A New Class of Lineage Expressions over Probabilistic Databases computable in P-time

Batya Kenig, Avigdor Gal, and Ofer Strichman

Technion, Israel Instutute of Technology, Haifa, Israel

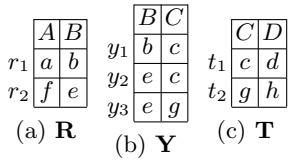
Abstract. We study the problem of query evaluation over tuple-independent probabilistic databases. We define a new characterization of lineage expressions called *disjoint branch acyclic*, and show this class to be computed in P-time. Specifically, this work extends the class of lineage expressions for which evaluation can be performed in PTIME. We achieve this extension with a novel usage of junction trees to compute the probability of these lineage expressions.

1 Introduction

Answering queries over probabilistic databases has drawn much attention in the database community in recent years. A model of tuple-independent (or tuple-level semantics) probabilistic databases was introduced by Cavallo and Pittarelli [5] and was extensively discussed in the literature, *e.g.*, [14, 8]. According to this model, each tuple t is annotated by an existence probability $p_t > 0$, meaning it appears in a possible world with probability p_t, independently of other tuples. This defines a probability distribution over all possible database instances.

Query evaluation over tuple-independent probabilistic databases is $\#P$-hard in general, even for simple conjunctive queries without self-joins [8]. Dalvi and Suciu have introduced a dichotomy classification of queries over tuple-independent probabilistic databases, where any query with a *safe plan* can be computed *extensionally* by extending the query operators to enable an efficient computation of the result's probability [8, 7]. The extensional approach is very efficient, but may be applied to a limited set of queries [19, 8].

As it turns out, even for queries without a safe plan there are database instances for which probabilities could be computed in PTIME. An intensional approach to evaluate queries over tuple-independent probabilistic databases considers both the query and the database instance. The query result is first computed and represented as a Boolean formula, termed a *lineage expression* [2], where each variable represents a tuple (see Table 1) in the database instance. Various inference algorithms can be used to compute the result tuple probabilities, either exactly [22] or approximately [23]. Roy et al. [24] and Sen et al. [25] showed a polynomial time algorithm for recognizing lineage expressions that can be transformed to a read-once form, and computing their probability. Their algorithm is applicable for lineages resulting from conjunctive queries without self joins.



(a) **R** (b) **Y** (c) **T**

Fig. 1: Probabilistic Database with variables

Consider the tuple-independent probabilistic database of Figure 1 and the Boolean conjunctive query

$$Q1():\text{-}R\left(x,y\right),Y\left(y,z\right) \tag{1}$$

$Q1$ is a conjunctive query without self-joins that has a safe plan [8]. Olteanu and Huang [22] showed that the lineage resulting from conjunctive queries without self-joins, that have a safe plan, always have a read-once equivalent. Indeed, the lineage expression of $Q1$ over the database in Fig. 1 is $r_1y_1 + r_2y_2 + r_2y_3$, which has an equivalent read-once form, $r_1y_1 + r_2\left(y_2 + y_3\right)$.
$Q2$ is an example of a query that does not have a safe plan:

$$Q2():\text{-}R\left(x,y\right),Y\left(y,z\right),T\left(z,w\right) \tag{2}$$

The lineage expression of $Q2$ over the database is

$$r_1y_1t_1 + r_2y_2t_1 + r_2y_3t_2 \tag{3}$$

It was shown [24, 17] that Expression 3 does not have an equivalent read-once form.

In this work we introduce a new class of lineage expressions called *disjoint branch acyclic lineage expressions*. Such lineage expressions are defined using restrictions on their respective hypergraph. Going back to Example 1, the lineage expression in Eq. 3 is disjoint branch acyclic, possessing a special structure that can be exploited for efficient computation.

We characterize disjoint branch acyclic lineage expressions and present, as part of the proof of the class computation time, an algorithm to compute the probability of this form in time that is polynomial in the size of the formula.

The rest of the paper is organized as follows: Section 2 introduces background on a specific class of chordal graphs, probability computation using probabilistic graphical models, and hypergraph acyclicity. Lineage acyclicity is presented in Section 3. Section 4 presents the main theorem of this work, proving it by showing an algorithm for probability computation of disjoint branch acyclic lineage expressions. We conclude in Section 8.

2 Preliminaries

At the heart of the proposed method for computing the probability of Boolean lineage expressions lies a graph with a specific structure termed *rooted directed*

path graph. This section introduces this class of graphs and discusses other notions significant to our proposed approach. We present a class of chordal graphs, namely *rooted directed path graphs* (Section 2.1) and discusses probability computation using probabilistic graph models (Section 2.2)). We conclude with the introduction of hypergraph acyclicity (Section 2.3).

A clique C of a graph $G(V, E)$ is a subset of V where every pair of nodes is adjacent. We denote by K_G the set of maximal cliques in G. For a vertex $v \in V$ we denote by K_v the set of maximal cliques in K_G that contain v. We use $T(V, E)$ to denote a tree. A subtree is a connected subgraph of a tree. In particular, a path in a tree can be viewed as a subtree. Whenever a subtree is induced from a subset of nodes $V' \subseteq V$ of a tree $T(V, E)$, we do not explicitly state its set of edges, but rather denote it using $T(V')$.

2.1 Classes of chordal graphs

A *chord* is an edge connecting two non-consecutive nodes in a cycle or path. G is *chordal* or *triangulated* if it does not contain any chordless cycles. Discovering whether G is chordal can be performed in time $O(|V| + |E|)$ [27].

A $P4$ denotes a chordless path with four vertices and three edges. A graph is considered to be $P4$-*free* if it does not contain a $P4$.

An *intersection graph* of a finite family of non-empty sets is obtained by representing each set by a vertex, and connecting two vertices if their corresponding sets intersect. Gavril [15] characterizes the connection between chordal graphs and intersection graphs, as follows.

Theorem 1 ([15]). *Let $G(V, E)$ be an undirected graph. The following statements are equivalent:*

1. *G is chordal.*
2. *There exists a tree $T(K_G)$ such that for every $v \in V$ the subgraph induced by K_v is a subtree $T(K_v)$.*
3. *G is the intersection graph of a family of subtrees of some tree T'.*

$T(K_G)$ is called a *junction tree* possessing the following *running intersection property*: for every pair of cliques $C_1, C_2 \in K_v$ every clique on the path from C_1 to C_2 in $T(K_G)$ belongs to K_v.

Let T' be a rooted directed tree, and consider a group of directed paths in T'. Let G be the intersection graph of directed paths in T'. Then G is a *Rooted Directed Path Graph*, and T' is called the *host tree* of G.

The following property (Theorem 2 [16]) defines a characteristic tree, associated with a rooted directed path graph. This tree is of prime concern in this work.

Theorem 2 ([16]). *A graph $G(V, E)$ is a rooted directed path graph (rdpg) iff there exists a rooted directed tree T_r whose vertex set is K_G, so that for every vertex $v \in V$, $T_r(K_v)$ is a directed path of T_r.*

Constructing the characteristic tree of a rooted directed path graph $G(V)$, if one exists, takes $O(|V|^4)$ [16].[1] The characteristic tree T_r of an rdpg $G(V, E)$ is, in fact, a special form of a junction tree (Definition 1), where every vertex $v \in V$ appears in exactly one branch of T_r. Such a junction tree is known as a *disjoint branch junction tree* (dbjt) [11].

Definition 1. *Let T_r be a junction tree with root r and children $r_1, r_2, ..., r_l$ roots of subtrees $T_{r_1}, ..., T_{r_l}$, respectively. A junction tree T_r is a* dbjt *if:*

1. *T_r contains a single node, r,* i.e., *$|T_r| = 1$, or*
2. *The following two conditions jointly hold: (a) $\forall r_i \neq r_j, C_{r_i} \cap C_{r_j} = \emptyset$; and (b) $\forall T_{r_i} \in \{T_{r_1}, T_{r_2}, ..., T_{r_l}\}$, T_{r_i} recursively complies with the conditions 1 and 2.*

An example of a rooted directed path graph and its corresponding characteristic tree (or dbjt) are presented in Figures 2b and 2c, respectively. Definition 1 characterizes the dbjt properties that enable the efficient computation we show in this work.

