The Problem with Probabilistic DAG Automata for Semantic Graphs

Ieva Vasiljeva[*] and **Sorcha Gilroy**[*] and **Adam Lopez**
Institute for Language, Cognition, and Computation
School of Informatics
University of Edinburgh
{vasiljeva.ieva, gilroysorcha}@gmail.com, alopez@inf.ed.ac.uk

Abstract

Semantic representations in the form of directed acyclic graphs (DAGs) have been introduced in recent years, and to model them, we need probabilistic models of DAGs. One model that has attracted some attention is the DAG automaton, but it has not been studied as a probabilistic model. We show that some DAG automata cannot be made into useful probabilistic models by the nearly universal strategy of assigning weights to transitions. The problem affects single-rooted, multi-rooted, and unbounded-degree variants of DAG automata, and appears to be pervasive. It does not affect planar variants, but these are problematic for other reasons.

1 Introduction

Abstract Meaning Representation (AMR; Banarescu et al. 2013) has prompted a flurry of interest in probabilistic models for semantic parsing. AMR annotations are directed acyclic graphs (DAGs), but most probabilistic models view them as strings (e.g. van Noord and Bos, 2017) or trees (e.g. Flanigan et al., 2016), removing their ability to represent coreference—one of the very aspects of meaning that motivates AMR. Could we we instead use probabilistic models of DAGs?

To answer this question, we must define probability distributions over sets of DAGs. For inspiration, consider probability distributions over sets of strings or trees, which can be defined by weighted finite automata (e.g. Mohri et al., 2008; May et al., 2010): a finite automaton generates a set of strings or trees—called a language—and if we assume that probabilities factor over its transitions, then any finite automaton can be weighted to define a probability distribution over this language. This assumption underlies powerful dynamic programming algorithms like the Viterbi, forward-backward, and inside-outside algorithms.

What is the equivalent of weighted finite automata for DAGs? There are several candidates (Chiang et al., 2013; Björklund et al., 2016; Gilroy et al., 2017), but one appealing contender is the *DAG automaton* (Quernheim and Knight, 2012) which generalises finite tree automata to DAGs explicitly for modeling semantic graphs. These DAG automata generalise an older formalism called *planar DAG automata* (Kamimura and Slutzki, 1981) by adding weights and removing the planarity constraint, and have attracted further study (Blum and Drewes, 2016; Drewes, 2017), in particular by Chiang et al. (2018), who generalised classic dynamic programming algorithms to DAG automata. But while Quernheim and Knight (2012) clearly intend for their weights to define probabilities, they stop short of claiming that they do, instead ending their paper with an open problem: "*Investigate a reasonable probabilistic model.*"

We investigate probabilistic DAG automata and prove a surprising result: **For some DAG automata, it is impossible to assign weights that define non-trivial probability distributions**. We exhibit a very simple DAG automaton that generates an infinite language of graphs, and for which the only valid probability distribution that can be defined by weighting transitions is one in which the support is a single DAG, with all other graphs receiving a probability of zero.

Our proof relies on the fact that a non-planar DAG automaton generates DAGs so prolifically that their number grows factorially in their size, rather than exponentially as in other automata. It holds for DAG automata that allow multiple roots or nodes of unbounded degree. But it breaks down when applied to the planar DAGs of Kamimura and Slutzki (1981), which are nevertheless too restrictive to model semantic graphs. Our result does

[*] Equal contribution. Work while Ieva Vasiljeva was at the University of Edinburgh

not mean that it is impossible to define a probability distribution for the language that a DAG automaton generates. But it does mean that this distribution does not factor over the automaton's transitions, so crucial dynamic programming algorithms do not generalise to DAG automata that are expressive enough to model semantic graphs.

2 DAGs, DAG Automata, and Probability

We are interested in AMR graphs like the one below for "Rahul bakes his cake" (Figure 1, left), which represents entities and events as nodes, and relationships between them as edges. Both nodes and edges have labels, representing the type of an entity, event, or relationship. But the graphs we model will only have labels on nodes. These node-labeled graphs can simulate edge labels using a node with one incoming and one outgoing edge, as in the graph on the right of Figure 1.

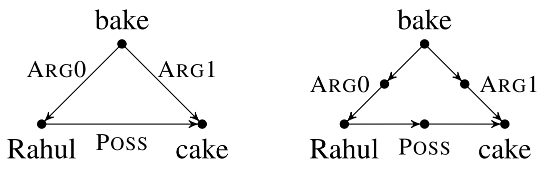

Figure 1: A graph with both node and edge labels (left) and an equivalent graph with only node labels (right).

Definition 1. *A node-labeled directed **graph** over a label set Σ is a tuple $G = (V, E, lab, src, tar)$ where V is a finite set of nodes, E is a finite set of edges, lab: $V \to \Sigma$ is a function assigning labels to nodes, src: $E \to V$ is a function assigning a source node to every edge, and tar: $E \to V$ is a function assigning a target node to every edge.*

Sometimes we will discuss the set of edges coming into or going out of a node, so we define functions IN: $V \to E^*$ and OUT: $V \to E^*$.

$$\text{IN}(v) = \{e \mid \text{tar}(e) = v\}$$
$$\text{OUT}(v) = \{e \mid \text{src}(e) = v\}$$

A node with no incoming edges is called a **root**, and a node with no outgoing edges is called a **leaf**. The **degree** of a node is the number of edges connected to it, so the degree of v is $|\text{IN}(v) \cup \text{OUT}(v)|$.

