Information and Computation 265 (2019) 57–76

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Language theoretic properties of regular DAG languages ☆

Johannes Blum [a], Frank Drewes [b,*]

[a] Department of Computer and Information Science, University of Konstanz, Germany
[b] Department of Computing Science, Umeå University, Sweden

ARTICLE INFO

Article history:
Received 11 June 2016
Received in revised form 15 January 2017
Available online 30 January 2019

ABSTRACT

We study sets of directed acyclic graphs, called regular DAG languages, which are accepted by a recently introduced type of DAG automata motivated by current developments in natural language processing. We prove (or disprove) closure properties, establish pumping lemmata, characterize finite regular DAG languages, and show that "unfolding" turns regular DAG languages into regular tree languages, which implies a linear growth property and the regularity of the path languages of regular DAG languages. Further, we give polynomial decision algorithms for the emptiness and finiteness problems, and show that deterministic DAG automata can be minimized and tested for equivalence in polynomial time.

1. Introduction

String and tree languages have been the subject of many years of research, with thousands of papers contributing to their theory. One of the application fields in which tree languages are especially useful is natural language processing, because trees can represent the syntactic structure of sentences in a convenient manner. However, trees are not equally well suited for representing semantic information. This is because an individual concept or entity can play several semantic roles in a sentence, but should be represented only once. Therefore, instead of trees, graph-structured representations are much more appropriate when it comes to the representation of natural language meaning. A few years ago, a comparatively simple representation of this kind was proposed, the *Abstract Meaning Representation* (AMR, see [2]). Researchers in natural language processing have started building corpora of such meaning representations and looking for formal models able to handle AMRs and similar structures.

AMRs may be formalized as vertex-labelled directed acyclic graphs (DAGs). Quernheim and Knight [17] have therefore proposed a notion of DAG automata for handling sets of DAGs, in the following called DAG languages. In fact, their notion of DAG automata was not the first one to be found in the literature; others had been proposed and studied in, e.g. [1,5,13,16]. However, none of these automata types was defined with the needs of processing meaning representations in mind. In one way or the other, all of them thus seem to be unnecessarily powerful from this point of view. Ideally, the automaton model should be just powerful enough, because this increases the likelihood that it comes with efficient algorithms and good closure properties. Moreover, too powerful a model may result in overfitting when machine learning approaches are employed to "learn" automata from corpora. Continuing the work by Quernheim and Knight, Chiang et al. [6] propose a simpler model of DAG automata and argue that this model is sufficiently powerful to capture the phenomena that are

☆ This article is an extended version of [4]. Most of the results have originally been proved in the master thesis [3] of the first author.

* Corresponding author.
E-mail addresses: johannes.blum@uni-konstanz.de (J. Blum), drewes@cs.umu.se (F. Drewes).

https://doi.org/10.1016/j.ic.2017.07.011
0890-5401/

relevant when modelling AMRs. A rule has the form $\{p_1, \ldots, p_m\} \overset{\sigma}{\longleftrightarrow} \{q_1, \ldots, q_n\}$, meaning that the outgoing edges of a vertex labelled with σ can be assigned the states $q_1, \ldots, q_n$ if its incoming edges are assigned the states $p_1, \ldots, p_m$. In the terminology of [15], a paper that shows the equivalence of three alternative ways of defining DAG automata, these DAG automata are thus of the *edge-marking* type. In the current paper, we continue and complement the work in [6] by studying the language-theoretic and algorithmic properties of these DAG automata and their languages, which we call the *regular DAG languages*. We refer to [6] for detailed comparisons of the various types of DAG automata in the references above with those considered in the current paper. Let us only mention that our DAG automata, intuitively, are ranked and ordered versions of those studied by Priese [16], but without the ability of the latter to express "initial" and "final" restrictions on the roots and leaves of the input DAGs. As a consequence, our DAG automata are considerably weaker, and thus turn out to have a number of nice properties. Just to mention an example, their path languages (i.e. the string languages obtained by reading the vertex labels on all root-to-leaf paths in the DAGs of a regular DAG language) are regular (Corollary 5.3) whereas those of the DAG languages in [16] are not even context-free. Indeed, from a linguistic point of view, the path languages of reasonable AMR languages should be regular (see the discussion of this question in [6]).

It should be pointed out that we, in contrast to [6], consider DAGs in which the incoming and outgoing edges of each vertex are ordered, i.e. they form two sequences of edges, while those in [6] are unordered, forming sets of edges. We choose the ordered setting because it yields a direct generalization of the theory of tree automata, and because certain notions of interest, such as determinism, are more useful in the ordered setting. From the relevant definitions, it is obvious that our DAG automata encompass those of [6] because one may simply view a rule $\{p_1, \ldots, p_m\} \overset{\sigma}{\longleftrightarrow} \{q_1, \ldots, q_n\}$ as a shorthand for the set of all rules $p_{i_1} \cdots p_{i_m} \overset{\sigma}{\longleftrightarrow} q_{j_1} \cdots q_{j_n}$ for which $\{i_1, \ldots, i_m\} = \{1, \ldots, m\}$ and $\{j_1, \ldots, j_n\} = \{1, \ldots, n\}$. All results proved in the present paper are thus easily seen to carry over to the setting in [6] (with the exception of those whose formulation does not make sense in that setting).

DAG automata in the sense of this paper accept only DAG languages of bounded vertex degree. In addition to these DAG languages, [6] also considers a notion of extended DAG automata in which this restriction is removed in a way similar to the theory of unranked tree automata. It is, however, shown in [6] that such DAG languages (and their corresponding DAG automata) can always be "binarized", thus encoding every DAG by one in which each vertex has at most two incoming and one outgoing edge, or else at most one incoming and two outgoing edges. This is inspired by the first-sibling next-child encoding known from the theory of unranked tree languages. Using this encoding it follows immediately that all results of the present paper hold for the extended DAG automata of [6] as well, which is why we do not consider extended DAG automata.

This paper is structured as follows. The next section compiles the basic notions and notation. In particular, it defines DAG automata and the DAG languages they accept. Section 3 studies the closure properties of the class of regular DAG languages. The results – closedness under union and intersection, but not under complementation – are easy and expected, but the proof of the non-closedness under complementation provides us with an opportunity to introduce an operation which will turn out to be central to the entire paper: the edge swap operation. This operation is exploited in Section 4 to prove two different pumping lemmata. From the second pumping lemma, we furthermore derive a characterization of the finite regular DAG languages: these are exactly those regular DAG languages which do not contain undirected cycles. The main result of Section 5 relates regular DAG languages and regular tree languages: "unfolding" the DAGs of a regular DAG language yields a regular tree language. As corollaries, we obtain a linear growth property, establish the regularity of the path language of a regular DAG language, and conclude that the emptiness problem is decidable in polynomial time. (The second and third of these results were already shown in [6], but with more complicated proofs.) In Section 6 we consider the finiteness problem and show that this problem is decidable in polynomial time as well. An auxiliary result worth mentioning in its own right is that DAG automata can, in polynomial time, be turned into reduced DAG automata, i.e. useless rules can be detected and removed in polynomial time. Section 7 is concerned with deterministic DAG automata, the main results being that these can be minimized in polynomial time, and that their equivalence problem is decidable in polynomial time as well. The paper ends with a few conclusions and possible directions for future work presented in Section 8.

The results of Sections 3, 4, and 6 were already presented in [4], though without the detailed proofs we provide here. The results in Sections 5 and 7 are new.

2. Regular DAG languages

In this section we introduce DAG automata and their languages, after compiling some basic mathematical notions used throughout the paper.

The set of non-negative integers, including 0, is denoted by $\mathbb{N}$. For $n \in \mathbb{N}$ we let $[n] = \{1, \ldots, n\}$, which in particular includes the case $[0] = \emptyset$. The composition of functions $f\colon A \to B$ and $g\colon B \to C$ is denoted by $g \circ f$, i.e. $(g \circ f)(a) = g(f(a))$ for all $a \in A$. Given a set A, its cardinality is denoted by $|A|$, and the set of all finite sequences (or strings) over A is denoted by A^*. A^* includes the empty string λ, which is of length 0. In general, the length of $w \in A^*$ is denoted by $|w|$, and $[w]$ denotes the smallest set A such that $w \in A^*$. The concatenation of sequences α, β is usually denoted by juxtaposition, but for the sake of clarity we may sometimes denote it by $\alpha \cdot \beta$ instead. The canonical extensions of a function $f\colon A \to B$ to functions from A^* to B^* and from 2^A to 2^B (the powersets of A and B, resp.), are denoted by f as well. In other words, $f(a_1 \cdots a_n) = f(a_1) \cdots f(a_n)$ for $a_1, \ldots, a_n \in A$, and $f(S) = \{f(a) \mid a \in S\}$ for $S \subseteq A$. A finite set Σ is also called an alphabet, and its elements symbols.

Definition 2.1 *(Σ-graph).* A *Σ-graph* over an alphabet Σ is a tuple $G = (V, E, lab, in, out)$ with the following components:

- V and E are the finite sets of *vertices* and *edges*, resp.,
- $lab\colon V \to \Sigma$ is the *vertex labelling*, and
- $in, out\colon V \to E^*$ assign to each vertex its incoming and outgoing edges in such a way that every edge $e \in E$ occurs exactly once in all the $in(u)$, $u \in V$, and exactly once in all the $out(v)$, $v \in V$.

Given a Σ-graph G as above, we often refer to its components by $V_G, E_G, lab_G, in_G, out_G$, resp. By the last condition in the definition of Σ-graphs, every edge $e \in E_G$ has unique vertices u, v such that $e \in [out_G(u)] \cap [in_G(v)]$. We call u the *source* and v the *target* of e and denote them by $src_G(e)$ and $tar_G(e)$ resp. Naturally, the *empty graph*, denoted by $\varnothing$, is the one with $V_G = \emptyset$. Given vertices $u, v \in V_G$, a sequence $e_1 \cdots e_n$ of edges $e_1, \ldots, e_n \in E_G$ is a *path from u to v* (or simply a *path* if u and v are irrelevant) if there are vertices $v_0, \ldots, v_n \in V_G$ such that $u = v_0$, $v_n = v$, and $\{src_G(e_i), tar_G(e_i)\} = \{v_{i-1}, v_i\}$ for all $i \in [n]$. G is *connected* if, for all $u, v \in V_G$, there exists a path from u to v in G. A path is *simple* if $v_i \neq v_j$ for all distinct $i, j \in [n]$, *empty* if $n = 0$, *directed* if $src_G(e_i) = v_{i-1}$ and $tar_G(e_i) = v_i$ for all $i \in [n]$, and a *cycle* if $n > 0$ and $u = v$.

Throughout the paper, the reader should keep in mind that, according to the above definitions, paths and cycles are undirected unless explicitly qualified as directed.

As usual, graphs G, G' are said to be *isomorphic* if there is a pair (g, h) of bijective mappings $g\colon V_G \to V_{G'}$ and $h\colon E_G \to E_{G'}$, called an *isomorphism*, such that $lab_G = lab_{G'} \circ g$, $h \circ in_G = in_{G'} \circ g$, and $h \circ out_G = out_{G'} \circ g$. Throughout this paper we will not distinguish between isomorphic graphs except in a few cases where it is necessary in order to avoid confusion.

Sometimes it is convenient to assign explicit ranks to the symbols of an alphabet, thus determining the number of incoming and outgoing edges a vertex labelled with this symbol is expected to have. We say that an alphabet Σ is *doubly ranked* (or simply *ranked*, for short) if a rank $rk_\Sigma(\sigma) \in \mathbb{N}^2$ is specified for every symbol $\sigma \in \Sigma$. Strictly speaking, the ranked alphabet is the pair (Σ, rk_Σ), but for simplicity we shall keep rk_Σ implicit.

Definition 2.2 *(DAG).* Let Σ be an alphabet. A *Σ-DAG* (or simply *DAG*) is a Σ-graph that does not contain any directed cycle. The set of all nonempty connected Σ-DAGs is denoted by $\mathcal{D}_\Sigma$. A *DAG language* is a subset of $\mathcal{D}_\Sigma$, for some alphabet Σ.

If the alphabet Σ is ranked, we restrict the set of all Σ-DAGs and its subset $\mathcal{D}_\Sigma$ to those DAGs G over Σ such that every vertex $v \in V_G$ satisfies $rk_\Sigma(lab_G(v)) = (|in_G(v)|, |out_G(v)|)$.

Note that Σ-DAGs may be disconnected, but DAG languages are subsets of $\mathcal{D}_\Sigma$ which contains only the connected Σ-DAGs. This choice will be motivated soon; see Observation 2.4 and the paragraph preceding it. We do, in fact, frequently need either one in the remainder of the paper.

As usual, we call a vertex $v \in V_G$ a *leaf* if $out_G(v) = \lambda$. Deviating slightly from traditional mathematical terminology, we call $v \in V_G$ a *root* if $in_G(v) = \lambda$. Thus, v being a root does *not* imply that every other vertex can be reached from v on a directed path.

After these preparations, we are ready to give our definition of DAG automata.

Definition 2.3 *(DAG automaton).* A *DAG automaton* is a triple $A = (Q, \Sigma, R)$ where Q is a finite set of *states*, Σ is an alphabet and R is a finite set of *rules* of the form $\alpha \overset{\sigma}{\longleftrightarrow} \beta$ where $\sigma \in \Sigma$ and $\alpha, \beta \in Q^*$. Such a rule is also called a σ-rule, and the state sequences α and β are its *head* and *tail*, resp.

A *run* of A on a DAG $G = (V, E, lab, in, out)$ is a mapping $\rho\colon E \to Q$ such that R contains the rule[1]

$$\rho(in(v)) \overset{lab(v)}{\longleftrightarrow} \rho(out(v))$$

for every vertex $v \in V$. A *accepts* G if such a run exists. The *DAG language accepted by A* is the set

$$L(A) = \{G \in \mathcal{D}_\Sigma \mid A \text{ accepts } G\}.$$

The class *RDL* of *regular DAG languages* consists of all DAG languages accepted by DAG automata.

For a vertex $v \in V$ with $lab(v) = \sigma$, a run ρ on G is said to *use* the rule $\rho(in(v)) \overset{\sigma}{\longleftrightarrow} \rho(out(v))$ in v. If the alphabet Σ of A is ranked, we say that A is ranked and require that all rules respect the ranks of symbols, in the sense that every rule $\alpha \overset{\sigma}{\longleftrightarrow} \beta$ satisfies $rk_\Sigma(\sigma) = (|\alpha|, |\beta|)$. Thus, both in the general and in the ranked case we have $L(A) \subseteq \mathcal{D}_\Sigma$.

