Computing Cost Estimates for Proof Strategies

Knut Hinkelmann Helge Hintze
DFKI, Postfach 2080, 67608 Kaiserslautern, F.R. Germany
e-mail: hinkelma@dfki.uni-kl.de

Abstract

In this paper we extend work of Treitel and Genesereth for calculating cost estimates for alternative proof methods of logic programs. We consider four methods: (1) forward chaining by semi-naive bottom-up evaluation, (2) goal-directed forward chaining by semi-naive bottom-up evaluation after Generalized Magic-Sets rewriting, (3) backward chaining by OLD resolution, and (4) memoing backward chaining by OLDT resolution. The methods can interact during a proof. After motivating the advantages of each of the proof methods, we show how the effort for the proof can be estimated. The calculation is based on indirect domain knowledge like the number of initial facts and the number of possible values for variables. From this information we can estimate the probability that facts are derived multiple times. An important valuation factor for a proof strategy is whether these duplicates are eliminated. For systematic analysis we distinguish between *in* costs and *out* costs of a rule. The *out* costs correspond to the number of calls of a rule. *In* costs are the costs for proving the premises of a clause. Then we show how the selection of a proof method for one rule influences the effort of other rules. Finally we discuss problems of estimating costs for recursive rules and propose a solution for a restricted case.

Contents

1 Introduction

Besides the traditional depth-first backward-chaining (top-down) strategy for evaluating logic programs there are a number of alternative proof methods. The motivation for considering alternative approaches comes from the following two main drawbacks of the depth-first search underlying most implementations. First, the operational semantics does not correspond to the model-theoretic semantics. The proof of a theorem may not terminate although the theorem is in the model of the program. Second, a large portion of the problem space may be searched redundantly if there are multiple ways in which a subgoal can be derived. A well-known example is the standard specification of Fibonacci numbers.

These disadvantages can be overcome by *memoing* or *caching* queries and their solutions for later use. This led to the development of extension tables [Dietrich, 1987] and the tabulation extension of OLD resolution [Tamaki and Sato, 1986]. Deductive databases contribute another motivation for alternative approaches: the tuple-oriented execution performs a lot of database accesses with small granularity. For the coupling with a database a set-oriented approach would be preferable, e.g. the Query-Subquery approach [Vieille, 1986].

Forward-chaining (bottom-up) evaluation corresponds to a model-generation approach. It is both complete and efficient because it avoids the derivation of duplicates. But if bindings for some argument positions are given in the query, it derives a lot of redundant facts which do not contribute to the proof. Recent developments in bottom-up query evaluation, which are based on program transformations, retain the focusing properties of top-down evaluation ([Ramakrishnan, 1988; Rohmer *et al.*, 1986; Beeri and Ramakrishnan, 1991; Sacca and Zaniolo, 1986]).

Which of these proof methods should be used for a logic programming system? In [Bancilhon and Ramakrishnan, 1988] performance evaluations of several recursive query evaluation algorithms are presented. They measure the computation cost of each method individually over five examples.

Instead of deciding on a proof method once and for all, however, it might be advantageous to have a collection of them in one system. Then the problem is to decide when to use which proof method. We will discuss criteria on which the selection of a proof strategy for one particular query depends. Thereby we will concentrate on criteria that affect the efficient execution. In [Treitel, 1986] algorithms for estimating the computation costs of forward and backward application of rules[1] for LLNR resolution have been presented. For one query both forward and backward chaining inference steps can be mixed. In the following sections of this paper we will extend this approach. The main difference lies in the proof methods that we consider. In our system we do not only have forward and backward reasoning but also goal-directed forward reasoning (after Generalized-Magic-Sets rewriting) and backward reasoning with tabulation. We will also clarify how the cost estimates for one rule have to be propagated to other rules by distinguishing between *in* costs and *out* costs

[1]We use the term *deduction rule* or short *rule* synonymously for *clause*.

of a rule.

2 Proof Methods

As pointed out by [Beeri and Ramakrishnan, 1991] there are two modes of information passing in evaluating a query to a logic program. The first is called *sideway information passing*. By solving a premise predicate variable bindings are obtained which can be passed to another premise in the same rule to restrict the computation for that predicate. In the second mode information is passed to a rule from the query by unification with the head of the rule; it is called *top-down propagation*.

In principle, logic programs may be evaluated by forward or backward chaining. Backward chaining supports both information passing modes but the declarative meaning of a program is contradicted by the termination problem of many implementations.

Example 1 The simple left-recursive ancestor program

```
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

with query

```
?- ancestor(john,A).
```

will not terminate, when evaluated by a depth-first, left-to-right backward-chaining proof method like OLD resolution. To avoid this problem, we have to change the order of the clauses and of the premises in the first clause to get a right-recursive program. It would be nice, however, if the programmer would not need to think about it. Therefore we are looking for complete methods.

By memoizing (sub)goals and their solutions for further use, we can get a complete version of backward-chaining evaluation (e.g. OLDT resolution [Tamaki and Sato, 1986], extension tables [Dietrich, 1987], QSQR [Vieille, 1986], RQA/FQI [Nejdl, 1987]). The principle of memoing is similar for all methods. The backward chaining system has an additional memory, the so-called solution table. At the first occurrence of a query Q, a new entry into the solution table is created. The solution list for Q is still empty. Then Q will be proved by ordinary backward chaining. Every solution of Q will be added to the solution list of Q. If the same query Q occurs multiple times – i.e. there already exists an entry for Q in the solution table – no new proof will be started but the solutions already in the solution list are retrieved. By this approach every query is proved only once. Additionally, some non-terminating loops are avoided: if a query Q occurs a second time but no solutions have been derived the evaluation stops. For a more detailed description of this approach and a proof of its completeness see for example [Tamaki and Sato, 1986].

Evaluating the above left-recursive program of Example 1 with a memoing method will terminate. Since memoing is space-consuming, the incomplete, non-memoing approach can still be useful but only for 'safe', e.g. non-recursive, programs. Besides being complete the memoing methods can also dramatically increase efficiency of programs by reusing previously computed solutions:

Example 2 A well-known example for redundant recomputation is the standard specification of Fibonacci numbers.