A **path** in a directed graph from node v to node v' is a sequence of edges $(e_1, \dots, e_n)$ where $\text{src}(e_1) = v$, $\text{tar}(e_n) = v'$ and $\text{src}(e_{i+1}) = \text{tar}(e_i)$ for all i from 1 to $n-1$. A **cycle** in a directed graph is any path in which the first and last nodes are the same (i.e., $v = v'$). A directed graph without any cycles is a **directed acyclic graph (DAG)**.

A DAG is **connected** if every pair of its nodes is connected by a sequence of edges, not necessarily directed. Because DAGs do not contain cycles, they must always have at least one root and one leaf, but they can have multiple roots and multiple leaves. However, our results apply in different ways to single-rooted and multi-rooted DAG languages, so, given a label set Σ, we distinguish between the set of all connected DAGs with a single root, $\mathcal{G}^1_\Sigma$; and those with one or more roots, $\mathcal{G}^*_\Sigma$.

2.1 DAG Automata

Finite automata generate strings by transitioning from state to state. Top-down tree automata generalise string finite automata by transitioning from a state to an ordered sequence of states, generating trees top-down from root to leaves; while bottom-up tree automata transition from an ordered sequence of states to a single state, generating trees bottom-up from leaves to root. The planar DAG automata of Kamimura and Slutzki (1981) generalise tree automata, transitioning from one ordered sequence of states to another ordered sequence of states (Section 4). Finally, the DAG automata of Quernheim and Knight (2012) transition from *multisets* of states to multisets of states, rather than from sequences to sequences, and this allows them to generate non-planar DAGs. We summarise the differences in Table 1 below.

Automaton	Transitions	Example
string	one-to-one	$p \to p'$
top-down tree	one-to-many	$p \to (p', q')$
bottom-up tree	many-to-one	$(p', q') \to p$
planar DAG	many-to-many	$(p, q) \to (p', q')$
non-planar DAG	many-to-many	$\{p, q\} \to \{p', q'\}$

Table 1: The forms of transitions in different automata.

For the remainder of this section and the next, we will focus only on non-planar DAG automata, and when we refer to DAG automata, we mean this type. To formally define them, we need a notation for multisets—sets that can contain repeated elements. A **multiset** is a pair (S, m) where S is a finite set and $m : S \to \mathbb{N}$ is a count function—that is, $m(x)$ counts the number of times x appears in the multiset. The set of all finite multisets over S is $M(S)$. When we write multisets, we will often simply enumerate their elements. For example, $\{p, q, q\}$ is the multiset containing one p and two

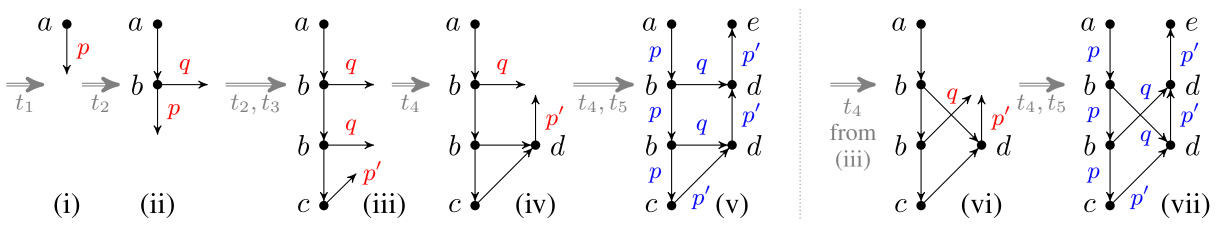

Figure 2: Two derivations using the automaton of Example 1. Parts (i), (ii), and (iii) are common to both derivations. Parts (iv) and (v) represent one completion, while (vi) and (vii) represent an alternative completion. Grey double edges denote derivation steps, labeled with the corresponding transition(s); red edge labels on partial graphs (i–iv) and (vi) denote frontier states; blue edge labels on complete graphs (v) and (vii) denote an accepting run.

q's, and since multisets are unordered, it can also be written $\{q, p, q\}$ or $\{q, q, p\}$. We write $\emptyset$ for a multiset containing no elements.

Definition 2. *A **DAG automaton** is a triple $A = (Q, \Sigma, T)$ where Q is a finite set of states; Σ is a finite set of node labels; and T is a finite set of transitions of the form $\alpha \xrightarrow{\sigma} \beta$ where $\sigma \in \Sigma$ is a node label, $\alpha \in M(Q)$ is the left-hand side, and $\beta \in M(Q)$ is the right-hand side.*

Example 1. Let $A = (Q, \Sigma, T)$ be a DAG automaton where $Q = \{p, p', q\}$, $\Sigma = \{a, b, c, d, e\}$ and the transitions in T are as follows:

$$\emptyset \xrightarrow{a} \{p\} \quad (t_1)$$
$$\{p\} \xrightarrow{b} \{p, q\} \quad (t_2)$$
$$\{p\} \xrightarrow{c} \{p'\} \quad (t_3)$$
$$\{p', q\} \xrightarrow{d} \{p'\} \quad (t_4)$$
$$\{p'\} \xrightarrow{e} \emptyset \quad (t_5)$$

2.1.1 Generating Single-rooted DAGs

A DAG automaton generates a graph from root to leaves. To illustrate this, we'll focus on the case where a DAG is allowed to have only a single root, and return to the multi-rooted case in Section 3.1. To generate the root, the DAG automaton can choose any transition with $\emptyset$ on its left-hand side—these transitions behave like transitions from the start state in a finite automaton on strings, and always generate roots. In our example, the only available transition is t_1, which generates a node labeled a with a dangling outgoing edge in state p, as in Figure 2(i). The set of all such dangling edges is the **frontier** of a partially-generated DAG.

