Controlling Backward Inference

David E. Smith
Science Center, Rockwell International Corporation, Palo Alto Laboratory, 444 High Street, Palo Alto, CA 94301, U.S.A.

ABSTRACT

Effective control of inference is a critical problem in artificial intelligence. Expert systems make use of powerful domain-dependent control information to beat the combinatorics of inference. However, it is not always feasible or convenient to provide all of the domain-dependent control that may be needed, especially for systems that must handle a wide variety of inference problems, or must function in a changing environment. In this paper, a domain-independent means of controlling inference is developed. The basic approach is to compute expected cost and probability of success for different backward inference strategies. This information is used to select between inference steps, and to compute the best order for processing conjunctions. The necessary expected cost and probability calculations rely on simple information about the contents of the problem solver's database, such as the number of facts of a given form, and the domain sizes for the predicates and relations involved.

1. Introduction

Control of inference has always been a critical problem in AI. Even for relatively simple problems, unguided inference leads to a combinatorial explosion in the number of different inference steps possible. To overcome this problem an inference procedure must be capable of selecting among the myriad of different inferences steps possible at each point in the problem solving process, so that only the best alternatives are attempted. In this paper we will be concerned with the problem of control for the case of backward or goal-directed inference. As an example, consider a simple set of rules about kinship.[1]

$$\mathsf{Brother}(c, d) \rightarrow \mathsf{Sibling}(c, d)\,,$$

$$\mathsf{Parent}(c, p) \wedge \mathsf{Parent}(d, p) \wedge c \neq d \rightarrow \mathsf{Sibling}(c, d)\,,$$

[1] We will use standard first-order predicate calculus for expressing facts about the world. Capitalized words will be used for constants and relations, and lower-case letters for variables.

Artificial Intelligence **39** (1989) 145–208
0004-3702/89/$3.50

$$\text{Mother}(c, p) \rightarrow \text{Parent}(c, p),$$

$$\text{Father}(c, p) \rightarrow \text{Parent}(c, p).$$

The first states that brothers are siblings, the second that siblings share a common parent, the third that mothers are parents, and the fourth that fathers are parents. Suppose that we have the problem of finding a single solution to the goal Sibling(Rob, s). The AND/OR tree for this goal is shown in Fig. 1. Initially, there are two different possible inference steps: the step generating the brother subgoal, and the step generating the conjunctive subgoal. If the conjunctive subgoal is chosen there are six different possible orderings for working on the conjuncts. For each of the two parent conjuncts there are two different inference steps possible: a step generating a father subgoal, and a step generating a mother subgoal. In this paper we will be concerned with the problem of how to automatically decide among all of these different alternatives.

1.1. Domain-dependent control

Most of the practical systems that have been built in AI overcome the control problem by using domain-dependent control information: information about the order in which to use particular rules, information about the order in which to process conjuncts in conjunctions, and information about the order in which to ask questions. Often this information is embedded in the system in subtle ways. For example, many systems rely on the order in which rules appear in the database and the order in which conjuncts appear in the premises of rules.

History has taught us that this kind of domain-dependent control information (whether implicit or explicit) constitutes a significant portion of the

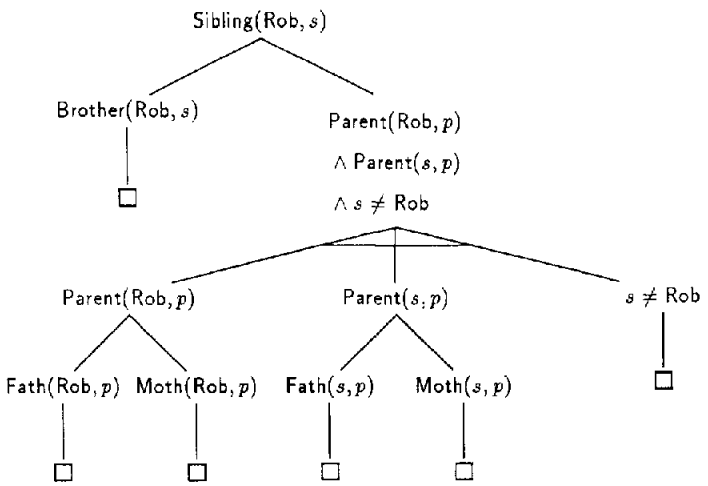

Fig. 1. Backward AND/OR graph for the kinship problem.

expertise in many domains, and is therefore indispensable for these domains. The trouble with domain-dependent control is that it is often inconvenient or impossible to provide all of the control information necessary for a domain. Consider our simple kinship example for a moment, and suppose that we have a database that contains mostly facts about fathers and brothers. We need to provide information that the brother rule should be applied before the parent rule when answering sibling queries, and that the father rule should be applied before the mother rule when answering parent queries. Unfortunately, the best order for solving the conjunction depends on the type of sibling query being solved. For the queries Sibling(Rob, s) and Sibling(s, t) the order given is best, but for the query Sibling(s, Rob) the order of the first two conjuncts should be reversed. For queries of the form Sibling(Rob, Joe) the third conjunct should be done first. Although it would be cumbersome, we could still provide all of this control information to a system. But now consider all of the different conjunctive questions that might be asked using only this trivial vocabulary; the user might want to know about grandparents or grandchildren (Parent(x, y) $\wedge$ Parent(y, z)), or aunts and uncles, nieces and nephews (Parent(x, y) $\wedge$ Sibling(y, z)), great uncles and aunts, cousins, maternal cousins, and so on. Although we might be able to supply ordering information for the most common possibilities it would certainly be a nuisance to have to provide the ordering information for all of these different possible conjunctive queries. Furthermore, the user could always cook up a conjunctive query that we had not anticipated. Thus, for systems that are expected to solve a wide range of problems, it is often inconvenient or impossible to provide all of the necessary control information.

This problem is further exacerbated if the system's database is changing over time. In this case, the best problem solving strategy may be changing. In the kinship example, if the database initially contains only mother and father facts the best strategy would be to go right to work on the conjunction and not bother with the brother rule. However, if lots of brother facts are suddenly added to the database, the best strategy changes from working on the conjunction first, to looking for brother facts first. Likewise, if the database contained lots of sister facts and the rule Sister(c, d) $\rightarrow$ Sibling(c, d) is added to the database, the best strategy changes to that of looking for sister facts first.

In sum, although domain-dependent control information is important for many applications, it is often inconvenient or impossible to provide all the necessary control information on a case by case basis. Instead we need more general principles.

1.2. Domain-independent control

Many different domain-independent control strategies have been developed and used in AI. Classic examples include depth-first and breadth-first backward chaining, the linear-input strategy for resolution, and the unit-preference

strategy for resolution. The trouble with these simple domain-independent strategies is that they are too weak to stop the combinatorial explosion that occurs in most inference problems.

Consider the unit-preference strategy, which says that we should always prefer to work on goals that contain only a single clause. One problem with this strategy is that the syntactic simplicity or complexity of a goal says nothing about whether or not it is likely to be true. In our kinship example, the unit-preference strategy says that we should always work on the brother clause before working on the conjunction. On the face of it, this control strategy looks good. However, it might be the case that the database contains very few brother facts but lots of mother and father facts. In this case it would make more sense to work on the conjunction first.

A second problem with unit-preference is that it does not take into consideration the potential cost of finding solutions to the subgoals under consideration. In the kinship example, we might have additional rules that would allow us to conclude facts about brotherhood by analyzing genetic characteristics of the individuals. Finding an answer to the brother clause could therefore provoke a very long and complex inference chain. Thus, the syntactic complexity of a clause only provides a measure of the cost of the next immediate inference step, which is only a crude lower bound on the cost of finding a solution by that path. A syntactically simple goal can generate complex subgoals, while a syntactically complex goal can have an immediate solution.

The same sorts of criticisms apply to other domain-independent control strategies. Generally, they are based on very simple local evaluations of the steps under consideration, and are therefore insensitive to the shape or structure of the search space and to the information present in the system's database. What is needed, is a domain-independent control strategy that is sensitive to *global* information about the shape of the search space for each problem, and to the content of the system's database.

1.3. The approach

The basic idea behind our approach to controlling inference is best explained by example. For the kinship problem, suppose that there are a number of brother facts in the system's database. We would generally want to try to generate the brother subgoal before trying the conjunction. The rationale for this decision is something like:

> If there is at least a decent chance of finding an answer to the problem by trying a path p, and the alternative paths are much more expensive, try p first.

Now suppose that we generate the brother subgoal, but fail to find a matching brother fact in the database. We then go on to generate the

conjunctive subgoal and try to find solutions to the parent conjuncts. For solving the parent conjuncts, if there are more father facts than mother facts in the database we would prefer to generate the father subgoal before the mother subgoal. The rationale for this is something like:

> If two paths are of roughly equal cost, pick the one that has greater chance of success.

Both of the above arguments are very simple examples of decision analysis or utility arguments; they weigh the likelihood that a given inference step will lead to a solution against the expected cost of finding a solution to the problem using that step.

It is important to realize that these arguments do not just apply to one specific problem, but rather to many similar problems. For example, the first argument above would apply to any sibling query independent of what the particular argument bindings happen to be. Likewise, the second argument would apply to any query of the parent relation. In general, these arguments are stating that one step will be *better* than another *on average* for an entire set of queries involving the same relation. What makes this kind of argument possible is that the search space is nearly identical for all problems involving the same relation. For example, we could apply the brother rule to any sibling query. Likewise, we could apply the father and mother rules to any query of the parent relation.

Our basic approach to control involves formalizing the kind of decision theoretic argument given above.

We will say that two goal expressions have the same form if they have the same relation, and the same set of arguments are bound to constants. For example, the goals Sibling(Rob, x) and Sibling(Joe, x) are of the same form, while the goals Sibling(x, Fred) and Sibling(Rob, Fred) are different forms. In general for a relation of arity n, there are 2^n different forms.

We then make use of simple probability theory to compute, for each possible goal form, the probability that each inference step will ultimately lead to a solution, and the expected cost of finding a solution in that portion of the search space lying below the inference step. This probability and expected cost information for each inference step will be combined into a single number that we will refer to as the *worth* of the inference step. We will define all of these quantities precisely in Section 2.1, but for now it is sufficient to think of the worth of an inference step as a statistical assessment of the best that can possibly be done by exploring the search space rooted at that step. As we will show in Section 2.5, if an inference procedure does a best-first search of the inference space, based on the worth assigned to each step, *the resulting inference strategy will have the lowest expected cost* for solving the problem. In other words, the resulting inference strategy will be the best single strategy, on average, for all problems of that particular form.