```
fib(0,1).
fib(1,1).
fib(N,F) :- N1 = N - 1, fib(N1,F1),
            N2 = N - 2, fib(N2,F2),
            F = F1 + F2.
```

The complexity can be reduced from exponential to linear by memoizing instead of recomputing the values of the first n-1 fibonacci numbers.

Forward chaining by naive or semi-naive evaluation is a *complete* fixpoint procedure [Bancilhon and Ramakrishnan, 1986]. Since pure forward chaining evaluation does not take into account a query, *sideway information passing* is the only information passing mode (see above). To restrict model generation to those ground facts relevant to answer a particular query, the Magic-Sets rewriting technique introduces auxiliary 'magic' predicates to simulate the second (top-down) information passing mode [Bancilhon *et al.*, 1986]. An additional fact – called Magic Seed – carries the bindings of the query; the arguments of the seed fact are exactly the variable bindings of the query. All relevant rules will get an additional premise that can be satisfied by magic facts. Thus, the variable bindings of the query are passed to the body of the applicable rules. The Generalized Magic Sets (GMS) transformation [Beeri and Ramakrishnan, 1991] extends the sideway information passing strategy from base predicates to derived predicates. The rewriting strategy depends on the particular strategy for sideway information passing. A detailed description of GMS rewriting would require too much space. Therefore we will demonstrate it with an example. For more information and an algorithm see [Beeri and Ramakrishnan, 1991; Balbin *et al.*, 1991].

In addition to the introduction of Magic Sets an adorned version of the program is created. The adornment of a predicate depends on the binding pattern of the goal for which it can be called: A predicate p^{bbf} is assumed to be evaluated with the first two arguments bound to a ground term and the third argument being a term with free variables.

Example 3 The simple ancestor program

```
ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
ancestor(X,Y) :- parent(X,Y).
```

with query

