DAG Automata for Meaning Representation

Frank Drewes
Umeå University, Sweden
drewes@cs.umu.se

Abstract

Languages of directed acyclic graphs (DAGs) are of interest in Natural Language Processing because they can be used to capture the structure of semantic graphs like those of Abstract Meaning Representation. This paper gives an overview of recent results on a family of automata recognizing such DAG languages.

1 Introduction

This paper attempts to survey, motivate, and explain recent work on automata that recognize sets of directed acyclic graphs (DAGs). These automata are called DAG automata and the languages they recognize regular DAG languages. While DAG automata are of interest in various areas of computer science, different application areas place different requirements on what constitutes a good model of such automata. Here we are interested in DAG automata that are suitable for capturing the structure of semantic graphs in Natural Language Processing, and in particular meaning representations such as Abstract Meaning Representation (AMR).

AMR was introduced by Banarescu et al. (2013) as a domain-independent graph notation for the semantics of meanings in natural language, and since then a quickly growing AMR bank for English has been built.[1] The purpose of such graphbanks is to enable research towards domain-independent semantic language processing, thus mirroring the advances in syntactic processing that have been made during the past 20 years thanks to the existence of large syntactic treebanks.

AMR serves a similar purpose in the realm of semantic representation as the well-known constituent tree does for syntactic representation. For the latter, we have a great variety of well-studied formal models for capturing the structure of correct representations, distinguishing them from faulty ones, and efficiently processing these formal objects. One of the simplest and at the same time most useful models is the finite-state tree automaton or, equivalently, the regular tree grammar (see Gécseg and Steinby (1984, 1997) and Drewes (2006, Appendix A)). Simplicity, though resulting in limited expressive power, is an is an asset in this context. It is to their simplicity that finite-state tree automata owe their usefulness. One of the advantages of the model is that it can easily be extended by weights (Fülöp and Vogler, 2009), then associating with every tree a value that indicates how "good" or perhaps probable the tree is. Such weighted automata are especially useful in Natural Language Processing where one rarely finds a clear dividing line between correct and wrong representations, and where one furthermore has to find ways to resolve ambiguities.

An AMR[2] is usually not a tree but a DAG such as the (somewhat simplified and abstracted) AMR in Figure 1. Its vertices are mostly PropBank concepts (Kingsbury and Palmer, 2002) connected by edges which are labelled by role labels, intuitively supplying the concepts with their semantic arguments. Readers familiar with dependency trees probably notice the similarity, but also the difference: if several concepts share a semantic argument the latter is still represented only once, being pointed to by several edges. This is what turns AMRs into DAGs, thus calling for formal models that allow to specify DAG languages or even weighted DAG languages. Unsurprisingly, it turns

[1] See http://amr.isi.edu.

[2] We use the abbreviation AMR to refer to the general concept of Abstract Meaning Representation as defined by Banarescu et al. (2013), but also to refer to individual graphs that follow the AMR specification.

Proceedings of the 15th Meeting on the Mathematics of Language, pages 88–99,
London, UK, July 13–14, 2017.

