Counting the Answers to a Query

Marcelo Arenas
PUC & IMFD Chile
marenas@uc.cl

Luis Alberto Croquevielle
PUC and IMFD Chile
lacroquevielle@uc.cl

Rajesh Jayaram
Google Research
rkjayaram@google.com

Cristian Riveros
PUC and IMFD Chile
cristian.riveros@uc.cl

ABSTRACT

Counting the answers to a query is a fundamental problem in databases, with several applications in the evaluation, optimization, and visualization of queries. Unfortunately, counting query answers is a #P-hard problem in most cases, so it is unlikely to be solvable in polynomial time. Recently, new results on approximate counting have been developed, specifically by showing that some problems in automata theory admit fully polynomial-time randomized approximation schemes. These results have several implications for the problem of counting the answers to a query; in particular, for graph and conjunctive queries. In this work, we present the main ideas of these approximation results, by using labeled DAGs instead of automata to simplify the presentation. In addition, we review how to apply these results to count query answers in different areas of databases.

1. INTRODUCTION

Query answering is arguably the most important problem in databases. In its full generality, the answers to a query come in different sizes and flavors: in graph databases, a query can retrieve nodes and paths; over documents, the answers are spans or subsections of files; and on relational databases, answers are given as a set of tuples. Although data models and query languages differ on the outcome, one output is ubiquitous in all scenarios: counting the number of query answers.

Counting query answers is indeed a fundamental problem in data management systems. In fact, most query languages include a `COUNT` clause for retrieving the number of answers of a query. For query optimization, counting the number of answers could help in the optimization process, when the number of partial answers of a subquery is needed. Even for user experience, displaying the number of answers could help a user to know how many of them there are, before overflowing the screen with answers. In all these scenarios, efficiently counting query answers is crucial for the performance of the database system.

The main challenge of counting query answers is that the number of answers could be exponential in the size of the data. For this reason, *directly counting* the answers by evaluating the query is computationally expensive in practice. Even if the number of solutions is polynomial or linear in the data size, this strategy is expensive for all the applications mentioned above. A better solution would be *symbolically counting* the number of answers; the query engine will not evaluate the query and, instead, it will symbolically obtain the number of solutions directly from the query and data. For instance, in the past, database researchers have used this approach for counting the number of answers of acyclic conjunctive queries without projection, the number of paths of a certain length in a graph database, and the number of strings accepted by a deterministic automaton.

The bad news is that the symbolic approach is not always possible in data management, since there are computational complexity barriers that do not allow efficient algorithms for counting query answers. In the late 70s, Valiant defined the class of #P-hard problems [26], which is consider as the counting analog of NP-hard problems. Specifically, Valiant shows that several counting problems from different areas are #P-hard [26, 27] and, similar to NP-hardness, researchers widely believed that there are no polynomial-time algorithms for solving them. Unfortunately, the #P-hard disease also contaminates counting query answers, even over data management problems whose decision versions are tractable. For instance, it can be decided in linear-time whether there is an answer for an acyclic conjunctive query, but it is #P-hard to count the number of answers for such a query [23]. Notice that in these results the query and the database are part of this input, so we are considering the combined complexity of the problem (as opposed to data complexity where the query is assumed to be fixed [28]). All of the results in this article are about the combined complexity of counting query answers.

One possible approach for tackling #P-hard problems is to approximate the counting, namely, to have an algorithm that, given the input, outputs a value $\bar{n}$ with a relative error of ε (i.e., $|n - \bar{n}| \leq \varepsilon \cdot n$ for the actual value n). If this algorithm is randomized, then the approximation is expected to be close to the actual value with a high probability. In addition, if this approximation algorithm runs in polynomial time, we have what is called a *Fully Polynomial-time Randomized Approximation Scheme* (FPRAS [18]). Interestingly, some #P-hard problems admit FPRASs, like counting the number of truth assignments that satisfy a propositional formula in DNF [19] and counting the number of perfect matchings in a bipartite graph [17]. Since counting the number of query answers in databases is usually complex (i.e., #P-hard), our goal is to obtain an FPRAS for this problem. Unfortunately, finding an FPRAS for a #P-hard problem is challenging, and there are only a few counting problems in data management that are known to admit FPRASs.

The work in [4, 5] provided new results on counting query answers, by developing techniques that solve counting problems related to query answering. Specifically, an FPRAS for the problem of counting the number of strings accepted by a non-deterministic finite automaton is given in [4]. Moreover, the extension of this result to tree automata is given in [5]. Interestingly, both problems live at the core of several query answering problems in areas like graph databases, information extraction, and conjunctive query evaluation, thus allowing the development of efficient approximation algorithms for such query answering problems.

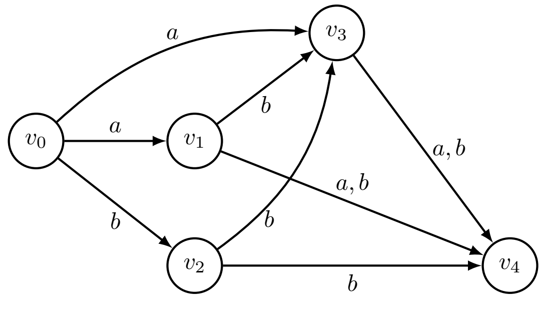
The goal of this document is to present the main ideas and results in [4, 5]. These results heavily rely on automata theory, which requires some knowledge of the reader in this topic. Instead, this paper provides a uniform setting for the results in [4, 5] based on a counting problem for labeled DAGs. In this way, we simplify the notation and the presentation of the main ideas.

The paper is organized as follows. In Section 2, we present the counting problem over labeled DAGs and the main ideas of its FPRAS. In Section 3, we extend these ideas to *succinct* labeled DAGs, which is the main artifact for the results in [5]. In Section 4, we present applications of these results for automata theory, graph databases, and conjunctive query evaluation.

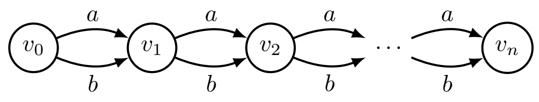
2. COUNTING IN LABELED DAGS

We begin by defining the main problem studied in this work. In what follows, assume that Σ is a fixed finite alphabet containing at least two symbols. A *labeled DAG* is a tuple $D = (V, E)$ such that V is a finite set of vertices, $E \subseteq V \times \Sigma \times V$ is a (finite) set of labeled edges, and the directed graph $G = (V, \{(u,v) \mid \exists a : (u,a,v) \in E\})$ is acyclic. The size of D is defined as $|D| = |V| + |E|$.

Each vertex u of a labeled DAG $D = (V, E)$ defines a language $\mathcal{L}_D(u)$ over the alphabet Σ. Formally, a vertex u of D is said to be a sink if u has no outgoing edges (i.e., $(u,a,v) \notin E$ for every $a \in \Sigma$ and $v \in V$). Then



(a) Labeled DAG D_1



(b) Labeled DAG D_2

Figure 1: Two labeled DAGs with alphabet $\{a, b\}$.

$\mathcal{L}_D(u) = \{\lambda\}$ if u is a sink vertex in D, where λ is the empty string. Otherwise, $\mathcal{L}_D(u)$ is recursively defined as follows:

$$\mathcal{L}_D(u) \;=\; \bigcup_{(u,a,v)\in E} \{a\} \cdot \mathcal{L}_D(v),$$

where, given two sets L_1 and L_2 of strings, $L_1 \cdot L_2$ is defined as the set of strings consisting of the concatenation of each string of L_1 with each string of L_2. Given that D is acyclic, it can be easily verified that the set of strings $\mathcal{L}_D(u)$ is correctly defined for every $u \in V$. As an example, we show a labeled DAG D_1 in Figure 1a. In this case, we have that $\mathcal{L}_{D_1}(v_4) = \{\lambda\}$ as v_4 is a sink vertex in D_1, and $\mathcal{L}_{D_1}(v_3) = \{a\} \cdot \mathcal{L}_{D_1}(v_4) \cup \{b\} \cdot \mathcal{L}_{D_1}(v_4) = \{a, b\}$. Moreover, we have that $\mathcal{L}_{D_1}(v_1) = \{b\} \cdot \mathcal{L}_{D_1}(v_3) \cup \{a\} \cdot \mathcal{L}_{D_1}(v_4) \cup \{b\} \cdot \mathcal{L}_{D_1}(v_4) = \{ba, bb, a, b\}$, and $\mathcal{L}_{D_1}(v_2) = \{ba, bb, b\}$. Finally, we have that $\mathcal{L}_{D_1}(v_0) = \{aba, abb, aa, ab, bba, bbb, bb\}$. Notice that for a labeled DAG D and a vertex u of D, it can be the case that $\mathcal{L}_D(u)$ is of exponential size in $|D|$. For example, for the labeled DAG D_2 in Figure 1b, it holds that $|\mathcal{L}_{D_2}(v_0)|$ is 2^n.

