignment of states in Q to the edges in R that is consistent with the annotation of (tentacles in) L'. For example, if the annotation of one of the tentacles is $((\xi_1, \xi_2), (\xi'_1, \xi'_2))$ and the incoming and outgoing edges of the corresponding node in R are assigned the multisets of states Q_{in} and Q_{out}, then it must be the case that Q_{in} takes A from ξ_1 to ξ_2 and Q_{out} takes A' from ξ'_1 to ξ'_2.

As mentioned, we leave the details of the construction to the reader. The resulting HRG generates the language $[\![G]\!] \cap [\![M]\!]$, thus showing that the class of hyperedge replacement languages is closed under intersection with extended DAG automata.

7.4 Recognition

In this section we present two parsing algorithms for extended DAG automata. The first algorithm is a reduction to the recognition problem of Section 5 for (non-extended) DAG automata, and demonstrates the close relationship between these two problems. The reduction is based on the binarization method presented in Section 6, and involves constructing a binarized DAG D' on the basis of a tree decomposition of the input graph D. The second algorithm we present operates directly on this tree decomposition, and can be implemented without using Algorithm 2 as a subroutine.

7.4.1 Reduction to Non-Extended Recognition. Let M be an extended DAG automaton, and let D be an input DAG. Informally, our reduction consists of the following steps:

- encode D into a binary DAG D';
- transform M into a binary, non-extended DAG automaton M';
- run M' on D' using Algorithm 2.

The binarization of D is done on the basis of a tree decomposition of D, using the techniques presented in Section 6. The automaton M' is also constructed using techniques similar to those presented in Section 6, as described in detail here.

Let Q be the state set of M and let $Q_A = Q \cup Q'$, where $Q' = \{q' \mid q \in Q\}$ is a set of fresh copies of the states in Q. We compile all the transitions on an input symbol $\sigma \in \Sigma$ into a single m-automaton A_σ over Q_A, using the states in Q and Q' to distinguish between incoming and outgoing edges. Suppose that $\langle \alpha_1, \sigma, \beta_1 \rangle, \ldots, \langle \alpha_n, \sigma, \beta_n \rangle$ are the transitions of M on σ. Let β'_ℓ, $1 \leq \ell \leq n$, be obtained from β_ℓ by replacing each $q \in Q$ with its copy $q' \in Q'$. Now, let $A_\sigma = (\Xi_\sigma, Q_A, \tau_\sigma, s_\sigma, \rho_\sigma)$ be an m-automaton such that $[\![A_\sigma]\!] = [\![\bigcup_{\ell=1}^{n} (\alpha_\ell \beta'_\ell)]\!]$.

Recall from Section 6 that all edges of D are copied into D' and that these copied edges are all attached to the leaf nodes of the treelets in D'. The remaining edges of D', that is, those edges that are newly added in the binarization of D, are called D'-auxiliary.

States in M' are symbols in Q or else pairs of states from the m-automata A_σ. States $q \in Q$ are used by M' at edges copied from D. Pairs of states from A_σ are used by M' at the D'-auxiliary edges. More specifically, consider a node v of D with $lab(v) = \sigma$, and the corresponding treelet T_v in D'. Let e be some D'-auxiliary edge in T_v with target node u, and let $\mathcal{E}$ be the set of all edges copied from D that are attached to the leaves of the sub-treelet of T_v rooted at node u; see Figure 20. A pair (i, j) with $i, j \in \Xi_\sigma$ is used by M' at e to indicate that A_σ can process the multiset of symbols from Q_A assigned to the edges from $\mathcal{E}$ by starting in state i and ending in state j.

We now specify in detail the transitions of M', which correspond one-to-one to the rules defined in Section 6.3. Let $\sigma \in \Sigma$ be an input symbol of M. The first two rules here apply at the root of a treelet T_v derived from a vertex v of D labeled with σ, and stipulate initial and final states of some computation in A_σ:

$$\forall i \in \Xi_\sigma \qquad \emptyset \xrightarrow{\sigma'/\rho_\sigma(i)} \{(s_\sigma, i)\} \tag{8}$$

$$\forall i, j \in \Xi_\sigma \qquad \emptyset \xrightarrow{\sigma'/\rho_\sigma(j)} \{(s_\sigma, i), (i, j)\} \tag{9}$$

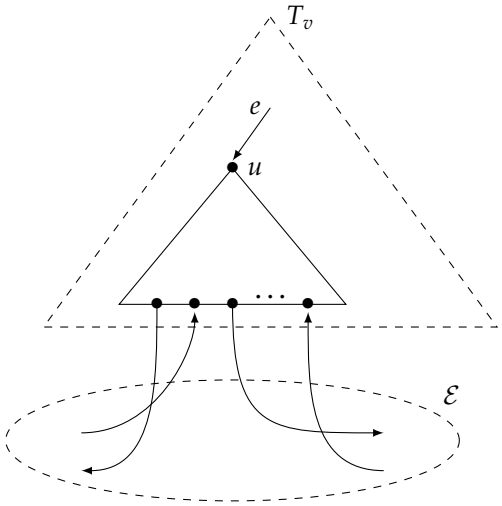


Figure 20
Set of edges $\mathcal{E}$ copied from D and attached to the leaves of the sub-tree of treelet T_v rooted at node u.