While there are edges on the frontier, the DAG automation must choose and apply a transition whose left-hand side matches some subset of them. In our example, the automaton can choose either t_2 or t_3, each matching the available p edge. The edges associated with the matched states are attached to a new node with new outgoing frontier edges specified by the transition, and the matched states are removed from the frontier. If our automaton chooses t_2, it arrives at the configuration in Figure 2(ii), with a new node labeled b, new edges on the frontier labeled p and q, and the incoming p state forgotten. Once again, it must choose between t_2 and t_3—it cannot use the q state because that state can only be used by t_4, which also requires a p' on the frontier. So each time it applies t_2, the choice between t_2 and t_3 repeats.

If the automaton applies t_2 again and then t_3, as it has done in Figure 2(iii), it will face a new set of choices, between t_4 and t_5. But notice that choosing t_5 will leave the q states stranded, leaving a partially derived DAG. We consider a run of the automaton successful only when the frontier is empty, so this choice leads to a dead end.

If the automaton chooses t_4, it has an additional choice: it can combine p' with *either* of the available q states. If it combines with the lowermost q, it arrives at the graph in Figure 2(iv), and it can then apply t_4 to consume the remaining q, followed by t_5, which has $\emptyset$ on its right-hand side. Transitions to $\emptyset$ behave like transitions to a final state in a finite automaton, and generate leaf nodes, so we arrive at the complete graph in Figure 2(v). If the p' state in Figure 2(iii) had instead combined with the upper q, a different DAG would result, as shown in Figure 2(vi-vii).

The DAGs in Figure 2(v) and Figure 2(vii) are planar, which means they can be drawn without crossing edges.[1] But this DAG automaton can also produce non-planar DAGs like the one in Figure 3. To see that it is non-planar, we first *contract* each dotted edge by removing it and fusing its endpoints into a single node. This gives us the *minor*

[1]While the graph in Figure 2(vii) is drawn with crossing $b - d$ edges, one of these edges can be redrawn so that they do not cross.

subgraph $K_{3,3}$, and any graph with a $K_{3,3}$ minor is non-planar (Wagner, 1937).

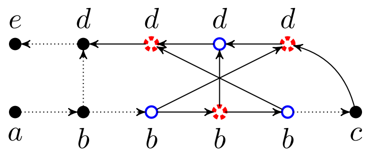

Figure 3: A non-planar graph that can be generated by the automaton of Example 1. When the dotted edges are contracted, we obtain $K_{3,3}$, the complete (undirected) bipartite graph over two sets of three nodes. One set is denoted by hollow blue nodes (○), the other by dotted red nodes (⁘).

2.1.2 Recognising DAGs and DAG Languages

We define the language generated by a DAG automaton in terms of recognition, which asks if an input DAG could have been generated by an input automaton. We recognise a DAG by finding a run of the automaton that could have generated it. To guess a run on a DAG, we guess a state for each of its edges, and then ask whether those states simulate a valid sequence of transitions.

A **run** of a DAG automaton $A = (Q, \Sigma, T)$ on a DAG $G = (V, E, \text{lab}, \text{src}, \text{tar})$ is a mapping $\rho : E \to Q$ from edges of G to automaton states Q. We extend ρ to multisets by saying $\rho(\{e_1, \ldots, e_n\}) = \{\rho(e_1), \ldots, \rho(e_n)\}$, and we call a run **accepting** if for all $v \in V$ there is a corresponding transition $\rho(\textsc{In}(v)) \xrightarrow{\text{lab}(v)} \rho(\textsc{Out}(v))$ in T. DAG G is **recognised** by automaton A if there is an accepting run of A on G.

Example 2. The DAGs in Figure 2(v) and 2(vii) are recognised by the automaton in Example 1. The only accepting run for each DAG is denoted by the blue edge labels.

The **single-rooted language** $L_s(A)$ of a DAG automaton A is $\{G \in \mathcal{G}^1_\Sigma \mid A \text{ recognizes } G\}$.

2.2 Probability and Weighted DAG Automata

Definition 3. *Given a language L of DAGs, a **probability distribution** over L is any function $p : L \to \mathbb{R}$ meeting two requirements:*

(R1) *Every DAG must have a probability between 0 and 1, inclusive. Formally, we require that for all $G \in L$, $p(G) \in [0, 1]$.*

(R2) *The probabilities of all DAGs must sum to one. Formally, we require $\sum_{G \in L} p(G) = 1$.*

R1 and R2 suffice to define a probability distribution, but in practice we need something stronger than R1: all DAGs must receive a *non-zero* weight, since in practical applications, objects with probability zero are effectively not in the language.