```
?- ancestor(john,A).
```

will be rewritten to

```
magic_ancestor_bf(john).
magic_ancestor_bf(Y) :- magic_ancestor_bf(X), parent(X,Y).
ancestor_bf(X,Y)     :- magic_ancestor_bf(X),
                        ancestor_bf(X,Z),
                        parent(Z,Y).
ancestor_bf(X,Y)     :- magic_ancestor_bf(X), parent(X,Y).
```

The initial bindings of the query are given by the seed `magic_ancestor_bf(john)` while the rule for `magic_ancestor_bf` is responsible for the simulation of the top-down propagation of variable bindings. The adornment *bf* indicates that the argument of `magic_ancestor_bf` delivers the bindings for the first argument of `ancestor`. This transformation is data independent. For every set of facts forward chaining of the rewritten program derives all the facts necessary to answer the query `ancestor(john,A)`.

Model generation without rewriting may pay if the query does not restrict the model generation so much. A trivial example would be that the program consists only of facts containing John's ancestors. But also for more realistic programs, if the query does not heavily restrict the model generation, it might be better to renounce rewriting. A preferable approach would be simply to select the relevant rules, execute them by simple bottom-up evaluation, and select the matching facts.

As part of the CoLab knowledge representation system [Boley *et al.*, 1993] we have extended a logic programming language with various alternative proof methods. These are a modification of OLDT resolution [Tamaki and Sato, 1986], where we can explicitly specify the tabulation predicates, and the semi-naive bottom-up evaluation [Bancilhon, 1985] with optional Generalized-Magic-Sets [Beeri and Ramakrishnan, 1991] rewriting. Thus, we distinguish four kinds of proof methods in our system:

Forward Chaining: semi-naive bottom-up evaluation

Goal-Directed Forward Chaining: semi-naive bottom-up evaluation after Generalized-Magic-Sets rewriting

Backward Chaining: top-down proof by OLD resolution

Memoing Backward Chaining: top-down proof by OLD resolution with tabulation (OLDT resolution)

There are interfaces between the forward-chaining and the backward-chaining implementation, such that for proving a theorem any combination of the four proof methods can be applied. The programmer may prejudice (part of) the strategy by explicitly determining the proof methods for individual rules. The control of the combined forward/backward-chaining system is rather complex: Evaluation starts with the bottom-up execution of the (potentially GMS-rewritten) rules that can be triggered by the facts of the program. The top-down reasoner is called for the remaining backward-provable premises of an applied rule (if any). The derived facts of this phase are added to the program. If the query has already been derived in this phase, execution stops. Otherwise, the backward-chaining component is applied reusing forward-derived facts without recomputing them. A detailed description of the system can be found in [Labisch, 1993].

As already mentioned above, the choice of the proof method depends on completeness and efficiency criteria. In the following sections we will present an approach for estimating the efficiency of a proof based on cost estimates. It extends the computation of cost estimates as described in [Treitel, 1986] and [Treitel and Genesereth, 1987], where only the first two types of evaluation (forward and backward chaining) were considered. The cost estimates can then be used to choose a proof method for every rule.

3 Proof Strategies

Since there are a number of proof methods available for logic programs, we have to decide when to apply which method. We will first define what a proof strategy is:

Definition 4 (Proof Strategy) *A* proof strategy *is an assignment of a proof method to each clause of a logic program P.*

Now we will consider at which level and time a decision for a proof method can be made. Possible levels on which a strategy can be determined are (in order of decreasing specificity):

Rule: Each rule is associated with one of the four proof methods. This could mean that two clauses defining the same relation are evaluated by different proof methods.

Definition: All clauses defining one predicate must be evaluated by the same proof method.

Module: Procedures can be collected to modules, for which a uniform proof method is seeked.

Program: There is no collaboration between proof methods. A goal is proved by selecting one of the available proof methods in advance, which is used for the whole program.

System: Only one proof strategy is available in the system and consequently there is no choice. This is the case for most of the logic programming languages. Prolog, for instance, only supports a kind of SLD-resolution.

In principle, control decisions can be made either at run time or at compile time. Decisions made at run time can benefit from up-to-date information (e.g. actual variable bindings) and therefore are more precise. Their overhead, however, may counteract their improvement. Decisions made at compile time might be less accurate, but they can be made once and used several times. Also they can be more complex because the time they consume themselves does not increase the waiting time for an answer. Our approach is a *compile-time* approach making a decision at *definition level* and taking into account information from the query. The decision for a rule's strategy depends on cost estimates for its application. Then an overall strategy is an assignment of one of the four proof methods to every clause of the program such that all rules defining a particular predicate are evaluated by the same method.

It should be noted that it can be advantageous to apply different proof methods to a single rule depending on the variable bindings of the query. This will not be considered by our approach, but can be achieved in combination with program specialization techniques like partial evaluation [Komorowski, 1992]. In this case, the rules defining a predicate p are duplicated and specialized for a particular query *p(a,X)*. After renaming the predicate we have a new predicate *p'*, which will be applied if p is called with first argument bound to a. The rules defining *p'* can be assigned a proof method different from that of p.

4 Cost Estimates for a Proof Strategy

For each clause the cost estimates for evaluating them by any of the available proof methods are calculated. The total cost of a strategy S for a program P is computed by the following equation:

$$cost(S) = \sum_{r \in P} v_f(r,S) * e_f(r) + v_b(r,S) * e_b(r) + v_{gf}(r,S) * e_{gf}(r) + v_{mb}(r,S) * e_{mb}(r) \quad (1)$$

where $e_f(r)$ is the estimated cost for applying rule r in forward direction by semi-naive evaluation, $e_b(r)$ is the estimated cost for applying r in backward direction, $e_{gf}(r)$ is the estimated cost for using a goal-directed forward-chaining approach (after rewriting), and $e_{mb}(r)$ is the estimated cost for applying memoing backward chaining by OLDT resolution. The parameters v_x play the role of selectors:

$$v_f(r,S) = \begin{cases} 1 & \text{if } r \text{ is executed as a forward rule according to strategy S} \\ 0 & \text{else} \end{cases}$$

$$v_b(r,S) = \begin{cases} 1 & \text{if } r \text{ is executed as a backward rule in strategy S} \\ 0 & \text{else} \end{cases}$$

$$v_{gf}(r,S) = \begin{cases} 1 & \text{if } r \text{ is executed as a goal-directed forward rule in S} \\ 0 & \text{else} \end{cases}$$

$$v_{mb}(r,S) = \begin{cases} 1 & \text{if } r \text{ is executed as a backward rule with memoing in S} \\ 0 & \text{else} \end{cases}$$

The task is to choose values for $v_f(r,S)$, $v_b(r,S)$, $v_{gf}(r,S)$, and $v_{mb}(r,S)$ such that $cost(S)$ is minimal.

5 Computing Cost Estimates

The decision is made on *indirectly domain-dependent* information. It is more informative than purely syntactic approaches which are applicable to any program without any advice concerning their content. On the other hand, they abstract from deep knowledge about the domain and the concrete data of the program. In particular, we consider

- estimates on the number of facts for each predicate,
- the probability of deriving duplicates,
- the distribution of possible variable instantiations, and
- the number of answers that are needed (one answer or all answers).
- Additionally, we take into account the degree of restrictions given by the query.

In this section we will first repeat some equations from [Treitel and Genesereth, 1987] for calculating the basic values, i.e. the number of (unique) rule instantiations. Then we will present a systematic way for computing the costs of rule execution with any of the available proof methods.

5.1 Number of Rule Instantiations

An important value for the computation of cost estimates is the number of facts, which can be computed by a rule, i.e. the number of consistent instantiations of the body of a rule. The instantiation of two literals is consistent if common variables have identical bindings. The probability of consistent instantiations depends on the number of possible values for common variables. Consider two literals `p(X,Y)` and `q(X,Y,Z)`. Let *n(Q)* be the number of possible instantiations for a literal Q and let *d(X)* be the number of possible values for a variable X. Then the probability that their instantiation is consistent is equal to $d(\mathtt{X})^{-1} * d(\mathtt{Y})^{-1}$ and thus the number of consistent instantiations is

$$n(\mathtt{p}(\mathtt{X},\mathtt{Y})) * n(\mathtt{q}(\mathtt{X},\mathtt{Y},\mathtt{Z})) * d(\mathtt{X})^{-1} * d(\mathtt{Y})^{-1} \ .$$

The number of consistent instantiations of a rule is calculated iteratively simulating a left-to-right information passing strategy: Let r be a rule $P \leftarrow Q_1, Q_2, \ldots, Q_n$, let β_i be the set of variables of premise Q_{i+1} that have already been bound by the premises $Q_1, Q_2, \ldots, Q_i$. Let $A(r,i)$ be the number of consistent instantiations of the first i premises. Then

$$\begin{aligned} A(r,1) &= n(Q_1) \\ A(r,i+1) &= A(r,i) * n(Q_{i+1}) * \prod_{V \in \beta_i} d(V)^{-1} \end{aligned} \tag{2}$$

and the number $n(P)$ of derivable facts is equal to $A(r,n)$.

Example 5 If the premises of a rule do not share any variable, the number of possible facts derived by a rule is computed by the product of the possible instantiations for each premise. Consider the following rule:

```
p(X,Y) <- s(X), t(Y).
```

If there are 100 possible instantiations for `s` and 10 possible instantiations for `t`, the rule can compute 1000 facts. If the premises share variables, these must have the same value at each occurrence, which reduces the number of possible derivations. Then the number of consistent instantiations is equal to the number of possible instantiations divided by the number of possible values for each multiple occurrence of a variable. Consider the rule

