CHOOSING DIRECTIONS FOR RULES

Richard Treitel and Michael R. Genesereth*

Logic Group, Knowledge Sytems Laboratory,
Computer Science Dept., Stanford University

ABSTRACT

In "expert systems" and other applications of logic programming, the issue arises of whether to use rules for forward or backward inference, i.e. whether deduction should be driven by the facts available to the program or the questions that are put to it. Often some mixture of the two is cheaper than using either mode exclusively. We show that, under two restrictive assumptions, optimal choices of directions for the rules can be made in time polynomial in the number of rules in a recursion-free logic program. If we abandon either of these restrictions, the optimal choice is NP-complete. A broad range of cost measures can be used, and can be combined with bounds on some element of the total cost.

I INTRODUCTION

In logic programming, decisions about which inference direction to use, based on rough estimates of the computational costs of each direction, are frequently taken by users. We would like to automate the choice between forward and backward inference, at least with respect to the cost of computation. Other optimisations, such as rule ordering and ordering of terms within rules, will be ignored, as will other considerations that could affect the choice of inference direction. The conflicts between forward and backward computation can be outlined as follows: solving a goal backwards may be much cheaper than doing the corresponding forwards deduction, because more variable bindings are available to constrain the computation. Or it may be more expensive, because several rules are applied, only one of which has enough facts to solve the goal. A fact that has been deduced forward and stored can be reused many times, or it may never be used at all. We will show how to attach numerical estimates to these factors and optimise the trade-offs.

A. Statement of problem

We consider a system whose inputs are a set F of facts (ground atomic formulae), a set R of rules (sentences having implicational force), and a set G of goals (which may be conjunctions). Rules and goals may contain variables, which are assumed to be universally quantified if in rules and existentially quantified if in goals. No function symbols appear in R or G. The purpose of the system is to solve each goal from G using the facts F and the rules R, and, in the case of goals containing variables, to find all the sets of variable bindings which make the goal true. The deductive mechanism used for both forward and backward inference will be a restricted form of resolution, and the members of F, R, and G are assumed to be in conjunctive normal form. We impose the restriction that clauses in R be Horn clauses, i.e. have exactly one positive literal. This literal is the consequent of the rule, the others being its antecedents.

The problem we consider is that of choosing an optimal subset R_f of R to be used forwards. Optimality is defined with respect to the sum of the times taken by all the deductions. For a program whose database of deduced clauses was kept from one run to the next, the daily cost of renting disk space would have to be added to the cost of the CPU cycles consumed per day. The set of facts for which storage costs are incurred can be changed, as discussed below. Bounds may be imposed on the space available for storing facts, or on the time taken by forwards inference or backwards inference or both.

B. Notation

We define a directed graph, called the *rule graph*, whose nodes are the members of R and which has an arc from a rule r to a rule s iff r's consequent is unifiable with one of s's antecedents. We say that s is a *successor* of r, and r a *predecessor* of s. The rule graph will be required to be acyclic, since the work we have done to date does not include techniques for estimating the costs of using recursive rules. We may add F and G to the rule graph, in the obvious places: a fact from F is the predecessor of those rules whose antecedents unify with it, and a goal from G is the successor of those rules whose consequents unify with it. We do not represent individual members of F and G in the rule graph, but sets of facts or goals that match some pattern; these patterns, rather than the individual facts or goals, are used to construct arcs in the rule graph. In Figure 1, we have put the facts at the bottom and the goals at the top.

For a node r in the rule graph, representing a rule from R, let $e_f(r)$ be the cost of using it forwards, and $e_b(r)$ the cost of using it backwards, assuming in each case that

*This work was supported by the Office of Naval Research, the National Institute of Health, and Martin-Marietta under contracts N00014-81-K-0004, NIH 5P41 RR 00785, and GH3-116803

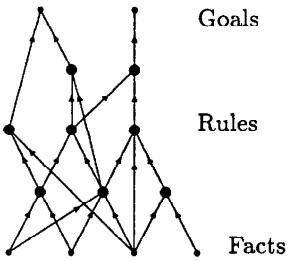

Figure 1: A fairly tangled rule graph

its antecedent facts are available in the database. Below we describe how to estimate these costs. We also define an indicator variable $v(r)$ for each r, with a value of 1 to denote using the rule forwards, and 0 for backwards. A complete set of values of $v(r)$ for all rules r will be called a *strategy*.

