Department of Computing Science
Umeå University
Sweden

Master's Thesis in Computing Science
(30 HP)

DAG Automata – Variants, Languages and Properties

Johannes Blum

submitted on May 25, 2015

Supervisor: Frank Drewes
Examiner: Jerry Eriksson

Abstract

We study final-state automata that work on directed acyclic graphs (DAGs). We consider ordered and unordered DAGs and show that they can be simulated by each other. Then we show that deterministic DAG automata are weaker then nondeterministic automata. Some properties of recognizable DAG languages are presented and the decidability of the finiteness problem is shown. Finally we compare tree and DAG automata and show that the path language of a DAG language is regular by proving that the tree unfolding of a DAG language preserves recognizability.

DAG-Automater – Varianter, Språk och Egenskaper

Sammanfattning

Vi studerar finita automater som arbetar på riktade acykliska grafer (DAG:ar). Vi betraktar ordnade och oordnade DAG:ar och visar att de kan simulera varandra. Sedan visar vi att deterministiska DAG-automater är svagare än icke-deterministiska automater. Några egenskaper av igenkännbara DAG-språk presenteras och avgörbarheten av ändlighetsproblemet visas. Slutligen jämför vi träd- och DAG-automater och visar att stigspråket av ett DAG-språk är reguljärt genom att bevisa att trädutvecklingen av ett DAG-språk bevarar igenkännbarheten.

Acknowledgement

I would like to thank my supervisor Frank Drewes for his support and the enlightening discussions.

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Partial and Total Orders 3
2.2 DAGs 3
2.3 Operations on DAGs and DAG Languages 5
2.4 DAG Automata 7

3 Ordered, Unordered and General DAGs **11**

4 Determinism **15**

5 Properties of Recognizable DAG Languages **19**
5.1 Closure Properties 19
5.2 Necessary Properties 22

6 Finiteness of DAG Automata **27**

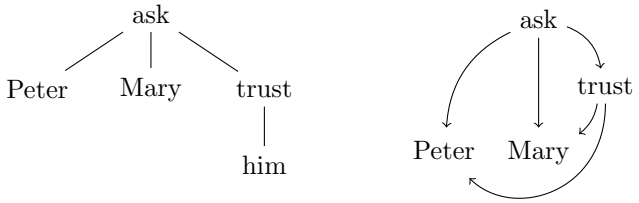
7 Connection to Tree Automata **33**

8 Conclusion and Future Work **39**

References **41**

1 Introduction

Automata that specify sets of trees, so called tree languages, have been subject of broad research in theoretical computer science during the past decades. Gécseg and Steinby [10, 11] and Comon et al. [7] provide a comprehensive introduction into the topic. One application of such tree automata can be found in computational linguistics, where trees are used to model the syntax of sentences given in a natural language. To illustrate this, Figure 1.1a shows a tree representing the syntactic structure of the sentence "Peter asks Mary to trust him".



(a) A syntactical tree (b) An AMR

Figure 1.1: Two representations of the sentence "Peter asks Mary to trust him"

The linguistic benefit of the tree automata model is based in its algorithmic and structural properties. In a good model one should for example be able to determine in polynomial time if the language recognized by a given automaton is empty or finite. Useful properties are moreover closedness under the boolean set operations and the ability to parse a given sentence efficiently. Tree automata inherit all these properties from the older finite-state automata (FSAs) that recognize regular string languages, to which they are closely related. One can actually consider tree automata as an extension of FSAs for which reason many results for FSAs can easily be applied in a generalized form to tree automata. The close relationship between finite-state tree and string automata is moreover visible in the regularity of the path languages of recognizable tree languages.

A recent approach to model natural language is Abstract Meaning Representation (AMR) which does not only represent the syntax of a given sentence but also its semantics [2]. As an example, the AMR of the sentence "Peter asks Mary to trust him" is shown in Figure 1.1b. AMRs are no longer trees but rather directed acyclic graphs (DAGs) as the meaning of a single word depends in general on several other words. This can be seen in Figure 1.1b where "trust" refers to "Mary" and "Peter".

Therefore a new automata model is required that is capable to handle languages of DAGs. An ideal DAG automaton would still provide the mentioned properties polynomial decidability of emptiness and finiteness, closedness under set operations, efficient parsing and regularity of the path languages. It should nevertheless have enough descriptive power for linguistic applications.

In the past several types of DAG automata have been proposed that differ mainly in the choice of the underlying DAGs. Kamimura and Slutzki studied so called *d-dags*,

planar DAGs that have one single root and a global order on the vertices [12, 13]. Bossut et al. [3, 4] discussed a generalization of Kamimura and Slutzki's approach. Their *p-dags* are not generally single rooted and connected, but still planar. The drawback of these DAGs is however that they can be used to model the derivations of type-0 grammars which implies the undecidability of the emptiness and the finiteness problem. AMRs are moveover not generally planar and can have several roots. Cheratonik [5] and Anantharaman et al. [1] investigated so called *t-dags* that are not necessary planar but share structure maximally which means that they do not contain two isomorphic subgraphs. This is an unnatural restriction from a linguistic perspective. The DAGs we consider are therefore a generalization of the unordered and ordered DAGs that Chiang et al. [6] and Quernheim and Knight [15] introduced with the intention to overcome the disadvantages of previous models. A similar approach was also discussed by Potthoff et al. [14]. Our DAGs are connected and nonempty. They can have several roots but are not necessarily planar. A drawback of this concept is still that the in- and outdegree of the vertices are bounded. This is however only a minor restriction that does not limit the linguistic benefit essentially.

The purpose of this thesis is to propose a DAG automata model that integrates the approaches of Chiang et al. and Quernheim and Knight and to investigate its properties. We show in Chapter 3 that it is of inferior relevance if then incident edges of every vertex are ordered or not, as ordered DAGs can be simulated by unordered DAGs and vice versa. Chapter 4 contains a discussion of determinism for the top-down and bottom-up case. We show that deterministic DAG automata are weaker than nondeterministic DAG automata which mirrors the fact that top-down deterministic tree automata are weaker than their nondeterministic counterpart. We introduce moreover the dual language of a given DAG language that is obtained by turning all contained DAGs upside down. This means that roots become leaves, leaves become roots and all edges change their direction. The dual language of a top-down deterministic DAG language is then bottom-up deterministic and vice versa, for which reason it suffices to consider only top-down or bottom-up deterministic languages. In Chapter 5 we show that recognizable DAG languages are closed under union, intersection and concatenation. This holds also for recognizable tree languages. The recognizable DAG languages are however not closed under complement, in contrast to their tree counterpart. To prove this we apply a basic property of DAGs, namely that one can swap two edges that are marked by an automaton with identical states. A similar result for tree languages is that one can exchange subtrees whose roots are marked with the same state. The fact that DAGs can contain multiple roots and several paths between two vertices effects however that the possibility to swap two edges has graver consequences. Based on these edge swaps we present two necessary properties of recognizable DAG languages that generalize the pumping lemma for tree languages and conclude a characterization of finite DAG languages. Chapter 6 contains a proof of the decidability of the finiteness problem for DAG automata in polynomial time. It is based on a similar proof of Chiang et al. that shows the polynomial decidability of emptiness. This is an remarkable difference to the DAGs discussed by Cheratonik and Anantharaman et al. where the emptiness problem is NP-complete. In Chapter 7 we preform a brief formal comparison of tree and DAG automata and present the tree unfolding of a DAG that transforms a given DAG into a tree by copying shared child vertices. We show that the tree unfolding preserves recognizability which implies the regularity of the path language of a recognizable DAG language.

2 Preliminaries

2.1 Partial and Total Orders

A *partial order* on a set S is relation $R \subseteq S \times S$ such that for all $a, b, c \in S$ we have

- $(a, a) \in R$ (reflexivity),
- if $(a, b) \in R$ and $(b, a) \in R$ we have $a = b$ (antisymmetry) and
- if $(a, b) \in R$ and $(b, c) \in R$ we have $(a, c) \in R$ (transitivity).

If a relation R is a partial order on a set S, the tuple (S, R) is called a *partially ordered set*. A partial order $R \subseteq S \times S$ is a *total order* if and only for all $a, b \in S$ we have $(a, b) \in R$ or $(b, a) \in R$. A *totally ordered set* is a tuple (S, R) where S is a set and R is a total order on S. If $T = (S, R)$ is a totally ordered set with $S = \{x_1, \ldots, x_n\}$ and we have $(x_1, x_2), (x_2, x_3), \ldots, (x_{n-1}, x_n) \in R$, we also identify T with the sequence $(x_1, x_2, \ldots, x_n)$ and write just $T = (x_1, x_2, \ldots, x_n)$. A totally ordered set $T = (S, R')$ is *compatible* with a partial order R on the set S if and only if $(a, b) \in R$ implies $(a, b) \in R'$ for all $a, b \in S$.

2.2 DAGs

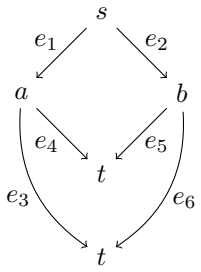
A *doubly ranked alphabet* is a set $\Sigma = \bigcup_{i,j} \Sigma_{i,j}$ where each $\Sigma_{i,j}$ is a finite set of *symbols* and we have $\Sigma_{i,j} \neq \emptyset$ only for finitely many i and j. Every symbol $\sigma \in \Sigma_{n,m}$ is a tuple $(id(\sigma), \mathcal{R}_{in}(\sigma), \mathcal{R}_{out}(\sigma))$ where $id(\sigma)$ is the *identifier* of σ and $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ are partial orders on the sets $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$ respectively. Moreover we call $head(\sigma) = n$ the *head rank* and $tail(\sigma) = m$ the *tail rank* of a symbol $\sigma \in \Sigma_{n,m}$.

We will use such symbols for labeling vertices of a graph where a vertex having n in- and m outgoing edges will get a symbol with head rank n and tail rank m. Intuitively the partial orders $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ specify furthermore in which order the in- and outgoing edges may occur. Formally we introduce *labeled graphs* as follows.

Definition 2.1 (Labeled Graphs)**.** A *labeled graph* is a tuple $G = (V, E, label, in, out)$ where V is the set of *vertices* and E is the set of *edges*. For every vertex $v \in V$, $in(v) = (E_{in}(v), R_{in}(v))$ and $out(v) = (E_{out}(v), R_{out}(v))$ are totally ordered sets where $E_{in}(v) \subseteq E$ and $E_{out}(v) \subseteq E$ are the *ingoing* and *outgoing edges* of v respectively such that for every edge $e \in E$ there is exactly one *start vertex* $v_s \in V$ and exactly one *end vertex* $v_e \in V$ with $e \in E_{out}(v_s)$ and $e \in E_{in}(v_e)$. Moreover $label : V \to \Sigma$ is a function such that $label(v)$ has head rank $|E_{in}(v)|$ and tail rank $|E_{out}(v)|$.

When the context is clear, we confuse in the following $in(v)$ and $E_{in}(v)$ as well as $out(v)$ and $E_{out}(v)$. For a labeled Graph G we refer moreover with $V_G, E_G, label_G, in_G$ and out_G to the components of G.

EXAMPLE 2.1: Figure 2.1 shows an example of a labeled graph which consists of five vertices $V = \{v_1, \ldots, v_5\}$ and six edges $E = \{e_1, \ldots, e_6\}$. Moreover we have $\Sigma_{0,2} = \{s\}, \Sigma_{1,2} = \{a, b\}$ and $\Sigma_{2,0} = \{t\}$ and assume that for every symbol $\sigma \in \Sigma$ the relations $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ are the $\leq$-relation on natural numbers.



v	$label(v)$	$in(v)$	$out(v)$
v_1	s	$\emptyset$	(e_1, e_2)
v_2	a	(e_1)	(e_3, e_4)
v_3	b	(e_2)	(e_5, e_6)
v_4	t	(e_4, e_5)	$\emptyset$
v_5	t	(e_3, e_6)	$\emptyset$

Figure 2.1: A labeled graph

A vertex v with $in(v) = \emptyset$ is also called a *root*, a vertex v with $out(v) = \emptyset$ is also called a *leaf*. An edge $e \in in(v) \cup out(v)$ is called *incident* to v. A *directed path* from a vertex v_1 to a vertex v_k is an alternating sequence of vertices and edges $(v_1, e_1, v_2, \ldots, e_{k-1}, v_k)$ such that $e_i \in out(v_i) \cap in(v_{i+1})$ for $i \in \{1, \ldots, k-1\}$. An *undirected path* between two vertices v_1 and v_k is an alternating sequence of vertices and edges $(v_1, e_1, v_2 \ldots, e_{k-1}, v_k)$ such that $e_i \in (out(v_i) \cap in(v_{i+1})) \cup (in(v_i) \cap out(v_{i+1}))$ for $i \in \{1, \ldots, k-1\}$. A *directed (undirected) cycle* is a directed (undirected) path $(v_1, e_1, v_2 \ldots, e_{k-1}, v_k)$ with $v_1 = v_k$ and $k > 1$. A labeled graph is *acyclic* if and only if it does not contain a directed cycle.

A labeled graph G' is a *subgraph* of a labeled graph G if and only if $V_{G'} \subseteq V_G, E_{G'} \subseteq E_G$ and $label_{G'}, in_{G'}$ and $out_{G'}$ are restrictions of $label_G, in_G$ and out_G to $V_{G'}$. A *connected component* of a labeled graph G is a subgraph C of G such that for every undirected path in G between two vertices v and v' we have either $v \in V_C$ and $v' \in V_C$ or $v \notin V_C$ and $v' \notin V_C$. If a labeled graph G consists of k connected components $C_1, \ldots C_k$ we write also $G = C_1 \& \ldots \& C_k$. For a set of graphs L we define

$$L^{\&} = \{C_1 \& \ldots \& C_k \mid k \geq 0 \text{ and } C_i \in L \text{ for } 1 \leq i \leq k\}$$

as the set of all graphs whose components are members of L.

A labeled graph G is called *connected* if and only if it consists of exactly one nonempty connected component. In the following we call a connected, nonempty and acyclic labeled graph also a *DAG* (*directed acyclic graph*) and denote the set of all DAGs over an alphabet Σ by D_Σ. A set of DAGs is also called a *DAG language*.

An *isomorphism* from a labeled graph G to a labeled graph G' is a tuple of bijections $g = (g_V, g_E)$ with $g_V : V_G \to V_{G'}$ and $g_E : E_G \to E_{G'}$ such that for every $v \in V_G$ we have $label_{G'}(g_V(v)) = label_G(v)$ and for $in_G(v) = (e_1, \ldots, e_n)$ and $out_G(v) = (f_1, \ldots, f_m)$ we have $in_{G'}(g_V(v)) = (g_E(e_1), \ldots, g_E(e_n))$ and $out_{G'}(g_V(v)) = (g_E(f_1), \ldots, g_E(f_m))$. Two labeled graphs G and G' are *isomorphic* if and only if there exists an isomorphism from G to G'. In that case we write also $G \simeq G'$. Two labeled graphs G and G' are moreover *disjoint* if and only if $V_G \cap V_{G'} = \emptyset$ and $E_G \cap E_{G'} = \emptyset$.

2.3 Operations on DAGs and DAG Languages

One can of course apply the boolean operations intersection, union and complement on DAG languages. These operations work however only on the set level, but from a language-theoretic perspective it is also interesting to investigate operations that modify the underlying DAGs. For strings such operations are for example concatenation or homomorphism and we will now introduce similar operations for DAG languages.