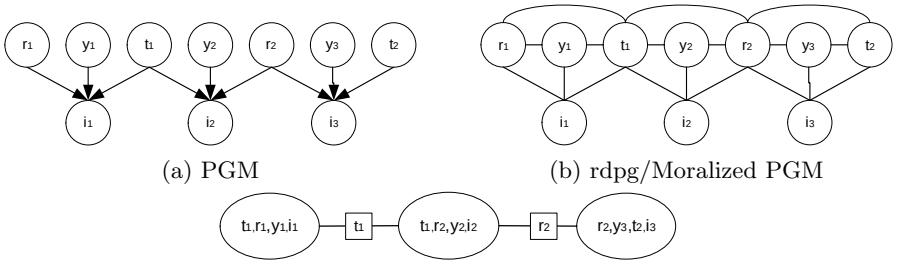
2.2 Probability Computation using Probabilistic Graph Models

Probabilistic Graphical Models (PGMs) refer to a set of approaches for representing and reasoning about large joint probability distributions [21]. A PGM is a graph in which nodes represent random variables and edges represent direct dependencies between them. An example of a directed PGM is given in Figure 2a, representing the lineage expression of Eq. 3.

Inference in PGMs is the task of answering queries over the probability distribution described by the graph and is, in general, #P-complete [21]. One of the well-known inference algorithms is the junction tree algorithm [21]. The algorithm is designed for undirected PGMs in which for every maximal clique C in the PGM, there exists a factor F_C that is a function from the set of assignments of C to the set of non-negative reals. The algorithm consists of two parts, *compilation* and *message passing*. The compilation part includes three steps, namely *moralization, triangulation* and *construction*, as follows. *Moralization*, in the case of a directed PGM, involves connecting all parents of a given node and dropping the direction of edges (*e.g.*, Figure 2b). *Triangulation* adds extra edges to create a chordal graph. The example graph obtained after moralization in Figure 2b is already chordal and therefore no edges need to be added. We note that this example is also an rdpg (see Section 2.1). *Construction* constructs a junction tree over the maximal cliques in the resulting graph $G(V)$. Figure 2c shows the junction tree of the graph in Figure 2b. Note that since the graph is a rdpg, its junction tree is in fact a dbjt.

The message-passing part has two steps. First, for each edge of the tree T a factor, defined over the variables in the intersection, is assigned. The factor

[1] To date, this is the most efficient published recognition algorithm for rooted directed path graphs [6]. There is also an unpublished linear time algorithm [9] for this class of graphs.



(a) PGM (b) rdpg/Moralized PGM

(c) Characteristic tree/dbjt

Fig. 2: PGM, moralization and junction tree

entries are initialized to 1. Then, neighboring nodes C_1, C_2 exchange messages through the factor defined on their intersection F_S, $S = C_1 \cap C_2$. The message from C_1 to C_2 is:

$$\mu_{C_1,C_2}(S) = \sum_{x \in C_1 \setminus S} F_1(C_1)$$

and the message from C_2 to C_1 is:

$$\mu_{C_2,C_1}(S) = \frac{\sum_{x \in C_2 \setminus S} F_2(C_2)}{\mu_{C_1,C_2}(S)}$$

After every pair of adjacent nodes in the junction tree have exchanged messages, each factor holds the marginal of the joint probability distribution of the entire variable set. The message-passing protocol is such that every edge in the tree is processed once in each direction. Therefore, the runtime of the message passing algorithm is $O(N \cdot D^k)$ where N is the number of nodes in the tree, D is the domain of the variables, and k is the size of the largest clique. $k-1$ is referred to as the *width* of the associated PGM and clearly, the inference algorithm is exponential in the PGM's width so bounded width implies tractability in graphical models. PGMs may have several different triangulations, affecting the size of the largest clique in the graph. The smallest width that can be obtained for a PGM is its *treewidth*, where the treewidth of a chordal graph is simply its width. Finding an optimal triangulation is known to be NP-complete. However, in this work we introduce a method to compute the probability of a family of lineage expressions in time that is polynomial in their treewidth.

2.3 Hypergraphs and Acyclicity

A *hypergraph* $H = (V, E)$ is a generalization of a graph where V is the set of nodes and the set of edges E is a set of non-empty subsets of V. Edges in a hypergraph are termed *hyperedges*. The *primal graph* $G(H) = (V, E^G)$ corresponding to a hypergraph H is the graph whose vertices are those of H and whose edges are the set of all pairs of nodes that occur together in some hyperedge of H

$(E^G = \{(u,v) : \{u,v\} \subseteq V, \exists e \in E, \{u,v\} \subseteq e\})$. A hypergraph H is *conformal* if every clique in its primal graph $G(H)$ is contained in a hyperedge of H. Acyclicity in a hypergraph is defined as follows.

Definition 2 (acyclicity [1]). *A hypergraph H is acyclic (or α-acyclic) if H is conformal and its primal graph $G(H)$ is chordal.*

Beeri et. al [1] showed that a hypergraph is acyclic iff it has a junction tree. Duris [11] also showed that for a restricted form of acyclic hypergraphs (called γ-acyclic) there exists a dbjt rooted at every node. An algorithm that constructs a dbjt in time $O(|V|^2)$ for γ-acyclic hypergraphs was also introduced there.

3 Disjoint Branch Acyclic Lineage (DBAL) Expressions

We now introduce a class of Boolean lineage expressions, connecting it to rooted directed path graphs. Let $f(V)$ denote a lineage expression of a set of literals V, resulting from a query q, as derived by the query engine. Lineage expressions of conjunctive queries are monotone formulas, where all literals are positive and only conjunctions and disjunctions are used. An *implicant* $p \subseteq V$ of f is a set of literals such that whenever they are true, f is true as well. An implicant of f is called a *prime implicant* if it cannot be reduced. We denote by f_{IDNF} f's DNF form containing only prime implicants. f_{IDNF} can be modeled as a hypergraph $H_f(V,E)$, where each literal corresponds to a node in the graph and each prime implicant corresponds to a hyperedge.

f_{IDNF} is not always available, and expanding f to its DNF form may result in an exponentially larger formula. Therefore, we now propose the construction of an alternative graph $G(f)$, built over $f(V)$. The set of literals V is the set of nodes of $G(f)$ and two nodes are connected iff they belong to a common prime implicant. $G(f)$ is exactly H_f's primal graph, *i.e.*, $G(H_f) = G(f)$. For a restricted set of queries, $G(f)$ can be built directly from f by using a method proposed by Roy et al. [24]. For the reasons stated above, we shall use $G(f)$ instead of $G(H_f)$ hereafter. We say that f is *conformal* if every maximal clique in $G(f)$ is contained in a prime implicant of f.

Definition 1 (Lineage expression acyclicity). *A lineage expression f is acyclic if $G(f)$ is chordal and f is conformal.*

Definition 2 (Disjoint Branch Acyclic Lineage Expressions). *A lineage expression f is a Disjoint Branch Acyclic Lineage Expression or DBAL if f is acyclic and $G(f)$ is a rooted directed path graph.*

Following the discussion in Section 2.1, DBAL expressions have a dbjt. The lineage expression in Eq. 3 (Section 1) is an example of a DBAL. Its corresponding dbjt is presented in Figure 2c.

4 DBAL Expression Probability Computation

In this section we prove that the probability of disjoint branch acyclic lineage (DBAL) expressions over tuple independent probabilistic databases can be computed in PTIME.

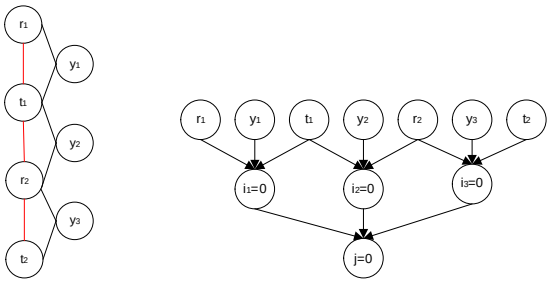
Theorem 1. *Let $f(V)$ be a DBAL expression. The probability $Pr(f = 1)$ can be computed in time $O(nk^2)$ where $n = |V|$ and k is the size of the largest clique in $G(f)$.*

At the heart of the proof is an algorithm for computing the probability of DBAL expressions in time that is quadratic in the size of the treewidth. This solution is unique since, to the best of our knowledge, it is the first time an algorithm that runs in time polynomial (quadratic) of the treewidth, as opposed to exponential, is introduced in the context of tuple independent probabilistic databases.