$E_f(r)$ always depends on the structure of R and on the numbers of facts in that part of F from which r's inputs are obtained, and $e_b(r)$ can depend both on these and on that part of G from which subgoals that invoke r are obtained. Moreover, $e_b(r)$ will depend on whether any of r's successors are used forwards. We can eliminate this dependence by simply insisting that all successors of a backwards rule be used backwards themselves, or equivalently, that R_f is closed under the operation of taking predecessors. We shall call this *coherence*, i.e. a coherent strategy will be one in which no rule is used forwards unless all its predecessors are. Some deductive systems do in fact enforce coherence, and others have a bias towards it.

II THE OPTIMAL COHERENT STRATEGY

In this paper we examine only the case where all strategies are required to be coherent. If additionally no rules generate duplicate answers (the same answer deduced from different sets of facts), then the optimal set R_f can be found by any linear programming method. The optimisation problem is NP-complete if there are such rules. By treating them separately from others, we can find the optimal set R_f in time bounded by a polynomial in the total number of rules times an exponential in the number of "bad" rules (and much more quickly than this in most cases).

A. No Duplicate Answers

Under the two restrictions mentioned above (coherence of strategy and no duplicate answers), the cost estimates $e_f(x)$ and $e_b(x)$ can be made to have only F, R, and G as implicit arguments. In particular, they do not depend on the directions of rules other than x. Then the problem of finding the set of rules to be used forwards in a least total cost strategy is just an integer programming problem. Namely, the total cost of a strategy (represented as a set of values for the $v(x)$) is

$$\sum_x v(x)e_f(x) + (1 - v(x))e_b(x)$$

and the coherence constraint turns into inequalities $v(y) \geq v(x)$ for each predecessor y of x. If this expression for the cost is minimised subject to these inequalities, the resulting values of the variables $v(x)$ give the optimal solution to the original problem.

This integer program is in fact a linear program, and hence solvable in time polynomial in the number of constraints, which is polynomial in the total number of literals in R. This is important, since integer programming in general is NP-complete. To see that we really have a linear program here, we must prove that if the above constraints were augmented by the inequalities $0 \leq v(x) \leq 1$ for all x and the whole system solved as a linear program, the solution obtained would in fact give integer values to all the $v(x)$. This is done by showing that a solution in fractional values would not be a vertex of the simplex defined by the constraints. It then follows that if the Simplex Algorithm is applied to the above equations and these constraints, it will find a solution in integers, which will be the solution to the integer program, corresponding to the optimal strategy.

[4] has considered a similar problem, different from this one mainly in the imposition of an upper bound on the total amount of storage available for the facts deduced (it is hard to tell whether he requires his strategies to be coherent). Such a bound can be added to this problem very simply, for if $e_s(x)$ is the estimated amount of space taken up by the facts deduced by rule x, and A is the total space available, we just add the inequality

$$\sum_x e_s(x)v(x) \leq A$$

to the linear program. The same would apply to a time bound or to a bound on any expression linear in the $v(x)$, when combined with any cost function linear in the $v(x)$.

Roussopoulos also expects that (translating into our terminology) the only facts stored permanently are those that are inputs to some backwards rule. Until now we have lumped space costs with time costs, thus assuming that every fact is stored. We can change this by re-defining the cost of a strategy as

$$\sum_x v(x)e_f(x) + (1 - v(x))e_b(x) + v'(x)e_s(x)$$

where $e_s(x)$ is the estimated cost of storing the facts deduced by rule x. The new variable $v'(x)$ is made to be 1 if x is a forward rule with a backward rule among its successors, and 0 otherwise, by the constraints

$$v'(x) \geq v(x) - v(y)$$

for all successors y of x. If x has no successors except goals then $v'(x)$ is made identical to $v(x)$. The rule graph must also be extended to include nodes for the facts in F, each of which has zero values for $e_f(x)$ and $e_b(x)$ and 1 for $v(x)$. These changes roughly double the numbers of variables and constraints in the linear programming problem.