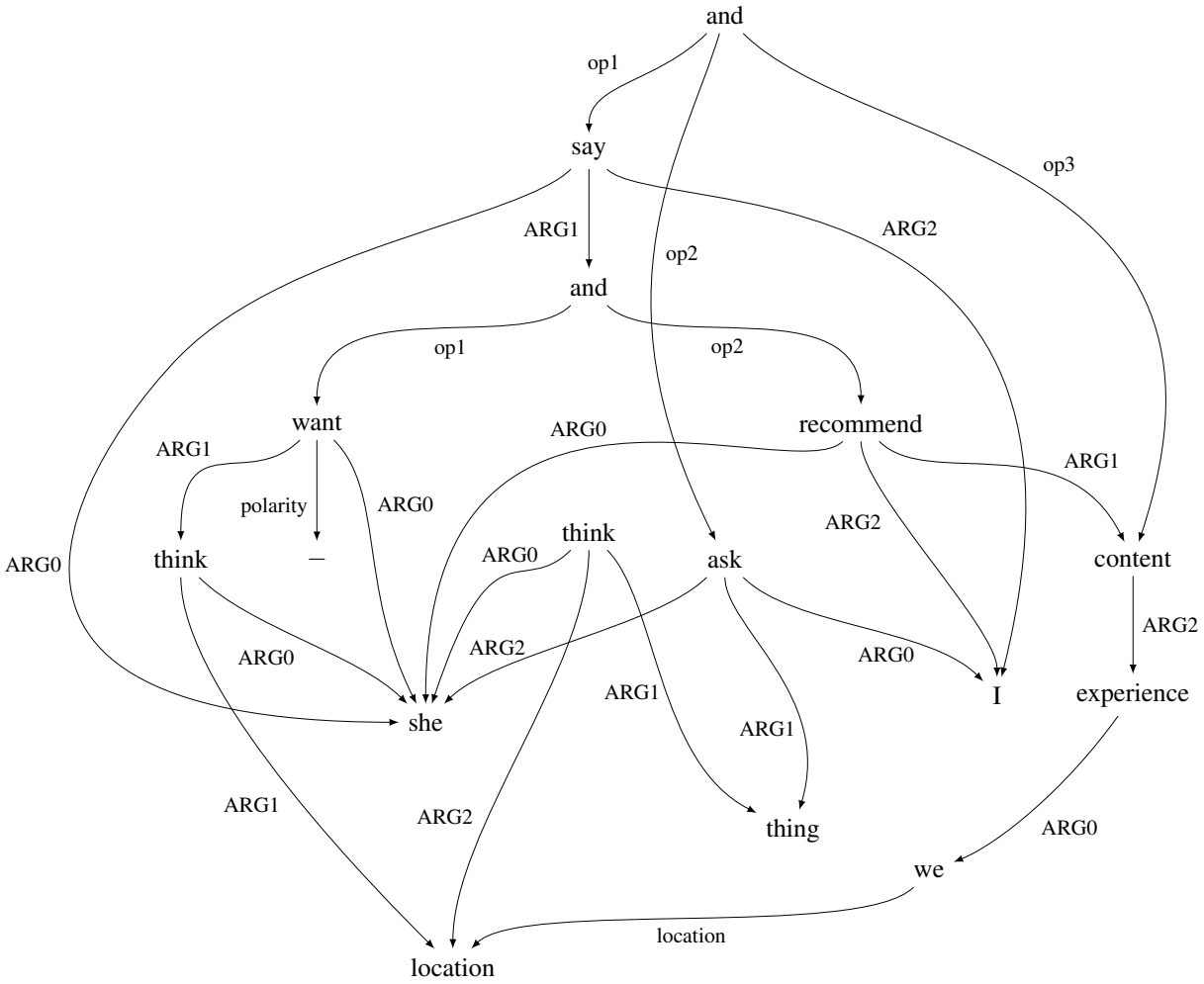

Figure 1: An AMR from the AMR Bank: "I asked her what she thought about where we'd be and she said she doesn't want to think about that, and that I should be happy about the experiences we've had (which I am)." The PropBank frame names (the vertex labels) have been simplified for the sake of readability. They can be found in Chiang et al. (2016), which also uses this example. As a side note, one may note that the AMR does not specify whether her saying was the answer to my asking, the other way around, or the two were independent. This, however, would contribute to a discussion about the limits of AMR rather than about formal automata models for AMR.

out that the greater complexity and expressivity of DAGs makes it all too easy to develop natural types of DAG automata that are surprisingly powerful, thus lacking the simplicity that is the great advantage of finite-state tree automata.

The amount of research that has been devoted to DAG automata models that extend finite-state tree automata to the realm of DAGs is rather limited. Moreover, researchers have come up with various *different* models that, owing to different intended areas of application, work on different types of DAGs and exhibit different properties. Most appear to be more powerful than what appears to be reasonable from a linguistic point of view, thus making the model more complex than desirable. One of the potential problems that comes with greater complexity is computational inefficiency, another one is that such models, when being trained, are prone to overfitting.

2 What Is a "Good" Automaton Model for Meaning Representation?

Let us have a look at a few aspects that come to mind when thinking about an appropriate automaton model for meaning representation such as AMR.

2.1 Types of DAGs

A typical meaning representation, such as the one in Figure 1 carries labels on both vertices and

edges. However, as an edge labelled a may easily be replaced by a sequence of two unlabelled edges with a vertex labelled a in between, one may prefer to view edge labels as "syntactic sugar" which, in the name of simplicity, can be removed from the core formalism.

As in the case of tree languages, DAG languages may be of bounded or unbounded vertex degree (usually called *ranked* and *unranked* in the context of tree languages). In principle, there is no bound on the number of incoming or outgoing edges of vertices in meaning representations because an arbitrary number of optional modifiers may be attached to a concept, and there may be any number of references to a vertex (cf. the 'she' vertex in Figure 1). Thus, the vertex degree is only bounded by the length of the sentence represented, and even this is only true as long as representations of single sentences are considered. In the future, one may very well want to consider AMRs that represent an entire text. However, it is certainly meaningful to study DAG languages of bounded degree as an important base case, especially if the two can be linked in some way. Such a link will be discussed in Section 7.

Another question is whether there should be an order on the incoming and outgoing edges of a vertex. Standard edge labels such as `arg0`, `arg1`, etc seem to indicate an order on outgoing edges. However, there can also be other labels, such as `polarity` and `location` in Figure 1, and incoming edges are not naturally equipped with an order in the first place. This indicates that meaning representations should be viewed as unordered DAGs. On the other hand, as an order only adds expressiveness without affecting the formal properties of the resulting model very much, one may wish to study both possibilities, especially as long as DAG languages of bounded degree are considered. One reason why the ordered case is of interest is that it can more readily be related to the case of regular tree languages (which consist of ordered trees). Another one is that the notion of determinism is rather meaningless in the unordered case.

One may consider DAGs with multiple roots or DAGs which are required to have a unique root.[3] As discussed by Chiang et al. (2016) AMRs should preferably be viewed as multiply-rooted DAGs because their standard representation turns them into singly-rooted graphs only by introducing cycles. Moreover, ensuring single-rootedness often requires the addition of a topmost 'and' vertex. This may become awkward later on if several related sentences shall be viewed as a collection of statements to be represented in a single DAG. As will be discussed below, there are also good formal reasons for not imposing the single-root requirement (and not providing DAG automata with this ability either).

In the literature one also finds DAG automata that work on planar DAGs obeying additional structural conditions (Kamimura and Slutzki, 1981), and finite-state tree automata applied to trees with maximal sharing, meaning that isomorphic subtrees are represented only once and can thus be processed only once even in the nondeterministic case (Charatonik, 1999; Anantharaman et al., 2005). Meaning representations are often non-planar once they get sufficiently complex, and may contain isomorphic substructures that represent distinct instances of otherwise identical concepts and relations.

Several other types of DAG automata are briefly discussed by Chiang et al. (2016). Together with (Quernheim and Knight, 2012; Blum and Drewes, 2016, 2017; Drewes, 2017), the latter represents the line of research discussed in this paper. Despite a rather different formal presentation, its DAG automata are closely related to those by Priese (2007), except for the fact that the latter includes a concept of initial and final states, and can thus in particular restrict the number of roots of accepted DAGs. The importance of this distinction will be discussed in Section 4.

2.2 Expressive Power vs Algorithmic and Language Theoretic Properties

As the expressive power of formal models increases, their algorithmic and language theoretic properties become less favourable. As mentioned initially, for meaning representations expressive power appears to be less important than simplicity and good computational properties. In practice, (weighted) DAG automata will have to be learned from and trained on large semantic graphbanks. For this to be feasible, DAG automata must be implemented as efficiently as possible, it must be possible to check their behaviour, and good closure properties may be required as well.