Section 4.1 introduces an example that will be used to demonstrate the algorithm and Section 4.2 discusses factor representation in our setting. Finally, Section 4.3 details the algorithm and presents lemmas 1 and 2 that argue for the correctness of the algorithm and its complexity, respectively, which together proves Theorem 1 above.

4.1 Illustrating Example

We first motivate and explain the algorithm approach using a simple example.



(a) Primal graph

(b) Bayesian net

Fig. 3: Illustration for Example 1

Example 1. Consider query $Q2() : -R(x, y), Y(y, z), T(z, w)$, presented earlier over the instance in Table 1. The lineage of the query is $j = r_1y_1t_1 + r_2y_2t_1 + r_2y_3t_2$. The primal graph corresponding to the query is given in Figure 3a. It

is easy to see that this lineage expression is not read-once since it has a P4: (r_1, t_1, r_2, t_2). Let us denote by $i_1 = r_1 y_1 t_1$, $i_2 = r_2 y_2 t_1$, and $i_3 = r_2 y_3 t_2$. We are ultimately interested in calculating the probability: $Pr(j = 1) = 1 - Pr(j = 0)$. If $j = 0$ then we know that $i_1 = i_2 = i_3 = 0$. These values can be seen as introduction of evidence in a Bayesian network, illustrated in Figure 3b. After moralization (see Section 2.2), the network is chordal and conformal and therefore has a junction tree depicted in Figure 4. In the Junction-Tree algorithm, any node may be selected as root. For this example let us select node $\{r_2, y_3, t_2\}$ as the root. □

In the classic junction tree algorithm, where factors are represented in tabular form, each entry in the factor table represents a single assignment, leading to a representation that is exponential in the number of variables in the table. We present linear sized factors, where each entry represents multiple assignments. See, for example, Figure 4, where asterisks represent wildcard assignments. Here, the number of entries in each factor is exactly the node's cardinality. The message passing is illustrated in Figure 4, and will be demonstrated in detail in Section 4.3. After the completion of the algorithm, the root node contains the marginal probability of its entries (see Section 2.2). Therefore, all the entries of the root node's factor are added in order to obtain the required probability.

4.2 Factor Representation and Projection

Hereinafter we shall use a tabular notation to represent a factor, where columns represent random variables, and rows correspond to a set of mutual exclusive value assignments. The notation introduced below is illustrated in Example 2. Given a factor over the set of random variables $X = (X_1, X_2, ..., X_n)$ we denote by $F_X[j, k]$ (or simply $F[j, k]$ whenever the variable set is clear from the context) the value of X_k in the jth entry. X_k's value may be the wildcard,'*', indicating that it can be either 0 or 1. The assignments represented by the jth entry are denoted by $F[j]$ and their overall probability is denoted by $Pr(F[j])$. Let $X' \subset X$ be a subset of the variables of factor F, we denote by $F[j, X']$ the values of variables X' in the jth entry of the factor. Finally, given an assignment $X = x$, we denote by $Pr(F[x])$ the probability corresponding to this entry in factor F.

Example 2. Consider the factor F in Table 2, representing the joint distribution of independent boolean random variables X_1, X_2, X_3. Using our notation, $F[2, 3] = *$ and $F[2] = [1, 0, *]$. Also, $Pr(F[3]) = p_{X_1} \cdot p_{\overline{X_3}}$ and $F[3, \{X_1, X_3\}] = [1, 0]$. Finally, $Pr(F[\{X_1 = 1, X_2 = 0, X_3 = *\}]) = p_{X_1}$. □

Each maximal clique in the primal graph of a DBAL expression corresponds to exactly one prime implicant of the lineage's IDNF form. As a result, each node in the corresponding junction tree contains a factor that represents a single DNF prime implicant of the lineage. We will refer to these as *DNF factors*. For each variable X in the expression we define a *base factor*, F^b_X. Base factors contain exactly two entries, with values $0, 1$ and their appropriate probabilities $p_{\overline{X}}, p_X$,

respectively. Each base factor is assigned to exactly one node in the junction tree.

Consider some DNF prime implicant d, containing k literals, $d = X_1 \cdot X_2 \cdot ... \cdot X_k$. The probability of $d = 0$ is computed as follows:

$$Pr(d=0) = Pr(X_1 = 0) + Pr(X_1 = 1, X_2 = 0) + ... + Pr(X_1 = 1, ..., X_{k-1} = 1, X_k = 0). \tag{4}$$

The k summands in Eq. 4 create a mutually exclusive and exhaustive set of configurations.

For illustration, consider Table 1 over $X_1, ..., X_k$. The "Pr" values are initialized to 1.

The asterisks in the table represent wildcard assignments, as follows:

X_1	X_2	...	...	X_k	Pr
0	*	*	*	*	1
1	0	*	*	*	1
1	1	0	*	*	1
1	...	...	...	*	1
1	1	1	...	0	1

Table 1: Factor Table

$$\begin{aligned}
&Pr(X_1 = 1, ..., X_{i-1} = 1, X_i = 0, X_{i+1} = *, ..., X_k = *) = \\
&\sum_{x_{i+1},...,x_k \in \{0,1\}} Pr(X_1 = 1, ...X_{i-1} = 1, X_i = 0, X_{i+1} = x_{i+1}, ..., X_k = x_k) = \\
&\sum_{x_{i+1},...,x_k \in \{0,1\}} Pr(X_{i+1} = x_{i+1}, ..., X_k = x_k | X_1 = 1, ..., X_{i-1} = 1, X_i = 0) \cdot Pr(X_1 = 1, ..., X_{i-1} = 1, X_i = 0) \\
&= Pr(X_1 = 1) \cdot ... \cdot Pr(X_{i-1} = 1) \cdot Pr(X_i = 0) \cdot \sum_{x_{i+1},...,x_k \in \{0,1\}} Pr(X_{i+1} = x_{i+1}, ..., X_k = x_k)
\end{aligned} \tag{5}$$

using the tuple independence assumption in the transition from the third to the fourth line of the equation. Informally, once we know that $X_j = 0$, then the implicant's value is false regardless of the values of its other literals.

At the beginning of the algorithm, the values in the "Pr" column of the factors depend on the assignment of the base factors to the nodes in the tree. For example, a factor over variables X_1, X_2, X_3 at the beginning of the algorithm is given in Table 2. For the sake of illustration, we assume that the base factor $F^b_{X_2}$ is assigned to a different node (DNF factor).

The proposed algorithm is actually a series of projections (defined below) over the linear-sized factors of the junction tree. In the general message passing algorithm [21], in which each entry in the factor represents a single configuration of the variables (and therefore the size of the factor is exponential in the number of variables), the probabilities of the entries with common values in the projected variables are simply added. This is not the case for the linear sized factors used in our setting. Definition 1 formalizes this notion of projection in our setting, and Example 3 demonstrates it.