In this work, we are interested in the following counting problem:

Problem:	#LDAG
Input:	A labeled DAG D and a vertex u of D
Output:	$\lvert\mathcal{L}_D(u)\rvert$

Why is #LDAG an interesting problem? The setting defined for labeled DAGs serves as an abstraction to represent many other counting problems. In fact, we will show in Section 4 some consequences of our results for labeled DAGs, which are obtained by representing different problems in this setting. As an example of this, below we show how the problem #DNF can be reduced

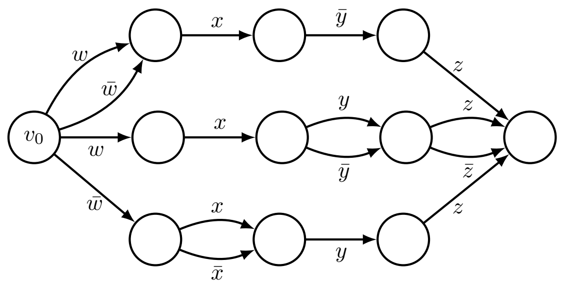


Figure 2: A labeled DAG encoding propositional formula $(x \wedge \neg y \wedge z) \vee (w \wedge x) \vee (\neg w \wedge y \wedge z)$.

to #LDAG.

Recall that #DNF is the problem of counting, given a propositional formula φ in DNF, the number of truth assignments that satisfy φ. For example, there are 7 truth assignments that satisfy the propositional formula $\psi = (x \wedge \neg y \wedge z) \vee (w \wedge x) \vee (\neg w \wedge y \wedge z)$, one of which is $\sigma(w) = 0$, $\sigma(x) = 1$, $\sigma(y) = 0$ and $\sigma(z) = 1$. To encode this counting problem for ψ as a counting problem for labeled DAGs, we represent each truth assignment for the set of variables $\{w, x, y, z\}$ as a string over the alphabet $\{w, \bar{w}, x, \bar{x}, y, \bar{y}, z, \bar{z}\}$, where symbol x is used to indicate that variable x is assigned value 1, while symbol $\bar{x}$ is used to indicate variable x is assigned value 0, and likewise for the other variables. Then the set of strings representing truth assignments that satisfy $(x \wedge \neg y \wedge z)$ can be succinctly encoded as the set of paths from v_0 to a sink vertex in the following DAG:

$$\begin{tikzcd} v_0 \arrow[r, bend left, "w"] \arrow[r, bend right, "\bar{w}"'] & \circ \arrow[r, "x"] & \circ \arrow[r, "\bar{y}"] & \circ \arrow[r, "z"] & \circ \end{tikzcd}$$

In this way, the set of truth assignments that satisfy ψ is represented by $\mathcal{L}_D(v_0)$, where D is the labeled DAG shown in Figure 2. In particular, the number of such truth assignments is equal to $|\mathcal{L}_D(v_0)|$. This idea can be easily generalized to any propositional formula in DNF.[1]

How difficult is #LDAG? It is straightforward to prove that #LDAG is in the complexity class #P of problems that can be expressed as the number of accepting paths of a polynomial-time nondeterministic Turing machine [26]. Besides, as #DNF is known to be #P-complete [24] and above we provide a reduction from #DNF to #LDAG, we conclude that #LDAG is #P-complete. Therefore, #LDAG is a difficult problem that is not expected to be solvable in polynomial time (under standard complexity theoretical assumptions). However, this result does not preclude the existence of efficient approximation algorithms for this problem. More precisely, the task is to obtain a fully polynomial-time randomized approximation scheme (FPRAS [18]) for #LDAG, which is a randomized algorithm satisfying the following conditions. The input of the algorithm is a labeled DAG D, a vertex u of D and an approximation error $\varepsilon \in (0, 1)$, and the task is to compute a value N such that

$$\Pr\big((1-\varepsilon)\cdot|\mathcal{L}_D(u)| \;\leq\; N \;\leq\; (1+\varepsilon)\cdot|\mathcal{L}_D(u)|\big) \;\;\geq\;\; \frac{3}{4}.$$

Moreover, there must exist a fixed polynomial $p(x, y)$ such that the algorithm executes at most $p(|D|, \frac{1}{\varepsilon})$ steps to compute N.

It is known that #DNF admits an FPRAS [20], so the existence of a reduction from #DNF to #LDAG does not preclude the existence of an FPRAS for #LDAG. In fact, we obtain the following positive result:

THEOREM 2.1. #LDAG *admits an FPRAS.*

In this section, we explain the main ideas behind the proof of this result.

2.1 The algorithmic template of an FPRAS for #LDAG

For the rest of this section, assume that the input of the approximation algorithm for #LDAG consists of a labeled DAG $D = (V, E)$, a vertex $v_s \in V$ and an approximation error $\varepsilon \in (0, 1)$. Moreover, for the sake of readability, assume that $\mathcal{L}(u)$ refers to the language $\mathcal{L}_D(u)$.

Given a vertex $v \in V$, define the *level* of v in D as the maximum length of a string in $\mathcal{L}(v)$, namely, $\text{level}(v) = \max_{s\in\mathcal{L}(v)} |s|$. Notice that $\text{level}(v) = 0$ if and only if v is a sink vertex. This definition is extended to each nonempty set of vertices $P \subseteq V$ as $\text{level}(P) = \max_{v\in P} \text{level}(v)$. Let $n = |V|$ and $m = \text{level}(V)$, and fix a value $\kappa = \lceil \frac{nm}{\varepsilon} \rceil$. Moreover, assume that $n \geq 2$ and $m \geq 2$, as otherwise #LDAG can be easily solved in polynomial time by an exhaustive computation. Then for each vertex $v \in V$, the approximation algorithm stores a number $N(v)$ and a set $S(v) \subseteq \mathcal{L}(v)$ such that:

- $N(v)$ is a $(1\pm\kappa^{-2})^{\ell}$-approximation of $|\mathcal{L}(v)|$, where $\ell = \text{level}(v)$, and
- $S(v)$ is a uniform sample from $\mathcal{L}(v)$ of size $2\kappa^7$.

The first condition requires that the following be true:

$$(1-\kappa^{-2})^{\ell}\cdot|\mathcal{L}(v)| \;\leq\; N(v) \;\leq\; (1+\kappa^{-2})^{\ell}\cdot|\mathcal{L}(v)|.$$

In particular, if $\ell = 0$ (a sink vertex), we should have that $N(v) = |\mathcal{L}(v)| = 1$. The second condition requires that each $w \in S(v)$ is a uniform and independent sample

[1]The reader may have noticed the similarity of the labeled DAGs used to encode DNF formulas to the notion of binary decision diagram used in knowledge compilation [8]. In fact, popular representations used in this area, such as ordered or free binary decision diagrams [8], can be easily encoded by using labeled DAGs. Besides, some nondeterministic variants of these representations [1] can also be encoded as labeled DAGs, thus allowing to transfer the results of this article to such representations.

1. For each vertex $v \in V$:
   (a) Compute $N(v)$ given $\{N(u), S(u) \mid \exists a : (v, a, u) \in E\}$. For $\text{level}(v) = 0$, the value $N(v) = 1$ is computed without any additional information.
   (b) Call a subroutine to sample polynomially many uniform elements from $\mathcal{L}(v)$ using the value $N(v)$ and the set $\{N(u), S(u) \mid \exists a : (v, a, u) \in E\}$.
   (c) Let $S(v) \subseteq \mathcal{L}(v)$ be the multiset of uniform samples obtained.
2. Return $N(v_s)$.

Figure 3: Template of the FPRAS for #LDAG (to be instantiated in Sections 2.2 and 2.3).

from $\mathcal{L}(v)$. Given this condition on the samples, it is possible to obtain duplicates of an element $w \in \mathcal{L}(v)$. In particular, if $|\mathcal{L}(v)| < 2\kappa^7$, then $S(v)$ has to contain duplicate elements. Therefore, we allow $S(v)$ to be a multiset (meaning that the strings w in $S(v)$ are not necessarily distinct). The number $N(v)$ and the set $S(v)$ can be understood as a summary of $\mathcal{L}(v)$ that are used in the approximation algorithm to estimate other quantities.

Having the above terminology, we provide in Figure 3 an algorithmic template of our FPRAS for #LDAG [4]. Notice that it proceeds like a dynamic programming algorithm, computing $N(v)$ and $S(v)$ for every vertex v of D in a depth-first search ordering. In particular, it first computes $N(v), S(v)$ for each sink vertex v, and then it computes $N(v), S(v)$ for each vertex v such that $\{N(u), S(u) \mid \exists a : (v, a, u) \in E\}$ has been computed. The final estimate for $|\mathcal{L}(v_s)|$ is $N(v_s)$, where v_s is the input vertex.

In the rest of this section, we show how to instantiate the template of our approximation algorithm.

2.2 Computing an estimate for a set of vertices

Recall that the input of the problem is a labeled DAG $D = (V, E)$ with $n = |V|$ and $m = \text{level}(V)$. Moreover, recall that we assume $n \geq 2$ and $m \geq 2$, and we define $\kappa = \lceil \frac{nm}{\varepsilon} \rceil$. Given $\ell \leq m$, define $V_\ell \subseteq V$ as the set of all vertices $v \in V$ such that $\text{level}(v) \leq \ell$, and define a sketch data structure $\text{sketch}[\ell] := \{N(v), S(v) \mid v \in V_\ell\}$. Moreover, assume that $\text{sketch}[\ell]$ has already been computed. In particular, $N(v)$ is a $(1 \pm \kappa^{-2})^\ell$-approximation of $|\mathcal{L}(v)|$, and $S(v)$ is a uniform sample from $\mathcal{L}(v)$ of size $2\kappa^7$ for each $v \in V_\ell$ (notice that if $N(v)$ is a $(1 \pm \kappa^{-2})^r$-approximation of $|\mathcal{L}(v)|$, then it is a $(1 \pm \kappa^{-2})^s$-approximation of $|\mathcal{L}(v)|$ for every $s \geq r$).