What makes it reasonable to look for a comparatively weak type of DAG automaton is that mean-

[3] We call a vertex a root if it has no incoming edges.

ing representations, viewed as DAG languages, seem to be relatively simple. One aspect that can be used as an indicator of sufficient simplicity is the complexity of the path languages of recognizable DAG languages. Given a DAG language (with labels only on the vertices, say), its path language consists of all strings obtained by reading the symbols on root-to-leaf paths in the DAGs of the language. It seems linguistically reasonable to assume that, for DAG languages formalizing meaning representations such as AMR, these path languages are regular. Hence, DAG automata that give rise to non-regular path languages are suspiciously powerful from the point of view of meaning representation and should be avoided in favour of simpler ones.

3 DAG Languages of Bounded Degree

Let us now give a formal definitions of DAGs and, afterwards, DAG automata on DAGs of bounded vertex degree.

3.1 DAGs

Let Σ be a (finite) alphabet of vertex labels. A *directed graph* over Σ is a tuple $G = (V, E, lab, src, tar)$ consisting of

- finite sets V and E of *vertices* and *edges*,
- mappings $src, tar\colon E \to V$ associating with each edge $e \in E$ a *source* $src(e)$ and a *target* $tar(e)$, and
- a mapping $lab\colon V \to \Sigma$ that assigns a label $lab(v)$ to every vertex $v \in V$.

For every vertex v we let $IN(v) = tar^{-1}(v)$ and $OUT(v) = src^{-1}(v)$ denote its sets of incoming and outgoing edges.

A path from u to v is an alternating sequence $v_0 e_1 \cdots v_{k-1} e_k v_k$ of vertices and edges such that $v_0 = u$, $v_k = v$, and $\{v_{i-1}, v_i\} = \{src(e_i), tar(e_i)\}$ for all $i \in \{1, \ldots, k\}$. The path is empty if $k = 0$, directed if $v_{i-1} = src(e_i)$ for all $i \in \{1, \ldots, k\}$, and a (simple) cycle if $v_1, \ldots, v_k$ are pairwise distinct and $u = v$. G is acyclic, a *DAG* for short, if it does not contain any nonempty directed cycle.

3.2 DAG Automata

Given a finite set Q, let us denote the set of all finite multisets over Q by $\mathcal{M}(Q)$. In other words, $\mathcal{M}(Q)$ is the set of all functions $M\colon Q \to \mathbb{N}$. A *DAG automaton* is a triple $A = (\Sigma, Q, R)$ consisting of

- an alphabet Σ,
- a finite set Q of states, and
- a finite set R of rules $I \xrightarrow{\sigma} O$, where $I, O \in \mathcal{M}(Q)$ and $\sigma \in \Sigma$.

A *run* of A on a DAG $D = (V, E, lab, src, tar)$ is a mapping $\rho\colon E \to Q$. For a set $E' = \{e_1, \ldots, e_n\}$ of edges in E, we let $\rho(E')$ denote the multiset $\{\rho(e_1), \ldots, \rho(e_n)\}$. (Formally, $\rho(E')(q) = |\{i \in \{1, \ldots, n\} \mid \rho(e_i) = q\}|$ is the number of times q occurs in $\rho(e_1), \ldots, \rho(e_n)$.) The run ρ is *accepting* if it is locally consistent with the rules in R, i.e., if

$$\rho(IN(v)) \xrightarrow{lab(v)} \rho(OUT(v))$$

is in R for every vertex $v \in V$. Naturally, a DAG D is *accepted* by A if there exists an accepting run of A on D. The *DAG language accepted by* A is the set $L(A)$ of all connected and nonempty DAGs accepted by A. Following Blum and Drewes (2017) we shall in the following call DAG languages of the form $L(A)$ *regular DAG languages*.

The fact that we restrict $L(A)$ to connected and nonempty DAGs deserves a brief reflexion. By the definition of acceptance, a DAG is accepted by A if and only if each of its connected components is accepted individually. Hence the set of all accepted DAGs is uniquely determined by $L(A)$. In contrast to $L(A)$ it is never empty (it always contains the empty DAG, which is the disjoint union of zero DAGs in $L(A)$), and it is finite if and only if $L(A)$ is empty. This shows that $L(A)$ is a much more meaningful object of study. In particular, with this definition of $L(A)$ it becomes sensible to ask whether emptiness and finiteness are decidable properties.

3.3 Variants

As discussed above, one may sometimes prefer to work with ordered DAGs instead of the unordered variant defined above. In that case, a DAG is conveniently defined to be a tuple (V, E, IN, OUT, lab), where $IN(v)$ and $OUT(v)$ are sequences of edges, and the components I and O of a rule $I \xrightarrow{\sigma} O$ are sequences rather than multisets of states. Similarly to the

definition above, a run ρ would be accepting if $\rho(IN(v)) \xrightarrow{lab(v)} \rho(OUT(v))$ is in R for every vertex $v \in V$, with ρ being extended to a function from sequences of edges to sequences of states in the canonical way.

For a semiring $\mathbb{S}$, a *weighted DAG automaton* with weights in $\mathbb{S}$ is obtained by turning R into a function that assigns a weight in $\mathbb{S}$ to every potential rule in such a way that all but a finite number of rules are assigned the weight zero. This works in both the ordered and the unordered case. The weight of a run on a DAG D is then the product of the weights of the rules applied at the individual vertices, and the weight of D is the sum of the weights of all runs on D. As usual, unweighted DAG automata are obtained as a special case by choosing the Boolean semiring as $\mathbb{S}$ and representing the set R of rules by its characteristic function.