X_1	X_2	X_3	Pr
0	*	*	$p_{\overline{X_1}}$
1	0	*	p_{X_1}
1	1	0	$p_{X_1} \cdot p_{\overline{X_3}}$

Table 2: Factor Table with partial base factors

Definition 1 (factor projection). *Let $F_{X \cup X'}$ be a factor over variables $X \cup X'$ where $X = \{X_1, ..., X_m\}$ and $X' = \{X'_1, ..., X'_l\}$. The projection of F over*

the variables in X, denoted $F_X = \prod_X F_{X \cup X'}$, is a new factor containing only variables X. The probability column in F_X is computed as follows:

$$Pr(F_X[j]) = \sum_{i \in [1,|X \cup X'|]: F_{X \cup X'}[i,X] = F_X[j]} Pr(F_{X \cup X'}[i])$$

The projection $\prod_X F_{X \cup X'}$ may be applied to $F_{X \cup X'}$ under the following conditions:

1. *The variables X', projected out of the factor $F_{X \cup X'}$, appear after (referring to column order) the variables X.*
2. *The base factors corresponding to the variables X' are included in factor $F_{X \cup X'}$ before the projection operation can be applied.*

Example 3. Consider Table 2 and the factor F_{X_1,X_2,X_3} over the variable set $\{X_1, X_2, X_3\}$. We start by projecting out the variable X_3. Condition 1 of Definition 1 is satisfied. As for Condition 2, X_3's probability, p_{X_3}, is already available in the factor, and therefore

$$F_{X_1,X_2} = \prod_{X_1,X_2} F_{X_1,X_2,X_3} = \begin{array}{cc|c} X_1 & X_2 & Pr \\ \hline 0 & * & p_{\overline{X_1}} \\ 1 & 0 & p_{X_1} \\ 1 & 1 & p_{X_1} \cdot p_{\overline{X_3}} \end{array}$$

Projecting out X_2 from F_{X_1,X_2} requires multiplying in X_2's base factor to satisfy Condition 2. Therefore,

$$F_{X_1} = \prod_{X_1} F_{X_1,X_2} = \begin{array}{c|c} X_1 & Pr \\ \hline 0 & p_{\overline{X_1}} \\ 1 & p_{X_1}(p_{\overline{X_2}} + p_{X_2} \cdot p_{\overline{X_3}}) \end{array}$$

□

4.3 Algorithm Description

Let C_i denote the set of variables in node i of the junction tree, $|C_i|$ its cardinality, and F_i its factor. In this section we use the factor notation defined in Section 4.2. B_r denotes the set of variables in node r, for which base factors have been assigned, i.e., $B_r = \{X : X \in C_r,\ F_X^b \textit{ is assigned to } r\}$. We denote by $children(i)$ and $p(i)$ the children and parent of node i in the junction tree, respectively. A message between node i and node j, $\mu_{i,j}(C_i \cap C_j)$ is a factor over the intersection of the two nodes. The number of entries in $\mu_{i,j}(C_i \cap C_j)$ is $|C_i \cap C_j| + 1$ (including the entry containing all ones).

The algorithm uses a partial order $\preccurlyeq$ over the variables in the junction tree T. We denote by $vars(\preccurlyeq)$ the set of variables over which $\preccurlyeq$ is defined.

The pseudocode of the algorithm over linear sized factors is given in algorithms 1 and 2. After the initial call to Algorithm 2 (Line 1 of Algorithm 1), the

Algorithm 1: Message Passing: Initial Call

Input: dbjt (see Definition 1) $T_{r'}$ with root r' corresponding to a lineage expression f.
Output: $Pr(f = 0)$

1: Call Algorithm 2 with parameters: $T_{r'}$ and $\preccurlyeq \leftarrow \emptyset$.
2: **Return** $\sum_{j=1}^{|C_{r'}|} Pr(F_{r'}[j])$.

algorithm performs a series of recursive calls to update the probabilities in the node factors of the junction tree.

Each message from a node i to its parent, $p(i)$, is a projection on factor F_i over the variables $C_i \cap C_{p(i)}$. According to the definition of projection (Definition 1), variables $C_i \cap C_{p(i)}$ should appear before $C_i \setminus C_{p(i)}$ in the factor table representation. Therefore, lines 1-5 of Algorithm 2 define an order over the variables in the root node r that was given as a parameter (C_r), such that projection over variables $C_r \cap C_{p(r)}$ is made possible. In Example 1, Figure 4, the root node contains ordered variables $\{r_2, y_3, t_2\}$. In the factor for the child node with variables $\{t_1, r_2, y_2\}$, r_2 appears before t_1 and y_2 because the message between this node and its parent is over variable r_2. Likewise, in the factor with variables $\{t_1, r_1, y_1\}$, t_1 appears before r_1 and y_1. The order is updated in line 5.

Lines 6-10 initialize a factor for node r based on the ordering $\preccurlyeq$ that was updated in lines 1-5, and according to the base factors assigned to this node. In Example 1 (Figure 4), the base factors for variables y_3 and t_2 are assigned to the root node, while the base factor for r_2 is assigned to the middle node (with variables $\{t_1, r_2, y_2\}$).

Lines 11-21 initiate a recursive call on the children of r. In Line 13, the messages from all children of r are collected. Each one of the messages, $\mu_{i,r}(C_{r_i} \cap C_r)$, received by the node in line 13 contains $|C_{r_i} \cap C_r| + 1$ entries, which form an exhaustive and mutual exclusive set of configurations. For example, consider the message $\mu_{1,2}(t_1)$ from node $\{t_1, r_1, y_1\}$ to node $\{t_1, r_2, y_2\}$ (Figure 4).

As in the case of projection, in order to add the probabilities of the entries in the messages, the appropriate base factors need to be multiplied in before the addition can take place. For example, in Figure 4, the base factor for t_1, $F^b_{t_1}$, is not part of the factor of node $\{t_1, r_1, y_1\}$, and therefore not part of the original message, $\mu_{1,2}(t_1)$, (containing only entries where $t_1 = 0$ and $t_1 = 1$). However, in order to augment the factor with the entry $t_1 = *$, the values in the probability column of the entries corresponding to $t_1 = 0$ and $t_1 = 1$ need to be added. In order for the resulting probability to be correct, the probabilities of the entries corresponding to $t_1 = 0$ and $t_1 = 1$ are multiplied by $p_{\overline{t_1}}$ and p_{t_1} respectively. The entry where $t_1 = *$ in Figure 4 was appended to the message because it will be used by node $\{t_1, r_2, y_2\}$. Such entries are calculated in lines 14-21 of the algorithm. There are exactly $|C_{r_i} \cap C_r|$ such entries added to the message which correspond to partial sums over the entries in the original message $\mu_{i,r}(C_{r_i} \cap C_r)$.

Algorithm 2: Message Passing: Main Procedure

Input: dbjt T_r with root r and a partial order $\preccurlyeq$.
Output: Factor F_r with correct probabilities

1: **if** $r \neq r'$ **then**
2: Define an order $\preccurlyeq_r$ over C_r s.t.: 1. $C_r \cap C_{p(r)}$ appear before $C_r \setminus C_{p(r)}$ 2. The order of variables $C_r \cap vars(\preccurlyeq)$ complies with $\preccurlyeq$.
3: **else**
4: define an arbitrary order over variables C_r
5: Update $\preccurlyeq$ according to the steps above.
{*Initialize node's factor based on* $\preccurlyeq$}
6: Define a linear-sized factor F_r based on $\preccurlyeq$.
7: **for** $j \leftarrow 1$ to $|C_r|$ **do**
8: $Pr(F_r[j]) \leftarrow 1.0$ {initialize factor entries}
9: **for** $X \in B_r$ **do**
10: $Pr(F_r[j]) \leftarrow Pr(F_r[j]) \cdot Pr(X = F_r[j, \{X\}])$
{*apply projection on subtrees*}
11: **for all** $r_i \in children(r)$ **do**
12: Recursively call the algorithm on subtree T_{r_i} with root r_i and (updated) ordering $\preccurlyeq$.
13: $\mu_{i,r}(C_r \cap C_{r_i}) \leftarrow \prod_{C_r \cap C_{r_i}}(F_{r_i})$ [project on the children's factor to get the message]
14: **for** $j \leftarrow 1$ to $|C_r \cap C_{r_i}| + 1$ **do**
15: $M_{i,r}[j] \leftarrow 1.0$ [iterate over entries in the message]
16: **for** $X \in ((C_r \cap C_{r_i}) \setminus B_{r_i})$ **do**
17: $M_{i,r}[j] \leftarrow M_{i,r}[j] \cdot Pr(X = \mu_{i,r}[j, \{X\}])$ $[Pr(X = *) = 1.0]$
18: $prob \leftarrow Pr(\mu_{i,r}[|C_{r_i} \cap C_r| + 1) \cdot M_{i,r}[|C_{r_i} \cap C_r| + 1]$ [initialize $prob$ according to entry [1,1,...,1]]
19: **for** $k \leftarrow |C_{r_i} \cap C_r|$ to 1 **do**
20: $prob \leftarrow prob + Pr(\mu_{i,r}[k]) \cdot M_{i,r}[k]$ [update $prob$]
21: $Pr(\mu_{i,r}[X_1 = 1, ..., X_{k-1} = 1, X_k = *, ..., X_{|C_{r_i} \cap C_r|} = *]) \leftarrow prob$
{*update factor using children's projected factors*}
22: **for** $j \leftarrow 1$ to $|C_r|$ **do**
23: **for all** $r_i \in children(r)$ **do**
24: $Pr(F_r[j]) \leftarrow Pr(F_r[j]) \cdot Pr(\mu_{i,r}[F_r[j, C_{r_i} \cap C_r]])$

Finally, lines 22-24 update F_r according to the messages received from its children.