In general, computing worth information for an inference step is an expensive process since it requires a search of the space lying below that inference step. However, each worth calculation applies to an entire class of problems (all problem of a particular form). Furthermore, the worth information for a given form can be used in computing the worth of inference steps that generate subgoals of that form. As a result (and as we will show in Section 2.4.3), for a given set of implications we can compute the worth of every inference step allowed by those implications in time that is linear in the number of implications. We can therefore precompute the worth for each potential inference step allowed by a database and store that information with the corresponding implication used in the inference step.

1.4. Assumptions

There is an important assumption inherent in the approach outlined above. By treating all queries of the same form alike, we are assuming that the queries expected and the facts in the database are uniformly distributed over the domains of the relations involved.[2] Thus, if there are five father facts in the database, and there are 1000 people in our universe, we would predict that the chance of success for a given father query would be quite low. This assumption may not always be correct. The queries expected may be predominantly about the individuals mentioned in the database, in which case the actual chances of success with a given query are much higher than we would predict. We do not currently take information about the likely distribution of queries into account.

In addition to the uniformity assumption, we also make several other assumptions. First, we will only consider backward inference over a database of implications and ground atomic facts. Further, we assume that none of the implications is recursive, and that none of them contain embedded functional expressions. None of these restrictions appears to be fundamental, but many of the details remain to be worked out. These restrictions are discussed further in Section 5.

We also will not consider the effects of either caching or redundancy in the search space on our selection of a strategy. (By redundancy we mean that the same answer can be found using more than one inference path in the search space.) Both of these restrictions are the manifestation of a broader assumption we refer to as the *independence assumption*. Two inference steps are *independent* if neither requires that the other be performed first. In our kinship example, the step of generating the brother subgoal, and the step of generating the conjunctive subgoal are independent, but the step of generating the conjunctive subgoal and the step of generating the father subgoal are not, since the latter requires that the former be performed first. The assumption is that

[2] Actually, the assumption is not quite this strong. We assume that the proportion of queries of a given form that can be answered by looking in the database is the same as the ratio of such facts in the database to the total domain size of the relation.

the cost of an inference step, and the likelihood that it will contribute an answer to the problem, are not affected by the performance of other independent inference steps. For example, in the kinship problem we assume that performing the inference step of generating the brother subgoal does not affect the cost or likelihood of any of the inference steps associated with the conjunctive subgoal.

Usually the independence assumption is correct for inference problems, but it is violated by redundancy in the search space or by the use of caching. In the case of redundancy, the likelihood that a redundant portion of the search space will produce answers to the problem drops to zero once the steps it is redundant with are performed. In the case of caching, once a useful solution is cached, the expected cost drops for those portions of the space that benefit from the cached result. The approach used in this paper still works when the independence assumption is violated, but does not guarantee an optimal strategy under these circumstances.

1.5. Organization

In Section 2, we will show how to calculate the worth of an inference step for the case of disjunctive inference (i.e. no conjunctions), where only a single answer is needed. This involves first developing formulas for the probability of success of database lookup steps, and for the probability of success and expected cost of arbitrary strategies. After this analysis, some reduction theorems are presented that allow us to find the best strategy and calculate worth in a computationally tractable way. Much of this analysis is similar to the analysis done by Simon and Kadane in [11]. Throughout this section, the reader should keep in mind that the objective is calculating worth information and that best-first search using worth information will lead to the strategy having the lowest expected cost.

In Section 3, we expand the treatment to the case where more than one answer is sought. In Section 4, we make use of the analysis of the first two sections to treat search spaces involving conjunctions. This involves first showing how to compare conjunctive strategies, and then showing how to find a good ordering without examining all of the possibilities. In Section 5, we discuss the relationship between this work and related work, and consider possible extensions of this analysis to the treatment of recursive inference and mixed forward and backward inference.

2. Disjunctive Search Spaces

2.1. Definitions

In order to rigorously define the worth of steps in a search space we must first introduce some notation for inference steps, database lookup steps, and *strategies*.

We will use the symbol D_g to refer to the step of performing a database lookup on the goal g. Similarly, we will use I_r^g to refer to the inference step of using the implication r to generate a subgoal from the goal g. Rather than assigning names to each implication, we will often refer to them by using a relation in the premise of the implication. Thus, in our kinship example, the step I_B^g will be used to refer to the step of using the brother/sibling rule to generate a subgoal from the goal g.

In general, an inference step will only be successful if the conclusion of the implication unifies with the goal. In order for this unification to have any chance of success, the goal and the conclusion of the implication must have the same relation. We will refer to a step having this property as a *legal* inference step.

Definition 2.1. A *strategy* for a query is a sequence of legal inference steps and database lookup steps such that the goal for each step is either the initial goal, or some subgoal that would result from the successful completion of a previous inference step in the strategy.

As an example, for a query of the parent relation, the inference step I_F^g followed by the database lookup step $D_{g'}$ where g' is the subgoal generated by the first step, would be a legal strategy. Likewise, the sequence $I_F^g I_M^g D_{g'} D_{g''}$, where g'' is the subgoal generated by I_M^g, would also be a legal strategy. However, the sequences $I_M^g D_{g'}$, $D_{g'} I_M^g$, and $D_{g'} I_F^g I_M^g$ would not be legal strategies, because the goal g' will not be generated prior to the step $D_{g'}$.

When specifying a strategy it is usually possible to abbreviate the above notation considerably. For inference steps it is often clear from the implication being used which goal or subgoal the inference step refers to. Likewise, for database lookup steps the relation in the goal is often enough to indicate the goal for the step. We will therefore omit this information when there is no ambiguity. Thus, the strategy $I_F^g I_M^g D_{g'} D_{g''}$ will be abbreviated as $I_F I_M D_F D_M$.

Note that a single strategy is generally applicable to all problems having the same relation. Thus, the strategies given above could be applied to any query of the parent relation.

We will use $P(s_c)$ to refer to the *probability* that a strategy s will solve an arbitrary problem in the class c. In other words, if the strategy were applied to all problems in the class c, the fraction $P(s_c)$ of these applications would actually solve the problem. Similarly, we use $E(s_c)$ to refer to the *expected cost* of using a strategy s to solve an arbitrary problem in the class c. In other words, if the strategy were applied to all problems in c, the average amount of computation required would be $E(s_c)$. Usually the class c under consideration will be clear from context and will be omitted.

We define the *utility* of a strategy for solving a problem as the ratio of the probability of success to the expected cost for the strategy, $U(s) \stackrel{\text{def}}{=} P(s)/E(s)$.

Let $S(x)$ refer to the set of all legal strategies that begin with the step x and contain only descendants of the step x. We then define the *best strategy*, $B(x)$, of a step x as the strategy in $S(x)$ having the highest utility. We define the *worth* of an inference step, $W(x)$, as the utility of the best strategy for x. These definitions are summarized below:

Definition 2.2.

$$U(s) \stackrel{\text{def}}{=} P(s)/E(s)\,,$$

$$B(x) \stackrel{\text{def}}{=} s\colon \max_{s \in S(x)} U(s)\,,$$

$$W(x) \stackrel{\text{def}}{=} U(B(x)) = \max_{s \in S(x)} U(s)\,.$$

As we have defined it, the worth of an inference step is an evaluation of how good the inference step is, since it is an assessment of the best that can possibly be done by exploring that part of the search space rooted at the step.

2.2. Local probability

The first step in evaluating the worth of inference steps is to figure out (1) the probability that simple database lookup steps will succeed, and (2) the probability that individual inference steps will work for a given goal expression.

2.2.1. *Database lookup*

Consider a particular goal expression of the form $R(\boldsymbol{B})$, where $\boldsymbol{B}$ is some vector of constants and variables. To compute the probability that there are one or more facts in the database matching this goal expression we need to know three things:

$m \stackrel{\text{def}}{=}$ the number of atomic facts in the database having the relation R,

$g \stackrel{\text{def}}{=}$ the number of different possible bindings $\boldsymbol{b}$ that have constants in the same positions as $\boldsymbol{B}$,

$h \stackrel{\text{def}}{=}$ the average number of solutions (in theory) for goals of the form $R(\boldsymbol{b})$ over all possible bindings $\boldsymbol{b}$ that have constants in exactly the same positions as $\boldsymbol{B}$.

The first quantity can be found by simply counting the atomic facts that contain R in the database.

The second quantity, the number of different possible bindings of the same form as $\boldsymbol{B}$, is just the cardinality of the domains for all arguments of R bound

by $\boldsymbol{B}$. Let $X_{\boldsymbol{B}}$ be the cross product of the domains for the arguments of R bound by $\boldsymbol{B}$:

$$X_{\boldsymbol{B}} = \prod_{\{i:\ \mathsf{Constant}(B_i)\}} \mathsf{Domain}(R, i)\ ,$$

then

$$g = \|X_{\boldsymbol{B}}\| = \prod_{\{i:\ \mathsf{Constant}(B_i)\}} \|\mathsf{Domain}(R, i)\|$$

(here $\|s\|$ refers to the cardinality of the set s). To compute this, we need to know the sizes of the domains for the arguments of the relation R.

The third quantity is the average number of solutions for goals of the form $R(\boldsymbol{B})$ given a complete model of the world.

$$h = \operatorname*{Avg}_{\boldsymbol{b} \in X_{\boldsymbol{B}}} \|R(\boldsymbol{b})\|\ .$$

This quantity is always bounded by the cardinality of the cross product of the domains for the remaining unbound arguments of R:

$$h \leqslant \prod_{\{i:\ \mathsf{Variable}(B_i)\}} \|\mathsf{Domain}(R, i)\|\ .$$

As described in [13], h can also sometimes be estimated using readily available information about the domain. For example, we might have knowledge that a person has two parents, or that a particular relation is functional when certain of its arguments are bound. In Section 2.2.2, this kind of information is used to compute h for one of the subgoals in the kinship problem.

Theorem 2.3. *Let $D_{R(\boldsymbol{B})}$ refer to the step of finding a solution in the database for a query of the form $R(\boldsymbol{B})$. The probability that this step will succeed is*

$$P(D_{R(\boldsymbol{B})}) = 1 - \binom{(g-1)h}{m} \Big/ \binom{gh}{m}\ . \tag{1}$$

The proof of this result can be found in Appendix C. (We have assumed that the set of bindings $\boldsymbol{B}$ lie within the appropriate domains for the relation R, i.e. $B_i \in \mathsf{Domain}(R, i)$. This assumption is often true for subgoals generated internally. When this is not true, the total probability must be multiplied by the probability that variable bindings lie within the appropriate domains, as suggested in [13].)

There are several useful special cases of the above theorem.