3.4 Example

As a simple but instructive example, let $\Sigma = \{a, b, \diamond\}$ and $Q = \{p, p', q, q'\}$ with the following rules (where ε denotes the empty sequence):

$$\varepsilon \xrightarrow{a} pp',\ p \xrightarrow{a} pp',$$
$$p' \xrightarrow{\diamond} q'$$
$$pq' \xrightarrow{b} q,\ qq' \xrightarrow{b} q,$$
$$pq' \xrightarrow{b} \varepsilon,\ qq' \xrightarrow{b} \varepsilon.$$

A run on one of the DAGs accepted by this DAG automaton is shown in Figure 2. Note that the second outgoing edge of every a is assigned the state p', which then becomes a q' by passing through $\diamond$, and every b requires an incoming q'. It follows that all accepted DAGs have equal numbers of as and bs. In the DAG of Figure 2 this makes sure that the path not containing any $\diamond$ (i.e., the one obtained by intersection with the regular language a^*b^*) is of the form $a^n b^n$.

4 What a Difference a Root Makes

The preceding example seems to show that the DAG automata discussed here are more powerful than intended, as the path language of $L(A)$ appears to be non-regular. In fact, path languages even seem to exceed the context-free languages as it is easy to add further letters in the same way, thus obtaining paths like $a^n b^n c^n d^n$. However, this is true only if $L(A)$ is restricted to DAGs with a unique root. As there is no way for A to ensure this, we can take a second accepted DAG, add it

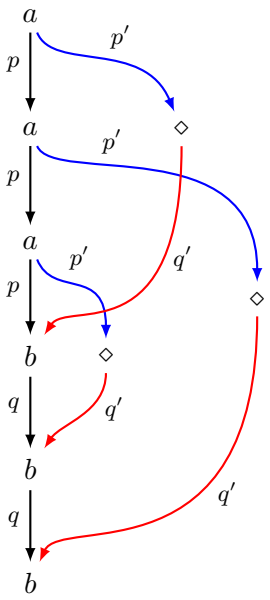

Figure 2: A run of the DAG automaton in Section 3.4; for better visual clarity edges carrying states p' and q' are drawn in blue and red, respectively.

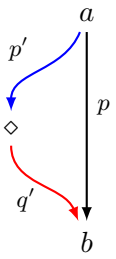

Figure 3: Another run of the DAG automaton in Section 3.4

disjointly to the one in Figure 2, and then connect the two by swapping the targets of two edges with the same state. This swapping operation turns out to be a powerful tool for proofs as it, by the definition of accepting runs, preserves acceptance. For example, by using the DAG in Figure 3 as the second component on can construct the accepting run in Figure 4 on a DAG that contains the paths $a^3 b$ and ab^3.

Thus, if we let $L_{\mathrm{u}}(A)$ denote the set of all DAGs in $L(A)$ having exactly one root, then the path language of $L_{\mathrm{u}}(A)$ may indeed be non-context-free. In contrast, it can be shown that the path language of $L(A)$ is regular for every DAG automaton A. In fact, we can *unfold* a DAG $D = (V, E, lab, src, tar)$ into a set of trees over Σ, as follows. For a vertex $v \in V$ with $lab(v) = \sigma$

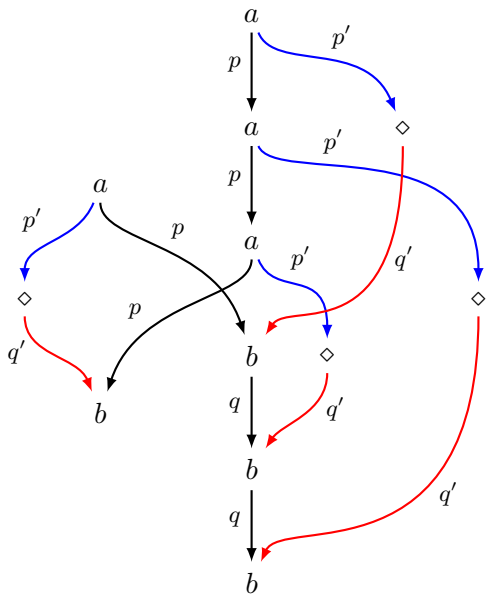

Figure 4: A possible combination of the runs in Figures 2 and 3

and $OUT(v) = \{e_1, \ldots, e_k\}$, let $unfold_D(v)$ be the tree whose root is labelled σ and whose direct subtrees are the trees $unfold_D(tar(e_i))$ for $i = 1, \ldots, k$. Note that this works for ordered DAGs as well as for unordered ones. In the former case $OUT(v) = e_1 \cdots e_k$ is a sequence of edges rather than a set and the direct subtrees are ordered accordingly.

Now the unfolding $unfold(D)$ of D yields the set of all trees $unfold_D(v)$ such that v is a root of D, and the unfolding of a DAG language L is given by $\bigcup_{D \in L} unfold(D)$. Pretending for the moment that we work in the ordered setting, one can then use edge swapping to show that $unfold(L(A))$ is a regular tree language, mainly by turning every rule $p_1 \cdots p_m \xrightarrow{\sigma} q_1 \cdots q_n$ into the set of rules $p_i \to \sigma(q_1, \ldots, q_n)$, obtaining a regular tree grammar. (An initial nonterminal S must be added, with all rules $S \to \sigma(q_1, \ldots, q_n)$ for which A contains the rule $\varepsilon \xrightarrow{\sigma} q_1 \cdots q_n$.)

However, for this argument to work properly one first has to remove useless rules from A because otherwise the regular tree grammar may generate trees that are unfoldings of DAGs not in $L(A)$. Useless rules can be detected by a technique that also allows us to decide whether $L(A)$ is empty. The key observation is that A can be seen as a Petri net whose places are the states of A. Every rule $I \xrightarrow{\sigma} O$ corresponds to a Petri net transition that consumes tokens from the places in I and produces tokens on the places in O. Again, this works in both the ordered and the unordered case as the Petri net is oblivious to an order on its places. Deciding the emptiness of $L(A)$ boils down to the question whether the Petri net has a *zero cycle*, meaning that it can take the empty configuration back to itself. This is because a run can be viewed as a top-down process that starts with the empty DAG. It then creates a root according to a rule, thus creating some outgoing "dangling" edges with states assigned to them. Then the process continues by applying rules, always taking some of the still unprocessed dangling edges, making them the incoming edges of a vertex and producing new dangling edges with assigned states (unless the vertex created is a leaf). Finally, all dangling edges (and thus "unused" states) must have vanished, corresponding to the null configuration of the Petri net.