If we consider two strings ω_1 and ω_2, their concatenation is defined as $\omega_1 \circ \omega_2 = \omega_1\omega_2$, so ω_2 is simply placed after ω_1. For graphs the situation is slightly more complicated as a graph may contain more than one root and one leaf and it is in general not clear where a concatenation should take place. Engelfriet and Vereijken solve this problem in [9] by using graphs that have numbered begin and end vertices. During the concatenation of two graphs G_1 and G_2 the i-th end vertex of G_1 is then identified with the i-th begin vertex of G_2. As our labeled graphs lack such numbered begin and end vertices we have to use a slightly different approach, but the basic principle remains the same. When we concatenate two labeled graphs G_1 and G_2 we use an injective mapping $\mu : V_{G_1} \to V_{G_2}$ to specify which vertices are identified with each other. So if we have $\mu(v_1) = v_2$ for two vertices $v_1 \in V_{G_1}$ and $v_2 \in V_{G_2}$ both vertices will be merged into a new node v during the concatenation. To avoid that the concatenation creates a directed cycle we allow only leaves of G_1 to be merged with roots of G_2. We call such a mapping also a *concatenation mapping*. If $label_{G_1}(v_1) = \sigma_1 \in \Sigma_{n,0}$ and $label_{G_2}(v_2) = \sigma_2 \in \Sigma_{0,m}$ the new vertex v gets moreover the label $\sigma_1\sigma_2 \in \Sigma_{n,m}$. The labeled graph obtained by concatenating two labeled graphs G_1 and G_2 according to a concatenation mapping μ is also denoted by $G_1 \circ_\mu G_2$. An example can be seen in Figure 2.2.

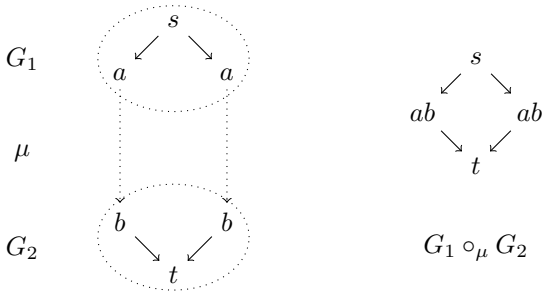


Figure 2.2: The concatenation of two DAGs

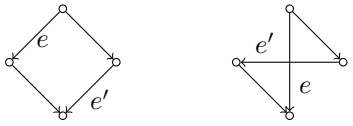
Now we extend the definition of concatenation on DAG languages. If L_1 and L_2 are DAG languages we define

$$L_1 \circ L_2 = \{G_1 \circ_\mu G_2 \mid G_1 \in L_1^{\&}, G_2 \in L_2^{\&}, \mu : V_{G_1} \to V_{G_2} \text{ is a concatenation mapping and } G_1 \circ_\mu G_2 \text{ is a DAG}\}.$$

This means that $L_1 \circ L_2$ contains all DAGs that can be obtained by concatenating two labeled graphs G_1 and G_2 whose connected components are members of L_1 and L_2 respectively.

Another operation that we can perform on labeled graphs is to swap two edges, i.e. to exchange their end vertices. An example of this is shown in Figure 2.3. The graph obtained by swapping two edges e and e' of a labeled graph G is also denoted by $G[e \bowtie e']$. Given two disjoint labeled graphs G_1 and G_2 we can also use such an edge

swap for connecting G_1 and G_2. If $e_1 \in E_{V_1}$ and $e_2 \in E_{V_2}$ are the two edges to be swapped, then the resulting graph is $(G_1 \& G_2)[e_1 \bowtie e_2]$.



G $\quad$ $G[e \bowtie e']$

Figure 2.3: An edge swap

An important property of the edge swap operation is that it does not create cycles when we connect two disjoint graphs as the following lemma shows.

Lemma 2.1. *Let G and G' be two disjoint acyclic labeled graphs and $e \in E_G$ and $e' \in E_{G'}$ be edges. Then the labeled graph $(G \& G')[e \bowtie e']$ is also acyclic.*

Proof. Let G and G' be two disjoint acyclic labeled graphs, let $e \in E_G$ and $e' \in E_{G'}$ be edges and assume that $(G \& G')[e \bowtie e']$ contains a cycle C. Then C must use the edges e and e', otherwise it would be a cycle in G or G'. When we swap e and e' a second time, $(G \& G')[e \bowtie e']$ falls again apart into the graphs G and G'. Moreover the cycle C falls apart into two cycles in G and G'. This is illustrated in Figure 2.4. Therefore our assumption that G and G are acyclic was wrong and $(G \& G')[e \bowtie e']$ needs to be acyclic. □

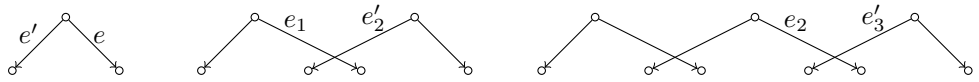
$(G \& G')[e \bowtie e']$ $\quad$ G $\quad$ G'

Figure 2.4: Construction of two cycles in G and G from a cycle in $(G \& G')[e \bowtie e']$

We can moreover connect k disjoint isomorphic copies of a labeled graph G by sytematically swapping copies of two edges $e, e' \in E_G$ between them. The resulting graph $G[e \bowtie e']^k$ is formally defined as

$$G[e \bowtie e']^1 = G$$
$$G[e \bowtie e']^{k+1} = (G[e \bowtie e']^k \& G_{k+1})[e_k \bowtie e'_{k+1}]$$

where G_{k+1} is an isomorphic copy of G disjoint with $G[e \bowtie e']^k$ and e_k and e'_k are the corresponding copies of e and e' in G_k for $k \geq 1$. An example for this is illustrated in Figure 2.5. One might note that this definition is ambigous as there is no unique isomorphic copy of G, but later on we will however consider every labeled graph just as a representative of its isomorphism class. Therefore we accept this small inaccuracy for the moment.



$G[e \bowtie e']^1$ $G[e \bowtie e']^2$ $G[e \bowtie e']^3$

Figure 2.5: The repeated edge swap

2.4 DAG Automata

Now we are ready to introduce DAG automata, an extension of finite-state string or tree automata that can also work on acyclic labeled graphs and DAGs.

Definition 2.2 (DAG automata)**.** A *DAG automaton* is a triple $A = (Q, \Sigma, R)$ where Q is a finite set of *states*, Σ is a doubly ranked alphabet and R is a set of *rules* of the form $\alpha \overset{\sigma}{\leftrightarrow} \beta$ where $\sigma \in \Sigma_{n,m}$ and $\alpha \in Q^n, \beta \in Q^m$.

For such a rule we call α the left-hand side and β the right-hand side. If a rule has an empty left- or right-hand side (in case the corresponding symbol has head or tail rank 0) this is denoted by λ. The semantics of a DAG automaton A working on an acyclic labeled graph G is defined as follows.

Definition 2.3 (Semantics of DAG automata)**.** Let $A = (Q, \Sigma, R)$ be a DAG automaton and $G = (V, E, label, in, out)$ be an acyclic labeled graph.

1. A *run* of the DAG automaton A on the graph G is a mapping $\rho : E \to Q$.

2. A run ρ of the DAG automaton A on the graph G is *accepting* if and only if for every vertex $v \in V$ with $in(v) = (e_1, \ldots, e_n)$ and $out(v) = (e'_1, \ldots, e'_m)$ there is a rule of the form $(p_1, \ldots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \ldots, q_m) \in R$ such that
   a) $label(v) = \sigma \in \Sigma_{n,m}$,
   b) there is a totally ordered set $(\pi_1, \ldots, \pi_n)$ which is compatible with $\mathcal{R}_{in}(\sigma)$ such that $\rho(e_i) = p_{\pi_i}$ for $1 \leq i \leq n$.
   c) there is a totally ordered set $(\pi'_1, \ldots, \pi'_m)$ which is compatible with $\mathcal{R}_{out}(\sigma)$ such that $\rho(e'_i) = q_{\pi'_i}$ for $1 \leq i \leq m$.

   If r is such a rule for a vertex v we say also that ρ uses r in v.

3. The DAG automaton A *accepts* G if and only if there is an accepting run ρ of A on G.

4. We define $L(A) = \{G \mid A \text{ accepts the DAG } G\}$ and say that A *recognizes* $L(A)$.

Remark 2.1. *According to the previous definition a DAG automaton can* run *on both connected and disconnected acyclic labeled graphs. It* recognizes *however only sets of DAGs, i.e. connected, nonempty acyclic labeled graphs. The restriction to connected graphs has some advantages as it allows us for example to have DAG automata that recognize a finite, nonempty language. If we allowed also disconnected graphs this would not be possible as one could simply duplicate the connected components of some accepted graph and the resulting graph would still be in the recognized language.*

The set of all acyclic labeled graphs that are accepted by a given DAG automaton A, including disconnected graphs, is in fact equal to $L(A)^{\&}$. From Definition 2.3 it follows

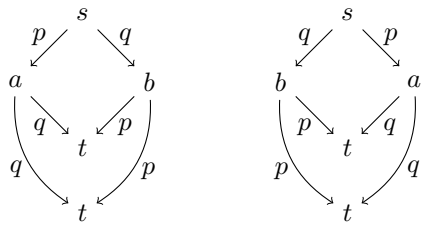
moreover that $G \in L(A)^{\&}$ if and only if for every connected component C of G we have $C \in L(A)$.

The restriction of $L(A)$ to nonempty *acyclic labeled graphs has the simple reason that the empty labeled graph $G_\emptyset$ is accepted by every DAG automaton with the run $\rho : \emptyset \to \emptyset$. This makes $G_\emptyset$ uninteresting for our investigations. By definition we have however $G_\emptyset \in L(A)^{\&}$ for every DAG automaton A.*

Remark 2.2. *If a DAG automaton A accepts an acyclic labeled graph G it accepts actually every labeled graph G' that is isomorphic to G as for an accepting run ρ of A on G and an isomorphism (g_V, g_E) from G' to G the run ρ' defined by $\rho'(e) = \rho(g_E(e))$ is an accepting run of A on G'. This is based on the fact that DAG automata work only on the structure of a given graph and its labels regardless of the names of the different vertices and edges. As previously mentioned we consider therefore every labeled graph G just as a representative of the isomorphism class $[G] = \{G' \mid G' \simeq G\}$ and confuse in the following G and $[G]$.*

To conclude this chapter we illustrate the operation of a DAG automaton on a given DAG with an example.

Example 2.2: Consider the DAG automaton $A = (Q, \Sigma, R)$ where $Q = \{p, q\}, \Sigma = \{s, a, b, t\}$ and $R = \{\lambda \overset{s}{\leftrightarrow} (p, q), (p) \overset{a}{\leftrightarrow} (q, q), (q) \overset{b}{\leftrightarrow} (p, p), (q, p) \overset{t}{\leftrightarrow} \lambda\}$. If we apply A to the graph G from Figure 2.1, we can construct an accepting run ρ of A on G as shown in Figure 2.6a.



(a) A run ρ (b) A run ρ'

Figure 2.6: Two runs of a DAG automaton

In the root v_1 of G the only rule we can use is $\lambda \overset{s}{\leftrightarrow} (p, q)$. The partial order $\mathcal{R}_{out}(s)$ now states in which order we may assign the states p and q to the outgoing edges e_1 and e_2 of v_1. In Example 2.1 we assumed the $\mathcal{R}_{out}(\sigma)$ is the $\leq$-relation for every symbol $\sigma \in \Sigma$. In that case we can only have $\rho(e_1) = p$ and $\rho(e_2) = q$. If we consider the left child of v_1 that has label a, we can use the rule $(p) \overset{a}{\leftrightarrow} (q, q)$ and obtain $\rho(e_3) = q$ and $\rho(e_4) = q$. In the other child of v_1 with label b we use the rule $(q) \overset{b}{\leftrightarrow} (p, p)$ and get $\rho(e_5) = p$ and $\rho(e_6) = p$. In the two root vertices the rule $(q, p) \overset{t}{\leftrightarrow} \lambda$ can then be used which means that ρ is an accepting run of the DAG automaton A on the DAG G and A accepts therefore G.

Assume now that $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ are the equality relation. This means that we can assign the states that occur on the left-hand and right-hand side of a rule in any order to the incident edges of every vertex. The DAG automaton A accepts in that case therefore also the DAG illustrated in Figure 2.6b.

Figure 2.7 shows two more DAGs accepted by the DAG automaton A and the corresponding accepting runs. It illustrates also 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.

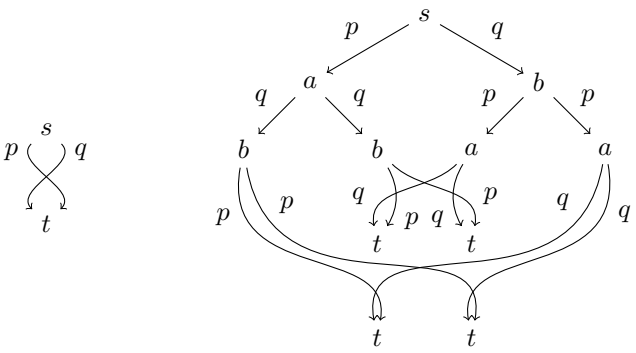


Figure 2.7: Two further runs

3 Ordered, Unordered and General DAGs

In the previous chapter we defined a doubly ranked alphabet as a set of symbols where each symbol σ is equipped with two partial orders $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$. Given a run of a DAG automaton on some DAG, the partial orders specify how the states of the in- and outgoing edges of every vertex can be reordered in order to match them to a rule of the automaton.

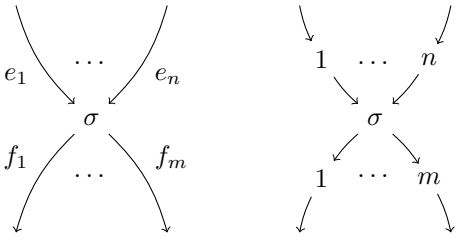
Quernheim and Knight [15] and Chiang et al. [6] considered however just ordered and unordered DAGs, respectively, that do not require the specification of such partial orders. In the ordered case the in- and outgoing edges of every vertex and the left- and right-hand sides of every rule are given as tuples of edges and states and a run is accepting iff the states assigned to the incident edges of every vertex match the states of some rule in the same order. In the unordered case, the incident edges of every vertex are given as sets whereas the left- and right-hand sides of every rule are given as multisets of states. In that case a run is accepting iff for every vertex the states of the incident edges match the states of some rule in an arbitrary order.

To embed these concepts into our DAG model, we call a labeled graph over an alphabet Σ *ordered* if and only if $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ are total orders for every symbol $\sigma \in \Sigma$. A labeled graph is moreover *unordered* if and only if for every symbol $\sigma \in \Sigma$ both $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ are the equality relation. This means that there is only one way to order the states of an accepting run on an ordered DAG (the i-th edge must have the i-th state) whereas for an unordered DAG every reordering is compatible with $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$. When we refer to any arbitrary DAG regardless of the partial orders of its alphabet we call it also a *general DAG*.

