Effects of Inference Direction on the Optimal Ordering of a Rule's Premises *

Richard Treitel
IntelliCorp
1975 El Camino Real West
Mountain View, Ca. 94040-2216

David E. Smith
Rockwell International Science Center
444 High St.
Palo Alto, CA 94301

Abstract

Previous work [TG87] has studied the problem of choosing inference directions (forwards or backwards) for a recursion-free set of Horn clause rules so as to minimise the expected execution time for solving a set of goals. Under a constraint on the choices of rule directions, called *coherence*, it was shown that the problem becomes quite tractable. Indeed, in a special case it admits of direct solution by a polynomial-time algorithm.

We now consider the same problem augmented by the simultaneous choice of an ordering for the premises of each rule. We prove that optimally ordering the premises of a single rule is NP-complete, and proceed to discuss the increase in difficulty attendant on trying to order the premises of several rules whose directions have already been chosen; this turns out to be strongly dependent on the level of detail with which the optimiser is expected to deal.

We show that the problem of choosing both directions and orderings so as to minimise expected execution time would still be NP-complete, even if the ordering problem for a single rule were easy. A two-phase algorithm which chooses the directions separately from the orderings cannot be powerful enough to achieve the optimal combination; we give an example of how such an algorithm fails.

Drawing on previous results about ordering of conjunctions [SG85], we present a method that offers some hope of performing the combined optimisation quickly and, moreover, does not depend on coherence. The method is based on a best-first search of possible sets of rule directions. For each partial set of direction choices, a suitable conjunct ordering algorithm helps determine a lower bound on the execution time, and these lower bounds are used to guide the search. We discuss a factor that will affect the speed of the search.

*This work was partially supported by the Defense Advanced Research Projects Agency under contract number N00039-86-C-0033 and by the Office of Naval Research under contract number N00014-81-K-0004

1 Introduction

Considerable attention has been paid to the need for correct ordering of the terms of a conjunctive goal if it is to be solved without wasted work. Recent contributions to this work include [IK84, SG85, Nat87]. Though these articles treat only the case of a goal, it is clear that the same considerations apply when the conjunction is the antecedent part of a rule being used in subgoaling or even in forwards inference. The choice between forwards and backwards inference has also been discussed in standard texts such as [Nil80, WOLB84], and more recently in [TG87]. Analogous problems in relational database optimisation are considered in [Rou82, FST86]. However, the work on conjunct ordering has assumed that the direction of inference is known in advance, while that on inference directions has paid no attention to conjunct ordering, with the exception of [FST86].

It is not hard to see that the optimal conjunct order can be different for forwards inference than for backwards inference. For example, given a rule for finding Senators' daughters

$$Female(x) \ \& \ Parent(x, y) \ \& \ Senator(y) \Rightarrow SD(x)$$

if we have to determine by backwards inference whether someone is the daughter of a Senator, it will be best to solve the conjuncts on the left hand side of the rule in the order shown. But if we are processing some election results and updating the set of Senators' daughters by forward inference, it would be ludicrously inefficent to use this order, since we would respond to every assertion of the form $Senator(y)$ by solving $Female(x)$, which would return the set of all known females.

Conversely, if the conjunct order is fixed in advance and we are choosing the inference direction, the order will affect the costs of forward inference and backward inference, and may bias our choice strongly towards one or other direction. Thus the optimisations of inference direction and conjunct ordering are interlocked. The principal objective of this article is to examine their effects on each other, and how they can be made to work together effectively.

1.1 The optimiser

We now proceed to explain the context in which we propose to study these optimisations. The overall goal is the re-organisation of a set of rules at "compile-time" (before running them) to reduce the expected cost of applying them to some set of problems.

Optimising a program at compile-time has several advantages and disadvantages compared with optimisation at run-time. Compile-time optimisation can use more computationally expensive methods, because its cost can be amortised over many runs of a program. Thus the program can be viewed globally, which is often more powerful than local optimisation. However, compile-time optimisation may produce a form of the program that will not be optimal for any one run, for two reasons. The optimised form will presumably be derived from some kind of compromise or average over the variation expected between individual runs, so as to give the best value for the total cost of all runs. And even if some run is exactly described by the average parameters that were given to the optimiser, the computations done during such a run probably will be modelled inexactly by the optimiser, leading to faulty optimisation. We regard these difficulties as both

unavoidable and acceptable, and so we expect to optimise at compile-time based on estimates of how the computation will go.

1.1.1 Task specification

We assume that the optimiser is given as its input the following four items:

- a set R of Horn clauses, which are the rules. We shall require that they be recursion-free, i.e. no predicate can be defined in terms of itself.
- a description G of the set of goals that will be presented, in the form of a set of <pattern, number> pairs, where the number is the expected number of goals that will match the pattern (see below for details). Goals are assumed to consist of negated literals, possibly containing free variables.
- a similar description F of the set of facts that will be supplied. Facts are assumed to be ground positive literals.
- the sizes of the domains over which each variable can range, or else the probability that two variables ranging over a given domain will have the same value at run-time (this will be explained further in Section 2.1.1).

These four inputs implicitly describe a logic program that will use the facts and rules to try to find all the solutions to each goal.

The optimiser will not tamper with any of the vocabulary used in the rules; it will be confined to making changes in the way rules are used. Its output will be a *strategy* in which each rule will be assigned an inference direction, forwards or backwards, and will be used only in that direction. Also, the order of solution of the conjuncts of each goal, and of the premises of each rule, will be defined by this strategy, and, in the case of a rule used backwards, different orders can be specified for different classes of goals to be solved by the rule. More details of why this is desirable, and when it is feasible and/or economical, will be given in Section 2.5.

1.1.2 Input representation

The descriptions of the sets of facts and goals will consist of pairs <pattern, number>, where each pattern represents a typical member of some set of facts or goals. For a persistent program, such as a database system, the numbers will be frequencies, representing (say) the expected numbers of inputs per day that will match each pattern. For an ephemeral program, whose storage is reclaimed after a finite running time, the numbers can be the absolute sizes of the sets of facts or goals matching the patterns. We shall rather loosely refer to the elements of F and G as either "patterns" or "sets" depending on whether or not we are concerned with the number of facts or goals that match the pattern.

The "matching" between patterns from F or G and actual facts or goals is not quite the same as the pattern-matching normally used in inference systems. Predicate symbols, and constants in

argument positions, match only themselves, as usual. We distinguish two kinds of variables in the patterns. A "free variable" can appear only in a goal pattern, and matches only against variables in the actual goal, rather than constants (and each occurrence of it must be matched by the same variable, so that pattern $P(x, x)$ will not match goal $P(y, z)$). A "bound variable" may appear in either fact or goal patterns, and cannot match a variable in an actual goal or fact, but can match any constant. Each occurrence of a bound variable must be matched by the same constant, so that $P(X, X)$ could match $P(1, 1)$ but not $P(1, 2)$. Patterns having the same predicate symbol and agreeing on their constant arguments, but differing as to which of their variables are bound, are treated as distinct. We use single lower-case letters to denote free variables in patterns, and single upper-case letters to denote bound variables; there is no connection between free and bound variables represented by upper- and lower-case versions of the same letter.

Constants in a pattern can, of course, match only themselves in a goal or fact. The appearance of a pattern containing a constant means not only that this constant is expected to appear in some set of facts or goals (the same pattern with a bound variable in place of the constant would allow for that), but also that there is some additional information specifying the frequency with which it will appear, which will presumably differ from the average.

For example, if F contains the pattern

$$Bought(Widget, N, B)$$

with the number 20 attached, this indicates that there will be twenty facts, each containing the name of some buyer B who bought some number N of widgets, but it says nothing about the sizes of orders for widgets. The goal pattern

$$Bought(X, 500, y)$$

with the number 1, means that there will be a single goal giving the name of an item X and asking for the name of everyone who bought 500 units of it.

1.2 Notation

We define a directed graph, called the *rule graph*, which has a node for each member of R and an arc from a rule r to a rule s iff r's premise is unifiable with one of s's premises. We say that s is a *successor* of r, and r a *predecessor* of s. Recursion in the ruleset would be reflected by a cycle in the rule graph. We require the rule graph to be acyclic, since the work we have done does not include techniques for dealing with cycles in it. Such techniques are being investigated by many researchers [NH80, MS81, HN84, Ull84].

There will be times when we add F and G to the rule graph, in the obvious places: a fact pattern from F is the predecessor of those rules having a premise that unifies with it, and a goal pattern from G is the successor of those rules whose conclusions unify with one of its conjuncts. In Figure 1, fact patterns are at the bottom and goal patterns at the top.

A complete set of directions and premise orderings for all rules r in R will be called a *strategy*. We shall call a strategy *coherent* if all predecessors of a forwards rule are used forwards. This also

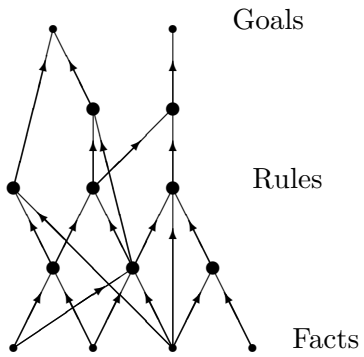

Figure 1: A moderately tangled rule graph

implies that if a rule is used backwards, so are all its successors. Thus a coherent strategy can be visualised as a "cut" in the rule graph, with the rules "below" the cut being used forwards and those "above" it backwards. When a set of rules is being used under a coherent strategy, forward and backward inference are coupled only via the set of facts inferred by the forward rules, which can be stored and used as the basis for backward inferences. We assume that answers inferred backwards are not stored, since it is not known whether they will be needed again (this is discussed further below). Coherence thus corresponds to a simple, intuitive model of how forward and backward inference might be used together. No claim is made that the optimal strategy for a problem will necessarily be coherent.

1.2.1 Rule costs

For a rule r we define the cost of using r forwards, written $c_f(r)$, to be the time taken to deduce all the facts that could be concluded by r, if the conclusions of r's predecessors had already been generated and were available for immediate retrieval. We define the backwards cost $c_b(r)$ to be the time taken trying to prove all the goals that would be fed to r from its successors (which could be rules from R or goal sets from G). Again, $c_b(r)$ is to be computed under the assumption that all of the conclusions of r's predecessors are available.