Petri nets with a zero cycle are also said to be *structurally cyclic*. Deciding this property is a special case of the Petri net reachability problem which can be solved in polynomial time (Drewes and Leroux, 2015). Consequently, emptiness of regular DAG languages can be decided in polynomial time as well. Moreover, the result in (Drewes and Leroux, 2015) is obtained by presenting a polynomial-time algorithm that computes the set of all transitions of the Petri net that occur in zero cycles. Since these transitions are exactly the useful rules of A, the algorithm detects the latter.

With these pieces, especially the removal of useless rules, in place it can furthermore be shown that the finiteness problem is solvable in polynomial time as well. On the other hand, considering $L_{\mathrm{u}}(A)$ instead of $L(A)$, detection of useless rules, emptiness, and finiteness all become as hard as general Petri net reachability. Though this problem is decidable as well (Mayr, 1984; Kosaraju, 1982), no primitive recursive upper bound on its complexity is known.

In summary:

1. The unfolding of a regular DAG language (of ordered DAGs) yields a regular tree language; in particular, since the path language of a regular DAG language obviously coincides with the path language of its unfolding, it is a regular string language. In contrast, the path language of $L_{\mathrm{u}}(A)$ is not necessarily context-free. Furthermore, the latter shows that the unfolding of $L_{\mathrm{u}}(A)$ is not necessar-

ily a context-free tree language.

2. There is a polynomial algorithm that detects the useless rules of a DAG automaton A. However, one should be aware that some of the remaining rules may be useless for generating DAGs in $L_{\mathrm{u}}(A)$. Detecting those requires to solve the general Petri net reachability problem, which may very well be one of the hardest decidable problems there is.

3. Similarly, the emptiness and finiteness problems can be solved in polynomial time for $L(A)$, but solving the same problems for $L_{\mathrm{u}}(A)$ again requires to solve the general Petri net reachability problem.

These results are especially interesting in the light of the fact that most notions of DAG automata known from the literature can simulate the DAG automata discussed here and are, in addition, able to restrict the number of roots to one.

5 Further Properties of Regular DAG Languages

5.1 Closure under Set Theoretic Operations

Unsurprisingly, the class of regular DAG languages turns out to be closed under union and intersection, using the standard constructions. Slightly less obvious is the fact that that it is *not* closed under complementation or set difference. Consider the language L of all connected nonempty DAGs over $\{a, b\}$ in which every vertex is either a root of out-degree 2 or a leaf of in-degree 2. Thus, the elements of L are simple undirected cycles. Clearly, L is regular, and so is the subset containing only the DAGs over $\{a\}$. Their difference $L_b = L \setminus L'$ is the set of all DAGs in L containing at least one b. Now, assuming that L_b is regular and fixing a DAG automaton that accepts it, consider a run on a DAG $D \in L_b$. If D is large enough, it contains two edges with the same state that can be swapped in such a way that the resulting DAG D' falls apart into two components. As D' is still accepted, each of its components is in L_b. However, if we choose D in such a way that it contains only one b, then only one component of D' can contain that b, a contradiction.

5.2 Pumping

Swapping edges also yields two pumping lemmata. Both work by iterated application of the swapping operation. Given a DAG D, two edges e, e' of D, and some $n \geq 0$, let $D(e \bowtie e')^n$ denote the DAG obtained from $n + 1$ isomorphic copies $D_0, \ldots, D_n$ of D by swapping the copy of e' in D_{i-1} with the copy of e in D_i, for all $i \in \{1, \ldots, n\}$. (As before, swapping two edges f and f' means that $tar(f)$ and $tar(f')$ become the targets of f' and f, respectively.)

Pumping Property 1 For every DAG automaton A there is a constant c such that every DAG $D \in L(A)$ with at least c edges contains edges e, e' such that

(1) A accepts $D(e \bowtie e')^n$ for all $n \geq 0$ and

(2) each $D(e \bowtie e')^n$ contains a connected component of size $\geq n$.

The construction used to prove this property (Blum and Drewes, 2017) actually creates connected components in (b) that grow at a constant rate, which means that regular DAG languages exhibit a linear growth property: if we sort the DAGs in a given regular DAG language by size (number of vertices, say), then there is a global constant d such that the sizes of two consecutive DAGs differ by at most d.

Pumping Property 2 If a regular DAG language L contains a DAG D that has an undirected cycle, then D contains an edge e such that $D(e \bowtie e)^n \in L$ for all $n \geq 0$.

The property follows from the simple fact that swapping an edge e on a cycle in one copy of D with its counterpart in a second copy of D creates a connected DAG. This property entails an interesting consequence, as it is not difficult to show that $\{D\}$ is regular for every connected nonempty DAG D that does *not* contain a cycle.[4] Hence, by the closedness under union, finite sets of DAGs that do not contain cycles are regular. Together with the above property this proves that a finite set of connected nonempty DAGs is regular if and only if none of its DAGs contains a cycle.

6 Recognition

One of the most central computational problems for automata is recognition, also known as the membership problem: given an object D, in this

[4] This is a special case of the more general fact that every regular tree language is a regular DAG language (of ordered DAGs).

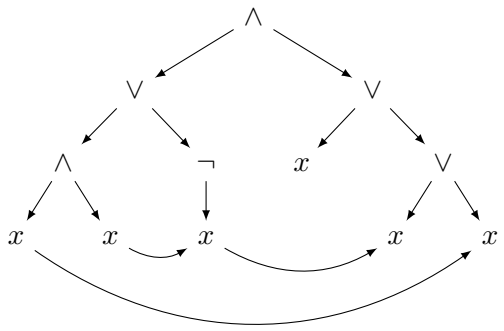

Figure 5: DAG representation of the propositional formula $((x_1 \wedge x_2) \vee \neg x_2) \wedge (x_3 \vee (x_2 \vee x_1))$

case a DAG, is it accepted by the automaton? In the weighted case, the answer is not *yes* or *no*, but the actual weight of D. Recognition comes in two flavors, depending on whether the automaton is part of the input – the *uniform recognition problem* – or not – the *non-uniform recognition problem*. For linguistic applications the potentially more difficult uniform version is the more relevant one, which should preferably be efficiently solvable. Unfortunately, it turns out that even non-uniform recognition is NP-complete, i.e., there exist fixed DAG automata whose accepted language is NP-complete.