Definition 4. *A probability distribution p has **full support** of L if and only if it meets condition R1'.*

(R1') *Every DAG must have a probability greater than 0 and less than or equal to 1. Formally, we require that for all $G \in L$, $p(G) \in (0, 1]$.*

While there are many ways to define a function that meets requirements R1' and R2, probability distributions in natural language processing are widely defined in terms of weighted automata or grammars, so we adapt a common definition of weighted grammars (Booth and Thompson, 1973) to DAG automata.

Definition 5. *A **weighted DAG automaton** is a pair (A, w) where $A = (Q, \Sigma, T)$ is a DAG automaton and $w : T \to \mathbb{R}$ is a function that assigns real-valued weights to the transitions of A.*

Since weights are functions of transitions, we will write them on transitions following the node label and a slash (/). For example, if $p \xrightarrow{a} q$ is a transition and 2 is its weight, we write $p \xrightarrow{a/2} q$.

Example 3. Let (A, w) be a weighted DAG automaton with $A = (Q, \Sigma, T)$, where $Q = \{p, q\}$, $\Sigma = \{a, b, c\}$, and the weighted transitions of T are as follows:

$$\emptyset \xrightarrow{a/0.5} \{p, q\} \quad (t'_1) \qquad \{p, q\} \xrightarrow{c/1} \emptyset \quad (t'_3)$$
$$\{p\} \xrightarrow{b/0.5} \{p\} \quad (t'_2)$$

We use the weights on transitions to weight runs.

Definition 6. *Given a weighted DAG automaton (A, w) and a DAG $G = (V, E, lab, src, tar)$ with an accepting run ρ, we extend w to compute the **weight of the run** $w(\rho)$ by multiplying the weights of all of its transitions:*

$$w(\rho) = \prod_{v \in V} w(\rho(\textsc{In}(v)) \xrightarrow{lab(v)} \rho(\textsc{Out}(v)))$$

Example 4. The DAG automaton of Example 3 generates the DAG in Figure 4, shown with its only accepting run in blue and the weighted transitions that generated it in grey. The weight of the accepting run is $0.5 \times 0.5 \times 0.5 \times 1 = 0.125$.

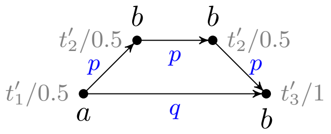

Figure 4: A DAG generated by the automaton in Example 3. Blue edge labels denote an accepting run; grey node labels denote weighted transitions used in the run.

Let $R_A(G)$ be the set of all accepting runs of a DAG G using the automaton A. We extend w to calculate the weight of a DAG G as the sum of the weights of all the runs that produce it:

$$w(G) = \sum_{\rho \in R_A(G)} w(\rho).$$

While all weighted DAG automata assign real values to DAGs, not all weighted DAG automata define probability distributions. To do so, they must also satisfy requirements R1 and R2.

Definition 7. *A weighted automaton* (A, w) *over language* $L(A)$ *is* ***probabilistic*** *if and only if function* $w : L(A) \to \mathbb{R}$ *is a probability distribution.*

Example 5. Consider the weighted automaton in Example 3. Every DAG generated by this automaton must use t'_1 and t'_3 exactly once, and can use t'_2 any number of times. If we let G_n be the DAG that uses t'_2 exactly n times, then the language L defined by this automaton is $\bigcup_{n\in\mathbb{N}} G_n$. Since $w(G_n) = w(t'_1)w(t'_2)^n w(t'_3)$ and $w(t'_1)$, $w(t'_2)$ and $w(t'_3)$ are positive, w satisfies R1 and:

$$\sum_{G\in L} w(G) = \sum_{n=0}^{\infty} w(G_n) = \sum_{n=0}^{\infty} w(t'_1)w(t'_2)^n w(t'_3)$$
$$= \sum_{n=0}^{\infty} 0.5^{n+1} = 1$$

Thus w also satisfies R2 and the weighted automaton in Example 3 is probabilistic.

Definition 8. *A probabilistic automaton* (A, w) *over language* $L(A)$ *is* ***probabilistic with full support*** *if and only if* w *has full support of* $L(A)$.

For every finite automaton over strings or trees, there is a weighting of its transitions that makes it probabilistic (Booth and Thompson, 1973), and it is easy to show that it can be made probabilistic with full support. For example, string finite automata have full support if for every state the sum of weights on its outgoing transitions is 1 and each weight is greater than 0.[2] But as we will show, this is not always possible for DAG automata.

3 Non-probabilistic DAG Automata

We will exhibit a DAG automaton that generates factorially many DAGs for a given number of nodes, and we will show that for any nontrivial assignment of weights, this factorial growth rate causes the weight of all DAGs to sum to infinity.

Theorem 1. *Let* $\mathcal{A}$ *be the automaton defined in Example 1. There is no* w *that makes* $(\mathcal{A}, w)$ *probabilistic with full support over* $L_s(\mathcal{A})$.