The next two rules apply at nodes that are internal to a treelet T_v. The unary rule does nothing, simply skipping the node, and the binary rule concatenates two sub-computations in A_σ:

$$\forall i,j \in \Xi_\sigma \qquad \{(i,j)\} \xrightarrow{\sigma'/1} \{(i,j)\} \tag{10}$$

$$\forall i,j,k \in \Xi_\sigma \qquad \{(i,k)\} \xrightarrow{\sigma'/1} \{(i,j),(j,k)\} \tag{11}$$

Finally, we need transitions to process leaf nodes at the treelets in D', where we need to simulate transitions of A_σ that process edges copied from the original DAG D. Recall that these copied edges are labeled with states in Q, whereas in A_σ states from Q' are indicative of outgoing edges. Thus, we use the following transitions:

$$\forall q \in Q,\ i,j \in \Xi_\sigma \qquad \{(i,j),q\} \xrightarrow{\sigma/\tau_\sigma(i,q,j)} \emptyset \tag{12}$$

$$\forall q \in Q,\ i,j \in \Xi_\sigma \qquad \{(i,j)\} \xrightarrow{\sigma/\tau_\sigma(i,q',j)} \{q\} \tag{13}$$

The following transitions handle the special case of a treelet consisting of a single node:

$$\forall q \in Q,\ i \in \Xi_\sigma \qquad \{q\} \xrightarrow{\sigma/\tau_\sigma(s_\sigma,q,i)\rho_\sigma(i)} \emptyset \tag{14}$$

$$\forall q \in Q,\ i \in \Xi_\sigma \qquad \emptyset \xrightarrow{\sigma/\tau_\sigma(s_\sigma,q',i)\rho_\sigma(i)} \{q\} \tag{15}$$

Once M' has been constructed from M, we can process each input DAG D by converting it into a binary DAG D' and then by running M' on D' using Algorithm 2.

Computational Analysis. Recall from Section 5 that Algorithm 2, when run on a non-extended DAG automaton M and a ranked DAG D, has a time complexity of

$$\mathcal{O}\left(|E_D| \cdot |Q|^{\mathrm{tw}(\mathcal{LG}(D))+1}\right)$$

where Q is the state set of M and $\mathrm{tw}(\mathcal{LG}(D))$ is the treewidth of the line graph of D.

An upper bound on the number of states of the binarized non-extended automaton M' just defined is $|Q| + m^2|\Sigma|$, where $m = \max_{\sigma\in\Sigma} |\Xi_\sigma|$. This is because states are either states of M or otherwise pairs of states of one of the m-automata A_σ. Furthermore, by Theorem 2, $\mathrm{tw}(\mathcal{LG}(D')) \leq 2(\mathrm{tw}(D)+1)$. The binarization construction ensures that $|E_{D'}|$ is $\mathcal{O}(|E_D|)$. Combining these facts, we have that the running time of Algorithm 2 on the binarized DAG D' and the binarized automaton M' is

$$\mathcal{O}\left(|E_D|(|Q| + m^2|\Sigma|)^{2\mathrm{tw}(D)+3}\right) \tag{16}$$

Thus, as with the algorithm of Section 6 for non-extended DAGs, the running time is exponential in the treewidth of the input DAG, linear in the total size of the input DAG, and, for a fixed treewidth, polynomial in the number of states of the extended automaton and in the number of states of the largest m-automaton A_σ for the transitions on a single input symbol σ.

7.4.2 Direct Recognition. We now present an alternative algorithm for processing D according to the extended automaton M. The algorithm uses the same ideas as before,

but works directly on D and M, without any preprocessing (binarization). Thus the alternative algorithm avoids the overhead of compiling (or computing on-the-fly) the large number of (non-extended) rules defined in the previous subsection.

Let T be a tree decomposition of DAG D. Recall from Definition 2 that a node b of T is called a bag, and $cont(b)$, the label of b, is a set of nodes of D. In what follows, we assume our tree decompositions are in a canonical form that has been introduced by Cygan et al. (2011). Tree decomposition T is **nice** if every bag b of T satisfies one of the following conditions.