Corollary 2.4. *If* $m = 1$ (*i.e. only one fact in the database has the desired relation*),

$$P(D_{R(B)}) = 1 - \binom{(g-1)h}{1} \Big/ \binom{gh}{1} = 1 - \frac{g-1}{g} = \frac{1}{g}.$$

Corollary 2.5. *If* $h = 1$ (*i.e. ground clauses or functional expressions*),

$$P(D_{R(B)}) = 1 - \binom{g-1}{m} \Big/ \binom{g}{m} = 1 - \frac{g-m}{g} = \frac{m}{g}. \tag{2}$$

This is as we would expect, since there are m answers spread out among g different possible queries of the same form.

Corollary 2.6. *If* $g = 1$ (*i.e. none of the variables in R are bound*),

$$P(D_{R(B)}) = \begin{cases} 0, & \text{if } m = 0, \\ 1, & \text{otherwise.} \end{cases}$$

2.2.2. *Example*

We can apply these equations to the kinship example shown in Fig. 1. Assume that the universe contains five individuals and that the database consists of the facts

Brother(Rob, Larry),

Father(Rob, Bill),

Mother(Rob, Terry),

Father(Pat, Bill),

Father(Larry, Bill).

We will use subscripts of f and b on a relation to refer to a goal expression in which the corresponding argument positions are bound and free respectively. Thus R_{fb} refers to a goal of the form $R(x, B)$, i.e. the first argument is a free variable and the second is bound to an unknown constant. Thus, $D_{R_{\text{fb}}}$ refers to the step of looking up a goal of the form $R(x, B)$ in the database.

For the goal expression, Brother(Rob, s), we note that there is one brother fact in the database, so $m = 1$. The domain for the first argument of this relation is the set of persons, which for our example has been limited to five distinct individuals. Thus $g = 5$.

$$P(D_{B_{\text{bf}}}) = 1/g = 1/5 = 0.2.$$

By similar reasoning, the probability for the step $D_{M_{bf}}$ is also 0.2.

The more interesting calculations are those for goal expressions containing the father relation. Consider the step $D_{F_{bf}}$. There are three fatherhood facts in the database so $m = 3$. Since Father(x, p) is functional when x is bound we can use the restricted form for functional expressions, equation (2):

$$P(D_{F_{bf}}) = m/g = 3/5 = 0.6 .$$

For the step $D_{F_{fb}}$ the functional relationship does not hold. In this case we must use the full power of (1). Again, $m = 3$ and $g = 5$, but for this query $h = 3$, since there are an average of three children per father in our world:

$$P_0(D_{F_{fb}}) = \binom{(g-1)h}{m} \Big/ \binom{gh}{m} = \binom{4\times 3}{3} \Big/ \binom{5\times 3}{3} = 0.484 .$$

So,

$$P(D_{F_{fb}}) = 1 - 0.484 = 0.516 .$$

Finally, consider the clause $x \neq y$ where both x and y are bound. Since the universe contains only five individuals there are twenty of these facts implicitly available in the database ($m = 20$). The product of the domain sizes, g, is 25 so

$$P(D_{\neq bb}) = m/g = 20/25 = 0.8 .$$

For a large database this probability would approach one.[3]

The complete data for the kinship problem in summarized in Table 1.

2.2.3. *Inference steps*

Computing the probability that an inference step will succeed is much simpler than the computation for database lookup steps. Suppose we have an implication of the form $\phi \rightarrow R(\boldsymbol{\alpha})$ and we want to know the probability that a goal of the form $R(\boldsymbol{B})$ will match the consequent of this implication. If $\boldsymbol{\alpha}$ contains variables in every position where $\boldsymbol{B}$ contains a constant the probability will be 1:[4]

[3] In general, to get a more realistic estimate for a clause like this we would have to make use of information about those previous conjuncts that bind the variables involved. For this example, the variables will be restricted to the domain of people by previous conjuncts.

[4] Actually we are being a bit sloppy here. If $\boldsymbol{\alpha}$ contains the same variable more than once (e.g. $\boldsymbol{\alpha} = \langle x, x \rangle$) we could only treat one of the two argument positions as a variable. Treitel [16, 18] gives an equation for computing probability that covers this more general case.

Table 1
Probability of finding a solution for queries of the four database relations Brother, Mother, Father and $\neq$, for each of the four different possible ways of binding the variables

	ff	bf	fb	bb
Brother(x, y)	1	0.2	0.2	0.04
Mother(x, y)	1	0.2	0.2	0.04
Father(x, y)	1	0.6	0.52	0.12
$x \neq y$	1	1	1	0.8

$$[\forall i(\text{Constant}(B_i) \rightarrow \text{Variable}(\alpha_i))] \rightarrow P(s) = 1 ,$$

where s refers to the inference step of applying $\phi \rightarrow R(\boldsymbol{\alpha})$ to the goal $R(\boldsymbol{B})$.

Suppose that there are some positions where both $\boldsymbol{\alpha}$ and $\boldsymbol{B}$ contain constants. Since we do not know the identity of the constants in $\boldsymbol{B}$, we need to consider the probability that the constants are the same. This probability is the inverse of the product of all domain sizes involved:

$$P(s) = \prod_{\{i:\ \text{Constant}(B_i) \wedge \text{Constant}(\alpha_i)\}} \frac{1}{\|\text{Domain}(R, i)\|} .$$

For our kinship example, the implications contain only variables in their conclusion. As a result, the probabilities will always be one for this example.

In the sections that follow it will be important to distinguish between two different probabilities:

(1) *Local probability*: the probability that an individual step can be performed.

(2) *Global probability*: the probability that a step or strategy will actually solve the overall problem.

To distinguish these two, we will henceforth use $L(x)$ to refer to the local probability of an inference step. Note that for inference steps the global probability is always zero since the inference step alone can never solve the problem. For example, in our simple kinship problem, reducing the goal Parent(Rob, p) to Father(Rob, p) cannot, by itself, solve the problem.

2.3. Computing E and P for a strategy

In the previous section we showed how to compute the probability of success for database lookup steps and how to compute the local probability of success for inference steps. We now turn to the task of computing the probability of

success and expected cost for strategies, given the local probability and cost of each inference step, and given the probability and cost for each database lookup step. We do this by recursively breaking the strategy into two pieces, calculating the probability and expected cost for each of the two pieces and then combining the two pieces.

There are two relatively simple cases, the combination of totally *independent* strategies and the combination of totally *dependent* strategies. We say that two strategies are *totally independent* if *no* step in either strategy depends on the ability to perform *any* step in the other strategy. In contrast, a strategy is *totally dependent* on another strategy if *every* step in the first strategy depends upon the ability to perform *every* step in the second strategy.

First consider the totally independent case as illustrated in Fig. 2. We want to know the probability of success and expected cost for the strategy st where the substrategies s and t are totally independent. In this case, the strategy s will always be performed. The cost of this is $E(s)$. The strategy t will only be performed if s fails to solve the problem. The probability of this is $P_0(s) \stackrel{\text{def}}{=} 1 - P(s)$. In this case, the additional expected cost will be $E(t)$. Therefore,

$$\begin{aligned} E(st) &= E(s) + P_0(s)E(t)\,, \\ P(st) &= P(s) + P_0(s)P(t)\,. \end{aligned} \tag{3}$$

Next consider the totally dependent case, where every step in the substrategy t depends on every step in the substrategy s, i.e. t cannot be performed unless s is performed first. This situation is illustrated in Fig. 3. As before, the strategy s will always be attempted. The expected cost of this is just $E(s)$. The chance that s is successfully completed is $L(s) = \Pi_{x \in s} L(x)$ (by the definition of total dependence). In this case, t will be performed (since s contains only inference steps it can never produce an answer) and the expected cost of this is $E(t)$. Thus,

$$\begin{aligned} E(st) &= E(s) + L(s)E(t)\,, \\ P(st) &= L(s)P(t)\,. \end{aligned} \tag{4}$$

Although the expected cost and probability of success for many strategies can be built up using only these two basic cases, in general a more powerful result is required.

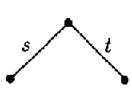

Fig. 2. Combining independent strategies.

Fig. 3. Combining dependent strategies.

Definition 2.7. We say that a strategy t is *commonly rooted* if every step in t depends upon exactly the same set of inference steps (excluding those in t itself).

Suppose that s and t are two strategies with t commonly rooted. To compute the probability and expected cost for st we need to separate out the parts of s that are independent of t from the parts on which t depends. Let s_t refer to the subsequence of steps in s on which t depends, and let $s_{\bar{t}}$ refer to the remaining subsequence of s containing only steps that are independent of t. (Since t is commonly rooted, these sets are the same for every member of t.) As with the simple cases the strategy s will be attempted in any case. The cost of this is $E(s)$. For t to be attempted there are two conditions that must be met. First, all of the inference steps in s on which t depends must have been successful. The probability of this is $L(s_t)$. Second, the strategy s must have failed to find an answer to the problem. Given the previous requirement, the probability of this is $P_0(s_{\bar{t}})$. Thus, the chance that t will be attempted is the product of these two terms. The expected cost and probability of success for st is therefore given by

$$\begin{aligned} E(st) &= E(s) + L(s_t)P_0(s_{\bar{t}})E(t)\,, \\ P(st) &= P(s) + L(s_t)P_0(s_{\bar{t}})P(t)\,. \end{aligned} \tag{5}$$

For convenience we define

$$P_0(s, t) = L(st)P_0(s_{\bar{t}})\,.$$

Intuitively, we can think of this as representing the probability that the strategy s fails in such a way that t is still possible.

2.3.1. *Example*

In Fig. 4, the inference space for the goal expression Parent(Rob, p) is shown. Using the above equations, together with the local probability data summarized in Table 1, E and P can be computed for the six different legal strategies for this goal. For purposes of the example assume that the cost of each inference step is I and the cost of looking up each answer in the database is D. Using equations (4) the probability and expected cost for the partial strategy $I_F D_F$ are

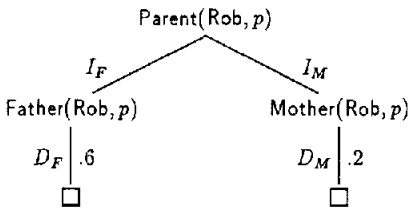

Fig. 4. Inference space for the goal Parent(Rob, p). Each step has been given a name, and nonzero probabilities are shown next to the corresponding branches.

$$P(I_F D_F) = L(I_F)P(D_F) = 0.6\,,$$

$$E(I_F D_F) = E(I_F) + L(I_F)E(D_F) = I + D\,.$$

Likewise, for the partial strategy $I_M D_M$ the probability and expected cost are

$$P(I_M D_M) = L(I_M)P(D_M) = 0.2\,,$$

$$E(I_M D_M) = E(I_M) + L(I_M)E(D_M) = I + D\,.$$

Using this information, together with equations (3), the probability and expected cost for the strategy $I_F D_F I_M D_M$ can be computed:

$$\begin{aligned} P(I_F D_F I_M D_M) &= P(I_F D_F) + P_0(I_F D_F)P(I_M D_M) \\ &= 0.6 + 0.4(0.2) = 0.68\,, \end{aligned}$$

$$\begin{aligned} E(I_F D_F I_M D_M) &= E(I_F D_F) + P_0(I_F D_F)E(I_M D_M) \\ &= (I + D) + 0.4(I + D) = 1.4(I + D)\,. \end{aligned}$$

Likewise, the probability and expected cost for the strategy $I_M D_M I_F D_F$ are

$$\begin{aligned} P(I_M D_M I_F D_F) &= P(I_M D_M) + P_0(I_M D_M)P(I_F D_F) \\ &= 0.2 + 0.8(0.6) = 0.68\,, \end{aligned}$$

$$\begin{aligned} E(I_M D_M I_F D_F) &= E(I_M D_M) + P_0(I_M D_M)E(I_F D_F) \\ &= (I + D) + 0.8(I + D) = 1.8(I + D)\,. \end{aligned}$$

The probability for the two is the same, as we would expect, and the expected cost for the latter is higher, since there is less chance of succeeding with the mother branch.