Given $P \subseteq V$, define $\mathcal{L}(P) = \bigcup_{v \in P} \mathcal{L}(v)$. The goal of this section is twofold; we first show how to compute an estimate of $|\mathcal{L}(P)|$ for every $P \subseteq V_\ell$, which is denoted by $N(P)$, and then we show how to compute an estimate of $N(v)$ for each vertex $v \in V_{\ell+1}$. Notice that the values $N(P)$ will play a crucial role for computing not only $N(v)$, but also the set of uniform samples $S(v)$.

Our first task is then to provide an algorithm to estimate $N(P)$, where P is a nonempty subset of V_ℓ. Notice that if $\mathcal{L}(v_1) \cap \mathcal{L}(v_2) = \varnothing$ for each pair v_1, v_2 of distinct vertices from P, then $|\mathcal{L}(P)| = \sum_{v \in P} |\mathcal{L}(v)|$. Hence an estimate of $N(P)$ can be easily constructed from the estimates $N(v)$ for the vertices $v \in P$: $N(P) = \sum_{v \in P} N(v)$. Unfortunately, the previous non-overlapping condition does not hold in general, as shown in Figure 1a, where $\mathcal{L}(v_1) \cap \mathcal{L}(v_2) \neq \varnothing$. Thus, $\sum_{v \in P} N(v)$ is an over-approximation of the size of $|\mathcal{L}(P)|$, and we need to find a way to deal with the nonempty intersections of the sets $\{\mathcal{L}(v) \mid v \in P\}$.

To solve the previous issue, fix a total order $\prec$ over P, and consider the following way to compute $|\mathcal{L}(P)|$:

$$|\mathcal{L}(P)| = \sum_{v \in P} |\mathcal{L}(v) \setminus \bigcup_{u \in P \,:\, u \prec v} \mathcal{L}(u)|.$$

The question then is how to compute an estimate of $|\mathcal{L}(v) \setminus \bigcup_{u \in P : u \prec v} \mathcal{L}(u)|$ for each vertex $v \in P$. To do this, consider the following reformulation of the previous equation:

$$|\mathcal{L}(P)| = \sum_{v \in P} |\mathcal{L}(v)| \cdot \frac{|\mathcal{L}(v) \setminus \bigcup_{u \in P : u \prec v} \mathcal{L}(u)|}{|\mathcal{L}(v)|} \quad (1)$$

We call the ratio $|\mathcal{L}(v) \setminus \bigcup_{u \in P : u \prec v} \mathcal{L}(u)| / |\mathcal{L}(v)|$ the intersection rate of v. Inspired by equation (1), we can estimate $|\mathcal{L}(P)|$ by using $N(v)$ to estimate $|\mathcal{L}(v)|$ and $S(v)$ to estimate the intersection rate of v. More precisely, we define the estimate $N(P)$ for $|\mathcal{L}(P)|$ as follows:

$$N(P) = \sum_{v \in P} N(v) \cdot \frac{|S(v) \setminus \bigcup_{u \in P : u \prec v} \mathcal{L}(u)|}{|S(v)|}$$

It is important to note that $N(P)$ can be computed in polynomial time in the size of $\text{sketch}[\ell]$. In fact, the set $S(v) \setminus \bigcup_{u \in P : u \prec v} \mathcal{L}(u)$ can be computed by iterating over each string $w \in S(v)$, and checking whether $w \notin \mathcal{L}(u)$ for each $u \in P$ such that $u \prec v$. Notice that this can be done in polynomial time in $|D|$ and $\frac{1}{\varepsilon}$, as the size of $S(v)$ is $2\kappa^7$, and it can be verified in polynomial time whether $w \in \mathcal{L}(v)$, given $w \in \Sigma^*$ and $v \in V$ as input. We call the ratio $|S(v) \setminus \bigcup_{u \in P : u \prec v} \mathcal{L}(u)| / |S(v)|$ the estimate of the intersection rate of v.

To show that $N(P)$ is a good estimate for $|\mathcal{L}(P)|$, we need that the estimate of the intersection rate of a vertex v is good approximation of the (actual) intersection rate of v. By a good approximation, we mean that the following condition $\mathcal{C}(\ell)$ holds at each level ℓ:

$$\mathcal{C}(\ell) \quad := \quad \forall v \in V_\ell \;\; \forall P \subseteq V_\ell$$

$$\left| \frac{|\mathcal{L}(v) \setminus \bigcup_{u \in P} \mathcal{L}(u)|}{|\mathcal{L}(v)|} - \frac{|S(v) \setminus \bigcup_{u \in P} \mathcal{L}(u)|}{|S(v)|} \right| < \frac{1}{\kappa^3}$$

As an immediate consequence of the definition of $\mathcal{C}(\ell)$, we have that $\mathcal{C}(k)$ holds for every $k \leq \ell$ if $\mathcal{C}(\ell)$ is true. Conditions $\{\mathcal{C}(\ell) \mid \ell \leq m\}$ are crucial to the results presented in this and the next section, and most of our analysis in Section 2 assumes they are true. In Section 2.4, we show that, by Hoeffding's inequality, condition $\mathcal{C}(m)$ holds with a probability that is exponentially large over κ.

As a first consequence of our assumption that condition $\mathcal{C}(\ell)$ is true, we show that $N(P)$ is a good estimate for $|\mathcal{L}(P)|$.

Proposition 2.2. *If $\mathcal{C}(\ell)$ holds, then $N(P)$ is a $(1 \pm \kappa^{-2})^{\ell+1}$-approximation of $|\mathcal{L}(P)|$ for every $P \subseteq V_\ell$.*

With the estimates of $|\mathcal{L}(P)|$ for every $P \subseteq V_\ell$, we are ready to give an estimate of $|\mathcal{L}(v)|$ for each vertex $v \in V_{\ell+1}$. For every $b \in \Sigma$ (recall that Σ is the fixed alphabet for the edge labels of D), define the set of vertices $P_b = \{u \in V \mid (v, b, u) \in E\}$. Thus, P_b is the set of all vertices that can be reached from v by following an edge with label b. Notice that $P_b \subseteq V_\ell$ for each $b \in \Sigma$, and that $\{P_b \mid b \in \Sigma\}$ partitions $\mathcal{L}(v)$ in the sense that $\mathcal{L}(v) = \biguplus_{b \in \Sigma} \{b\} \cdot \mathcal{L}(P_b)$, where $A \uplus B$ represents the disjoint union of sets A and B. Hence, the previous equation implies that

$$|\mathcal{L}(v)| = \sum_{b \in \Sigma} |\{b\} \cdot \mathcal{L}(P_b)| = \sum_{b \in \Sigma} |\mathcal{L}(P_b)|. \quad (2)$$

If we assume $\mathcal{C}(\ell)$ holds, then we have by Proposition 2.2 that $N(P_b)$ is a $(1 \pm \kappa^{-2})^{\ell+1}$-approximation of $|\mathcal{L}(P_b)|$ for each $b \in \Sigma$. That is, we know that the following condition holds for each $b \in \Sigma$:

$$(1 - \kappa^{-2})^{\ell+1} \cdot |\mathcal{L}(P_b)| \leq N(P_b) \leq (1 + \kappa^{-2})^{\ell+1} \cdot |\mathcal{L}(P_b)|.$$

Hence, we obtain that

$$\sum_{b \in \Sigma} (1 - \kappa^{-2})^{\ell+1} \cdot |\mathcal{L}(P_b)| \leq \sum_{b \in \Sigma} N(P_b) \leq \sum_{b \in \Sigma} (1 + \kappa^{-2})^{\ell+1} \cdot |\mathcal{L}(P_b)|,$$

and we can conclude from equation (2) that

$$(1 - \kappa^{-2})^{\ell+1} |\mathcal{L}(v)| \leq \sum_{b \in \Sigma} N(P_b) \leq (1 + \kappa^{-2})^{\ell+1} \cdot |\mathcal{L}(v)|.$$

Therefore, we have that $N(v) = \sum_{b \in \Sigma} N(P_b)$ is a $(1 \pm \kappa^{-2})^{\ell+1}$-approximation of $|\mathcal{L}(v)|$, so we can derive an estimate $N(v)$ for $|\mathcal{L}(v)|$ by using the previously calculated estimates for V_ℓ.

Notice that for each $v \in V_{\ell+1}$, the computation of $N(v)$ is deterministic if we assume that $\mathcal{C}(\ell)$ holds. Specifically, the estimate $N(v)$ is exact for each vertex $v \in V_0$. Next, for each level $k \leq \ell$, we have that $\mathcal{C}(k)$ holds, and we can compute $N(v)$ for level $k+1$ by using $\{N(u) \mid u \in V_k\}$ (in fact, by using some of the sets in $\{N(P) \mid P \subseteq V_k\}$). Then, we continue with the computation assuming that $\mathcal{C}(k+1)$ is true, until we reach level ℓ. Therefore, by filling the sets $\{S(u) \mid u \in V_\ell\}$ with uniform samples, and assuming that $\mathcal{C}(\ell)$ holds, we can compute each estimate $N(v)$ for $v \in V_{\ell+1}$ guaranteeing that $N(v)$ is a $(1 \pm \kappa^{-2})^{\ell+1}$-approximation of $|\mathcal{L}(v)|$. We summarize this fact in the following proposition.