It will always be the case that $c_f(r)$ depends on the structure of R and on the sizes of the fact sets in that part of F from which r's inputs come; $c_b(r)$ can depend both on these and on that part of G from which the goals to be proved by r arise. Of course, both $c_f(r)$ and $c_b(r)$ will depend on the order of the premises of r; $c_b(r)$ will also depend on the premise orders of all rules reachable from r in the rule graph, and on their directions. We can eliminate the dependence on directions by insisting on a coherent strategy.

1.2.2 Expenses of premises

We can also split up the cost of deduction in a different way. Define the *expense* of a premise p to be the cost of all the work needed to find the answers to p by backward inference and/or direct lookup, without assuming that any facts are available for lookup except for the facts described by

F and the conclusions of any rules that are definitely known to be used forwards. The expense of p thus may include the cost of long chains of deduction, and is always a function of the directions of the rules "below" the rule or goal mentioning p, so it should be regarded as an implicit function of a strategy. The expense of a set of literals described by a pattern-number pair as specified above, is just the number times the expense of a typical literal matching the pattern. The sum of the expenses of all goal sets in G gives the total cost of all backward inference done under the strategy. The expenses of the premises in a conjunction are also important because they determine the optimal order in which to solve them.

The estimation of computational costs is enormously simplified if we assume that the expense of a premise is independent of whether it has been solved before as a goal or subgoal, since different rules that may require some of the same facts as inputs can be treated independently of each other. But this assumption holds only if the results of backward inference are not stored, since this might make it much cheaper to solve a premise that had already been solved once. It might even make it more expensive, if the stored answers were retrieved and the inference also performed *de novo*.

1.3 Structure of this article

In Section 2 we prove the NP-completeness of conjunct ordering on its own, and survey some previously published algorithms for ordering the conjuncts of a single goal. We note the conditions under which these algorithms can be trusted, and give one that works in the most general case we consider. In Section 2.5 we describe how a suitable conjunct ordering algorithm can be used to obtain the optimal orderings for all the rules if all inference directions are already known.

In Section 3 we give a brief review of the results of [TG87] concerning the choice of directions for rules once the orderings of their premises are known.

In Section 4 we give an intuitive argument that conjunct ordering and direction choice should not be done separately, and an example of what can go wrong if they are; a formal proof that the combined optimisation problem is NP-complete (even in a case where either subproblem on its own can be solved in polynomial time) is relegated to the Appendix.

In Section 5, we show that all is not lost: a conjunct ordering algorithm can be invoked as part of a search of the space of sets of rule directions. Once the directions have thus been found, the correct order of the conjuncts for each rule will be returned by one last application of the ordering algorithm.

Section 6 briefly discusses alternatives to the algorithmic approach adopted here, and Section 7 addresses some possible glitches and minor technical points.

2 Conjunct Ordering

We begin by examining the simplest case of finding the optimal order for the conjuncts of a single conjunction. In this case, all the input facts are available by lookup at an expense of 1 unit per fact. Even this yields an NP-complete problem, as we shall prove. We then review an algorithm

that solves this problem exactly and go on to discuss two approximate algorithms for a more general version of the problem, in which the conjuncts can have different expenses. We propose an algorithm, based on best-first search, for solving this more general problem exactly, and prove a result that offers an increase in speed over simple best-first search. Finally, we show that the optimal conjunct orderings can be determined for a set of rules, provided that the inference directions for them have already been found.

2.1 The simple conjunct ordering problem

Under the assumption that the computational cost of solving a conjunction is just the cost of looking up answers to conjuncts, or is linear in this cost, the cost of solving a set of conjuncts in a given order can be expressed in terms of the number of answers to each conjunct and the "selectivity" relationships between different conjuncts. Our treatment of this is adapted from that in [SG85], but with two further assumptions.

2.1.1 Assumptions

We define the *selectivity factor* of a premise and a list of bindings for some of its variables to be the proportion of the answers to the premise that agree with the bindings in the list. For example, if there are 150 facts like $Parent(A, B)$ and Jones has 6 children, then the selectivity factor of the premise $Parent(x, y)$ and the binding of Jones to x is 0.04. We shall assume that, when more than one variable is bound, this factor is a product of individual selectivity factors for the variables that are bound. This is more formally designated as the *Argument Independence Assumption*, namely that the probability that a randomly chosen answer to the premise will agree at one argument position with a randomly chosen binding value is independent of the bindings for other arguments. This is quite a strong simplifying assumption. In terms of the example used earlier, it would imply that all Senators had the same number of daughters.

We shall also assume that the selectivity factor for each variable is independent of the identity of the premise(s) that bound it. Thus the selectivity factor is determined solely by the identity of the variable, or rather the identity of the domain over which it ranges. This will hold true if there is no "clumping" (preferential occurrence of one value over another) in the constants that occur in the run-time data, or, alternatively, if the occurrences of values from a given domain are distributed the same way for every bound variable taking values from that domain (i.e. everybody clumps in the same way). We call this the *Uniform Data Assumption*.

We denote the selectivity factor for a variable v by $S(v)$.

2.1.2 Cost of an ordering

We shall assume that the number of answers to each conjunct p is known, and denote it by $\|p\|$. If $p_1 \dots p_n$ is a sequence of conjuncts, write $\|p_1 \dots p_n\|$ for the expected number of answers to it. Denote by p_i^j the result of substituting into p_i the bindings from a solution to $p_1 \dots p_j$, where $j < i$.

For a sequence of length N we get

$$\begin{aligned} \|p_1 \ldots p_N\| &= \|p_1 \ldots p_{N-1}\| \|p_N^{N-1}\| \\ &= \prod_{i=1}^{N} \|p_i^{i-1}\| \end{aligned} \tag{1}$$

where we define $\|p_1^0\| = \|p_1\|$. We also have

$$\|p_i^{i-1}\| = \|p_i\| \times \prod_v S(v) \tag{2}$$

where the product is taken over all variables v that occur both in p_i and in $p_1 \ldots p_{i-1}$.

Let $Expense(p_i^{i-1})$ be the expense of solving one instance of p_i^{i-1}, and $ExpPerAns(p_i^{i-1})$ be this expense divided by $\|p_i^{i-1}\|$. The cost of solving $p_1 \ldots p_N$ as a conjunctive goal is given by

$$Cost(p_1 \ldots p_N) = \sum_{i=1}^{N} \|p_1 \ldots p_{i-1}\| Expense(p_i^{i-1}) \tag{3}$$

$$= \sum_{i=1}^{N} \|p_1 \ldots p_i\| ExpPerAns(p_i^{i-1}) \tag{4}$$

When all the $ExpPerAns$ values are 1, this reduces to

$$Cost(p_1 \ldots p_N) = \sum_{i=1}^{N} \|p_1 \ldots p_{i-1}\| \tag{5}$$

2.1.3 Complexity

Theorem 1 *Given a set of conjuncts p_i, the problem of finding a permutation of it that has minimal Cost as defined above is NP-complete even under the Argument Independence and Uniform Data Assumptions.*

Proof: The proof proceeds by relating the quantity $\|p_1 \ldots p_i\|$ to the set of variables that appear in both $p_1 \ldots p_i$ and $p_{i+1} \ldots p_N$. The problem of arranging the conjuncts so as to keep the size of this set below some bound is shown to be at least as hard as a known NP-complete problem.

The NP-complete problem which we reduce to this problem is "Minimum Cut Linear Arrangement", found at [GJ79, page 201]. This problem is stated as follows: given a graph G with N vertices, and an integer K, find whether there is an ordering function f from the vertices to the integers $1 \ldots N$ such that, for every i with $1 < i < N$, the number of edges (u, v) of the graph such that $f(u) \leq i < f(v)$ is at most K. Suppose we are given an instance of this problem. Reduce it to a conjunct ordering problem as follows:

Without loss of generality, there is no edge connecting a node to itself. For each edge in G, create a variable, and for each vertex, a predicate symbol. The conjunct corresponding to each vertex consists just of the vertex's predicate symbol and the variables of the edges that are incident to it. If a vertex has degree d, let the number of answers of its conjunct be N^d. For all variables v let

$S(v)$ be N^{-2}. We shall prove that there exists an ordering of the query with *Cost* less than N^{K+1} iff there exists a linear arrangement of the kind demanded. If we could find the optimal conjunct ordering in polynomial time then we could certainly determine whether an ordering with cost less than N^{K+1} existed, also in polynomial time, since evaluating the cost of an ordering certainly takes polynomial time using equation 5.

Define $Cut(i)$ to be the number of edges (u, v) in the graph such that $f(u) \leq i < f(v)$. Observe that, for a given ordering of the vertices in the graph (terms in the query), $Cut(i)$ precisely the number of variables which appear just once in $p_1 \dots p_i$. This is a consequence of the problem reduction, because every variable appears exactly twice in the query (each arc has exactly two endpoints). Also let $W(i)$ be the number of variables that appear twice in $p_1 \dots p_i$. Then $2W(i) + Cut(i)$ is just the total number of variable occurrences in $p_1 \dots p_i$, and since the arity of each predicate has been made equal to the degree of the corresponding vertex, $2W(i) + Cut(i)$ is also equal to the sum of the degrees of the first i vertices.

Now from equations 1 and 2 we get

$$\|p_1 \dots p_i\| = \prod_{j=1}^{i} \|p_j\| \prod_{w} S(w)$$

where the product is over all variables w appearing twice in $p_1 \dots p_i$. Recalling that the number of answers to a conjunct is N^d where d is the degree of its corresponding vertex, we see that

$$\prod_{j=1}^{i} \|p_j\| = N^{2W(i)+Cut(i)}$$

Therefore, since $S(w) = N^{-2}$ for all w, for any i from 1 to $N - 1$,

$$\|p_1 \dots p_i\| = N^{Cut(i)}$$

and $\|p_1 \dots p_N\| = 1$.

Thus if the vertices of the graph are arranged so that $Cut(i) \leq K$ for all relevant i, the cost of the corresponding ordering of the terms of the query will, according to equation 3, be at most $1 + (N - 1)N^K$ which is less than N^{K+1}, whereas if there is some i such that $Cut(i) > K$, then the cost will be at least N^{K+1}. This is what we wanted.