- b has no children and $cont(b) = \emptyset$.
- b has one child b_1, and $b = b_1 \cup \{v\}$ for some node $v \notin cont(b_1)$. In this case, b is said to **introduce** v.
- b has one child b_1, and $b \cup \{v\} = b_1$ for some node $v \notin cont(b)$. In this case, b is said to **forget** v.
- b has one child b_1 with $b = b_1$, and b is additionally labeled with an edge e such that $src(e), tar(e) \in cont(b)$. In this case b is said to **introduce** e. For every edge e, exactly one bag introduces e.
- b has two children b_1 and b_2, and $cont(b) = cont(b_1) = cont(b_2)$. In this case b is called a **join** bag.

It can be shown from the procedure of Cygan et al. (2011) for constructing nice tree decompositions that the number of bags in a nice tree decomposition of D is $\mathcal{O}(|E_D|)$.

Similarly to Section 7.4.1, we compile all transitions of M on an input symbol $\sigma \in \Sigma$ into a single m-automaton $A_\sigma = (\Xi_\sigma, Q_A, \tau_\sigma, s_\sigma, \rho_\sigma)$. Let $\Xi = \bigcup_{\sigma \in \Sigma} \Xi_\sigma$. In what follows, given a bag b of T, we denote by $\Phi(b)$ the set of all functions $\phi\colon cont(b) \to \Xi \times \Xi$ such that, if $v \in cont(b)$ with $lab(v) = \sigma$, then $\phi(v) \in \Xi_\sigma \times \Xi_\sigma$. In words, function ϕ assigns a pair of states from A_σ to each node v of D in a bag b, where σ is the label of v. Similarly to Section 7.4.1, the intended meaning of a pair $\phi(v) = (i, j)$ is as follows. Let v be some node in $cont(b)$ and let T' be the subtree of T rooted at b. Let also $\mathcal{E}$ be the set of all edges of D that are introduced by bags from T'. When processing the edges in $\mathcal{E}$ that are attached to v, A_σ can begin in state i and end in state j. For compactness, we use the notation $v \mapsto ij$ for the map $\phi\colon \{v\} \to \{(i, j)\}$ such that $\phi(v) = (i, j)$. Let $\phi \in \Phi(b)$ and consider a node v of D (which may or may not be in $cont(b)$). We then write $v \mapsto ij, \phi$ to denote the function $\phi'\colon cont(b) \cup \{v\} \to \Xi \times \Xi$ such that $\phi'(u) = \phi(u)$ for every $u \in cont(b) \setminus \{v\}$, and $\phi'(v) = (i, j)$.

The recognition algorithm (Algorithm 3) processes the input DAG D by visiting its edges in the order in which they appear in a bottom–up walk through the tree decomposition T, computing a partial analysis of M for D. It uses function ϕ to group into equivalence classes all partial analyses that share the same assignment of pairs of states of the appropriate A_σ to the nodes in $cont(b)$, and it uses dynamic programming to compute the overall weight of the computations in the same equivalence class.

The algorithm maintains a chart with entries $\mathsf{chart}_b[\phi] \in \mathbb{K}$, for each b and for each $\phi \in \Phi(b)$. Thus, if m is again the size of the largest Ξ_σ, chart_b has at most $m^{2|cont(b)|}$ entries and could be thought of as an order-$2|cont(b)|$ tensor. Each entry $\mathsf{chart}_b[\phi]$ is the total weight of derivations of the processed part of the graph where, if $v \in cont(b)$ and $\phi(v) = (i, j)$, the m-automaton processing the incident edges of v starts in state i and stops in state j.

Algorithm 3 Direct DAG recognition based on extended automata.

for each bag b, bottom-up **do**
  if b is a leaf **then**
    $\mathsf{chart}_b[\emptyset] = 1$
  else if b introduces node v with $lab(v) = \sigma$ **then**
    for $i \in \Xi_\sigma$ and $\phi \in \Phi(b_1)$ **do**
      $\mathsf{chart}_b[v \mapsto ii, \phi] = \mathsf{chart}_{b_1}[\phi]$
  else if b introduces edge e pointing from v to v', $lab(v) = \sigma$ and $lab(v') = \sigma'$ **then**
    for $i, k \in \Xi_\sigma$, $i', k' \in \Xi_{\sigma'}$ and $\phi \in \Phi(b)$ **do**
      $\mathsf{chart}_b[v \mapsto ik, v' \mapsto i'k', \phi]$
        $= \bigoplus_q \bigoplus_{j \in \Xi_\sigma, j' \in \Xi_{\sigma'}} \mathsf{chart}_{b_1}[v \mapsto ij, v' \mapsto i'j', \phi] \otimes \tau_\sigma(j, q, k) \otimes \tau_{\sigma'}(j', q', k')$
  else if b forgets node v with $lab(v) = \sigma$ **then**
    for $\phi \in \Phi(b)$ **do**
      $\mathsf{chart}_b[\phi] = \bigoplus_j \mathsf{chart}_{b_1}[v \mapsto s_\sigma j, \phi] \otimes \rho_\sigma(j)$
  else if b is a join bag with $cont(b) = \{v_1, \ldots, v_\ell\}$ and $lab(v_p) = \sigma_p$ $(1 \leq p \leq \ell)$ **then**
    for $i_1, k_1 \in \Xi_{\sigma_1}, \ldots, i_\ell, k_\ell \in \Xi_{\sigma_\ell}$ **do**
      $\mathsf{chart}_b[v_1 \mapsto i_1k_1, \ldots, v_\ell \mapsto i_\ell k_\ell]$
        $= \bigoplus_{j_p \in \Xi_{\sigma_p}} \mathsf{chart}_{b_1}[v_1 \mapsto i_1j_1, \ldots, v_\ell \mapsto i_\ell j_\ell] \otimes \mathsf{chart}_{b_2}[v_1 \mapsto j_1k_1, \ldots, v_\ell \mapsto j_\ell k_\ell]$