6.1 NP-Completeness

An NP-complete regular DAG language is surprisingly easy to construct. Take the regular tree language representing propositional formulas over $\wedge$, $\vee$, $\neg$, and x, where x denotes an (anonymous) occurrence of a propositional variable. Link all occurrences of x that represent the same variable by a linear chain of edges. An example of such a DAG representation of the propositional formula $((x_1 \wedge x_2) \vee \neg x_2) \wedge (x_3 \vee (x_2 \vee x_1))$ is shown in Figure 5. Now a DAG automaton can verify that a formula is satisfiable, as follows: use states t and f representing truth values and rules such as $tt \xrightarrow{x} t$ and $ff \xrightarrow{x} f$, which guess a consistent assignment of truth values to all occurrences of the same variable, and $ff \xrightarrow{\wedge} f$, $tf \xrightarrow{\wedge} f$, $ft \xrightarrow{\wedge} f$, $tt \xrightarrow{\wedge} t$, which implement the operators. Moreover, for every rule $I \xrightarrow{\sigma} t$ there is a rule $I \xrightarrow{\sigma} \varepsilon$. Clearly, such an automaton accepts a DAG representing a propositional formula φ in the way described above if and only if φ is satisfiable.

6.2 Recognition

The NP-completeness result above indicates that, in general, there may be no efficient recognition algorithms for regular DAG languages. However, according to statistics reported by Chiang et al. (2016), DAGs actually arising from real-world AMRs are usually rather benign. In particular, although the treewidth of AMRs is unbounded in principle, real-world AMRs seem to have a small treewidth. Among 20 000 AMRs from the AMR Bank the maximum treewidth turned out to be 4, which was reached by only 31 AMRs, and the average treewidth was 1.55. It thus seems to be a relevant goal to develop recognition algorithms that work well on DAGs of small treewidth.

Let us recall here that a tree decomposition of a graph $G = (V, E, lab, src, tar)$ is a tree whose vertices, called *bags* here, are labelled with subsets of V such that

1. the union of all bags is V,[5]
2. for each edge $e \in E$ there is at least one bag containing both $src(e)$ and $tar(e)$, and
3. for each vertex $v \in V$ the bags containing v form a connected subgraph of the tree.

The treewidth of G is the smallest width of any of its tree decompositions, the width of a tree decomposition being the maximum size of its bags minus one. (A tree has treewidth 1 because it suffices to use a bag of size two for each edge e, namely $\{src(e), tar(e)\}$.)

A basic recognition algorithm generalizes the well-known forward algorithm by Baum (1972). Intuitively, it works as follows: To describe the algorithm, let us call a vertex together with its incident edges a *star*. For each such star in the input DAG, the algorithm records a set of candidate assignments of states to its edges. The initial candidate assignments to be recorded are given by the rules: every rule that could potentially be applied to the star (i.e., has the right vertex label and numbers of states in the left- and right-hand side) yields a candidate assignment to be recorded.

Now, the algorithm repeatedly chooses two stars s_1, s_2 that share an edge.[6] It contracts the

[5] For the sake of brevity, we often identify a bag with the set of vertices it is labelled with.

[6] To be precise, one also has to cover the case where more than one edge is shared between s_1 and s_2, but here we disregard this possibility.

edge, merging s_1 and s_2 into a single (possibly larger) star s. The candidate assignments to be recorded in s are obtained as follows: assume that s_1 and s_2 have edges $e_0, \ldots, e_m$ and $f_0, \ldots, f_n$, where $e_0 = f_0$ is the one being contracted. Then s has edges $e_1, \ldots, e_m, f_1, \ldots, f_n$, and for all candidate assignments $p_0, \ldots, p_m$ and $q_1, \ldots, q_n$ recorded in s_1 and s_2 with $p_0 = q_0$ the assignment $p_1, \ldots, p_m, q_1, \ldots, q_n$ is recorded in s. When the process stops, only one star s is left, which consists of a single vertex with no edges. The DAG is accepted if s records the empty candidate assignment, and is rejected if it contains no assignment at all.

With only little modification the algorithm may be used for weighted DAG automata, then computing the weight of the input DAG. In this case, each of the recorded candidate assignments is associated with a weight, which in the initial step is the weight of the rule in question. When two candidate assignments are combined in the process of merging two stars s_1, s_2, their weights are multiplied. If one of the candidate assignments for the combined star emerges in several ways from combinations of candidate assignments recorded in s_1, s_2, the resulting weights are summed up.

6.3 Efficiency of Recognition

The reader may have observed that the order in which edges are contracted may strongly affect the size of stars occurring during the execution of the algorithm, thus making the algorithm more or less efficient. Treating every vertex in the way described means that we are actually working with the *linegraph* $\mathcal{LG}(D)$ of D. It is obtained by turning every edge into a vertex and every vertex v into a clique on the incident edges of v (which are now vertices). The optimal contraction order can be read off an optimal tree decomposition of $\mathcal{LG}(D)$. A closer inspection of these facts (Chiang et al., 2016) revels that the algorithm can be implemented to run in time

$$O(|E| \cdot |Q|^{tw(\mathcal{LG}(D))+1}),$$

where $tw(G)$ denotes the treewidth of a graph G. The exponential dependency on the treewidth of $\mathcal{LG}(D)$ (rather than on the treewidth of D itself) is bad; as soon as D contains a vertex v of degree k the treewidth of $\mathcal{LG}(D)$ is at least $k - 1$ since there must be a bag in the tree decomposition that covers the entire clique of size k that corresponds to v.

