Efficient and Scalable Induction of Logic Programs using a Deductive Database System

Michel Ferreira - Ricardo Rocha – Tiago Soares - Nuno A. Fonseca
LIACC & DCC-FC, University of Porto

Introduction

Inductive Logic Programming (ILP) tries to derive an intensional representation of data (a theory) from its extensional one, which includes positive and negative examples, as well as facts from a background knowledge. This data is primarily available from relational database management systems (RDBMS), and has to be converted to Prolog facts in order to be used by most ILP systems. Furthermore, the operations involved in ILP execution are also very database oriented, including selections, joins and aggregations. We thus argue that the Prolog implementation of ILP systems can profit from a hybrid execution between a logic system and a relational database system, that can be obtained by using a coupled Deductive Database (DDB) system. This hybrid execution is completely transparent for the Prolog programmer, with the deductive database system abstracting all the Prolog to relational algebra translation.

Deductive Databases

One of the main features of DDB is the representation of data both *extensionally* and *intensionally*.

- Extensional representation

```
edge(10,12).
edge(10,110,15).
edge(12,10).
```

- Intensional representation

```
direct_cycle(A,B):- edge(A,B), edge(B,A).
```

The extensional representation is usually associated with persistent and secondary storage. It mimics the data representation used in relational database tables, using logic facts instead of tuples. The intensional representation uses logic rules to represent further data. Applying some type of resolution, such as Prolog's SLD-resolution, to these rules allows the derivation of data for which no extensional representation is kept on the database. This intensional representation of deductive databases has a limited parallel on the *views* definitions of relational database systems.

Logic System / RDBMS Interface

On a typical coupled deductive database system, the predicates defined extensionally in database relations usually require a directive such as:

```
:- db_import(edge_r,edge,my_conn).
```

This directive is meant to associate a predicate `edge/2` with a relation `edge_r` that is accessible through a connection with the database system named `my_conn`. What this directive commonly does is asserting a clause such as:

```
edge(A,B) :-
   translate(proj(A,B),edge(A,B),Query),
   db_query(my_conn,Query,ResultSet),
   db_row(ResultSet,[A,B]).
```

The `translate/3` predicate, translates a query written in logic to an SQL expression that is understood by database systems. For example, the query goal

```
?- edge(10,B).
```

will generate the call

```
translate(proj(10,B),edge(10,B),Query)
```

exiting with `Query` bound to

```
SELECT 10, A.attr2
FROM edge_r A
WHERE A.attr1=10
```

Views can also be defined using:

```
:- db_view(direct_cycle(A,B),
     (edge(A,B),edge(B,A)),
     my_conn).
```

Which will generate the following SQL query:

```
SELECT A.source, A.dest
FROM edge_r A, edge_r B
WHERE B.source=A.dest AND B.dest=A.source
```

ILP – An Example

Positive Examples

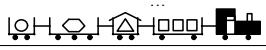
```
eastbound(east1).
eastbound(east2).
eastbound(east3).
eastbound(east4).
eastbound(east5).
```

Negative Examples

```
eastbound(east6).
eastbound(east7).
eastbound(east8).
eastbound(east9).
eastbound(east10).
```

Background Knowledge

```
has_car(east1,car_11).
has_car(east1,car_13).
short(car_12).
closed(car_12).
long(car_13).
open_car(car_13).
shape(car_11,rectangle).
shape(car_13,rectangle).
wheels(car_11,2).
wheels(car_13,3).
load(car_11,rectangle,3).
load(car_13,hexagon,1).
...
has_car(east1,car_12).
has_car(east1,car_14).
short(car_14).
long(car_11).
open_car(car_11).
open_car(car_14).
shape(car_12,rectangle).
shape(car_14,rectangle).
wheels(car_12,2).
wheels(car_14,2).
load(car_12,triangle,1).
load(car_14,circle,1).
...
```



In the Michalski train problem the theory to be found should explain why trains are traveling eastbound. There are five examples of trains known to be traveling eastbound, which constitutes the set of positive examples, and five examples of trains known to be traveling westbound, which constitutes the set of negative examples. All our observations about these trains, such as size, number, position, contents of carriages, etc, constitutes our background knowledge.