Proposition 2.3. *If $\mathcal{C}(\ell)$ holds, then $N(v)$ is a $(1 \pm \kappa^{-2})^{\ell+1}$-approximation of $|\mathcal{L}(v)|$ for every $v \in V_{\ell+1}$.*

Recall that our goal is to compute an estimate of $|\mathcal{L}(v_s)|$, where v_s is the input vertex. The following proposition shows that such an estimate is obtained after processing all levels in the labeled DAG D.

Proposition 2.4. *If $\mathcal{C}(m)$ holds, then $N(v_s)$ is a $(1 \pm \varepsilon)$-approximation for $|\mathcal{L}(v_s)|$.*

In the next section, we show how to compute the set $S(v)$ using sketch$[\ell]$, namely, how to generate a uniform sample from $\mathcal{L}(v)$. Specifically, we show that assuming $\mathcal{C}(\ell)$ holds, we can obtain uniform samples from the sets $\mathcal{L}(v)$ such that condition $\mathcal{C}(\ell+1)$ will hold with high probability.

2.3 Uniform sampling from a vertex

To carry out our main approximation algorithm, we must implement the algorithm template in Figure 3, whose input is a labeled DAG $D = (V, E)$, where $n = |V|$, $m = \text{level}(V)$, $m \geq 2$ and $n \geq 2$. In the previous section, we implemented Step (a) of this algorithm and, thus, the goal of this section is to implement the sampling subroutine in Step (b). This procedure is based on a sample technique proposed in [18], but modified to suit our setting.

Recall that V_ℓ is the set of all vertices $v \in V$ such that $\text{level}(v) \leq \ell$. Let $v \in V_\ell$, and assume that the condition $\mathcal{C}(\ell - 1)$ holds. Notice that by Proposition 2.3, once we have $\mathcal{C}(\ell - 1)$, we immediately get the estimate $N(v)$ of $|\mathcal{L}(v)|$. The procedure to sample a uniform element of the set $\mathcal{L}(v)$ is as follows. We initialize a string w_0 to be the empty string. Then we construct a sequence of strings $w_1, \ldots, w_k$, where each element w_i is of the form $w_{i-1} \cdot b_i$ with $b_i \in \Sigma$, and we define the result of the sample procedure to be w_k. In other words, we sample a string w of $\mathcal{L}(v)$ by building a *prefix* of the sample, symbol by symbol. To ensure that w is an element of $\mathcal{L}(v)$ chosen uniformly, we also consider a sequence of sets $P_0, \ldots, P_k$ constructed as follows. The first set is $P_0 = \{v\}$. Then at the i-th step, we consider the set of vertices that can be reached from P_i by reading letter b, namely, for $b \in \Sigma$ define:

$$P_{i,b} = \{u \mid \exists u' \in P_i : (u', b, u) \in E\}.$$

Further, define the set $P_{i,\lambda} = \{u \in P_i \mid \text{level}(u) = 0\}$, namely, all the sink vertices in P_i. Similar to the previous section, the sets $\{P_{i,b} \mid b \in \Sigma\}$ and $P_{i,\lambda}$ induce a

```
Sample(P, w, φ)
    1. Compute $P_b = \{u \mid \exists u' \in P : (u', b, u) \in E\}$ for every $b \in \Sigma$ and $P_\lambda = \{u \in P \mid \text{level}(u) = 0\}$.
    2. Choose $b \in \{0, 1, \lambda\}$ with probability:
           $p_b = \frac{N(P_b)}{N(P_\lambda) + \sum_{a \in \Sigma} N(P_a)}$.
    3. If $b = \lambda$, then with probability $\varphi$ return $w$, otherwise return fail.
    4. Else return Sample($P_b, w \cdot b, \frac{\varphi}{p_b}$).
```

Figure 4: Sampling algorithm for the FPRAS.

partition of the set $\mathcal{L}(P_i)$:

$$\mathcal{L}(P_i) \quad = \quad \mathcal{L}(P_{i,\lambda}) \uplus \biguplus_{b \in \Sigma} \{b\} \cdot \mathcal{L}(P_{i,b}).$$

Notice that $\mathcal{L}(P_{i,\lambda}) = \{\lambda\}$ if $P_{i,\lambda} \neq \varnothing$, and $\mathcal{L}(P_{i,\lambda}) = \varnothing$ otherwise. Therefore, our sampling algorithm estimates the size $N(P_{i,b})$ of $\mathcal{L}(P_{i,b})$ for $b \in \Sigma \cup \{\lambda\}$, and chooses b with probability proportional to its size, namely, $N(P_{i,b})/\big(N(P_{i,\lambda}) + \sum_{a \in \Sigma} N(P_{i,a})\big)$. First, assume we choose $b \in \Sigma$. Then we define $b_{i+1} = b$, append the symbol to obtain $w_{i+1} = w_i \cdot b_{i+1}$, define P_{i+1} as $P_{i,b}$, and continue with the recursion on w_{i+1} and P_{i+1}. Hence, we have that P_{i+1} is the set of vertices such that there exists a path labeled by w_{i+1} that connects v with some vertex of P_{i+1}. Instead, if we choose $b = \lambda$, then we stop the procedure at this step, let say the k-th step, and w_k is the candidate output string. Since there could be an error in estimating the sizes of the partitions, it may be the case that some string were chosen with slightly larger probability than others. To remedy this and obtain a perfectly uniform sampler, at every step of the algorithm we store the probability with which we chose a partition. Thus at the end, we have computed exactly the probability φ with which we sampled the string w_k. We can then reject this sample with probability proportional to φ, which gives a perfect sampler. As long as no string is too much more likely than another to be sampled, the probability of rejection will be a constant, and we can simply run our sampler $O(\log(\frac{1}{\mu}))$-times to get a sample with probability $1 - \mu$ for every $\mu > 0$.

This procedure is given in Figure 4. We call it with the initial parameters **Sample**($\{v\}$, λ, φ_0), corresponding to the goal of sampling a uniform element of $\mathcal{L}(P) = \mathcal{L}(v)$. Here, φ_0 is a value that we will later choose. Notice that at every step of the sampling algorithm in Figure 4, we have that $|\mathcal{L}(P)|$ is precisely the number of strings in $\mathcal{L}(v)$ which have the prefix w, as $\mathcal{L}(P)$ is the set of strings x such that $w \cdot x \in \mathcal{L}(v)$.

To get some intuition of the sampling algorithm in Figure 4, assume for the moment that we can compute each p_b exactly, namely, $p_b = |\mathcal{L}(P_b)|/|\mathcal{L}(P)|$. Now the probability of choosing a given element $x \in \mathcal{L}(v)$ with $|x| = k$ can be computed as follows. Let $P_0 = \{v\}$, P_1, ..., P_k be the sets obtained by the sequence of recursive calls to **Sample** until it stops. Ignoring for a moment the possibility of returning **fail**, we have that w is the string returned by **Sample**($\{v\}$, λ, φ_0). Thus, the probability we choose x is:

$$\begin{aligned}\mathbf{Pr}(w = x) &= \frac{|\mathcal{L}(P_1)|}{|\mathcal{L}(P_0)|} \cdot \frac{|\mathcal{L}(P_2)|}{|\mathcal{L}(P_1)|} \cdots \frac{|\mathcal{L}(P_k)|}{|\mathcal{L}(P_{k-1})|} \cdot \frac{1}{|\mathcal{L}(P_k)|} \\ &= \frac{1}{|\mathcal{L}(P_0)|}.\end{aligned}$$

Now at the point of return, we also have that $\varphi = \varphi_0/\mathbf{Pr}(w = x)$. Thus, if $\varphi_0/\mathbf{Pr}(w = x) \leq 1$, then the probability that x is output is simply φ_0. The following is then easily obtained:

FACT 1. *Assume that p_b ($b \in \Sigma \cup \{\lambda\}$) in the sampling algorithm in Figure 4 satisfies that*

$$p_b \quad = \quad \frac{|\mathcal{L}(P_b)|}{|\mathcal{L}(P)|}.$$

*If $0 < \varphi_0 \leq 1/|\mathcal{L}(v)|$ and $w \neq$ **fail** is the output of the call **Sample**($\{v\}, \lambda, \varphi_0$), then for every $x \in \mathcal{L}(v)$, it holds*

$$\mathbf{Pr}(w = x) \quad = \quad \varphi_0.$$

*Moreover, the algorithm outputs $w =$ **fail** with probability $1 - |\mathcal{L}(v)| \cdot \varphi_0$.*

This shows that, conditioned on not failing, the above is a uniform sampler. However, Fact 1 was obtained under the strong assumption that each probability p_b can be computed exactly. Hence, in the next result one can prove that with high probability the same result holds if we approximate p_b with $N(P_b)/\big(N(P_\lambda) + \sum_{a \in \Sigma} N(P_a)\big)$ (instead of assuming that $p_b = |\mathcal{L}(P_b)|/|\mathcal{L}(P)|$).