We note here that trees over an alphabet Σ, in the usual sense of tree language theory, are a special case of DAGs. A DAG automaton "is" a finite-state tree automaton if all rules $\alpha \overset{\sigma}{\longleftrightarrow} \beta$ satisfy $|\alpha| \leq 1$. Rules with $|\alpha| = 0$ process the root

[1] Recall that we extend functions such as ρ in the canonical way to sequences and sets.

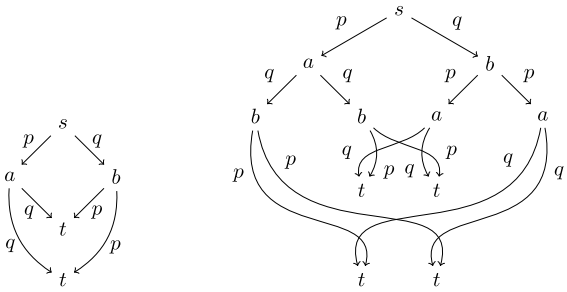


Fig. 1. Runs of the DAG automaton in Example 2.5.

of a tree while the others process its internal vertices. Because of this distinction between roots and non-roots the explicit definition of initial and final states becomes unnecessary to match top-down and bottom-up finite-state tree automata, resp. It is thus straightforward to show that a tree language over Σ is regular (in the sense of tree language theory) if and only if it is a regular DAG language in the sense defined above.

It may further be of interest to note that the class of regular DAG languages is also closed under edge reversal: for a DAG G, let $G^{\updownarrow} = (V_G, E_G, lab_G, out_G, in_G)$, and, for a DAG automaton $A = (Q, \Sigma, R)$, $A^{\updownarrow} = (Q, \Sigma, R^{\updownarrow})$ where $r^{\updownarrow} = (\beta \xleftrightarrow{\sigma} \alpha)$ for every rule $r = (\alpha \xleftrightarrow{\sigma} \beta)$. Then it follows immediately from the relevant definitions that $L(A^{\updownarrow}) = L(A)^{\updownarrow}$. This yields a strong duality principle: properties and results regarding DAG automata and their languages are invariant under edge reversal, i.e., they continue to hold if DAGs are "turned upside down".

As a final remark regarding Definition 2.3, we emphasize that $L(A)$ only contains nonempty connected DAGs, even though DAG automata can *run* on arbitrary DAGs. This definition of $L(A)$ is motivated by the fact that A accepts a disjoint union of DAGs if and only if it accepts each of its connected components. More precisely, given two disjoint DAGs G_1 and G_2 (where disjointness means that $V_{G_1} \cap V_{G_2} = \emptyset = E_{G_1} \cap E_{G_2}$), let us denote their union by $G_1 \,\&\, G_2$, i.e. $G_1 \,\&\, G_2 = (V_{G_1} \cup V_{G_2}, E_{G_1} \cup E_{G_2}, lab_{G_1} \cup lab_{G_2}, in_{G_1} \cup in_{G_2}, out_{G_1} \cup out_{G_2})$, where the union of functions is defined by virtue of them being binary relations. If G_1 and G_2 are not disjoint, we assume that they are silently replaced by disjoint copies to be able to build $G_1 \,\&\, G_2$. For a set L of DAGs, let $L^{\&}$ be the smallest set of DAGs such that $\varnothing \in L^{\&}$ and $\{G \,\&\, G' \mid G \in L, G' \in L^{\&}\} \subseteq L^{\&}$, i.e. $L^{\&}$ consists of all DAGs of the form $G_1 \,\&\, \cdots \,\&\, G_n$ such that $n \in \mathbb{N}$ and $G_1, \ldots, G_n \in L$. Then we have the following as a direct consequence of the definition of runs:

Observation 2.4. For every DAG automaton A, $L(A)^{\&}$ is the set of all DAGs accepted by A.

Thus, our definition of $L(A)$ makes it reasonable to speak about DAG automata accepting finite, empty, and infinite DAG languages, whereas $L(A)^{\&}$ is never empty (it always contains the empty DAG $\varnothing$), and is finite if and only if it equals $\{\varnothing\}$, which is the case if and only if $L(A) = \emptyset$.

Example 2.5. Consider the DAG automaton $A = (Q, \Sigma, R)$ where $Q = \{p, q\}$, $\Sigma = \{s, a, b, t\}$ and $R = \{\lambda \xleftrightarrow{s} pq, p \xleftrightarrow{a} qq, q \xleftrightarrow{b} pp, qp \xleftrightarrow{t} \lambda\}$. Thus, A is in fact ranked, with $rk_\Sigma(s) = (0, 2)$, $rk_\Sigma(a) = rk_\Sigma(b) = (1, 2)$, and $rk_\Sigma(t) = (2, 0)$. A run of A on a DAG G is shown in Fig. 1 on the left. In such drawings incoming and outgoing edges are ordered from left to right in order to avoid having to use an explicit numbering.

The run can be constructed in a top-down manner, as follows. In the root of G the only applicable rule is $\lambda \xleftrightarrow{s} pq$. Now, since the left child of the root has the label a and its incoming edge carries the state p, we can apply the rule $p \xleftrightarrow{a} qq$ to it. To the right child we apply the rule $q \xleftrightarrow{b} pp$. This leaves us with the two leaves. In each of it the rule $qp \xleftrightarrow{t} \lambda$ can be used. This completes the run, thus showing that A accepts G.

The right part of Fig. 1 shows a run on another DAG accepted by A. It also illustrates the fact that $L(A)$ is not finite as one can systematically construct a sequence of DAGs of increasing size that are accepted by A.

3. Closure properties of *RDL*

Before discussing more interesting properties of the class *RDL*, let us note that it is, similarly to the case of tree languages, closed under union and intersection:

Lemma 3.1 *(Closedness under union and intersection). RDL is closed under union and intersection.*

Proof. The proofs are standard. Consider DAG automata $A = (Q, \Sigma, R)$ and $A' = (Q', \Sigma', R')$, where $Q \cap Q' = \emptyset$. Then $(Q \cup Q', \Sigma \cup \Sigma', R \cup R')$ accepts $L(A) \cup L(A')$ and $(Q \times Q', \Sigma \cap \Sigma', R'')$ accepts $L(A) \cap L(A')$, where R'' consists of all rules

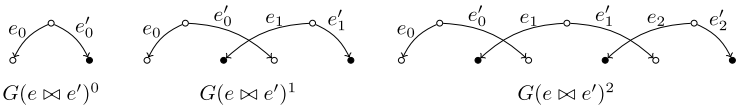


Fig. 2. The repeated edge swap between $k+1$ copies of G; $G(e \bowtie e')^0$ is isomorphic to G and each further copy G_{k+1} is attached to $G(e \bowtie e')^k$ by swapping its edge e with the edge e' of the preceding copy.

$(p_1, p'_1)\cdots(p_m, p'_m) \overset{\sigma}{\longleftrightarrow} (q_1, q'_1)\cdots(q_n, q'_n)$ such that $(p_1 \cdots p_m \overset{\sigma}{\longleftrightarrow} q_1 \cdots q_n) \in R$ and $(p'_1 \cdots p'_m \overset{\sigma}{\longleftrightarrow} q'_1 \cdots q'_n) \in R'$. The correctness of this construction should be obvious. □

In view of the lemma above, it may be an interesting observation that $RDL^{\&} = \{L^{\&} \mid L \in RDL\}$ is *not* closed under union. To see this, consider two arbitrary regular DAG languages $L_1 = \{G_1\}$ and $L_2 = \{G_2\}$ where G_1 and G_2 are DAGs that are not isomorphic. We have $L_i^{\&} \in RDL^{\&}$ for $i = 1, 2$. Assume now that there is a DAG automaton A that accepts a DAG G if and only if $G \in L_1^{\&} \cup L_2^{\&}$. Then A accepts $G_1 \,\&\, G_2$ as we can mix connected components of graphs accepted by A, but we have $G_1 \,\&\, G_2 \notin L_1^{\&} \cup L_2^{\&}$.

We are now going to prove the less obvious result that *RDL* is not closed under complement. In fact, more interesting than the result itself is the technique used to prove it, which is based on a simple but very useful operation on DAGs called edge swap. This operation will frequently be used later in this article.

Definition 3.2 *(Edge swap).* Let $G = (V, E, lab, in, out)$ be a DAG. Two edges $e_0, e_1 \in E$ are *independent* if there is no directed path from $tar_G(e_i)$ to $src_G(e_{1-i})$ for $i \in \{0, 1\}$. In this case, the *edge swap of e_0 and e_1* is defined and yields the DAG $G[e_0 \bowtie e_1] = (V, E, lab, h \circ in, out)$ given by

$$h(e) = \begin{cases} e_{1-i} & \text{if } e = e_i \text{ for some } i \in \{0, 1\} \\ e & \text{otherwise.} \end{cases}$$

Note that, by the independence requirement, $G[e_0 \bowtie e_1]$ is indeed a DAG. Indeed, assume that $G[e_0 \bowtie e_1]$ contains a directed cycle. The cycle contains at least one of e_0, e_1. If it contains only e_i, then the rest of the cycle is a directed path in G from $tar_G(e_{1-i})$ to $src_G(e_i)$. If both edges are part of the cycle, then G contains a directed path from $tar_G(e_0)$ to $src_G(e_0)$, which cannot be because G is acyclic.

The reader should also note that independence is automatically guaranteed in $(G \,\&\, G')[e \bowtie e']$, where $e \in E_G$ and $e' \in E_{G'}$.

For $k \in \mathbb{N}$, we can moreover connect $k+1$ disjoint isomorphic copies of a DAG G by systematically swapping copies of two (not necessarily distinct) edges $e, e' \in E_G$ between them. For this, choose disjoint isomorphic copies $G_0, G_1, \ldots$ of G. For $i \in \mathbb{N}$ let e_i and e'_i be the copies of e and e', resp., in G_i. Then the graph $G(e \bowtie e')^k$ is formally defined as follows, for $k \in \mathbb{N}$ (see also the illustration in Fig. 2):

$$G(e \bowtie e')^0 = G_0 \quad \text{and} \quad G(e \bowtie e')^{k+1} = (G(e \bowtie e')^k \,\&\, G_{k+1})[e'_k \bowtie e_{k+1}].$$

The usefulness of edge swaps is due to the fact that, given a run, independent edges that are assigned the same state can obviously be swapped without affecting the validity of the run. Thus, we have the following lemma.

Lemma 3.3 *(Edge swap preserves runs). Let A be a DAG automaton and let $G \in L(A)^{\&}$. Then for each pair of independent edges $e_1, e_2 \in E_G$, every run ρ of A on G that satisfies $\rho(e_1) = \rho(e_2)$ is also a run on $G[e_1 \bowtie e_2]$. In particular, we have $G[e_1 \bowtie e_2] \in L(A)^{\&}$ if such a run ρ exists.*

Edge swapping allows us to prove that *RDL* is not closed under complement.

Theorem 3.4 *(Non-closedness under complementation). There is a ranked alphabet Σ and a regular DAG language $L \subseteq \mathcal{D}_\Sigma$ such that $\mathcal{D}_\Sigma \setminus L$ is not regular.*

Proof. Consider the ranked alphabets $\Sigma = \{s, a, b\}$ and $\Sigma_0 = \{s, a\}$, where the rank of s is $(0, 2)$ and that of a and b is $(2, 0)$. Thus, all vertices in DAGs over Σ are binary roots (labelled by s) or leaves (with labels in $\{a, b\}$). Then $L = \mathcal{D}_{\Sigma_0} \in RDL$ is the DAG language accepted by the DAG automaton $A = (\{p\}, \Sigma, \{\lambda \overset{s}{\longleftrightarrow} pp, pp \overset{a}{\longleftrightarrow} \lambda\})$.

Now consider $\bar{L} = \mathcal{D}_\Sigma \setminus L$, the set of all DAGs in $\mathcal{D}_\Sigma$ that contain at least one vertex with label b. Assume that there is a DAG automaton $\bar{A}$ with $L(\bar{A}) = \bar{L}$. In particular, $\bar{A}$ accepts, for every $i \geq 1$, the DAG G_i with $V_{G_i} = \{s_1, \ldots, s_i, a_1, \ldots, a_{i-1}, b\}$, built as follows. For $j \in [i]$ we let $lab_{G_i}(s_j) = s$ and for $j \in [i-1]$ we let $lab_{G_i}(a_j) = a$, while $lab_{G_i}(b) = b$. Every vertex a_j has two incoming edges e_j^1 and e_j^2 coming from $s_{(j-2 \bmod i)+1}$ and s_j, respectively, and vertex b has two incoming edges from s_{i-1} and s_i; see the left DAG in Fig. 3.

Since $G_i \in L(\bar{A})$, there exists a run of $\bar{A}$ on G_i for every $i \in \mathbb{N}$. As $\bar{A}$ has only a finite number of states, we can now choose i large enough such that for every run ρ of $\bar{A}$ on G_i there are distinct a_k, a_l such that the states of their first incoming edges

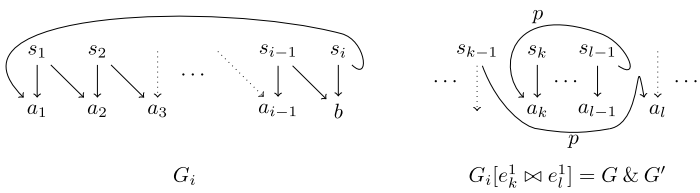


Fig. 3. The generic DAG G_i and its decomposition into $G \,\&\, G'$.

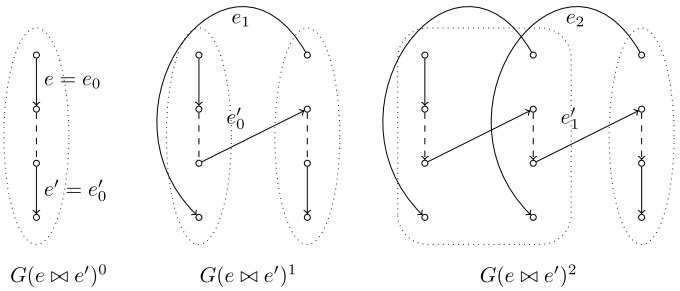


Fig. 4. An example of the graphs $G = G(e \bowtie e')^0$, $G(e \bowtie e')^1$ and $G(e \bowtie e')^2$.