B. Duplicate Answers Present

Some rules can generate duplicate answers to a goal, corresponding to different values of some variable which appears in the rule's antecedents but not in its consequent. For example, given the rule

$$A(x,y)\ \&\ B(y,z) \Rightarrow C(x,z)$$

if facts $A(1,2), B(2,3), A(1,4)$, and $B(4,3)$ were available, then $C(1,3)$ could be deduced twice, once with $y=2$ and once with $y=4$. If the rule's conclusions are being stored in the database, the duplicates will disappear, but if the rule is being used in backwards inference and its conclusions forgotten as soon as they are used, then duplicates will not even be detected.

If some rule r_1 has a predecessor r_2 which generates duplicates, the number of inputs supplied to r_1 will depend on whether r_2 is used forwards or backwards, and this clearly affects the cost of using r_1 backwards. The cost of using other predecessors of r_1 can also be affected by this, because the number of clauses resolving against their positive literal can change. So if r_3 is such a rule, the cost of using r_3 backwards will depend on $v(r_2)$. Only the backward costs can be affected, for a rule used forwards in a coherent strategy receives no duplicates from its predecessors. This makes it impossible to express the cost of r_3 as a linear function of $v(r_3)$ and $v(r_2)$, since the value of $v(r_2)$ affects it only when $v(r_3)=0$; terms containing the product of the two indicator variables would be required to express this. Thus the linear program cannot be used.

In fact, the task of optimally choosing the $v(x)$ under these conditions is NP-complete [6], so that it cannot be solved by linear programming unless P=NP. But this is not as discouraging as it may seem to be. Considerable care was required to construct the logic program used in the proof, and we anticipate that most programs encountered in practice would not display the features that make it necessary to explore many strategies. We therefore expect that a heuristic-guided A^* search could solve most such optimisation problems quite fast.

The appropriate heuristic for this search is a lower bound on the cost of any strategy containing a given *partial strategy* (a set of values for some of the $v(x)$). Such a lower bound can be obtained by assuming that all duplicate answers are magically eliminated. This allows us to make estimates of the costs of using all rules in each direction, which will certainly be no higher than the true costs. Feeding these estimates, and the values for those $v(x)$ that are included in the partial strategy, into a linear programming algorithm, we get a cost which cannot be higher than that of any complete strategy that extends the partial strategy. As soon as the rules which generate the most duplicates (relative to their number of unique answers) are in the partial strategy, the lower bound will be fairly accurate, so the search will be well focussed towards good strategies if these rules are among the first ones added. And once a partial strategy includes directions for all rules that can generate duplicates, the optimal complete strategy containing it is returned immediately by the linear program.

III THE DEDUCTIVE METHOD

In order to describe the estimation of the computational costs of using rules, we must specify precisely the version of resolution which we assume to be used for deductions.

Binary resolution is sound and complete but inefficient. We impose on it the restriction that the complementary literals which are resolved away must each be the first in their respective clauses. This is similar to lock resolution [1]. Thus the number of possible resolutions on a given clause set is substantially reduced, but in general completeness is sacrificed. With some more effort it can be restored.

The implementation that we shall consider includes a stack or *agenda* of clauses to be resolved against. To use a clause, we add it to the agenda. We then repeatedly pop the top clause off the agenda, store it in the database perhaps, resolve it against all possible clauses in the database, and add the resolvents to the agenda. The literals from the clause that was found in the database come before those from the clause taken off the agenda (this causes subgoals to be solved before work on their parent goal is resumed), and the order of literals within each parent clause is carried over unchanged. When the agenda is empty, we have deduced all the consequences of the new clause.

When the clauses entered by the user are all Horn, and the non-unit clauses have their positive literal at one end or the other, this kind of resolution begins to look very much like traditional forward or backward chaining. Consider a rule $A(x,y)\ \&\ B(y,z) \Rightarrow C(x,z)$, which can be written in two ways:

$$\neg A(x,y)\neg B(y,z)C(x,z)$$
$$C(x,z)\neg A(x,y)\neg B(y,z)$$

With the first of these, facts like $A(1,2)$ and $B(2,3)$ will resolve, in that order, giving $C(1,3)$, which is just what would emerge from forward inference. The second form of the rule can resolve with a goal like $C(v,w)$ to give a clause $\neg A(v,y)\neg B(y,w)$, which is just a conjunction of subgoals whose answers would give the answer to the goal, and this looks like a backward chaining step.