PROPOSITION 2.5. *Assume that condition $\mathcal{C}(\ell - 1)$ holds. If $w \neq$ **fail** is the output of **Sample**($\{v\}, \lambda, \frac{e^{-5}}{N(v)}$), then for every $x \in \mathcal{L}(v)$:*

$$\mathbf{Pr}(w = x) \quad = \quad \frac{e^{-5}}{N(v)}.$$

*Moreover, it outputs **fail** with probability at most $1 - e^{-9}$. Thus, conditioned on not failing, the algorithm returns a uniform sample $x \in \mathcal{L}(v)$.*

It is important to mention that, in order for Proposition 2.5 to be correct, we need that condition $\mathcal{C}(\ell - 1)$ holds. In fact, the sampling procedure uses values $N(P)$ for approximating the real values $|\mathcal{L}(P)|$, which is supported by Proposition 2.2 that shows that $N(P)$ is a good estimate for $|\mathcal{L}(P)|$ if $\mathcal{C}(\ell - 1)$ holds. In the next section, we prove that $\mathcal{C}(m)$ holds with exponentially high probability, which implies that $\mathcal{C}(\ell)$ holds with exponentially high probability for every $\ell \leq m$.

2.4 Bounding the probability of breaking the main assumption

Recall that the input of the problem is a labeled DAG $D = (V, E)$, and that $n = |V|$, $m = \text{level}(V)$ and $\kappa = \lceil \frac{nm}{\varepsilon} \rceil$. As it was previously discussed, the computation of the sketch composed by the estimates $N(v)$ and sets $S(v)$ is subject that the condition $\mathcal{C}(m)$ holds. Therefore, this section is aimed to bound the probability that $\mathcal{C}(\ell)$ is false for each level $\ell \leq m$ and show that, indeed, this probability is exponentially low.

First, assume that we are back to a level $\ell \leq m$, condition $\mathcal{C}(\ell-1)$ holds, and we want to bound the probability that condition $\mathcal{C}(\ell)$ holds. In other words, we want to provide a lower bound for $\mathbf{Pr}(\mathcal{C}(\ell) \mid \mathcal{C}(\ell-1))$, for which we will use Hoeffding's inequality.

Proposition 2.6. (Hoeffding's inequality [16]) *Let $X_1, \ldots, X_t$ be independent random variables bounded by the interval $[0,1]$ such that $\mathbb{E}[X_i] = \mu$. Then for every $\delta > 0$, it holds that*

$$\mathbf{Pr}\left(\left| \frac{1}{t} \sum_{i=1}^{t} X_i - \mu \right| \geq \delta \right) \leq 2e^{-2t\delta^2}.$$

For the first level $\ell = 0$, the condition $\mathcal{C}(0)$ certainly holds. Now assume that we are at some level ℓ, and recall that V_ℓ is the set of all vertices $v \in V$ such that level$(v) \leq \ell$. If $\mathcal{C}(\ell-1)$ holds, then we know by Proposition 2.5 that for each $v \in V_\ell$, it is possible to fill $S(v)$ with $2\kappa^7$ uniform samples of $\mathcal{L}(v)$. Consider an arbitrary subset $P \subseteq V_\ell$, and let $S(v) = \{w_1, \ldots, w_t\}$ be the uniform sample of $\mathcal{L}(v)$ of size $t = 2\kappa^7$. For each w_i, consider the random variable X_i such that $X_i = 1$ if $w_i \in (\mathcal{L}(v) \setminus \bigcup_{u \in P} \mathcal{L}(u))$, and 0 otherwise. Then we have that:

$$\begin{aligned} \mathbb{E}[X_i] &= \frac{|\mathcal{L}(v) \setminus \bigcup_{u \in P} \mathcal{L}(u)|}{|\mathcal{L}(v)|}, \\ \sum_{i=1}^{t} X_i &= |S(v) \setminus \textstyle\bigcup_{u \in P} \mathcal{L}(u)|, \end{aligned}$$

and $t = |S(v)|$. Therefore, by Hoeffding's inequality we infer that:

$$\mathbf{Pr}\left(\left| \frac{|\mathcal{L}(v) \setminus \bigcup_{u \in P} \mathcal{L}(u)|}{|\mathcal{L}(v)|} - \frac{|S(v) \setminus \bigcup_{u \in P} \mathcal{L}(u)|}{|S(v)|} \right| \geq \frac{1}{\kappa^3} \;\middle|\; \mathcal{C}(\ell-1) \right) \leq 2e^{-4\kappa}$$

Notice that in the previous inequality the condition $\mathcal{C}(\ell-1)$ does not change the assumptions of Hoeffding's inequality. We can bound $\mathbf{Pr}(\neg\mathcal{C}(\ell) \mid \mathcal{C}(\ell-1))$ by taking the union bound over all vertices v and all possible subsets $P \subseteq V_\ell$:

$$\mathbf{Pr}\left(\exists v \in V_\ell \;\; \exists P \subseteq V_\ell \;\; \left| \frac{|\mathcal{L}(v) \setminus \bigcup_{u \in P} \mathcal{L}(u)|}{|\mathcal{L}(v)|} - \frac{|S(v) \setminus \bigcup_{u \in P} \mathcal{L}(u)|}{|S(v)|} \right| \geq \frac{1}{\kappa^3} \;\middle|\; \mathcal{C}(\ell-1) \right) \leq$$

$$n2^n \cdot 2e^{-4\kappa} \leq e^{4n} \cdot e^{-4\kappa} \leq e^{2nm} \cdot e^{-4\kappa} \leq e^{-2\kappa}.$$

We conclude that, at level ℓ, the probability $\mathbf{Pr}(\mathcal{C}(\ell) \mid \mathcal{C}(\ell-1)) \geq 1 - e^{-2\kappa}$. Hence, we obtain the following lower bound for the upper level m:

$$\begin{aligned} \mathbf{Pr}(\mathcal{C}(m)) &= \mathbf{Pr}(\bigwedge_{\ell=0}^{m} \mathcal{C}(\ell)) \\ &= \prod_{\ell=1}^{m} \mathbf{Pr}(\mathcal{C}(\ell) \mid \bigwedge_{\ell'=0}^{\ell-1} \mathcal{C}(\ell')) \\ &= \prod_{\ell=1}^{m} \mathbf{Pr}(\mathcal{C}(\ell) \mid \mathcal{C}(\ell-1)) \\ &\geq \prod_{\ell=1}^{m} (1 - e^{-2\kappa}) = (1 - e^{-2\kappa})^m. \end{aligned}$$

Moreover, it is possible to prove that $(1 - e^{-2\kappa})^m \geq 1 - e^{-\kappa}$. Therefore, putting everything together, we obtain the desired lower bound for the probability that condition $\mathcal{C}(m)$ holds.

Proposition 2.7. *The probability that $\mathcal{E}(m)$ holds is bounded below by $1 - e^{-\kappa}$.*

3. COUNTING IN SUCCINCT LABELED DAGS

In this section, we go one step further in abstraction by considering *succinct* labeled DAGs. Intuitively, we can generalize the idea of labeled DAGs by using edges of the form (u, r, v), where r defines a set of labels *succinctly encoded* in some representation. Then we can extend the counting problem #LDAG to succinct labeled DAGs. Interestingly, this problem is the key algorithmic step to provide efficient approximation algorithms for counting problems related to tree automata and conjunctive queries, among other applications. In what follows, we define our notion of succinct representation, extend labeled DAGs to its succinct version, and then present our results for the extension of #LDAG to succinct labeled DAGs.

As in the previous section, fix a finite alphabet Σ containing at least two elements. Then a *succinct set-representation schema* (or succinct set for short) is composed by a (possible infinite) infinite set of representations $\mathcal{R}$. For technical reasons, we assume the existence of a size function $|\cdot|$ such that for every $r \in \mathcal{R}$, the number $|r|$ represents the size (e.g., number of bits) to store r in the underlying computational model (e.g., the RAM model). We use each $r \in \mathcal{R}$ to represent a set of strings over Σ, so we consider a function set such that set$(r) \subseteq \Sigma^*$. Here, each string $w \in$ set(r) must be of polynomial size with respect to r, that is, there exists a polynomial g such that $|w| \leq g(|r|)$ for every $r \in \mathcal{R}$ and $w \in$ set(r) (where $|w|$ is the usual length of a string).

Furthermore, the number of strings represented by r must be at most exponential in the size of r, that is, there exists a polynomial h such that $|\text{set}(r)| \leq 2^{h(|r|)}$ for every $r \in \mathcal{R}$. The last assumption is to ensure that the number of bits to encode the number $|\text{set}(r)|$ will be at most polynomial in $|r|$. Finally, we assume that there exist polynomial-time algorithms for some key problems to handle the representations in $\mathcal{R}$. More precisely, we first assume that membership in each representation $r \in \mathcal{R}$ can be solved in polynomial time; that is, we suppose that there exists an algorithm M that, given $w \in \Sigma^*$ and $r \in \mathcal{R}$, verifies whether $w \in \text{set}(r)$ in polynomial time on $|w|$ and $|r|$. Second, we assume that the number of strings in each representation $r \in \mathcal{R}$ can be efficiently approximated; that is, we suppose that there exists an algorithm C that, given $r \in \mathcal{R}$ and $\varepsilon \in (0,1)$, outputs an ε-approximation of $|\text{set}(r)|$ in polynomial-time on $|r|$ and $\frac{1}{\varepsilon}$. Finally, we assume that the strings of each $r \in \mathcal{R}$ can be almost-uniformly generated; that is, we suppose that there exist an algorithm U that, given $r \in \mathcal{R}$ and $\varepsilon \in (0,1)$, outputs an ε-uniform sample from set(r) (i.e., with probability $(1 \pm \varepsilon)\frac{1}{|\text{set}(r)|}$) in polynomial-time on $|r|$ and $\frac{1}{\varepsilon}$. In summary, a succinct set $\mathcal{S}$ is a tuple

$$\mathcal{S} \;=\; (\mathcal{R}, |\cdot|, \text{set}, M, C, U)$$

satisfying all the aforementioned properties.