The correctness of the algorithm for disjoint branch junction trees is given in Lemma 1. The proof is omitted due to space considerations. It is available in the full version of this paper [20].

Lemma 1. *Let $T_{r'}$ be a dbjt with root r', corresponding to lineage expression f. After running Algorithm 1 on $T_{r'}$, $Pr(F_{r'}[j])$, $j \in [1, |C_{r'}|]$ contains the marginal probability corresponding to the configurations represented by the jth entry of this factor.*

Lemma 2. *The complexity of algorithms 1 and 2 on a disjoint branch junction tree of size n is $O(n \cdot k_{MAX}^2)$ where $k_{MAX} = MAX_{i=1..n}|C_i|$.*

Proof. The loop in lines 6-10 is performed in $O(|C_r|^2)$ since $|B_r| \leq |C_r|$. Similarly, the loop in lines 14-17 is performed in $O((|C_r \cap C_{r_i}| + 1)^2)$, but since subsumption cannot occur in the junction tree, $|C_r| > |C_r \cap C_{r_i}|$, we arrive again at runtime of $O(|C_r|^2)$. The loop in lines 19-21 takes time $O(|C_r \cap C_{r_i}|)$.

A node r in the tree receives messages from all of its neighbors, except its parent in the algorithm. Since the children create a partition of a subset of the variables in the node, then the number of children can be at most $|C_r|$. The

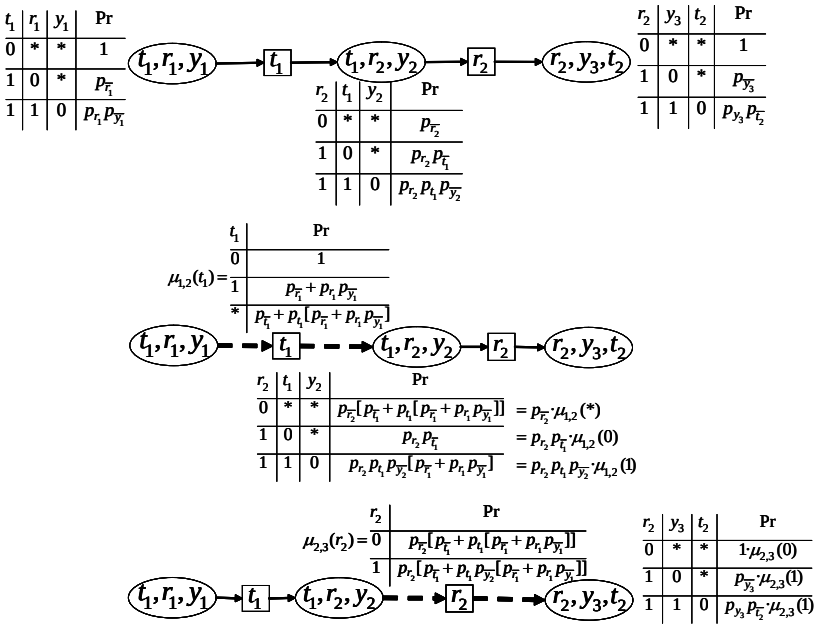


Fig. 4: Message Passing using Alg. 2 over $f = t_1r_1y_1 + t_1r_2y_2 + r_2y_3t_2$

number of entries for which the probability is updated is exactly $|C_r|$, therefore the total runtime is $\sum_{i=1}^{n} O(|C_r|^2) = O(n \cdot (k_{MAX}^2))$.

Algorithms 1 and 2 along with Lemmas 1 and 2 complete the proof for the main theorem of this section, Theorem 1. Overall, we have shown that DBAL expressions, having a dbjt, can be evaluated in polynomial time.

5 Recognition of acyclic and dbal expressions

Let $f(V)$ be a monotone lineage expression resulting from some query. We consider two cases in the recognition procedure, when f_{IDNF} is available and when it is not. We denote $n = |V|$, and m the number of prime implicants in f_{IDNF}. The decision diagram in Figure 5 outlines the steps of the recognition process.

If f_{IDNF} is available then we can build H_f and directly test whether it is acyclic in time $O(m+n)$, by applying the Maximum cardinality Search algorithm restricted to hypergraphs [27]. If H_f is not acyclic, then it cannot have a jt (or dbjt) and the procedure ends. Otherwise, we construct the primal graph $G(f)$ and test whether it has a dbjt by applying the algorithm of Gavril [16]. If f has a dbjt, then we can compute its probability in time $O(nk^2)$ using the algorithm described in Section 4. If not, we can use the classic junction tree algorithm (Sec. 2.2) and compute the probability in time $O(n2^k)$.

The query engine may produce any (not necessarily DNF) lineage expression. Expanding f to its DNF form may result in an exponentially larger formula.

Therefore, Definitions 1 and 2 cannot be applied directly as in the case where f_{IDNF} is available. In order to generate the required primal graph $G(f)$, we apply the method introduced by Roy et al. [24] which enables constructing $G(f)$ in time $O(n^3)$, without having to go through its IDNF representation. This method is applicable for lineages resulting from CQ^- and a restricted set of UCQ^-. The primal graph construction method, as well as the set of queries applicable for it are described in Sec. 6. In order to test whether f is acyclic (Definition 1), we need to check that it is conformal and that $G(f)$ is chordal. Testing for conformality will require validating that every maximal clique in $G(f)$ corresponds to a prime implicant in f. This is a simple procedure deferred to the appendix (Sec. B).

In both cases, if f is acyclic and $G(f)$ is a rooted directed path graph [16] then a dbjt can be constructed for the lineage expression. The algorithm presented in Section 4.3 computes f's probability in time that is polynomial in the size of the formula $O(n \cdot k^2)$.

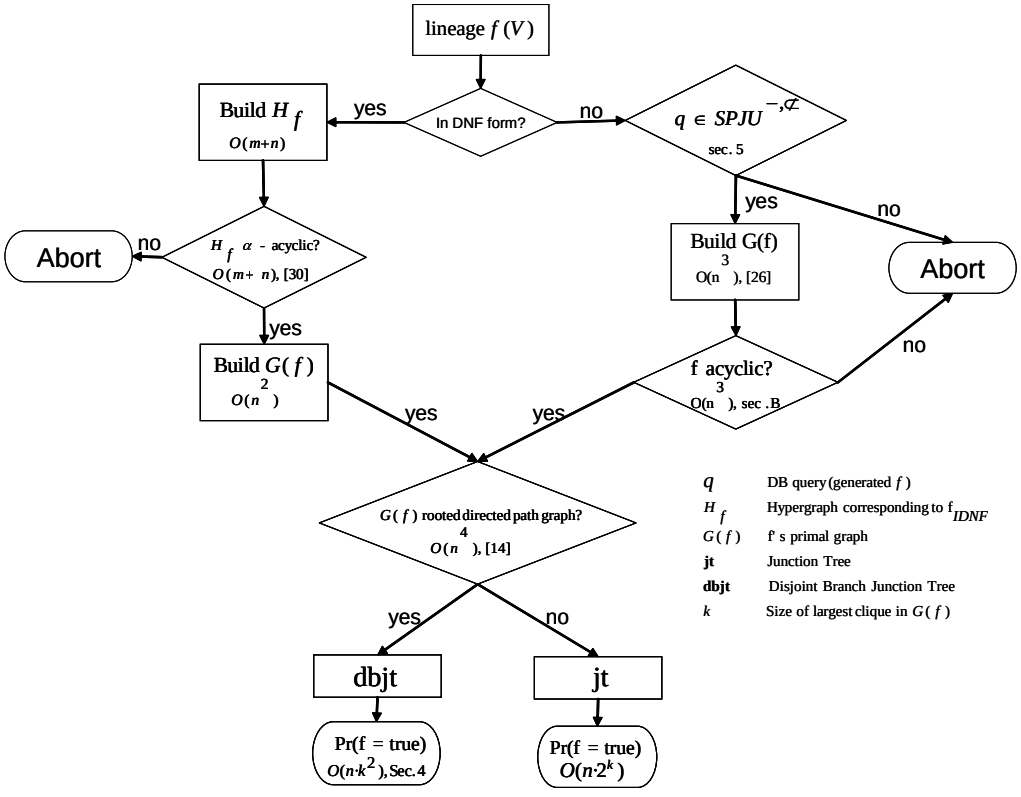


Fig. 5: Probability calculation for f in IDNF form

6 Primal graph construction

General lineage expressions that result from the query engine are not necessarily given in irredundant DNF form. Expanding a general lineage expression to its DNF form may cause an exponential blowup in the size of the expression. Roy et al. [24] introduces a method which enables to bypass this transformation, by constructing f's primal graph without having to go through its IDNF representation. This method operates directly on f and completes in time that is polynomial in the size of f. The method is applicable in cases where the DNF generated by expanding f using the distributivity rule is actually its $IDNF$. This implies that no absorption (*e.g.*, $rs + rst = rs$) can occur during f's expansion. Lineage expressions resulting from Select Project Join (SPJ) queries without self-joins (SPJ^-) adhere to this condition (Lemma 1 in [24]).