For the other four possible strategies the computation is a bit more involved.

Consider the strategy $I_F I_M D_F D_M$. First we use the equations for combining independent strategies (3) to compute the probability and expected cost for the partial strategy $I_F I_M$:

$$P(I_F I_M) = P(I_F) + P_0(I_F)P(I_M) = 0\,,$$

$$E(I_F I_M) = E(I_F) + P_0(I_F)E(I_M) = I + I = 2I\,.$$

Next, we use the general combination equation (5) to compute the probability and expected cost for the partial strategy $I_F I_M D_F$:

$$\begin{aligned} P(I_F I_M D_F) &= P(I_F I_M) + L(I_F)P_0(I_M)P(D_F) \\ &= 0 + 1(1)(0.6) = 0.6\,, \end{aligned}$$

$$\begin{aligned} E(I_F I_M D_F) &= E(I_F I_M) + L(I_F)P_0(I_M)E(D_F) \\ &= 2I + 1(1)D = 2I + D\,. \end{aligned}$$

Finally, we use the general combination equation again to compute the probability and expected cost for the strategy $I_F I_M D_F D_M$:

$$\begin{aligned} P(I_F I_M D_F D_M) &= P(I_F I_M D_F) + L(I_M)P_0(I_F D_F)P(D_M) \\ &= 0.6 + 1(1 - 0.6)(0.2) = 0.68\,, \end{aligned}$$

$$\begin{aligned} E(I_F I_M D_F D_M) &= E(I_F I_M D_F) + L(I_M)P_0(I_F D_F)E(D_M) \\ &= (2I + D) + 1(1 - 0.6)D = 2I + 1.4D\,. \end{aligned}$$

The computations for the other three remaining strategies are similar. The results are summarized in Table 2. As we would expect, the probability of

Table 2
Probability and expected cost of finding a solution to goals of the form Parent(Rob, p) for each of the six different strategies

Strategy	P	E
$I_F D_F I_M D_M$	0.68	$1.4(I + D)$
$I_F I_M D_F D_M$	0.68	$2I + 1.4D$
$I_F I_M D_M D_F$	0.68	$2I + 1.8D$
$I_M D_M I_F D_F$	0.68	$1.8(I + D)$
$I_M I_F D_M D_F$	0.68	$2I + 1.8D$
$I_M I_F D_F D_M$	0.68	$2I + 1.4D$

success is independent of the order, and the expected cost is lowest for the strategy $I_F D_F I_M D_M$.

2.4. Calculating worth

Recall that in Section 1.3 we defined the worth of an inference step as the utility of the best strategy beginning with that step;

$$W(x) \stackrel{\text{def}}{=} U(B(x)) = \max_{s \in S(x)} U(s) ,$$

where the utility of a strategy was defined as the ratio of the probability of success and the expected cost of the strategy. For each inference step x we could therefore enumerate every possible strategy in $S(x)$ (the subspace rooted at x), calculate the utility for each strategy (using the expected cost and probability equations developed in the previous section), and compare to find the best strategy.

Although this is an easy task for our simple kinship example, in general it is quite intractable. The number of legal strategies for a search tree is greater than the number of permutations of the leaf nodes of the tree. For a simple binary search space of uniform depth k, the number of leaf nodes is 2^k, so the number of legal strategies exceeds $2^k!$, which grows very rapidly indeed.

Fortunately, this combinatorial process is entirely unnecessary. In fact, as we will show in this section, the best strategies for all steps sanctioned by a given database can be found in time proportional to the number of rules in the database.

2.4.1. *Indivisible blocks*

The way that we find best strategies efficiently is to start at the bottom of the search space (the leaf nodes) and incrementally assemble pieces of a strategy that we can demonstrate must belong together.

The basic idea can be seen in Fig. 5. Here b and c are steps such that c is an immediate successor of b. Suppose that c has utility at least as high as b and at least as high as each of b's other descendants. Since c is the best of b's successors, and is at least as good as b itself, bc is better than b alone (see Lemma A.1). Thus, we say that b and c form an *indivisible block*. As we will show, the indivisible block bc belongs in a best strategy for b.

More formally, we define an *indivisible block* recursively as follows:[5]

Definition 2.8 (Indivisible block). Any individual step is an indivisible block. In addition, if b and c are indivisible blocks such that,

[5]The notion of an indivisible block, and this definition, closely follow those of Simon and Kadane [11].

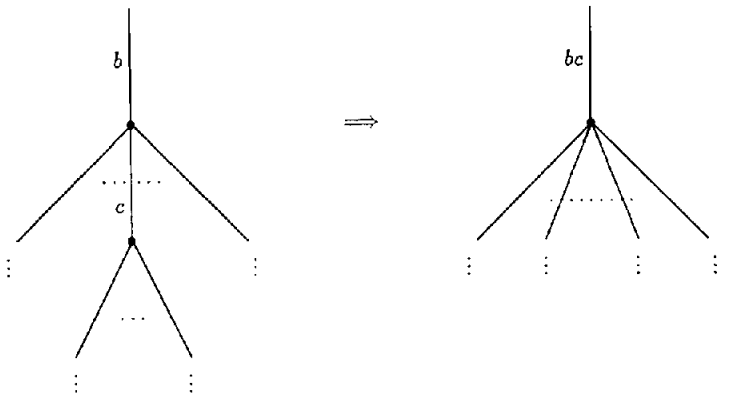

Fig. 5. Forming indivisible blocks.

(1) c is an immediate successor of b,
(2) $U(c) \geq U(b)$,
(3) for all other steps x that are successors of b, or descendants of those successors, either $U(c) \geq U(x)$, or $U(c) \geq U(x')$ where x is contained in x' and x' is an indivisible block,

then bc is also an indivisible block.

A *maximal indivisible block* is defined as an indivisible block all of whose successors have utility less than or equal to that of the block itself. (Maximal refers to utility of the block, not its size.) Note that there may be more than one maximal indivisible block for a given step. However, all maximal indivisible blocks for a step are the same up to interchange of substrategies with equal utility, or the addition of strategies with utility equal to that of the block.

Using these definitions, we can now state the principal result that allows us to compute worth efficiently.

Theorem 2.9 (Best strategy). *A best strategy for a step is a maximal indivisible block beginning with that step.*

The worth for a step will then be the utility of a maximal indivisible block beginning with that step. The proof of this theorem can be found in Appendix A.

2.4.2. *Example*

As an example of the construction of indivisible blocks, and computation of worth, consider the simple kinship problem shown in Fig. 6. Here the problem is to find a guardian of Rob. Assume that there are three father facts and one mother fact in the database as before. To make things more interesting we

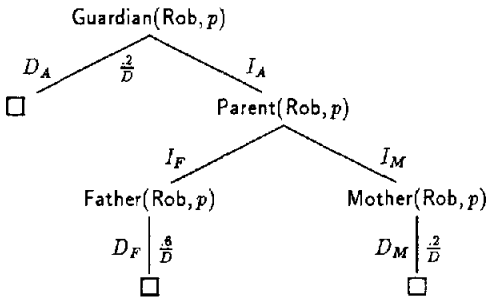

Fig. 6. Indivisible block example.

have included the additional axiom

$$\text{Parent}(x, p) \rightarrow \text{Guardian}(x, p) ,$$

and have added a single guardian fact to the database. Utility numbers (as calculated in Section 2.3.1) are shown next to the three database lookup steps. The utilities for the inference steps are zero, since they cannot produce an answer to the problem.

Initially, we note that the three database lookup steps are maximal indivisible blocks, since they do not have any successors. The worth of these steps is therefore just their utility.

Next, we construct the maximal indivisible blocks for the inference steps I_F and I_M. The step I_F has utility zero, and has only one descendant D_F, which has utility of $0.6/D$. According to our definition, this pair is an indivisible block. Since I_F has no other descendants, the block $I_F D_F$ is maximal. This block is therefore the best strategy for the step I_F and the utility of the block is the worth of the step I_F.

The analysis for the inference step I_M is similar; $I_M D_M$ is the maximal indivisible block for I_M so the worth for I_M is just the utility of this block. In Fig. 7 a reduced search space is shown, where the individual steps I_F, D_F, I_M, and D_M are replaced by the blocks $I_F D_F$ and $I_M D_M$. The utilities for the indivisible blocks (calculated in Section 2.3.1) are shown next to the corresponding branches.

We now consider the step I_A. The utility of the inference step I_A is zero, so both of its successor blocks, $I_F D_F$ and $I_M D_M$, have greater utility. The utility of $I_F D_F$ is greater than that of $I_M D_M$, so by our definition $I_A I_F D_F$ is an indivisible block. After performing this reduction the tree is as shown in Fig. 8.

The block $I_A I_F D_F$ has one remaining successor, the block $I_M D_M$. However, its utility is less than that of the block $I_A I_F D_F$. As a result, no further reduction is possible, and the block $I_A I_F D_F$ is maximal. The worth of the step I_A is therefore the utility of the strategy $I_A I_F D_F$.

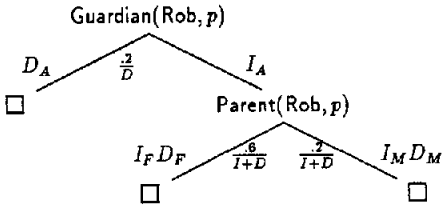

Fig. 7. First reduction for the kinship problem. The utility is shown next to each block where it is nonzero.

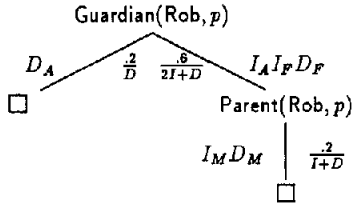

Fig. 8. Second reduction for the kinship problem.