Assuming that the underlying alphabet $\Sigma = \{0,1\}$, a simple example of a succinct set is given by the propositional formulas in DNF. In particular, a string $w \in \{0,1\}^*$ represents a truth assignment, and each representation $\alpha \in \mathcal{R}$ is a formula in DNF (encoded as a string over the alphabet $\{0,1\}$). The size $|\alpha|$ of a propositional formula α is defined as the length of the string encoding it. Moreover, set(α) is defined as the set of all truth assignments that satisfy α. In particular, if α mentions n propositional variables, then set(α) is a subset of $\{w \in \Sigma^* \mid |w| = n\}$. Moreover, given that #DNF admits a fully polynomial-time randomized approximation scheme [19], it is possible to verify that the propositional formulas in DNF satisfy all the properties of a succinct set; in particular, there exist polynomial-time algorithms for approximate counting and uniform generation of truth assignments satisfying a formula in DNF.

From now on, assume that $\mathcal{S} = (\mathcal{R}, |\cdot|, \text{set}, M, C, U)$ is a fixed succinct set such that, for every $a \in \Sigma$, there exists $r \in \mathcal{R}$ satisfying that set$(r) = \{a\}$ (later we will explain why this technical condition is needed). The notion of succinct set is introduced to have a succinct way to represent the alphabet of a labeled DAG and the language associated to each of its vertices. Formally, a *succinct labeled DAG* is a tuple $D = (V, E)$ such that V is a finite set of vertices, $E \subseteq V \times \mathcal{R} \times V$ is a finite set of edges labeled by representations from $\mathcal{S}$, and the directed graph $G = (V, \{(u,v) \mid \exists r : (u,r,v) \in E\})$ is acyclic. The size of D is defined as $|D| = |V| + \sum_{(u,r,v)\in E} |r|$. Moreover, each vertex u of D defines a language, denoted by $\mathcal{L}_D(u)$, over the alphabet

$$\bigcup_{(u,r,v)\in E} \text{set}(r).$$

Formally, if u is a sink vertex, then $\mathcal{L}_D(u) = \{\lambda\}$ (recall that λ is the empty string); otherwise, $\mathcal{L}_D(u)$ is recursively defined as:

$$\mathcal{L}_D(u) \;=\; \bigcup_{(u,r,v)\in E} \text{set}(r) \cdot \mathcal{L}_D(v).$$

Notice that the previous definition is based on the definition of the language associated to a node of a (non-succinct) labeled DAG. However, there are some differences between these definitions. First, each edge (u,r,v) from D succinctly encodes a set set(r) that can contain an exponential number of possible symbols. Second, the elements of each set set(r) cannot be directly accessed; for instance, there may not be an algorithm that can enumerate the elements of set(r) with a polynomial-time delay between two output elements. Instead, we only know that three problems can be solved efficiently for these sets: checking whether an element belongs to set(r), generating almost-uniformly at random an element from set(r), and approximating the size of set(r).

In this work, we are interested in the following counting problem for succinct labeled DAGs.

Problem:	#SuccLDAG
Input:	A succinct labeled DAG D and a vertex u of S
Output:	$\lvert\mathcal{L}_D(u)\rvert$

Recall that for every $a \in \Sigma$, there exists $r_a \in \mathcal{R}$ such that set$(r_a) = \{a\}$. Given this condition, we know that an input (D,u) of the counting problem #LDAG can be viewed as an input of #SuccLDAG just by replacing each edge (v,a,v') by (v,r_a,v'). Hence, we have that #SuccLDAG is a generalization of #LDAG, from which it is possible to conclude that #SuccLDAG is a #P-complete problem (the proof of the membership of #SuccLDAG in #P is left as an exercise for the reader). Interestingly, as for the case of #LDAG, it is possible to obtain an efficient approximation algorithm for #SuccLDAG.

Theorem 3.1. #SuccLDAG *admits an FPRAS.*

In the rest of this section, we give an overview of the main difficulties of #SuccLDAG, compared to the case of (non-succinct) labeled DAGs. Given a succinct labeled DAG $D = (V,E)$, we can proceed as in Section 2 to compute an estimate for $|\mathcal{L}_D(u)|$. Namely, for each vertex $v \in V$ such that level$(v) = \ell$ (recall the definition of level(v) from Section 2), we can store a number $N(v)$

and a set $S(v) \subseteq \mathcal{L}_D(v)$ such that, $N(v)$ is a $(1 \pm \kappa^{-2})^\ell$-approximation of $|\mathcal{L}_D(v)|$ and $S(v)$ is a uniform sample from $\mathcal{L}_D(v)$ of polynomial size. Given the estimates of level $\ell - 1$, obtaining $N(v)$ for vertices of level ℓ will be similar than for labeled DAGs. Instead, the central challenge is to design a polynomial time algorithm to sample from the set $\mathcal{L}_D(v)$, allowing the construction of $S(v)$.

To understand the main challenge for sampling from $\mathcal{L}_D(v)$, we can try to proceed as in Section 2. Given a set of vertices $P \subseteq V$, define $\mathcal{L}_D(P) = \bigcup_{v \in P} \mathcal{L}_D(v)$ and:

$$\text{set}(P) = \bigcup_{r \,:\, \exists v \exists v' : (v \in P \,\wedge\, (v,r,v') \in E)} \text{set}(r).$$

In other words, set(P) are all the first symbols of strings in $\mathcal{L}_D(P)$. Then, starting from a vertex v, we can sample one symbol at a time, building a sequence of strings $w_1, \ldots, w_k$, where $w_i = w_{i-1} \cdot b_i$ and b_i is the next symbol. This strategy gives the sequence of sets $P_0, \ldots, P_k$ where $P_0 = \{v\}$, and P_{i+1} is defined from P_i and the next symbol $b \in \text{set}(P_i)$. Specifically, for each symbol $b \in \text{set}(P_i)$, define the set of vertices:

$$P_{i,b} = \{u \mid \exists u' \in P_{i-1} : (u', r, u) \in E \wedge b \in \text{set}(r)\}.$$

Then we can choose b with probability (approximately) $|\mathcal{L}_D(P_{i,b})|/|\mathcal{L}_D(P_i)|$, and continue with the recursion with the set $P_{i+1} = P_{i,b}$.

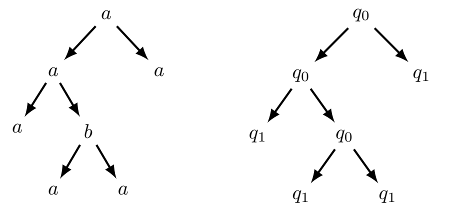
But there is a fundamental problem with this approach in the succinct case: the set of possible symbols set(P_i) is of exponential size with respect to D. Hence, we cannot estimate $|\mathcal{L}(P_{i,b})|$ for each $b \in \text{set}(P_i)$. Instead, our approach will be to approximate the behavior of the above "idealistic" algorithm that estimates all these sizes, by sampling from set(P_i) without explicitly estimating the sampling probabilities. Namely, we can sample a string $w \in \mathcal{L}_D(P)$ for a set $P \subseteq V$, by sampling the next symbol from a "proxy" distribution which is close to the true distribution over set(P), where each b is chosen with probability $|\mathcal{L}_D(P_b)|/|\mathcal{L}_D(P)|$. For building this proxy distribution, we exploit the membership algorithm M, the approximate counting algorithm C, and the almost-uniform sampling algorithm from succinct set $\mathcal{S}$. The reader is refer to [5] for the technical details of our sampling algorithm to build $S(v)$ from $\mathcal{L}_D(v)$.

4. APPLICATIONS OF OUR RESULTS

For the last part of this paper, we present applications of the results in Sections 2 and 3 to automata theory, graph databases, and conjunctive query evaluation.

4.1 Automata theory

Given a non-deterministic finite automaton (NFA) A over an alphabet Σ, let $\mathcal{L}(A)$ be the set of strings in Σ^* that are accepted by A. Then the following is a



(a) Σ-tree t (b) Run of T over t

Figure 5: Execution of a tree automaton T with alphabet $\Sigma = \{a, b\}$, initial state q_0, and transitions (q_1, a), (q_0, a, q_0, q_1), (q_0, a, q_1, q_0) and (q_0, b, q_1, q_1).

fundamental counting problem for automata.

Problem:	#NFA
Input:	An NFA A and a natural number n (given in unary)
Output:	$\lvert\{w \in \mathcal{L}(A) \mid \lvert w\rvert = n\}\rvert$

The problem #NFA is known to be #P-complete, so this problem is not expected to be solvable in polynomial time. However, as a corollary of the results in Section 2, it is possible to obtain an efficient approximation algorithm for this problem.

