Virtual Joins With Nonexistent Links

Hassan Khosravi, Oliver Schulte*, and Bahareh Bina

Simon Fraser University
School of Computing Science
Burnaby, B.C. V5A 1S6
Canada
{hkhosrav,oschulte,bba18}@cs.sfu.ca

Abstract. Many approaches to multi-relational learning require the computation of database frequencies in the presence of nonexistent links. The corresponding ILP problem is to compute the number of groundings that satisfy a given conjunction of literals in a relational database, where one or more of the literals is negated. We present a fast new dynamic programming algorithm for this problem. The database table joins performed by our algorithm are restricted to joins of tables already existing in the database. Evaluation on three data sets confirms the efficiency of our algorithm; computing frequencies for negated literals added about 15% to the cost of computing frequencies for positive literals only.

1 Introduction

A basic task in algorithmic learning is to compute the frequencies with which various events occur in the data; for example, maximum likelihood estimation of model parameters requires finding the data frequencies. The corresponding problem for ILP rule learning is to compute the frequency with which a conjunction of literals holds in a database (interpretation). The key step is to determine the number of groundings (substitution of variables by constants) for which the conjunction is true in a database. For instance, the frequency of the conjunction $Male(X), Female(Y), Friend(X,Y)$ requires computing the number of constant pairs (a,b) such that the database entails that a is a male friend of female b. In this paper we present an efficient algorithm for computing database frequencies that involve *negative* literals (e.g., $NOT\ Friend(X,Y)$). In typical databases relationship tables are sparse (e.g., fewer tuples satisfy $Friend(X,Y)$ than $NOTFriend(X,Y)$), so the number of groundings for a positive literal is much smaller than the number for its negation. The key point of our algorithm is to avoid explicitly enumerating the satisfying groundings for a negative literal. Instead, it derives the number of satisfying groundings from the positive literal case using the laws of probability. The algorithm can be applied as a subroutine with any ILP system whose rule language allows negated literals (e.g., FOIL [6]). For systems whose rule language allows positive literals only, it is often natural to consider an extension with negative literals; our algorithm addresses a computational obstacle for such extensions.

Another application area is multi-relational data mining and statistical-relational learning (SRL). A relationship table in the database corresponds to a positive literal with more than two variables corresponds to a relationship table in the database. A join of tables corresponds to a conjunction of positive literals. The complement of a database table (i.e., the tuples that are *not* contained in a table) corresponds to a negative literal.

* This work was supported by a Discovery Grant to Schulte from the Natural Sciences and Engineering Council of Canada (NSERC).

Presented at the 19th International Conference on Inductive Logic Programming (ILP'09).

In database terminology, our algorithm computes the size of a join that involves the complements of tables. Our algorithm is a type of *virtual join* that computes the join size without actually materializing the complemented tables. The main SRL application is to determine database frequencies that involve the *absence* of a relationship link. This is important for many recent SRL systems that model uncertainty about the existence of links [2, 4, 1]. The frequency with which a condition holds in a database can be found from the join size corresponding to the condition.

Related Work [2, Sec.5.8.4.2] considered the problem of computing data frequencies conditional on the absence of *a single* relationship from frequencies conditional on its presence. Our dynamic programming algorithm extends the single-table solution to multiple tables with link chains that may involve both present or absent links. [1, Prop.12.4] proves that the problem of computing the number of groundings of an arbitrary first-order clause is #P-complete. The worst-case performance of our virtual join algorithm is exponential in the number of variables in the conjunction. If the number of variables is treated as a constant, our algorithm runs in polynomial time. The algorithm benefits from the fact that the number of variables that a learning algorithm would consider in practice is linear in the number of relationship predicates (tables). In data mining research, the most closely related problem is sometimes referred to as "virtual join": computing frequencies in a table join without materializing the join [7, 8]. Virtual joins have so far been considered for chains of existing relationships only.