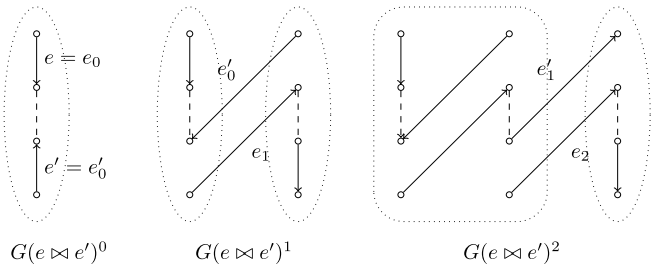


Fig. 5. The components of $G(e \bowtie e')^n$ do not necessarily increase in size.

are identical, i.e. $\rho(e_k^1) = \rho(e_l^1)$. Now, swapping e_k^1 and e_l^1 makes G_i fall apart into two connected components, i.e. we have $G_i[e_k^1 \bowtie e_l^1] = G \,\&\, G'$ for DAGs G and G', as is illustrated in Fig. 3 on the right.

According to Lemma 3.3 this means $G \,\&\, G' \in L(\bar{A})^{\&}$, which shows that $G, G' \in L(\bar{A})$ by the definition of $L(\bar{A})^{\&}$. But one of these DAGs contains only vertices with labels s and a and is thus in L, a contradiction. □

4. Necessary properties of regular DAG languages

The proof of Theorem 3.4 relies on a fact very frequently used in automata theory, namely that an automaton has to reuse some states when it processes an input of a certain size. In particular, this is often used to prove pumping lemmata for automata-like formalisms of various descriptions. We now present a pumping lemma for regular DAG languages whose proof proceeds along these lines, using Lemma 3.3 to pump the DAG by connecting it to an isomorphic copy of itself without invalidating the run.

Lemma 4.1 *(Pumping lemma 1). For every $L \in RDL$ there is a constant $k \in \mathbb{N}$ such that for all DAGs $G \in L$ with $|E_G| > k$ there are two edges $e, e' \in E_G$ such that for all $n \in \mathbb{N}$ the DAG $G(e \bowtie e')^n$ is in $L^{\&}$. Furthermore, each $G(e \bowtie e')^n$ contains a connected component C_n such that $|V_{C_n}| > n$.*

Proof. Let $L \in RDL$ be a regular DAG language and $A = (Q, \Sigma, R)$ be a DAG automaton with $L(A) = L$. Let $G \in L$ with $|E_G| > |Q|$. Given a run ρ of A on G, there are two edges $e, e' \in E_G$ such that $\rho(e) = \rho(e')$ as ρ must use one state at least twice. By induction on n, and using Lemma 3.3 it thus follows that $G(e \bowtie e')^n \in L(A)^{\&}$ for all $n \in \mathbb{N}$. This is illustrated in Fig. 4.

But the proof is not finished yet, because the DAGs $G(e \bowtie e')^n$ may fall apart into n connected components of bounded size. As illustrated in Fig. 5 this can happen if e and e' have different orientations in the sense that every (undirected) path from the target of e to the source of e' contains e or e'. We show that, otherwise, the situation is nicer:

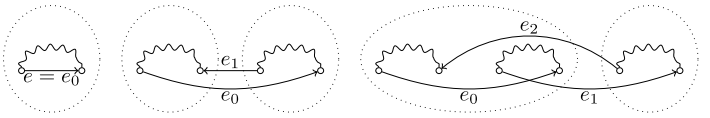


Fig. 6. Cycles in the DAGs G_0, G_1 and G_2.

Claim. Let G_1, G_2 be DAGs and $e_1, e_1' \in E_{G_1}$, $e_2, e_2' \in E_{G_2}$ be edges such that in each G_i ($i \in [2]$) there is a path π_i from $tar_{G_i}(e_i)$ to $src_{G_i}(e_i')$ that contains neither e_i nor e_i'. Then $\pi_1 e_1 \pi_2$ is a path from $tar_H(e_1)$ to $src_H(e_2')$ in $H = (G_1 \& G_2)[e_1 \bowtie e_2']$ (that contains neither e_1 nor e_2').

To see that the claim indeed holds, note that π_i ($i \in [2]$) is also a path in H, by the assumption that the π_i do not involve the swapped edges, and that e_1 points from $src_H(e_1')$, the end vertex of π_1, to $tar_H(e_2)$, the start vertex of π_2. Obviously, $\pi_1 e_1 \pi_2$ contains neither e_1 nor e_2', because neither π_1 nor π_2 does.

Now, returning to our DAG G and its edges e, e', assume that there is a path from $tar_G(e)$ to $src_G(e')$ containing neither e nor e'. By induction on n, applying the claim to $G_1 = G(e \bowtie e')^{n-1}$ and $G_2 = G$ in the inductive step, it follows that $G(e \bowtie e')^n$ has a path that, in particular, contains each of the $n+1$ copies of e'. Consequently, they all belong to the same connected component of $G(e \bowtie e')^n$, which means that this component contains more than $n+1$ vertices as two copies of e' share at most one vertex. This is the situation depicted in Fig. 4.

Now, for $m \in \mathbb{N}$ let $\mathcal{D}_m$ be the set of all DAGs $G \in L$ such that the maximum length of all shortest paths between two vertices is at most m. As R is finite, the degree of vertices of DAGs in L is bounded, and thus $\mathcal{D}_m$ is finite. Let k be the maximum number of edges of all $G \in \mathcal{D}_{2|Q|}$. Then every DAG $G \in L$ with $|E_G| > k$ contains a simple path π of length greater than $2|Q|$. Thus, a run of A on such a DAG G assigns the same state to three distinct edges on π, of which at least one pair e, e' must point into the same direction, which means that there is a path from $tar_G(e)$ to $src_G(e')$ (or vice versa). As we saw above, this implies indeed that $G(e \bowtie e')^n$ contains a connected component C_n such that $|V_{C_n}| > n$, thus completing the proof. □

Looking a bit more closely at the construction of C_n in the proof of Lemma 4.1 it should be obvious that there are subgraphs H_0, H, H_1 of G such that each C_n, $n \geq 1$, consists of $n+1$ components that are linked together by $e_0', \ldots, e_{n-1}'$ (see Fig. 4), namely an initial copy of H_0, $n-1$ copies of H, and a final copy of H_1. It follows that there are constants c, c' such that $|V_{C_n}| = c + nc'$. Hence we obtain as a corollary to (the proof of) Lemma 4.1 that regular DAG languages exhibit the usual linear growth property: if the DAGs in a regular DAG language L are ordered by size, then the gap between two successive ones is bounded by a constant. (Note that finite DAG languages trivially have this property.)

Corollary 4.2 *(Linear growth). For every DAG language $L \in RDL$, if $L = \{G_0, G_1, G_2, \ldots\}$ where $|V_{G_i}| \leq |V_{G_{i+1}}|$ for all $i \in \mathbb{N}$, then there is a constant $k \in \mathbb{N}$ such that $|V_{G_{i+1}}| \leq |V_{G_i}| + k$ for all $i \in \mathbb{N}$.*

Lemma 4.1 requires a DAG of a certain size. However, one can pump up *every* DAG that contains a simple undirected cycle, regardless of its size.

Lemma 4.3 *(Pumping lemma 2). For every DAG language $L \in RDL$ and every DAG $G \in L$ that contains a simple (undirected) cycle there is an edge $e \in E_G$ such that $G(e \bowtie e)^n \in L$ for all $n \in \mathbb{N}$.*

Proof. Let e be an edge which belongs to a simple cycle in G. Since $G \in L$, Lemma 3.3 tells us that $G_n = G(e \bowtie e)^n$ is in $L^{\&}$ for all $n \in \mathbb{N}$. It remains to be shown that G_n is connected. For this, we argue by induction on n that all copies of e in each G_n are part of the same cycle. By the choice of e this is true for G_0. Up to isomorphism we have $G_{n+1} = (G_n \& G)[e' \bowtie e]$, where e' is the relevant copy of e in G_n. Thus the cycle in G_n and the one in G together form a bigger cycle in G_{n+1} that contains all the copies of e. This is illustrated in Fig. 6. Consequently, for all $n \in \mathbb{N}$ the entire DAG G_n is connected. □

Thus no finite DAG language in *RDL* contains a DAG with a simple undirected cycle. On the other hand, all finite DAG languages whose DAGs do not contain any such cycle belong to *RDL*. This yields the following characterization.

Theorem 4.4 *(Characterization of finite DAG languages). Let L be a finite DAG language. Then $L \in RDL$ if and only if there is no DAG $G \in L$ that contains a simple undirected cycle.*

Proof. By Lemma 4.3 and the closedness of *RDL* under union it suffices to show that $\{G\} \in RDL$ for every DAG G that does not contain a simple cycle.

Given such a DAG G, build $A = (Q, \Sigma, R)$ as follows: $Q = E_G$, and for every vertex v of G, R contains the rule $r(v) = in_G(v) \overset{lab(v)}{\longleftrightarrow} out_G(v)$.

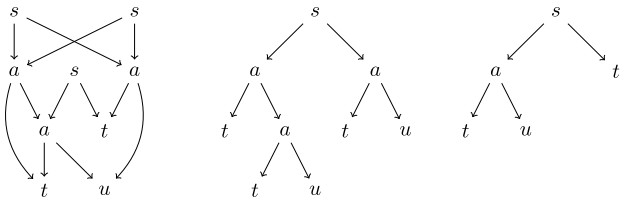


Fig. 7. A DAG G and the tree unfolding $tree_r(G)$ for two different roots r.

We have to show that $L(A) = \{G\}$. For two rules $r(v) = \alpha \overset{\sigma}{\longleftrightarrow} \beta$ and $r(v') = \alpha' \overset{\sigma'}{\longleftrightarrow} \beta'$ we have $[\beta] \cap [\alpha'] \neq \emptyset$ if and only if there is an edge from v to v' in G. Consider now a DAG $G' \in L(A)$ and a run ρ of A on G'. We show that ρ uses every rule in R exactly once. If ρ uses at least one rule more than once then, since G' is connected, there is a $k > 0$ and a sequence $v_0, \ldots, v_k \in V_G$ of rules $r(v_i) = \alpha_i \overset{\sigma_i}{\longleftrightarrow} \beta_i$ $(0 \leq i \leq k)$ such that $v_0, \ldots, v_{k-1}$ are pairwise distinct, $v_0 = v_k$, and $([\beta_{i-1}] \cap [\alpha_i]) \cup ([\alpha_{i-1}] \cap [\beta_i]) \neq \emptyset$ for all $i \in [k]$. But this means that there is a simple cycle from v_0 to v_0 in G, contradicting the assumption. Thus, ρ uses every rule at most once. Moreover we can observe that for every state $q \in Q$ there is exactly one rule such that q occurs in its head and exactly one rule such that q occurs in its tail. Therefore the run ρ must use every rule at least once (as $G' \neq \emptyset$ and G is connected).

This means that we have $|V_{G'}| = |V_G|$ and $|E_{G'}| = |E_G|$, and we can define the following isomorphism (g_V, g_E) from G to G'. Let $v \in V_G$ with $r(v) = e_1 \cdots e_m \overset{\sigma}{\longleftrightarrow} f_1 \cdots f_n$, and assume that v' is the vertex ρ uses $r(v)$ in, where $in_{G'}(v') = e'_1 \cdots e'_m$ and $out_{G'}(v') = f'_1 \cdots f'_n$. Then we let $g_V(v) = v'$, $g_E(e_i) = e'_i$ and $g_E(f_j) = f'_j$ for all $i \in [m]$ and $j \in [n]$. It should be clear that (g_V, g_E) is an isomorphism, which shows that $L(A) = \{G\}$. □

5. From regular DAG languages to regular tree languages

A *tree* can formally be defined as a connected DAG G such that $|in_G(v)| \leq 1$ for all $v \in V_G$. Accordingly, a *tree automaton* is a DAG automaton in which all rules $\alpha \overset{\sigma}{\longleftrightarrow} \beta$ satisfy $|\alpha| \leq 1$. This definition of tree automata is equivalent to the traditional definition of top-down (or bottom-up) tree automata. To see this, recall that, traditionally, a run in such an automaton assigns states to vertices. A rule is of the form $q(\sigma) \to q_1 \cdots q_k$, consisting of a vertex label σ and states $q, q_1, \ldots, q_k$. The intended reading is that if σ is the label of a vertex v having k children and that vertex is assigned the state q, then its children may be assigned the states $q_1, \ldots, q_k$. A tree is accepted if there is a valid run in which the root is assigned one of the so-called initial states of the automaton. In a DAG automaton, a corresponding rule would be $q \overset{\sigma}{\longleftrightarrow} q_1 \cdots q_k$. Moreover, if q is an initial state, then there is the additional rule $\lambda \overset{\sigma}{\longleftrightarrow} q_1 \cdots q_k$. This explicit distinguishability of the root renders initial states unnecessary in DAG automata. The equivalence of both formalisms should be obvious.

Thus, the traditional class *RTL* of regular tree languages is obtained by restricting *RDL* to the languages consisting exclusively of trees, and these are just the DAG languages accepted by tree automata in the above sense. Despite this very direct relationship, we have already seen a classical result that does not carry over from *RTL* to *RDL*: in contrast to *RTL*, *RDL* is not closed under complement (see Theorem 3.4). Moreover, we will see in Section 7 that not all regular DAG languages can be accepted by a deterministic DAG automaton.

We now define how DAGs can be unfolded into trees.

Definition 5.1. Let G be a DAG and r be a root of G. The *tree unfolding* of G with respect to r is defined as $tree_r(G) = (V, E, label, in, out)$ where

- $V = \{v_p \mid v \in V_G \text{ and } p \text{ is a directed path from } r \text{ to } v\}$,
- $E = \{e_p \mid p \text{ is a directed path in } G \text{ from } r \text{ to a vertex } v \neq r\}$

and, for all $v_p \in V$,

- $lab(v_p) = lab_G(v)$,
- $in(v_p) = \begin{cases} e_p & \text{if } p \text{ is nonempty} \\ \lambda & \text{else} \end{cases}$
- $out(v_p) = e_{p\,e_1} \cdots e_{p\,e_n}$, where $e_1, \ldots, e_n$ are the edges such that $out_G(v) = e_1 \cdots e_n$.