The worths for all of the different steps in the space are summarized in Fig. 9.

2.4.3. *An algorithm*

By taking advantage of Theorem 2.9 it is possible to efficiently compute worth information for any given database and set of rules. Since we are assuming no recursive rules, the set of relations represented in the database and the rules containing those relations form a directed acyclic graph (DAG) with each rule forming an arc from its antecedent to its conclusion.[6] The basic process involves starting at the bottom of the graph (with those relation symbols that do not appear in the conclusion of any rule) and working upwards through the graph, building indivisible blocks and storing worth information with each rule. For example, suppose we start with a relation R that appears in the database, but does not appear in the conclusion of any rule. We first compute the worth of database lookups on R for all possible ways of binding the variables in R. We then compute the worth for all rules that contain R in their premise. For each of these rules we consider the relation Q appearing in its conclusion. As

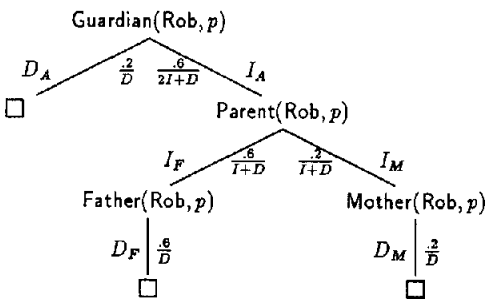

Fig. 9. Worth numbers for steps in the kinship problem.

[6] More generally, a rule containing a conjunctive premise will form a bundle of arcs from its antecedent clauses to its consequent clause. However, in this section we are only considering disjunctive search spaces.

before, we compute the worth of database lookups on Q. But unlike the first case, Q is not at the bottom of the DAG, so there may be other rules containing Q in their conclusion. Before we can proceed upward we must compute the worth for these rules, if it is not already known. To do this, we recursively descend to the bottom of the DAG again, and work upwards along each of these paths. Once we have finished this process for all rules containing Q in their conclusion, we then move upwards and compute the worth of all rules containing Q in their premise. When the top of the graph is reached the process is complete. A detailed version of the algorithm can be found in Appendix B.

Theorem 2.10. *The algorithm described above is effectively linear in the number of rules and relation symbols in a database. Each database relation is checked once, and for each rule we check each of its immediate successors once.*[7]

Although there is some extra storage involved in keeping a list of each block's successors during indivisible block construction, the amount of storage required is also linear in the number of relations and rules in the database.

2.4.4. *Incremental updating*

In the previous section we described an algorithm for computing the worth information for all rules in a database. As it turns out, we can update this information in an incremental fashion as new facts and rules are added to, or removed from the database. The basic procedure involves starting with the relation involved in the database update, recomputing the worth for queries of that relation, and as before, working upwards through the DAG recomputing the worth for rules that are affected by the relation.

However, there are two additional observations that allow this updating to be performed in a much more efficient manner:

(1) Most database additions and changes do not change the overall constitution of the database very much.

(2) A small change in the worth of an inference step will always translate into an even smaller change in the worth of its parent steps. (This is a direct result of Lemma A.1.)

As a result, most changes to a database will not need to be propagated very far in the worth network. Instead we can wait until the cumulative effect of many changes becomes significant, before further propagating changes in the worth network.

[7] To apply the reduction theorem it is necessary to sort the successors of each step according to worth. The sorting requires $n \log n$ time in the number of successors, but normally the number of successors for a block is small compared to the total number of rules in the database.

To be more precise, suppose that a new fact containing the relation R is added to the database. We first compute new information about the chance of success for database lookup of queries involving that relation. Suppose that the new numbers aren't much different than the old ones (say within 10%). In this case, we simply discard the new numbers. Alternatively, if the change is a significant one (or the net effect of several consecutive changes is significant in comparison to the original numbers), we compute new worth information for all rules that contain this relation in their premise. If these worth numbers do not change significantly, we again discard them and stop propagating. If any of the numbers change significantly, we compute new worth information for all rules that have the conclusion relation in their premise, and so on.

The effect of this approach is to batch together related changes to the database until the net effect on the worth network is significant.

2.5. Optimal inference

As hinted in Section 1.3 if an inference procedure chooses the step of greatest worth at each stage in the problem solving process it will follow the optimal strategy. We can now make this precise.

Theorem 2.11 (Optimality). *The complete strategy with the lowest expected cost for solving a problem consists of the steps in the space ordered by decreasing worth (respecting the allowed partial ordering on steps).*

The proof of this theorem can be found in Appendix A.

As an example, we return to the simple kinship problem from the previous section. The worth information for this example was computed and summarized in Fig. 9. For this example, there are two possibilities. If the cost of doing inference is higher than the cost of looking up answers in the database, i.e. $I > D$, the worth of the step D_A is greater than the worth of the step I_A. Thus the optimal strategy begins with D_A. After D_A there is only one other step possible, I_A, so it will be the second step in the strategy. Once I_A is performed I_F and I_M become possible. I_F has the highest worth of the two. After I_F is performed D_F is possible. Since D_F has higher worth than I_M, it is chosen next. Finally, I_M and D_M are chosen. Thus, for this case the optimal strategy is $D_A I_A I_F D_F I_M D_M$, as illustrated in Fig. 10.

Alternatively, if the cost of database reference is higher than the cost of doing inference, i.e. $I < D$, I_A has the highest worth initially. After it is performed I_F, I_M, and D_A are all possible. I_F has higher worth than either I_M or D_A so it is chosen next. D_F then becomes possible, and since it has the highest worth, it is selected. Of the two remaining possible steps, D_A has the highest worth, so D_A is performed, followed by I_M and finally D_M. Thus, for this case the optimal strategy is $I_A I_F D_F D_A I_M D_M$, as illustrated in Fig. 11.

Intuitively, this result is very satisfying. If the cost of database lookup is very

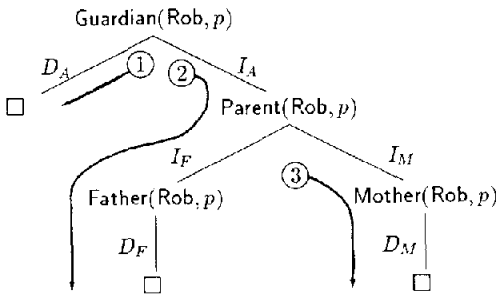

Fig. 10. Optimal ordering for the case $I > D$.

high, we want to try the database lookup that has the highest possible chance of answering the question. For this example, this is the father branch. The other two branches have equal chance of success, so the cheaper one is taken first.

An interesting thing to note about these strategies is that they both have a sort of depth-first character to them. Intuitively this makes sense; there is no point in doing an inference step unless we intend to follow up on it immediately. As it turns out, a system choosing inference steps on the basis of greatest worth will always have this depth-first character. Since inference steps have utility zero, the maximal indivisible block for an inference step will always contain at least one database lookup step that lies below it in the space. As a consequence every inference step has at least one successor with worth greater than or equal to that of the step itself. If an inference step is chosen, a successor will have greater worth and will be chosen next. The inference procedure will therefore go all the way to the end of at least one branch beginning with that step.

This depth-first character can be important in some applications, because it reduces the amount of storage required to keep track of the set of active subgoals. It also means that the reasoning process will appear coherent to an

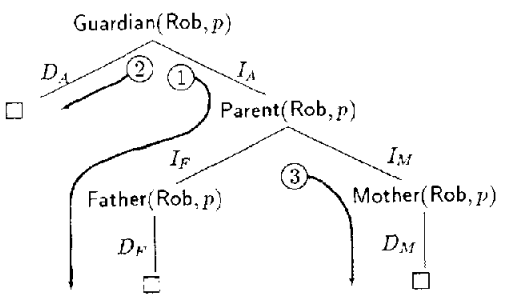

Fig. 11. Optimal ordering for the case $D > I$.

outside observer. This is often important if the system asks questions of the user, or must give explanations of its reasoning behavior.

3. Finding Multiple Answers

In the previous section we showed how to assign a worth number to each possible inference step so that a best-first search based on this worth information would result in following the optimal strategy (the strategy with lowest expected cost) for problems where only a single answer was needed.

Unfortunately, the optimal strategy for finding a single answer is not necessarily the same as the optimal strategy for finding n answers. To see this, consider a simple search space containing two branches a and b, as shown in Fig. 12. Suppose that branch a is cheap and is likely to produce a single answer, while branch b is expensive but is likely to produce two or more answers. If the problem involves finding only one answer, the optimal strategy is to first do a (because it is cheaper), followed by b if a fails. However, if the problem involves finding two answers, the optimal strategy is to do b first. This is because we will have to explore b anyway in order to obtain two answers. If b succeeds a will not even be necessary, so there is no reason to waste time exploring it first.

3.1. The optimal strategy

In actuality, the optimal strategy for finding n answers to a problem may be a *conditional strategy*. By this we mean that decisions about what step to choose may depend upon how many answers were found using previous steps. As an example of this phenomenon consider the simple search space containing three branches, a, b and c, as shown in Fig. 13. Suppose that branch a is cheapest and has probability 0.7 of producing a single answer, branch b is more expensive but is guaranteed to produce an answer, and branch c is very expensive but is guaranteed to produce three answers. Suppose the problem involves finding two answers. Because of its low cost, a is the optimal first step. If an answer is found, b is the best next step. However, if an answer is not found using a, c must be done anyway in order to solve the problem, so c is the best next step.

In this section we will first show a negative result, that we cannot define worth numbers W_n that will yield the optimal strategy when n answers are required. We will then concentrate on two promising approximations for W_n

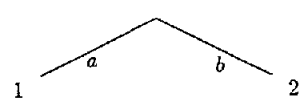

Fig. 12. Multiple answer example.

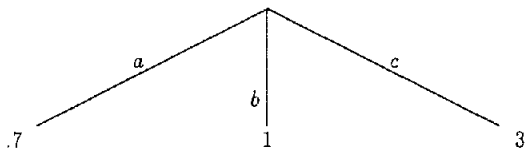

Fig. 13. Conditional strategies.

and offer some conjectures about the worst case performance of these approximations.

3.2. Undefinability of W_n

There is a fundamental problem with trying to assign numbers W_n to inference steps for the case where more than one answer is sought. The trouble is that the worth of a step no longer depends only on the steps that lie below it in the search space. In the example given in Fig. 12, the step a would have been valuable for the overall problem of finding two answers to the query if there were another cheap step c that could produce a single answer. This unfortunate property leads to a very pessimistic theorem about any attempt to define such a quantity:

Theorem 3.1 (Undefinability). *There is no possible way of assigning numbers W_n to possible inference steps in a database so that a best-first search based on W_n will always result in the lowest expected cost for finding $n > 1$ solutions.*

Proof. Consider the example shown in Fig. 14. Suppose that step b can produce one answer with certainty, and has a very small probability, 0.001, of producing a second answer, step c can produce two answers with certainty, and step d produces only one answer with certainty. Furthermore suppose that the cost of steps a and b are small, $E(a) = E(b) = 1$, the cost of d is significantly more expensive, $E(d) = 10$, and the cost of c is exceptionally expensive, $E(c) = 100$.