Finally, the conjunct ordering problem is no harder than NP-complete, since the time needed to evaluate $Cost(p_1 \dots p_N)$ is polynomial in N. □

2.2 Conjunct ordering with equal expenses

Smith [SG85] has proposed an algorithm for finding the optimal ordering when all facts are available at cost 1, based on a "best-first" search of the space of sequences of subsets of the conjuncts. The search starts with the empty sequence and generates new ones by adding single conjuncts to the cheapest sequence found to date. In order to generate cheaper sequences first, different candidates for p_i would be added to a sequence $p_1 \dots p_{i-1}$ in ascending order of their $\|p_i^{i-1}\|$ values. It might

seem that all conjuncts not already in the sequence would be candidates for insertion at the next position, but various features of the problem allow some possibilities to be ruled out at once. One of the most useful of these is that if

$$\|p_j^{j-2}\| < \|p_{j-1}^{j-2}\|$$

then the sequence obtained by transposing p_{j-1} and p_j will have lower cost than $p_1 \ldots p_j$, and so sequences that have $p_1 \ldots p_j$ as an initial subsequence cannot be optimal. This is called the *Adjacency Restriction*; it means that, if the search algorithm generates a sequence ending in some conjunct p_{j-1}, the next one to be added to this sequence, p_j, should be one that does not satisfy the above inequality. As a corollary, the conjunct p such that $\|p^{j-2}\|$ is maximal should never be added in the $(j-1)$st position.

These two constraints alone allow a large proportion of the possible permutations of all the conjuncts of a rule to be eliminated. In the case of a rule with 8 conjuncts, all but $1,385$ of $8! = 40,320$ possible permutations are ruled out. [SG85] also proves that if the conjunct having the lowest expected number of answers of any conjunct not yet in the sequence is expected to have fewer than 1 answer, then that conjunct can be added next, with at most a trifling increase over the optimal cost. This observation and the Adjacency Restriction lead to vast reductions in the number of nodes generated by the best-first search.

All these results apply only to conjuncts whose expense is equal to the number of answers to them (or rather, linearly related, with the same constant of proportionality for all conjuncts; see [SG85] for a proof that this is sufficient). The Adjacency Restriction in fact applies to any pair of conjuncts having the same marginal expense per answer, and the result about the cheapest conjunct applies if its marginal expense per answer is no higher than that for any other conjunct remaining. When the answers to some conjuncts can be looked up while others must be found by backwards inference, these conditions frequently fail to hold, so the algorithm loses much of its power. Some other useful, though frequently inapplicable, restrictions on choice of the next conjunct to add can be found in [Nat87].

2.3 Conjunct ordering with unequal expenses

Warren [War81] suggests ordering a set of conjuncts by starting with the one that is cheapest to solve, i.e. minimising $\|p_1\|$, and then repeatedly adding the one that is cheapest given the variable bindings from the the preceding ones, i.e. minimising $\|p_i^{i-1}\|$. While this yields an ordering very quickly (its running time is $O(n^2)$), it can easily be shown that there are cases where it does not yield the optimal one. Warren claims that in practice it usually yields a good one. However, examples can be found where the ordering it gives is arbitrarily far from optimal. When 4-coloring a map, all conjuncts initially have the same number of answers, and the "cheapest-first" test would be no better than choosing arbitrarily between them, but in fact it can be very important to choose the correct country to start at. So Warren's heuristic on its own is certainly not adequate.

The problem of optimally ordering conjuncts having unequal expenses has been studied by Ibaraki and Kameda [IK84], who present an algorithm for finding the "optimal" ordering in time $O(n^2 \log n)$, subsequently improved by Krishnamurthy *et al.* [KBZ86] to $O(n^2)$. This algorithm,

though, depends on assumptions about both the set of conjuncts and the final ordering that do not always hold. In particular, it applies only where the query corresponds, as described in the NP-completeness proof above, to a graph that is a tree (and has no "hyperedges", i.e. no variable appears in three or more conjuncts). Most rules that have enough conjuncts for their ordering to be a non-trivial problem do not yield a tree, so that this algorithm is not directly applicable. Krishnamurthy suggests that the graph could be coerced into tree form by deleting some edges, which corresponds to pretending that some variables have $S(v) = 1$.

[KBZ86] also looks only at orderings such that p_i always has at least one variable in common with $p_1 \ldots p_{i-1}$, even though [SG85] points out that the optimal ordering need not have this property. In fairness, the possible optimality of an ordering not having this property seems (although this has not been proved) to be a consequence of an implicit assumption of ours about expenses, namely that an indexing scheme good enough to return only a few candidate answers when given a highly specific query is used. This underlies the belief that the expense of a premise is a linear function of the number of the answers, and is independent of which of its variables are bound. Since large databases do not often index a relation on all possible attributes, it is unlikely that they would satisfy this assumption, and so this restriction on orderings considered may not be serious in the context of query optimisation for such a database.

2.4 An admissible algorithm

A weaker result than the Adjacency Restriction remains true when conjuncts have different expenses per answer. For any two sequences containing the same set of conjuncts, only the cheaper of the two need be retained and expanded further: any sequence that began with the more expensive of these two could be made cheaper by simply substituting the cheaper sequence in its place, and so cannot possibly be optimal. This is in fact an instance of an extremely general principle in search, namely that of all paths from the start state to any given state, only the cheapest is of interest (see [Nil80]). We can do a best-first search of the space of sequences of conjuncts while keeping track of which sets of conjuncts have appeared, and of the costs of the best known permutation of each such set. Should a sequence be generated that turns out to be a permutation of one already seen, we can throw away whichever sequence has higher cost, and if necessary, adjust the costs of sequences that were obtained by extending it.

The main computational cost involved here seems to be that of recognising a permutation of a previously seen sequence, or rather an instance of a previously seen set of conjuncts. The number of such sets is 2^n, where n is the number of conjuncts. To find (or fail to find) some sequence among these sets would require a number of comparisons of sets roughly equal to the logarithm of the number of sets found so far, assuming that the data structure used to store the sets has been well chosen, and this logarithm would be at most $\log(2^n) = n$. If the conjuncts in each set were sorted according to some canonical order, the cost of comparing any two sequences would be a number of conjunct comparisons less than or equal to the length of the shorter sequence, which again is bounded above by n. So the total cost of checking whether a sequence had been seen before would be at most $O(n^2)$ comparisons of conjuncts, which does not seem at all unreasonable if it leads to

a significant reduction in the number of sequences generated.

The number of sequences that are actually generated by the best-first search will usually be less than 2^n, because in non-pathological cases the search will generate only the best few orderings and a few worse ones, so that many possible sets of conjuncts will never appear. Thus the space needed to store the best cost known so far for each set of conjuncts encountered should not be prohibitively large in the average case. The $O(n^2)$ bound found above is also a worst-case, approached only when most of the 2^n possible sets have been generated.

2.4.1 Filtering permutations

This pruning mechanism will work best if at least a few good sequences are generated early, so that others with unnecessarily high cost can be eliminated rapidly. Unfortunately, there seems to be no guarantee that this will happen. To make it more likely, we can use the approximate algorithms of Warren or Krishnamurthy as "filters" on sequences generated: every sequence $p_1 \ldots p_i$ that is not found in the table of sets of conjuncts already seen can be submitted to an approximate algorithm, and if the permutation returned is cheaper, it should be used and the other dropped. The table should still be used, since running either approximate algorithm will take at least as long as looking a set of conjuncts up in the table, and if expenses must be computed that are not already known, it could take much longer.

Since the best-first search will initially generate sequences consisting of cheap conjuncts, and only gradually consider more expensive ones as it is forced to do so, the benefits of using Warren's cheapest-first heuristic as a filter are uncertain, and the tree query algorithm of Krishnamurthy is probably preferable for this purpose. Since the tree query algorithm requires estimating the expense of several conjuncts, which is likely to take much longer than looking up sets of conjuncts, it seems best to do this filtering only rarely, such as when a conjunct that was not in any previously seen sequence has just been added. This would take care of the main weakness of the best-first algorithm, which is that it will postpone adding an expensive conjunct until it has to, although it might in fact be better to "grasp the nettle" early.

We shall refer to the conjunct ordering algorithm so obtained as "filtered best-first search".

2.4.2 The subset test

Note that the best-first search will often generate a sequence of conjuncts without generating every subset of that sequence. Indeed it had better do so, since otherwise it will generate at least 2^n subsequences while trying to order a set of n conjuncts, and while this is not as bad as $n!$, it is slow for n greater than about 5.

It is tempting to replace the test for set equality between pairs of sequences with a test for containment: if the conjuncts in a costly sequence are a subset of those in a cheaper one, then it seems that the shorter sequence should be discarded. This turns out to be legitimate only under both the Argument Independence Assumption and another assumption, namely that no constant which appears in some conjunct of a pattern representing a conjunctive subgoal can also appear

in one of the rules that could be used to solve that conjunct, or in a pattern representing a set of facts that contribute to its solution. We call this the *Equal Opportunity Assumption*: no rule or fact pattern shall have a better chance of participating in the solution of some goal pattern because of the appearance of a known constant.

Note that the appearance of a constant in a pattern means far more than its appearance in an actual goal or fact – after all, if no constant could appear in both goals and facts, then most goals would be insoluble. But we are using bound variables in goal and fact patterns to represent most occurrences of constants in the actual goals and facts; the appearance of an explicit constant, rather than a bound variable, means that that constant value is expected to appear more often (or perhaps less often) than most others, and so it is not surprising that results depending on inequalities between numbers of answers should fail to hold in the presence of such expectations.

Lemma 1 *Given a conjunct C to be solved by some combination of backward inference and lookup using a rule set R and a set of facts described by a description F, and given a strategy for using R, then if C, R, and F satisfy the Equal Opportunity Assumption and the facts satisfy the Argument Independence Assumption, the expense per answer of C does not decrease when some of its arguments change from free to bound variables or from bound variables to constants, and its total expense does not increase.*

Proof: The proof for the case where a free variable changes to a bound one is easy. Without loss of generality, suppose that there is only one variable to be changed from free to bound. Consider the clause patterns generated in estimating the expense of the conjunct C_1 with the free variable, and those that would be generated by doing the same inferences starting with a conjunct C_2 that was identical to C_1 except in having a bound variable V in place of one of C_1's free variables. The estimated set of answers to C_2 would clearly be smaller by a factor $S(V)$, since it would have had to unify at some point with a bound variable of some fact pattern, or perhaps with a constant. Of the other sets of partial answers generated on the path to the answer, those coming before this unification will have the same size in each case, and those after will be smaller if the variable was bound initially. So the expense of C_2 will be at least $S(V)$ times the expense of C_1, but not greater than it (since every inference operation done to solve C_2 would also be used in the solution of C_1), and the number of answers will be exactly $S(V)$ times as large, whence the expense per answer cannot decrease. This is illustrated in Figure 2.