Proposition 4.1. *#NFA admits an FPRAS.*

The extension of finite automata to trees has proved to be very useful in many database applications [22, 25]. Such an extension is defined as follows. Given an alphabet Σ, a Σ-tree is recursively defined as follows. Each symbol $a \in \Sigma$ is a Σ-tree. If t_1 and t_2 are Σ-trees and $a \in \Sigma$, then $a(t_1, t_2)$ is a Σ-tree. For example, $a(b(a, a(b, b)), a)$ is a tree over the alphabet $\{a, b\}$, which can be depicted as shown in Figure 5a. Moreover, the number of nodes of a Σ-tree t, denoted by $|t|$, is recursively defined as: $|a| = 1$ and $|a(t_1, t_2)| = 1 + |t_1| + |t_2|$, for each $a \in \Sigma$. For example, for the tree t shown in Figure 5a, it holds that $|t| = 7$.

A (top-down) tree automaton T over an alphabet Σ is a tuple (Q, Σ, Δ, q_0) such that Q is a finite set of states, $\Delta \subseteq (Q \times \Sigma) \cup (Q \times \Sigma \times Q \times Q)$ is the transition relation, and $q_0 \in Q$ is the initial state. Given a Σ-tree t, a run of T over t is a Q-tree ρ satisfying the following conditions. Given $q \in Q$, denote by T_q the tree automaton obtained from T by putting q as the initial state, that is, $T_q = (Q, \Sigma, \Delta, q)$. If $t = a$, where $a \in \Sigma$, then ρ must be the Q-tree q_0 and (q_0, a) must be a tuple in Δ. Moreover, if $t = a(t_1, t_2)$, where $a \in \Sigma$ and t_1, t_2 are Σ-trees, then ρ must be a Q-tree of the form $q_0(\rho_1, \rho_2)$, where $(q_0, a, q_1, q_2) \in \Delta$ for some $q_1, q_2 \in Q$, ρ_1 is a run of T_{q_1} over t_1, and ρ_2 is a run of T_{q_2} over t_2. For example, Figure 5b shows a run of a tree automaton. A Σ-tree t is accepted by tree automaton T if there

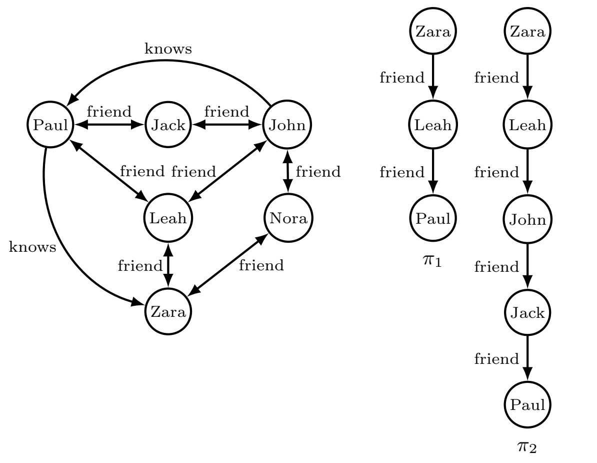


Figure 6: A graph database, and two paths π_1 and π_2 in it.

exists a run of T over t. The set of all Σ-trees accepted by T is denoted by $\mathcal{L}(T)$.

As for the case of automata over strings, the following is a fundamental counting problem for tree automata.

Problem:	#TA
Input:	A tree automaton T and a natural number n (given in unary)
Output:	$\lvert\{t \in \mathcal{L}(T) \mid \lvert t\rvert = n\}\rvert$

It is straightforward to prove that #TA is in #P. Then from the #P-hardness of #NFA, it is possible to conclude that #TA is a #P-complete problem, so #TA is not expected to be solvable in polynomial time. However, as a corollary of the results in Section 3, it is possible to obtain an efficient approximation algorithm for this problem.

PROPOSITION 4.2. *#TA admits an FPRAS.*

For the sake of presentation, #TA is defined in this section for binary trees where each node has either no children or two children. However, the previous result can be extended to ranked trees, where each node can have at most k children for a natural number k.

4.2 Graph databases

Given a set Σ of labels, a graph database G is a pair (V, E) where V is a set of vertices and $E \subseteq V \times \Sigma \times V$ is a set of labeled edges. For example, Figure 6 shows a graph database storing data about people and their relationships; in particular, the set of labels for this graph database is {friend, knows}, so that a triple (a, friend, b) indicates that a and b are friends, while a triple (a, knows, b) indicates that a knows b.

Path queries are a fundamental way to retrieve information from graph databases [2, 11]. In its simplest form, a path query Q over a graph database $G = (V, E)$ is a triple (a, r, b), where $a, b \in V$ and r is a regular expression over the set Σ of edge labels for G. An answer to Q over G is a path from a to b whose labels conform to r. Formally, such a path is a sequence $\pi = v_0, p_1, v_1, p_2, \ldots, p_n, v_n$ of vertices and labels such that $(v_i, p_{i+1}, v_{i+1}) \in E$, $a = v_0$ and $b = v_n$. Moreover, π is said to conform to r if the string $p_1 p_2 \cdots p_n$ is in the regular language defined by r. For example, $Q_1 = (\text{Zara}, \text{friend}^*, \text{Paul})$ is a path query over the graph database in Figure 6, for which the paths π_1 and π_2 shown in the same figure are answers. Thus, an answer to Q_1 over the graph database shown in Figure 6 is a path of friends from Zara to Paul. The set of answers of a path query Q over a graph database G is denoted by $Q(G)$. Clearly, $Q(G)$ can be an infinite set, as paths can contain cycles, so there can be an infinite number of them in a graph. For this reason, the length of the paths to be retrieved must also be specified when posing a path query; the length of a path $\pi = v_0, p_1, \ldots, p_n, v_n$, denoted by $|\pi|$, is defined as n. Hence, in the most classical view of the query answering problem in graph databases, a query is either a pair (Q, n) or a pair $(Q, \text{shortest})$ [9], where Q is a path query, n a natural number and shortest is a reserved keyword. Given a graph database G, a path π is an answer to (Q, n) over G if $\pi \in Q(G)$ and $|\pi| = n$. Moreover, a path π is an answer to $(Q, \text{shortest})$ over G if $\pi \in Q(G)$ and $|\pi| = \ell$, where $\ell = \min_{\pi' \in Q(G)} |\pi'|$. For example, assuming that $Q_1 = (\text{Zara}, \text{friend}^*, \text{Paul})$ and G is the graph database shown in Figure 6, the path π_2 in the same figure is an answer to $(Q_1, 4)$ over G, given that $|\pi_2| = 4$. However, π_2 is not an answer to the query $(Q_1, \text{shortest})$ over G since $\min_{\pi \in Q_1(G)} |\pi| = 2$. On the other hand, π_1 is an answer to $(Q_1, \text{shortest})$ over G.

In the case of graph databases and path queries, the following are the fundamental counting query answers problems to be solved:

Problem:	#PATHGD
Input:	A graph database G, a path query Q and a natural number n (given in unary)
Output:	$\lvert\{\pi \in Q(G) \mid \lvert\pi\rvert = n\}\rvert$

Problem:	#SHORTESTPATHGD
Input:	A graph database G and a path query Q
Output:	$\lvert\{\pi \in Q(G) \mid \lvert\pi\rvert = \ell\}\rvert$, where $\ell = \min_{\pi \in Q(G)} \lvert\pi\rvert$

It is possible to prove that both problems #PATHGD and #SHORTESTPATHGD are #P-complete [3, 21], so they are not expected to be solvable in polynomial time. However, by using the results of Section 2, it is possible to prove that both problems admit efficiently approximations.

Proposition 4.3. *Both #PathGD and #ShortestPathGD admit an FPRAS.*

4.3 Acyclic conjunctive queries

Conjunctives queries (CQs) are expressions of the form $Q(\bar{x}) \leftarrow R_1(\bar{y}_1), \ldots, R_n(\bar{y}_n)$ where each R_i is a relational symbol, each $\bar{y}_i$ is a tuple of variables, and $\bar{x}$ is a tuple of output variables with $\bar{x} \subseteq \bar{y}_1 \cup \cdots \cup \bar{y}_n$. Conjunctive queries are the most common class of queries used in database systems, so the computational complexity of the tasks related to their evaluation is a fundamental object of study. Given as input a database instance D and a conjunctive query $Q(\bar{x})$, the *query evaluation problem* is defined as the problem of computing $Q(D) \coloneqq \{\bar{a} \mid D \vDash Q(\bar{a})\}$. Namely, $Q(D)$ is the set of answers $\bar{a}$ to Q over D, where $\bar{a}$ is an assignment of the variables $\bar{x}$ which agrees with the relations R_i. The corresponding *query decision problem* is to verify whether or not $Q(D)$ is empty. It is well known that even the query decision problem is NP-complete for conjunctive queries [6]. Thus, a major focus of research in the area has been to find tractable special cases [29, 7, 13, 15, 14, 10, 12].