The *tree unfolding* of a DAG language L is

$$tree(L) = \{tree_r(G) \mid G \in L \text{ and } r \text{ is a root of } G\}.$$

An example of the tree unfolding of a DAG G is shown in Fig. 7.

$$A = (\{p,q\},\{s,a,t,u\},R) \qquad A_t = (\{p,q\},\{s,a,t,u\},R_t)$$

$$R = \{\ \lambda \overset{s}{\longleftrightarrow} pq,\ \lambda \overset{s}{\longleftrightarrow} qp,\ pq \overset{a}{\longleftrightarrow} qp,\ qp \overset{a}{\longleftrightarrow} pq,\ pp \overset{t}{\longleftrightarrow} \lambda,\ qq \overset{t}{\longleftrightarrow} \lambda,\ pq \overset{u}{\longleftrightarrow} \lambda\ \}$$

$$R_t = \{\ \lambda \overset{s}{\longleftrightarrow} pq,\ \lambda \overset{s}{\longleftrightarrow} qp,\ p \overset{a}{\longleftrightarrow} qp,\ q \overset{a}{\longleftrightarrow} qp,\ q \overset{a}{\longleftrightarrow} pq,\ p \overset{a}{\longleftrightarrow} pq,\ p \overset{t}{\longleftrightarrow} \lambda,\ q \overset{t}{\longleftrightarrow} \lambda,\ p \overset{u}{\longleftrightarrow} \lambda,\ q \overset{u}{\longleftrightarrow} \lambda\ \}$$

Fig. 8. A DAG automaton A and the corresponding tree automaton A_t.

Our aim is to show that the tree unfolding preserves regularity, i.e. if L is a regular DAG language then $tree(L)$ is a regular tree language. Let us say that the *size* of a DAG automaton $A = (Q, \Sigma, R)$ is $|A| = \sum_{r\in R} |r|$ where $|r| = |\alpha\beta| + 1$ for every rule $r = (\alpha \overset{\sigma}{\longleftrightarrow} \beta)$.

Theorem 5.2 *(Unfolding preserves regularity). For every language $L \in RDL$, the tree language $tree(L)$ is a regular tree language. More specifically, for every DAG automaton A, there is a tree automaton A_t with $|A_t| \le |A|^2$, computable in quadratic time, such that $L(A_t) = tree(L(A))$.*

Proof. Consider a DAG automaton $A = (Q, \Sigma, R)$. Without loss of generality, we may assume that A is *reduced* in the sense that it contains only useful rules. Here, a rule $r \in R$ is said to be useful if there exists a DAG G and a run ρ such that ρ uses r in some vertex of G.

Now, let $A_t = (Q, \Sigma, R_t)$ where

$$R_t = \{p_i \overset{\sigma}{\longleftrightarrow} q_1 \cdots q_n \mid (p_1 \cdots p_m \overset{\sigma}{\longleftrightarrow} q_1 \cdots q_n) \in R,\ i \in [m]\} \cup \{(\alpha \overset{\sigma}{\longleftrightarrow} \beta) \in R \mid \alpha = \lambda\}.$$

The construction of A_t is illustrated by an example in Fig. 8. Note that this turns each rule $r = (\alpha \overset{\sigma}{\longleftrightarrow} \beta)$ into $|\alpha + 1| \le |r|$ rules of size at most $|r|$ each, which means that $|A_t| \le |A|^2$.

We claim that $L(A_t) = tree(L(A))$.

We start by proving the straightforward inclusion, namely that $L(A_t) \supseteq tree(L(A))$. Let therefore $G \in L(A)$ and consider a root r of G and a run ρ of A on G. Then we can construct a run ρ_t of A_t on $T = tree_r(G)$ as follows. For every edge $e_p \in E_T$ with $p = p'\hat{e}$ for a path p' and an edge $\hat{e}$ we set $\rho_t(e_p) = \rho(\hat{e})$, i.e. we select the state that ρ assigns to the last edge of the path p. To show that ρ_t is a run consider a vertex $v_p \in V_T$ with $lab(v_p) = \sigma$ and the corresponding vertex $v \in V_G$. Let $out_G(v) = e_1 \cdots e_n$. Then we have $out_T(v_p) = e_{p\,e_1} \cdots e_{p\,e_n}$. If $in_T(v_p) = \lambda$ we also have $in_G(v) = \lambda$ and there is a rule $\lambda \overset{\sigma}{\longleftrightarrow} \rho(e_1) \cdots \rho(e_n) \in R$. By definition of R_t and the choice of ρ_t there is then a rule $(\lambda \overset{\sigma}{\longleftrightarrow} \rho_t(e_{p\,e_1}) \cdots \rho_t(e_{p\,e_n})) \in R_t$. Assume now that $in_T(v_p) = e_p$ for some path p in G. Then we have $\hat{e} \in in_G(v)$ where $\hat{e}$ is the last edge of p. As ρ is a run of A on G this means that there must be a rule $\cdots \rho(\hat{e}) \cdots \overset{\sigma}{\longleftrightarrow} \rho(e_1) \cdots \rho(e_n)$, and thus $(\rho_t(e_p) \overset{\sigma}{\longleftrightarrow} \rho_t(e_{p\,e_1}) \cdots \rho_t(e_{p\,e_n})) \in R_t$. Therefore ρ_t is a run of A_t on T.

It remains to show that $L(A_t) \subseteq tree(L(A))$. Let therefore $T \in L(A_t)$ be a tree and ρ_t be a run of A_t on T. We construct a DAG $C \in L(A)$ such that $T = tree_r(C)$ for a root r of C. For this, consider a vertex $v \in V_T$. If v is not the root of T, then the rule ρ_t uses in v is of the form $p \overset{\sigma}{\longleftrightarrow} q_1 \cdots q_n$. Then there is a corresponding rule $r_v = (\cdots p \cdots \overset{\sigma}{\longleftrightarrow} q_1 \cdots q_n) \in R$ and as A is reduced there is a DAG $G_v \in L(A)$ such that there is a run ρ_v of A on G_v which uses the rule r_v in some vertex $v^* \in V_{G_v}$. Similarly, in the root z of T, ρ_t uses a rule in R, and thus there is a DAG G_z and a run ρ_z of A on G_z that uses this rule in some vertex z^*.

We take the DAGs G_v $(v \in V_T)$ and connect them through swaps of some edges incident to the vertices v^* in order to create a DAG G such that $tree_{z^*}(G) = T$. To see how this can be done, note that each ρ_v uses a rule in v^* that has the correct tail, i.e. ρ_v assigns just those states to the edges leaving v^* which ρ_t assigns to the edges leaving v. Since this holds for all v we only have to redirect the ith edge leaving v^* to w^*, where w is the ith child of v in T. To keep the combined run on the DAGs G_v intact, this is done by edge swaps, as follows.

Let $e \in E_T$ have the source v and the target w, where e is the ith outgoing edge of v. Since the tails of the rules used by ρ_t and ρ_v in v and v^*, resp., are identical, the ith outgoing edge e^* of v^* in G_v satisfies $\rho_t(e) = \rho_v(e^*)$. Further, ρ_w uses a rule in w^* such that $\rho_t(e)$ occurs in the head of that rule. In other words, $in_{G_w}(w^*)$ contains an edge ${}^*e$ such that $\rho_w({}^*e) = \rho_t(e) = \rho_v(e^*)$. It follows that the edge swap of e^* and ${}^*e$ in G_v and G_w is well defined.

If $V_T = \{v_0, \ldots, v_k\}$ and $E_T = \{e_1, \ldots, e_k\}$, define $G = (G_{v_0} \& \ldots \& G_{v_k})[e_1^* \bowtie {}^*e_1] \ldots [e_k^* \bowtie {}^*e_k]$. As argued in the previous paragraph G is well defined, and Lemma 3.3 yields that $G \in L(A)^{\&}$.

The preceding construction is illustrated in Fig. 9. It shows the construction of G from a tree $T \in L(A_t)$ with $tree_{z^*}(G) = T$ for a root z and the automata A and A_t from Fig. 8.

Consider now a vertex $v \in V_T$ and the ith child v_i of v. Then the i-th child of v^* in G is v_i^*. Moreover by construction we have $lab_T(v) = lab_G(v^*)$ and $lab_T(v_i) = lab_G(v_i^*)$. Thus, if C is the connected component of G that contains the vertex z^*, then we have $C \in L(A)$ and $T = tree_{z^*}(C)$. This shows that $L(A_t) \subseteq tree(L(A))$ and completes the proof. □

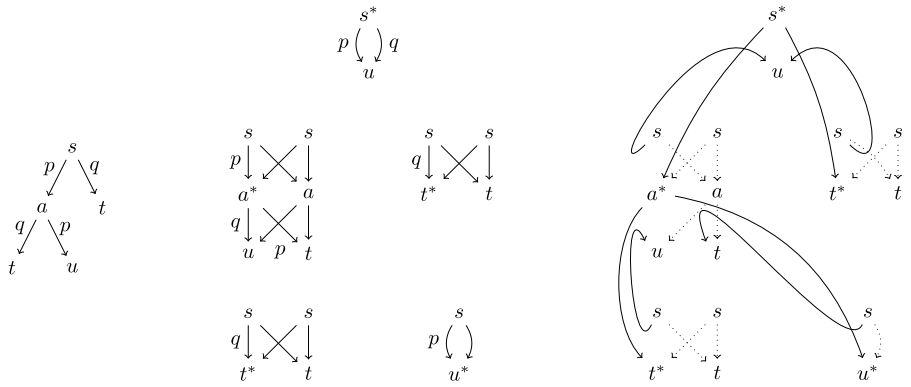


Fig. 9. Construction of a DAG G from a tree T with $T = Tree_{z^*}(G)$ for some root z^* (labelled by s). Asterisks are added to certain vertex labels to indicate the vertices v^*; they are not part of the labels.

Theorem 5.2 allows us to draw further conclusions about the properties of regular DAG languages, for instance about path languages. The *path language* of a DAG G consists of all sequences of symbols that occur on a root-to-leaf path in G. Formally,

$$paths(G) = \{lab(v_0) \cdots lab(v_n) \mid n \in \mathbb{N}, v_0, \ldots, v_n \in V_G, v_0 \text{ is a root and } v_n \text{ is a leaf of } G, \text{ and for every } i \in [n] \text{ there is } e \in E_G \text{ with } src_G(e) = v_{i-1} \text{ and } tar_G(e) = v_i\}.$$

The path language of a DAG language L is defined accordingly as

$$paths(L) = \bigcup_{G \in L} paths(G).$$

It is well known that the path language of a regular tree language is a regular string language. Obviously, for a DAG G and a string p we have $p \in paths(G)$ if and only if $p \in paths(tree_z(G))$ for some root z of G. To perform the construction in the proof of Theorem 5.2 effectively, we have to turn a DAG automaton into a reduced one. It will be shown in Section 6 that this can in fact be done by identifying, in polynomial time, the useless rules of the automaton and discarding them. Together with this, Theorem 5.2 yields the following corollary, which was also shown in [6], but without the second part, using a non-constructive argument based on the Myhill–Nerode theorem.

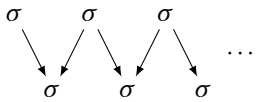
Corollary 5.3 *(Regularity of the path language). For every DAG automaton A the string language $paths(L(A))$ is a regular string language. A finite automaton accepting $paths(L(A))$ can be constructed from A in polynomial time.*

Trivially, $paths(L(A))$ is empty if and only if $L(A)$ is. Hence the previous corollary, together with the well-known polynomial decidability of the emptiness problem for finite automata, yields a polynomial emptiness test, which is another result that was shown by means of a direct proof in [6]:

Corollary 5.4 *(Polynomial decidability of emptiness). For DAG automata A as input, it is decidable in polynomial time whether $L(A) = \emptyset$.*

6. The finiteness problem

The finiteness problem for DAG automata asks whether the language $L(A)$ accepted by a given DAG automaton A is finite. We show that this question can be answered in polynomial time as well. At first sight, it may seem to be a consequence of the results of the previous section, just like Corollary 5.4. However, a DAG language L can be infinite even if $tree(L)$ is finite. For example, it may consist of all DAGs of the form



As we have already seen in Theorem 4.4, the infiniteness of a regular DAG language L may arise from a cycle in a DAG $G \in L$. We can observe that the rules used by a corresponding automaton along that cycle are also organized in a cyclic

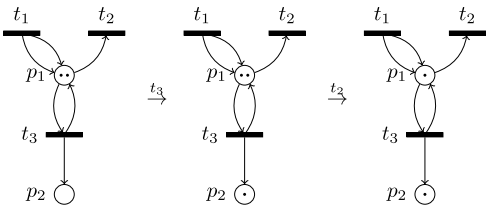


Fig. 10. A firing sequence $\phi_1 \xrightarrow{t_3} \phi_2 \xrightarrow{t_2} \phi_3$ of a Petri net N.

$A = (Q, \Sigma, R)$ with
$R = \{r_1, r_2, r_3, r_4\}$ for

$r_1 = \lambda \xleftrightarrow{s} pq$
$r_2 = p \xleftrightarrow{a} qq$
$r_3 = q \xleftrightarrow{b} pp$
$r_4 = qp \xleftrightarrow{t} \lambda$

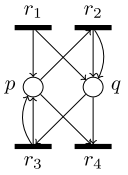


Fig. 11. A DAG automaton A and the corresponding Petri net N_A.

manner. Similar to the finiteness test for regular string or tree automata, it suffices to identify cycles in the rules of a given DAG automaton A in order to decide if $L(A)$ is infinite.

One may however only consider rules that are *useful* in the sense that they are used by the DAG automaton in some run. Therefore we now show that any given DAG automaton can be turned into an equivalent *reduced* one that contains only useful rules. Formally, a rule r of a given DAG automaton A is *useful* if and only if there is a run ρ of A on some DAG G such that ρ uses r in some vertex of G whereas A is *reduced* if and only if each of its rules is useful. Extending the idea that Chiang et al. [6] used to prove Corollary 5.3, we will make use of Petri nets in order to determine which rules are useful and which can be deleted.

Recall that a Petri net is a bipartite unlabelled graph[2] $N = (V, E, in, out)$, where $V = P \cup T$ for disjoint sets P and T of *places* and *transitions*, and $src_N(e) \in P$ iff $tar_N(e) \in T$ for all $e \in E$.[3] A *configuration of* N is a function $\phi\colon P \to \mathbb{N}$ where $\phi(p)$ is interpreted as the number of *tokens* on p.

For $(u, v) \in (P \times T) \cup (T \times P)$ let $\Delta(u, v) = |[out_N(u)] \cap [in_N(v)]|$ be the number of edges from u to v. Given a configuration ϕ, a transition t is *enabled* for a place p if $\phi(p) \geq \Delta(p, t)$. The transition is enabled if it is enabled for all places. If t is enabled, it can *fire* by consuming $\Delta(p, t)$ tokens from p and adding $\Delta(t, p)$ tokens to p, for every place p. In other words, the firing of t leads from ϕ to ϕ' defined by $\phi'(p) = \phi(p) - \Delta(p, t) + \Delta(t, p)$ for all $p \in P$. We denote this as $\phi \xrightarrow{t} \phi'$.