It is now clear why the Equal Opportunity Assumption was needed: a constant appearing in both C_2 and R or F could unify with itself during the solution of C_2, so that the selectivity factor that would apply if it had been unified with a bound variable would be replaced by 1.0. But if it will only unify with variables (bound or free) and with different constants, then the factor will either be zero (representing a failure of unification) or else equal to that for a bound variable. So if C_2 has a constant where C_1 has a bound variable, a similar argument to that given above will apply: the clause sets generated starting from C_1 and C_2 will be the same size up to some point, after which the sets on the C_2 side will be smaller by some factor (which may be zero). So the total expense is smaller for C_2 and the expense per answer is larger. □

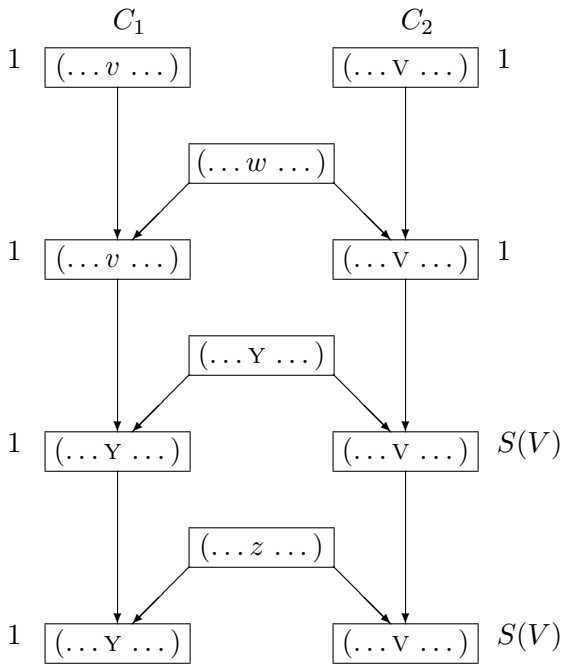

Figure 2: Bound versus free variable

Lemma 2 *Given R, F, and a strategy, as in the above Lemma, with the facts described by F satisfying the Argument Independence Assumption, and a rule r such that the Equal Opportunity Assumption is satisfied with respect to $R - \{r\}$ and F by every premise of r, then if a sequence $p_1 \ldots p_m$ of premises of r is an initial subsequence of the optimal ordering of these premises, but has a higher cost than another sequence $q_1 \ldots q_n$, and the set $\{p_1 \ldots p_m\}$ is a subset of $\{q_1 \ldots q_n\}$, then*

$$\|p_1 \ldots p_m\| > \|q_1 \ldots q_n\|$$

Proof: We can assume that $q_1 \ldots q_n$ is the cheapest sequence satisfying this condition, in the sense that if we were to remove from it any conjuncts not found in $p_1 \ldots p_m$ we would cause an increase in its cost. This implies, in particular, that q_n is one of the p_i. Byassuming also that

$$\|p_1 \ldots p_m\| \leq \|q_1 \ldots q_n\|$$

we shall obtain a contradiction.

Consider the sequence $q'_1 \ldots q'_m$ obtained by removing from $q_1 \ldots q_n$ all conjuncts that do not appear in $p_1 \ldots p_m$, without changing the relative order of those that remain. This sequence is a permutation of $p_1 \ldots p_m$ and so has the same number of answers as it and, by the assumption of optimality, costs at least as much as it, whence it costs more than $q_1 \ldots q_n$. Let $k(i)$ for i from 1 to m be the position in $q_1 \ldots q_n$ of q'_i, so that $q_{k(i)}$ is the same as q'_i, and $k(m) = n$. Referring to equation (4), we can compare the terms in $Cost(q'_1 \ldots q'_m)$ involving q'_i with those in $Cost(q_1 \ldots q_n)$

involving $q_{k(i)}$. Since $q_1 \ldots q_{k(i)-1}$ binds at least as many variables as $q'_1 \ldots q'_{i-1}$, by Lemma 1 the term in $q_{k(i)}$ cannot be less, so the only way for $q_1 \ldots q_n$ to cost less than $q'_1 \ldots q'_m$ is for at least some of the $\|q_1 \ldots q_{k(i)}\|$ factors to be less than the corresponding $\|q'_1 \ldots q'_i\|$. However, by assumption,

$$\|q_1 \ldots q_n\| \geq \|p_1 \ldots p_m\| = \|q'_1 \ldots q'_m\|$$

Let j be the largest integer such that $\|q_1 \ldots q_{k(j)}\| < \|q'_1 \ldots q'_j\|$. Clearly $j < m$; consider the sequence formed by removing from $q_1 \ldots q_n$ only those conjuncts appearing later than $q_{k(j)}$ that are not in $p_1 \ldots p_m$. We shall show that this sequence, which we shall write (abusing notation slightly) as $\|q_1 \ldots q_{k(j)}, q'_{j+1} \ldots q'_m\|$ is cheaper than $q_1 \ldots q_n$, contrary to our assumption. For its first $k(j)$ conjuncts, it is identical to $q_1 \ldots q_n$. Thereafter, the conjuncts that appear in it are no more bound than their counterparts in $q_1 \ldots q_n$, so that their *ExpPerAns* is no higher; all we now need is to show that the number of answers produced at such a conjunct, which we may write $\|q_1 \ldots q_{k(j)}, q'_{j+1} \ldots q'_i\|$ for some $i > j$, is no more than $\|q_1 \ldots q_{k(i)}\|$. The conjuncts from q'_{j+1} to q'_i in this sequence are at least as bound as they are in $q'_1 \ldots q'_m$, so they have at most as many answers. Hence (using equation 1)

$$\frac{\|q_1 \ldots q_{k(j)}, q'_{j+1} \ldots q'_i\|}{\|q_1 \ldots q_{k(j)}\|} \leq \frac{\|q'_1 \ldots q'_i\|}{\|q'_1 \ldots q'_j\|}$$

But, by the definition of j,

$$\frac{\|q'_1 \ldots q'_i\|}{\|q'_1 \ldots q'_j\|} < \frac{\|q_1 \ldots q_{k(i)}\|}{\|q_1 \ldots q_{k(j)}\|}$$

so $\|q_1 \ldots q_{k(j)}, q'_{j+1} \ldots q'_i\| < \|q_1 \ldots q_{k(i)}\|$ and we are done. □

Theorem 2 *Given R, F, and a strategy, as described, with the facts described by F satisfying the Argument Independence Assumption, and a rule r such that the Equal Opportunity Assumption is satisfied with respect to $R - \{r\}$ and F by every premise of r, then if a sequence $p_1 \ldots p_m$ of premises of r has a higher cost than another sequence $q_1 \ldots q_n$, but the set $\{p_1 \ldots p_m\}$ is a subset of $\{q_1 \ldots q_n\}$, then $p_1 \ldots p_m$ cannot be the beginning of an optimal sequence of the premises of r, and so can be ignored in the search for such a sequence.*

Proof: The proof is by contradiction, starting with the assumption that $p_1 \ldots p_m$ is the beginning of an optimal sequence. We saw in Lemma 2 that $p_1 \ldots p_m$ has more answers than $q_1 \ldots q_n$. Now, in the putatively optimal sequence $p_1 \ldots p_N$ of which $p_1 \ldots p_m$ is the beginning, let $m_1 > m$ be minimal such that p_{m_1} is not among $q_1 \ldots q_n$. The conditions of Lemma 2 are satisfied by $p_1 \ldots p_{m_1-1}$ and $q_1 \ldots q_n$, unless these two sequences are permutations of each other (but then, since $q_1 \ldots q_n$ is the cheaper of the two, $p_1 \ldots p_{m_1-1}$ cannot be the beginning of an optimal sequence). So if we set $q_{n+1} = p_{m_1}$ and compute the costs of the sequences $p_1 \ldots p_{m_1}$ and $q_1 \ldots q_{n+1}$ using equation 3, since $p_1 \ldots p_{m_1}$ binds no more variables than $q_1 \ldots q_{n+1}$, we find (applying Lemma 1) that the q sequence is cheaper.

But now Lemma 2 applies also to this pair of sequences, which are strictly longer than the original ones. Since we clearly cannot repeat this extension more than finitely many times before obtaining a pair of sequences that are just permutations of each other, it follows that the Theorem is true. □

So the filtered best-first search algorithm can safely replace the test for set equality of sequences of conjuncts by a containment test, under the assumptions we have stated. The sets of conjuncts corresponding to sequences already seen can be searched quite fast if suitable data structures are used, so that this result does offer a real prospect of speeding up the algorithm.

2.5 Conjunct ordering with known rule directions

Here we show that it is possible to find the optimal conjunct orderings for a set of rules once we know the directions in which they will be used, provided that a backwards rule is allowed to have a distinct conjunct ordering for different subgoal patterns that can invoke it. We shall assume that the directions make up a coherent strategy, as defined in Section 1.2.

2.5.1 Ordering forwards rules

If a rule r is used forwards, then by coherence r's predecessors are used forwards too, and so r's inputs are all going to be available in the database and can be looked up at known expense per answer. So we can apply the algorithm of [SG85], or the filtered best-first search if necessary, and immediately obtain the optimal ordering of r's premises, valid over all strategies in which r is used forwards.

2.5.2 The need for multiple orderings for backwards rules

The backwards case is more complicated. It may easily happen that a rule is invoked many times in different ways, to solve different subgoals. These different invocations would lead to different instantiations of the rule's premises. Since the expenses, and expenses per answer, of these different instantiations can vary widely, it is likely that the optimal conjunct orderings would vary between different invocations.

In particular, if the Equal Opportunity Assumption is contravened, a rule may be invoked by two goals from the same set (i.e. they both match the same pattern) that contain different constants and would, ideally, be solved using different permutations of the rule's premises. Obviously, the way to achieve this ideal behaviour is by examining each such goal at run-time and choosing the correct ordering. Since we have restricted ourselves in this article to compile-time optimisations only, we must generate orderings with an eye to minimising the total expense of solving the collection of subgoals that will be presented to a rule by one of its successors.