Proof. In any run of the automaton, transition t_1 is applied exactly once to generate the single root, placing a p on the frontier. This gives a choice between t_2 and t_3. If the automaton chooses t_2, it keeps one p on the frontier and adds a q, and must then repeat the same choice. Suppose it chooses t_2 exactly n times in succession, and then chooses t_3. Then the frontier will contain n edges in state q and one in state p'. The only way to consume all of the frontier states is to apply transition t_4 exactly n times, consuming a q at each step, and then apply t_5 to consume p' and complete the derivation. Hence in any accepting run, t_1, t_3 and t_5 are each applied once, and t_2 and t_4 are each applied n times, for some $n \geq 0$. Since transitions map uniquely to node labels, it follows that every DAG in $L_s(\mathcal{A})$ will have exactly one node each labeled a, c, and e; and n nodes each labeled b and d.

When the automaton applies t_4 for the first time, it has n choices of q states to consume, each distinguished by its unique path from the root. The second application of t_4 has $n-1$ choices of q, and the ith application of t_4 has $n - (i-1)$ choices. Therefore, there are $n!$ different ways to consume the q states, each producing a unique DAG.

Let $f(n)$ be the weight of a run where t_2 has been applied n times, and to simplify our notation, let $B = w(t_1)w(t_3)w(t_5)$, and $C = w(t_2)w(t_4)$. Let $c(n)$ be the number of unique runs where t_2 has been applied n times. By the above:

$$f(n) = w(t_1)w(t_2)^n w(t_3)w(t_4)^n w(t_5) = BC^n$$
$$c(n) = n!$$

Now we claim that any DAG in $L_s(\mathcal{A})$ has exactly one accepting run, because the mapping of

[2]Assuming no epsilon transitions, in our notation for DAG automata restricted to strings this would include transitions to $\emptyset$, which correspond to states with a final probability of 1 (Mohri et al., 2008).

node labels to transitions also uniquely determines the state of each edge in an accepting run. For example, a b node must result from a t_2 transition and a d node from a t_4 transition, and since the output states of t_2 and input states of t_4 share only a q, any edge from a b node to a d node must be labeled q in any accepting run. Now let $G \in L_s(\mathcal{A})$ be a DAG with n nodes labeled b. Since G has only one accepting run, we have:

$$w(G) = f(n)$$

Let L_n be the set of all DAGs in $L_s(\mathcal{A})$ with n nodes labeled b. Then $L_s(\mathcal{A}) = \bigcup_{n=0}^{\infty} L_n$ and:

$$\begin{aligned}\sum_{G \in L_s(\mathcal{A})} w(G) = \sum_{n=0}^{\infty} \sum_{G \in L_n} w(G) &= \sum_{n=0}^{\infty} c(n) f(n) \\ &= \sum_{n=0}^{\infty} (n!)(BC^n)\end{aligned}$$

Hence for $(\mathcal{A}, w)$ to be probabilistic with full support, R1' and R2 require us to choose B and C so that, respectively, $BC^n \in (0, 1]$ for all n and $\sum_{n=0}^{\infty} n!BC^n = 1$. Note that this does not constrain the component weights of B or C to be in $(0, 1]$—they can be any real numbers. But since R1' requires BC^n to be positive for all n, both B and C must also be positive. If either were 0, then BC^n would be 0 for $n > 0$; if either were negative, then BC^n would be negative for some or all values of n.

Now we show that any choice of positive C causes $\sum_{G \in L_s(\mathcal{A})} w(G)$ to diverge. Given an infinite series of the form $\sum_{n=0}^{\infty} a_n$, the **ratio test** (D'Alembert, 1768) considers the ratio between adjacent terms in the limit, $\lim_{n\to\infty} \frac{|a_{n+1}|}{|a_n|}$. If this ratio is greater than 1, the series diverges; if less than 1 the series converges; if exactly 1 the test is inconclusive. In our case:

$$\lim_{n\to\infty} \frac{|(n+1)!BC^{n+1}|}{|n!BC^n|} = \lim_{n\to\infty} (n+1)|C| = \infty.$$

Hence $\sum_{G \in L_s(\mathcal{A})}$ diverges for any choice of C, equivalently for any choice of weights. So there is no w for which $(\mathcal{A}, w)$ is probabilistic with full support over $L_s(\mathcal{A})$. □

Note that any automaton recognising $L_s(\mathcal{A})$ must accept factorially many DAGs in the number of nodes. Our proof implies that there is *no* probabilistic DAG automaton for language $L_s(\mathcal{A})$, since no matter how we design its transitions—each of which must be isomorphic to one in $\mathcal{A}$ apart from the identities of the states—the factorial will eventually overwhelm the constant factor corresponding to C in our proof, no matter how small it is.

Theorem 1 does not rule out *all* probabilistic variants of $\mathcal{A}$. It requires R1'—if we only require the weaker R1, then a solution of B=1 and C=0 makes the automaton probabilistic. But this trivial distribution is not very useful: it assigns all of its mass to the singleton language $\{ \begin{tikzcd} a \arrow[r] & c \arrow[r] & e \end{tikzcd} \}$.

Theorem 1 also does not mean that it is impossible to define a probability distribution over $L_s(\mathcal{A})$ with full support. If, for every DAG G with n nodes labeled b, we let $p(G) = \frac{1}{2^{n+1}n!}$, then:

$$\sum_{G \in L_s(\mathcal{A})} w(G) = \sum_{n=0}^{\infty} \frac{1}{2^{n+1}n!} n! = \sum_{n=0}^{\infty} \frac{1}{2^{n+1}} = 1$$

But this distribution does not factor over transitions, so it cannot be used with the dynamic programming algorithms of Chiang et al. (2018).