Computational Analysis. The processing of a join bag in the algorithm takes time $m^{3(\mathrm{tw}(D)+1)}$ because it iterates over triples of states i_h, j_h, k_h for each of the w nodes in the join bag, where w can be as large as $\mathrm{tw}(D) + 1$. The processing of a bag that introduces an edge involves iterating over $m^{2(\mathrm{tw}(D)-1)}$ values of ϕ, m^6 values of i, i', j, j', k, and k', and $|Q|$ values of q, for a total time of $|Q|m^{2(\mathrm{tw}(D)+2)}$. The other types of bags result in strictly lower complexities, giving a total running time of:

$$\mathcal{O}\left(|E_D|(|Q|m^{2(\mathrm{tw}(D)+2)} + m^{3(\mathrm{tw}(D)+1)})\right) \tag{17}$$

This bound is slightly tighter than Equation (16), though similar qualitatively: The running time is exponential in the treewidth of the input DAG, linear in the total size of the input DAG, and polynomial in the number of states of the extended automaton and the transition automata.

8. Conclusion

We have aimed to develop a formalism for DAG automata that lends itself to efficient algorithms for processing semantic graphs such as Abstract Meaning Representations. In particular, motivated by the success of finite-state methods in natural language processing, we have tried to develop a graph analog of standard finite-state automata for strings. The resulting formalism, despite having a straightforward and intuitive definition, differs from previously developed formalisms including those of Kamimura and Slutzki (1981), Charatonik (1999), Priese (2007), and Quernheim and Knight (2012). We have shown that our choice of definitions allows a number of desirable properties

to carry over from finite-state automata for strings, including the regularity of path languages, the polynomial decidability of emptiness and finiteness, and the ability to intersect with hyperedge replacement grammars, which can be viewed as a graph analog of context-free grammars.

However, recognition in general for our formalism remains an NP-complete problem, a major difference from finite-state automata for strings. Motivated by the need for practical algorithms, we study the complexity of this problem in detail. Whereas most previous theoretical work on graph automata deals with general complexity classes such as decidability or NP-completeness, we develop more specific asymptotic complexity results with respect to a number of parameters of the input problem. Our binarization technique allows recognition in time exponential in the treewidth of the input graph. This is a major improvement over the naïve strategy, which is exponential in the treewidth of the *line graph* of the input graph, which itself is at least the degree of the input graph. For semantic representation from the AMR Bank, the maximum treewidth is 4, and the maximum degree is 17. This indicates that the binarization technique is essential to making recognition practical.

Finally, we show how to extend our formalism to DAGs of unbounded degree, which is necessary for handling natural language phenomena such as coreference and optional modifiers. We show that our algorithms and complexity results apply essentially unchanged in this extended setting.

Real-world systems based on our formalism will have to address a number of problems not touched upon in this article, including determining the appropriate set of states and node labels for a particular application. Another avenue for future work is the possibility of rules that process a larger fragment of the input DAG in one transition, as with "extended" rules for tree automata (Maletti et al. 2009). Finally, while we have studied recognition with DAG automata, the development of formalisms for transducers between DAGs and either strings, trees, DAGs, or even general graphs, remains an important area for future work.

Appendix A. Binary Tree Decomposition

We provide an explicit proof for the fact, mentioned in Section 6.2, that tree decompositions can efficiently be transformed into binary tree decompositions of the same width.

Theorem 8
Every tree decomposition of a graph G without isolated nodes can in linear time be transformed into a binary tree decomposition of G of the same width and of size linear in the number of edges of G.