Note that we cannot optimise for the entire "workload" of a backwards rule, because we do not know it soon enough. If rule r_1 has successors r_2 and r_3, then to order the premises of r_2 we need to know the expense of the one that will be solved by r_1. This is in turn determined by the order of r_1's premises, which we have to choose before we know what goals will be presented to r_1 by r_3. This faces us with an unpalatable choice. Either we must tie together the conjunct ordering tasks for all of r_1's successors, evaluating a list of orderings for all the successors with respect to the set of subgoals it would generate for r_1 to solve and the consequent ordering of r_1's premises, or we must produce several conjunct orders for r_1 as it is invoked by different successors with different

sets of goals. The first choice would lead to an unacceptably complicated and expensive ordering algorithm, which would moreover generate "compromise" orderings for r_1's premises that would not necessarily be even close to optimal for any of the individual goals that r_1 would solve. The second would in effect generate several different versions of r_1, each of which would be seen by each of r_1's predecessors as another successor, and the number of different orders generated lower down the rule graph could grow rapidly.

One way to cope with this is to restrict the variety of workloads for which we are willing to generate orders for a rule's premises. We may, for example, divide the possible patterns invoking a rule into equivalence classes based on what set of variables are bound, ignoring any constants that appear in the patterns. This will often suffice to keep the numbers manageable, since there will usually be only a few sets of variables that can be left unbound without driving the expense absurdly high and eliminating the pattern from consideration. By ignoring constants we also free ourselves to use the subset test that was justified in Theorem 2. This will, however, sacrifice some potentially valuable information, if it was desired that the optimiser should take account of the differing frequencies of occurrence of different constants.

To ensure that the different versions of each rule are used correctly at run-time, we can use a simple syntactic device, which infringes slightly on the letter, though not the spirit, of our restriction against modifying the vocabulary with which the rules are written. Namely, for each different combination of bound and free variables with which a literal may appear as a goal, generate a variant of the name of its predicate (perhaps by suffixing it with a suitable string of b's and f's to indicate bound and free arguments), and use this new predicate in the conclusion of a new version of the rule whose premises are ordered optimally for that set of bound variables. Also replace each occurrence of the old predicate name in a premise of a backwards rule with the correct new name, relative to the variables that will be bound when it comes to be solved at run-time. If the goals from G cannot be re-written in this way, then one small piece of run-time intervention will be necessary, looking at each top-level goal and checking which of its arguments are bound before sending it to the appropriate rule(s) for solution.

2.5.3 Orderings for backwards rules

We can now choose conjunct orderings for the backwards rules in a coherent strategy quite simply, by working recursively from the top down as outlined above. To find the best ordering for the premises of a given rule working on a given set of goals/subgoals, run the filtered best-first search algorithm over it. Whenever this algorithm needs to know the expense of some premise that cannot be solved by direct lookup, make a recursive call to find the conjunct orderings for the rules that will be used to solve the premise, and then compute its expense by equation 3 or 4, with the right variable bindings substituted into $p_1 \ldots p_N$. As long as there are no cycles in the rule graph, this recursion will always ground out at premises whose answers are found in F or deduced by forwards rules.

At some cost in accuracy, we may be able to reduce the number of times the conjunct ordering algorithm is applied to each rule. Recall that Lemma 1 showed that, under the Argument

Independence and Equal Opportunity assumptions, the expense per answer of a conjunct never decreases, and its total expense never increases, when some of its arguments change from variables to constants. Since the number of answers to a conjunct is independent of conjunct ordering and can be estimated without any search, this lemma gives us both lower and upper bounds on the expense of a conjunct in terms of others similar to it. For example, if the variables x and y both range over domains of size 10, and $P(x, y)$ is expected to have 30 answers and a total expense of 75 units, then $P(X, y)$ will be expected to have $30/10 = 3$ answers, and its expense per answer must be at least $75/30 = 2.5$, for a total expense of at least 7.5. If we have also estimated the expense of $P(X, Y)$ to be 1.8 units (for an expected $30/100 = 0.3$ answers) then we can bound the expense of $P(X, y)$ above by 18 units.

If we use the upper bounds so obtained during the choice of orderings for the premises of rules, we will get a set of orderings that will not necessarily be optimal, but we can be sure that if we use them, the expected cost of the computation will be no higher than that predicted by equation (3) or (4) using the upper bounds on expenses. This may be adequate for some purposes. We shall see below that it can also be useful to run the ordering algorithms using lower bounds on expenses.

Anyway, we can now find the optimal conjunct orderings for any set of backwards rules whose inputs are available by lookup, and this can be dovetailed with the above result for forwards rules to give the optimal orderings for any coherent set of directions.

3 Choosing directions for rules

Of course, the algorithms of the previous Section will be of no use unless we can find directions for the rules somehow. The reverse result to the one just mentioned, that directions can be found easily given an ordering (or several orderings) for the premises of each rule, and subject to the coherence constraint, is contained in [TG87]. We briefly review that work here.

3.1 Total cost of a set of directions

The estimation of the computational cost for a strategy on a set of rules, facts, and goals described as in Section 1.1 is done according to equations (1 – 4). It also requires, for each premise that unifies with some pattern from F and each conclusion that unifies with some pattern from G, an estimate of the number of facts or goals from the set described by that pattern that will actually so unify at run-time. If we are using the Argument Independence and Uniform Data Assumptions, we can obtain these numbers by taking the product of the set size from F or G with the selectivity factors for the bound variables that become bound to each other or to constants during the unification; otherwise, we must assume that more detailed information on the selectivity of these unifications is available.

The cost of forwards inference is then estimated by iterating up the rule graph, starting from the rules whose immediate predecessors are all in F. Equation 4, with all the $ExpPerAns$ values set to the marginal cost of retrieving one more fact, gives the cost of using such a rule forwards; the sequence $p_1 \ldots p_N$ is just the premises of the rule. To process rules whose predecessors are not

in F, all we need is the descriptions of the sets of facts matching each of their premises, and this can be found by applying equation 1 to the predecessors.

For backwards inference, we start at the top of the rule graph, with the rules whose immediate successors are in G. For each goal set that can invoke a rule, we take our estimate of the number of goals from the set that will unify with the rule's conclusion, and multiply it by the cost of solving the conjunction of the rule's premises under the bindings from this unification. This cost is given by equation 3. The computation is not as easy as in the case of forwards inference, because some of the $Expense(p_i^{i-1})$ values will not be known, namely those for premises p_i that are solved using other backwards rules. We obtain these expenses by recursively evaluating the cost of a goal set with pattern p_i^{i-1} and number $\|p_1 \ldots p_{i-1}\|$ in the same way; since there are assumed to be no cycles in the rule graph, this recursion always grounds out on facts either described in F or deduced by forwards inference.

Adding up the cost of forwards inference and the cost of each goal set then gives us the total cost of a strategy (in this context, a *strategy* is simply a set of directions for all the rules).

3.2 Decomposing the total cost

The ability to compare any two strategies and see which is cheaper, while theoretically adequate for finding the best strategy, is not a very powerful tool for searching the space of coherent strategies. We would prefer to have a means of pruning this space, for example by being able to determine that any strategy which uses a certain rule forwards is cheaper than a strategy obtained from it by reversing only the direction of that rule (though such a comparison may not be directly useful, since one or other strategy may well be incoherent). To do such pruning, we need to decompose the total cost of a strategy into a sum of costs for individual rules, and have the cost of using a given rule in a given direction be the same across a wide class of strategies.

Such decomposition is generally possible only with respect to coherent strategies. For if a forwards rule has a predecessor that is used backwards, then the backwards rule will be run when the forwards rule is, quite independently of whether any other rules are interested in the results of the forwards rule. Thus part of the cost of the backwards rule will be independent of the goals solved by the program, whereas if all the rules above it in the rule graph were used backwards, the entire cost of using the rule clearly would depend on the goals. So the cost of using a rule backwards depends on the direction of its successor, and therefore on the strategy within which the rule is used. This dependence is eliminated by the coherence condition, because the direction of a backwards rule's successors becomes known.

The cost of using a rule backwards can also depend on the directions of its predecessors, since looking up a set of answers to a premise of a rule can have different cost from invoking a rule to find those answers (even though we are, for this purpose, ignoring the cost of *running* the other rule). In [TG87] a method of allocating the cost of computation is described that eliminates this dependence, by transferring part of the cost of using a rule onto its predecessors. There it is shown that, provided no facts can be proven in more than one way (by different rules, or the same rule

from different sets of input facts), the cost of a coherent strategy can be decomposed into a sum of strategy-independent costs of individual rules.

3.3 Finding the best strategy

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, if we define an indicator variable $v(r)$ for each rule r, with a value of 1 to denote using the rule forwards, and 0 for backwards, the total cost of a strategy (represented as a set of values for the $v(r)$) is

$$\sum_{r \in R} (v(r)c_f(r) + (1 - v(r))c_b(r))$$

and the coherence constraint on rule directions turns into the inequality $v(r_1) \geq v(r_2)$ for each predecessor r_1 of r_2. If this expression for the cost is minimised subject to these inequalities, the resulting values of the variables $v(r)$ give the optimal solution to the original problem.

This integer program can be shown to be 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. Moreover, a simple network-flow algorithm, rather than general linear programming techniques, can be used to solve the problem quite rapidly.

3.4 Duplicate answers

When some facts can be proved more than once, the problem of choosing rule directions optimally is decomposed into a best-first search of the space of sets of directions for rules that may be involved in such duplicate proofs, and a linear program to choose the directions of the remaining rules. Linear programming can also be used to attempt to constrain the directions for the rules that do give duplicate answers, thus sometimes shortening the best-first search; the details can be found in [TG87].

4 Conjunct Ordering with Changing Directions

Unfortunately, it is difficult to be sure of the conjunct orderings that will be used in the optimal strategy, so that the result of [TG87] is not directly useful. The cause of this is that conjunct ordering depends on knowing the expenses of a rule's premises, which are determined by (among other things) rule directions, and rule directions are determined by rule costs which depend on conjunct orderings. We now describe how this interdependence can cause a simple hill-climbing algorithm based on alternating between direction choice and conjunct ordering to get stuck at a local optimum. This is no accident: the combined optimisation of directions and orderings is NP-complete, so that no combination of fast algorithms could be expected to solve it.