```
p(X,Y) <- s(X,Z), t(Z,Y).
```

If the number d(Z) of possible values for variable `Z` is 5 and there are again 100 possible instantiations for `s` and 10 possible instantiations for `t`, then the number of derivable facts is computed by $100 * 10/5$, i.e. there are 200 possible derivations.

5.2 Unique Rule Instantiations

Different instantiations of the variables in a rule may lead to identical instantiations of the conclusion. This may be the case, for example, if rule instantiations differ only for variables, which do only occur in the premises. These multiple derivations of identical facts are called *duplicates*. A substantial factor for the selection of a method is whether duplicates are eliminated or proved redundantly. Therefore this has to be considered by the cost estimation. This means that we need to know not only how many consistent instantations of a rule can be found but also how many of them are *unique*.

Consider a rule $p(X) \leftarrow Q_1, Q_2, \ldots, Q_n$. Let again $d(X)$ be the number of possible values for a variable X. We first assume that every possible value for a variable occurs with equal probability. Then the number $E_X(m)$ of unique derivations after m derivation steps is computed by the following recursive formula:

$$\begin{aligned} E_X(1) &= 1 \\ E_X(m+1) &= E_X(m) + P_u(m) \ . \end{aligned}$$

with $P_u(m)$ being the probability that the m-th fact has not already been derived:

$$P_u(m) = 1 - \frac{E_X(m)}{d(X)}$$

The recursive definition for $E_X(m)$ can be approximated by the following formula:

$$E_X(m) = \frac{1 - (1 - d(X)^{-1})^m}{d(X)^{-1}} \ .$$

Now we generalize this formula for conclusions with multiple variables: Let $P \leftarrow Q_1, Q_2, \ldots, Q_n$ be a rule, let $X_1, \ldots, X_k$ be the variables of P and let $m = n(P) = A(r, n)$ be the number of consistent instantiations for P computed by formula (2). Then the number of unique instantiations $n_{unique}(P)$ is approximated by the formula

$$n_{unique}(P) = E_{X_1,\ldots,X_k}(m) = \frac{1 - (1-p)^m}{p} \qquad \text{with} \quad p = \prod_{i=1}^{k} d(X_i)^{-1} \tag{3}$$

Example 6 Assume that the rule