To derive a theory with the desired properties, many ILP systems follow some kind of *generate-and-test* approach to traverse the *hypotheses space*. A general ILP system thus spends most of its time evaluating hypotheses, either because the number of examples is large or because testing each example is computationally hard. For instance, a possible sequence of hypotheses (clauses) for the Michalski train problem would be:

```
eastbound(A) :- has_car(A,B).
eastbound(A) :- has_car(A,C).
eastbound(A) :- has_car(A,D).
eastbound(A) :- has_car(A,E).
eastbound(A) :- has_car(A,B), short(B).
eastbound(A) :- has_car(A,B), open_car(B).
eastbound(A) :- has_car(A,B), shape(B,rectangl).
eastbound(A) :- has_car(A,B), wheels(B,2).
eastbound(A) :- has_car(A,B), load(B,circle,1).
...
```

Coverage Computation

To derive a theory with the desired properties, many ILP systems follow some kind of *generate-and-test* approach to traverse the *hypotheses space*. For each of these clauses the ILP algorithm computes its *coverage*, that is, the number of positive and negatives examples that can be deduced from it. If a clause covers all of the positive examples and none of the negative examples, then the ILP system stops. Otherwise, an alternative stop criteria should be used, such as, the number of clauses evaluated, the number of positive examples covered, or time. A simplified algorithm for the coverage computation of a clause is presented next.

```
coverage(Clause,SPos,SNeg) :-
   assert(Clause),
   reset_counter(pos,0),
   (
      positive_examples(X),
      process(Clause,X,GoalP),
      once(GoalP),
      incr_counter(pos),
      fail
   ;
      true
   ),
   counter(pos,SPos),
   reset_counter(neg,0),
   (
      negative_examples(Y),
      process(Clause,Y,GoalN),
      once(GoalN),
      incr_counter(neg),
      fail
   ;
      true
   ),
   counter(neg,SNeg),
   retract(Clause).
```

Coverage Computation with a DDB

We propose, implement and evaluate several approaches of coding ILP coverage computation using DDB technology, with different distributions of work between the logic system and the database system:

- Relation-level approach
- View-level approach
- View-level/Once approach
- Aggregation/View approach

Aggregation/View Approach

We restrict the theories we are inducing to non-recursive theories, so that we can drop the assertion of the current clause to the program code and use the `db_view/3` predicate with the aggregation operation `count/1` on an attribute of the relation holding the positive or negative examples. Also, the view now includes the positive or negative examples relation as a goal co-joined with the goals in the body of the current clause. The join should only test for the existence of one tuple in the body goals for each of the examples. We introduce a predicate `exists/1`, extending the Prolog to SQL compiler, which is translated to a SQL expression involving an existential sub-query.

```
coverage(':-'(H,B),SPos,SNeg,Conn) :-
   process(pos,H,HPos,KPos),
   run_view(count_positive_examples(SPos),
     (SPos is count(KPos,(HPos,exists(B)))),
     Conn),
   process(neg,H,HNeg,KNeg),
   run_view(count_negative_examples(SNeg),
     (SNeg is count(SNeg,(KNeg,exists(B)))),
     Conn).
```

Evaluation

We used the MYDDAS DDB and the April ILP system to evaluate all the strategies:

Problem	Examples	Relations	Tuples
train	10	10	240
p.m8.l27	200	8	321,576
p.m11.l15	200	11	440,000
p.m15.l29	200	15	603,000
p.m21.l18	200	21	844,200

Problems Characterization

Aproach	Problem				
	train	p.m8.l27	p.m11.l15	p.m15.l29	p.m21.l18
April	0.002	15.331	50.447	33,972.225	>1 day
Relation	0.515	35,583.984	>1 day	>1 day	>1 day
View	0.235	n.a	n.a	n.a	n.a
View/Once	0.208	99.837	628.409	2,975.051	33,229.210
Aggregation	0.105	5.330	14.850	251.192	734.800

Coverage Computation Evaluation

The performance results in execution speed for coverage computation are very significant and show a tendency to improve as the size of the problems grows. The size of the problems is exactly our most significant result, as the storage of data-sets in database relations allows an increase of more than 2 orders of magnitude in the size of the problems than can be approached by ILP systems.

Acknowledgements

This work has been partially supported by MYDDAS (POSC/EIA/59154/2004) and by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa POSC. Tiago Soares is funded by FCT PhD grant SFRH/BD/23906/2005.