Now suppose we were trying to find two answers to the problem H. Because the chance of finding a second answer using b is remote it is likely that we will have to try c anyway. As a consequence, the optimal strategy is to try c first, making it unnecessary to ever try step b. For a best-first search procedure to

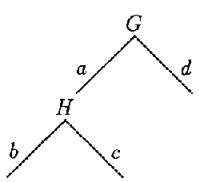

Fig. 14. Undefinability example.

find this strategy using worth information we would have to define W_2 so that $W_2(c) > W_2(b)$.

Now suppose that we were trying to find two answers to the problem G. In this case we could use step b to find the first answer, and step d to find the second. Since step b is cheaper, and has a small chance of finding a second answer by itself, the optimal strategy for this case is *abdc*. For a best-first search procedure to find this strategy $W_2(b) > W_2(d) > W_2(c)$. But this contradicts the earlier requirement that $W_2(c) > W_2(b)$. □

The above result is actually unduly pessimistic. Although we can't define W_n to achieve optimal behavior, there are some possible definitions for W_n that lead to reasonable behavior under most circumstances, and seem to have bounded worst case behavior.

3.3. Using the single solution strategy

The simplest definition to try for W_n is to set it to W_1, the worth information for finding single solutions. As illustrated earlier, this can be suboptimal. But the question is, how bad is it to take this approach? As it turns out, the answer depends upon how we define the problem of finding n answers. Suppose the object is to find as many answers as possible, up to n. Under this *persistence assumption* it appears that there are reasonable bounds on the performance of this approach.

Conjecture 3.2. *Under the persistence assumption, if the optimal strategy for finding a single answer, o_1, is used for finding n answers, the expected cost will be less than n times as large as the expected cost if the optimal strategy for finding n answers, o_n, were used:*

$$E_n(o_1) < nE_n(o_n) .$$

A proof of this conjecture has not yet been found, but our reason for believing it can be seen by considering the following example. Imagine that we have a collection of n possible steps, each of which has a very high probability of producing a single answer and has expected cost E. The utility for each of these steps will be $1/E$. Suppose that we also have a step that will produce n answers, but has cost slightly higher than E. The utility of this step is $1/(E + \varepsilon)$. The optimal strategy for finding a single solution will therefore consist of the sequence of n single solution steps followed by the more expensive step. If we use this strategy for finding n solutions we will end up doing the first n steps in the strategy for an expected cost of nE. In contrast, the optimal strategy for finding n solutions will produce an expected cost of $E + \varepsilon$, where ε can be made arbitrarily small.

This example seems to illustrate the worst possible behavior for this ap-

proach. If any of the single solution steps became more expensive they would be delayed until after the multiple solution step. Alternatively, if any of the single solution steps have a chance of producing more than one answer, the expected cost of the entire strategy would be decreased. In the typical case the optimal strategy for finding a single solution will therefore perform much better than indicated by the above conjecture. In fact, for many examples, it seems that the two strategies o_1 and o_n are identical.

To see why the persistence assumption is necessary in the above conjecture, consider the following example: Suppose we have two different possible strategies for solving a problem, a and b. Let a be certain of finding a single solution, $P(a) = 1$, but with high expected cost $E(a) = 1/\varepsilon$. In contrast let b be a strategy that will either find two solutions, or none at all, with probability of success $P(b) = \varepsilon$, but very low cost $E(b) = 2$. Using this information we note that $U(a) = \varepsilon$ while $U(b) = \frac{1}{2}\varepsilon$. As a result the better strategy for finding a single solution is to try a before b. However, if we were looking for two answers we must try b in any case, so we should try it first. If b finds two answers we will be done. Alternatively, if b fails to find any solutions, the entire enterprise has failed, since a cannot possibly produce two answers. No matter what happens with b we should never attempt a. The optimal strategy for finding two answers therefore consists of b alone. The expected cost of this strategy is $E(b) = 2$. In contrast, the expected cost of the best strategy for finding a single answer is $E(ab) = 1/\varepsilon + 2$, which can be made arbitrarily large.

While the persistence assumption is generally a reasonable one, it is worth noting that, even in the absence of this assumption, the single answer approximation will generally perform well, as long as the majority of queries do not ask for more answers than can be provided.

3.4. Improving on W_1

The principal deficiency of using W_1 for problems where n answers are needed is that W_1 does not distinguish between steps that will provide only one solution, and those that can provide many solutions with almost no additional work. If we are searching for many solutions, the latter would be much more desirable. In this section we will show an approximate definition for W_n, that takes this distinction into account.

In order to do this we first need to introduce some additional notation. We use $P_{\geqslant n}(s)$, $P_n(s)$, and $P_{<n}(s)$ to refer to the probability that the strategy s will produce at least n answers, exactly n answers, and fewer than n answers to a particular problem. Note that

$$P_n(s) = P_{\geqslant n}(s) - P_{\geqslant n+1}(s)\,,$$

$$P_{<n}(s) = 1 - P_{\geqslant n}(s)\,.$$

Similarly, we use $E_n(s)$ to refer to the expected cost of using the strategy s to find n answers to a particular problem. For convenience, we will abbreviate $P_{\geq 1}(s)$ by $P(s)$ and $E_1(s)$ by $E(s)$.

In Appendix C we show how $P_{\geq n}$ and E_n can be computed for a strategy. The development is analogous to that in Sections 2.2 and 2.3, although the details are mathematically more complex.

We next define the *strict* utility of a strategy for finding n answers as

$$U_n(s) \stackrel{\text{def}}{=} P_{\geq n}(s)/E_n(s) .$$

However, the strict utility is not really the quantity we are interested in. The reason is that it does not take into account the potential usefulness of a strategy that only produces some fraction of the total number of desired answers for a problem. Consider the example in Fig. 12 again. For the step a we would have $U_2(a) = 0$, even though a is potentially useful for problems where two or more answers are sought.

As a result, we define the *relative utility* of a strategy for finding n answers as

$$U'_n(s) \stackrel{\text{def}}{=} \max_{i \leq n} iU_i(s) = \max_{i \leq n} i(P_{\geq i}(t)/E_i(t)) .$$

The extra factor of i in this definition accounts for the fact that a strategy producing n answers will be n times as valuable as one that only produces one answer if both have the same expected cost and chance of success. We then define the best strategy and worth as before

$$B_n(x) \stackrel{\text{def}}{=} s: \max_{s \in S(x)} U'_n(s) ,$$

$$W_n(x) \stackrel{\text{def}}{=} \max_{s \in S(x)} U'_n(s) .$$

We can then use W_n to guide a best-first search procedure when looking for n answers to a problem. However, if k answers are found using a particular step, we should base subsequent search decisions on W_{n-k} rather than W_n. (This lends a conditional flavor to the search.) Using best-first search in this way appears to guarantee behavior that is very close to optimal.

Conjecture 3.3. *Under the persistence assumption, let o_n refer to the optimal (conditional) strategy for finding n answers to a problem. Let o^* refer to the (conditional) strategy that would be followed by using best-first search based on W_i, where i is the remaining number of answers needed at each stage of the search. The expected cost of using o^* to find n answers will be within a factor of two of the expected cost of using the optimal strategy:*

$$E_n(o^*) < 2E_n(o_n)$$

No proof of this conjecture has been found either.[8] Our reason for believing it can be seen by considering an apparent worst case example. Suppose that we have $n-1$ steps, $s_1, \ldots, s_{n-1}$, that will produce a single answer, and have expected cost E. For these steps

$$U'_k(s_i) = U(s_i) = 1/E .$$

Suppose that we also have a step, t, that produces n answers but costs $n(E+\varepsilon)$. For this step

$$U'_n(t) = \frac{n}{n(E+\varepsilon)} = \frac{1}{E+\varepsilon} .$$

As a result the strategy that would be chosen by best-first search on W_n would be $s_1, \ldots, s_{n-1}, t$ having expected cost $E_n = (n-1)E + n(E+\varepsilon)$. The optimal strategy is $t, s_1, \ldots, s_{n-1}$ having expected cost of $E_n = n(E+\varepsilon)$. For large n and small ε the difference between these two strategies approaches a factor of 2.

3.5. Computational considerations

So far we have concentrated solely on the theory for developing a good evaluation function W_n when n answers are needed. We have ignored the serious problem of computing W_n in a tractable fashion.

For the approximation $W_n = W_1$ (discussed in Section 3.3), no additional computation or storage is required beyond that discussed in Section 2 for single answer queries.

For the more sophisticated definition developed in Section 3.4 the picture is not so rosy. Computing W_n in a naive way would require a search through all possible strategies lying below each inference step. As discussed in Section 2.4 this is computationally intractable for any reasonably sized database. In Section 2.4 we got around this problem by developing an incremental method for assembling indivisible blocks. This meant that we could calculate W_1 for the implications in a database in time proportional to the number of such implications.

Unfortunately our indivisible block construction does not apply so readily to finding B_n. The trouble is that Theorem 2.9 no longer applies in the n solution case. In the diagram of Fig. 5 it is only when $U_i(y) \geq U_i(x)$ and $U_i(y) \geq U_i(z)$ for all i and all other descendants z that we can actually collapse the steps x and y into a single indivisible block. In other words, the entire utility

[8] The proof is likely to be quite hard because of the conditional nature of these strategies. Computing the expected cost of a conditional strategy involves weighting the expected cost of each branch of the conditional strategy by the probability that the branch will be chosen.

distribution for y must be greater than the other utility distributions. Otherwise we must construct indivisible blocks separately for different values of n. It is not clear whether effective reduction theorems can be found that will allow the incremental construction of indivisible blocks for $n > 1$.

An alternative to computing W_n accurately is to approximate it by $U'_n(B_1(x))$. This still requires calculating and storing $P_{\geq n}$ and E_n while assembling maximal indivisible blocks, but avoids the search that might otherwise be required to find best strategies for n answers.

It appears that this approximation is a sensible one. It is fairly easy to show that

$$W_1(x) \leq U'_n(B_1(x)) \leq W_n(x) \leq nW_1(x) .$$

In other words, the approximation lies between the actual value for W_n and the approximation of using W_1. Recall that the improvement W_n gives over W_1 results from elevating the worth of steps that lead to more than one answer in proportion to the number of answers produced. This effect is still present in the approximation, although the approximation will not take account of strategies other than B_1. As a result, the approximation will generally yield better performance than simply using W_1 but will not perform quite as well as if W_n were calculated accurately.