Proof. Let T be a tree decomposition of G of width k. As shown by Kloks (1994, Lemma 2.2.5), it may be assumed without loss of generality that the size of T is at most the number of nodes of G. If a bag b has children $b_1, \ldots, b_k$ with $k > 2$, add a new bag b', let b_1 and b' be the children of b, and $b_2, \ldots, b_k$ those of b'. Define $cont(b') = cont(b) \cap (cont(b_2) \cup \cdots \cup cont(b_k))$. Clearly, the resulting tree T' is still a tree decomposition of width k. Repeating this step will eventually result in a tree decomposition in which every bag has at most two children.

Next, assign to every edge e of G a unique bag $b(e)$ such that $\{src(r), tar(e)\} \subseteq cont(b(e))$. Any leaf b such that $b \neq b(e)$ for all edges e can be removed from the tree, because the nodes in $cont(b)$ are not isolated and are thus contained in other bags. Doing

this repeatedly yields a tree decomposition which is a binary tree such that every leaf is of the form $b(e)$ for one or more edges e (but not all $b(e)$ need to be leaves).

Finally, for every bag b such that there are (pairwise distinct) edges $e_1, \ldots, e_\ell$ with $b(e_1) = \cdots = b(e_\ell) = b$, add a comb whose spine consists of $\ell - 1$ bags with the same contents as b, and whose leaves are bags $b_1, \ldots, b_\ell$ with $cont(b_i) = \{src(e_i), tar(e_i)\}$. Now define $edg(b_i) = e_i$ for $i = 1, \ldots, \ell$. Obviously, the width of the tree decomposition stays the same, and now the mapping $b \mapsto edg(b)$ is a bijection between the leaves of the tree decomposition and the edges of G. We also have that $cont(b) = \{src(edg(b)), tar(edg(b))\}$ for every bag b which is a leaf, as required. To see that the size of the resulting tree decomposition is linear in the number of edges of G, it suffices to notice that the first step doubles the size of T in the worst case, the second step reduces its size, and the third step adds at most two bags for each edge of G. This completes the proof. □

Acknowledgments

We are grateful to the anonymous reviewers for their useful suggestions and to Sorcha Gilroy and Parker Riley for comments on drafts of this article. The authors were funded in part by NSF grant IIA-0530118 PIRE to the Fred Jelinek Memorial Workshop; by ARO grant W911NF-10-1-0533; by the LINDAT/CLARIN project of the Ministry of Education, Youth and Sports of the Czech Republic under projects LM2010013 and LM2015071; by NSF grant IIS-1349902; and by the Italian Ministry of Education, Universities and Research (MIUR) under project PRIN No. 2010LYA9RH_006.

References

Aalbersberg, IJsbrand J., Grzegorz Rozenberg, and Andrzej Ehrenfeucht. 1986. On the membership problem for regular DNLC grammars. *Discrete Applied Mathematics*, 13:79–85.

Abend, Omri and Ari Rappoport. 2013. Universal conceptual cognitive annotation (UCCA). In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 228–238, Sofia.

Abney, Steven, David McAllester, and Fernando Pereira. 1999. Relating probabilistic grammars and automata. In *Proceedings of the 37th Annual Conference of the Association for Computational Linguistics (ACL-99)*, pages 542–549, College Park, MD.

Aho, Alfred V. and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation and Compiling*, volume 1. Prentice-Hall, Englewood Cliffs, NJ.

Allauzen, Cyril and Mehryar Mohri. 2006. A unified construction of the Glushkov, follow, and Antimirov automata. In *Proceedings of Mathematical Foundations of Computer Science 2006*, pages 110–124, Stará Lesná.

Anantharaman, Siva, Paliath Narendran, and Michael Rusinowitch. 2005. Closure properties and decision problems of DAG automata. *Information Processing Letters*, 94(5):231–240.

Arnborg, Stefen, Derek G. Corneil, and Andrzej Proskurowski. 1987. Complexity of finding embeddings in a k-tree. *SIAM Journal of Algebraic and Discrete Methods*, 8:277–284.

Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the Linguistic Annotation Workshop*, pages 178–186, Sofia.

Bar-Hillel, Yehoshua, Micha Perles, and Eli Shamir. 1961. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14(2):143–172.

Bauderon, Michel and Bruno Courcelle. 1987. Graph expressions and graph rewriting. *Mathematical Systems Theory*, 20:83–127.

Baum, Leonard E. 1972. An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes. In *Inequalities III: Proceedings of the Third Symposium on Inequalities*, pages 1–8, Los Angeles, CA.

Björklund, Henrik, Frank Drewes, and Petter Ericson. 2016. Between a rock and a hard place—uniform parsing for hyperedge replacement DAG grammars. In *Proceedings of the 10th International Conference on Language and Automata Theory and Applications*, volume 9618 of *Lecture*

Notes in Computer Science, pages 521–532, Prague.