For forward inference to be complete, either the facts must be presented in the same order as that in which the negative literals of the rule appear (in the above example, the rule could not resolve with $B(2,3)$ unless $A(1,2)$ had already been taken off the agenda) or else all facts, including those deduced by forwards rules, must be kept in the database for as long as there is any rule that might wish to resolve against them. If $B(2,3)$ was in fact taken off the agenda first, it would have to be stored until $A(1,2)$ came along and generated $\neg B(2,z)C(1,z)$, which would then resolve against it. In general it would also be necessary to store the non-unit resolvents. An alternative to this would be to have several versions of a forward rule, namely one beginning with each input literal. The version that began with the literal corresponding to the last of a set of facts to be presented would resolve against this fact and then against all the others, if they had been stored; there would be no need to keep intermediate resolvents. This would

lead to extra deductive cost due to abortive use of versions of a rule when not enough facts were there for it to succeed.

IV ESTIMATING COSTS OF DEDUCTION

We estimate the costs of running our form of resolution on a set of rules, facts, and goals by means of a simulation, in which we represent each set of similar clauses that will arise during the computation by a clause, called the set's *pattern*, and a number, namely the expected number of instances of that pattern that will be generated. The sets F and G are represented this way, since we do not expect to know exactly what facts or goals will be in them. If we regard a clause in R as having itself as pattern and the number 1.0, then we see that the basic step needed for estimating costs is to simulate resolutions between pairs of such clause sets and estimate their costs.

The simulator accepts the sets of patterns from F, R, and G as input, and obtains descriptions (in terms of pattern and set size) of all the sets of clauses that will be generated. It has an agenda like the one used for the resolution, so that the effects of putting clauses on the agenda in different orders can be simulated. We can combine the output of the simulator with knowledge about how long each elementary operation (unification, substitution, and so on) will take, to arrive at actual time estimates. It is then necessary to decompose the simulated cost into a sum of rule costs.

A. The Number of Resolvents

Here we describe how to estimate the number of resolvents generated at each node in the rule graph, given estimates for the numbers of propositions matching each pattern in F and G. We also need to know, for each variable in any clause of R, the size of the domain of values over which that variable will range. This is important for two reasons. First, some of the equations involved (which have been omitted for space reasons) are couched in terms of the probability of a typical instance of some clause pattern being generated, so that in order to derive a cost estimate, we need to know the number of potential instances of this pattern. Domain sizes also affect the probability that two variables will have been bound to different values, which in turn affects the chance that a unification will be successful; however, this probability may be known independently. Note that "domain" here refers to the set of values expected to be encountered during a particular run of the program, rather than to a set of theoretically possible values.

1. Simulated unification

It is useful to distinguish the set of variables in a pattern which will have had constants substituted for them at the time when unification is attempted. We call these "bound variables" of the pattern, meaning that the simulation must know that they will be bound at run-time to constants whose values are not known yet. The other variables in the pattern will be referred to as its "free variables", or "variables" if there is no ambiguity. Now clearly the pattern of a set of resolvents will be just the result of resolving the patterns of the two parent sets. However, when a pattern that has bound variables is unified with another pattern, this represents some unifications at run-time in which constants will have been substituted for these bound variables, and the unification may fail if two unequal constants have appeared. So, when the simulator is unifying two patterns, it must take special note of their bound variables.

In the absence of specific information, we can estimate the probability of successful unification between a bound variable and a constant or another bound variable by assuming that all values in the domain of the bound variable are equally likely to occur. This is called the *equal frequency assumption*: no value appears more often than another of the same type. The probability of a unification succeeding is then the reciprocal of the number of possible values in the domain.

If the distribution of actual constants in the facts and goals does not conform to the equal frequency assumption, the estimated numbers of resolvents may be arbitrarily badly wrong. Two safety mechanisms are possible for this. The first is to specify that some value is going to be over- or under-represented relative to the average; this could be done for several values. The second is to allow the user to give the probability of successful unification directly. For example, in a Computer Science Department where all the students have ages between 12 and 50, the probability of a random student being the same age (in years) as another may actually be 0.2 or so. The simulator can simply use this value instead of subtracting 12 from 50 and taking the reciprocal.