```
p(X) <- s(X,Z), t(Z,Y).
```

has 100 consistent instantiations, i.e. $n(\texttt{p(X)}) = 100$, and the domain of `X` has cardinality 200, i.e. $d(\texttt{X}) = 200$. Using formula (3) above we get

$$78 \quad < \quad E_X(100) \quad = \quad \frac{1 - (0.995)^{100}}{200^{-1}} \quad = \quad 78.8 \quad < \quad 79$$

Thus, from 100 derived facts we get probably 21–22 duplicates if they are not eliminated.

For these approximations it has been assumed that every possible value for a variable occurs with equal probability. If it is known that some values occur more or less often, we can use weight factors. The formula

$$E_X(n, a) = 1 - (1 - (g(a) * d(X)^{-1}))^n$$

computes the probability that variable X is instantiated with value a after n derivation steps. The weight $g(a)$ is a measure for the frequency of the value a being an instantiation for X.

5.3 Separating In-Costs and Out-Costs for a single Rule

With these basic values we can now compute the costs for evaluating a rule with any of the available proof methods. For systematic analysis we distinguish between *in* costs and *out* costs of a rule. The *out* costs correspond to the number of calls of a rule. *In* costs are the costs for proving the premises of a clause. Then the total costs of a rule are equal to the product of *in* costs and *out* costs because for each call the premises have to be tested.

The main value for *in* costs of a rule is the number of consistent instantiations for its premises. Here the elimination of duplicates has to be taken into account. However, premises are not proved by the rule itself, but by other rules. Thus, we see how the costs of one rule are influenced by the proof method of other rules. Let *I(r,i)* be the number of instantiations for the i^{th} premise of rule *r*. Depending on the proof method of the rule that derives these instantiations, its value is calculated either by formula (2) or (3):

$$I(r,i) = \begin{cases} E_{X_1,\ldots,X_k}(A(r,i)) & \text{if the i}^{th} \text{ premise is proved by a} \\ & \text{duplicate-eliminating method} \\ A(r,i) & \text{if duplicates are not elimated.} \end{cases}$$

For goal-directed reasoning (i.e. goal-directed forward chaining, backward chaining and memoing backward chaining) we have to consider that a rule is called with a query, such that not all instantiations are computed. Therefore we compute their costs with respect to a binding pattern.

Forward Chaining

The *in* costs of a forward rule are equal to the number of instantiations of the premises. To these costs we have to add the costs S for storing each of the unique derived facts.

$$InCosts_f(r) = \sum_{i=1}^{n} I(r,i) + E_{X_1,\ldots,X_k}(A(r,n)) * S$$

The *out* costs for forward chaining are equal to 1 because each instantiation is computed exactly once and can be used multiple times. This means that the forward costs for a rule r are simply the *in* costs:

$$\begin{aligned} e_f(r) &= InCosts_f(r) * OutCosts_f(r) \\ &= \sum_{i=1}^{n} I(r,i) + E_{X_1,\ldots,X_k}(A(r,n)) * S \end{aligned} \tag{4}$$

Goal-directed Forward Chaining

Goal-directed forward chaining is based on a rewriting of the rules to restrict the derivation of facts. The cost estimates must reflect the effort of evaluating the rewritten rules: the rewriting algorithm adds an additional premise with a *magic* predicate to the original rule and introduces new rules to derive instantiations for this new predicate. Thus, for goal-directed forward chaining the number of premises is increased because of the magic predicates, but the number of instantiations is reduced. The Magic-Sets rewriting depends on the binding pattern of the query. Consequently, the cost estimates are also computed with respect to a binding pattern.

Given a rule $p(\bar{X}) \leftarrow Q_1, Q_2, \ldots, Q_n$ and a query $? - p(\bar{X}_b)$ with binding pattern (adornment) *ad*. $\bar{X}$ stands for a vector of terms involving variables and constants and $\bar{X}_b$ stands for the ground terms of $\bar{X}$. The rewriting algorithm generates a new rule r_{ad} with additional premise $magic\_p^{ad}(\bar{X}_b)$:

$$r^{ad}: \qquad p^{ad}(\bar{X}) \leftarrow magic\_p^{ad}(\bar{X}_b), Q_1, Q_2, \ldots, Q_n$$

and k rules $m.r_1^{ad}, \ldots, m.r_k^{ad}$ defining $magic\_p^{ad}$, one for each rule defining p.

Rewriting does not effect the calculation of a rule's *out* costs. Because the *out* costs for forward-chaining rules are always equal to 1, this is also true for each of the rules that result from rewriting. The *in* costs, however, must be calculated differently. They are calculated by adding to the *in* costs of the original rules the costs of deriving the magic facts, which are equal to the costs for the rules $m\_p_i^{ad}$:

$$\begin{aligned} e_{gf}(r) &= InCosts_{gf}(r, ad) \\ &= InCosts_f(r) + \sum_{i=1}^{k} e_f(m.r_i^{ad}) \\ &= InCosts_f(r) + \sum_{i=1}^{k} InCosts(m.r_i^{ad}) \ . \end{aligned} \tag{5}$$

Backward Chaining

Similar to forward chaining the *in* costs for backward chaining depend on the number of rule instantiations. The number of instantiations is less, however, because some of the variables are assumed to be already instantiated by the query. Multiple instantiations in backward chaining mode are computed by backtracking steps. But for backtracking an additional price has to be paid for restoring the environment. We assume that these additional costs B are constant. If we assume that a rule $p(X, Y) \leftarrow Q_1, Q_2, \ldots, Q_n$ is called with unbound arguments in the query (i.e. the adornment *ad* is *ff*), the *in* costs are

$$InCosts_b(r, ff) = \sum_{i=1}^{n} I(r, i) * B \ .$$

For ordinary backward chaining a rule can be called from many places with identical binding pattern. Each time a rule is called all the instantiations are derived again. This is taken into consideration by the *out* costs. The *out* costs are equal to the number of calls for the rule with repect to a binding pattern for the query. If only one solution is needed, the *out* costs are 1. Most frequently, even if for the topmost goal one solution is sufficient, identical subgoals may be proved multiple times (cp. the Fibonacci numbers of Example 2).

Let the rule r be called by premise $i + 1$ in the body of another rule r'. The number of valid partial instantiations of premises $1, \ldots, i$ (as given by the intermediate solutions $A(r', i)$) corresponds to the calls of the backward chaining rule r. To calculate the number of calls for rule r we can use formula (2). The final *out* costs are computed by adding up the number of partial instantiations of every calling rule. There is no upper bound for the *out* costs.

To compute the cost estimates of a rule for backward chaining we have to sum up the product of *in* and *out* costs for each binding pattern *ad*:

$$e_b(r) = \sum_{ad} (InCosts_b(r, ad) * OutCosts_b(r, ad)) \ . \tag{6}$$

Memoing Backward Chaining

While for ordinary backward rules the *out* costs are equal to the number of calls for the rule, for the memoing method from these costs the probability for duplicates has to be subtracted. This means that the *out* costs are less than or equal to the number of the unique instantiations of the query Q with adornment *ad*. On the other hand, we have to add the costs S for storing the derived facts.

$$OutCosts_{mb}(r, ad) \quad \leq \quad n_{unique}(Q) * S \quad \leq \quad A(r, n) * S \ .$$

The costs of looking up the tables, however, are much smaller than for redoing the proof and are therefore neglected. Thus, the *in* costs for backward chaining with memoing are the same as without memoing and we get:

$$e_{mb}(r) = \sum_{ad} (InCosts_{mb}(r, ad) * OutCosts_{mb}(r, ad)) \ . \tag{7}$$

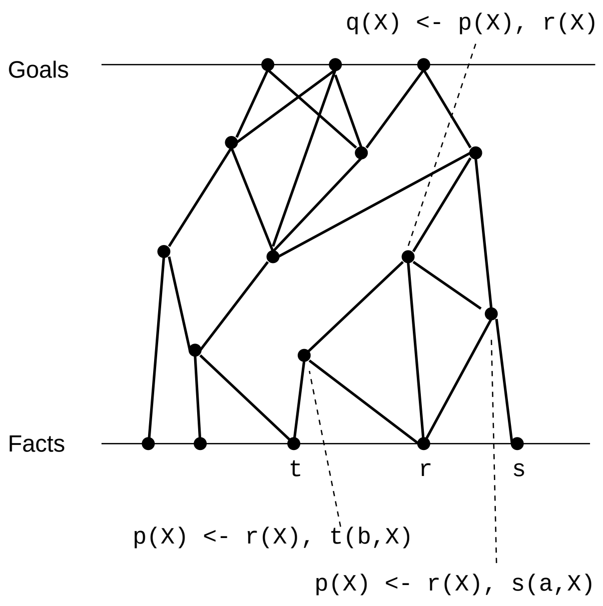

FIGURE 1: A rule graph

6 Propagating Cost Estimates

Now that we can compute cost estimates of individual rules we must find a strategy with optimal cost value, i.e. we have to minimize *cost*(S) as defined by formula (1) in Section 4. To illustrate the types of proof strategies we will extend the definition of a rule graph introduced in [Treitel, 1986]:

Definition 7 (Rule Graph [Treitel, 1986]) *A* rule graph *is a directed graph. The nodes in the graph are labeled by rules. There is an arc from rule* r *to rule* s *iff* r*'s output literal (the head of the clause) is unifiable with one of* s*'s input (body) literals. The rule* r *is said to be a predecessor of* s *and* s *a successor of* r.

Fig. 1 shows a rule graph with sample rules. At the bottom we see the facts and at the top we see the goal for the proof. The direction of an arc is from bottom to top. A proof strategy is drawn in the rule graph by using different kinds of arrows for the different evaluation methods of rules. Downward arrows starting from a node denote a backward rule and upward arrows ending in a node denote a forward rule (see Fig. 2). No rule can have both downward arrows starting from its corresponding nodes and upward arrows ending in its node.

An important aspect is the sequence of calculation. On the one hand, we separated *in* costs and *out* costs to determine the influences of the evaluation. In Section 5.3 we saw that the *in* costs of a rule depend on the number of consistent instantiations of its premises, where we have to take into account, whether duplicates are eliminated. On the other hand, the distinction between *in* and *out* costs

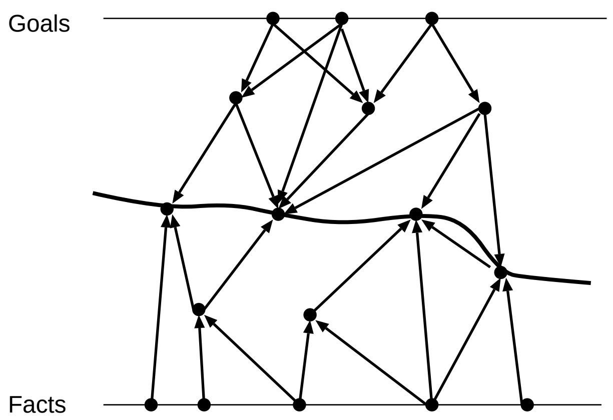

FIGURE 2: A coherent rule graph

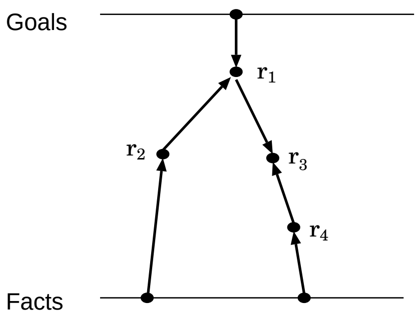

FIGURE 3: The rule graph for example 8

shows that the direction of one rule can only have a restricted influence on the costs of other rules.

Example 8 The simple rule system

$r_1:$	`p1(Y) <- p2(Y), p3(Y).`
$r_2:$	`p2(Y) <- b1(Y,W).`
$r_3:$	`p3(Y) <- p4(X,Y).`
$r_4:$	`p4(X,Y) <- b2(X,Y,Z).`

is represented by the rule graph of Fig. 3. From the arrows we see that r_2, r_4 are forward rules and r_1, r_3 are backward rules. The number of derived facts can be calculated bottom-up in the graph rule by rule. For example, we have to consider r_2 and r_3 before r_1. The number of instantiations for `p4(X,Y)` and for `p2(Y)` can be calculated from the base facts for `b1` and `b2`. The number of instantiations for `p4(X,Y)` again is used to compute the number of instantiations for `p1(Y)`. The *in* costs for each rule are computed in the same order as the facts because they require the number of instantiations, while the *out* costs are computed top-down using the values of the *in* costs and the number of derived facts. The *out* costs mainly depend on the number of times the rules are evaluated.

In general, the problem of finding an optimal strategy is NP-complete if only forward and backward chaining of rules are available [Treitel and Genesereth, 1987]. We have considered memoing backward chaining and goal-directed forward chaining as additional proof methods. Thus, the search space for the optimal strategy is increased by giving more alterntives to evaluate a rule. Restrictions on allowed strategies can reduce the effort. Coherence is an important property of strategies:

Definition 9 (Coherence) *A strategy is called* coherent*, if all successors of a backward rule are also backward rules and all predecessors of a forward rule are also forward rules. Otherwise the strategy is called* incoherent.

In the rule graph a coherent strategy can be identified, if it is possible to make a cut through the arcs, such that all rules below the cut are forward rules and all rules above are backward rules (Fig. 2).

The algorithm for computing an optimal strategy depends on the A* algorithm [Nilsson, 1980] and a lower bound of the estimates for the *in* costs and the number of derived facts of the rules. If only coherent strategies are allowed, the cost estimates for a rule do not depend on the direction of other rules. An important reason for this is that in a coherent strategy only rules at the cutting edge can change their direction for the strategy to remain coherent. This means that changing the direction of one rule cannot require changing the direction of any other rule. As [Treitel and Genesereth, 1987] and [Treitel, 1986] showed, for coherent strategies where *no* rules can generate duplicates an optimal strategy can be computed with effort $O(N^3)$, where N is the number of rules.

For incoherent strategies, changing the proof method of one clause can influence the costs for neighboring rules in the rule graph. This can lead to the consequence that the proof method for other rules should be changed. In our system a strategy is not required to be coherent. Any combination of proof methods is allowed. But also for incoherent strategies the propagation of costs (and consequently changing the direction of rules) is restricted. Considering these restrictions can improve the propagation algorithm. Most important, forward rules have the function of a wall for propagation, because their *out* costs are always equal to 1. This means that any changes of their *in* costs does not affect the *out* costs. Additionally, it does not matter how often their derived facts are used. This means that changing the direction of any rule that uses the result of a forward rule does not affect its costs.

7 Recursive Rules

The calculation of cost estimates works fine for nonrecursive rules. If we allow recursion we need, besides the indirect domain knowledge (Section 5), additional direct knowledge about the application domain. For example, it is hard to cope with transitivity and with equivalence relations.

Example 10 Consider the rules