Blum, Johannes. 2015. DAG automata—variants, languages and properties. Master's thesis, Umeå University.

Blum, Johannes and Frank Drewes. 2016. Properties of regular DAG languages. In *Proceedings of the 10th International Conference on Language and Automata Theory and Applications*, volume 9618 of *Lecture Notes in Computer Science*, pages 427–438, Prague.

Böhmová, Alena, Jan Hajič, Eva Hajičová, and Barbora Hladká. 2003. The Prague Dependency Treebank: A three-level annotation scenario. In A. Abeillé, editor, *Treebanks: Building and Using Parsed Corpora*. Kluwer, pages 103–127.

Bossut, Francis, Max Dauchet, and Bruno Warin. 1988. Automata and rational expressions on planar graphs. In *Mathematical Foundations of Computer Science 1988*, pages 190–200, Carlsbad.

Bossut, Francis, Max Dauchet, and Bruno Warin. 1995. A Kleene theorem for a class of planar acyclic graphs. *Information and Computation*, 117(2):251–265.

Boyd, Stephen and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press.

Brown, Peter F., Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. 1991. Word-sense disambiguation using statistical methods. In *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics (ACL-91)*, pages 264–270, Berkeley, CA.

Charatonik, Witold. 1999. Automata on DAG representations of finite trees. Technical report MPI-I-1999-2-001, Max Planck Institute for Informatics, Saarbrücken, Germany.

Charniak, Eugene. 1997. Statistical parsing with a context-free grammar and word statistics. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97*, pages 598–603, Providence, RI.

Chi, Zhiyi. 1999. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25:131–160.

Chiang, David. 2012. Hope and fear for discriminative training of statistical translation models. *Journal of Machine Learning Research*, 13(1):1159–1187.

Chiang, David, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. 2013. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL-13)*, pages 924–932, Sofia.

Collins, Michael. 1997. Three generative, lexicalised models for statistical parsing. In *Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics*, pages 16–23, Madrid.

Collins, Michael and James Brooks. 1995. Prepositional phrase attachment through a backed-off model. In *Proceedings of the Third Workshop on Very Large Corpora*, pages 27–38, Cambridge, MA.

Comon, Hubert, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. 2002. *Tree Automata Techniques and Applications*. Internet publication available at `http://www.grappa.univ-lille3.fr/tata`.

Courcelle, Bruno. 1990. The monadic second–order logic of graphs. I. Recognizable sets of finite graphs. *Information and Computation*, 85:12–75.

Cygan, Marek, Jesper Nederlof, Marcin Pilipczuk, Michał Pilipczuk, Johan M. M. van Rooij, and Jakub Onufry Wojtaszczyk. 2011. Solving connectivity problems parameterized by treewidth in single exponential time. ArXiv:1103.0534.

Drewes, Frank, Hans-Jörg Kreowski, and Annegret Habel. 1997. Hyperedge replacement graph grammars. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, pages 95–162.

Drewes, Frank and Jérôme Leroux. 2015. Structurally cyclic Petri nets. *Logical Methods in Computer Science*, 11:1–9.

Droste, Manfred and Stefan Dück. 2015. Weighted automata and logics on graphs. In *Proceedings of the 40th International Symposium on Mathematical Foundations of Computer Science (MFCS 2015), Part I*, volume 9234 of *Lecture Notes in Computer Science*, pages 192–204, Milan.

Droste, Manfred and Paul Gastin. 1999. The Kleene-Schützenberger theorem for formal power series in partially commuting variables. *Information and Computation*, 153:47–80.

Eisner, Jason. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the 40th Annual Conference of the Association for Computational Linguistics (ACL-02)*, pages 1–8, Philadelphia, PA.

Esparza, Javier and Mogens Nielsen. 1994. Decidability issues for Petri nets—a survey. *Elektronische Informationsverarbeitung und Kybernetik*, 30:143–160.

Feige, Uriel, MohammadTaghi Hajiaghayi, and James R. Lee. 2005. Improved approximation algorithms for minimum-weight vertex separators. In *STOC '05: Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing*, pages 563–572, Baltimore, MD.

Flanigan, Jeffrey, Chris Dyer, Noah A. Smith, and Jaime Carbonell. 2016. Generation from abstract meaning representation using tree transducers. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 731–739, San Diego, CA.

Flanigan, Jeffrey, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A. Smith. 2014. A discriminative graph-based parser for the abstract meaning representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL-14)*, pages 1426–1436, Baltimore, MD.

Frank, Ove and David Strauss. 1986. Markov graphs. *Journal of the American Statistical Association*, 81(395):832–842.

Frick, Markus, Martin Grohe, and Christoph Koch. 2003. Query evaluation on compressed trees (extended abstract). In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS 2003)*, pages 188–197, Ottawa.