The advantage of this approach that adds partial orders to the alphabet is that it is more general and includes both ordered and unordered DAGs. It covers moreover also mixed cases where the ingoing edges of every vertex are for example unordered whereas the outgoing states are ordered. In the following we show however that ordered, unordered and general DAGs are actually equivalent, i.e. that any given DAG language L can be transformed into an ordered or unordered DAG language L' such that given a DAG automaton A with $L(A) = L$ we can construct a DAG automaton A' with $L(A') = L'$. This motivates also why we will only consider ordered DAGs in the following chapters.

We start by showing the equivalence of general and unordered DAGs. Consider therefore an arbitrary DAG G. Then we can transform G into an unordered DAG $U(G)$ as follows. For every vertex $v \in V_G$ with n ingoing and m outgoing edges we insert a vertex u_i with label i into the i-th ingoing edge of v for $i \in \{1, \ldots, n\}$. In the same way we insert vertices $v_1, \ldots, v_m$ with labels $1, \ldots, m$ into the outgoing edges of v. This is illustrated in Figure 3.1. Observe that we actually insert two vertices into every edge as each edge is incident to one start and one end vertex. Moreover we modify the alphabet Σ of G such that $U(G)$ is actually an unordered DAG. Therefore we set $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ to the equality relation for all $\sigma \in \Sigma$. Furthermore we add the symbols $\{1, \ldots, k\}$ to

$\Sigma_{1,1}$ where k is the maximum of all head and tail ranks of Σ under the assumption that $\{1, \ldots, k\}$ and Σ are disjoint.



(a) A vertex of G (b) The vertices of $U(G)$

Figure 3.1: Construction of the unordered DAG $U(G)$ from a DAG G

This construction is injective, i.e. for every (unordered) DAG G' there is at most one DAG G with $G' = U(G)$. We can also apply it to DAG languages and define $U(L) = \{U(G) \mid G \in L\}$. The following theorem shows now that the encoding of general DAGs by unordered DAGs that is provided by U preserves recognizability.

Theorem 3.1. *Let A be a DAG automaton. Then we can construct a DAG automaton A_u such that $L(A_u) = U(L(A))$.*

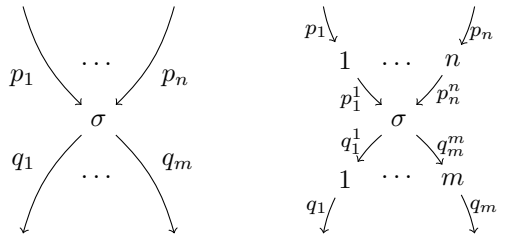
Proof. Let $A = (Q, \Sigma, R)$ be a DAG automaton. Then we construct $A_u = (Q_u, \Sigma_u, R_u)$ as follows. For every rule $(p_1, \ldots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \ldots, q_m) \in R$ and all totally ordered sets $(\pi_1, \ldots, \pi_n)$ and $(\pi'_1, \ldots, \pi'_m)$ that are compatible with $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ respectively we have rules

$$(p_{\pi_i}) \overset{i}{\leftrightarrow} (p^i_{\pi_i}),$$
$$(p^1_{\pi_1}, \ldots, p^n_{\pi_n}) \overset{\sigma}{\leftrightarrow} (q^1_{\pi'_1}, \ldots, q^m_{\pi'_m}) \text{ and}$$
$$(q^j_{\pi'_j}) \overset{j}{\leftrightarrow} (q_{\pi'_j})$$

in R_u for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$.

Let now G be a DAG. We show that A accepts G if and only if A_u accepts $U(G)$. For the only if part assume that A accepts G. This means there is an accepting run ρ of A on G. Now we construct an accepting run ρ_u of A_u on $U(G)$. Consider therefore a vertex $v \in V_G$ with $label_G(v) = \sigma$, $in_G(v) = (e_1, \ldots, e_n)$ and $out_G(v) = (f_1, \ldots, f_m)$ and let $\rho(e_i) = p_i$ and $\rho(f_j) = q_j$ for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$. During the construction of $U(G)$ a vertex u_i was inserted into every edge e_i such that e_i is now split into two edges e^1_i and e^2_i with $in_{U(G)}(u_i) = e^1_i$ and $out_{U(G)}(u_i) = e^2_i$. For these two edges we set now $\rho_u(e^1_i) = p_i$ and $\rho_u(e^2_i) = p^i_i$. In a similar way we set the states of the in- and outgoing edge of each vertex v_j that was inserted into the j-th outgoing edge of v to q^j_j and q_j respectively. This is also illustrated in Figure 3.2.

The run ρ is accepting which means that the totally ordered sets $(1, \ldots, n)$ and $(1, \ldots, m)$ are compatible with $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ respectively. Therefore we have



(a) Run ρ of A on G (b) Run ρ_u of A_u on $U(G)$

Figure 3.2: Two corresponding runs of the DAG automata A and A_u

rules

$$(p_i) \overset{i}{\leftrightarrow} (p_i^i),$$
$$(p_1^1, \ldots, p_n^n) \overset{\sigma}{\leftrightarrow} (q_1^1, \ldots, q_m^m) \text{ and}$$
$$(q_j^j) \overset{j}{\leftrightarrow} (q_j)$$

in R_u for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$. To show that ρ_u is accepting consider now a vertex $v' \in V_{U(G)}$. If v' was inserted into an ingoing edge of a vertex $v \in V_G$, we have $\mathit{label}(v') = i \in \{1, \ldots, n\}$ and we can apply the rule $(p_i) \overset{i}{\leftrightarrow} (p_i^i)$ to v' as the states of its in- and outgoing edges match p_i and p_i^i. Analogously we can also apply the rule $(q_j^j) \overset{j}{\leftrightarrow} (q_j)$ if v' was inserted into the j-th outgoing edge. If we have moreover $v' \in V$, we can apply $(p_1^1, \ldots, p_n^n) \overset{\sigma}{\leftrightarrow} (q_1^1, \ldots, q_m^m)$ which means that we have a matching rule for every vertex of $U(G)$, so ρ_u is an accepting run of A_u on $U(G)$.

For the if part assume that there is an accepting run ρ_u of A_u on $U(G)$. Then we can construct the following accepting run ρ of A on G. Consider a vertex $v \in V_G$ with $\mathit{label}_G(v) = \sigma$ and let $\mathit{in}_{U(G)}(v) = (e_1, \ldots, e_n)$ and $\mathit{out}_{U(G)}(v) = (f_1, \ldots, f_m)$. As the run ρ_u is accepting there must be a rule $(p_1^1, \ldots, p_n^n) \overset{\sigma}{\leftrightarrow} (q_1^1, \ldots, q_m^m) \in R_u$ such that $\rho_u(e_i) = p_i^i$ and $\rho_u(f_j) = q_j^j$ for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$. Moreover there must also be rules $(p_i) \overset{i}{\leftrightarrow} (p_i^i)$ and $(q_j^j) \overset{j}{\leftrightarrow} (q_j)$ that comply with the run ρ_u for $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$.

Let now $\mathit{in}_G(v) = (e_1, \ldots, e_n)$ and $\mathit{out}_G(v) = (f_1, \ldots, f_m)$. Then we set $\rho(e_i) = p_i$ for $i \in \{1, \ldots, n\}$ and $\rho(f_j) = q_j$ for $j \in \{1, \ldots, m\}$. Because we have $(p_1^1, \ldots, p_n^n) \overset{\sigma}{\leftrightarrow} (q_1^1, \ldots, q_m^m) \in R_u$, the totally ordered sets $(p_1, \ldots, p_n)$ and $(q_1, \ldots, q_m)$ are compatible with $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ and we have $(p_1, \ldots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \ldots, q_m) \in R$. Therefore we can apply a rule of A in every vertex of G which means that ρ is an accepting run of A on G. Hence A accepts a DAG G if and only if A_u accepts $U(G)$ and we have $L(A_u) = U(L(A))$. □

Thus, we have shown that, in principle, it would suffice to consider unordered DAGs. In a similar but simpler way we can show that it suffices to consider ordered DAGs. Given some DAG G over an alphabet Σ we can easily turn it into an ordered DAG $O(G)$ just by replacing the partial orders $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ of every symbol $\sigma \in \Sigma$ with $\leq$, the less-or-equal relation on the natural numbers. Analogously to the unordered case we define $O(L) = \{O(G) \mid G \in L\}$. This construction preserves recognizability as

the following theorem shows.

Theorem 3.2. *Let A be a DAG automaton. Then we can construct a DAG automaton A_o such that $L(A_o) = O(L(A))$.*

Proof. Let $A = (Q, \Sigma, R)$ be a DAG automaton. Then we construct $A_o = (Q, \Sigma_o, R_o)$ as follows. For every rule $(p_1, \ldots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \ldots, q_m) \in R$ and all totally ordered sets $(\pi_1, \ldots, \pi_n)$ and $(\pi'_1, \ldots, \pi'_m)$ that are compatible with $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ we have $(p_{\pi_1}, \ldots, p_{\pi_n}) \overset{\sigma}{\leftrightarrow} (q_{\pi'_1}, \ldots, q_{\pi'_m}) \in R_o$. In other words, we simply include all rules whose orders are compatible with those given by Σ.

Let now G be a DAG. We show that A accepts G if and only if A_o accepts $O(G)$. For the only if part assume that there is an accepting run ρ of A on G and consider a vertex $v \in V_G$ with $label_G(v) = \sigma$, $in_G(v) = (e_1, \ldots, e_n)$ and $out_G(v) = (f_1, \ldots, f_m)$. As ρ is accepting there is a rule $(p_1, \ldots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \ldots, q_m) \in R$ and totally ordered sets $(\pi_1, \ldots, \pi_n)$ and $(\pi'_1, \ldots, \pi'_m)$ which are compatible with $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$ such that $\rho(e_i) = p_{\pi_i}$ and $\rho(f_j) = q_{\pi'_j}$. This means that we have $(p_{\pi_1}, \ldots, p_{\pi_n}) \overset{\sigma}{\leftrightarrow} (q_{\pi'_1}, \ldots, q_{\pi'_m}) \in R_o$. Moreover the totally ordered sets $(1, \ldots, n)$ and $(1, \ldots, m)$ are compatible with $\leq$ and we have $\rho(e_i) = p_{\pi_i}$ and $\rho(f_j) = q_{\pi'_j}$, so ρ is also an accepting run of A_o on $O(G)$.

For the if part assume that there is an accepting run ρ of A_o on $O(G)$ and consider a vertex $v \in V_G$ with $label_G(v) = \sigma$, $in_G(v) = (e_1, \ldots, e_n)$ and $out_G(v) = (f_1, \ldots, f_m)$. Then there is a rule $(p_{\pi_1}, \ldots, p_{\pi_n}) \overset{\sigma}{\leftrightarrow} (q_{\pi'_1}, \ldots, q_{\pi'_m}) \in R_o$ such that $\rho(e_i) = p_{\pi_i}$ and $\rho(f_j) = q_{\pi_j}$ and such that there is a rule $(p_1, \ldots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \ldots, q_m) \in R$ for which $(\pi_1, \ldots, \pi_n)$ and $(\pi'_1, \ldots, \pi'_m)$ are totally ordered sets that are compatible with $\mathcal{R}_{in}(\sigma)$ and $\mathcal{R}_{out}(\sigma)$. This means that ρ is also an accepting run of A on G. Therefore A accepts G if and only if A_o accepts $O(G)$ and we have $L(A_o) = O(L(A))$. □

So from Theorems 3.1 and 3.2 we can draw the conclusion that it is actually sufficient to consider only unordered or ordered DAGs. Therefore we will limit our considerations in the following chapters to the ordered case and assume from now on that every DAG is ordered unless stated otherwise. The decision for the ordered case is however not totally arbitrary as the only situation where it is really meaningful to distinguish between determinism and nondeterminism is when we have ordered DAGs and the order of all states is fixed. What this actually means will be clarified in the next chapter. All of the following results hold nevertheless also for unordered or general DAGs except those that require inevitably determinism. Conversely all results that can be shown for unordered DAGs hold also for ordered or general DAGs.

4 Determinism

In the previous chapter we already mentioned the term determinism briefly. Intuitively a deterministic DAG automaton A allows the stepwise construction of an accepting run on a given DAG $G \in L(A)$ without having to guess at some point because there are several rules that could be applied next. Depending on the direction in which one can construct such a run we distinguish between top-down determinism where we start with the root vertices and bottom-up determinism where we start with the leaf vertices. Formally we define top-down and bottom-up deterministic DAG automata as follows.

Definition 4.1 (Determinism)**.** Let $A = (Q, \Sigma, R)$ be a DAG automaton.

1. A is *top-down deterministic* iff for every left-hand side α and every $\sigma \in \Sigma$ there is at most one rule of form $\alpha \overset{\sigma}{\leftrightarrow} \beta \in R$.

2. A is *bottom-up deterministic* iff for every right-hand side β and every $\sigma \in \Sigma$ there is at most one rule of form $\alpha \overset{\sigma}{\leftrightarrow} \beta \in R$.

A *nondeterministic* DAG automaton is moreover a normal DAG automaton without any such restriction, that can, but does not need to be top-down or bottom-up deterministic. Based on these definitions we introduce the following classes of DAG languages.

Definition 4.2 (DAG language classes)**.**

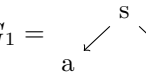
1. $L(ND)$ is the set of languages that can be recognized by nondeterministic DAG automata.

2. $L(DTD)$ is the set of languages that can be recognized by top-down deterministic DAG automata.

3. $L(DBD)$ is the set of languages that can be recognized by bottom-up deterministic DAG automata.

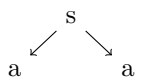
From the previous definitions it follows that $L(DTD) \subseteq L(ND)$ and $L(DBD) \subseteq L(ND)$. In the following we will see that these inclusions are actually strict. We consider at first the top-down case and adapt therefore a well know example that was originally used to separate top-down deterministic tree automata from nondeterministic tree automata.

Lemma 4.1. *We have* $L(DTD) \subsetneq L(ND)$.

Proof. Consider the DAGs $G_1 =$  and $G_2 =$  and the language $L = \{G_1, G_2\}$. We have $L \in L(ND)$ as for the DAG automaton

$$A = (\{p, q\}, \{s, a, b\}, \{\lambda \overset{s}{\leftrightarrow} (p, q), \lambda \overset{s}{\leftrightarrow} (q, p), p \overset{a}{\leftrightarrow} \lambda, q \overset{b}{\leftrightarrow} \lambda\})$$

we have $L(A) = L$.

There is however no top-down deterministic DAG automaton that recognizes L. To show this assume that there is a top-down deterministic DAG automaton $A = (Q, \Sigma, R)$ with $L(A) = L$. Then A accepts G_1 which means that there are rules $\lambda \overset{s}{\leftrightarrow} (p_1, p_2), p_1 \overset{a}{\leftrightarrow} \lambda, p_2 \overset{b}{\leftrightarrow} \lambda \in R$ for some states $p_1, p_2 \in Q$. Moreover A accepts G_2 which means that there are also rules $\lambda \overset{s}{\leftrightarrow} (p_3, p_4), p_3 \overset{b}{\leftrightarrow} \lambda, p_4 \overset{a}{\leftrightarrow} \lambda \in R$. But A is top-down deterministic which actually means that $p_1 = p_3$ and $p_2 = p_4$, so we have $\lambda \overset{s}{\leftrightarrow} (p_1, p_2), (p_1) \overset{a}{\leftrightarrow} \lambda, (p_2) \overset{a}{\leftrightarrow} \lambda \in R$. Therefore A also accepts the DAG 

and it follows that $L(A) \neq L$. □

To show that $L(DBD) \subsetneq L(ND)$ one could now simply turn the DAGs from the previous proof upside down by switching the direction of every edge which transforms roots into leaves and leaves into roots. Considering the reverse DAG does however not only work in that particular case. It is actually a basic property of the chosen DAG model that we can turn every DAG upside down. To show this we introduce the concept of *duality* as follows.