Fig. 10 shows an example of a firing sequence of a Petri net where the transitions and places are drawn in the usual way as bars and circles, resp. and the tokens are displayed as dots within the different places. In the first configuration ϕ_1 the places p_1 and p_2 hold two and zero tokens respectively. Then transition t_3 fires which consumes one token from p_1 and adds one token to p_1 and p_2 each. Finally t_2 fires which consumes one token from p_1.

As observed in [6], every DAG automaton $A = (Q, \Sigma, R)$ gives rise to a Petri net $N_A = (P \cup T, E, in, out)$ where $P = Q$, $T = R$ and for every rule $r = p_1 \cdots p_m \xleftrightarrow{\sigma} q_1 \cdots q_n$ there is an edge from every p_i to r ($i \in [m]$) and one from r to every q_i ($i \in [n]$). Fig. 11 shows an example of a simple DAG automaton A and the corresponding Petri net N_A.

The idea behind this construction is that, if we view a run as a top-down process, using r can be seen as an action that consumes states $p_1, \ldots, p_m$ and produces states $q_1, \ldots, q_n$. Every run ρ of A on a DAG G that uses the rules $r_1, \ldots, r_k$ in a top-down fashion gives thus rise to a firing sequence $\phi_0 \xrightarrow{r_1} \ldots \xrightarrow{r_k} \phi_k$ in which the initial configuration has no states at all, i.e. ϕ_0 is the *null configuration* $\mathbf{0}$ with $\mathbf{0}(p) = 0$ for all $p \in P$. Since the run uses a rule in each vertex of G, all tokens are eventually consumed, i.e. we have $\phi_k = \mathbf{0}$ as well. We call such a (nonempty) firing sequence a *zero cycle*. Fig. 12 shows an example of a run ρ of the DAG automaton A from Fig. 11 on a DAG G and a corresponding valid firing sequence of the Petri net N_A.

The converse of the above also holds, namely that, given a valid zero cycle $\phi_0 \xrightarrow{r_1} \ldots \xrightarrow{r_k} \phi_k$ of the Petri net N_A, we can construct a run ρ of A on some DAG G. The construction starts with the empty DAG, adding vertices $v_1, \ldots, v_k$ one after another. The intermediate DAGs constructed may be incomplete in the sense that they have a number of dangling edges. More precisely, before v_i is added, for every $p \in P$ there are exactly $\phi_{i-1}(p)$ dangling edges that are marked with state p. If $r_i = p_1 \ldots p_m \xleftrightarrow{\sigma} q_1 \ldots q_n$ ($i \in [k]$) a vertex v_i with label σ and n outgoing edges $e_1, \ldots, e_n$ is added and we set $\rho(e_i) = q_i$ for $i \in [n]$. The incoming edges of v_i are chosen from the set of previously dangling edges, marked with $p_1, \ldots, p_m$. Such edges are guaranteed to exist because we know that r_i could fire. Since $\phi_k = \mathbf{0}$ we created a run ρ on a valid DAG G after

[2] An unlabelled graph is a $\{\bullet\}$-graph in the sense of Definition 2.1, where the component *lab* is irrelevant and thus dropped because $\bullet$ is the only label.
[3] Recall the definitions of $src_N(e)$ and $tar_N(e)$ from the paragraph below Definition 2.1.

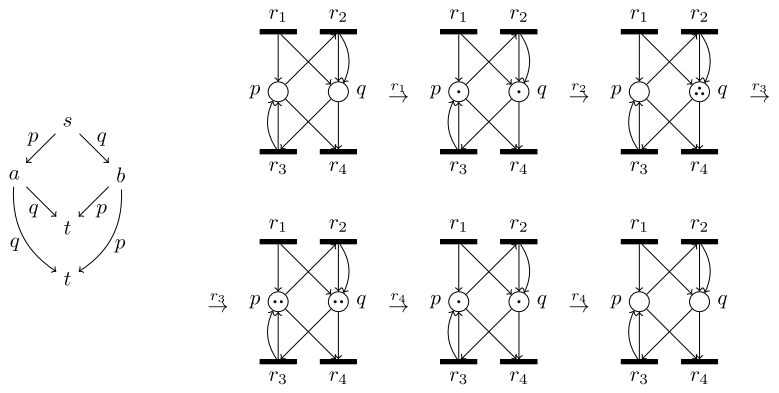


Fig. 12. A run and a corresponding firing sequence.

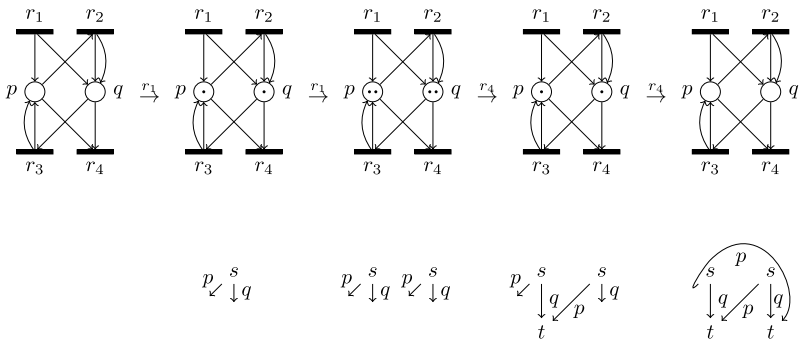


Fig. 13. Construction of a graph G from a firing sequence $\mathbf{0} \xrightarrow{r_1} \dots \xrightarrow{r_k} \mathbf{0}$ of N_A.

step k, i.e. there are no dangling edges left. Note however that G is neither uniquely determined nor necessarily connected. An example for the described construction is illustrated in Fig. 13.

A rule of a given DAG automaton A is therefore useful if and only it occurs in a zero cycle of N_A. In other words, an equivalent reduced DAG automaton A_{red} can be constructed by keeping only those rules that occur in a zero cycle of N_A. This set is denoted by $\Lambda(N_A)$ in [9], where it was shown to be computable in polynomial time; see [9, Theorem 6.2]. This leads to the following lemma.

Lemma 6.1. *For every DAG automaton A a reduced DAG automaton A_{red} with $L(A) = L(A_{red})$ can be computed in polynomial time.*

Having shown how rules that are not useful can be eliminated we can now turn back to the original problem, namely deciding whether a given DAG accepts a finite language. As previously mentioned we will do this by identifying cycles in the automaton. The definition of a cycle makes use of the notion of *marked rules*. For a DAG automaton $A = (Q, \Sigma, R)$ a *marked rule* is a triple $\hat{r} = (r, i, j)$ such that

1. $r = (\alpha \overset{\sigma}{\longleftrightarrow} \beta)$ is a rule in R and
2. $i, j \in [|\alpha\beta|]$ are distinct numbers, called the *entry* and the *exit* of $\hat{r}$.

The states at positions i and j in $\alpha\beta$ are called the *entry state* and *exit state* of $\hat{r}$, resp. Moreover, we say that $\hat{r}$ is *head entered* if $i \in [|\alpha|]$ and *tail entered* otherwise. Similarly, it is *head exited* if $i \in [|\alpha|]$ and *tail exited* otherwise.

We can now define what a cycle in A is. To simplify the notation in both this definition and the rest of the section, in the presence of an index $k > 0$ clear from the context, we let $\mathrm{succ}(i) = (i \bmod k) + 1$, i.e. we count modulo k.

Definition 6.2. Let $A = (Q, \Sigma, R)$ be a DAG automaton. A *cycle* in A is a nonempty sequence of marked rules $\hat{r}_1, \dots, \hat{r}_k$ of A such that, for all $i \in [k]$,

1. the exit state of $\hat{r}_i$ is equal to the entry state of $\hat{r}_{\mathrm{succ}(i)}$ and
2. $\hat{r}_i$ is tail exited if and only if $\hat{r}_{\mathrm{succ}(i)}$ is head entered.

A is called *cyclic* if and only there is some cycle in A.

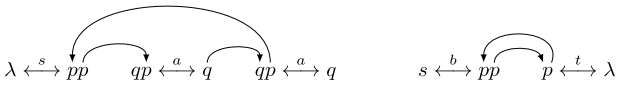


Fig. 14. Examples of the definition of a cycle in a DAG automaton.

The intuition is that a cycle is a sequence of rules in which each rule overlaps with the succeeding one in a cyclic fashion, i.e. modulo k. Given a cycle $\hat{r}_1, \ldots, \hat{r}_k$ in a DAG automaton A, we will construct a run ρ of A on some DAG G which uses the rules $r_1, \ldots, r_k$ along an undirected path $e_1 \cdots e_k$ of G and assigns the exit state of $\hat{r}_i$ (which, at the same time, is the entry state of $\hat{r}_{\mathrm{succ}(i)}$) to the edge e_i. The edges e_1 and e_k will moreover have the same orientation, so the DAG G can be pumped up as we saw in the proof of Lemma 4.1.

Example 6.3. To illustrate this consider the DAG $A = (Q, \Sigma, R)$ with $R = \{r_1, \ldots, r_4\}$ where

$$\begin{aligned} r_1 &= \lambda \xleftrightarrow{s} pp, \\ r_2 &= qp \xleftrightarrow{a} q, \\ r_3 &= p \xleftrightarrow{b} q, \quad \text{and} \\ r_4 &= p \xleftrightarrow{t} \lambda. \end{aligned}$$

This DAG automaton contains a cycle as shown in Fig. 14 on the left, where the directed arcs point from exits to entries. On the right, however, no valid cycle is shown because the entry of the second marked rule must not coincide with its exit.

We now show that a reduced DAG automaton accepts a finite language if and only if it does not contain any cycle.

Lemma 6.4. *Let A be a reduced DAG automaton. Then $L(A)$ is finite if and only if A is not cyclic.*

Proof. Let $A = (Q, \Sigma, R)$, and assume first that $L(A)$ is infinite. Then there is no bound on the length of (undirected) paths in DAGs in $L(A)$. It follows that there is a run ρ on some DAG G and there is a simple undirected path $\pi = e_1 \cdots e_k \in E_G$ for some $k > 1$ connecting vertices $v_0, \ldots, v_k$ such that

(a) $\rho(e_1) = \rho(e_k)$ and
(b) $v_0 = src_G(e_1)$ if and only if $v_{k-1} = src_G(e_k)$, i.e. e_1 and e_k have the same orientation on π.

Such a path π exists because Q is finite and there are only two possible orientations of edges; cf. the last paragraph of the proof of Lemma 4.1.

Let r_i be the rule used by ρ in v_i, for all $i \in [k-1]$. Define the marked rule $\hat{r}_i = (r_i, j, j')$ such that j and j' are the positions at which e_i and e_{i+1} occur in $in_G(v_i)out_G(v_i)$. Then $j \neq j'$ because π is a simple path. Moreover, the exit state of $\hat{r}_i$ is equal to the entry state of $\hat{r}_{\mathrm{succ}(i)}$, and $\hat{r}_i$ is tail exited if and only if $\hat{r}_{\mathrm{succ}(i)}$ is head entered. (For $i = k-1$, this is true by (a) and (b).) Hence, A is cyclic.

For the other direction assume that A is cyclic. Let $\hat{r}_1, \ldots, \hat{r}_k$ be a cycle in A, where p_i ($i \in [k]$) is the exit state of $\hat{r}_i$ (and thus the entry state of $\hat{r}_{\mathrm{succ}(i)}$), and ℓ_i, ℓ_i' are the entry and the exit of r_i, resp. As A is reduced there is a DAG $G_i \in L(A)$ for every $i \in [k]$ such that there exists a run ρ_i of A on G_i which uses r_i in a vertex $v_i \in V_{G_i}$. Let e_i and e_i' be the ℓ_i-th and the ℓ_i'-th edge in $in_{G_i}(v_i)out_{G_i}(v_i)$. Then $\rho_i(e_i') = p_i = \rho_{\mathrm{succ}(i)}(e_{\mathrm{succ}(i)})$, and we can show the following by induction on i:

Claim. For a simple nonempty path $f_1 \cdots f_h$ from u to v in a DAG H, say that f_h is reversed if $src_H(f_h) = v$. For all $i \in [k]$, the DAG $(G_1 \,\&\, \cdots \,\&\, G_i)[e_1' \bowtie e_2] \cdots [e_{i-1}' \bowtie e_i]$ contains a path of length $i+1$ that ends in e_i'. Furthermore, e_i' is reversed if and only if $\hat{r}_i$ is head exited.

The claim is true for $i = 1$, because G_1 contains the path $e_1 e_1'$, in which e_1' is reversed if and only if $\hat{r}_1$ is head exited. Now, for $i > 1$, assume that $H = (G_1 \,\&\, \cdots \,\&\, G_{i-1})[e_1' \bowtie e_2] \cdots [e_{i-2}' \bowtie e_{i-1}]$ contains a path π of the form claimed, and consider $H' = (H \,\&\, G_i)[e_{i-1}' \bowtie e_i]$. We have two cases: Either $\hat{r}_{i-1}$ is tail exited and $\hat{r}_i$ is head entered or $\hat{r}_{i-1}$ is head exited and $\hat{r}_i$ is tail entered. In the first case, $tar_{G_i}(e_i) = v_i$ and thus $tar_{H'}(e_{i-1}') = v_i$. Since e_{i-1}' is not reversed in π, redirecting it to v_i in H' does not destroy the property of being a path. Hence, π is a path to v_i in H', and since $v_i \in \{src_{G_i}(e_i'), tar_{G_i}(e_i')\}$ this means that $\pi e_i'$ is a path in H'. Moreover, e_i' is reversed if and only if $tar_{H'}(e_i') = tar_{G_i}(e_i') = v_i$, if and only if $\hat{r}_i$ is head exited. The second case is similar: we have $src_{G_i}(e_i) = v_i$. Since e_{i-1}' is reversed in $\pi = \pi_0 e_{i-1}'$, the path $\pi_0 e_i$ is a path to v_i in H'. As in the previous case, appending e_i' to this path yields the claimed path ending in e_i', where e_i' is reversed if and only if $\hat{r}_i$ is head exited. This completes the proof of the claim.

Consequently, $G = (G_1 \,\&\, \cdots \,\&\, G_k)[e_1' \bowtie e_2] \cdots [e_{k-1}' \bowtie e_k]$ contains a path of length $k+1$. However, by the definition of cycles in DAG automata, appending a cycle to itself yields a longer cycle, which means that there is no bound on the lengths of paths in the DAGs in $L(A)$. Therefore $L(A)$ is infinite. □

Using the previous two lemmata we obtain the main result of this section.