Roy et al.[24] insinuate that this method can also be applied to unions of conjunctive queries without self joins, provided that the unions are performed last. However, this class of queries does not exclude the possibility of absorption occurring in the resulting lineage. For example, given the schema, instance and query below, the resulting lineage is $f = rs + rst = rs$.

$$R(A, C), S(B, C, D), T(D, E)$$
$$R(r : [1, c]), S(s : [2, c]), T(t : [c, d])$$
$$Q = (R \bowtie_{R.A<S.B} S) \cup (R \bowtie_C S \bowtie_D T)$$

To avoid absorption we require that for every pair of SPJ^- being unified, the set of relations in one is not a subset of the other, and that the unions are performed last. We denote this set of queries as $SPJU^{-\not\subset}$.

To conclude, if f is a lineage expression resulting from an $SPJU^{-\not\subset}$ we can apply the method proposed in [24] to build a primal graph $G(f)$ in time that is polynomial in $|f|$. Once $G(f)$ is acquired, the remaining steps are identical to those presented in Figure 5.

7 Acyclic and and other lineage expressions

In this section we examine the relationships between acyclic (mainly γ-acyclic) and other classes, namely read-once and OBDD, of lineage expressions.

γ-acyclic lineage expressions (Def. 1) have a dbjt rooted at any one of its nodes [11]. We will use this definition in order to prove the relationship between read-once lineage expressions and those that have a dbjt.

Definition 1 (γ-acyclicity [12]). *A hypergraph is γ-acyclic if it has no pair (E, F) of incomparable, non-disjoint hyperedges such that in the hypergraph that results by removing $E \cap F$ from every hyperedge, what is left of E is connected to what is left of F.*

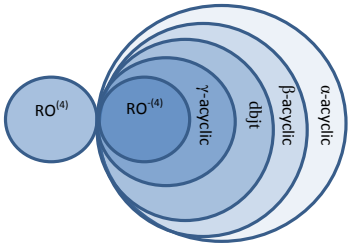


Fig. 6: Read-once and acyclic lineage expressions

Figure 6 illustrates the relationships between γ-acyclic, dbjt and read-once lineage expressions. Example 1 provides a lineage expression that is γ-acyclic but not read-once (Eq. 3). As it turns out, there are also read-once expressions that are do not have a dbjt. As an example, consider the following read-once lineage expression:

$$f = ab + bc + cd + da = (a + c)(b + d)$$

This expression is read-once but does not have a dbjt because it has a chordless cycle of length 4 in its primal graph (denoted as $RO^{(4)}$). Actually, in this case, f is not even α-acyclic because its primal graph is not chordal, and therefore it also does not have a jt. Theorem 1 states that such read-once lineage expressions are precisely those that do not have a dbjt. Theorem 1 shows that any read-once lineage expression whose primal graph does not contain chordless cycles is also γ-acyclic, and therefore has a dbjt (rooted at every node [11]). We summarize the relationship between dbjt and read-once lineage expressions in Table 3.

Theorem 1. *Let f be a read-once lineage expression. f is γ-acyclic iff its primal graph $G(f)$ does* not *contain chordless cycles.*

We note that the co-occurrence graph of read-once expressions cannot contain chordless cycles whose length is larger than 4 because that would mean that they have a $P4$.

Proof. ($\Rightarrow$) If f is γ-acyclic, then it is also α-acyclic, therefore its primal graph cannot contain chordless cycles, specifically not those of length 4.
($\Leftarrow$) Let H_f denote the hypergraph corresponding to f's IDNF. Since no term in f's IDNF can subsume another, there is no subsumption relationship between hyperedges in H_f. We denote by $G(f)$ its primal graph.

Assume, by contradiction, that f is γ-cyclic. According to Definition 1, this means that H_f contains two hyperedges E, F s.t. $E \cap F = Q \neq \emptyset$ and removing Q from all hyperedges in H_f results in E and F staying connected in the resulting hypergraph H^*.

Consider the *shortest* path between E and F in H^*, where $x_E \in E \setminus F$, $x_F \in F \setminus E$

$$E, x_E, S_1, x_1, S_2, x_2, ..., S_m, x_F, F$$

We illustrate this path in Figure 7a using boldface edges. Such a path results in a chordless path containing $m+1$ nodes in f's primal graph, *e.g.*, Figure 7b. Since f is read-once, $G(f)$ does not contain a $P4$, therefore $m \leq 2$. We separate into cases:

Case 1: $m = 1$, Figure 7c. In this case there is some hyperedge S such that $S \supseteq \{x_E, x_F\}$, and thus the edge (x_E, x_F) appears in $G(f)$. There must be some node $x'_E \in E$ such that the edge $(x'_E, x_F) \notin G(f)$. Otherwise, x_F is connected to every node in E, forming a clique containing nodes $E \cup \{x_F\}$. Since f is read-once and therefore conformal [17], there must be a hyperedge S' such that $S' \supseteq (E \cup \{x_F\}) \supset E$, contradicting our assumption that H_f was built based on f's IDNF form. Using the same reasoning, there must be a node $x'_F \in F$ such that the edge $(x_E, x'_F) \notin G(f)$, otherwise the hyperedge F is contained in some other hyperedge, again bringing us to a contradiction. This means that the following chordless path appears in $G(f)$: x'_E, x_E, x_F, x'_F. If there is an edge between x'_E and x'_F, then we arrive at a contradiction that $G(f)$ does not contain chordless cycles. If no such edge exists, then the path is a $P4$, contradicting the fact that f is read-once.

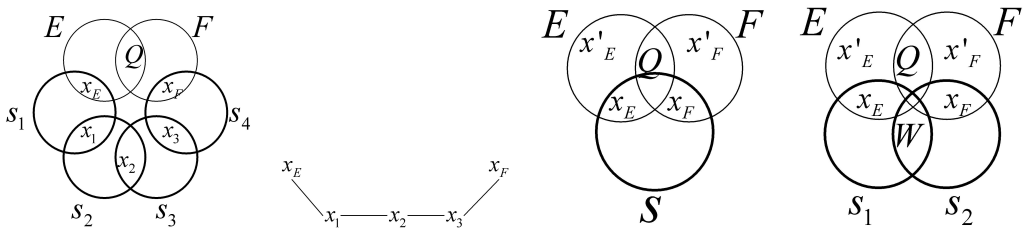
Case 2: $m = 2$, Figure 7d. Here, there exist two distinct hyperedges S_1, S_2, $S_1 \cap S_2 = W \neq \emptyset$ on the path from E to F. This implies that x_E and x_F do not belong to a common hyperedge and due to f's conformality, no edge can exist between them in $G(f)$. If there exist two nodes, $q \in Q$ and $w \in W$ such that $(q, w) \notin G(f)$, then the following cycle is chordless: (q, x_E, w, x_F, q), contradicting our assumption that $G(f)$ does not contain chordless cycles. Otherwise, every two nodes in Q and W share an edge in $G(f)$.