Definition 4.3. 1. For a symbol $\sigma = (id(\sigma), \mathcal{R}_{in}(\sigma), \mathcal{R}_{out}(\sigma))$ the *dual symbol* is $dual(\sigma) = (id(\sigma), \mathcal{R}_{out}(\sigma), \mathcal{R}_{in}(\sigma))$.

2. Let $G = (V, E, label, in, out)$ be a DAG. Then $dual(G) = (V, E, label_d, out, in)$ is the *dual DAG* of G which we obtain by switching the directions of the edges of G and defining $label_d(v) = dual(label(v))$.

3. Let L be a DAG language. Then $dual(L) = \{dual(G) \mid G \in L\}$ is the *dual DAG language* of L.

4. Let $\mathcal{L}$ be a class of DAG languages. Then we define $dual(\mathcal{L}) = \{dual(L) \mid L \in \mathcal{L}\}$.

5. Let $A = (Q, \Sigma, R)$ be a DAG automaton. Then $dual(A) = (Q, \Sigma_d, R_d)$ is the *dual DAG automata* of A where $\Sigma_d = \{dual(\sigma) \mid \sigma \in \Sigma\}$ and $\beta \overset{dual(\sigma)}{\longleftrightarrow} \alpha \in R_d$ iff $\alpha \overset{\sigma}{\leftrightarrow} \beta \in R$, i.e. we obtain $dual(A)$ by swapping the left- and right-hand sides of the rules of A.

We can easily see that *dual* is an *involution* which means that we have $dual(dual(G)) = G$ for all DAGs G. The same holds for DAG automata and it follows moreover from Definition 4.3 that a DAG automaton A is top-down deterministic if and only if its dual DAG automaton $dual(A)$ is bottom-up deterministic.

As a consequence of this, top-down and bottom-up DAG automata are actually equally powerful as the following lemma shows.

Lemma 4.2. *Let A be a DAG automaton. Then we have $dual(L(A)) = L(dual(A))$.*

Proof. Let $A = (Q, \Sigma, R)$ be a DAG automaton. We show at first that $dual(L(A)) \subseteq L(dual(A))$. Consider therefore a DAG $G \in L(A)$. Then there is a run ρ of A on G such that for every vertex $v \in V_G$ with $label_G(v) = \sigma$, $in_G(v) = (e_1, \ldots, e_n)$ and $out_G(v) = (f_1, \ldots, f_m)$ there is a rule $(p_1, \ldots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \ldots, q_m) \in R$ such that $\rho(e_i) = p_i$ and $\rho(f_i) = q_i$. For the corresponding vertex v of the dual DAG $G' = dual(G)$ we have then $label_{G'}(v) = dual(\sigma)$, $in_{G'}(v) = (f_1, \ldots, f_m)$ and $out_{G'}(v) = (e_1, \ldots, e_n)$. Moreover there is a rule $(q_1, \ldots, q_m) \overset{dual(\sigma)}{\longleftrightarrow} (p_1, \ldots, p_n) \in R_d$, so ρ is an accepting run of $dual(A)$ on G' and we have $dual(L(A)) \subseteq L(dual(A))$. As *dual* is an involution it follows now that $L(dual(A)) \subseteq dual(L(A))$. Therefore we have $dual(L(A)) = L(dual(A))$. □

This means that all results for top-down deterministic DAG automata hold for bottom-up DAG deterministic automata and vice versa with respect to duality. Therefore it follows for example from Lemma 4.1 that $L(DBD) \subsetneq L(NB)$ as previously mentioned.

The following theorem concludes our discussion of deterministic automata and summarizes the previous observations.

Theorem 4.1. *1. We have $L(DTD) \cup L(DBD) \subsetneq L(ND)$.*

2. We have $L(DTD) = dual(L(DBD))$ and $L(DBD) = dual(L(DTD))$.

3. $L(DTD)$ and $L(DBD)$ are incomparable.

Proof. 1. Consider the language L from the proof of Lemma 4.1 and let $L' = L \cup dual(L)$. We have $L' \notin L(DTD)$ as the proof of Lemma 4.1 shows that every top-down deterministic DAG automaton A that recognized L' would also accept the DAG 

$\notin L'$. We have moreover $L' = dual(L')$ which means that $L' \notin L(DBD)$. Therefore we have $L' \notin L(DTD) \cup L(DBD)$. Nevertheless $L' \in L(ND)$ holds as we can easily combine a DAG automaton A that recognizes L with its dual automaton $dual(A)$ to a DAG automaton A' with $L(A') = L'$ (see also Theorem 5.1).

2. Follows from Lemma 4.2.

3. Consider the language L from the proof of Lemma 4.1. We have $L \notin L(DTD)$ but the constructed DAG automaton A with $L(A) = L$ is actually bottom-up deterministic. Therefore we have $L \in L(DBD)$. Moreover we have $dual(L) \notin L(DBD)$ but $dual(L) \in L(DTD)$, so $L(DTD)$ and $L(DBD)$ are incomparable.

□

5 Properties of Recognizable DAG Languages

5.1 Closure Properties

In this section we will investigate the closure properties of DAG languages. We consider at first the set theoretic operations intersection, union and complement before showing closedness under concatenation. Cheratonik [5] and Chiang et al. [6] showed already that the class of recognizable DAG languages is closed under intersection and union. For completeness we present their proofs briefly which apply basically the standard construction for tree automata (cf. [7]).

Theorem 5.1. *1. $L(ND)$ and $L(DTD)$ are closed under intersection.*

2. $L(ND)$ is closed under union.

Proof. Let $L_1, L_2 \in L(ND)$ be DAG languages and let $A_1 = (Q_1, \Sigma, R_1)$ and $A_2 = (Q_2, \Sigma, R_2)$ be DAG automata with $L_1 = L(A_1)$ and $L_2 = L(A_2)$. We assume moreover that Q_1 and Q_2 are disjoint. This is possible as we can just rename the states in Q_2.

1. The following DAG automaton $A_\cap = (Q_1 \times Q_2, \Sigma, R_\cap)$ recognizes $L_1 \cap L_2$.

   For every symbol σ in $\Sigma_{n,m}$ and all rules $(p_1^1, \dots p_n^1) \overset{\sigma}{\leftrightarrow} (q_1^1, \dots, q_m^1) \in R_1$ and $(p_1^2, \dots p_n^2) \overset{\sigma}{\leftrightarrow} (q_1^2, \dots, q_m^2) \in R_2$ we let $(p_1^1 p_1^2, \dots p_n^1 p_n^2) \overset{\sigma}{\leftrightarrow} (q_1^1 q_1^2, \dots, q_m^1 q_m^2) \in R_\cap$. If ρ is an accepting run of $A_\cap$ on a DAG G with $\rho(e_i) = p_i^1 p_i^2$, the runs ρ_1 and ρ_2 with $\rho_1(e_i) = p_i^1$ and $\rho_2(e_i) = p_i^2$ are accepting runs of A_1 and A_2 on G and for accepting runs ρ_1 and ρ_2 of A_1 and A_2 on G with $\rho_1(e_i) = p_i^1$ and $\rho_2(e_i) = p_i^2$ the run ρ with $\rho(e_i) = p_i^1 p_i^2$ is an accepting run of $A_\cap$ on a DAG G.

   Therefore we have $L(A_\cap) = L_1 \cap L_2$ and $L(ND)$ is closed under intersection.

   If A_1 and A_2 are top-down deterministic, the constructed DAG automaton $A_\cap$ is also top-down deterministic which means the same holds for $L(DTD)$.

2. For the DAG automaton $A_\cup = (Q_1 \cup Q_2, \Sigma, R_1 \cup R_2)$ we have $L(A_\cup) = L_1 \cup L_2$.

   To show this assume that ρ is an accepting run of $A_\cup$ on a DAG G. As Q_1 and Q_2 are disjoint and G is connected, the run ρ uses only rules of either R_1 or R_2, so ρ is an accepting run of A_1 or A_2 on G. Every accepting run ρ of A_1 or A_2 on a DAG G is moreover also an accepting run of $A_\cup$ on G, so we have $L(A_\cup) = L_1 \cup L_2$ and $L(ND)$ is closed under union.

□

Remark 5.1. *Closedness under union holds nevertheless not for $L(DTD)$. To show this consider the two DAG languages*

$$L_1 = \left\{ \begin{array}{ccc} & s & \\ \swarrow & & \searrow \\ a & & b \end{array} \right\} \text{ and } L_2 = \left\{ \begin{array}{ccc} & s & \\ \swarrow & & \searrow \\ b & & a \end{array} \right\}$$

which can both be recognized of a top-down deterministic DAG automaton. For their union we have however $L_1 \cup L_2 \notin L(DTD)$ as we can see in the proof of Lemma 4.1.

Remark 5.2. *Another interesting observation is that $L(ND)^{\&}$ defined by $L(ND)^{\&} = \{L^{\&} \mid L \in L(ND)\}$ is not closed under union either. This can be shown by the following example. Consider two recognizable DAG languages $L_1 = \{G_1\}$ and $L_2 = \{G_2\}$ where G_1 and G_2 are DAGs that are not isomorphic. For these languages we have $L_1^{\&} \in L(ND)^{\&}$ and $L_2^{\&} \in L(ND)^{\&}$. Consider now a DAG automaton A that accepts a graph G if and only if $G \in L_1^{\&} \cup L_2^{\&}$. Then A accepts also $(G_1 \& G_2)$ as we can mix connected components of graphs accepted by A, but we have $(G_1 \& G_2) \notin L_1^{\&} \cup L_2^{\&}$.*

Now we show that DAG automata are not closed under complement, in contrast to finite-state string and tree automata. The idea is to consider an alphabet $\Sigma = \{s, a, b\}$ and the DAG language $D_{\{s,a\}}$ which is recognizable. Then the complement of L consists of the DAGs which contain at least one b, a language which is not recognizable as Potthoff et al. showed in [14]. This is caused by the fact that DAG automata are not able to distinguish between edges with the same state as the following lemma shows.

Lemma 5.1. *Let A be a DAG automaton and let $G \in L(A)^{\&}$. Then for all edges $e_1, e_2 \in E_G$ and every accepting run ρ of A on G with $\rho(e_1) = \rho(e_2)$ we have $G[e_1 \bowtie e_2] \in L(A)^{\&}$.*

Proof. Let A be a DAG automaton, $G \in L(A)^{\&}$ be an acyclic labeled graph and $e_1, e_2 \in E_G$ edges for which there is an accepting run ρ of A on G with $\rho(e_1) = \rho(e_2)$. If we swap the two edges e_1 and e_2, the states of the in- and outgoing edges do not change for any vertex. Therefore ρ is also an accepting run of A on $G[e_1 \bowtie e_2]$ and we have $G[e_1 \bowtie e_2] \in L(A)^{\&}$. □

This is an important property of DAG automata we will reuse several times later on. With its help we can now show the following theorem.

Theorem 5.2. *$L(ND)$ and $L(DTD)$ are not closed under complement.*

Proof. Consider the alphabet $\Sigma = \Sigma_{0,2} \cup \Sigma_{2,0}$ where $\Sigma_{0,2} = \{s\}$ and $\Sigma_{2,0} = \{a, b\}$ and let $L = D_{\{s,a\}}$ be the set of all DAGs over Σ which do not contain a vertex with label b. Then we have $L \in L(DTD)$ as for the top-down deterministic DAG automaton $A = (\{p\}, \Sigma, \{\lambda \overset{s}{\leftrightarrow} (p,p), (p,p) \overset{a}{\leftrightarrow} \lambda\})$ we have $L = L(A)$.

Now consider the complement $\bar{L}$ of L with respect to D_Σ which consists of all DAGs over Σ that contain at least one vertex with label b. In the following we will show that there is no DAG automaton that recognizes $\bar{L}$. Assume therefore that there is a DAG automaton $\bar{A}$ such that $L(\bar{A}) = \bar{L}$. Consider the following generic DAG G_i with $V_{G_i} = \{s_0, \ldots, s_{i+1}, a_0, \ldots, a_i, b\}$. Every vertex s_j has label s for $j \in \{0, \ldots, i+1\}$, every vertex a_j has label a for $j \in \{0, \ldots, i\}$ and vertex b has label b. Moreover every vertex a_j has two ingoing edges e_j^1 and e_j^2 from $s_{j-1 \bmod (i+2)}$ and s_j respectively and vertex b has two ingoing edges from s_i and s_{i+1}. This is also illustrated in Figure 5.1.

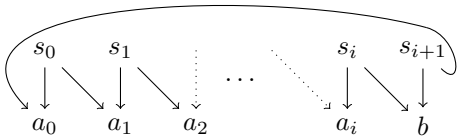


Figure 5.1: The generic DAG G_i

For every i we obviously have $G_i \in \bar{L}$ which means that $\bar{A}$ accepts G_i. We can now choose a value c such that for every run ρ of $\bar{A}$ on G_c there are two vertices a_k and a_l such that the states of their first ingoing edges are identical, i.e. $\rho(e_k^1) = \rho(e_l^1)$. This is possible as $\bar{A}$ has only a finite number of states.

Now let ρ be an accepting run of $\bar{A}$ on G_c and let a_k and a_l be two vertices of G_c with $\rho(e_k^1) = \rho(e_l^1) = p$. If we swap the two edges e_k^1 and e_l^1, the DAG G_c falls apart into two connected components, i.e. we have $G_c[e_k^1 \bowtie e_l^1] = C_1 \& C_2$ for two DAGs C_1 and C_2. This is illustrated in Figure 5.2.

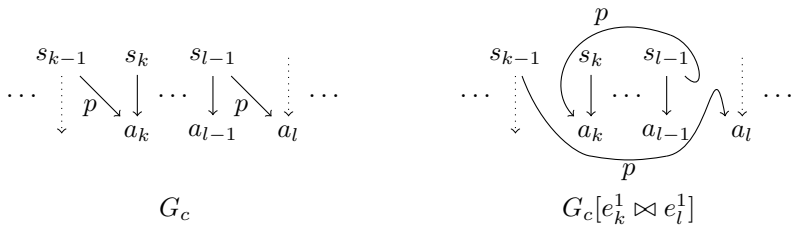


Figure 5.2: The DAG G_c and the labeled graph $G_c[e_k^1 \bowtie e_l^1]$

According to Lemma 5.1 we have $(C_1 \& C_2) \in L(\bar{A})^{\&}$ which means that $C_1 \in L(\bar{A})$ and $C_2 \in L(\bar{A})$. But one DAG $C \in \{C_1, C_2\}$ contains only vertices with labels s and a, so we actually have $C \notin \bar{L}$. This means that $\bar{L} \neq L(\bar{A})$, so there is no DAG automaton which recognizes $\bar{L}$. Therefore $L(ND)$ and $L(DTD)$ are not closed under complement. □

Finally we consider concatenation and prove the closedness of $L(ND)$ and $L(DTD)$ under this operation by showing that two DAG automata A_1 and A_2 can be combined such that the resulting automaton recognizes the language $L(A_1) \circ L(A_2)$.