Contributions A dynamic programming algorithm is described for computing the number of database frequencies of literal conjunctions that may involve both positive and negative literals. The number of satisfying groundings of such conjunctions can be computed from the database frequency. Using the laws of probability, the algorithm infers frequencies for negative literals with two variables or more from frequencies for the corresponding positive literals; it never enumerates groundings for negative literals. In our experiments, the computational overhead was only about 15% compared to restricting conjunctions to positive relationship literals (standard table joins) only. The datasets discussed are available for ftp download from ftp://ftp.fas.sfu.ca/pub/cs/oschulte, and our code is available upon request.

Paper Organization We define the relational logical vocabulary to which our algorithm applies (conjunctions of literals with variable types). Then, we lay out the design of the virtual join algorithm and give pseudocode. We give a complexity analysis and discuss the cases where the algorithm is efficient. Empirical results on three data sets show very good performance compared to SQL-based approaches that directly count the number of satisfying groundings for negative literals.

2 Notation and Problem Definition

We consider conjunctions of first-order literals with typed variables; Table 1 shows the logical vocabulary. Our algorithm can easily be extended to a language with function symbols. Several ILP systems include type constraints (e.g. [5]). Type constraints also serve to represent integrity constraints of databases, such as foreign key constraints. They are important for determining the number of groundings of a formula because variables of a given type must take values in the domain of the type. Let a list $T_1, T_2, \ldots$ of types be given. Each variable X is assigned a type T, which we denote by X^T. A **term** θ is a constant or a variable; the notation $\boldsymbol{\theta}$ denotes a vector or list of terms. An **atom** is a formula of the form $P(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is of the right arity for P. A **literal** L is an atom (positive literal) or the negation of an atom (negative literal). The formulas we consider are **conjunctions of literals**, or for short just conjunctions. We use the Prolog-style notation $L_1, \ldots, L_n$ for $L_1 \wedge \cdots \wedge L_n$, and vector notation $\mathbf{L}$, $\mathbf{C}$ for conjunctions of literals.

Description	Notation	Examples
Connectives	$\wedge, \neg$	
Constants	$a_1, a_2, \ldots$	$jack, 101, A, B, 1, 2$
Predicate Symbols	P, R and variants	$Student, Course, Registered$
Variables	$X_1, X_2, \ldots$	S_1, S_2, C_1, C_2

Table 1. Definition of Logical Vocabulary. The examples refer to a university database with two entity tables $Student$ and $Course$, and a relationship $Registered$ linking them.

A literal is **ground** if it contains no variables. A **relationship literal** is one with two or more variables. We assume that a ground literal L is either true or false in a database instance (interpretation) $\mathcal{D}$; we write $\mathcal{D} \models L$ if L is true in $\mathcal{D}$, and $\mathcal{D} \not\models L$ otherwise. A database instance $\mathcal{D}$ assigns a domain of constants to each type T, which we denote as $dom_{\mathcal{D}}(T)$. The **domain of a variable** X^T of type T is the same, so $dom_{\mathcal{D}}(X^T) = dom_{\mathcal{D}}(T)$. A **table join** of two or more tables contains the rows in the Cartesian products of the tables whose values match on common fields. A **grounding** γ for a set of variables $X_1, \ldots, X_k$ assigns a constant of the right type to each variable X_i (i.e., $\gamma(X_i) \in dom_{\mathcal{D}}(X_i)$). If γ is a grounding for all variables that occur in a conjunction $\mathbf{C}$, we write $\gamma\mathbf{C}$ for the result of replacing each occurrence of a variable X in $\mathbf{C}$ by the constant $\gamma(X)$. The number of groundings that satisfy a conjunction $\mathbf{C}$ in $\mathcal{D}$ is defined as

$$|\mathbf{C}|_{\mathcal{D}} = |\{\gamma : \mathcal{D} \models \gamma\mathbf{C}\}|$$

where γ is any grounding for the variables in $\mathbf{C}$. Approaches that combine probabilistic concepts with logic (e.g., [6]) often employ a natural probability assignment for conjunctions of literals defined by

$$P_{\mathcal{D}}(\mathbf{C}) = \frac{|\mathbf{C}|_{\mathcal{D}}}{|dom_{\mathcal{D}}(X_1)| \times \cdots \times |dom_{\mathcal{D}}(X_k)|} \tag{1}$$