A natural way to define distributions using a DAG automaton is to define two conditional probabilities: one over the choice of nodes to rewrite, given a frontier; and one over the choice of transition, given the chosen nodes. The latter factors over transitions, but the former does not, so it also cannot use the algorithms of Chiang et al. (2018).[3]

Theorem 1 only applies to single-rooted, non-planar DAG automata of bounded degree. Next we ask whether it extends to other DAG automata, including those that recognise multi-rooted DAGs, DAGs of unbounded degree, and planar DAGs.

3.1 Multi-rooted DAGs

What happens when we consider DAG languages that allow multiple roots? In one reasonable interpretation of AMRbank, over three quarters of the DAGs have multiple roots (Kuhlmann and Oepen, 2016), so we want a model that permits this.[4]

Section 2.1.1 explained how a DAG automaton can be constrained to generate single-rooted languages, by restricting start transitions (i.e. those

[3] In this model, the subproblems of a natural dynamic program depend on the set of possible frontiers, rather than subsets of nodes as in the algorithms of Chiang et al. (2018). We do not know whether this could be made efficient.

[4] AMR annotations are single-rooted, but they achieve this by duplicating edges: every edge type, like ARG0, has an inverse type, like ARG0-OF. The number cited here assumes edges of the second type are converted to the first type by reversing their direction.

with $\emptyset$ on the left-hand side) to a single use at the start of a derivation. To generate DAGs with multiple roots, we simply allow start transitions to be applied at any time. We still require the resulting DAGs to be connected. For an automaton A, we define its **multi-rooted language** $L_m(A)$ as $\{G \in \mathcal{G}^*_\Sigma | A \text{ recognises } G\}$.

Although one automaton can define both single- and multi-rooted DAG languages, these languages are incomparable. Drewes (2017) uses a construction very similar to the one in Theorem 1 to show that single-rooted languages have very expressive path languages, which he argues are too expressive for modeling semantics.[5] Since the constructions are so similar, it natural to wonder if the problem that single-rooted automata have with probabilities is related to their problem with expressivity, and whether it likewise disappears when we allow multiple roots. We now show that multi-rooted languages have the same problem with probability, because any multi-rooted language contains the single-rooted language as a sublanguage.

Corollary 1. *Let $\mathcal{A}$ be the automaton defined in Example 1. There is no w that makes $(\mathcal{A}, w)$ probabilistic with full support over $L_m(\mathcal{A})$.*

Proof. By their definitions, $L_s(\mathcal{A}) \subset L_m(\mathcal{A})$, so:

$$\sum_{G \in L_m(\mathcal{A})} w(G) = \sum_{G \in L_s(\mathcal{A})} w(G) + \sum_{G \in L_m(\mathcal{A}) \setminus L_s(\mathcal{A})} w(G)$$

The first term is ∞ by Theorem 1 and the second is positive by R1', so the sum diverges. Hence there is no w for which $(\mathcal{A}, w)$ is probabilistic with full support over $L_m(\mathcal{A})$. □

3.2 DAGs of Unbounded Degree

The maximum degree of any node in any DAG recognised by a DAG automaton is bounded by the maximum number of states in any transition, because any transition $\alpha \xrightarrow{\sigma} \beta$ generates a node with $|\alpha|$ incoming edges and $|\beta|$ outgoing edges. So, the families of DAG languages we have considered all have bounded degree.

[5]The **path language** of a DAG is the set of strings that label a path from a root to a leaf, and the path language of a DAG language is the set of all such strings over all DAGs. For example, the path language of the DAG in Figure 2(v) is $\{abde, abbdde, abbcdde\}$. Berglund et al. (2017) show that path languages of multi-rooted DAG automata are regular, while those of single-rooted DAG automata characterised by a partially blind multi-counter automaton.

DAG languages with unbounded degree could be useful to model phenomena like coreference in meaning representations, and they have been studied by Quernheim and Knight (2012) and Chiang et al. (2018). These families generalise and strictly contain the family of bounded-degree DAG languages, so they too, include DAG automata that cannot be made probabilistic.

3.3 Implications for semantic DAGs

We introduced DAG automata as a tool for modeling the meaning of natural language, but the DAG automaton in Theorem 1 is very artificial, so it's natural to ask whether it has any real relevance to natural language. We will argue informally that this example illustrates a pervasive problem with DAG automata—specifically, we conjecture that the factorial growth we observe in Theorem 1 arises under very mild conditions that arise naturally in models of AMR.

Consider object control in a sentence like "I help Ruby help you" and its AMR in Figure 5.

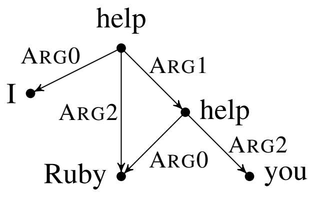

Figure 5: The AMR for "I help Ruby help you".

We can extend the control structure unboundedly with additional helpers, as in "I help Briony help Kim-Joy help Ruby help you", and this leads to unboundedly long repetitive graphs like the one in Figure 6. These graphs can be cut to separate the sequence of "help" predicates from their arguments, as illustrated by the dashed blue line.