4.1 Local Improvements

What we would like to be able to do is start with some set of presumed expenses for all literals, obtain orderings of the premises of each rule, and then run an algorithm from [TG87] to determine the direction of each rule. We could then find the optimal conjunct ordering(s) for each rule as explained in Section 2.5. But since the newly found conjunct orderings would affect the rule costs on which the directions were based, we would then have to re-compute the directions, completing a feedback loop. It is important to know whether this feedback would be positive or negative. If we took the truly optimal strategy, perturbed it by swapping two adjacent conjuncts in one rule, and then ran the direction choice algorithm over it followed by a conjunct ordering algorithm, would this give us back the optimal strategy? Or would it give both sub-optimal directions and sub-optimal conjunct orderings?

Elementary economic theory suggests that the feedback will be positive, i.e. will take an imperfect strategy to a worse one. If some input to a rule becomes cheaper, presumably the conjunct ordering will change so that more of it will be demanded and used. An increase in demand for answers to some subgoal will then cause increases in the estimated costs of using backward inference in its proof, and so may lead to some rule directions being changed from backwards to forwards; this is analogous to economies of scale in production, and would lead to a decrease in the marginal expense of proving the subgoal, since the forward inference is only done once and resembles a fixed cost. Lower marginal expenses then cause the input to appear even cheaper, reinforcing the initial change.

4.2 The invisible handcuff

For an example of how market forces can lead to a non-optimal distribution of effort, consider the rules

$$\begin{aligned} A(x,y)\ \&\ B(y,z) &\Rightarrow P(x,y,z) \\ C(x,y)\ \&\ B(y,z) &\Rightarrow Q(x,y,z) \\ T(x,y)\ \&\ U(x,y) &\Rightarrow A(x,y) \\ V(x,y)\ \&\ W(x,y) &\Rightarrow B(x,y) \\ Y(x,y)\ \&\ Z(x,y) &\Rightarrow C(x,y) \end{aligned}$$

All variables have selectivity factors of 1/8. There are 32 facts for each of the six predicates T, U, V, W, Y, Z, and two sets of 7 goals, with patterns $P(K, y, H)$ and $Q(K, y, H)$, where K and H are bound variables.

We shall show that, with these numbers, the optimal strategy is to infer B forwards and all the other predicates backwards, and to solve B first in the rules for P and Q. But we shall also see that taking the rules exactly as written and inferring A and C forwards gives a locally optimal strategy, in the sense that any single change to either the ordering of conjuncts or the directions of rules would lead to an increase in the total estimated cost. To find the optimal strategy by starting from

this one would require a co-ordinated change to both directions and orderings, which is outside the scope of the simple hill-climbing algorithm described above.

We shall adopt a simplified model of deduction costs, in which only the cost of generating clauses is considered, and this cost is assumed to be 1 unit per clause independent of length or other measures of complexity. This is done purely for illustrative purposes. Also for simplicity, we begin by considering costs for the trio of rules needed to deduce P; the analysis for the rules about C and Q is similar.

Since T and U have the same number of answers, it does not matter which of them appears first in the rule for A, so let us assume that this rule is ordered as written. The expected number of true instances of A is $32 \times 32/8^2 = 16$, and that of P is $16 \times 16/8 = 32$. The cost of deducing A forwards is 32 clauses of the form $U(X,Y)\ A(X,Y)$ and 16 of the form $A(X,Y)$ for a total of 48. The same applies for B and C.

The cost of solving A backwards with one variable bound, as would be done if the P rule were ordered as written above and used backwards, is (for each goal) one subgoal of the form $T(K,y)\ U(K,y)\ B(y,H)$, $32/8 = 4$ subgoals of the form $U(<K,Y)\ B(Y,H)$, and for each of these an expected $32/8^2 = 0.5$ subgoals of the form $B(Y,H)$, giving $4 \times 0.5 = 2$ such clauses expected, making altogether $1 + 4 + 2 = 7$ per goal. Over 7 goals, this gives an expected cost of 49, so that A should be computed forwards.

The costs of deducing P forwards and backwards can similarly be calculated as $16 + 32 = 48$ and $7 \times (16/8 + 32/8^2) = 17.5$ units, so the rule for P will be best used backwards.

Now that we know this, we can be sure that the clauses of the form $B(Y,H)$, as described above, will indeed be generated, and there will be 14 of them. Solving each of these will involve generating one subgoal of the form $V(Y,H)\ W(Y,H)$ and an expected $32/8^2 = 0.5$ subgoals of the form $W(Y,H)$, for an expected total cost of $14 \times 1.5 = 21$. This is less than 48, so that B should be deduced backwards if it appears in second place in the rule for P.

So the optimal strategy for these three rules will compute A forwards and B and P backwards, at a total cost of $48 + 21 + 17.5 = 86.5$ units. Exchanging the conjuncts in the P rule will lead to a strategy having exactly the same cost, since the problem is symmetric with respect to exchanging the letters A and B.

But now if we re-introduce the other predicates and rules, the optimiser may decide to use them the same way, with C forwards and Q backwards. The total cost of backward inference to deduce B will now be 42 units instead of 21, but this is still less than 48, so there appears to be no reason to compute B forwards. And there is no incentive to compute A or C backwards (each of these would incur extra cost of 1 unit), nor to change the orderings of the premises of the rules for P or Q (either of these moves would raise the cost of inferring B backwards by $49 - 21 = 28$ units and leave other costs unchanged). But if all of these changes are made at once, we get the optimal strategy described above. The total cost becomes 48 units at B, 21 at each of A and C, and 17.5 at each of P and Q for a total of $48 + 2 \times (21 + 17.5) = 48 + 42 + 35 = 125$ which is much less than $2 \times 86.5 = 173$.

4.3 Complexity of choosing rule directions

In this example, the optimal strategy was easy to see. However, it can be proved that in general the problem of finding the optimal coherent strategy (including conjunct ordering) is NP-complete, even if estimating the costs for the rules is in P. The proof is by reduction from the Vertex Cover problem [GJ79, page 46], that of finding a smallest set of vertices of an undirected graph such that every edge of the graph is incident to a vertex in the set; this problem is known to be NP-complete. We show how to reduce any graph to a triple (F, R, G), such that the optimal coherent strategy for this problem corresponds to a minimal vertex cover, using predicates like A above to represent nodes of the graph, and P for its edges. This reduction tells us that an upper bound on the time needed to find a vertex cover is given by the time needed to find the optimal strategy (plus the costs of reducing the graph to the triple, and of transforming the strategy to the vertex cover; these costs can be shown to be slight). Details of the problem reduction are relegated to Appendix A.

To prove strategy optimisation as a whole NP-complete, we would also need to show that the cost of a strategy can be estimated in time polynomial in the size of the input data, namely F, R, and G. There appear to be rule sets for which it could take exponential time to estimate the costs of some strategies according to the methods described here, but slightly less accurate estimates could be obtained in polynomial time. This is discussed in more detail in [Tre86].

4.4 Independence of the two complexity results

An important feature of the NP-completeness proof in the Appendix is that it uses rules with just two premises, so that it does not depend on the NP-completeness of conjunct ordering; the cost of finding the optimal strategy is NP-complete with respect to the number of *rules* present, rather than to the number of *premises*. The proof would still work if conjunct ordering took constant time. It applies equally to the case where choice of rule directions can be done in polynomial time and to that where it is NP-complete.

More practically, even if a set of rules had some property that made it possible to do all the conjunct ordering in polynomial time (for example, if it satisfied the assumptions made in [IK84]), finding the optimal strategy would still be NP-complete.

5 Finding Directions

The NP-completeness result suggests strongly that some kind of exhaustive search will be needed to find the optimal strategy. The example of how a simple hill-climbing algorithm got "trapped" at a local optimum also suggests an architecture for the search. The trap is strongly reminiscent of the well-known Prisoner's Dilemma from game theory, in which the course of action that appears optimal to each prisoner alone is sub-optimal for the two of them taken together. It is easy to avoid this if the two prisoners can communicate (and can trust each other). So, it seems that we must modify the direction choice and conjunct ordering algorithms to communicate with each other. In this section we discuss how to arrange such communication, and describe a way of generating and evaluating *partial strategies* to guide the search.

A partial strategy is any set of directions and/or conjunct orderings for some subset of the rules R. A *completion* of a partial strategy is any full strategy which agrees with the partial strategy.

5.1 Which is to be master?

The communication between the algorithms for choosing directions and orderings needs to be organised so that one algorithm takes account of what the other one will do. For example, we might revise the direction-choosing algorithm to ignore any pre-existing conjunct orderings and instead evaluate the merit of a set of directions with respect to the new orderings that would be chosen given those directions. Or the ordering algorithm could be revised to search for an ordering that was cheap given the changes in directions that would result if it were adopted, rather than assuming a prior set of directions. Either way, one algorithm will be "in control", using the other one as a subroutine. Since the ordering algorithms we have described are all fundamentally local in nature, dealing with one rule at a time, whereas the algorithm for choosing directions is global, considering the entire set of rules, it seems more appropriate to let the choice of directions be the "outer loop", invoking the ordering algorithm as necessary.

The decomposition of a strategy's cost into a sum of individual rule costs, as described in Section 3, is now impossible, because the cost of using a rule backwards will not be fixed but will depend strongly on the ordering of the premises of its successors. We will therefore use a best-first or A^* search [Nil80] of the space of sets of directions for the rules, growing our partial strategies by choosing a new direction for one rule at a time. An ordering algorithm will be called on each new partial strategy to allow re-estimation of the cost, but the orderings it returns will not form part of the partial strategy being grown. Only when directions for all rules have been chosen will it be possible to determine the correct conjunct orderings.

An A^* search requires an estimator function from partial strategies to costs, which must give a lower bound on the cost of any completion of a partial strategy. We now describe this estimator.

5.2 Lower bounds for partial strategies

Given a coherent partial set of directions, we can obtain a lower bound on the cost of backwards deduction in any completion thereof with the help of the ordering procedure described above in Section 2.5. We apply this procedure only to the rules specified as backwards, assuming that all facts not deduced by these rules are available directly, and then estimate the cost of running them over the goals described by G under this assumption using equation 4. Should the assumption turn out false, then the expenses of the goals will increase, since the expenses of the premises of the rules will rise, so the procedure definitely yields a lower bound.

The cost of the rules that are used forwards in the partial strategy can be found exactly, without any assumptions about other rules, since their inputs will (by the coherence assumption) always be available directly. Rules whose direction is not specified by the partial strategy are ignored, since there is no sensible way to estimate their costs when we cannot be sure of their direction. The final lower bound is obtained by adding the bounds obtained for backwards and forwards inference.