4. Conjunctions

4.1. Conjunctive strategies

In Section 2.1 we defined a strategy as being any legal sequence of steps. What exactly does this mean for solving conjunctions? Consider the AND/OR space for the conjunctive portion of the kinship example reproduced in Fig. 15. The strategies often used in solving conjunctions involve first finding a solution to one conjunct, substituting this solution into the remaining conjuncts, finding a solution to a second conjunct, and so forth. If a solution cannot be found to a

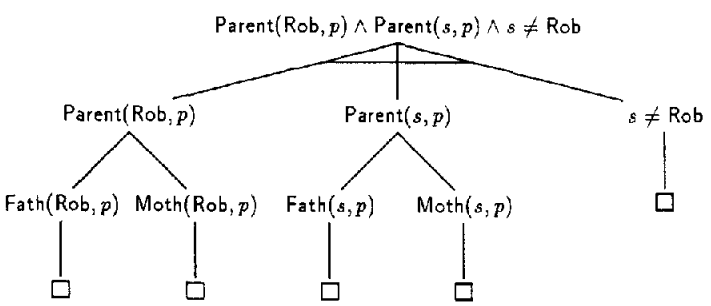

Fig. 15. AND/OR space for the conjunction in the kinship problem.

conjunct, backtracking occurs, and an additional solution must be found to the preceding conjunct. In the kinship problem, one such strategy would be to first look for a solution to the conjunct Parent(Rob, p), using the optimal strategy for finding a single solution to goals of the form $\text{Parent}^{\text{bf}}$, then to look for a solution to the conjunct Parent(s, p) (where p is now bound) using the optimal strategy for finding a single solution to goals of the form $\text{Parent}^{\text{fb}}$, and so forth. We will refer to such strategies as *serial* conjunctive strategies.[9]

In reality, serial strategies constitute only a small fraction of the class of all possible strategies for solving a conjunction. In the kinship example, we might first choose to reduce the first conjunct to its two subgoals, then reduce the second conjunct to its two subgoals, before actually doing a database lookup on any of the subgoals. Strategies like this are quite sensible for a conjunctive problem where lots of backtracking is expected. This is because the inference steps for later conjuncts do not have to be repeated for each additional answer found to earlier conjuncts.

Another case where nonserial strategies can be important is when one or more of the conjuncts lead to conjunctive subgoals. In this case the best strategy may be to interleave work on some of the subconjuncts with work on other conjuncts, or on their subconjuncts.

As an example, consider the problem of finding a red sports car,

$$\text{find } x\text{: } \text{SportsCar}(x) \wedge \text{Red}(x) .$$

Assume we are given a database containing cars, red things, acceleration data for various artifacts, and the rules

$$\text{Car}(x) \wedge \text{Fast}(x) \rightarrow \text{SportsCar}(x) ,$$

$$\text{Acceleration}(x, y) \wedge y \geq 10\ \text{ft/s}^2 \rightarrow \text{Fast}(x) .$$

If we were looking only for sports cars, the best strategy would be to enumerate the cars (provided there were only a small number), and then check each one to see whether or not it is fast. Likewise, if facts about sports cars were available directly in the database, and we were looking for a red sports car, the best strategy would be to look through the sports cars and check each one to see if it is red (as opposed to looking through the set of all red things and checking each to see if it is a sports car). However, given that we must compute the set of sports cars, the best strategy is to map through the set of cars, throw out the ones that are not red, and then finally check to see if each remaining car is fast or not. This is because it is cheaper to check and see whether something is red by database access, than to determine whether or not

[9] This term is due to Korf [5].

it is fast by doing inference. Thus, in this case, the best strategy involves working on the Red(x) conjunct in between the two subconjuncts Car(x) and Fast(x).

In theory, there is nothing to prevent us from considering such nonserial strategies. Using the equations developed in Section 2 we could evaluate the expected cost for each one, and compare to find the best strategy for processing a conjunction. Unfortunately, unless reduction theorems can be developed that drastically limit the number of nonserial strategies that must be considered, this is computationally intractable. For example, in our simple kinship example, there are well over 100 different nonserial strategies. In contrast, there are only six serial strategies that must be considered, corresponding to the six different orderings for the conjuncts. As a consequence, in this paper we will consider only serial strategies for processing conjuncts.

4.2. Computing P and E for conjunctions

Consider the problem of calculating the probability of success and expected cost for finding a single answer to the conjunction $A \wedge B$. Let a and b be the best strategies for finding solutions to A and B independently. Let b' be the best strategy for finding solutions to B after the variables in A have been bound. Assume that the conjuncts will be solved in the order AB, and that the strategy a will be used for finding solutions to A, and b' will be used for finding solutions to B once a solution to A has been found (see Fig. 16).

Using the notation introduced in Section 3.4 we can express the expected cost and probability of success for solving the conjunction $A \wedge B$ using the conjunctive strategy ab'. To start with there is probability $P(a)$ that at least one solution will be found to A using a. If a solution is found to A, there will be probability $P(b')$ that a solution will be found to B for those bindings. So the first term for $P(ab')$ is the product of these two probabilities. There is probability $P_0(b')$ that a solution to B cannot be found for the first solution to A found by a. In this case there is probability $P_{\geq 2}(a)$ that a second solution to A can be found using a, and probability $P(b')$ that a solution to B will be found for this second solution to A. Continuing in this fashion, we get:

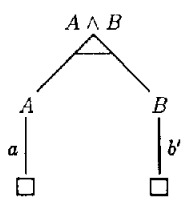

Fig. 16. Simple conjunctive inference space where the strategy a is used on the conjunct A and the strategy b' is used on the conjunct B.

$$P(ab') = P_{\geqslant 1}(a)P(b') + P_0(b')P_{\geqslant 2}(a)P(b') + P_0(b')^2 P_{\geqslant 3}(a)P(b') + \cdots$$
$$= P(b') \sum_{i=0}^{\infty} P_0(b')^i P_{\geqslant i+1}(a) . \tag{6}$$

The development for $E(ab')$ is similar. To start with, we have the cost $E(a)$ of finding the first solution to A. There is probability $P(a)$ that this will be successful. If so, there will be the additional cost $E(b')$ of trying to find a solution to B. If this fails we will accrue the additional cost, $E_2(a) - E(a)$, of finding a second solution to A using a, and so forth. Letting $\Delta E_i(s) \stackrel{\text{def}}{=} E_i(s) - E_{i-1}(s)$, we get:

$$E(ab') = [E(a) + P(a)E(b')] + P_0(b')[P_{\geqslant 1}(a)\,\Delta E_2(a) + P_{\geqslant 2}(a)E(b')] + P_0(b')^2[P_{\geqslant 2}(a)\,\Delta E_3(a) + P_{\geqslant 3}(a)E(b')] + \cdots$$
$$= \sum_{i=0}^{\infty} P_0(b')^i [P_{\geqslant i}(a)\,\Delta E_{i+1}(a) + P_{\geqslant i+1}(a)E(b')]$$
$$= E(a) + E(b')\frac{P(ab')}{P(b')} + \sum_{i=1}^{\infty} P_0(b')^i P_{\geqslant i}(a)\,\Delta E_{i+1}(a) . \tag{7}$$

4.2.1. *Example*: *Exact calculation*

Using the above equations, together with data for E_n and $P_{\geqslant n}$ computed in Appendix C, and summarized in Tables 5 and 6, we can compute the probability of solving the conjunction Parent(Rob, p) $\wedge$ Parent(s, p) using either the strategy $S_{P_{\text{bf}}}S_{P_{\text{fb}}}$ or the strategy $S_{P_{\text{ff}}}S_{P_{\text{bb}}}$ (the probability is the same for the two strategies).

$$P(S_{P_{\text{bf}}}S_{P_{\text{fb}}}) = P(S_{P_{\text{fb}}}) \sum_{i=0}^{\infty} P_0^i(S_{P_{\text{fb}}})P_{\geqslant i+1}(S_{P_{\text{bf}}})$$
$$= 0.6[0.68 + 0.4(0.12)] = 0.44 .$$

Using this information, and equation (7), the expected cost can be computed for the two strategies $S_{P_{\text{bf}}}S_{P_{\text{fb}}}$ and $S_{P_{\text{ff}}}S_{P_{\text{bb}}}$.

$$E(S_{P_{\text{bf}}}S_{P_{\text{fb}}}) = E(S_{P_{\text{bf}}}) + E(S_{P_{\text{fb}}})\frac{P(S_{P_{\text{bf}}}S_{P_{\text{fb}}})}{P(S_{P_{\text{fb}}})} + \sum_{i=1}^{\infty} P_0^i(S_{P_{\text{fb}}})P_{\geqslant i}(S_{P_{\text{bf}}})\,\Delta E_{i+1}(S_{P_{\text{bf}}})$$
$$= 1.4(I + D) + 1.5(I + D)\frac{0.44}{0.6} + 0.4(0.68)(0.6)(I + D)$$
$$= 2.65(I + D) ,$$

$$E(S_{P_{ff}}S_{P_{bb}}) = E(S_{P_{ff}}) + E(S_{P_{bb}})\frac{P(S_{P_{ff}}S_{P_{bb}})}{P(S_{P_{bb}})} + \sum_{i=1}^{\infty} P_0^i(S_{P_{bb}})P_{\geq i}(S_{P_{ff}})\,\Delta E_{i+1}(S_{P_{ff}})$$

$$= (I + D) + 1.88(I + D)\frac{0.44}{0.16} + (0.84D + 0.84^2D + 0.84^3(I + D))$$

$$= 6.3I + 8.4D\,.$$

Since $E(S_{P_{bf}}S_{P_{fb}}) < E(S_{P_{ff}}S_{P_{bb}})$, the strategy $S_{P_{bf}}S_{P_{fb}}$ is preferable to the strategy $S_{P_{ff}}S_{P_{bb}}$.

4.2.2. *Approximate calculations*

As a practical matter, equations (6) and (7) are difficult to use, because they require computation of $P_{\geq i}$ and E_i for $i > 1$. However, there are some simple approximations that will make the computation much easier. First, we make the assumption that $\Delta E_i(a) \approx E(a)$; i.e. that the cost for each additional answer using a is the same as that of finding the first answer. Using this approximation, equation (7) becomes:

$$E(ab') \approx E(a) + E(b')\frac{P(ab')}{P(b')} + E(a)P_0(b')\frac{P(ab')}{P(b')}$$

$$= \left[1 + \frac{P(ab')P_0(b')}{P(b')}\right]E(a) + \frac{P(ab')}{P(b')}E(b')\,. \tag{8}$$

Using this approximation we also get a fairly simple expression for utility:

$$U(ab') \stackrel{\text{def}}{=} P(ab')/E(ab')$$

$$\approx \frac{P(ab')}{\left[1 + \frac{P(ab')P_0(b')}{P(b')}\right]E(a) + \frac{P(ab')}{P(b')}E(b')}$$

$$= \frac{1}{P(a)\left[\frac{P_0(b')}{P(b')} + \frac{1}{P(ab')}\right]\frac{1}{U(a)} + \frac{1}{U(b')}}\,. \tag{9}$$

It is interesting to note that both (8) and (9) are monotonically increasing in $P(ab')$. That is, if $P(ab')$ increases, so does the cost $E(ab')$ and the utility $U(ab')$.