Theorem 6.5 *(Polynomial decidability of finiteness). The finiteness problem for DAG automata is decidable in polynomial time.*

Proof. Given an arbitrary DAG automaton A we can decide if $L(A)$ is finite by constructing an equivalent reduced DAG automaton A_{red} and checking if A_{red} is cyclic. According to Lemma 6.1 the construction of A_{red} can be performed in polynomial time and it should be clear that the same holds for checking whether A_{red} is cyclic. □

7. Deterministic DAG automata, minimization, and equivalence

The DAG automaton in our very first example (Example 2.5) is in fact *top-down deterministic*, meaning that for all $\alpha \in Q^*$, $\sigma \in \Sigma$, and $n \in \mathbb{N}$ there is at most one $\beta \in Q^*$ with $|\beta| = n$ such that $\alpha \overset{\sigma}{\longleftarrow} \beta$ is a rule in A. In such a case, an input DAG permits at most one run, and this run can be constructed deterministically from the roots downward. A dual notion of bottom-up determinism is obtained by using the dual requirement instead: for all $\beta \in Q^*$, $\sigma \in \Sigma$, and $m \in \mathbb{N}$ there is at most one $\alpha \in Q^*$ with $|\alpha| = m$ such that $\alpha \overset{\sigma}{\longleftarrow} \beta$ is a rule in A. Regular DAG languages that can be accepted by (top-down or bottom-up) deterministic DAG automata are said to be (top-down or bottom-up) deterministically regular.

It is well known that the class of top-down deterministic regular *tree* languages is strictly included in the class of regular tree languages, the standard example being the finite tree language $L_0 = \{f(a,b), f(b,a)\}$ (where trees are denoted as terms). Clearly, a top-down deterministic DAG automaton accepting a tree language is also deterministic if viewed as a top-down finite-state tree automaton. Consequently, it follows from the fact that a tree language is regular if and only if it is a regular DAG language, that top-down deterministic DAG automata fail to accept all regular DAG languages. By duality the very same result holds in the bottom-up case, in contrast to the fact that bottom-up deterministic finite-state tree automata are equally powerful as nondeterministic ones. In fact, looking a bit more closely at the argument, the entire class of deterministically regular DAG languages is strictly included in *RDL*. This is because, for a DAG automaton A, removing all rules $\alpha \overset{\sigma}{\longleftarrow} \beta$ with $|\alpha| > 1$ yields a DAG automaton that accepts $\{G \in L(A) \mid G \text{ is a tree}\}$. As this construction preserves top-down determinism, it implies (together with its dual) that the regular DAG language $L_0 \cup L_0^{\updownarrow}$ is not deterministically regular at all.

Observation 7.1. There exist finite regular DAG languages that are not deterministically regular.

In the remainder of this section we will show that deterministic DAG automata can be minimized. As in the case of string and tree automata, it turns out that minimal deterministic DAG automata are unique up to state renaming, which implies that the equivalence problem is decidable. By the duality principle of DAG automata, it suffices to consider top-down deterministic DAG automata. The result then automatically follows for bottom-up deterministic ones. Furthermore, it suffices to consider DAG automata over ranked alphabets, because a DAG automaton over an unranked alphabet can be turned into one over a ranked alphabet by turning every rule $p_1 \cdots p_m \overset{\sigma}{\longleftarrow} q_1 \cdots q_n$ into $p_1 \cdots p_m \overset{\sigma_{m,n}}{\longleftarrow} q_1 \cdots q_n$. In the rest of this section, we therefore assume that this transformation is implicitly made by distinguishing between symbols that differ only by their ranks, thus in effect working as if the considered DAG automata were using a ranked alphabet.

As usual, the idea is to partition the states of the DAG automaton into sets of states that are equivalent in the sense that they behave identically within the automaton. If two states p and p' are equivalent we can shrink the automaton by replacing p and p' with one single state that behaves like both p and p' do. Also as usual, we determine which states are equivalent by discovering distinguishable pairs of states. These are defined as follows.

Definition 7.2 *(Distinguishable state pair).* Let $A = (Q, \Sigma, R)$ be a reduced top-down deterministic DAG automaton. The set of all *distinguishable* pairs of states is defined inductively to be the smallest symmetric binary relation $\Delta_A \subseteq Q^2$ such that $(p, p') \in \Delta_A$ if there exist $\sigma \in \Sigma$ and $\alpha, \alpha' \in Q^*$ such that one of the following hold:

1. R contains a σ-rule with the head $\alpha\, p\, \alpha'$ but no σ-rule with the head $\alpha\, p' \alpha'$, or
2. there are rules $\alpha\, p\, \alpha' \overset{\sigma}{\longleftarrow} q_1 \cdots q_n$ and $\alpha\, p' \alpha' \overset{\sigma}{\longleftarrow} q_1' \cdots q_n'$ in R such that $(q_j, q_j') \in \Delta_A$ for some $j \in [n]$.

A *witness* for the fact that $(p, p') \in \Delta_A$ (a *witness for* (p, p'), for short) is a nonempty sequence over $Q^* \times (Q \times \Sigma) \times Q^*$ defined as follows: In the first case above $(\alpha, p{:}\sigma, \alpha')$ is a witness; in the second case, if ω is a witness for $(q_j, q_j') \in \Delta_A$, then $(\alpha, p{:}\sigma, \alpha') \cdot \omega$ is a witness for $(p, p') \in \Delta_A$.

Example 7.3. Consider the DAG automaton $A = (\{p, p', q, {}'q\}, \{s, a, t\}, R)$ where

$$
\begin{aligned}
R = \{ \quad \lambda &\overset{s}{\longleftarrow} pq, \\
p &\overset{a}{\longleftarrow} p', \\
q &\overset{a}{\longleftarrow} q', \\
p'p' &\overset{t}{\longleftarrow} \lambda, \\
q'q' &\overset{t}{\longleftarrow} \lambda \quad \}.
\end{aligned}
$$

Then $(p', q') \in \Delta_A$ because we have $p'p' \xleftrightarrow{t} \lambda \in R$ but R contains no t-rule with the head $p'q'$. Hence, $(p', p'{:}t, \lambda)$ is a witness for $(p', q') \in \Delta_A$. In turn, this means that $(p, q) \in \Delta_A$ as well, because there are rules $(p \xleftrightarrow{a} p') \in R$ and $(q \xleftrightarrow{a} q') \in R$ and p' and q' are distinguishable. The corresponding witness is $(\lambda, p{:}a, \lambda)(p', p'{:}t, \lambda)$.

In order to construct the minimal automaton $A_{\min}$ we will compute all distinguishable pairs of the automaton A. This can be done in polynomial time by a standard iterative discovery algorithm, as follows:

1. Mark all pairs (p, p') and (p', p) such that (p, p') satisfies the first condition of Definition 7.2.
2. Mark all pairs (p, p') such that, for some already marked pair (q_j, q'_j), the second condition of Definition 7.2 is satisfied.
3. Repeat the previous step as long as at least one new pair is discovered.
4. Return the set of all marked pairs as Δ_A.

Intuitively if p and p' are distinguishable, then they behave differently in the automaton. Conversely we will see later that two states p and p' behave identically if they are not distinguishable. Therefore we formally define the equivalence of states as follows.

Definition 7.4 *(Equivalent states).* Let $A = (Q, \Sigma, R)$ be a reduced top-down deterministic DAG automaton. States $p, p' \in Q$ are *equivalent*, $p \equiv p'$, if $(p, p') \notin \Delta_A$.

The $\equiv$-relation is an equivalence relation as the following lemma shows.

Lemma 7.5. *The relation $\equiv$ is an equivalence relation.*

Proof. We need to show that $\equiv$ is reflexive, symmetric and transitive. Reflexivity and symmetry follow directly from Definitions 7.2 and 7.4, because Δ_A is irreflexive and symmetric. For transitivity let $p \equiv q$ and $q \equiv r$. W.l.o.g. we show that there is no witness for the pair (p, r). We proceed by induction on the length $\ell \geq 1$ of witnesses to show that no such witness can exist.

Assume that $(\alpha, p{:}\sigma, \alpha')$ is a witness for (p, r). Then there is a σ-rule with head $\alpha\, p\, \alpha'$ in R but none with head $\alpha\, r\, \alpha'$. We have however $p \equiv q$ which means that R does contain a σ-rule with head $\alpha\, q\, \alpha'$. In turn, using the fact that $q \equiv r$, R also contains a σ-rule with head $\alpha\, r\, \alpha'$, a contradiction.

Assume now that there are no $p', q', r' \in Q$ with $p' \equiv q'$ and $q' \equiv r'$ such that (p', r') has a witness of length ℓ, but that $(\alpha_{\ell+1}, p{:}\sigma_{\ell+1}, \alpha'_{\ell+1}) \cdot \omega$ is a witness of length $\ell + 1$ for (p, r). This means that there are rules $\alpha_{\ell+1}\, p\, \alpha'_{\ell+1} \xleftrightarrow{\sigma_{\ell+1}} q_1 \cdots q_n$ and $\alpha_{\ell+1}\, r\, \alpha'_{\ell+1} \xleftrightarrow{\sigma_{\ell+1}} q'_1 \cdots q'_n$ in R such that, for some $j \in [n]$, ω is a witness for $(q_j, q'_j) \in \Delta_A$. As we have $p \equiv q$ and $q \equiv r$ there is a rule $\alpha_{\ell+1}\, q\, \alpha'_{\ell+1} \xleftrightarrow{\sigma_{n+1}} q''_1 \cdots q''_n$ such that there are no witnesses for (q_j, q''_j) or (q''_j, q'_j) for any j, which means that $q_j \equiv q''_j$ and $q''_j \equiv q'_j$ for all $j \in [n]$. Accordingly, by the induction hypothesis, there is no witness of length ℓ for any of the pairs (q_j, q'_j), contradicting the assumption that ω is such a witness. □

This means that $\equiv$ partitions the state set Q of A into equivalence classes. We show that we can minimize A by replacing every state p with its equivalence class $\langle p \rangle = \{q \in Q \mid p \equiv q\}$. Formally we define the DAG automaton $A_{\min}$ obtained from A as follows.

Definition 7.6. Let $A = (Q, \Sigma, R)$ be a reduced top-down deterministic DAG automaton. We let $A_{\min} = (Q_{\min}, \Sigma, R_{\min})$ where

$$\begin{aligned} Q_{\min} &= \{\langle p \rangle \mid p \in Q\} \text{ and} \\ R_{\min} &= \{\langle p_1 \rangle \cdots \langle p_m \rangle \xleftrightarrow{\sigma} \langle q_1 \rangle \cdots \langle q_n \rangle \mid (p_1 \cdots p_m \xleftrightarrow{\sigma} q_1 \cdots, q_n) \in R\}. \end{aligned}$$

Obviously $A_{\min}$ can be constructed in polynomial time from A. However, we have to show that A and $A_{\min}$ are actually equivalent, i.e. that $L(A) = L(A_{\min})$. This is done in the following lemma.

Lemma 7.7. *For every reduced top-down deterministic DAG automaton A we have $L(A) = L(A_{\min})$.*

Proof. Let $A = (Q, \Sigma, R)$ be a reduced top-down deterministic DAG automaton. For a run ρ of A on a DAG G, $\rho_{\min}$ defined by $\rho_{\min}(e) = \langle \rho(e) \rangle$ for all $e \in E_G$ is obviously a run of $A_{\min}$ on G. It follows that $L(A) \subseteq L(A_{\min})$.

It remains to be shown that $L(A_{\min}) \subseteq L(A)$. For this, we prove the following claim:

Claim. If $(\langle p^*_1 \rangle \cdots \langle p^*_m \rangle \xleftrightarrow{\sigma} \langle q^*_1 \rangle \cdots \langle q^*_n \rangle) \in R_{\min}$ then for all $p_1 \in \langle p^*_1 \rangle, \ldots, p_m \in \langle p^*_m \rangle$ there are $q_1 \in \langle q^*_1 \rangle, \ldots, q_n \in \langle q^*_n \rangle$ such that $(p_1 \cdots p_m \xleftrightarrow{\sigma} q_1 \cdots q_n) \in R$.

The assumptions of the claim readily imply that there are states $p'_1 \in \langle p^*_1 \rangle, \ldots, p'_m \in \langle p^*_m \rangle$ and $q'_1 \in \langle q^*_1 \rangle, \ldots, q'_n \in \langle q^*_n \rangle$ such that $(p'_1 \cdots p'_m \xleftarrow{\sigma} q'_1 \cdots q'_n) \in R$. As p_1 and p'_1 are equivalent there is a rule $(p_1 p'_2 \cdots p'_m \xleftarrow{\sigma} q''_1 \cdots q''_n) \in R$ such that $q''_1 \in \langle q^*_1 \rangle, \ldots, q''_n \in \langle q^*_n \rangle$. In the same way we can iteratively replace the remainder $p'_2 \cdots p'_m$ of the head with $p_2 \cdots p_m$ and obtain a rule of the desired form. This proves the claim.

Consider now a DAG $G \in L(A_{\min})$ and let $\rho_{\min}$ be a run of $A_{\min}$ on G. We iteratively construct a run ρ of A on G by traversing G in a top-down manner. The run will satisfy $\rho(e) \in \langle \rho_{\min}(e) \rangle$ for all $e \in E_G$. For every root v of G we define $\rho(v)$ as follows. Let $lab_G(v) = \sigma$ and $out_G(v) = f_1 \cdots f_n$, and assume that $\lambda \xleftarrow{\sigma} \langle q^*_1 \rangle \cdots \langle q^*_n \rangle$ is the rule used by $\rho_{\min}$ in v. Then there is a rule $(\lambda \xleftarrow{\sigma} q_1 \cdots q_n) \in R$ with $q_i \in \langle q^*_i \rangle$ for $i \in [n]$ that ρ can use in v, and we choose $\rho(f_i) = q_i$ for $i \in [n]$.

Consider now a vertex $v \in V_G$ with $in_G(v) = e_1 \cdots e_m$ such that $\rho(e_i)$ is already defined for every incoming edge e_i. (Such a vertex must exist because G is acyclic.) Let $lab_G(v) = \sigma$ and $out_G(v) = f_1 \cdots f_n$, and let $\langle p^*_1 \rangle \cdots \langle p^*_m \rangle \xleftarrow{\sigma} \langle q^*_1 \rangle \cdots \langle q^*_n \rangle$ be the rule that $\rho_{\min}$ uses in v. We have $\rho(e_i) \in \langle p^*_i \rangle$ for $i \in [m]$. By the claim proved above, this means that R contains a rule $\rho(e_1) \cdots \rho(e_m) \xleftarrow{\sigma} q_1 \cdots q_n$ with $q_i \in \langle q^*_i \rangle$ for all $i \in [n]$. Thus, we can choose $\rho(f_i) = q_i$ for $i \in [n]$, which eventually yields a complete run of A on G, finishing the proof that $G \in L(A)$. □

From the previous proof it also follows that the minimal DAG automaton $A_{\min}$ is reduced. Indeed, by construction a rule $r_{\min} = (\langle p^*_1 \rangle \cdots \langle p^*_m \rangle \xleftarrow{\sigma} \langle q^*_1 \rangle \cdots \langle q^*_n \rangle)$ cannot be in $R_{\min}$ unless R contains a rule $r = (p_1 \cdots p_m \xleftarrow{\sigma} q_1 \cdots q_n)$ with $p_i \in \langle p^*_i \rangle$ and $q_j \in \langle q^*_j \rangle$ for $i \in [m]$ and $j \in [n]$. As A is reduced there is a run ρ of A on some DAG G such that ρ uses r in a vertex $v \in V_G$. The run constructed in the proof of the inclusion $L(A) \subseteq L(A_{\min})$ thus uses $r_{\min}$ in v. Note furthermore that every state $\langle p \rangle \in Q_{\min}$ occurs in at last one rule in $R_{\min}$ as every state $q \in Q$ occurs in a least one rule $r \in R$.