```
t(X,Y) <- g(X,Y).
t(X,Y) <- g(X,Z, t(Z,Y).
t(X,Y) <- t(Y,X).
```

and the two fact bases:

```
DB1:    g(1,2).  g(2,3).  g(3,4).  g(4,5).  g(5,6).
        g(6,7).  g(7,8).  g(8,9).  g(9,10).


DB2:    g(1,2).  g(2,3).  g(3,2).  g(1,3).  g(3,1).
        g(8,9).  g(9,10). g(10,8). g(9,8).
```

The rule system derives 100 tuples for `t` if using the facts from `DB1` and only 18 tuples for `t` if using the facts from `DB2`. This difference cannot be detected by our approach for calculating cost estimates, because it uses only indirect domain knowledge (number of facts and possible values for the variables) which is identical for both databases.

But for a restricted form of recursion (e.g. without transitivity and equivalence relations) we can calculate cost estimates[2]. In a first step we identify clusters of mutually recursive rules. For propagating costs we collaps the rule graph treating all the mutually recursive rules of one cluster as a single node (Fig. 4). All the rules of a cluster are required to be evaluated in the same direction. To calculate the number of facts derived by the rules of one cluster we use an iterative approach. This iterative approach corresponds to the semi-naive strategy for evaluating recursive rules. A problem with this approach, however, is that – as for the corresponding evaluation – the iterative computation of possible facts is not guaranteed to terminate if the number of possible facts is infinite.

Consider for instance the clauses defining `ancestor` in Section 2. These two rules are collected into one cluster and the number of possible facts is estimated: Let's assume that our program contains `parent`-relations of at least four generations. To calculate the cost estimates we use the additional domain knowledge that every

[2]For the remaining cases the user can fix the proof method and give a constant cost estimation value.

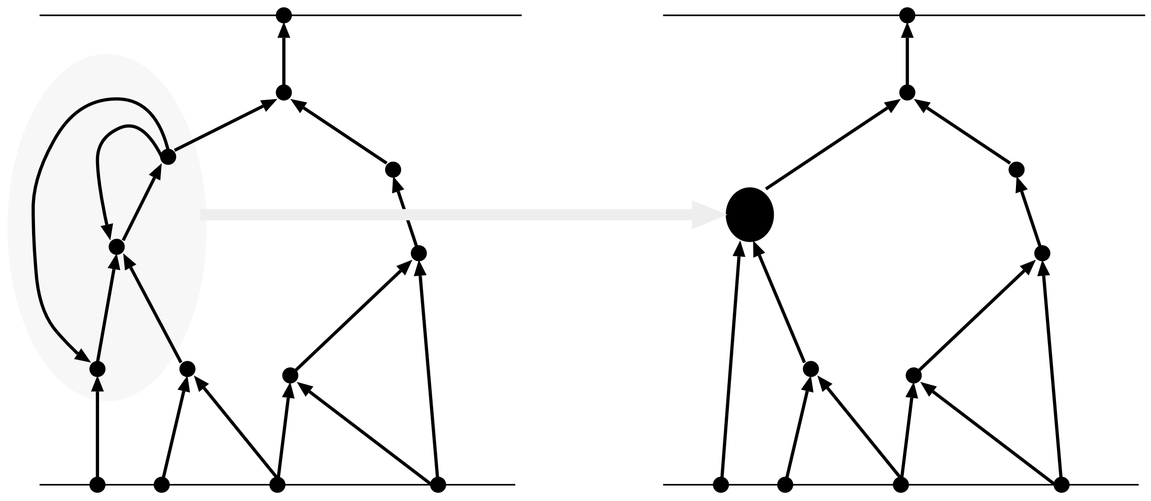

FIGURE 4: Collapsing nodes in recursive rule graphs

person has at least two parents. By iteration of four steps we can see that there are at least 30 solutions for every query of `ancestor` with the first argument bound, e.g. `?- ancestor(john,X)`. This estimated number of possible facts can then be used to compute the *in* costs of the rules calling `ancestor`.

The problems of calculating cost estimates for recursive rules must not be confused with the termination and completeness problems of the proof method. One problem is that there may be an infinite number of solutions. Then a complete system will derive all these solutions and thus will not terminate. However, this must not be the case for the calculation of cost estimates which must be finite. Completeness is a problem only for recursive programs. In Section 2 we saw that incompleteness occurs only if we execute recursive rules by ordinary backward chaining. To reduce termination problems we do not allow this method. If backward chaining is appropriate, the memoing version must be used.

8 Conclusion

We have presented an approach for selecting an efficient proof strategy for logic programs, which is based on cost estimates for evaluating each rule with any of the available proof methods. The basic information for calculation is the number of facts for each base predicate and the cardinality of the domain for each variable. From these values we can estimate the probability that an actually derived fact has been derived already in a previous step. An important valuation factor for a proof strategy is, whether these duplicates are eliminated or whether they lead to further redundant derivations by evaluating other rules. Additionally, we consider storage costs for derived facts and the expense for backtracking. To systematically analyze the influence that the evaluation of one rule has for other rules, we distinguish between *in* costs and *out* costs of a rule. In general, the *in* costs for forward reasoning are higher than for backward reasoning because either the number of instantiations is not restricted by a query or the effort of evaluating magic rules has to be added. For *out* costs the opposite is true because forward rules are evaluated exactly once and their results can be used several times.

The calculation of cost estimates for a strategy may vary with the particular implementations of the proof methods and with the intended application of the program: For instance, an enormous efficiency gain would be reached if we can use matching instead of unification. This is possible if the rules are required to be range-restricted, i.e. every variable in the head of a clause has to be bound by a literal in the body. This decreases the *in* costs for a forward chaining rule because the premises have to be tested against *ground* facts only requiring just matching instead of unification. This means that instead of simply counting the number of instantiations and backtracking steps they have to be multiplied by different factors for forward and backward chaining, respectively.

The memoing version of backward chaining can dramatically reduce the complexity of a proof compared to backward chaining without memoing. Since it also forces termination in many cases, memoing backward chaining should be preferred for safety reasons. On the other hand, the tabulation of solutions may be very space consuming. Since it cannot be determined automatically whether rules are safe, it should be in the responsiblity of the programmer to annotate rules as 'safe'. Only for safe rules backward chaining without tabulation of solutions should be selected as an allowed proof method.