Note that, when we extend a partial set of directions by choosing forwards inference for another rule, this has no effect on our estimate for the cost of backwards inference (the previous estimate of this had already assumed that the rule in question was used forwards). And under coherence, the estimated costs of the rules that were already used forwards cannot change, since their costs are only affected by their predecessors, which must have already been forwards. So in this case the new lower bound cost estimate is found by just adding the cost of the new forwards rule(s) to the earlier estimate.

When we instead choose backwards inference for the next rule, this will still have no effect on the estimated costs of the rules that were used forwards (and still are). It will affect the estimates for the rules above the new backwards rule(s), since the expenses of their premises will increase. Thus some re-computation of old estimates will be needed. However, we do not by any means have to re-compute the cost of a new partial strategy from the ground up.

5.3 Searching for a coherent strategy

Given this lower bound on costs of completions of a partial strategy, we can embark on an A^* search of the space of such strategies. The lower bounds obtained here will include zero costs for rules that are not mentioned in the partial strategy, so that at first they will not be very realistic, but this will improve steadily as more rules are included. The conjunct orderings obtained during the early stages of the search will be provisional for many backwards rules, because we will only have loose lower bounds for the expenses of their premises. The order in which new rules are added turns out to be important.

The efficiency of the search will depend strongly on how realistic the lower bound estimate is, especially in the later stages. Pearl [Pea83] has shown that the expected time complexity of an A^* search increases with the exponential of the proportional error in the heuristic estimate used. Inaccuracy can, however, be tolerable in the early stages of the search, when the purpose of the lower bound estimate is mainly to prevent (or defer) consideration of partial strategies that can be seen to involve very high costs. Later on, the emphasis shifts: a number of fairly good partial strategies have been developed, and the cost estimates are used to distinguish the best from the good, so accuracy matters more.

More formally, the nature of the A^* search guarantees that the only partial strategies that are expanded, i.e. have another direction added to them, are those whose lower bound estimate is less than the eventual cost estimate for the optimal strategy. To see this, notice that the list of open partial strategies will at all times contain some partial strategy that is a subset of the optimal strategy, and therefore has a cost estimate that is less than the optimal cost. Any partial strategy whose cost estimate is higher than the optimal cost will therefore never be the cheapest on the list, and will never be selected for expansion.

So, if by using a more accurate lower bound estimate we can push the estimate for some partial strategy above the optimal cost, then we have saved some work; if we increase the estimate but it stays below the optimal cost, we have merely postponed an expansion step. A partial strategy with few rules in it will usually have a small estimated cost using any method, so the chance that

we can show it to be non-optimal is small; this chance increases as the partial strategies include more rules and the lower bound estimates come closer to the truth.

This means that in the early stages of the search, we may be content with a quick and inaccurate lower bound estimate, since we are mostly interested in detecting the worst partial strategies. But later we shall want more accuracy, since it can save us the effort of expanding more strategies and running the cost estimator over them. In particular, it will be dangerous if, at this stage, rules that have high costs are still left out of a partial strategy; such omissions would make accuracy impossible. We should therefore expand our early partial strategies by adding rules which are capable of deducing large numbers of facts, or which seem likely to be invoked often. This does not guarantee that the search will converge rapidly, but can be expected to avoid the grossest inefficiencies.

5.4 Pruning the search

It is also possible to rule out some partial strategies immediately. For this we will use a "strawman" strategy, one that can be found quickly and may not be very good but is believed to be better than a randomly chosen one. Such a strawman can be obtained by doing a few iterations of local improvement as suggested in Section 4, or perhaps in other ways. We might even pay a compliment to the author of a set of rules by using the conjunct orderings they wrote and simply running an algorithm drawn from [TG87] to choose rule directions based on these orderings.

Certainly any partial strategy whose lower bound estimate is above the estimated cost of the strawman should be discarded. This can be done even before the estimation is complete, if a running total of costs is kept by the estimator and compared with the strawman's cost. Better yet, any rule whose forwards cost (plus, in a coherent strategy, the forwards costs of its predecessors and so on) exceeds the the strawman's cost can immediately be fixed as a backwards rule, before even generating any partial strategies. Logic programmers often write rules that would, if used forwards, have very large or even infinite numbers of answers, so this is likely to be useful in practice.

6 Simpler Algorithms

The procedure we have given for finding the optimal strategy is complex in both concept and implementation, and even with the improvements suggested, may turn out to have unacceptable performance. We therefore consider in this section whether simpler algorithms that do not aim at the precisely optimal strategy may be preferable.

6.1 Hill climbing

We saw in Section 4 that a simple alternation between improving the choice of directions for rules and changing the order of their premises can, in a small example, converge on a strategy whose cost exceeds the optimal cost by over 40%. We conjecture that this difference could be made much larger by complicating the example further. So, if we wish to optimise by making incremental

changes to a complete strategy, rather than by growing a partial one, the changes must be made with more foresight, which is to say that larger increments of change must be contemplated. The disadvantages of hill-climbing algorithms that can only "see" one step ahead are familiar to any human who has climbed a hill in thick fog.

However, it does little good to thin the fog out a little by devising an algorithm that looks, say, three steps ahead, if the hill is a mile wide; and looking even three steps ahead is very much more expensive than one, if we consider every possible sequence of three steps (even after eliminating those that retrace themselves). It therefore seems imperative to concentrate on choosing our steps carefully.

6.2 Hotspots

When competent programmers are improving the efficiency of a logic program (or any other kind), they do not consider all changes of a given size. They are inclined to concentrate on the parts of the program where most of the work is done, and look for ways to improve these, possibly at the cost of some extra work elsewhere. It should be possible, by means of the cost estimation tools that we have already described, to find the "hotspots" in a logic program in much the same way as conventional optimisers find inner loops in a FORTRAN one, and work on them according to their importance. Thus the optimiser can consider a few major changes in places where they can do the most good.

This approach has difficulties of its own, though, because the large cost attributed to a rule may be attributable either to a poor choice of direction (or ordering) for that rule itself, or to unwise use of some other rule related to it. A hotspot eliminator would therefore need to identify a few places where the fire might be coming from, and consider how (or whether) each of them could be extinguished without causing a flare-up nearby. Such flare-ups are quite probable, because a change to the order in which a set of conjuncts are processed will usually result in one of them being solved more often than before. It is within the experience of human programmers that a chain of several changes to a program, each of them triggered by the previous one, may be needed to effect an improvement in its performance; this implies, once again, that a search through an irregular search space must be conducted.

The guidance for such a search would probably be heuristic. It would have to dismiss some possibilities without thorough estimation of their results. While an "algorithmic" search can also do this, it would do so by use of a threshold whose placement would be arbitrary and would partake of the quality of a heuristic. This heuristic quality is in fact not likely to be entirely absent even from an implementation of the algorithm described in Section 5, which was seen to depend for its efficiency on the order in which rules are added to a partial strategy; we have not proposed, and do not know, any criterion guaranteed to select the most important rule for this purpose.

7 Discussion

7.1 Incoherent strategies

The negative results of earlier sections still apply to the incoherent case. Fortunately, so does the positive result of Section 2.5, that the correct orderings for premises can be found once the rule directions are all known. So we can use the same search technique, but it will in general take longer, since coherence can no longer be used to restrict the set of partial strategies searched.

More importantly, without the coherence assumption we cannot say anything about how often a backwards rule will be invoked until the directions of at least some of its successors are known. This has the effect of weakening the lower bound estimates, on which the efficiency of the search depends so strongly. We can counteract this weakening to some extent by a policy of always adding to the current partial strategy a rule with at least one successor whose direction already is known (initially only goals have known directions). This does conflict with the criterion given for selection of the next rule to add to a partial strategy, and so will have an effect on the efficiency of the search.

7.2 Duplicate answers

Though we have alluded several times to the extra difficulty caused by the possibility of a fact being deduced twice from different inputs, or by different rules from the same inputs, the linear programming algorithm from [TG87] is the only one seriously affected by duplicates. All the other techniques described here, in particular those of Sections 2.5 and 5.3, are insensitive to the existence of duplicates.

7.3 Recursion

In general, accurate estimation of the cost of using a set of rules which contain recursive definitions is equivalent to solving the Halting Problem. We make no claim that the methods described here are applicable to rulesets that include recursion. However, there may exist classes of problems for which the depth of a recursion can be estimated, and thence its cost.

7.4 Utility of doing optimisation

The issue of how much optimisation to do before running the optimised program is endemic to all forms of optimisation: there is almost always a way to make any optimiser "smarter" and even more expensive, and there is seldom a way to predict accurately the reduction in running time to be gained from an extra expenditure of time on optimisation. Precautions against doing too much or too little optimisation are discussed in both [Smi85] and [Tre86]. The latter points out that, in cases like this where the optimiser requires estimates of the cost of running the program, it may be doubtful whether optimisation confers any benefit at all on the user, since the estimates may be in error. One way to improve the estimates is to observe the unoptimised program running on a

"live" problem, but this approach, taken to its logical extreme, suffices to guarantee that the cost of optimisation will exceed the cost of running. However, if many similar runs of the program are planned, it may be well worth while to refine a strategy by observing and improving it over several runs.

7.5 Violation of assumptions

The assumptions made in order to allow estimation of the costs of running rules are open to criticism as being unrealistic compared to real-world logic programming, where the data are often neither uniform nor independent. However, the main use of these two assumptions was in the derivation of the cost expressions used in the complexity proofs, where it was shown that, even under these strong simplifying assumptions, the optimisation in question was NP-complete; without them, it is not likely to become easier. The other use of the Argument Independence Assumption was in Section 2.4.2, where it was used (along with the Equal Opportunity Assumption) to prove two lemmata that were useful in enabling a modification to the conjunct ordering algorithm that offered to make it run faster. Since these lemmata affected only the speed and not the validity of the algorithm, we would simply have to fall back on the slower version if we expected that the assumptions would not be true.

8 Conclusions

We have described several algorithms for ordering the conjuncts of a conjunctive goal, including a highly general one, and have suggested ways to endow it with acceptable performance. We have sketched an algorithm for choosing the directions in which rules should be used. We have shown that these two optimisations are interdependent, and that the problem of optimising both directions and orderings when neither is given beforehand has a significantly higher computational complexity than either problem on its own. Finally, we have indicated how the ordering algorithm can be incorporated in an algorithm that solves the combined problem, and we have pointed out that the order in which rules are considered by this algorithm can be expected to have a major effect on its performance.