We described so far how the minimal DAG automaton $A_{\min}$ is constructed and showed that it is equivalent to the original DAG automaton A. To prove the next theorem, which is the main result of this section, it remains to be shown that $A_{\min}$ is actually minimal and unique.

Theorem 7.8. *Let A be a top-down deterministic DAG automaton. Then a minimal top-down deterministic DAG automaton $A_{\min}$ with $L(A) = L(A_{\min})$ can be computed in polynomial time. Moreover, every minimal top-down deterministic DAG automaton that accepts $L(A)$ is identical to $A_{\min}$ up to a bijective renaming of states.*

Proof. We may assume that $A = (Q, \Sigma, R)$ is reduced, as we already know that we can otherwise compute a reduced DAG automaton equivalent to A in polynomial time. From A, $A_{\min} = (Q_{\min}, \Sigma, R_{\min})$ can be constructed in polynomial time, and Lemma 7.7 shows that $L(A_{\min}) = L(A)$.

To prove minimality, suppose there is another top-down deterministic DAG automaton $A' = (Q', \Sigma, R')$ with $L(A') = L(A)$ and $|Q'| < |Q_{\min}|$. Then there are two DAGs $G_p, G_q \in L(A_{\min})$ such that the following holds. There are two edges $e_p \in E_{G_p}$ and $e_q \in E_{G_q}$ such that the runs ρ_p and ρ_q of $A_{\min}$ on G_p and G_q satisfy $\rho(e_p) = p \neq q = \rho(e_q)$ whereas the runs ρ'_p and ρ'_q of A' on G_p and G_q satisfy $\rho'_p(e_p) = \rho'_q(e_q)$. (Otherwise, a bijective mapping from Q' to $Q_{\min}$ could be constructed by mapping, for every DAG G and every edge $e \in E_G$, $\rho'(e)$ to $\rho(e)$, where ρ and ρ' are the runs of $A_{\min}$ and A', resp., on G.)

By the construction of $A_{\min}$ we have $(p, q) \in \Delta_{A_{\min}}$. Let $(\alpha_1, p_1{:}\sigma_1, \alpha'_1) \cdots (\alpha_k, p_k{:}\sigma_k, \alpha'_k)$ be a corresponding witness (with $p_1 = p$). For $j \in [k]$, let $r_j \in R_{\min}$ be the unique σ_j-rule in $R_{\min}$ with the head $\alpha_j p_j \alpha'_j$. As $A_{\min}$ is reduced, for every rule $j \in [k]$ there is a DAG $G_j \in L(A_{\min})$ such that a run ρ_j of $A_{\min}$ on G_j uses the rule r_j in a vertex $v_j \in V_{G_j}$. From Definition 7.4 it follows that for every $j \in [k-1]$ the tail of r_j can be written as $\beta_j p_{j+1} \beta'_j$. We can therefore iteratively connect the DAGs G_j and G_{j+1} for $j = 1, \ldots, k-1$ by swapping the $(|\beta_j|+1)$-th outgoing edge of v_j and the $(|\alpha_{j+1}|+1)$-th incoming edge of v_{j+1}. By Lemma 3.3 the resulting DAG G is still accepted by $A_{\min}$.

Since $p = p_1$ we can also furthermore connect G with G_p by swapping the $(|\alpha_i|+1)$-th incoming edge of v_1 and the edge e_p. Again, the resulting DAG G^* continues to be accepted by $A_{\min}$. Fig. 15 illustrates how we obtain the DAG G^* by connecting $G_1, \ldots, G_k$ and G_p.

Consider now the DAG G' that is obtained by swapping the edges e_p and e_q between the DAGs G^* and G_q. The DAG G' is also illustrated in Fig. 15. We show that G' is not accepted by $A_{\min}$. Assume therefore that there is a run ρ' of $A_{\min}$ on G'. Remember that the rule to be used in a certain vertex of $v \in G'$ is unambiguously defined by the rules that are used in the parents of v. The parents and ancestors of the vertex from which the edge e_q originates were not affected by any swap operations, therefore we have $\rho'(e_q) = q$. The same holds for the parents and ancestors of any other incoming edge of the vertex v_1. The run ρ' uses therefore a rule with head $\alpha_1 q \alpha'_1$ in the vertex v_1. This means the run ρ' marks e_2, the $(|\alpha_2|+1)$-th incoming edge of v_2 with some state q_2. All other incoming edges of v_2 are marked with the same state as in the run of $A_{\min}$ on G^*, for which reason ρ' needs to use a rule with head $\alpha_1 q_2 \alpha'_2$ in the vertex v_2. Inductively it follows that the run ρ' needs to use a rule with head $\alpha_j q_j \alpha'_j$ in the vertex v_j for every $j \in [k]$. As $(\alpha_1, p_1{:}\sigma_1, \alpha'_1), \ldots, (\alpha_k, p_k{:}\sigma_k, \alpha'_k)$ is a witness there is however no σ_k-rule with head $\alpha_k q_k \alpha'_k$, which means that there is no rule $r \in R_{\min}$ that can be used in the vertex v_k. Hence no run of $A_{\min}$ on G' exists, and we have $G' \notin L(A_{\min})$.

Consider now the DAG automaton A' which assigns the same state, say z, to both edges e_p and e_q in the DAGs G_p and G_q, resp. We assumed that we have $L(A') = L(A_{\min})$ which means that A' accepts the DAG G^*. By the same reasoning as above, the run of A' on G^* assigns z to e_p as we did not modify anything in the vertices "above" e_p. Hence, we can

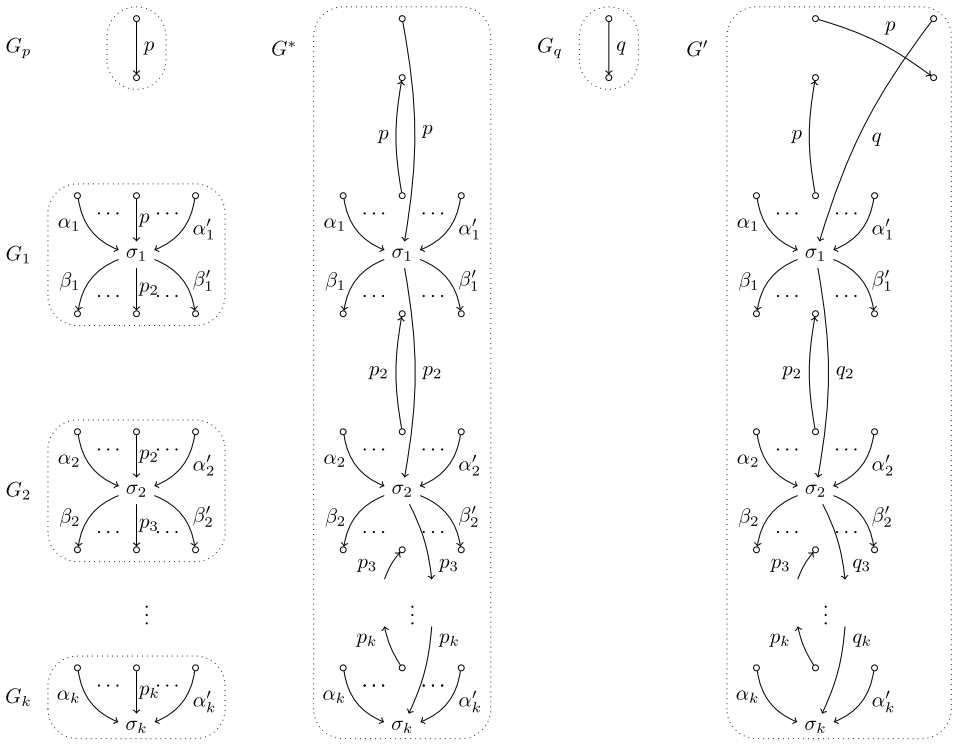


Fig. 15. Construction of the DAGs G^* and G'.

connect G^* and G_q by swapping e_p and e_q, and the resulting DAG G' is accepted by A'. This shows that $L(A') \setminus L(A_{\min}) \neq \emptyset$, contradicting the assumption that $A_{\min}$ and A' are equivalent.

We finally show that $A_{\min}$ is unique up to a renaming of states, i.e. that for every top-down deterministic DAG automaton $A' = (Q', \Sigma, R')$ with $L(A') = L(A_{\min})$ and $|Q'| = |Q_{\min}|$ there is a bijection $f\colon Q_{\min} \to Q'$ such that

$$R' = \{f(\alpha) \overset{\sigma}{\longleftrightarrow} f(\beta) \mid \alpha \overset{\sigma}{\longleftrightarrow} \beta \in R_{\min}\}.$$

Assume to the contrary that no such bijection exists. Then there are two DAGs $G, G' \in L(A_{\min})$ for which the following holds: There are $e \in E_G$ and $e' \in E_{G'}$ such that the runs ρ and ρ' of $A_{\min}$ on G and G', resp., satisfy $\rho(e) \neq \rho'(e')$ but the corresponding runs κ and κ' of A' on G and G' satisfy $\kappa(e) = \kappa'(e')$. As was shown above this implies that $L(A_{\min}) \neq L(A')$, a contradiction. □

We close this section by showing that the equivalence problem is decidable in polynomial time for top-down deterministic DAG automata. Formally the equivalence problem asks, for two DAG automata A and A' given as input, whether $L(A) = L(A')$.

In order to answer this question for two top-down deterministic DAG automata A and A' we can construct the corresponding minimal automata $A_{\min} = (Q, \Sigma, R)$ and $A'_{\min} = (Q', \Sigma', R')$ as above. Since the minimal DAG automaton is unique we have $L(A) = L(A')$ if and only if $A_{\min}$ and $A'_{\min}$ represent the same DAG automaton. If we have $|Q| \neq |Q'|$, or $|R| \neq |R'|$ this is clearly not the case. Otherwise we need to check if there is a bijection $f\colon Q \to Q'$ such that $R' = \{f(\alpha) \overset{\sigma}{\longleftrightarrow} f(\beta) \mid \alpha \overset{\sigma}{\longleftrightarrow} \beta \in R\}$.

We show that one can decide in polynomial time if such a bijection $f\colon Q \to Q'$ exists. Assume that there are rules $p_1 \cdots p_m \overset{\sigma}{\longleftrightarrow} q_1 \cdots q_n$ and $p'_1 \cdots p'_m \overset{\sigma}{\longleftrightarrow} q'_1 \cdots q'_n$ in R and R', resp. If we already know that $f(p_i) = p'_i$ for every $i \in [m]$ (which, in particular, is the case if $m = 0$) it follows that $f(q_i) = q'_i$ for all $i \in [n]$. Using this principle we can iterate over R and construct f in a stepwise manner. If at some point we come to contradictory results this means that f cannot be bijective, i.e. $A_{\min}$ and $A'_{\min}$ are not equivalent. If no such situation occurs, both automata are equivalent as we successfully created a bijection $f\colon Q \to Q'$ with the required property.

However, while processing a rule $\alpha \overset{\sigma}{\longleftrightarrow} \beta$ we must ensure that $f(p)$ has already been defined for all $p \in [\alpha]$. This can be done by iterating through R in a topological order. We say that a *topological ordering* of R is a linear ordering of R as a list $r_1, \ldots, r_k$ such that for every rule $r_i = (\alpha \overset{\sigma}{\longleftrightarrow} \beta)$ and every $p \in [\alpha]$ there is a rule $r_j = (\alpha' \overset{\sigma'}{\longleftrightarrow} \beta')$ with $j < i$ such that $p \in [\beta']$. In order to compute a topological ordering of R we can use Algorithm 1.

Lemma 7.9. *Algorithm 1 computes a topological ordering of R in polynomial time.*

Algorithm 1: Construction of a topological ordering.

Input: A reduced top-down deterministic DAG automaton $A = (Q, \Sigma, R)$

1 $P = \emptyset$ // Produced states
2 $L = ()$ // Topological ordering
3 **while** $R \neq \emptyset$ **do**
4 Choose a rule $r = \alpha \xleftarrow{\sigma} \beta$ such that $\alpha \in P^*$
5 $P = P \cup [\beta]$
6 $R = R \setminus \{r\}$
7 Append r to L
8 **end**

Algorithm 2: Comparison of two minimal automata.

Input: Minimal top-down deterministic DAG automata $A_{\min} = (Q, \Sigma, R)$ and $A'_{\min} = (Q', \Sigma', R')$

1 **if** $|Q| \neq |Q'|$ *or* $|R| \neq |R'|$ **then return** false
2 Compute a topological ordering $r_1, \ldots, r_k$ of R
3 **for** $i = 1, \ldots, k$ **do**
4 Let $r_i = \alpha \xleftarrow{\sigma} \beta$ where $\beta = q_1 \cdots q_m$
5 Select the rule $(f(\alpha) \xleftarrow{\sigma} \beta') \in R'$ where $\beta' = q'_1 \cdots q'_m$
6 **if** *there is no such rule* **then return** false
7 **for** $j = 1$ *to* m **do**
8 **if** $f(q_i)$ *undefined* **then** set $f(q_i) = q'_i$
9 **else if** $f(q_i) \neq q'_i$ **then return** false
10 **end**
11 **end**
12 **return** true

Proof. Clearly, L as computed by the algorithm is a topological ordering, provided that line 4 is correct, i.e. every time it is executed, a suitable rule does exist. Assume therefore that Algorithm 1 is just about to execute line 4 and let r be a rule that has not been inserted into the list L yet. As A is a reduced DAG automaton there is a DAG $G \in L(A)$ such that there is a run ρ of A on G which uses the rule r in a vertex $v \in V_G$. If $\rho(in_G(v)) \in P^*$ then r fulfills the requirement in line 4. Otherwise we consider an edge $e \in [in_G(v)]$ such that $\rho(e) \notin P$. Then the rule r' used in $src_G(e)$ has not been added to L yet, and we repeat, now checking r'. Since G is acyclic, this search will eventually end, thus arriving at a rule that fulfills the criterion. This shows that the algorithm is correct.