Beginning with the work in [29], a fruitful line of research for finding tractable cases for CQs has been to study the *degree of acyclicity* of a CQ. In particular, the *treewidth* $\text{tw}(Q)$ of Q [7, 15], and more generally the *hypertree width* $\text{hw}(Q)$ of Q [14], are two primary measurements of the degree of acyclicity. A class $\mathcal{C}$ of conjunctive queries has bounded treewidth (hypertree width) if $\text{tw}(Q) \leq k$ ($\text{hw}(Q) \leq k$) for every $Q \in \mathcal{C}$, for a fixed constant k. It is known that the query decision problem can be solved in polynomial time for every class $\mathcal{C}$ of CQs with bounded treewidth [7, 15] or bounded hypertree width [14].

In addition to evaluation, counting the number of answers to a conjunctive query is a fundamental problem, in particular because the optimization process of a relational query engine requires, as input, an estimate of the number of answers to a query (without evaluating it). Unfortunately, counting the number of answer of a conjunctive query is more challenging than evaluating it. Specifically, given as input a conjunctive query Q and database D, computing $|Q(D)|$ is #P-complete even when $\text{hw}(Q) = 1$ [23], that is, for so called *acyclic* CQs [29]. However, these facts do not preclude the existence of efficient approximation algorithms for classes of CQs with a bounded degree of acyclicity. In fact, we show in this article how the results in Section 4.1 can be used to construct a fully polynomial-time randomized approximation scheme for every class of CQs with bounded hypertree width. Since $\text{hw}(Q) \leq \text{tw}(Q)$ for every CQ Q [14], this result also includes every class of CQs with bounded treewidth. Specifically, consider the following family of counting problems.

Problem:	#k-HW
Input:	A CQ Q such that $\text{hw}(Q) \leq k$ and a database D
Output:	$\lvert Q(D)\rvert$

Then it is possible to prove that:

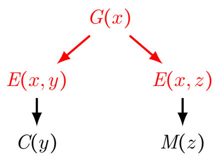
Proposition 4.4. *The problem #k-HW admits an FPRAS for every $k \geq 1$.*

Recall that in Proposition 4.2, we show that the problem #TA admits an FPRAS. In what follows, we give a high-level overview of a reduction to #TA from the counting problem for the class of acyclic conjunctive queries, that is, from the class of CQs Q such that $\text{hw}(Q) = 1$. This reduction can be properly formalized, and it can be extended to each problem #k-HW, so to conclude from Proposition 4.2 that each problem #k-HW admits an FPRAS.

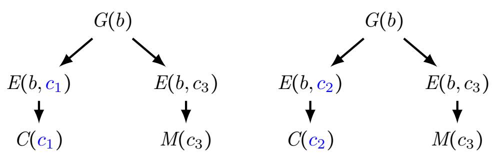
Consider a CQ $Q_1(x) \leftarrow G(x), E(x,y), E(x,z), C(y), M(z)$. This query is said to be acyclic because it can be encoded by a *join tree*, that is, by a tree t where each node is labeled by the relations occurring in the query, and which satisfies that each variable in the query induces a connected subtree of t [29]. In particular, a join tree for $Q_1(x)$ is depicted in Figure 7a, where the connected subtree induced by variable x is marked in red. An acyclic conjunctive query Q can be efficiently evaluated by using a join tree t encoding it [29]; in fact, a tree *witnessing* the fact that $\bar{a} \in Q(D)$ can be constructed in polynomial time. For example, if $D_1 = \{G(a),\ G(b),\ E(a,c_1),\ E(b,c_1),\ E(b,c_2),\ E(b,c_3),\ C(c_1),\ C(c_2),\ M(c_3)\}$, then b is an answer to Q_1 over D_1, which is witnessed by the two trees shown in Figure 7b. Notice that the assignments to variable y that distinguish these two trees are marked in blue.

As shown in Figure 7b, there is no one-to-one correspondence between the answers to an acyclic CQ and their witness trees, so #1-HW cannot be directly solved by counting such trees. However, we observe that in a witness tree t, if only output variables are given actual values and non-output variables are assigned an anonymous symbol $\star$, then there *will* be a one-to-one correspondence between answers to a query and witnesses. Such structures are denoted as *anonymous* trees, an example of which is given in Figure 7c. But how can we specify when an anonymous tree is valid? For example, if t' is the anonymous tree obtained by replacing b by a in Figure 7c, then t' is not a valid anonymous tree, because a is not an answer to Q_1 over D_1. At this point, tree automata come to the rescue, as they can be used to specify the validity of such anonymous trees. In this way, #1-HW can be reduced to #TA.

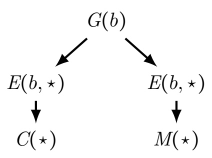
Proposition 4.4 can be extended to the case of unions of conjunctive queries with a bounded degree of acyclicity. More precisely, a union of CQs (UCQ) is an expression of the form $Q(\bar{x}) \leftarrow Q_1(\bar{x}) \vee \cdots \vee Q_m(\bar{x})$, where each



(a) A join tree.



(b) Two witness trees for the answer b.



(c) An anonymous tree for b.

Figure 7: Join, witness, and anonymous trees for a CQ.

$Q_i(\bar{x})$ is a conjunctive query, and the same tuple $\bar{x}$ of output variables is used in the CQs $Q_1(\bar{x})$, …, $Q_m(\bar{x})$. The set of answers of Q over a database D, denoted by $Q(D)$, is defined as $Q(D) = \bigcup_{i=1}^{m} Q_i(D)$. Concerning to our investigation, we are interested in the following family of counting problem for unions of CQs.

Problem:	#k-UHW
Input:	A database D and a union of CQs $Q(\bar{x}) \leftarrow Q_1(\bar{x}) \vee \cdots \vee Q_m(\bar{x})$ such that $\text{hw}(Q_i) \leq k$ for all $i \in \{1,\ldots,m\}$
Output:	$\lvert Q(D)\rvert$

As expected, #k-UHW is #P-complete [23] for every $k \geq 1$. However, #1-UHW remains #P-hard even if we focus on the case of UCQs without existentially quantified variables, as opposed to the case of CQs where #1-HW can be solved in polynomial time if existential quantifiers are not allowed [23]. Nevertheless, by using Proposition 4.4, it is possible to provide the following positive result.

Proposition 4.5. *The problem* #k-UHW *admits an FPRAS for every* $k \geq 1$.

5. REFERENCES

[1] A. Amarilli, F. Capelli, M. Monet, and P. Senellart. Connecting knowledge compilation classes and width parameters. *Theory Comput. Syst.*, 64(5):861–914, 2020.
[2] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):68, 2017.
[3] M. Arenas, S. Conca, and J. Pérez. Counting beyond a yottabyte, or how SPARQL 1.1 property paths will prevent adoption of the standard. In *WWW*, pages 629–638, 2012.
[4] M. Arenas, L. A. Croquevielle, R. Jayaram, and C. Riveros. #NFA Admits an FPRAS: efficient enumeration, counting, and uniform generation for logspace classes. *J. ACM*, 68(6):48:1–48:40, 2021.
[5] M. Arenas, L. A. Croquevielle, R. Jayaram, and C. Riveros. When is approximate counting for conjunctive queries tractable? In *STOC*, pages 1015–1027, 2021.
[6] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.
[7] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. In *ICDT*, pages 56–70, 1997.
[8] A. Darwiche and P. Marquis. A knowledge compilation map. *J. Artif. Intell. Res.*, 17:229–264, 2002.
[9] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, O. van Rest, H. Voigt, D. Vrgoc, M. Wu, and F. Zemke. Graph pattern matching in GQL and SQL/PGQ. In *SIGMOD*, pages 2246–2258, 2022.
[10] J. Flum and M. Grohe. *Parameterized Complexity Theory.* Springer, 2006.
[11] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An evolving query language for property graphs. In *SIGMOD*, pages 1433–1445, 2018.
[12] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: Questions and answers. In *PODS*, pages 57–74, 2016.
[13] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. In *FOCS*, pages 706–715, 1998.
[14] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.
[15] M. Grohe, T. Schwentick, and L. Segoufin. When is the evaluation of conjunctive queries tractable? In *STOC*, pages 657–666, 2001.
[16] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.
[17] M. Jerrum, A. Sinclair, and E. Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *J. ACM*, 51(4):671–697, 2004.
[18] M. R. Jerrum, L. G. Valiant, and V. V. Vazirani. Random generation of combinatorial structures from a uniform distribution. *Theor. Comput. Sci.*, 43:169–188, 1986.
[19] R. M. Karp and M. Luby. Monte-carlo algorithms for enumeration and reliability problems. In *FOCS*, pages 56–64, 1983.
[20] R. M. Karp, M. Luby, and N. Madras. Monte-carlo approximation algorithms for enumeration problems. *J. Algorithms*, 10(3):429–448, 1989.
[21] K. Losemann and W. Martens. The complexity of regular expressions and property paths in SPARQL. *ACM Trans. Database Syst.*, 38(4):24:1–24:39, 2013.
[22] F. Neven. Automata theory for XML researchers. *SIGMOD Rec.*, 31(3):39–46, 2002.
[23] R. Pichler and S. Skritek. Tractable counting of the answers to conjunctive queries. *J. Comput. Syst. Sci.*, 79(6):984–1001, 2013.
[24] J. S. Provan and M. O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM J. Comput.*, 12(4):777–788, 1983.
[25] T. Schwentick. Automata for XML - A survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007.
[26] L. G. Valiant. The complexity of computing the permanent. *Theor. Comput. Sci.*, 8:189–201, 1979.
[27] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.
[28] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982.
[29] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.