There must be some pair of nodes $x'_E \in E$ and $w \in W$ such that the edge $(x'_E, w) \notin G(f)$. Otherwise, all nodes in E and W are connected, forming a clique containing nodes $E \cup W$. Since f is read-once and therefore conformal [17], there must be a hyperedge S' such that $S' \supseteq (E \cup W) \supset E$, contradicting our assumption that H_f was built based on f's IDNF form. Therefore, since $(x'_E, w) \notin G(f)$, and $(x_E, x_F) \notin G(f)$ then the following path is chordless x'_E, x_E, w, x_F. If there is an edge between x'_E and x_F in $G(f)$, then the cycle x'_E, x_E, w, x_F, x'_E is chordless bringing us to a contradiction that $G(f)$ does not contain chordless cycles. Otherwise, the path x'_E, x_E, w, x_F is a $P4$ contradicting the fact that f is read-once.

dbjt	**read-once**	**Example**
+	+	$abc + bde = b(ac + de)$
+	-	$abc + bde + dfg$
-	+	$ab + bc + cd + da = (a + c)(b + d)$
-	-	$ab + bc + cd + de + ae$

Table 3: Read-only and γ-acyclic relation examples

An Ordered Binary Decision Diagram (OBDD) [4] is a representation of a Boolean formula as a rooted DAG where each internal node is labeled with a



(a) Example path in H^* (b) Path in $G(H_f)$ (c) Case 1, $m = 1$ (d) Case 2, $m = 2$

Fig. 7: Paths between hyperedges E and F

Boolean variable with two outgoing edges, labeled 0 and 1, and has two leaves, labeled 0 and 1 as well. Every path from the root to a leaf visits the variables in the same order. A Boolean function's probability can be calculated in time that is linear in the size of the OBDD. The size of the OBDD for a formula f with n variables depends exponentially on its pathwidth, q, $O(n \cdot 2^q)$ [13]. The pathwidth is determined by the order of the variables, but in general, finding an ordering which produces the optimal OBDD is NP-hard [3], and its approximation counterpart is NP-hard as well [26]. Jha and Suciu [18] introduce the concept of *expression treewidth*, which is the smallest treewidth of any expression DAG that represents a given lineage expression. They show that if a lineage expression has a bounded expression treewidth then it also has a polynomial size OBDD. Specifically, if the expression width is k then they prove that it has an OBDD whose width is at most $2^{(k+1)2^{k+1}}$. Research on read-once queries [17, 24, 25] showed how to find such DAG expressions for read-once lineages. Furthermore, read-once expressions have an expression treewidth of 1. However, finding the expression DAG that minimizes the treewidth is, in general, an NP-hard problem [18].

8 Conclusions

We have presented *disjoint branch acyclic lineage expressions*, a new class of lineage expressions of queries over tuple independent probabilistic databases, and shown that probability computation over this class can be done in low polynomial data complexity. In future work, we intend to search for other characterizations, based on the rooted directed path graph, which may be applicable to this problem. We believe that the proposed algorithm may be applicable in other settings that enable factors of sub-exponential size. An obvious extension is where the lineage expression is given in irredundant CNF form. Adapting the proposed algorithm to such settings is another research direction worth pursuing.

Acknowledgments

The work was carried out in and partially supported by the Technion–Microsoft Electronic Commerce research center.

References

1. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30:479–513, July 1983.
2. O. Benjelloun, A. Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB Journal*, 17(2):243–264, 2008.
3. B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-Complete. *IEEE Trans. Comput.*, 45(9):993–1002, Sept. 1996.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, Aug. 1986.
5. R. Cavallo and M. Pittarelli. The theory of probabilistic databases. In *VLDB*, pages 71–81, 1987.
6. S. Chaplick. *Path Graphs and PR-trees*. PhD thesis, Charles University, Prague, Jan. 2012.
7. N. Dalvi, K. Schnaitter, and D. Suciu. Computing query probability with incidence algebras. In *PODS*, pages 203–214. ACM, 2010.
8. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB Journal*, 16:523–544, October 2007.
9. P. F. Dietz. *Intersection Graph Algorithms*. PhD thesis, Cornell University, Aug. 1984.
10. D. Duris. Some characterizations of gamma and beta-acyclicity of hypergraphs. Technical report, University Paris Diderot, 2008.
11. D. Duris. Some characterizations of γ and β-acyclicity of hypergraphs. *Inf. Process. Lett.*, 112(16):617–620, 2012.
12. R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30:514–550, 1983.
13. A. Ferrara, G. Pan, and M. Y. Vardi. Treewidth in verification: Local vs. global. In *LPAR 2005*, pages 489–503, 2005.
14. N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems*, 15:32–66, 1994.
15. F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *Journal of Combinatorial Theory, Series B*, 16(1):47 – 56, 1974.
16. F. Gavril. A recignition algorithm for the intersection graphs of directed paths in directed trees. *Discrete Mathematics*, 13:237 – 249, 1975.
17. M. Golumbic, A. Mintz, and U. Rotics. Factoring and recognition of read-once functions using cographs and normality. In *DAC*, June 2001.
18. A. Jha and D. Suciu. On the tractability of query compilation and bounded treewidth. In *ICDT*, pages 249–261, 2012.
19. A. K. Jha and D. Suciu. Knowledge compilation meets database theory: compiling queries to decision diagrams. In *ICDT*, pages 162–173, 2011.
20. B. Kenig, A. Gal, and O. Strichman. Answering queries with acyclic lineage expressions over probabilistic databases. Technical Report IE/IS-2012-04, Technion – Israel Institute of Technology, June 2012. http://ie.technion.ac.il/tech_reports/1347292525_GammaAcycBoolExps.pdf.

21. D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, August 2009.
22. D. Olteanu and J. Huang. Using OBDDs for efficient query evaluation on probabilistic databases. *Scalable Uncertainty Management*, pages 326–340, 2008.
23. D. Olteanu, J. Huang, and C. Koch. Approximate confidence computation in probabilistic databases. In *ICDE*, pages 145–156, 2010.
24. S. Roy, V. Perduca, and V. Tannen. Faster query answering in probabilistic databases using read-once functions. In *ICDT*, pages 232–243, 2011.
25. P. Sen, A. Deshpande, and L. Getoor. Read-once functions and query evaluation in probabilistic databases. *PVLDB*, 3(1):1068–1079, 2010.
26. D. Sieling. The nonapproximability of obdd minimization. *Information and Computation*, 172:103–138, 1998.
27. R. E. Tarjan and M. Yannakakis. Addendum: Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 14(1):254–255, 1985.

A Proofs

We prove the equivalence between Definition 1 and the definition of Duris: "A junction tree T of hypergraph H is a *dbjt* if the hyperedges of H belonging to different branches of T are disjoint" [10].

Theorem 1. *Definition 1 and* dbjts are equivalent.

Proof. We recall that each node of the junction tree corresponds to a set of variables from one or more hyperedges of hypergraph H. A dbjt requires that nodes on different branches of the tree are disjoint.

Let T_r be a dbjt according to [10]. Therefore, for each node A in T_r with children $A_1, ..., A_m$, $Var(A_i) \cap Var(A_j) = \emptyset$, for each $i, j \in [1, m], i \neq j$. Therefore complying to Definition 1.

Now, let T_r comply to Definition 1. Let us assume, by contradiction, that it is not a dbjt, i.e, the same variable Z appears in two nodes A, B on different branches. Since T_r is a junction tree, it complies to the junction tree property, thus Z must appear in the least common ancestor of nodes A, B, $LCA(A, B)$ (such a least common ancestor exists because T_r is a tree). Since A, B are on different branches, then node $LCA(A, B)$ has at least two children with a common variable, Z, in contradiction to Definition 1.

We prove the correctness of the algorithm presented in Section 4. Namely, we prove Lemma 1.