The while loop is executed $|R|$ times. Line 4 can be performed in time $O(|R| \cdot M)$, where M is the maximum of the lengths of all heads and tails of rules in R. Each of the remaining steps can be performed in time $O(M)$, which yields a total running time of $O(|R|^2 \cdot M)$. □

As previously described we can now use the topological ordering to compare the minimized DAG automata $A_{\min}$ and $A'_{\min}$ by trying to construct the mentioned bijection $f\colon Q \to Q'$. Algorithm 2 shows how this can be done.

The correctness proof is, at the same time, the proof of the second main result of this section.

Theorem 7.10 *(Decidability of equivalence). The equivalence problem for deterministic DAG automata can be decided in polynomial time.*

Proof. In order to decide if two top-down deterministic DAG automata A and A' are equivalent, construct two reduced DAG automata A_{red} and A'_{red} that are equivalent to A and A' resp. Then construct DAG automata $A_{\min} = (Q, \Sigma, R)$ and $A'_{\min} = (Q', \Sigma, R')$ equivalent to A_{red} and A'_{red}, resp., and apply Algorithm 2. We have to show that Algorithm 2 is correct and runs in polynomial time.

If Algorithm 2 returns true then the automata $A_{\min}$ and $A'_{\min}$ are equivalent as the algorithm successfully constructed a bijection $f\colon Q \to Q'$ such that $R' = \{f(\alpha) \xleftarrow{\sigma} f(\beta) \mid (\alpha \xleftarrow{\sigma} \beta) \in R\}$. If the algorithm returns false, there may be several reasons:

- $|Q| \neq |Q'|$ or $|R| \neq |R'|$.
- A rule $\alpha \xleftarrow{\sigma} \beta$ in R was encountered such that R' did not contain a σ-rule with the head $f(\alpha)$.
- A rule $\alpha \xleftarrow{\sigma} q_1 \cdots q_m$ in R with a corresponding rule $f(\alpha) \xleftarrow{\sigma} q'_1 \cdots q'_m$ in R' was encountered such that, for some $i \in [m]$, $f(q_i) \neq q'_i$ had already been defined in an earlier iteration.

In all cases there is no bijection $f\colon Q \to Q'$ such that $R' = \{f(\alpha) \xleftarrow{\sigma} f(\beta) \mid (\alpha \xleftarrow{\sigma} \beta) \in R\}$. Regarding the second and third case, this is because one can easily prove the following loop invariant: if there exists a bijection of the required type, then f is subset of that bijection (viewing mappings as special cases of binary relations). In other words, if the algorithm returns false then $L(A_{\min}) \neq L(A'_{\min})$, as required.

Let us finally examine the running time of Algorithm 2. Again, let M be the maximum of the lengths of all heads and tails of rules in R. The outer loop is executed $|R|$ times whereas the inner loop is executed $O(M)$ times during each execution of the outer loop. All remaining operations within the outer loop can be performed in constant time, which results in a total running time of $O(|R| \cdot M)$ for the outer loop. Line 1 can be performed in time $O(|R|)$, and line 2 requires $O(|R|^2 \cdot M)$ steps, which yields an overall running time of $O(|R|^2 \cdot M)$. □

8. Conclusions and directions for future work

In this paper we have studied the properties of regular DAG languages. The results resemble the well-known results for regular string and tree languages almost perfectly. Despite this fact, the techniques used to prove these results differ considerably from the classical ones. In particular, edge swaps turned out to be the basis of a versatile proof technique employed almost everywhere throughout the paper.

Together with the fact that regular tree languages are a special case of regular DAG languages, the results of this paper seem to indicate that the DAG automaton is indeed a natural notion that is worth studying. Interestingly, there is a prominent point at which the properties of the class of regular DAG languages differ from the classical cases of strings and trees: as shown in [6] there exist NP-complete regular DAG languages. In other words, even the non-uniform membership problem is NP-complete. As pointed out by an observant referee, the non-uniform membership problem is solvable in linear time on input DAGs of bounded treewidth by Courcelle's theorem (see, e.g., [7, Theorem 6.4(1)]) or in logarithmic space by [12]. This is because acceptance by a (fixed) DAG automaton can easily be expressed by a monadic second-order formula. A first attempt to arrive at a more practical algorithm (which is polynomial on DAGs of bounded treewidth) is made in [6]. Future work should investigate the membership problem further and study other special cases for which efficient membership tests exist, or alternative algorithms.

DAG automata share the NP-completeness of their non-uniform membership problem with another formalism that is being discussed as a candidate for describing languages of AMRs, namely context-free graph grammars and in particular hyperedge-replacement grammars (see [8,11] for overviews and lots of further references). Using a Bar–Hillel-type construction, it is shown in [6] that the class of hyperedge-replacement languages is closed under intersection with regular DAG languages. It is straightforward to see that both classes are incomparable, but it may be fruitful to have a closer look at the relation between the two, because both have been suggested as potentially useful formalisms for meaning representation in natural language processing.

The results about deterministic DAG automata raise some obvious open questions. Is the equivalence problem of (nondeterministic) DAG automata decidable? What is the relation between the class of regular DAG languages and its bottom-up and top-down deterministic subclasses? Are there natural characterizations of these classes, perhaps in terms of each other? The example discussed at the end of Section 2 is the union of a top-down deterministic and a bottom-up deterministic regular DAG language, but this seems to be due to the simplicity of the example.

As was pointed out by a referee, there are further language theoretic questions one should answer. For example, it seems likely that the Parikh image of a regular DAG language is semilinear. We currently do not know whether this is true. Another question concerns "inverses" of Theorem 5.2: given a tree language L, are there natural constructions that yield regular DAG languages L' with $L = \mathit{tree}(L')$? Let us discuss an example of such a (perhaps reasonably natural) construction. Consider a tree automaton $A = (Q, \Sigma, R)$ and assume, for the sake of simplicity, that Σ is ranked, each symbol having one of the ranks $(0,2)$, $(1,2)$, and $(1,0)$. We want construct a DAG automaton $A' = (2^Q, \Sigma', R')$ that "folds" subtrees of trees generated by A into DAGs with vertices having two incoming edges, thus turning the ranks of symbols into $(0,2)$, $(2,2)$, and $(2,0)$, respectively. In a first step, we apply a powerset construction to A, thus obtaining $\hat{A} = (2^Q, \Sigma, \hat{R})$. In $\hat{A}$ a state $P \subseteq Q$ can be the root label of a run on a subtree if and only if, in A, every $p \in P$ can. (Note that the goal is *not* to make $\hat{A}$ deterministic, but to incorporate intersection into its rules in order to make folding possible below.) The rules in $\hat{R}$ are given as follows:

1. For every $\sigma \in \Sigma$ of rank $(0,2)$ and all $Q_1, Q_2 \subseteq Q$, $\hat{R}$ contains the rule
$$\lambda \xleftarrow{\sigma} Q_1 Q_2$$
if there are $q_1 \in Q_1, q_2 \in Q_2$ such that $(\lambda \xleftarrow{\sigma} q_1 q_2) \in R$.
2. For $\sigma \in \Sigma$ of rank $(1,2)$ and all $P, Q_1, Q_2 \subseteq Q$, $\hat{R}$ contains the rule
$$P \xleftarrow{\sigma} Q_1 Q_2$$
if, for every $p \in P$, there are $q_1 \in Q_1, q_2 \in Q_2$ such that $(p \xleftarrow{\sigma} q_1 q_2) \in R$.
3. For $\sigma \in \Sigma$ of rank $(1,0)$ and all $P \subseteq Q$, $\hat{R}$ contains the rule
$$P \xleftarrow{\sigma} \lambda$$
if $(p \xleftarrow{\sigma} \lambda) \in R$ for all $p \in P$.

The reader may wish to verify that, indeed, $L(\hat{A}) = L(A)$.

Now, we let R' consist of the following three types of rules for $\sigma \in \Sigma$:

1. All rules of the form $\lambda \xleftarrow{\sigma} Q_1 Q_2$ in $\hat{R}$ are also in R'.
2. For all pairs of rules $P_1 \xleftarrow{\sigma} Q_1 Q_2$ and $P_2 \xleftarrow{\sigma} Q_1 Q_2$ in $\hat{R}$, R' contains the rule $P_1 P_2 \xleftarrow{\sigma} Q_1 Q_2$.
3. Similarly, for all pairs of rules $P_1 \xleftarrow{\sigma} \lambda$ and $P_2 \xleftarrow{\sigma} \lambda$ in $\hat{R}$, R' contains the rule $P_1 P_2 \xleftarrow{\sigma} \lambda$.

It is not difficult to show that $L(\hat{A}) = tree(L(A'))$. For the inclusion '$\supseteq$', one assumes that there is a run ρ of A' on a DAG G and considers a root r of G. Every edge e of $tree_r(G)$ has a pre-image $\varphi(e)$ in G, and from the construction of R' it should be clear that defining $\rho'(e) = \rho(\varphi(e))$ for all $e \in E_{tree_r(G)}$ yields a run of $\hat{A}$ on $tree_r(G)$. This shows that $L(\hat{A}) \supseteq tree(L(A'))$. The proof of the other direction can be obtained as follows. Let $\hat{\rho}$ be a run of $\hat{A}$ on a tree T. Let us say that all leaves are at height 0, and a vertex which is not a leaf is at height $i+1$ if i is the maximum of the heights of its two children. Suppose that h is the height of the root of T. Now, copy all vertices at height > 0 together with their incoming and outgoing edges. This way, the vertices at height 0 (the leaves) end up with two incoming edges each. The run is extended to the larger DAG by assigning all copies of edges the same state as the original edge. Repeating this procedure for $i = 1, \ldots, h-1$, each time copying all vertices at height $> i$, yields a DAG G (of exponential size) together with a correct run in A'. Further, it should be clear that $tree_r(G) = T$ for all roots r of G, thus showing that $L(\hat{A}) \subseteq tree(L(A'))$.

Another type of interesting questions regarding DAG automata, especially in view of their use in natural language processing, concerns algorithmic learning. Practical algorithms for learning DAG automata from corpora could prove highly useful. On the more theoretical side, one may attempt to learn special cases of DAG automata (such as deterministic ones) in the style of Gold or Angluin, which seems to be a challenging task. Angluin-style learning could be obtained from Myhill–Nerode-like theorems, e.g. for deterministically regular DAG languages, but it seems to be unclear how such results could be established. A closely related question is that of training, using the weighted extension of DAG automata proposed in [6]. In such a DAG automaton every rule is given a weight taken from some semiring. Consequently, a run has an overall weight, namely the product of the weights of all rules constituting the run. In turn, this assigns a weight to each DAG, namely the sum of the weights of all runs on that DAG. The weights can e.g. reflect the likelihood of an AMR to correctly describe the meaning of a given sentence, and one could try to devise training algorithms for weighted DAG automata similar to training algorithms for weighted tree automata (see [10,14]).

Acknowledgment

We thank the referees for carefully reading the manuscript and making numerous useful suggestions for improving the paper.

References

[1] S. Anantharaman, P. Narendran, M. Rusinowitch, Closure properties and decision problems of dag automata, Inf. Process. Lett. 94 (5) (2005) 231–240.
[2] L. Banarescu, C. Bonial, S. Cai, M. Georgescu, K. Griffitt, U. Hermjakob, K. Knight, P. Koehn, M. Palmer, N. Schneider, Abstract meaning representation for sembanking, in: Proc. 7th Linguistic Annotation Workshop, ACL 2013 Workshop, 2013.
[3] J. Blum, DAG Automata – Variants, Languages and Properties, Master thesis, Umeå University, 2015.
[4] J. Blum, F. Drewes, Properties of regular DAG languages, in: A. Dediu, J. Janoušek, C. Martín-Vide, B. Truthe (Eds.), Proc. 10th Intl. Conf. on Language and Automata Theory and Applications, in: Lecture Notes in Computer Science, vol. 9618, 2016, pp. 427–438.
[5] W. Charatonik, Automata on Dag Representations of Finite Trees, Research Report MPI-I-1999-2-001, Max-Planck-Institut für Informatik, Saarbrücken, 1999.
[6] D. Chiang, F. Drewes, D. Gildea, A. Lopez, G. Satta, Weighted dag automata for semantic graphs, 2016, unpublished manuscript.
[7] B. Courcelle, J. Engelfriet, Graph Structure and Monadic Second-Order Logic – A Language-Theoretic Approach, Cambridge University Press, 2012.
[8] F. Drewes, A. Habel, H.-J. Kreowski, Hyperedge replacement graph grammars, in: G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations, World Scientific, Singapore, 1997, pp. 95–162, Ch. 2.
[9] F. Drewes, J. Leroux, Structurally cyclic Petri nets, Log. Methods Comput. Sci. 11 (4) (2015) 15.
[10] J. Graehl, K. Knight, J. May, Training tree transducers, Comput. Linguist. 34 (3) (2008) 391–427.
[11] A. Habel, Hyperedge Replacement: Grammars and Languages, Lecture Notes in Computer Science, vol. 643, Springer, 1992.
[12] A. Jakoby, M. Elberfeld, T. Tantau, Logspace versions of the theorems of bodlaender and courcelle, in: Proc. 51st Annual IEEE Symp. Foundations of Computer Science, FOCS 2010, 2010, pp. 143–152.
[13] T. Kamimura, G. Slutzki, Parallel and two-way automata on directed ordered acyclic graphs, Inf. Control 49 (1981) 10–51.
[14] K. Knight, J. May, Applications of weighted automata in natural language processing, in: W. Kuich, M. Droste, H. Vogler (Eds.), Handbook of Weighted Automata, Springer, 2009, pp. 571–596, Ch. 4.
[15] A. Potthoff, S. Seibert, W. Thomas, Nondeterminism versus determinism of finite automata over directed acyclic graphs, Bull. Belg. Math. Soc. Simon Stevin 1 (2) (1994) 285.
[16] L. Priese, Finite automata on unranked and unordered DAGs, in: T. Harju, J. Karhumäki, A. Lepistö (Eds.), Proc. 11th Intl. Conf. on Developments in Language Theory, DLT 2007, in: Lecture Notes in Computer Science, vol. 4588, 2007, pp. 346–360.
[17] D. Quernheim, K. Knight, Towards probabilistic acceptors and transducers for feature structures, in: Proc. 6th Workshop on Syntax, Semantics and Structure in Statistical Translation, Association for Computational Linguistics, 2012, pp. 76–85.