Theorem 5.3. *$L(ND)$ and $L(DTD)$ are closed under concatenation.*

Proof. Let $L_1, L_2 \in L(ND)$ be DAG languages and $A_1 = (Q_1, \Sigma_1, R_1)$ and $A_2 = (Q_2, \Sigma_2, R_2)$ be DAG automata with $L_1 = L(A_1)$ and $L_2 = L(A_2)$ such that Q_1 and Q_2 are disjoint. Then we construct the following DAG automaton $A = (Q, \Sigma, R)$. For every rule $r \in R_1 \cup R_2$ we have $r \in R$, if moreover $\alpha \overset{\sigma_1}{\leftrightarrow} \lambda \in R_1$ and $\lambda \overset{\sigma_2}{\leftrightarrow} \beta \in R_2$ we let $\alpha \overset{\sigma_1\sigma_2}{\longleftrightarrow} \beta \in R$. Now we show that $L(A) = L_1 \circ L_2$.

Consider a DAG $G \in L_1 \circ L_2$. Then the DAG G can be obtained by concatenating k DAGs $C_1, \ldots, C_k \in L_1 \cup L_2$. For all $i \in \{1, \ldots, k\}$ there is moreover an accepting run ρ_i of A_1 or A_2 on C_i. The run ρ with

$$\rho(e) = \rho_i(e) \text{ if } e \in E_{C_i}$$

is now an accepting run of A on G. To show this consider a vertex v of G. If v was part of the DAG C_i before the concatenation and has not been merged with another vertex during the concatenation we can apply the same rule to v that was used for the run ρ_i on C_i as we have $\rho(e) = \rho_i(e)$ for all incident edges e of v.

If v has been created by merging two vertices v_1 and v_2 of two DAGs $C_i \in L_1$ and $C_j \in R_2$ and the rules used by the accepting runs ρ_i and ρ_j on C_i and C_j were $(p_1, \ldots, p_n) \overset{\sigma_1}{\leftrightarrow} \lambda$ and $\lambda \overset{\sigma_2}{\leftrightarrow} (q_1, \ldots, q_m)$ respectively we can now use the rule $(p_1, \ldots, p_n) \overset{\sigma_1\sigma_2}{\longleftrightarrow} (q_1, \ldots, q_m)$ because we have $\rho(e) = \rho_i(e)$ for all ingoing edges e of v and $\rho(e) = \rho_j(e)$ for all outgoing edges e of v.

Consider now a DAG $G \in L(A)$. To show that $G \in L_1 \circ L_2$ we perform the concatenation backwards by splitting G into k DAGs $C_1, \ldots, C_k$. This is done by splitting every vertex v of G which has a label $\sigma_1\sigma_2 \in \Sigma_1 \times \Sigma_2$ into an upper vertex v_1 with label σ_1 and $in(v_1) = in(v)$ and a lower vertex v_2 with label σ_2 and $out(v_2) = out(v)$.

Then for every such C_i we have $C_i \in L_1$ or $C_i \in L_2$. To show this consider an accepting run ρ of A on G and a vertex v of some DAG C_i. If v exists also in G we can apply the same rule to v that was used in the run ρ. If v was created by splitting a vertex v' of G and ρ used the rule $\alpha \overset{\sigma_1\sigma_2}{\longleftrightarrow} \beta$, we can now apply the rule $\alpha \overset{\sigma_1}{\leftrightarrow} \lambda$ if v is the upper vertex or $\lambda \overset{\sigma_2}{\leftrightarrow} \beta$ if v is the lower vertex. Therefore we have $C_i \in L(A_1) \cup L(A_2)$. Moreover we can obtain G by concatenating $C_1, \ldots, C_k$, so we have $G \in L_1 \circ L_2$ and therefore $L(A) = L_1 \circ L_2$. □

The closure properties of $L(ND)$ and $L(DTD)$ shown in this section are summarized in Table 5.1

	$L(ND)$	$L(DTD)$
$\cap$	yes	yes
$\cup$	yes	no
$^-$	no	no
$\circ$	yes	yes

Table 5.1: Closure properties of DAG language classes

5.2 Necessary Properties

In this section we present two necessary properties for recognizable DAG languages. Like the pumping lemmas for regular languages or recognizable tree languages are these based on the fact that a DAG automaton has to reuse some states if the DAG it works on becomes too big. Then we can pump the DAG up by connecting it to an isomorphic copy through an edge swap without changing the acceptance behaviour of the automaton. This is based on the concept that DAG automata are not able to distinguish between edges with the same state as we already showed in Lemma 5.1. With the help of this result we can now show the following property of recognizable DAG languages which can be seen as a generalization of the pumping lemma for recognizable tree languages.

Lemma 5.2. *Let $L \in L(ND)$ be a recognizable DAG language. Then there is a constant $k > 0$ 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 $n \geq 1$ we have $G[e \bowtie e']^n \in L^{\&}$.*

Proof. Let $L \in L(ND)$ be a recognizable DAG language and $A = (Q, R, \Sigma)$ be a DAG automaton with $L(A) = L$. Then we choose $k = |Q|$. Now consider a DAG $G \in L$ with $|E_G| > k$ and an accepting run ρ of A on G. Then there are two edges $e, e' \in E_G$ such that $\rho(e) = \rho(e')$ as ρ must use one state at least twice.

Now we show by induction that there is an accepting run ρ_n of A on $G[e \bowtie e']^n$ for all $n \geq 1$ with $\rho_n(e'_n) = \rho(e)$. Recall therefore that we defined $G[e \bowtie e']^1 = G$ and $G[e \bowtie e']^{n+1} = (G[e \bowtie e']^n \& G_{n+1})[e_n \bowtie e'_{n+1}]$ where $G_{n+1} \simeq G$ and e_n and e'_n are copies of e and e' in G_n. This is also illustrated in Figure 5.3.

For $n = 1$ we have $G[e \bowtie e']^1 = G \in L$ and for the accepting run $\rho_1 = \rho$ of A on $G[e \bowtie e']^1$ we have $\rho_1(e_1) = \rho(e')$. Assume now that there is an accepting run ρ_n of A

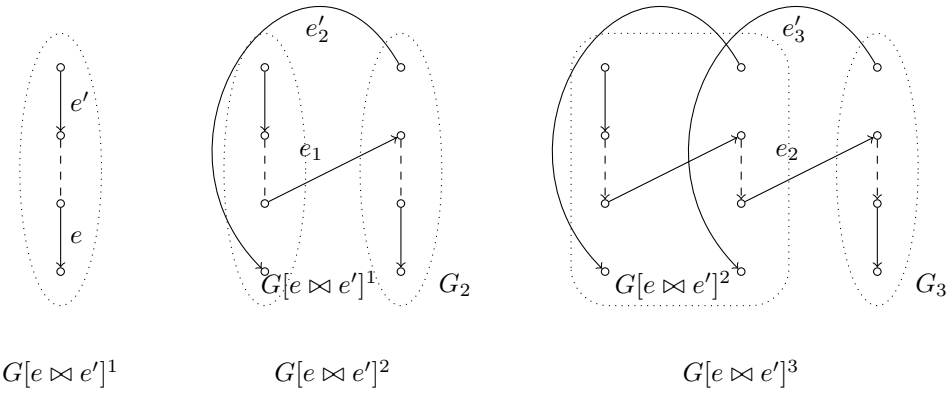


Figure 5.3: An example of the graphs $G[e \bowtie e']^1, G[e \bowtie e']^2$ and $G[e \bowtie e']^3$

on $G[e \bowtie e']^n$ for some n with $\rho_n(e'_n) = \rho(e)$. Let moreover ρ' be an accepting run of A on G_{n+1} with $\rho'(e'_{n+1}) = \rho(e)$. Then the run ρ_{n+1} defined by

$$\rho_{n+1}(e) = \begin{cases} \rho'(e) & \text{if } e \in E_{G_{n+1}} \\ \rho_n(e) & \text{else} \end{cases}$$

is an accepting run of A on $G[e \bowtie e']^{n+1}$ as it assigns the same states to the in- and outgoing edges of every vertex of $G[e \bowtie e']^{n+1}$ as the accepting runs ρ' and ρ_n (see also Lemma 5.1). Moreover we have $\rho_{n+1}(e'_{n+1}) = \rho(e')$. This means that we have $G[e \bowtie e']^n \in L^{\&}$ for all $n \geq 1$. □

Remark 5.3. *In contrast to the pumping lemma for tree or string languages Lemma 5.2 does however not directly imply that every DAG $G \in L$ gets actually pumped up by the swap $G[e \bowtie e']^n$. The reason is that $G[e \bowtie e']^k$ may fall apart into k connected components $C_1, \ldots, C_k$ with $|V_{C_1}| = \cdots = |V_{C_k}| = |V_G|$. As illustrated in Figure 5.4 this can happen if e and e' have different directions which means that there is no path between the end vertex of e and the start vertex of e' that does not contain the edge e or e'. If e and e' point into the same direction, G gets however pumped up because all edges e_i are part of the same connected component C of $G[e \bowtie e']^k$, which means that every $G[e \bowtie e']^k$ contains a connected component C with $|V_C| > k$ as two e_i, e_j with $i \neq j$ share at most one vertex. This is also the case in Figure 5.3.*

If we therefore choose $k = 2|Q|$ there have to be three edges of G such that an accepting run assigns the same state to all of them. As G is connected at least two of these edges must have the same direction. So in this case there is actually an infinite sequence of DAGs (G_i) with $|V_{G_n}| < |V_{G_{n+1}}|$ with $G_n \in L$ for all n.

One feature of Lemma 5.2 is that we require a DAG of a certain size in order to be able to connect it to a disjoint copy. But as DAGs are allowed to contain undirected cycles, it is actually possible to pump up every DAG that contains such a cycle with only one edge swap between the DAG and a copy of it. We formalize this property in the following lemma.

Lemma 5.3. *For every DAG language $L \in L(ND)$ and every DAG $G \in L$ that contains an undirected cycle there is an edge $e \in E_G$ such that $G[e \bowtie e]^n \in L$ for all $n \geq 1$.*

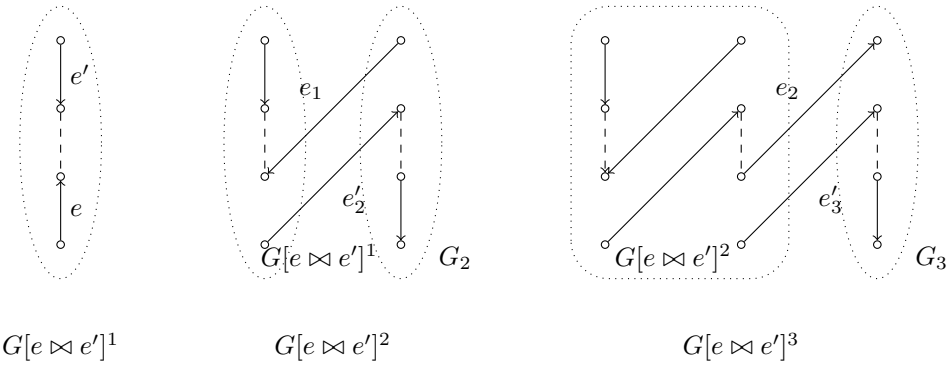


Figure 5.4: A DAG G where the size of the components of $G[e \bowtie e']^k$ does not increase

Proof. Let $L \in L(ND)$ be a recognizable DAG language and $G \in L$ be a DAG containing an undirected cycle C. Then there is a DAG automaton A with $L(A) = L$. Let moreover e be an edge which belongs to the cycle C.

For $n = 1$ we have $G[e \bowtie e]^1 = G \in L$. The DAG $G[e \bowtie e]^1$ contains moreover a cycle C_1 (namely C) to which e belongs and there is an accepting run ρ_1 of A on $G[e \bowtie e]^1$.

Recall that the DAG $G[e \bowtie e]^{n+1}$ is constructed by taking the DAG $G[e \bowtie e]^n$ and a copy G_{n+1} of G and swapping the copies of e in $G[e \bowtie e]^n$ and G_{n+1} which we refer to by e_n and e_{n+1} respectively. This is illustrated in Figure 5.5.

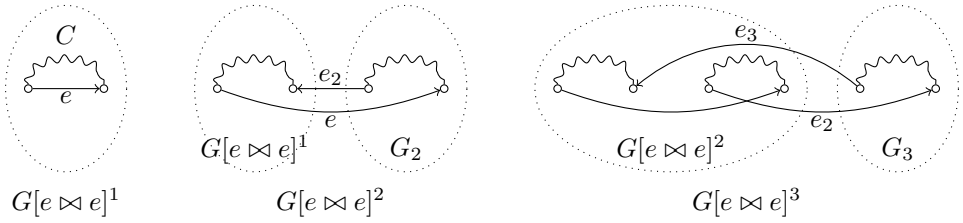


Figure 5.5: The DAGs G_1, G_2 and G_3

Assume now for some n that we have $G[e \bowtie e]^n \in L$, that $G[e \bowtie e]^n$ contains a cycle C_n which contains the edge e_n and that there is an accepting run ρ_n of A on $G[e \bowtie e]^n$ with $\rho_n(e_n) = \rho_1(e_1)$. Let morover ρ' be an accepting run of A on G_{n+1} with $\rho'(e'_{n+1}) = \rho(e)$. Then the run ρ_{n+1} defined by

$$\rho_{n+1}(e) = \begin{cases} \rho'(e) & \text{if } e \in E_{G_{n+1}} \\ \rho_n(e) & \text{else} \end{cases}$$

is an accepting run of A on $G[e \bowtie e]^{n+1}$. The resulting DAG $G[e \bowtie e]^{n+1}$ is still connected as the cycle C_n and the copy of the cycle C in G_{n+1} form now one big cycle C_{n+1}. This cycle C_{n+1} uses moreover the edge e_{n+1} and we have $\rho_{n+1}(e_{n+1}) = \rho_1(e_1)$.

Therefore we have $G[e \bowtie e]^n \in L$ for all $n \geq 1$. □

This lemma implies that there is no finite DAG language containing a DAG with

an undirected cycle that can be recognized by a DAG automaton. There are however recognizable DAG languages that are finite as the following lemma shows.

Lemma 5.4. *For every DAG G that contains no undirected cycle there is a DAG automaton A with $L(A) = \{G\}$.*

Proof. Let G be a DAG that contains no undirected cycle and consider a DAG automaton $A = (Q, \Sigma, R)$. For every vertex v of G with label σ and $in(v) = (e_1, \ldots, e_n)$ and $out(v) = (f_1, \ldots, f_m)$ we have $(p_{e_1}, \ldots, p_{e_n}) \overset{\sigma}{\leftrightarrow} (p_{f_1}, \ldots, p_{f_m}) \in R$. If v is a root or leaf we set the left-hand or right-hand side of the corresponding rule to λ.