Lemma 1. *Let $T_{r'}$ be a dbjt with root r', corresponding to lineage expression f. After running Algorithm 1 on $T_{r'}$, $Pr(F_{r'}[j])$, $j \in [1, |C_{r'}|]$ contains the marginal probability corresponding to the configurations represented by the jth entry of this factor.*

Proof. First, we note that the ordering defined in lines 1-5 of Algorithm 2 can always be achieved. $vars(\preccurlyeq) \cap C_r$ can only contain variables that belong to an ancestor of r. Since $T_{r'}$ is a tree, r is connected to its ancestors only via $p(r)$. Therefore, due to the junction tree property,

$$vars(\preccurlyeq) \cap C_r \subseteq C_{p(r)} \Rightarrow vars(\preccurlyeq) \cap (C_r \setminus C_{p(r)}) = (vars(\preccurlyeq) \cap C_r) \setminus (vars(\preccurlyeq) \cap C_{p(r)}) = \emptyset$$

Therefore, the set of variables that should appear last in F_r (referring to column order), $C_r \setminus C_{p(r)}$, is not a part of $vars(\preccurlyeq)$, and therefore no contradiction with $\preccurlyeq$ can occur.

The proof is by induction on the size of the tree $T_{r'}$. Let $C_{r'} = \{X_1, X_2, ..., X_l\}$. For the base case ($|T_{r'}| = 1$), the lineage expression, f, contains a single prime implicant, $f = X_1 \cdot ... \cdot X_l$. All base factors are multiplied into the factor $F_{r'}$. Assuming, w.l.g., that the order of the variables is $X_1, X_2, ..., X_l$, the jth entry of the factor ($j \leq l$) is
$F_{r'}[j] = X_1 = 1, X_2 = 1, ..., X_{j-1} = 1, X_j = 0, X_{j+1} = *, ..., X_l = *$

and

$$Pr(F_{r'}[j]) = Pr(X_1 = 1) \cdots Pr(X_{j-1} = 1) \cdot Pr(X_j = 0)$$

which is the correct probability for this entry. Entries in the factor form an exhaustive and mutual exclusive set of configurations, and adding their corresponding probability columns yields the probability $Pr(f = 0)$.

Assume the correctness of the algorithm for trees of size $|T_{r'}| \leq n$ and let $T_{r'}$ be a tree of size $|T_{r'}| = n + 1$. Let us look at the disconnected subtrees created by removing the root r' from $T_{r'}$. We are left with rooted subtrees $T_{r_1}, T_{r_2}, ..., T_{r_l}$ whose sizes are less or equal n. For each pair of subtrees T_{r_i}, T_{r_j}, $\{i, j\} \subseteq [1, l], i \neq j$, $T_{r_i} \cap T_{r_j} = \emptyset$. Otherwise, due to the running intersection property of junction trees, $C_{r_i} \cap C_{r_j} \neq \emptyset$ in contradiction to it being a dbjt.

By the induction hypothesis, the factors $F_{r_1}, ..., F_{r_l}$ contain the correct probabilities in their probability column. The message from child r_i to r' is $\mu_{i,r'}(C_{r_i} \cap C_{r'}) = \prod_{C_{r_i} \cap C_{r'}}(F_{r_i})$. Due to the induction hypothesis and the correctness of the projection operation (see Definition 1), we assume the correctness of the message factors. Since the subtrees are independent, containing disjoint sets of variables, the overall probability can be calculated by simply multiplying the probabilities in the appropriate entries in messages $\mu_{1,r'}, ..., \mu_{l,r'}$ with the probability values in factor $F_{r'}$.

To justify the multiplication operation in line 31, we show that the required information is available in the messages $\mu_{r_1,r'}, ..., \mu_{r_l,r'}$ received by the root r'. Let us look at entry

$$F_{r'}[j] = \{X_1 = 1, ..., X_{j-1} = 1, X_j = 0, X_{j+1} = *, ..., X_l = *\}$$

The message required from child r_i, $\mu_{i,r'}(C_{r_i} \cap C_{r'})$, where $C_{r_i} \cap C_{r'} = \{X_{i_1}, ..., X_{i_m}\}$ are ordered according to their placement in factor F_{r_i}, may take one of the following forms:

- $X_{i_1} = 0, X_{i_2} = *, ..., X_{i_m} = *$: According to the way the factors (and their projections) are structured $\mu_{i,r'}(C_i \cap C_{r'}) = \prod_{C_{r_i} \cap C_{r'}}(F_{r_i})$ contains this entry.
- $X_{i_1} = 1, X_{i_2} = 1, ..., X_{i_j} = 0, X_{i_{j+1}} = *..., X_{i_m} = *$: According to the way the factors (and their projections) are structured $\mu_{i,r'}(C_{r_i} \cap C_{r'}) = \prod_{C_{r_i} \cap C_{r'}}(F_{r_i})$ contains this entry.
- $X_{i_1} = 1, ..., X_{i_{j-1}} = 1, X_{i_j} = *, ..., X_{i_m} = *$: The probability for this entry is calculated in lines 23-27 of Algorithm 2. Its value is

$$\sum_{k=i_j}^{|C_i \cap C_{r'}|+1} Pr(\mu_{i,r'}[k]) \cdot M_{i,r'}[k].$$

This computation is correct based on the induction hypothesis which leads to the correctness of the entries in the factor F_{r_i}, and of the probabilities in the entries of $\mu_{i,r'}$. The mutual exclusiveness of the entries enables adding them (after the multiplication with the missing base factors, $M_{i,r'}[k]$).

The ordering induced by r_i's parent $p(r_i) = r'$ excludes other forms.

To summarize, due to the subtrees independence, the correctness of the messages passed, and the availability of the needed message values, the overall probability is computed correctly.

We now extend the scope of a Lemma 1, used by Roy et al. [24], for the case of $SPJU^{-\not\subset}$.

Lemma 2. *Let f be a lineage expression of a query $Q \in SPJU^{-\not\subset}$. The DNF generated by expanding f using only the distributivity rule is in fact the IDNF of f up to idempotency (*i.e.*, repetition of the same prime implicant is allowed).*

Proof. Let g be the DNF generated from f by applying distributivity repeatedly. Every implicant in g will result from some $SPJ_i^- \in Q$. Due to the absence of self-joins, every implicant in g has exactly one tuple from every relation in the query. Since there is no subsumption relationship between the relations for each pair of queries in Q, then the set of variables in one implicant cannot be a strict subset of the set of variables in the other. Specifically, absorption (*e.g.*, $xy+xyz = xy$) does not apply. Therefore, g is the IDNF of f (up to commutativity and associativity).

The rest of the lemmas introduced by Roy et al. [24] when showing the correctness of the co-occurrence graph computation, either rely on the "no self join" assumption or rely, directly or indirectly, on the equivalent of Lemma 2. Therefore, we can conclude that the correctness of the co-occurrence graph computation, as presented by Roy et al., applies to the $SPJU^{-\not\subset}$ class as well.

B Testing conformality of monotonic boolean expressions given their primal graph

According to Definition 2, H_f is α-acyclic if it is conformal and its primal graph $G(H_f)$ is chordal. Chordality is tested using *maximum cardinality search* [27]. If the primal graph is not chordal, then its corresponding hypergraph cannot be α-acyclic and therefore it cannot have a junction tree, unless triangulated. Since this work focuses on acyclic lineage expressions, we assume chordality.

If f's primal graph is chordal, we need to test its hypergraph H_f for conformality. In our setting this means that every maximal clique in the primal graph $G(H_f)$ corresponds to a prime implicant in f. This would require iterating over all of the maximal cliques in $G(H_f)$, and there can be an exponential number of them in a non-chordal graph. However, once chordality is established, such iteration can be achieved in time $O(n + m)$ where n is the number of vertices and m is the number of edges in the graph [27].

Let $var(f)$ denote the set of variables in f, $n = |var(f)|$, $f(\phi)$ the result of f under assignment $\phi : var(f) \to \{true, false\}$. Due to f's monotonicity, the following two conditions are sufficient to make a set of literals $p \subseteq var(f)$ a prime implicant of f:

1. The assignment
$$\phi_p(t) = \begin{cases} 1 & t \in p \\ 0 & otherwise \end{cases}$$
satisfies f.
2. For every $l \in p$, the assignment
$$\phi_p^l(t) = \begin{cases} 1 & t \in p \setminus \{l\} \\ 0 & otherwise \end{cases}$$
does not satisfy f.

Under the assumption that applying an assignment to a formula is done in time $O(n)$, then the time it takes to verify whether a set of literals p is a prime implicant of f is $O(|p| \cdot |n|)$. Since chordal graphs have a linear number of maximal cliques, then the total time for checking the conformality of the hypergraph corresponding to a formula f with a chordal primal graph is $O(|p| \cdot n^2) = O(n^3)$.