A NP-completeness of choosing rule directions

Here we give the details of the problem reduction for the NP-completeness proof mentioned in Section 4. We add a few remarks about the scope of applicability of the proof.

A.1 The problem reduction

For each vertex v of the graph, create a predicate $P_v(x)$ of one argument, and for each edge from v to another vertex w, create a predicate $P_{vw}(x)$, but only one such predicate for each edge, i.e. either P_{vw} or P_{wv} but not both. The idea is to arrange things so that to deduce P_{vw} cheaply we

need to deduce at least one of P_v and P_w forwards, and if either of P_v or P_w has been deduced forwards, P_{vw} can be deduced backwards at low additional cost. The costs will be such that the optimal strategy must, for every P_{vw}, cause at least one of P_v and P_w to be deduced forwards, but does this for a minimal number of such P_v. We shall show how to construct R, F, and G so that this holds. Let the graph have V vertices and E edges; we shall show that there is some value for E such that the reduction works for all graphs with at least that many edges. We exclude graphs that have a node of degree E; the optimal vertex cover is obvious for such graphs anyway.

We need to show that the numbers of bits needed to encode the graph itself and the (F, R, G) triple defining the optimisation problem are related by some polynomial. The number of distinct predicates is $3V + E$; the number of rules in R is just equal to the sum of the numbers V and E of vertices and edges in the graph, and the number of bits required to represent each rule is bounded above by some expression linear in $\log V + \log E$ (since each literal has to have enough bits to identify its predicate symbol); each predicate appears at most once in F and in G, and the number of bits of information attached to the predicate, describing the selectivity factors of its variables, the number of answers to it that are in F, etc., will turn out to be a linear function of $\log E$ (these numbers will be powers of E), so that the total number of bits used in this way is a linear function of $E \log E$.

The transformation of an optimal strategy, once found, into a vertex cover is simplicity itself: for each vertex v of the original graph, check whether the corresponding P_v is deduced forwards, and include v in the cover iff this is so. This clearly does not take exponential time.

The rules are

$$\begin{aligned} P_v(x, y) \ \& \ P_w(y, z) &\Rightarrow P_{vw}(x, y, z) \\ P_{v1}(x, y) \ \& \ P_{v2}(x, y) &\Rightarrow P_v(x, y) \end{aligned}$$

All variables have selectivity factors of $1/d$. There are n_1 facts for each predicate P_{v1} and n_2 for each P_{v2}, and n_3 goals of the form $P_{vw}(K, y, H)$ for every P_{vw}, where K and H are bound variables. So the number of true instances of each P_v, written $N(P_v)$, is $n_1 n_2/d^2$, and $N(P_{vw})$ is $n_1^2 n_2^2/d^5$. Set

$$\begin{aligned} d &= E^3 \\ n_1 &= E^5 \\ n_2 &= E^5 \\ n_3 &= 2E^4/(2E - 1) \end{aligned}$$

Then

$$\begin{aligned} N(P_v) &= E^4 \\ N(P_{vw}) &= E^5 \end{aligned}$$

For brevity, we shall use P_v to stand for the predicate which appears first in a P_{vw} rule, even though we are considering strategies which permute the premises. Likewise P_w will stand for the predicate of the other premise. Since P_{v1} and P_{v2} have the same number of answers, it does not matter which of them appears first in the rule for P_v.

A.2 The costs of different permutations

In what follows, we shall write G_i for the cost of generating a single resolvent with i literals in it, S_i for the cost of storing it, and I_j and L_j for the costs of indexing on, and returning one match to, a premise with j arguments. We shall make the plausible assumptions that $G_i \leq G_{i+1}$, $S_i \leq S_{i+1}$, $I_i \leq I_{i+1}$, and $L_i \leq L_{i+1}$ for every sensible value of i.

The cost of deducing P_{vw} forwards will always be at least $N(P_{vw})S_1$, which is proportional to E^5. The cost of inferring P_{vw} backwards, ignoring the costs of P_v and P_w, will depend on whether the rule for P_{vw} has to look up its inputs directly or look up and invoke a rule to deduce them backwards. Looking up the inputs will be more expensive iff there are (on average) fewer than one of them to be found. The expected numbers of inputs are in fact $N(P_v)/d = E$ at the first premise and $N(P_{vw})/d^2 = 1/E$ at the second, so the worst-case cost for solving the n_3 goals for P_{vw} backwards is n_3 times the sum of

$I_3 + L_3$	to index and look up the rule for P_{vw}
G_2	to generate the pair of subgoals
$I_2 + \frac{N(P_v)}{d} L_2$	to index and look up the answers for P_v
$\frac{N(P_v)}{d}(G_1 + I_2 + L_2)$	to generate the next subgoal and look up the rule for it
$(N(P_v)N(P_w)/d^3)G_0$	to generate the final answers

for a total of

$$n_3 \left(I_3 + L_3 + G_2 + I_2 + \frac{N(P_v)}{d}(L_2 + G_1 + I_2 + L_2 + \frac{N(P_w)}{d^2} G_0) \right)$$

all of whose terms are dominated by E^5. So the rule for P_{vw} will be best used backwards, if E is large enough. Now that we are sure of the direction of these rules, we can (and must) associate the costs of looking up the answers to P_v and P_w, or of looking up the rules that compute them, with the decisions to use these rules forwards or backwards.

The cost of deducing P_v forwards is

$$c_1 = I_2 + n_1(L_2 + G_2 + I_2) + N(P_v)(L_2 + G_1 + S_1)$$

with some additional cost for looking up the stored answers when P_{vw} is subsequently deduced. Let $m_1(v)$ is the number of rules mentioning P_v in their first premise, and $m_2(v)$ likewise for the second premise, then this extra cost will be $n_3 m_1(v)(N(P_v)L_2/d) + n_3 m_2(v)(N(P_{vw})/d^3)$. Since $m_1(v) + m_2(v) = deg(v) < E$ (by assumption), c_1 contains only two terms that are not clearly dominated by others, namely $n_1(L_2 + G_2 + I_2) + n_3 m_1(v) N(P_v) L_2/d$.

The total cost of solving P_v backwards with one variable bound, as would be done if it were the first premise of a P_{vw} rule used backwards, i.e. if $m_1(v) \geq 1$, is

$$c_2 = n_3 m_1(v) \left(I_2 + L_2 + G_3 + I_2 + \frac{n_1}{d}(L_2 + G_2 + I_2) + \frac{N(P_v)}{d} L_2 \right)$$

Since $n_3 > d$, this is greater than the sum of the two dominant terms in c_1. This gives us the first half of what we want for the problem reduction: for each edge vw, there is a node v corresponding to a rule used forwards.

For inferring P_w backwards with both variables bound we get, for each P_{vw} rule invoking this P_w at its second premise, a cost of

$$c_3 = n_3 m_2(w) \left(\frac{N(P_v)}{d}(L_2 + G_2 + I_2)) + \frac{n_1 N(P_v)}{d^3}(L_2 + G_1 + I_1) + \frac{N(P_{vw})}{d^2} L_2 \right)$$

and the dominant term here is $n_3 m_2(w) N(P_v)(L_2+G_2+I_2)/d = 2m_2(w)E^5(L_2+G_2+I_2)/(2E-1)$ which, since $m_2(w) < E$, is still less than the dominant term of c_1, namely $n_1(L_2+G_2+I_2)$, so that P_w will definitely be inferred backwards unless it appears in the first premise of some P_{vw} rule.

This makes it clear that the optimal strategy corresponds to a minimal vertex cover, with the rules that are used forwards corresponding to the nodes that are in the cover. Note that this proof does not need the coherence assumption, and so would also apply to the search for an optimal incoherent strategy with premise re-ordering. Admittedly the optimal strategy displayed is a coherent one, but this is just because the rules for P_{vw} are more expensive to use forwards, for a reason that is independent of the directions of their predecessors. There are also no duplicates involved.

References

[FST86] S. Finkelstein, M. Schkolnick, and P. Tiberio. *Physical Database Design for Relational Databases*. Technical Report RJ5034, IBM Almaden Research Center, February 1986.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman, 1979.

[HN84] L.J. Henschen and S.A. Naqvi. Compiling queries in recursive first order databases. *Journal of the ACM*, 31(1):47–85, January 1984.

[IK84] T. Ibaraki and T. Kameda. On the optimal nesting order for computing N-relational joins. *ACM Transactions on Database Systems*, 9(3):482–502, 1984.

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. *Efficient Optimization of Non-recursive Queries*. Technical Report, Microelectronics and Computer Technology Corporation, February 1986.

[MS81] D. P. McKay and S. Shapiro. Using active connection graphs for reasoning with recursive rules. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 368–374, Vancouver, August 1981.

[Nat87] K.S. Natarajan. Optimizing backtrack search for all solutions to conjunctive problems. In *Tenth International Joint Conference on Artifical Intelligence*, pages 955–958, August 1987.

[NH80] S. A. Naqvi and L. J. Henschen. Performing inferences over recursive data bases. In *Proceedings of the First National Conference on Artificial Intelligence*, pages 263–265, American Association for Artificial Intelligence, August 1980.

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, 1980.

[Pea83] Judea Pearl. Knowledge versus search: a quantitative analysis using A*. *Artificial Intelligence*, 20(1):1–14, January 1983.

[Rou82] Nicholas Roussopoulos. Indexing views in a relational database. *ACM Transactions on Database Systems*, 7(2):258–290, June 1982.

[SG85] David E. Smith and Michael R. Genesereth. Ordering conjunctive queries. *Artificial Intelligence*, 26(2):171–215, 1985.

[Smi85] David E. Smith. *Controlling Inference*. PhD thesis, Stanford University, August 1985.

[TG87] R. J. Treitel and Michael R. Genesereth. Choosing directions for rules. *Journal of Automated Reasoning*, 3(4):395–431, December 1987.

[Tre86] R.J. Treitel. *Sequentialization of Logic Programs*. PhD thesis, Stanford University, 1986.

[Ull84] Jeffrey D. Ullman. *Implementation of Logical Query Languages for Databases*. Technical Report STAN-CS-84-1000, Stanford University, May 1984.

[War81] D.H.D. Warren. Efficient processing of interactive relational database queries expressed in logic. In *Proceedings of the Seventh VLDB conference*, pages 272–281, IEEE, 1981.

[WOLB84] Larry Wos, Ross Overbeek, Ewing Lusk, and Jim Boyle. *Automated Reasoning: Introduction and Applications*. Prentice-Hall, 1984.