We show now that $L(A) = \{G\}$. Every rule $r \in R$ corresponds to exactly one vertex $v_r \in V_G$ and for two rules $r = \alpha \overset{\sigma}{\leftrightarrow} \beta$ and $r' = \alpha' \overset{\sigma'}{\leftrightarrow} \beta'$ we have $\beta \cap \alpha' \neq \emptyset$ if and only if there is an edge from v_r to $v_{r'}$ in G. Consider now a DAG $G' \in L(A)$ and an accepting run ρ of A on G'. We show that the run ρ uses every rule $r \in R$ exactly once. If ρ uses a rule $r_1 \in R$ more than once, there is a sequence of rules $r_1, \ldots, r_k$ such that $r_1 = r_k$ and for $r_i = \alpha_i \overset{\sigma_i}{\leftrightarrow} \beta_i$ we have $(\beta_i \cap \alpha_{i+1}) \cup (\alpha_i \cap \beta_{i+1}) \neq \emptyset$ for $i \in \{1, \ldots, k-1\}$. But this means that there is an undirected cycle $v_{r_1}, \ldots, v_{r_k}$ in G, contradicting Lemma 5.2 which implies that $L(A)$ is not finite. 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 left-hand side and exactly one rule such that q occurs in its right-hand side. Therefore the accepting run ρ must use every rule at least once.

This means that we have $|V_{G'}| = |Q| = |V_G|$ and we can define the following isomorphism (g_V, g_E) from G' to G. Let $v \in V_{G'}$, $in(v) = (e'_1, \ldots, e'_n)$ and $out(v) = (f'_1, \ldots, f'_m)$ and let $r = (p_{e_1}, \ldots, p_{e_n}) \overset{\sigma}{\leftrightarrow} (p_{f_1}, \ldots, p_{f_m}) \in R$ be the rule that ρ uses in v. Then we can choose $g_V(v) = v_r$, $g_E(e'_i) = e_i$ and $g_E(f'_i) = f'_i$. Therefore G' and G are isomorphic and we have $L(A) = \{G\}$. □

A direct consequence of the previous two lemmata and the closedness of $L(ND)$ under union is the following equivalence for finite DAG languages.

Corollary 5.1. *Let L be a finite DAG language. Then we have $L \in L(ND)$ if and only if there is no DAG $G \in L$ that contains an undirected cycle.*

6 Finiteness of DAG Automata

In this chapter we will investigate the finiteness problem for DAG automata. It consists in the question if the language recognized by a given DAG automaton is finite. This is similar to the emptiness problem which asks for a given DAG automaton if it recognizes a nonempty language. Chiang et al. showed that the emptiness problem can be reduced to the problem to decide if a Petri net is structurally cyclic which implies that it can be decided in polynomial time [6].

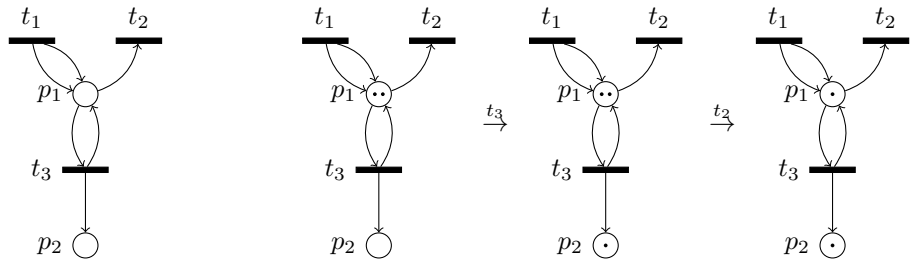
We will prove that the finiteness problem can also be decided in polynomial time but to be able to do so we show at first that it is possible to eliminate all "useless" rules of a given DAG automaton. We call a rule r of a DAG automaton $A = (Q, \Sigma, R)$ *productive* if and only if there is a DAG $G \in L(A)$ such that there is an accepting run of A on G that uses r in some vertex $v \in V_G$. A DAG automaton $A = (Q, \Sigma, R)$ is called *reduced* if and only if every rule $r \in R$ is productive.

Now we follow the idea of Chiang et al. and show that we can compute the productive rules of a given DAG automaton by considering an equivalent problem for Petri nets. A *Petri net* is a *bipartite unlabeled graph* $N = (V, E, in, out)$. *Unlabeled* means simply that N has no labeling function whereas *bipartite* means that we have $V = T \cup P$ for disjoint sets T and P such that there are only edges from vertices $t \in T$ to vertices $p \in P$ and vice versa. A vertex $t \in T$ is also called *transition*, a vertex $p \in P$ is also called *place*. Figure 6.1a shows an example of a Petri net where the transitions are drawn in the usual way as bars whereas the places are drawn as circles.

For a Petri net $N = (T \cup P, E, in, out)$ a *configuration* is a total function $\phi : P \to \mathbb{N}$ that assigns a certain number $\phi(p)$ of tokens to every place $p \in P$. The semantics of a Petri net is now that the different transitions can *fire* which consumes and produces tokens and causes the configuration of the Petri net to change. The effect of a firing of a transition t on a place p is that it consumes k tokens from p and adds l tokens to p where k is the number of edges from p to t and l is the number for edges from t to p. Given a configuration ϕ of a Petri net $N = (T \cup P, E, in, out)$, a firing of a transition t leads formally to the configuration ϕ' defined by

$$\phi'(p) = \phi(p) - |out(p) \cap in(t)| + |out(t) \cap in(p)|.$$

A firing of a transition t is moreover only valid if we have $\phi(p) - |out(p) \cap in(t)| \geq 0$ for all places $p \in P$ as a transition cannot consume more tokens from a place as the place actually holds. If the firing of a transition t leads from a configuration ϕ to a configuration ϕ' we denote this also as $\phi \xrightarrow{t} \phi'$. A *firing sequence* is a sequence of firings $(\phi_1 \xrightarrow{t_1} \phi_2, \phi_2 \xrightarrow{t_2} \phi_3, \ldots, \phi_{k-1} \xrightarrow{t_{k-1}} \phi_k)$ such that every firing is valid. We denote such a firing sequence also as $\phi_1 \xrightarrow{t_1} \phi_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{k-1}} \phi_k$. Figure 6.1b shows an example of a firing sequence $\phi_1 \xrightarrow{t_3} \phi_2 \xrightarrow{t_2} \phi_3$ of the Petri net N from Figure 6.1a where 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.



(a) A Petri net N

(b) A firing sequence $\phi_1 \xrightarrow{t_3} \phi_2 \xrightarrow{t_2} \phi_3$ of N

Figure 6.1: A Petri net and a firing sequence

Being familiar with Petri nets we can now show how the productive rules of a given DAG automaton can be computed.

Lemma 6.1. *Let A be a DAG. Then we can construct a reduced DAG automaton A_{red} with $L(A) = L(A_{red})$ in polynomial time.*

Proof. Let $A = (Q, \Sigma, R)$ be a DAG automaton and consider the Petri net $N_A = (T \cup P, E, in, out)$ where $T = R$, $P = Q$ and for every rule $r = (p_1, \ldots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \ldots, q_m)$ there is an edge from every p_i to r for $i \in \{1, \ldots, n\}$ and an edge from r to every q_i for $i \in \{1, \ldots, m\}$. Figure 6.2 shows an example of a simple DAG automaton A and the corresponding Petri net N_A.

$$
\begin{aligned}
A &= (Q, \Sigma, R) \text{ with} \\
R &= \{r_1, r_2, r_3, r_4\} \text{ for} \\
r_1 &= \lambda \overset{s}{\leftrightarrow} (p, q) \\
r_2 &= (p) \overset{a}{\leftrightarrow} (q, q) \\
r_3 &= (q) \overset{b}{\leftrightarrow} (p, p) \\
r_4 &= (q, p) \overset{t}{\leftrightarrow} \lambda
\end{aligned}
$$

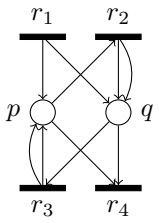


Figure 6.2: A DAG automaton A and the corresponding Petri net N_A

The idea behind this construction is that every rule $(p_1, \ldots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \ldots, q_m)$ can be seen as a transition that consumes states $p_1, \ldots, p_n$ and produces states $q_1, \ldots, q_m$. Every accepting run ρ of A on an acyclic labeled graph G that uses the rules $r_1, \ldots, r_k$ in a top-down fashion can now be seen as a firing sequence $\phi_1 \xrightarrow{r_1} \ldots \xrightarrow{r_k} \phi_{k+1}$ of the corresponding transitions. In the initial configuration there are no states at all, therefore we have $\phi_1 = \mathbf{0}$ where $\mathbf{0}$ is the *null configuration* defined as $\mathbf{0}(p) = 0$ for all $p \in P$. The run ρ then uses some rules with an empty left-hand side in the root vertices of G and the corresponding transitions just produce tokens. In the interior vertices of G tokens are then both produced and consumed until the rules used in the leaf vertices just consume all remaining tokens. If we consider some edge $e \in G$, the rule used in the start vertex of e adds one token to $\rho(e)$ whereas the rule used in the end vertex consumes one token from $\rho(e)$. The final configuration therefore needs also to be $\phi_{k+1} = \mathbf{0}$. This means

that $\mathbf{0} \xrightarrow{r_1} \ldots \xrightarrow{r_k} \mathbf{0}$ is a valid nonempty firing sequence of N_A if there is an accepting run ρ of A on some acyclic labeled graph G that uses the rules $r_1, \ldots, r_k$ in a top-down fashion. Figure 6.3 shows an example of an accepting run ρ of the DAG automaton A from Figure 6.2 on a DAG G and a corresponding valid firing sequence of the Petri net N_A. We can see that the run ρ uses the rule r_1 in the root vertex. Starting from the null configuration the transition r_1 therefore fires and adds one token to p and q each. The rule r_2 was used next in the vertex a, so r_2 fires, consumes on token from p and adds two tokens to q. Then r_3 fires consuming one token from q and adding two tokens to p before r_4 fires twice and consumes all the tokens from p and q.

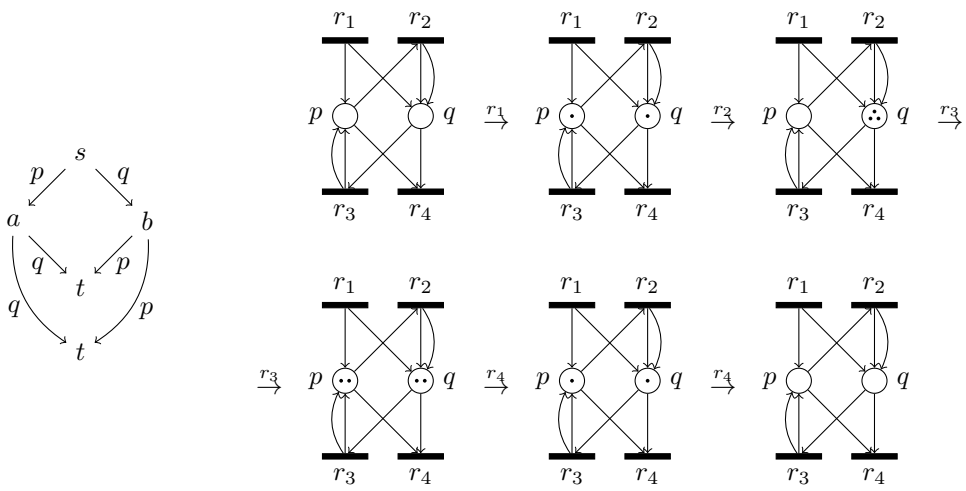


Figure 6.3: An accepting run and a corresponding firing sequence

Now we consider the opposite direction and show that there is also an accepting run ρ of A on some acyclic labeled graph G that uses the rules $r_1, \ldots, r_k$ if $\mathbf{0} \xrightarrow{r_1} \ldots \xrightarrow{r_k} \mathbf{0}$ is a valid firing sequence of N_A. Let therefore $\mathbf{0} \xrightarrow{r_1} \ldots \xrightarrow{r_k} \mathbf{0}$ be a valid nonempty firing sequence of N_A. Then we can construct the following acyclic labeled graph G and a corresponding run ρ of A on G. We start our construction with the empty DAG. For the first rule $r_1 = \lambda \xrightarrow{\sigma_1} (q_1, \ldots, q_m)$ we insert a vertex v_1 with label σ_1 and m outgoing edges $e_1, \ldots, e_m$. Moreover we set $\rho(e_i) = q_i$ for $i \in \{1, \ldots, m\}$. The graph G is for the moment incomplete as the edges $e_1, \ldots, e_m$ lack their end vertices, but this is fixed in the later steps as follows. For $i \in \{2, \ldots, k\}$ and every rule $r_i = (p_1, \ldots, p_n) \xrightarrow{\sigma_i} (q_1, \ldots, q_m)$ we insert a new vertex v_i with label σ_i and connect it to n dangling edges $e_1, \ldots, e_n$ which are marked with the states $p_1, \ldots, p_n$. Moreover we insert m outgoing edges of v_i and mark them in ρ with states $q_1, \ldots, q_m$. This means that the insertion of every vertex v_i consumes n dangling edges that have no end vertex and creates m new dangling edges. After the insertion of v_i there are therefore $\phi_{i+1}(p)$ dangling edges marked with state p. As $\phi_{k+1} = \mathbf{0}$ the graph G therefore contains no dangling edges and from our construction it follows that ρ is an accepting run of A on G. An example for the construction of a graph $G \in L(A)^{\&}$ from a valid firing sequence $\mathbf{0} \xrightarrow{r_1} \ldots \xrightarrow{r_k} \mathbf{0}$ of N_A is illustrated in Figure 6.4.

This means that a rule r of a DAG automaton A is productive if and only if there is a valid firing sequence $\mathbf{0} \xrightarrow{r_1} \ldots \xrightarrow{r_k} \mathbf{0}$ of N_A with $r = r_i$ for some i. Therefore the DAG

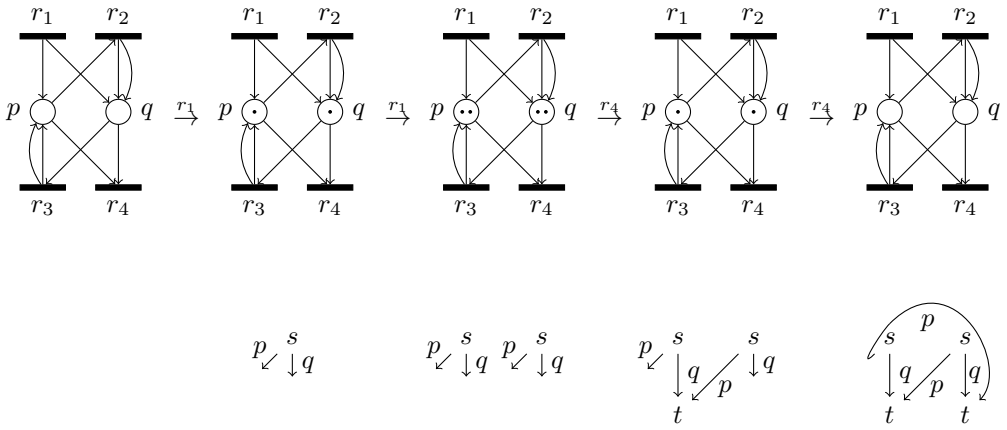


Figure 6.4: Construction of a graph G from a firing sequence $\mathbf{0} \xrightarrow{r_1} \ldots \xrightarrow{r_k} \mathbf{0}$ of N_A

automaton $A_{red} = (Q, \Sigma, \Lambda(N_A))$ where

$$\Lambda(N_A) = \{t \mid t = t_i \text{ for some valid firing sequence } \mathbf{0} \xrightarrow{t_1} \ldots \xrightarrow{t_k} \mathbf{0} \text{ of } N_A\}$$

is reduced and we have $L(A) = L(A_{red})$. In the proof of the decidability of structurally cyclic Petri nets Drewes and Leroux showed that the set $\Lambda(N_A)$ is computable in polynomial time [8]. Given a DAG automaton A we can therefore construct a reduced DAG automaton A_{red} with $L(A) = L(A_{red})$ in polynomial time. □

Using this result we can now show that the finiteness problem is decidable in polynomial time.