Fujiyoshi, Akio. 2010. Recognition of directed acyclic graphs by spanning tree automata. *Theoretical Computer Science*, 411(38–39):3493–3506.

Fülöp, Zoltán and Werner Kuich. 2009. Weighted tree automata and tree transducers. In Werner Kuich, Manfred Droste, and Heiko Vogler, editors, *Handbook of Weighted Automata*. Springer, chapter 3, pages 69–104.

Gildea, Daniel and Daniel Jurafsky. 2000. Automatic labeling of semantic roles. In *Proceedings of the 38th Annual Conference of the Association for Computational Linguistics (ACL-00)*, pages 512–520, Hong Kong.

Gogate, Vibhav and Rina Dechter. 2004. A complete anytime algorithm for treewidth. In *Uncertainty in Artificial Intelligence (UAI)*, pages 201–208, Banff.

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

Habel, Annegret. 1992. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer.

Habel, Annegret and Hans-Jörg Kreowski. 1987. May we introduce to you: Hyperedge replacement. In *Proceedings of the Third International Workshop on Graph Grammars and Their Application to Computer Science*, volume of 291 *Lecture Notes in Computer Science*. pages 15–26, Warrenton, VA

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

Hovy, Eduard, Mitchell Marcus, Martha Palmer, Lance Ramshaw, and Ralph Weischedel. 2006. Ontonotes: The 90% solution. In *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*, pages 57–60, New York, NY.

Jensen, Finn V., Steffen L. Lauritzen, and Kristian G. Olesen. 1990. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quarterly*, 4:269–282.

Johnson, Mark, Stuart Geman, Stephen Canon, Zhiyi Chi, and Stefan Riezler. 1999. Estimators for stochastic "unification-based" grammars. In *Proceedings of the 37th Annual Conference of the Association for Computational Linguistics (ACL-99)*, pages 535–541, College Park, MD.

Kamimura, Tsutomu and Giora Slutzki. 1981. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49:10–51.

Kaminski, Michael and Shlomit S. Pinter. 1992. Finite automata on directed graphs. *Journal of Computer and System Sciences*, 44:425–446.

Kaplan, Ronald M. and Joan Bresnan. 1982. Lexical-Functional Grammar: A formal system for grammatical representation. In Joan Bresnan, editor, *The Mental Representation of Grammatical Relations*. MIT Press, Cambridge, MA, pages 173–281.

Kloks, Ton. 1994. *Treewidth. Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer.

Lafferty, John, Andrew McCallum, and Fernando Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Machine Learning: Proceedings of the Eighteenth International Conference (ICML 2001)*, pages 282–289, Stanford, CA.

LaLonde, Wilf R. 1977. Regular right part grammars and their parsers. *Communications of the Association for*

Computing Machinery, 20(10): 731–741.
Lange, Klaus Jörn and Emo Welzl. 1987. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. *Discrete Applied Mathematics*, 16:17–30.
Lari, Kamran and Steve J. Young. 1990. The estimation of stochastic context-free grammars using the Inside-Outside algorithm. *Computer Speech and Language*, 4:35–56.
Lautemann, Clemens. 1990. The complexity of graph languages generated by hyperedge replacement. *Acta Informatica*, 27:399–421.
Li, Xiang, Thien Huu Nguyen, Kai Cao, and Ralph Grishman. 2015. Improving event detection with abstract meaning representation. In *Proceedings of the First Workshop on Computing News Storylines*, pages 11–15, Beijing.
Liu, Fei, Jeffrey Flanigan, Sam Thomson, Norman Sadeh, and Noah A. Smith. 2015. Toward abstractive summarization using semantic representations. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1077–1086, Denver, CO.
Lohrey, Markus and Sebastian Maneth. 2006. The complexity of tree automata and XPath on grammar-compressed trees. *Theoretical Computer Science*, 363:196–210.
Maletti, Andreas, Jonathan Graehl, Mark Hopkins, and Kevin Knight. 2009. The power of extended top-down tree transducers. *SIAM Journal on Computing*, 39(2):410–430.
Marcus, Mitchell P., Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.
May, Jonathan. 2016. Semeval-2016 task 8: Meaning representation parsing. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 1063–1073, San Diego, CA.
McNaughton, Robert and Hisao Yamada. 1960. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, EC-9:39–47.
Mooney, Raymond J. 2007. Learning for semantic parsing. In *Computational Linguistics and Intelligent Text Processing: Proceedings of the 8th International Conference (CICLing 2007)*, pages 311–324, Mexico City.
Ochmański, Edward. 1985. Regular behaviour of concurrent systems. *Bulletin of the European Association for Theoretical Computer Science*, 27:56–67.
Oepen, Stephan, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Silvie Cinkova, Dan Flickinger, Jan Hajic, and Zdenka Uresova. 2015. Semeval 2015 Task 18: Broad-coverage semantic dependency parsing. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 915–926, Denver, CO.
Oepen, Stephan, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Dan Flickinger, Jan Hajic, Angelina Ivanova, and Yi Zhang. 2014. Semeval 2014 task 8: Broad-coverage semantic dependency parsing. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*, pages 63–72, Dublin.
Oepen, Stephan and Jan Tore Lønning. 2006. Discriminant-based MRS banking. In *International Conference on Language Resources and Evaluation (LREC)*, pages 1250–1255, Genoa.
Pan, Xiaoman, Taylor Cassidy, Ulf Hermjakob, Heng Ji, and Kevin Knight. 2015. Unsupervised entity linking with abstract meaning representation. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1130–1139, Denver, CO.
Peng, Xiaochang, Linfeng Song, and Daniel Gildea. 2015. A synchronous hyperedge replacement grammar based approach for AMR parsing. In *Proceedings of the Nineteenth Conference on Computational Natural Language Learning*, pages 32–41, Beijing.
Petrov, Slav, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 433–440, Sydney.
Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago.
Potthoff, Andreas, Sebastian Seibert, and Wolfgang Thomas. 1994. Nondeterminism versus determinism of finite automata over directed acyclic graphs. *Bulletin of the Belgian Mathematical Society – Simon Stevin*, 1:285–298.
Priese, Lutz. 2007. Finite automata on unranked and unordered DAGs.