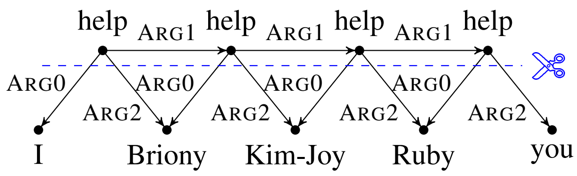

Figure 6: The AMR for "I help Briony help Kim-Joy help Ruby help you" shown with a cut.

Let a **cut** be a set of edges such that removing them splits the graph into two connected subgraphs: one containing the root, and the other containing all the leaves. Any cut in a complete graph could have been the frontier of a partially-derived graph. What if the number of edges in a cut—or **cut-width**—can be unbounded, as in the language of AMR graphs that model object control?

Since a DAG automaton can have only a finite number of states, there is some state that can occur unboundedly many times in a graph cut. All edges in a cut with this state can be rewired by permuting their target nodes, and the resulting graph will still be recognised by the automaton, since the rewiring would not change the multiset of states into or out of any node. If each possible rewiring results in a unique graph then the number of recognised graphs will be factorial in the number of source nodes for these edges, and the argument of Theorem 1 can be generalised to show that no weighting of any DAG automaton over the graph language makes it probabilistic with full support. For example, in the graph above, all possible rewirings of the ARG2 edges result in a unique graph.[6] Although edge labels are not states, their translation into node labels implies that they can only be associated to a finite number of transitions, hence to a finite number of states in any corresponding DAG automaton. A full investigation of conditions under which Theorem 1 generalises is beyond the scope of this paper.

Conjecture 1. *Under mild conditions, if language $L(A)$ of a DAG automaton A has unbounded cut-width, there is no w that makes (A, w) probabilistic with full support.*

4 Planar DAG Automata

The fundamental problem with trying to assign probabilities to non-planar DAG automata is the factorial growth in the number of DAGs with respect to the number of nodes. Does this problem occur in planar DAG automata?

Planar DAG automata are similar to the DAG automata of Section 2 but with an important difference: they transition between *ordered* sequences of states rather than unordered multisets of states. We write these sequences in parentheses, and their order matters: (p, q) differs from (q, p). We write ϵ for the empty sequence. When a planar DAG automaton generates DAGs, it keeps a strict order over the set of frontier states at all times. A transition whose left-hand side is (p, q) can only be applied to adjacent states p and q in the frontier, with p preceding q. The matched states are replaced in the frontier by the sequence of states in the transition's right-hand side, maintaining order.

Example 6. Consider a planar DAG automaton with the following transitions:

$$\begin{array}{rll} \epsilon \xrightarrow{a} (p) & (t''_1) \\ (p) \xrightarrow{b} (p, q) & (t''_2) \\ (p) \xrightarrow{c} (p') & (t''_3) \end{array} \qquad \begin{array}{rll} (p', q) \xrightarrow{d} (p') & (t''_4) \\ (p') \xrightarrow{e} \epsilon & (t''_5) \end{array}$$

In the non-planar case, n applications of t_2 can generate $n!$ unique DAGs, but n applications of the corresponding transition t''_2 in this automaton can only generate one DAG. To see this, consider the partially derived DAG on the left of Figure 7, with its frontier drawn in order from left to right. The p' state can only combine with the q state immediately to its right, and since dead-ends are not allowed, the only possible choice is to apply t''_4 twice followed by t''_5, so the DAG on the right is the only possible completion of the derivation.

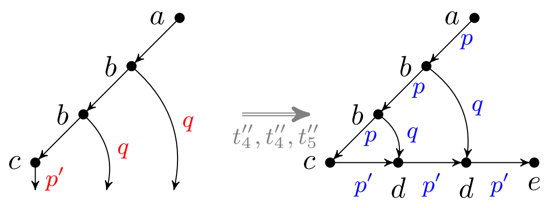

Figure 7: A partial derivation using the planar DAG automaton of Example 6 (left; red edge labels denote frontier states) and its only possible completion (right; blue edge labels denote an accepting run).

This automaton is probabilistic when $w(t''_1) = w(t''_2) = 1/2$, $w(t''_3) = w(t''_4) = w(t''_5) = 1$, and indeed the argument in Theorem 1 does not apply to planar automata since the number of applicable transitions is linear in the size of the frontier. But planar DAG automata have other problems that make them unsuitable for modeling AMR.

The first problem is that there are natural language constructions that naturally produce non-planar DAGs in AMR. For example, consider the sentence "Four contestants mixed, baked and ate cake." Its AMR, shown in Figure 8, is not planar because it has a $K_{3,3}$ minor, and it is easy to see from this example that any coordination of three predicates sharing two arguments produces this structure. In the first release of AMR, 117 out of 12844 DAGs are non-planar.

The second problem is that planar DAG au-

[6]This is also a problem linguistically, since many of the rewired graphs no longer model object control.