Theorem 6.1. *For a given DAG automaton A it is decidable in polynomial time if $L(A)$ is finite.*

Proof. Let $A = (Q, \Sigma, R)$ be a DAG automaton. We define a *cycle* in A to be a sequence of rules $\alpha_0 \overset{\sigma_0}{\leftrightarrow} \beta_0, \ldots, \alpha_k \overset{\sigma_k}{\leftrightarrow} \beta_k$ such that $\alpha_0 \overset{\sigma_0}{\leftrightarrow} \beta_0 = \alpha_k \overset{\sigma_k}{\leftrightarrow} \beta_k$ and such that there exists a sequence of states $p_0, \ldots, p_{k-1}$ that satisfies the two following conditions. For $i \in \{0, \ldots, k-1\}$ we have $p_i \in (\beta_i \cap \alpha_{i+1}) \cup (\alpha_i \cap \beta_{i+1})$ and if we have $p_i = p_{i+1 \mod k}$ then the state p_i occurs at least twice in the rule r_{i+1}. A DAG automaton A that contains a cycle is also called *cyclic*.

The intuition is that a cycle is a sequence $r_0, \ldots, r_k$ of rules such that a rule r_i overlaps with the preceding and the succeeding rule in two states $p_{i-1 \mod k}$ and p_i respectively. For such a cycle $r_0, \ldots, r_k$ we will construct a DAG $G \in L(A)$ that contains an undirected path $(v_0, e_0, v_1, \ldots, e_{k-1}, v_k)$ such that there is an accepting run of A on G that uses the rule r_i in the vertex v_i and assigns the state p_i to the edge e_i. Therefore the states $p_{i-1 \mod k}$ and p_i need to be either distinct or occur at least twice in r_i, otherwise we cannot use r_i in v_i.

Example 6.1: To illustrate this consider the DAG $A = (Q, \Sigma, R)$ with $R = \{r_1, \ldots, r_5\}$ where $r_1 = \lambda \overset{s}{\leftrightarrow} (p, p), r_2 = (p) \overset{a}{\leftrightarrow} (q, q), r_3 = (q) \overset{s}{\leftrightarrow} (p), r_4 = (p, p) \overset{s}{\leftrightarrow} \lambda$ and $r_5 = (q, q) \overset{s}{\leftrightarrow} \lambda$. For this DAG automaton A the sequence r_1, r_2, r_3, r_1 is a cycle as p, q, p is a valid sequence of overlapping states that satisfies the required conditions. The sequence

r_3, r_4, r_4 is however no cycle because the state p (which is the only state that r_3 and r_4 have in common) occurs only once in r_3.

Let now $A = (Q, \Sigma, R)$ be a reduced DAG automaton. We show that $L(A)$ is finite if and only if A is not cyclic. For the if part assume that A is not cyclic, let $G \in L(A)$ be a DAG and ρ be an accepting run of A on G. Then ρ can use every rule $r \in R$ at most once. To show this assume that ρ uses the same rule $r \in R$ in two distinct vertices v_0 and v_k of G. As G is connected there is now an undirected path $(v_0, e_0, v_1, \ldots, e_{k-1}, v_k)$ between v_0 and v_k. The rules $r_0, \ldots, r_k$ used by ρ in the vertices $v_0, \ldots, v_k$ along this path overlap in the states $\rho(e_0), \ldots, \rho(e_{k-1})$ and yield a cycle in A. Therefore we have $|V_G| \leq |R|$ which means that $L(A)$ is finite, as over a fixed alphabet Σ with bounded head and tail ranks there is only a finite number of DAGs with at most $|R|$ vertices.

For the only if part assume now that A is cyclic, let $r_0, \ldots, r_k$ be a cycle in A and $p_0, \ldots, p_{k-1}$ a corresponding sequence of states. As A is reduced there is a DAG $G_i \in L(A)$ for every $i \in \{0, \ldots, k\}$ such that there exists an accepting run ρ_i of A on G_i which uses the rule r_i in a vertex $v_i \in V_{G_i}$. As $r_0, \ldots, r_k$ is a cycle every such vertex v_i is moreover incident to two distinct edges e_i and e'_i with $e_i \in in(v_i)$ iff $p_{i-1 \mod k} \in \alpha_i$ and $e'_i \in in(v_i)$ iff $p_i \in \alpha_i$ such that $\rho_i(e_i) = p_i = \rho_{i+1}(e'_{i+1})$ for $i \in \{0, \ldots, k-1\}$. Under the assumption that the different DAGs G_i are pairwise disjoint we can now connect all these DAGs by swapping the edges e_i and e'_{i+1} for all $i \in \{0, \ldots, k-1\}$. An example for this is also illustrated in Figure 6.5. It shows the construction of a graph $(G_0 \& \ldots \& G_3)[e_0 \bowtie e'_1] \ldots [e_2 \bowtie e'_3]$ for the cycle r_1, r_2, r_3, r_1 in the DAG automaton A from Example 6.1.

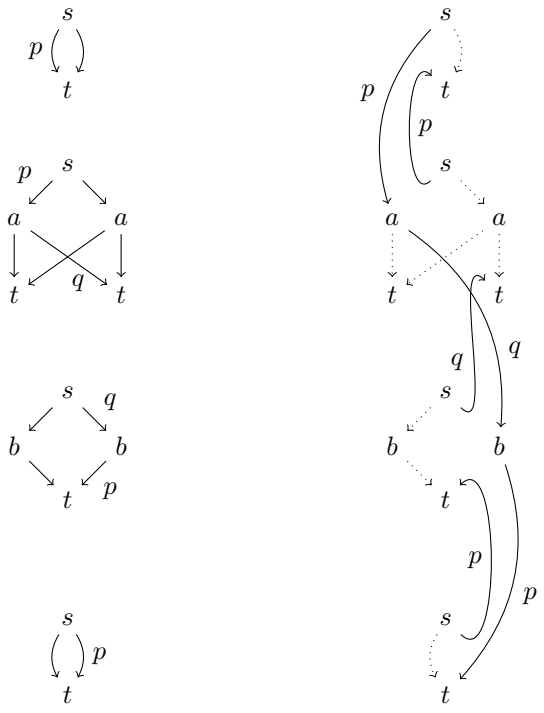


Figure 6.5: Construction of the graph G from DAGs G_0, G_1 and G_2

The resulting labeled graph $G = (G_0 \& \dots \& G_k)[e_0 \bowtie e'_1] \dots [e_{k-1} \bowtie e'_k]$ is then still accepted by A according to Lemma 5.1 and an accepting run ρ can simply be constructed by combining the different ρ_i, i.e. by choosing

$$\rho(e) = \rho_i(e) \text{ if } e \in E_{G_i}$$

This means that ρ uses the same rule in v_0 and v_k as we have $r_0 = r_k$. Therefore there are two edges e and e' incident to v_0 and v_k with $\rho(e) = \rho(e')$. The idea is now to pump G up by connecting it iteratively to an isomorphic copy of itself with a swap of e' and a copy of e which leads to a sequence of graphs $G[e' \bowtie e]^k$.

The crucial part is however to show that there is actually also a sequence of connected DAGs of increasing size as G and $G[e' \bowtie e]^k$ are in general not connected. Observe therefore that it follows from the definition of a cycle and the choice of e_i and e'_i that e_i is an outgoing edge of v_i in G_i if and only if e'_{i+1} is an ingoing edge of v_{i+1} in G_{i+1}. The swap of e_i and e'_{i+1} has hence the effect that there is an edge from v_i to v_{i+1} in G which means that there is a connected component C of G that contains a path ω from v_1 to v_k. Then we choose e as the edge between v_0 and v_1 and e' as an corresponding edge incident to v_k such that $\rho(e) = \rho(e')$ and $e \in in(v_0)$ iff $e' \in in(v_k)$. If we connect the DAG C now to a disjoint copy of itself by swapping e and the copy of e', the resulting graph $C[e \bowtie e']^2$ contains a path from v_k to the copy of v_k. If we connect $C[e \bowtie e']^2$ to another copy of C we create a path from the copy of v_k to a second copy of v_k. This is also illustrated in Figure 6.6.

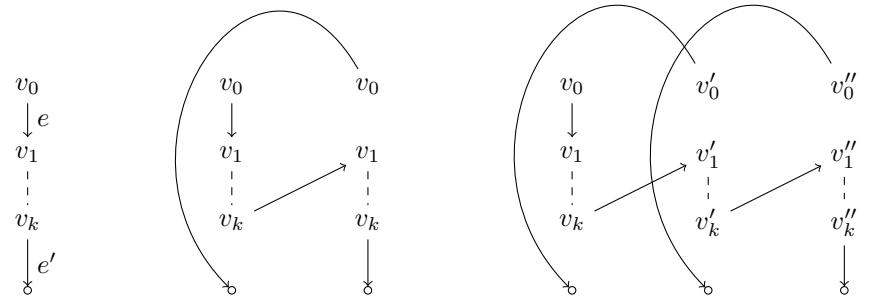


Figure 6.6: The graphs $C[e' \bowtie e]^1, C[e' \bowtie e]^2$ and $C[e' \bowtie e]^3$

Every graph $C[e \bowtie e']^k$ contains therefore a connected component C_k with $|V_{C_k}| \geq k$ and $|E_{C_k}| \geq (k-1)$. As we have $C \in L(A)$ it follows from Lemma 5.1 that $C[e \bowtie e']^k \in L(A)^{\&}$ and as C_k is connected we have $C_k \in L(A)$ for every $k \geq 1$. This means that $L(A)$ is not finite.

The language recognized by a reduced DAG automaton A is therefore finite if and only if A is not cyclic. Given an arbitrary DAG automaton A we can hence decide if $L(A)$ is finite by constructing an equivalent reduced DAG automaton A_{red} with $L(A) = L(A_{red})$ 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. The finiteness problem for DAG automata is therefore decidable in polynomial time. □

7 Connection to Tree Automata

In the previous chapters trees have already been mentioned several times as DAGs are a generalization of trees and they have been investigated quite extensively. In the following we perform a basic comparison of DAGs and trees and the corresponding automata and draw some further connections between them.

A *tree* can formally be defined as a DAG over a doubly ranked alphabet Σ where every symbol $\sigma \in \Sigma$ has head rank 1 or 0. We call such a doubly ranked alphabet also a *tree alphabet*. From this definition it follows that every tree T has exactly one root $v_r \in V_T$ and that every vertex $v \neq v_r$ has exactly one ingoing edge. A *tree automaton* is a DAG automaton over a tree alphabet. The semantics of tree and DAG automata is identical and every tree automaton recognizes therefore a set of trees.

Remark 7.1. *This definition differs from the traditional tree automata model where the vertices of an input tree are marked with states, not the edges. The states are moreover usually partitioned into accepting and non-accepting states where the semantics is that an accepting run needs to assign an accepting state to the root. For the sake of simplicity we reuse however our definitions from Chapter 2 as both approaches are fully equivalent. Without going into details it is actually fairly easy to transfer one automata model into the other. The idea is that one can simply identify the state of a vertex with the state of its ingoing edge and vice versa because every vertex has at most one ingoing edge. An example is illustrated in Figure 7.1 where the rules of a traditional top-down tree automaton with accepting state q_a and a corresponding automaton in DAG notation are shown. Both automata recognize the language of all binary trees where the leftmost leaf is labeled with an a whereas the remaining leaves have the label b. For a detailed description of the semantics of the traditional tree automata model we refer to Comon et al. [7].*

(a) Traditional notation	(b) DAG notation
$q_a(s(x,y)) \to q_a(x)q_b(y)$	$\lambda \overset{s}{\leftrightarrow} (q_a, q_b)$
$q_a(f(x,y)) \to q_a(x)q_b(y)$	$(q_a) \overset{f}{\leftrightarrow} (q_a, q_b)$
$q_b(f(x,y)) \to q_b(x)q_b(y)$	$(q_b) \overset{f}{\leftrightarrow} (q_b, q_a)$
$q_a(a) \to a$	$(q_a) \overset{a}{\leftrightarrow} \lambda$
$q_b(b) \to b$	$(q_b) \overset{b}{\leftrightarrow} \lambda$

Figure 7.1: Two notations of a tree automaton

It is indeed even possible to use a vertex-marking approach for DAG automata where the automata assign states to the vertices of a DAG and not to the edges [14]. For DAGs the presented edge-marking model is in most cases however more convenient for which reason it is also more popular.

As previously indicated, DAG automata can be interpreted as an extension of the tree automata model where the restriction that every vertex needs to have at most one parent is removed. Tree automata are themselves in turn an extension of finite-state automata that work on strings which can be seen as DAGs where each vertex has at most one predecessor and at most one successor. From the previous definition it follows immediately that the class of recognizable tree languages is a subset of the recognizable DAG languages. Many results for tree languages hold moreover also directly for the DAG case or can be generalized as we saw for the pumping lemma. There are however some remarkable differences to DAG languages. The class of recognizable tree languages is for example closed under complement [7] which does not hold for the class of recognizable DAG languages (see Theorem 5.2). For trees there is moreover no difference between determinism and nondeterminism in the bottom-up case in contrast to the top-down case [7].

We show now how a given acyclic labeled graph G can be transformed into a tree that has a structure similar to the one of G. This is done by an unfolding of G such that for every vertex $v \in V_G$ with n ingoing and m outgoing edges the resulting tree will contain n copies of v with one ingoing edge and m outgoing edges to copies of the children of v. In case the graph G has k roots this procedure yields a disconnected labeled graph consisting of k trees and we refer to each tree with the corresponding root of G. Formally we define this as follows.

Definition 7.1. 1. Let Σ be a doubly ranked alphabet. The tree alphabet $\mathit{Tree}(\Sigma)$ is then defined as $\mathit{Tree}(\Sigma) = \{\sigma_t \mid \sigma \in \Sigma\}$ with $\mathit{head}(\sigma_t) = \min(\mathit{head}(\sigma), 1)$ and $\mathit{tail}(\sigma_t) = \mathit{tail}(\sigma)$.

2. Let G be an acyclic labeled graph and r be a root of G. The *tree unfolding* of G with respect to r is then $\mathit{Tree}_r(G) = (V, E, \mathit{label}, \mathit{in}, \mathit{out})$ where

$$
\begin{aligned}
V &= \{v_p \mid v \in V_G \text{ and } p \text{ is a directed path from } r \text{ to } v\} \\
E &= \{e_p \mid p \text{ is a directed path in } G \text{ from } r \text{ to a vertex } v \neq r\} \\
\mathit{label}(v_p) &= \mathit{label}_G(v)_t \\
\mathit{in}(v_p) &= \begin{cases} (e_p) & \text{if } p \text{ is nonempty} \\ \emptyset & \text{else} \end{cases} \\
\mathit{out}(v_p) &= (e_{p \circ e_1}, \ldots, e_{p \circ e_m}) \text{ for } \mathit{out}_G(v) = (e_1, \ldots, e_m)
\end{aligned}
$$

   where $p \circ e$ is the path obtained by appending the edge e to the path p.