In *Proceedings of the 11th International Conference on Developments in Language Theory (DLT 2007)*, volume 4588 of *Lecture Notes in Computer Science*. pages 346–360, Turku.

Quattoni, Ariadna, Michael Collins, and Trevor Darrell. 2004. Conditional random fields for object recognition. In *Advances in Neural Information Processing Systems (NIPS-17)*, pages 1097–1104, Vancouver.

Quernheim, Daniel and Kevin Knight. 2012. Towards probabilistic acceptors and transducers for feature structures. In *Proceedings of the Sixth Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST)*, pages 76–85, Jeju Island.

Ramshaw, Lance and Mitch Marcus. 1995. Text chunking using transformation-based learning. In *Proceedings of the Third Workshop on Very Large Corpora*, pages 82–94, Cambridge, MA.

Ratnaparkhi, Adwait. 1996. A maximum entropy model for part-of-speech tagging. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 133–142, Philadelphia, PA.

Reutenauer, Christophe. 1990. *The Mathematics of Petri Nets*. Prentice-Hall.

Rose, Donald J. 1970. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597–609.

Salvati, Sylvain. 2014. MIX is a 2-MCFL and the word problem in $\mathbb{Z}^2$ is solved by a third-order collapsible pushdown automaton. *Journal of Computer and System Sciences*, 81:1252–1277.

Shafer, Glenn R. and Prakash P. Shenoy. 1990. Probability propagation. *Annals of Mathematics and Artificial Intelligence*, 2:327–353.

Shieber, Stuart M. 1986. *An Introduction to Unification-Based Approaches to Grammar*. CSLI Publications, Stanford, CA.

Smith, Noah A. and Mark Johnson. 2007. Weighted and probabilistic context-free grammars are equally expressive. *Computational Linguistics*, 33(4):477–491.

Snijders, Tom A. B. 2002. Markov chain Monte Carlo estimation of exponential random graph models. *Journal of Social Structure*, 3(2):1–40.

Soon, Wee Meng, Hwee Tou Ng, and Daniel Chung Long Lim. 2001. A machine learning approach to coreference resolution of noun phrases. *Computational Linguistics*, 27(4):521–544.

Thomas, Wolfgang. 1991. On logics, tilings, and automata. In *Proceedings of the 18th International Colloquium on Automata, Languages and Programming*, pages 441–454, Madrid.

Thomas, Wolfgang. 1996. Elements of an automata theory over partial orders. In *Proceedings of the DIMACS Workshop on Partial Order Methods in Verification*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. pages 25–40, Princeton, NJ.

Vijay-Shanker, K., David J. Weir, and Aravind K. Joshi. 1987. Characterizing structural descriptions produced by various grammatical formalisms. In *Proceedings of the 25th Annual Conference of the Association for Computational Linguistics (ACL-87)*, pages 104–111, Stanford, CA.

Wang, Chuan, Nianwen Xue, and Sameer Pradhan. 2015a. Boosting transition-based AMR parsing with refined actions and auxiliary analyzers. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 857–862, Beijing.

Wang, Chuan, Nianwen Xue, and Sameer Pradhan. 2015b. A transition-based algorithm for AMR parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 366–375, Denver, CO.