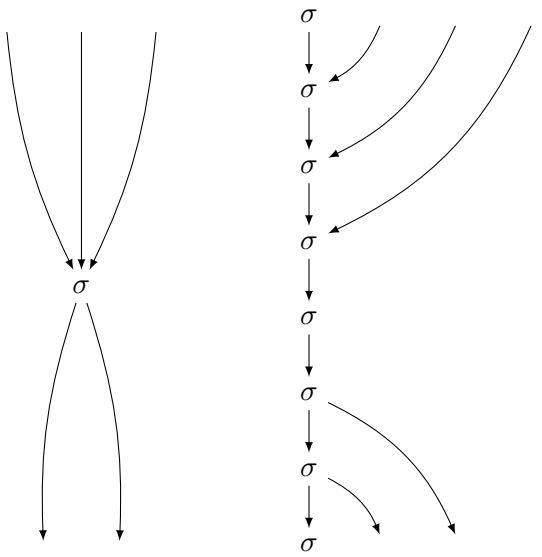

Figure 6: Binarizing a vertex of in-degree 3 and out-degree 2

A potential improvement can be achieved by a method called binarization, which is inspired by the well-known first-sibling next-child encoding of trees of unbounded rank by binary trees. The idea is to replace every vertex by a sub-DAG in which each vertex has in-degree $\leq$ 2 and out-degree $\leq$ 1 or else in-degree $\leq$ 1 and out-degree $\leq$ 2. A simple binarization scheme is illustrated in Figure 6 for a vertex of in-degree 3 and out-degree 2. Applying this schema to all vertices in a given DAG yields a binary DAG, and it is straightforward to modify the rules of a given DAG automaton in such a way that the automaton accepts the binarized version of the original DAG language.

By turning from an input DAG D to its binarized version D' we have thus reduced the degree of vertices, and hence the size of cliques in $\mathcal{LG}(D')$, to three. However, this is not enough in order to increase the efficiency of the algorithm because any fixed binarization such as the one above may be structurally incompatible with an optimal tree decomposition of D, thus actually making $tw(D')$ and $tw(\mathcal{LG}(D'))$ much larger than $tw(D)$.

The solution to this dilemma is provided by the fact that, for every tree decomposition of a graph G, there is a binary tree decomposition of G (i.e., a tree decomposition which is a binary tree) of the same width. If we, instead of binarizing vertices according to a fixed scheme, base our binarization

on an optimal binary tree decomposition of D, it can be shown that a better binarization is obtained.

Here is a rough outline of the method. Consider a binary tree decomposition T of D. It can be shown that we may, without affecting the width of T, build it in such a way that every edge e of D is covered by an explicitly assigned leaf bag of T, these bags being distinct for distinct edges. By the definition of tree decompositions, the bags containing a given vertex v of D form a subtree T_v of T. This tree contains one leaf for each edge e incident on v. Hence we can binarize D by replacing every vertex v by the corresponding T_v, attaching each of the original incident edges of v to the corresponding leaf of T_v. Intuitively, using the subtrees T_v of T to binarize vertices ensures that the structure of the resulting binary DAG D_T is compatible with the tree decomposition T. This makes it possible to "refine" T into a tree decomposition of D_T, in this way showing that, if the width of T is $k \geq 1$, then

$$tw(\mathcal{LG}(D_T)) \leq 2(k+1).$$

In particular, choosing a tree decomposition T of width $tw(D)$, we get

$$tw(\mathcal{LG}(D_T)) \leq 2(tw(D)+1).$$

As in the case of the specific binarization above, it is not very difficult to turn the original DAG automaton A into a DAG automaton A' that accepts the language of all binarized DAGs D_T such that $D \in L(A)$ and T is a binary tree decomposition of D. The application of a rule of A to a vertex v is simulated by rules that read the sub-DAG T_v, gathering the states on the original incoming and outgoing edges of v one by one. A disadvantage of this method is that it increases the number of states exponentially, but at the same time the inequalities above imply that the exponent $tw(\mathcal{LG}(D))$ is bounded from above by $2tw(D)+1$. Altogether, running the recognition algorithm above on the binarized versions of D and A yields a running time that is exponential in $|Q|$ and $tw(D)$ rather than in $tw(\mathcal{LG}(D))$. For a detailed discussion as well as formal constructions and proofs see Chiang et al. (2016).

7 DAG Languages of Unbounded Degree

Let us briefly discuss the case of unbounded vertex degree. Regular DAG languages are of bounded vertex degree simply because the set of rules of a DAG automaton is finite, and every rule applies only to vertices with a fixed in- and out-degree. If we want to change this, we need to represent an infinite set of rules $I \xrightarrow{\sigma} O$ in a finite manner. Similarly to the well-known case of unranked tree languages, we do this by replacing I and O by regular expressions over the alphabet of states. However, unrestricted regular expressions can specify arbitrary semilinear sets, which seems unreasonably powerful from a linguistic perspective: conditions such as "σ has twice as many incoming edges labelled with state q as it has incoming states labelled with state q or q'" lack linguistic motivation and would thus make the model overly powerful.

Extended DAG automata (Chiang et al., 2016), therefore, use Ochmański's c-regular expression (Ochmański, 1985), which are regular expressions in which the Kleene star is only applied to subexpressions over a unary alphabet. For example, $(qq)^*pp^* + qq^*$ is a c-regular expression that specifies the language of all finite multisets $M \in \mathcal{M}(\{p,q\})$ which either contain an even number of qs and at least one p, or otherwise at least one q and no p. (Recall that we are interested in multisets of states rather than sequences, because we are dealing with unordered DAGs. In the ordered case the above c-regular expression would be interpreted as specifying the set of all nonempty strings over $\{p,q\}$ which either consist of an even number of qs followed by at least one p, or contain no p at all.)