2. Estimating set sizes

The estimated number of clauses in the set of resolvents is the probability of successful unification times the number of attempted unifications (which is just the product of the estimated numbers of clauses for the parent sets). In general, two literals being unified by the simulator may contain several pairs of constants or bound variables that must be equal for unification to succeed. We make an *argument independence assumption*, under which the event of one pair being equal is independent of other pairs, so the probabilities can be multiplied.

However, some of the bound variables in the parent patterns may not correspond to anything in the resolvent pattern. It may happen that some pairs of parent clauses will differ only in the binding of such a variable, so that duplicates of some instances of the resolvent can occur, as was indicated above. If the resolvents are stored in the database, these duplicates will presumably be detected and eliminated, reducing the number of clauses that are available to subsequent resolutions. The appropriate changes to the estimated set sizes have been given in [5].

Given directions for each rule in R, and the patterns and

estimated sizes for sets of terms in F and G, we can now iteratively obtain descriptions of all the sets of resolvents generated. This approach is clearly not adequate for dealing with recursive rules in R, which correspond to cycles in the rule graph. Techniques for dealing with recursive rules are being investigated by many researchers [3,2,7].

B. Breaking down the costs

Since a clause pattern can be derived via a sequence of resolutions involving several rules from R, we need some way of assigning the costs associated with a set of clauses to one and only one rule, or perhaps to a goal, so that the total cost of a strategy is equal to the sum of costs over all the rules and goals, and so that the cost numbers for each direction of each rule accurately reflect the consequences of using that rule in that direction. We make this assignment by considering the first literal of the clause set's pattern, which must have been obtained by applying some number (possibly zero) of substitutions to a literal of a rule or goal from R or G. We charge the costs associated with the set against this rule or goal.

Minor adjustments must be made to this even in the coherent case, since the number of database lookup operations done by a rule depends on which, if any, of its predecessors are used forwards. It turns out to be possible to remove this variability in the cost of a rule by assuming that it looks up all its inputs in the database, and then adjusting the costs of backward rules to reflect the fact that their answers do not get looked up and so do not contribute to lookup costs. In the incoherent case, it is impossible to define the cost of using the rule backwards independently of the rest of the strategy it is used in.

V CONCLUSIONS

We have shown how a certain optimisation on logic programs can be performed cheaply under a fairly commonly encountered set of conditions. It is difficult to quantify the benefits available from this optimisation, since problems can easily be conceived which would take arbitrarily long to solve if only one of forward and backward inference were used, but are soluble in modest amounts of time by an appropriate combination of the two. Human programmers, confronted with such problems, will usually make sensible choices; the claimed advantages for this procedure are that it gives the precisely optimal strategy, and that it can easily be tailored to the performance of any inference engine by adjusting the calculations of $e_f(x)$ and $e_b(x)$.

Note that although the cost estimation methods fail on recursive sets of rules, the optimisation algorithms do not. If estimates e_f and e_b were available for such rules, the coherence condition requires that any set of mutually recursive rules be used in the same direction as each other, so for the purposes of optimisation they could be treated like one rule, and the linear program or the search algorithm could be used.

The problem of finding the optimal incoherent strategy, under the assumptions used here, is discussed in [6]. The obvious next extension to this work will be the study of how to optimise the ordering of negative literals within clauses together with the directions in which the clauses are used. Another important direction for future research will be the investigation of "adaptive" or "mixed" methods, which use information gathered at run-time to change or control a generic strategy devised at compile-time.

REFERENCES

[1] Robert S. Boyer. *Locking: A Restriction of Resolution.* PhD thesis, University of Texas at Austin, August 1971.

[2] L.J. Henschen and S.A. Naqvi. Compiling queries in recursive first order databases. *Journal of the ACM*, 31(1):47–85, January 1984.

[3] D. P. McKay and S. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh IJCAI*, pages 368–374, August 1981.

[4] Nicholas Roussopoulos. Indexing views in a relational database. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

[5] D. E. Smith. *Controlling Inference.* PhD thesis, Stanford University, July 1985.

[6] R.J. Treitel. *Sequentialising Logic Programs.* PhD thesis, Stanford University, 1986.

[7] Jeffrey D. Ullman. *Implementation of Logical Query Languages for Databases.* Technical Report STAN-CS-84-1000, Stanford University, May 1984.