Bry showed by partial evaluation of an upside-down meta-interpreter that – with respect to the proved subgoals – bottom-up reasoning of a Magic Set-rewritten program is equivalent to top-down reasoning of the original program [Bry, 1990]. Since bottom-up evaluation avoids multiple derivations of lemmas, the cost estimates for a top-down proof with tabulation and goal-directed bottom-up reasoning are comparable. But for real implementations the effort for accessing previous solutions and for satisfying premises may differ. For example, having matching instead of unification for forward chaining reduces the costs compared to backward chaining. Also, while bottom-up evaluation and the QSQR top-down evaluation are set-oriented, OLDT resolution is tuple-oriented. Therefore we have separated the computation of their cost estimates.

Goal-directed forward chaining requires additional effort for program rewriting, in particular if there is a large number of rules. This effort has not been considered by the cost calculation. It is assumed that the program will be rewritten at compile-time. This is usually the case for deductive databases and many applications of logic programs, where queries are embedded into fixed application scenarios. On the other hand, there may be applications of information systems, where not every possible query can be anticipated at compile time. Then the system has to react on various kinds of unpredicted queries, such that program rewriting would considerably increase the answer time and must not be ignored for cost estimation.

Access to external data has not been taken into account in any way, although it may considerably influence the selection of a proof method. If some facts reside in a database instead of main memory, the number of accesses is a considerable value. For this reason, deductive databases heavily prefer set-oriented evaluation instead of tuple-oriented approaches.

In summary, the decision for a proof strategy can depend on general as well as query-specific factors. Many of these influences are considered by the cost estimates presented in this paper.

References

[Balbin *et al.*, 1991] I. Balbin, G. S. Port, K. Ramamohanarao, and K. Meenakshi. Efficient bottom-up computation of queries on stratified databases. *Journal of Logic Programming*, 11:295–344, 1991.

[Bancilhon and Ramakrishnan, 1986] Francois Bancilhon and Raghu Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD Conference*, pages 16–52. ACM, 1986.

[Bancilhon and Ramakrishnan, 1988] Francois Bancilhon and Raghu Ramakrishnan. Performance evaluation of data intensive logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 441–517. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1988.

[Bancilhon *et al.*, 1986] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15. ACM, 1986.

[Bancilhon, 1985] F. Bancilhon. A note on the performance on rule-based systems. Technical Report DB-022-85, MCC, 1985.

[Beeri and Ramakrishnan, 1991] Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10:255–299, October 1991.

[Boley *et al.*, 1993] Harold Boley, Philipp Hanschke, Knut Hinkelmann, and Manfred Meyer. COLAB: A hybrid knowledge compilation laboratory. Research Report RR-93-08, DFKI, Kaiserslautern, Germany, January 1993. Also to appear in *Annals of Operations Research.*

[Bry, 1990] Francois Bry. Query evaluation in recursive databases: bottom-up and top-down reconciled. *Data and Knowledge Engineering*, 5:289–312, 1990.

[Dietrich, 1987] S. W. Dietrich. Extension tables: Memo relations in logic programming. In *SLP-87*, 1987.

[Komorowski, 1992] J. Komorowski. An introduction to partial deduction. In A. Pettorossi, editor, *Meta-Programming in Logic, Uppsala, Sweden, June 1992 (Lecture Notes in Computer Science, vol. 649)*, pages 49–69. Berlin: Springer-Verlag, 1992.

[Labisch, 1993] Thomas Labisch. Developing a combined forward/backward-chaining system for logic programs in a hybrid expertsystem shell. Master's thesis, Universität Kaiserslautern, June 1993. In German.

[Nejdl, 1987] Wolfgang Nejdl. Recursive strategies for answering recursive queries – the RQA/FQI strategy. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, pages 43–50, Brighton, 1987.

[Nilsson, 1980] Nils J Nilsson. *Principles of Artificial Intelligence.* Tioga, Palo Alto, CA, 1980.

[Ramakrishnan, 1988] Raghu Ramakrishnan. Magic templates: A spellbinding approach to logic programms. In R.A. Kowalski and K.B. Bowen, editors, *Proceedings of the 5th International Conference and Symposium on Logic Programming*, 1988.

[Rohmer *et al.*, 1986] J. Rohmer, R. Lescoeur, and J. M. Kerisit. The alexander method - a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, pages 273–285, 1986.

[Sacca and Zaniolo, 1986] D. Sacca and C. Zaniolo. The generalized counting method for recursive logic queries. In *First International Conference on Database Theory*, 1986.

[Tamaki and Sato, 1986] Hisso Tamaki and Taisuke Sato. OLD resolution with tabulation. In E. Shapiro, editor, *Third International Conference on Logic Programming (ICLP)*, LNCS 225, pages 505–512, London, July 1986. Springer Verlag.

[Treitel and Genesereth, 1987] Richard Treitel and Michael R. Genesereth. Choosing directions for rules. *Journal of Automated Reasoning*, 3:395–431, 1987.

[Treitel, 1986] Richard Treitel. Sequentialization of logic programs. Technical Report STAN-CS-86-1135, Stanford University, Department of Computer Science, November 1986.

[Vieille, 1986] Laurent Vieille. Recursive axioms in deductive databases: The query/subquery approach. In L. Kerschberg, editor, *Proceedings of the First International Conference on Expert Database Systems*, April 1986.