where $X_1, \ldots, X_k, k > 0$, is the set of variables that occur in $\mathbf{C}$. This probability is the ratio of the number of groundings that satisfy a formula in a database over the number of groundings that are possible given the type constraints. In the next section we present an algorithm for computing the quantity $P_{\mathcal{D}}(\mathbf{C})$ when a database instance $\mathcal{D}$ and a conjunction of literals $\mathbf{C}$ is given as input. The number of groundings of a conjunction can be computed from the probability of the conjunction. Alternatively, our algorithm can be used to compute the number of groundings directly by simply replacing $P_D(\mathbf{C})$ with $|\mathbf{C}|_{\mathcal{D}}$ in the algorithm.

3 The Virtual Join Algorithm

The key constraint that our algorithm seeks to satisfy is to avoid enumerating the number of tuples that satisfy a negative relationship literal. A numerical example illustrates why this is necessary. Consider a university database with 20,000 Students, 1,000 Courses and 2,000 TAs. If each student is registered in 10 courses, the size of a $Registered$ table is 200,000, or in our notation $|Registered(S, C)|_{\mathcal{D}} = 2 \times 10^5$. So the number of complementary student-course pairs is $|\neg Registered|_{\mathcal{D}} = 2 \times 10^7 - 2 \times 10^5$, which is a much larger number that is too big for most database systems. If we consider joins, complemented tables are even more difficult to deal with: suppose that each course has at most 3 TAs. Then $|Registered(S, C), TA(T, C)|_{\mathcal{D}} < 6 \times 10^5$, whereas $|\neg Registered(S, C), \neg TA(T, C)|_{\mathcal{D}}$ is on the order of 4×10^{10}.

Outline and Example. The basic idea behind our algorithm can be described as follows. Let $\mathbf{C}$ be a conjunction of literals and L a literal. From basic probability laws, we have the relation

$$P(\mathbf{C}, \neg L) = P(\mathbf{C}) - P(\mathbf{C}, L). \tag{2}$$

So the computation of a probability involving a negative relationship literal $\neg L$ can be recursively reduced to two computations, one with the positive literal L and one with neither L nor $\neg L$, each of which contains one less negative relationship literal. In the base case, all literals are positive, so the problem is to find $P_{\mathcal{D}}(\mathbf{C})$ for database instance $\mathcal{D}$ where $\mathbf{C}$ contains positive relationship literals only. This can be done with a standard database table join (further optimizations are discussed below). Figure 1 illustrates the recursion in an example with two negative relationship literals. Our program computes the required probabilities in ascending order by the number of relationship literals, so that the results of previous computations can simply be looked up rather than recomputed through a recursive function call. The pseudocode is shown as Algorithm 1.

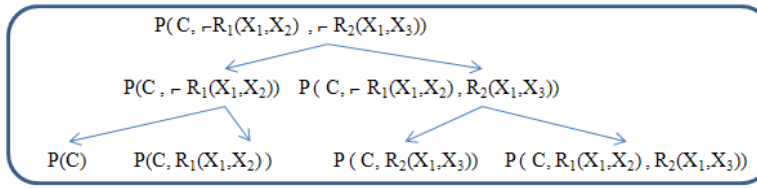

Fig. 1. A example to show how probabilities for conjunctions with negative relationship literals are determined by probabilities for conjunctions with positive literals only.