In the following, we denote the set of multisets denoted by a c-regular expression α by $[\![\alpha]\!]$. An *extended DAG automaton* is defined like a DAG automaton, except that its set R of rules now consists of extended rules of the form $\alpha \xrightarrow{\sigma} \beta$, where α and β are c-regular expressions over Q. An ordinary rule $I \xrightarrow{\sigma} O$ is an *instance* of $\alpha \xrightarrow{\sigma} \beta$ if $I \in [\![\alpha]\!]$ and $O \in [\![\beta]\!]$. A run ρ on a DAG $D = (V, E, lab, src, tar)$ is accepting if, for every vertex $v \in V$, $\rho(IN(v)) \xrightarrow{\sigma} \rho(OUT(v))$ is an instance of a rule in R.

7.1 The Weighted Case

Extended DAG automata can be made weighted by using weighted c-regular expressions in their rules. The semantics of a weighted c-regular expression α over Q is a function $[\![\alpha]\!] : \mathcal{M}(Q) \to \mathbb{S}$, where $\mathbb{S}$ is the semiring of weights. The weight of

an ordinary rule $I \xrightarrow{\sigma} O$ is

$$\sum_{(\alpha \xrightarrow{\sigma} \beta) \in R} [\![\alpha]\!](I) \cdot [\![\beta]\!](O),$$

using the addition and multiplication of $\mathbb{S}$. Now, every run gets as its weight the product of the weights of the rule instances applied in it, and every input DAG gets as its weight the sum of the weights of all its runs.

7.2 Transferring Results by Binarization

Every (weighted) c-regular expression has an equivalent (weighted) finite automaton working on multisets rather than strings, a so-called m-automaton. Essentially, such an m-automaton is an ordinary finite automaton whose behaviour is invariant under reordering the symbols in the input string (which thus, in effect, is treated as a multiset). Using such an m-automaton to implement the rules of an extended DAG automaton, binarization works in essentially the same way as for non-extended DAG automata. In other words, for every extended DAG automaton A there is a binary DAG automaton A' such that $L(A')$ is the set of all binarized representations of DAGs in $L(A)$, and similarly for the weighted case.

It follows immediately that emptiness and finiteness are decidable, and that the recognition algorithm for DAG automata can be used to implement recognition for extended DAG automata, all with essentially the same running times as in the non-extended case.[7] Even without the use of binarization it is clear that the second pumping property of Section 5.2 holds for extended DAG automata as well, since runs are still invariant under edge swapping. The first pumping property, however, does not carry over because its proof relies on the fact that, in the case of DAGs of bounded degree, large DAGs contain long simple paths on which states must eventually repeat. This is not true anymore for DAGs of unbounded degree.

Acknowledgments

I thank everyone who contributed to the work on DAG automata either directly or by discussing ideas and providing opinions at various occasions. In particular, this includes Johannes Blum, David Chiang, Daniel Gildea, Adam Lopez, Giorgio Satta, the members of the research group Foundations of Language Processing at Umeå University, and the participants of the 2014 Johns Hopkins summer workshop in Prague and the Dagstuhl Seminars 15122 and 17142.

[7] Naturally, here the input size must include the m-automata that implement the c-regular expressions appearing in the rules. These m-automata may, however, be compiled into a single one to reduce the input size by representing common parts only once.

References

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

Laura Banarescu, 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 *Proc. 7th Linguistic Annotation Workshop, ACL 2013 Workshop*.

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

Johannes Blum and Frank Drewes. 2016. Properties of regular DAG languages. In A.H. Dediu, J. Janoušek, C. Martín-Vide, and B. Truthe, editors, *Proc. 10th Intl. Conf. on Language and Automata Theory and Applications*. volume 9618 of *Lecture Notes in Computer Science*, pages 427–438.

Johannes Blum and Frank Drewes. 2017. Language theoretic properties of regular DAG languages. Submitted.

Witold Charatonik. 1999. Automata on DAG representations of finite trees. Research Report MPI-I-1999-2-001, Max-Planck-Institut für Informatik, Saarbrücken.

David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. 2016. Weighted DAG automata for semantic graphs. Submitted.

Frank Drewes. 2006. *Grammatical Picture Generation – A Tree-Based Approach*. Texts in Theoretical Computer Science. An EATCS Series. Springer.

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

Frank Drewes and Jérôme Leroux. 2015. Structurally cyclic Petri nets. *Logical Methods in Computer Science* 11(4:15).

Zoltán Fülöp and Heiko Vogler. 2009. Weighted tree automata and tree transducers. In Werner Kuich,

Manfred Droste, and Heiko Vogler, editors, *Handbook of Weighted Automata*, Springer, chapter 9, pages 313–403.

Ferenc Gécseg and Magnus Steinby. 1984. *Tree Automata*. Akadémiai Kiadó, Budapest. Online version available under https://arxiv.org/abs/1509.06233.

Ferenc Gécseg and Magnus Steinby. 1997. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages.* Vol. 3: *Beyond Words*, Springer, chapter 1, pages 1–68.

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

Paul Kingsbury and Martha Palmer. 2002. From treebank to propbank. In *Proc. 3rd Intl. Conf. on Language Resources and Evaluation (LREC 2002)*.

S. Rao Kosaraju. 1982. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*. pages 267–281.

Ernst W. Mayr. 1984. An algorithm for the general Petri net reachability problem. *SIAM J. Comput.* 13:441–460.

Edward Ochmański. 1985. Regular behaviour of concurrent systems. *Bulletin of the European Association for Theoretical Computer Science* 27:56–67.

Lutz Priese. 2007. Finite automata on unranked and unordered DAGs. In T. Harju, J. Karhumäki, and A. Lepistö, editors, *Proc. 11th Intl. Conf. on Developments in Language Theory (DLT 2007)*, volume 4588 of Lecture Notes in Computer Science, pages 346–360.

Daniel Quernheim and Kevin Knight. 2012. Towards probabilistic acceptors and transducers for feature structures. In *Proc. 6th Workshop on Syntax, Semantics and Structure in Statistical Translation*. Association for Computational Linguistics, pages 76–85.