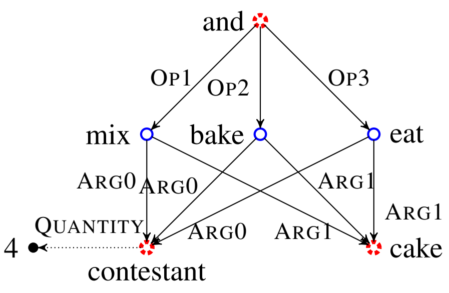

Figure 8: An AMR for the sentence “Four contestants mixed, baked and ate a cake”. As in Figure 3, contracting the dotted edge yields a $K_{3,3}$ minor, with one set denoted by hollow blue nodes (○), the other by dotted red nodes (⁘).

tomata model Type-0 string derivations by design (Kamimura and Slutzki, 1981). This seems more expressive than needed to model natural language and means that many important decision problems are undecidable—for example, emptiness, which is decidable in polynomial time for non-planar DAG automata (Chiang et al., 2018).

5 Conclusions

Table 2 summarises the properties of several different variants of DAG automata. It has been argued that all of these properties are desirable for probabilistic models of meaning representations (Drewes, 2017). Since none of the variants supports all properties, this suggests that no variant of the DAG automaton is a good candidate for modeling meaning representations. We believe other formalisms may be more suitable, including several subfamilies of hyperedge replacement grammars (Drewes et al., 1997) that have recently been proposed (Björklund et al., 2016; Matheja et al., 2015; Gilroy et al., 2017).

	non-planar				planar
bounded degree	yes		no		yes
roots	1	1+	1	1+	1
probabilistic	no	no	no	no	?
decidable	yes	yes	yes	yes	no
regular paths	no	yes	no	yes	no

Table 2: DAG automata variants and their properties.

Acknowledgements

This work was supported in part by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh. We thank Esma Balkir, Nikolay Bogoychev, Shay Cohen, Marco Damonte, Federico Fancellu, Joana Ribeiro, Nathan Schneider, Miloš Stanojević, Ida Szubert, Clara Vania, and the anonymous reviewers for helpful discussion of this work and comments on previous drafts of the paper.

References

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria.

Martin Berglund, Henrik Björklund, and Frank Drewes. 2017. Single-rooted dags in regular dag languages: Parikh image and path languages. In *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms*, pages 94–101, Umeå, Sweden.

Henrik Björklund, Frank Drewes, and Petter Ericson. 2016. Between a rock and a hard place - uniform parsing for hyperedge replacement DAG grammars. In *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings*, pages 521–532.

Johannes Blum and Frank Drewes. 2016. Properties of regular DAG languages. In *Language and Automata Theory and Applications - 10th International Conference, LATA 2016, Prague, Czech Republic, March 14-18, 2016, Proceedings*, pages 427–438.

T.L. Booth and R.A. Thompson. 1973. Applying probability measures to abstract languages. *IEEE Transactions on Computers*, 22(5):442–450.

David Chiang, Jacob Andreas, Daniel Bauer, Karl Moritz Hermann, Bevan Jones, and Kevin Knight. 2013. Parsing graphs with hyperedge replacement grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 924–932, Sofia, Bulgaria.

David Chiang, Frank Drewes, Daniel Gildea, Adam Lopez, and Giorgio Satta. 2018. Weighted DAG automata for semantic graphs. *Computational linguistics*, 44(1).

Jean D’Alembert. 1768. *Opuscules*, volume V.

Frank Drewes. 2017. Dag automata for meaning representation. In *Proceedings of the 15th Meeting on the Mathematics of Language*, pages 88–99, London, UK.

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

Jeffrey Flanigan, Chris Dyer, Noah A. Smith, and Jaime Carbonell. 2016. Generation from abstract meaning representation using tree transducers. In *Proc. of NAACL-HLT*, pages 731–739.

Sorcha Gilroy, Adam Lopez, Sebastian Maneth, and Pijus Simonaitis. 2017. (Re)introducing regular graph languages. In *Proceedings of the 15th Meeting on the Mathematics of Language (MoL 15)*, pages 100–113.

Tsutomu Kamimura and Giora Slutzki. 1981. Parallel and two-way automata on directed ordered acyclic graphs. *Information and Control*, 49(1):10–51.

Marco Kuhlmann and Stephan Oepen. 2016. Squibs: Towards a catalogue of linguistic graph banks. *Computational Linguistics*, 42(4):819–827.

Christoph Matheja, Christina Jansen, and Thomas Noll. 2015. *Tree-Like Grammars and Separation Logic*, pages 90–108. Springer International Publishing, Cham.

Jonathan May, Kevin Knight, and Heiko Vogler. 2010. Efficient inference through cascades of weighted tree transducers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, ACL '10, pages 1058–1066.

Mehryar Mohri, Fernando C. N. Pereira, and Michael Riley. 2008. Speech recognition with weighted finite-state transducers. In Larry Rabiner and Fred Juang, editors, *Handbook on Speech Processing and Speech Communication, Part E: Speech recognition*, pages 69–88. Springer.

Rik van Noord and Johan Bos. 2017. Neural semantic parsing by character-based translation: Experiments with abstract meaning representations. *Computational Linguistics in the Netherlands Journal*, 7:93–108.

Daniel Quernheim and Kevin Knight. 2012. Towards probabilistic acceptors and transducers for feature structures. In *Proceedings of the Sixth Workshop on Syntax, Semantics and Structure in Statistical Translation*, SSST-6 '12, pages 76–85.

Klaus Wagner. 1937. Über eine eigenschaft der ebenen komplexe. *Mathematische Annalen*, 114(1):570–590.