Complexity Analysis. All database accesses involve only conjunctions of positive literals, so the *virtual join algorithm never materializes the complement of a table*. The worst-case run-time and storage requirements are exponential in m, the number of negative relationship literals. The intention is that the algorithm should be used when the size of the database is much larger than the number of the relationship predicates used in the query, i.e., $|\mathcal{D}| >> m$. In typical cases, we have table sizes on the order of 10^5 while the length of clauses is in the 10s. So the dynamic programming update operations in lines 7–11 can typically be carried out in main memory.The key complexity factor for the table joins is the number of distinct variables that occur in the conjunction. Repeated occurrences of the same variable amount to selection conditions on the variable that actually reduce the size of the join. Joins involving disjoint groups of literals (e.g. $P(X_1, X_2), Q(Y_1, Y_2)$) can be carried out separately, multiplying the results. We expect groups of literals that are linked by shared variables to be feasibly small in light of the following case considerations: (1) Joins that repeatedly involve the same variable (e.g., $P_1(X_1, X_2), P_2(X_1, X_2), P_3(X_1, X_2)$) constrain groundings for different predicates to match on the X_1 field. In this case the tuple ID representation [7] is effective. Also, an entity type E usually participates in few relationships, so we may assume that the number of tables participating in a join that repeatedly involves a variable of type E is a small constant d. (2) The most expensive type of join for our algorithm is a relationship chain like $P_1(X_1, X_2), P_2(X_2, X_3), P_3(X_3, X_4)$. In ILP and SRL applications, relationship chains have small constant length, e.g. $k \approx 3$. One reason for this

Algorithm 1 The Virtual Join Algorithm for computing the number of groundings that satisfy a conjunction of literals in a given database (interpretation).

Input: database instance $\mathcal{D}$; conjunction of literals of the form $(\mathbf{C}, \neg L_1, \ldots, \neg L_m)$, with $m > 0$ negative relationship literals L_i, where $\mathbf{C}$ contains only positive relationship literals.
Output: The database probability $P_{\mathcal{D}}(\mathbf{C}, \neg L_1, \ldots, \neg L_m)$ defined in Equation (1). This is the number of groundings that satisfy $\mathbf{C}, \neg L_1, \ldots, \neg L_m$, divided by the maximum number of possible groundings.

for all subsets of literals $\mathbf{L} \subseteq \{L_1, \ldots, L_m\}$ **do**
  Find $P_{\mathcal{D}}(\mathbf{C}, \mathbf{L})$ using a standard table join.
end for
if $m = 1$ **then**
  Exit and return $P_{\mathcal{D}}(\mathbf{C}, \neg L_1) := P_{\mathcal{D}}(\mathbf{C}) - P_{\mathcal{D}}(\mathbf{C}, L_1)$.
end if
for $i = 1$ to $m - 1$ **do**
  for all subsets of literals $\mathbf{L} \subseteq \{L_{i+1}, \ldots, L_m\}$ **do**
    Assign $P_{\mathcal{D}}(\mathbf{C}, \neg L_1, \ldots, \neg L_{i-1}, \neg L_i, \mathbf{L}) :=$
    $P_{\mathcal{D}}(\mathbf{C}, \neg L_1, \ldots, \neg L_{i-1}, \mathbf{L}) - P_{\mathcal{D}}(\mathbf{C}, \neg L_1, \ldots, \neg L_{i-1}, L_i, \mathbf{L})$.
  end for
end for
Return $P_{\mathcal{D}}(\mathbf{C}, \neg L_1, \ldots, \neg L_m) := P_{\mathcal{D}}(\mathbf{C}, \neg L_1, \ldots, \neg L_{m-1}) - P_{\mathcal{D}}(\mathbf{C}, \neg L_1, \ldots, L_{m-1})$

is the cost of searching through the space of chains. Another is that relationship chains longer than about 3 are difficult for users to understand. It can be shown that any join of tables can be partitioned into groups with disjoint variables each of size at most $k \cdot d$, so when these parameters are small constants, the required table joins can be broken down into small independent joins.

4 Empirical Evaluation

We describe results for three relational datasets. All experiments were done on a QUAD CPU Q6700 with a 2.66GHz CPU and 8GB of RAM.

Methods. We compared the dynamic programming algorithm described as Algorithm 1, abbreviated DP, with two SQL-based methods. (1) SQL-AR: Naively use joins to build a table that enumerates all groundings. (2) SQL-JT: Apply the given selection conditions first and then carry out table joins.