3. Let L be a DAG language. The *tree unfolding* of L is then

$$\mathit{Tree}(L) = \{\, \mathit{Tree}_r(G) \mid G \in L \text{ and } r \text{ is a root of } G\}.$$

An example of the tree unfolding of a DAG G is shown in Figure 7.2.

For simplicity we confuse in the following the symbol σ and its tree counterpart σ_t when the context is clear. Now we show that the tree unfolding preserves recognizability, i.e. if L is a recognizable DAG language then $\mathit{Tree}(L)$ is a recognizable tree language.

Theorem 7.1. *Let $L \in L(ND)$ be a recognizable DAG language. Then $\mathit{Tree}(L)$ is a recognizable tree language.*

Proof. Let $L \in L(ND)$ be a recognizable DAG language and $A = (Q, \Sigma, R)$ be a reduced DAG that recognizes L. Then we define the tree automaton $A_t = (Q, \mathit{Tree}(\Sigma), R_t)$ as

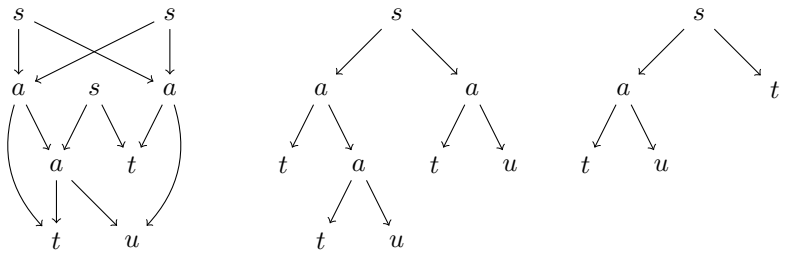


Figure 7.2: A DAG G and the tree unfolding $Tree_r(G)$ for two different roots r

follows. For every rule $(p_1, \dots, p_n) \overset{\sigma}{\leftrightarrow} (q_1, \dots, q_m) \in R$ and all $i \in \{1, \dots, n\}$ we have $(p_i) \overset{\sigma}{\leftrightarrow} (q_1, \dots, q_m) \in R_t$. An example for this is shown in Figure 7.3.

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

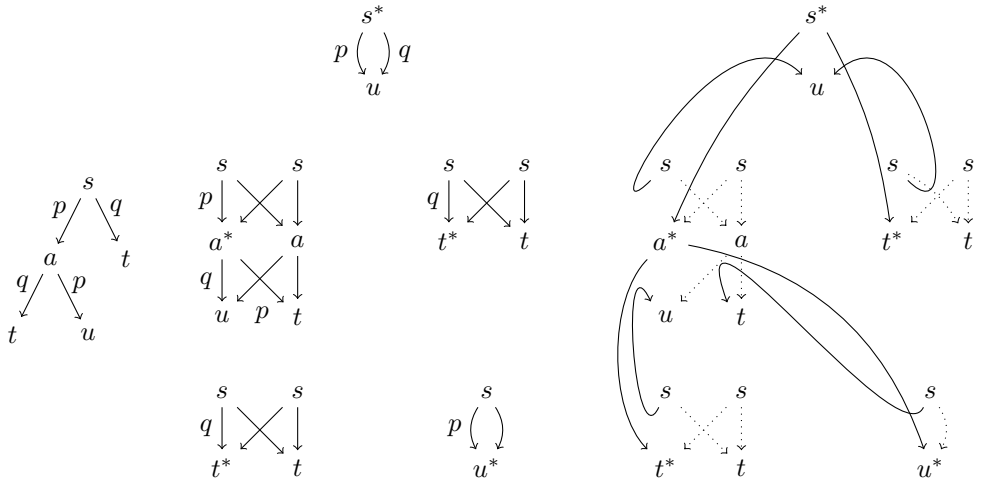
$$\begin{aligned} R = \{&\lambda \overset{s}{\leftrightarrow} (p, q), \lambda \overset{s}{\leftrightarrow} (q, p), \\ &(p, q) \overset{a}{\leftrightarrow} (q, p), \\ &(q, p) \overset{a}{\leftrightarrow} (p, q), \\ &(p, p) \overset{t}{\leftrightarrow} \lambda, (q, q) \overset{t}{\leftrightarrow} \lambda, \\ &(p, q) \overset{u}{\leftrightarrow} \lambda\} \end{aligned} \qquad \begin{aligned} R_t = \{&\lambda \overset{s}{\leftrightarrow} (p, q), \lambda \overset{s}{\leftrightarrow} (q, p), \\ &(p) \overset{a}{\leftrightarrow} (q, p), (q) \overset{a}{\leftrightarrow} (q, p), \\ &(q) \overset{a}{\leftrightarrow} (p, q), (p) \overset{a}{\leftrightarrow} (p, q), \\ &(p) \overset{t}{\leftrightarrow} \lambda, (q) \overset{t}{\leftrightarrow} \lambda, \\ &(p) \overset{u}{\leftrightarrow} \lambda, (q) \overset{u}{\leftrightarrow} \lambda\} \end{aligned}$$

Figure 7.3: A DAG automaton A and the corresponding tree automaton A_t

Then we have $L(A_t) = Tree(L)$. We start by proving that $L(A_t) \supseteq Tree(L)$. Let therefore be $G \in L$ a DAG, r be a root of G and ρ be an accepting run of A on G. Then we can construct an accepting run ρ_t of A_t on $T = Tree_r(G)$ as follows. For every edge $e_p \in E_T$ with $p = p' \circ \hat{e}$ 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 an accepting run consider a vertex $v_p \in V_T$ with $label(v_p) = \sigma$ and the corresponding vertex $v \in V_G$. Let $out_G(v) = (e_1, \dots, e_m)$. Then we have $out_T(v_p) = (e_{p \circ e_1}, \dots, e_{p \circ e_m})$. If $in_T(v_p) = \emptyset$ we also have $in_G(v) = \emptyset$ and there is a rule $\lambda \overset{\sigma}{\leftrightarrow} (\rho(e_1), \dots, \rho(e_m)) \in R$. By definition of R_t and the choice of ρ_t there is then a rule $\lambda \overset{\sigma}{\leftrightarrow} (\rho_t(e_{p \circ e_1}), \dots, \rho_t(e_{p \circ e_m})) \in R_t$. 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 an accepting run of A on G there is now a rule $(\dots, \rho(\hat{e}), \dots) \overset{\sigma}{\leftrightarrow} (\rho(e_1), \dots, \rho(e_m))$. Then there is also a rule $(\rho_t(e_p)) \overset{\sigma}{\leftrightarrow} (\rho_t(e_{p \circ e_1}), \dots, \rho_t(e_{p \circ e_m})) \in R_t$. Therefore ρ_t is an accepting run of A_t on T.

It remains to show that $L(A_t) \subseteq Tree(L)$. Let therefore $T \in L(A_t)$ be a tree and ρ_t be an accepting run of A_t on T. Then we will construct a DAG $C \in L$ such that $T = Tree_r(C)$ for a root r of C. For this, consider a vertex $v \in V_T$ and the rule $(p) \overset{\sigma}{\leftrightarrow} (q_1, \dots, q_m) \in R_t$ that ρ_t uses in v. Then there is a corresponding rule $r_v = (\dots, p, \dots) \overset{\sigma}{\leftrightarrow} (q_1, \dots, q_m) \in R$ and as A is reduced there is a DAG $G_v \in L$ such that there is an accepting run ρ_v of A on G_v which uses the rule r_v in some vertex $v^* \in V_{G_v}$. For $V_T = \{v_1, \dots, v_n\}$ we now construct an acyclic labeled graph G by taking the DAGs $G_{v_1}, \dots, G_{v_n}$ and connecting them through swaps of some edges incident to the vertices $v_1^*, \dots, v_n^*$. These edge swaps are done as follows. Let $v \in V_T$ be a vertex, e be the

i-th outgoing edge of v and e^* be the i-th outgoing edge of the vertex v^*. Then we have $\rho_t(e) = \rho_v(e^*)$ as the right-hand sides of the rules used by ρ_t and ρ_v in v and v^* respectively are identical. If the end vertex of the edge e is moreover a vertex u there is an edge $f^* \in in_{G_u}(u^*)$ with $\rho_t(e) = \rho_u(f^*)$ as ρ_t uses a rule with the left-hand side $(\rho_t(e))$ in u and $\rho_t(e)$ occurs therefore also in the left-hand side of the rule used by ρ_u in u^*. Then we can connect G_v and G_u by swapping the edges e^* and f^* as there are accepting runs on G_v and G_u that assign the same states to them. If we perform this swap for all non-leaf vertices $v_1, \ldots, v_k$ of T we obtain an acyclic labeled graph $G = (G_{v_1} \& \ldots \& G_{v_n})[e_1^* \bowtie f_1^*] \ldots [e_k^* \bowtie f_k^*]$ and according to Lemma 5.1 we have $G \in L(A)^{\&}$. An example for this is also illustrated in Figure 7.4. It shows the construction of a graph $G \in L(A)^{\&}$ from a tree $T \in L(A_t)$ with $Tree_r(G) = T$ for a root r and the automata A and A_t from Figure 7.3. Note that the asterisks added to certain vertex labels indicate just the vertices v^* but are not part of the label.



(a) A tree T (b) The DAGs $G_{v_1}, \ldots, G_{v_n}$ (c) The graph G

Figure 7.4: Construction of a DAG G from a tree T with $T = Tree_r(G)$ for some root r

Consider now a vertex $v \in V_T$ and a child v_i of v. If v_i is the i-th child of v in T, the i-th child of v^* in G is v_i^*. Moreover by construction we have $label_T(v) = label_G(v^*)$ and $label_T(v_i) = label_G(v_i^*)$. Let now r be the root of T and let C be the connected component of G that contains the vertex r^*. Then we have $C \in L(A)$ and $T = Tree_{r^*}(C)$ which means that $L(A_t) \subseteq Tree(L)$. Therefore we have $L(A_t) = Tree(L)$. □

This result allows us to draw some conclusions about the path languages of recognizable DAG languages. The *path language* of a DAG G consists of all sequences of symbols that occur on a root-to-leaf path in G and is defined formally as

$$path(G) = \{label(v_1) \cdots label(v_n) \mid (v_1, e_1, v_2, \ldots, e_{n-1}, v_n) \text{ is a directed path from a root } v_1 \text{ to a leaf } v_n \text{ of } G$$

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

$$path(L) = \bigcup_{G \in L} path(G)$$

It is well known that the path language of a recognizable tree language is a regular string language and the following lemma shows that the same holds for recognizable DAG languages.

Lemma 7.1. *For every DAG language $L \in L(ND)$ the language $path(L)$ is a regular string language.*

Proof. Let $L \in L(ND)$ be a recognizable DAG language and let $G \in L$ be a DAG. Then we can observe that $\sigma_1 \cdots \sigma_n \in path(G)$ if and only if $\sigma_1 \cdots \sigma_n \in path(Tree_r(G))$ for some root r of G. This follows from the definition of the tree unfolding as for a vertex $v \in V_G$ and a copy v' of v in a tree unfolding of G the children of v and v' have the same labels (disregarding the modified head ranks). As $path(T)$ is a regular string language for every tree T that can be recognized by a tree automaton it follows now that $path(G)$ is a regular string language. □

8 Conclusion and Future Work

In this thesis we established some basic properties of DAG languages and DAG automata. Some of them are generalizations of known results for tree languages and tree automata, others require however vertices with multiple ingoing edges and can therefore not be applied to trees. We showed that DAG languages are closed under union, intersection and concatenation, but not under complement. The latter is shown by a swap of two equally marked edges. This is a very utile technique that exploits an elementary property of the discussed DAGs which can also be used to pump a language up. We presented moreover a characterization of finite DAG languages and showed that the finiteness problem is decidable in polynomial time. Finally we proved that the path language of a recognizable DAG language is regular.

The following questions remain open and could be of interest for future research:

- **Equivalence** Is the equivalence problem, which asks if two DAG automata accept the same language, decidable? If yes, can it be decided in polynomial time?
- **Universality** Is the universality problem, which asks if a given DAG automaton accepts all DAGs over its alphabet, decidable? If yes, can it be decided in polynomial time?
- **Properties of Deterministic Languages** What are the properties of deterministic DAG languages?
- **Polynomial Membership** For which DAG languages is the membership problem, which asks if a DAG language contains a given DAG, decidable in polynomial time?

A generalization of the Myhill-Nerode Theorem to DAG languages would result in decidability of the equivalence problem for deterministic DAG automata. It would moreover solve the universality problem for deterministic DAG automata as it is fairly easy to construct an universal top-down deterministic DAG automaton that accepts all DAGs over its alphabet. The idea is to consider an automaton with only one state p and to have a rule $(p, \dots, p) \overset{\sigma}{\leftrightarrow} (p, \dots, p)$ for every symbol σ where the occurrences of p on the left- and right-hand side correspond to the head and tail rank of σ.

Chiang et al. showed that the membership problem is in general NP-complete for recognizable DAG languages [6]. For deterministic DAG languages it is however decidable in polynomial time as deterministic DAG automata allow parsing in polynomial time. The main question for the polynomial membership problem is therefore if there are weaker sufficient properties than determinism.

References

[1] Siva Anantharaman, Paliath Narendran, and Michaël Rusinowitch. Closure properties and decision problems of dag automata. *Inf. Process. Lett.*, 94(5):231–240, 2005.

[2] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract meaning representation for sembanking. In *Proc. 7th Linguistic Annotation Workshop, ACL 2013 Workshop*, 2013.

[3] Francis Bossut, Max Dauchet, and Bruno Warin. Automata and rational expressions on planar graphs. In *Mathematical Foundations of Computer Science 1988, MFCS'88, Carlsbad, Czechoslovakia, August 29 - September 2, 1988, Proceedings*, pages 190–200, 1988.

[4] Francis Bossut, Max Dauchet, and Bruno Warin. A kleene theorem for a class of planar acyclic graphs. volume 117, pages 251–265, 1995.

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

[6] David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. Practical algorithms for dag automata. Unpublished manuscript, 2015.

[7] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2008. release November, 18th 2008.

[8] Frank Drewes and Jérôme Leroux. Structurally cyclic petri nets. Unpublished manuscript, 2015.

[9] Joost Engelfriet and Jan Joris Vereijken. Context-free graph grammars and concatenation of graphs. *Acta Informatica*, pages 34–773, 1995.

[10] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadéniai Kiadó, Budapest, Hungary, 1984.

[11] Ferenc Gécseg and Magnus Steinby. Tree automata. In Grzegorz Rozenberg and Arto Salomaa, editors, *Handbook of formal languages. Vol. 3, Beyond words*, chapter 1, pages 1–68. Springer, 1997.

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

[13] Tsutomu Kamimura and Giora Slutzki. Transductions of dags and trees. *Mathematical Systems Theory*, 15(3):225–249, 1982.

[14] Andreas Potthoff, Sebastian Seibert, and Wolfgang Thomas. Nondeterminism versus determinism of finite automata over directed acyclic graphs. *Bulletin of the Belgian Mathematical Society Simon Stevin*, 1(2):285, 1994.

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