Data Sets. We manually created a small dataset for a university. The entity tables contain 38 students, 10 courses, and 6 professors. The *Registered* table has 92 rows and the RA table has 25 rows. The second dataset is the MovieLens dataset from the UC Irvine machine learning repository. It contains two entity tables: $User$ with 941 tuples and $Item$ with 1,682 tuples, and one relationship table $Rated$ with 80,000 ratings. The third dataset is a modified version of the financial dataset from the PKDD 1999 cup. We have two entity tables: $Client$ with 5,369 tuples and $Account$ with 4,500 tuples. Two relationship tables, $CreditCard$ with 5,369 tuples and $Disposition$ with 892 tuples relate a client with an account.

Results. For each dataset, we first added all available relationship literals, then randomly selected predicates for selection conditions that correspond to descriptive attributes (e.g., age), and considered all combinations of value assignments to the predicates to form literals. Each combination of the relationship literals with selection conditions defines a test conjunction. In the University database, we enumerated all groundings and checked the results of the DP algorithm directly to confirm the correctness

of our implementation. For each conjunction examined, the number of groundings is exactly the same as those computed by the algorithm. Because the dataset is small, the runtime of all three methods are very similar. In the MovieLens database, we selected age and $gender$ from the User table, rating from the $Rated$ table, and randomly 5 of the genres from the item table as predicates for the input of the algorithms. An illustrative conjunction is $Rated(U, I), rating(U, I) = 4, age(U) = 20, gender(U) = M, genre(I) = action$. Both SQL-AR and SQL-JT are clearly slower than our DP algorithm. The DP algorithm spends most of its runtime on queries involving positive literals for which it uses SQL queries on join tables. The additional computation cost with negative literals is only 15% (67 sec). On the Financial Dataset, we selected 8 attributes in total covering all four tables. The naive SQL-AR method was unable to return any results. The SQL-JT method did not terminate for negative literals. In contrast, the DP algorithm had no problem with both positive and negative literals. The increase in run time for conjunctions with negative literals was again only about 15%.

Data Set	# groundings	Positive literals only				Positive + negative literals			
		# conjunctions	SQL-AR	SQL-JT	DP	# conjunctions	SQL-AR	SQL-JT	DP
University	2,280	144	233	239	239	240	388	396	490
MovieLens	1,582,762	960	562	451	451	1,152	955	1,564	518
Financial	24,160,500	1,620	NA	465	465	3,240	NA	NA	540

Table 2. Total runtimes in seconds for our test cases.

5 Conclusion

In this paper we proposed an algorithm for the problem of computing the number of groundings that satisfy a given conjunction of literals in a relational database, where one or more of the literals is negated. A computational bottleneck for analysis with negated literals is that materializing tuples that do *not* satisfy a relationship is generally not possible because there are too many. We have presented a Virtual Join algorithm that solves this challenge by reducing the problem of finding the number of satisfying groundings for conjunctions with negative relationship literals to finding the number of satisfying groundings for conjunctions with positive relationship literals only.

References

1. P. Domingos and M. Richardson. Markov logic: A unifying framework for statistical relational learning. In *Intro. to Statistical Relational Learning* [3], chapter 12, pages 339–367.
2. L. Getoor, N. Friedman, D. Koller, A. Pfeffer, and B. Taskar. Probabilistic relational models. In *Introduction to Statistical Relational Learning* [3], chapter 5, pages 129–173.
3. Lise Getoor and Ben Tasker. *Introduction to statistical relational learning*. MIT Press, 2007.
4. Kristian Kersting and Luc De Raedt. Bayesian logic programming: Theory and tool. In *Introduction to Statistical Relational Learning* [3], chapter 10, pages 291–318.
5. S. Muggleton and J. Firth. Relational rule induction with cprogol4.4. In S. Dzeroski and N. Lavrac, editors, *Relational Data Mining*, chapter 7. Springer Verlag, 2001.
6. J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
7. X. Yin and J. Han. Exploring the power of heuristics and links in multi-relational data mining. In *Foundations of Intelligent Systems (ISMIS)*, pages 17–27, 2008.
8. X. Yin, J. Han, J. Yang, and P. S. Yu. Crossmine: Efficient classification across multiple database relations. In *CB Mining and Inductive Databases*, pages 172–195, 2004.