From (9), we can derive useful bounds on utility:

Theorem 4.1. *If*

$$\chi \stackrel{\text{def}}{=} P(a)\left[\frac{P_0(b')}{P(b')} + \frac{1}{P(ab')}\right]$$

we get

$$\frac{\min\{U(a), U(b')\}}{1+\chi} \leqslant U(ab') \leqslant \frac{\max\{U(a), U(b')\}}{1+\chi}$$

and

$$U(ab') \leqslant \min\left\{\frac{U(a)}{\chi}, U(b')\right\}.$$

Proof. For the first inequality there are two cases:

Case 1: $U(a) \geqslant U(b')$.

$$\frac{\min\{U(a), U(b')\}}{1+\chi} = \frac{U(b')}{1+\chi} = \frac{U(a)U(b')}{U(a)+U(a)\chi} \leqslant U(ab')$$

$$\leqslant \frac{U(a)U(b')}{U(b')+U(b')\chi} = \frac{U(a)}{1+\chi} = \frac{\max\{U(a), U(b')\}}{1+\chi}.$$

Case 2: $U(a) \leqslant U(b')$.

$$\frac{\min\{U(a), U(b')\}}{1+\chi} = \frac{U(a)}{1+\chi} = \frac{U(a)U(b')}{U(b')+U(b')\chi} \leqslant U(ab')$$

$$\leqslant \frac{U(a)U(b')}{U(a)+U(a)\chi} = \frac{U(b')}{1+\chi} = \frac{\max\{U(a), U(b')\}}{1+\chi}.$$

For the second inequality, since $U(a)$, $U(b')$, and χ are all positive,

$$\frac{U(a)U(b')}{U(a)+U(b')\chi} \leqslant \frac{U(a)U(b')}{U(b')\chi} = \frac{U(a)}{\chi}.$$

Likewise,

$$\frac{U(a)U(b')}{U(a)+U(b')\chi} \leqslant \frac{U(a)U(b')}{U(a)} = U(b'). \qquad \square$$

Next, we note that it is possible to set fairly tight bounds on the quantity $P(ab')$:

Theorem 4.2. $P(a)P(b) \geqslant P(ab') \geqslant P(a)P(b')$.

Proof. The right-hand inequality follows trivially since $P(a)P(b')$ is just the first term in the expansion for $P(ab')$, and all terms in the expansion are positive. The left-hand inequality follows because the chance of finding a

solution to the conjunction must be less than the chance of finding solutions to each of the two conjuncts taken independently. □

Note that these bounds on $P(ab')$ are quite easy to compute. We can then use this information in equations (8) and (9) to compute bounds on $E(ab')$ and $U(ab')$. As a practical matter, we expect to use the lower bound estimates for probability of success and utility. This will tend to penalize conjunctions slightly, which is not unreasonable, because there is extra overhead involved in performing the backtracking and variable binding required for solving conjunctions.

4.2.3. *Example: Approximate calculations*

Using Theorem 4.2, we get:

$$P(S_{P_{\mathrm{bf}}})P(S_{P_{\mathrm{ff}}}) \geq P(S_{P_{\mathrm{bf}}}S_{P_{\mathrm{ff}}}) \geq P(S_{P_{\mathrm{bf}}})P(S_{P_{\mathrm{fb}}})$$

or

$$0.68 \geq P(S_{P_{\mathrm{bf}}}S_{P_{\mathrm{ff}}}) \geq 0.68(0.6) = 0.41 .$$

Note that the lower bound is quite close to the actual value of 0.44 computed earlier.

Using these bounds in (8) we get:

$$3.73(I + D) \geq E(S_{P_{\mathrm{bf}}}S_{P_{\mathrm{fb}}}) \geq 2.8(I + D)$$

and

$$13.38(I + D) \geq E(S_{P_{\mathrm{ff}}}S_{P_{\mathrm{bb}}}) \geq 8.46(I + D) .$$

Again, we note that since $E(S_{P_{\mathrm{bf}}}S_{P_{\mathrm{fb}}}) < E(S_{P_{\mathrm{ff}}}S_{P_{\mathrm{bb}}})$, the strategy $S_{P_{\mathrm{bf}}}S_{P_{\mathrm{fb}}}$ is preferable to the strategy $S_{P_{\mathrm{ff}}}S_{P_{\mathrm{bb}}}$.

Finally, the utilities of the two strategies can be calculated using equation (9):

$$0.18/(I + D) \geq U(S_{P_{\mathrm{bf}}}S_{P_{\mathrm{fb}}}) \geq 0.15/(I + D)$$

and

$$0.052/(I + D) \geq U(S_{P_{\mathrm{ff}}}S_{P_{\mathrm{bb}}}) \geq 0.05/(I + D) .$$

4.3. Finding the best ordering

Using the equations developed above we could build up the expected cost for every possible ordering in a conjunction. The best ordering would then be the cheapest one. This is not unreasonable if there are only a few conjuncts, as in the kinship example. However, for a large conjunction the number of possibilities becomes prohibitive.

Our solution is to do a best-first search of the space of possible orderings.

We start with a null conjunction and gradually build up good candidate orderings from back to front, using utility to choose the best candidate sequence at each iteration. More precisely, if c is the set of conjuncts to be ordered we use the following procedure:

Order(c):
  Candidates $\leftarrow \{\}$
  $d \leftarrow \langle\rangle$
  Until $c - d = \emptyset$ do:
    For each $x \in c - d$ do:
      Candidates $\leftarrow$ Candidates $\cup\ x|d$
    $d \leftarrow x \in$ Candidates: max $U(x|_{c-x})$
    Candidates $\leftarrow$ Candidates $- d$
  Return(d)

There are several important things to note about this algorithm. When the utility of a candidate sequence is evaluated, all variables contained in conjuncts not yet part of the sequence are assumed to be bound. This is because those variables will be bound in any completion of the candidate. Intuitively, what this evaluation function does is to prefer orderings where easy conjuncts (after variable binding) are postponed until the end. This tends to minimize the amount of backtracking that will need to take place, and, as a result, tends to lead to the best ordering rapidly.

The termination condition for this algorithm is that the candidate with highest utility contain all of the conjuncts. From Theorem 4.1 we know that adding another conjunct to the front of any candidate will always cause the utility of the candidate to go down. This guarantees admissibility.

To see how this search works, consider our simple kinship example again. Initially, there are three conjuncts that we could put at the end of the sequence, so we generate these three possibilities. For all three, all variables will be bound by the time they are processed. Of these three possibilities the conjunct $s \neq$ Rob with s bound has much higher utility (because its probability of success is much greater) than either Parent(Rob, p) with p bound or Parent(s, p) with both s and p bound. We therefore choose to expand the candidate $s \neq$ Rob. The two possibilities generated are

Parent(Rob, p) $\wedge$ $s \neq$ Rob with s and p bound

and

Parent(s, p) $\wedge$ $s \neq$ Rob with p bound.

The latter has much higher utility since there is greater chance of satisfying the Parent relation with only one argument bound than with both arguments

bound. The utility of this candidate is also greater than that for either of the two remaining singleton candidates generated in the first step. As a result, we immediately generate the candidate

$$\text{Parent}(\text{Rob}, p) \wedge \text{Parent}(s, p) \wedge s \neq \text{Rob}\,.$$

The utility of this candidate is still higher than that for the remaining candidates, so the search is terminated.

It is fairly straightforward to integrate the conjunct ordering process into the algorithms for computing worth information given in Section 2.4.3. When a conjunctive premise is encountered for a rule, the techniques discussed here are applied to find the best ordering for the conjunction, for each different way of binding the variables in the conclusion of the rule. These orderings are then stored with the rule. The expected cost and probability of success for the conjunction is then used to compute the worth of the conjunction, and hence of the rule.

5. Discussion

5.1. Possible extensions

There are a number of different ways in which the analysis given in this paper needs to be extended. Many of these correspond to the assumptions mentioned in Section 1.4. In this section, we offer some preliminary ideas on how the results given in this paper might be extended to cover these possibilities.

5.1.1. *Handling recursion*

In our calculation of probability of success and expected cost for strategies, we did not consider the possibility of recursive inference. In the case where we are only seeking a single answer, most recursion can simply be eliminated, as shown in [14]. For this case, we can therefore treat recursive inference steps as having zero worth.

However, if more than one answer is sought, the matter becomes much more difficult. In general, the recursive space will produce new answers to some depth of recursion, and the remainder of the space below that will be redundant. In [14], we develop a run-time method for deciding when to halt recursive inference. Briefly, it involves suspending repeating subgoals, and caching the answers to any goals with repeating subgoals. When new solutions are cached, they must also be tried in all suspended repeating subgoals. This method takes advantage of the fact that the space looks identical at each level of recursion.

The trouble is that we have no accurate way of estimating the number of answers that can be produced by a recursive space, and, therefore, have no

way of computing P_n for recursive steps. One possibility is to try to estimate P_n by computing upper and lower bounds. We can get a lower bound on this number by assuming that all answers produced below level one will be redundant. To do this, we could compute P_n for a recursive step assuming that the same rule cannot be used again in finding the solutions to any of the subgoals generated by the step. In other words, we would only consider other ways of satisfying the premises of the rule.

It might also be possible to compute an upper bound on P_n in some cases. We could iterate the calculation suggested above, using our lower bound calculation for P_n for the recursive premises of the rule. If this iteration converges, we should have an upper bound on P_n.

It remains to be seen how accurate such estimates would be and what effect the inaccuracy would have on strategy selection. If we were to use lower bounds only, the effect would be to postpone recursive inference until later in the strategy. This might be justifiable because of the extra overhead involved in the run-time control of recursive inference.

5.1.2. *Caching and redundancy*

Throughout this paper we have assumed that inference actions in separate branches of the search space have independent cost and probability of success. In other words, the performance of one such inference action does not affect the probability of success or the expected cost of any other inference action that is not a descendant of it. There are at least two cases where inference actions can violate the independence assumption: when facts are cached, and when there is a redundancy in the search space.

First consider redundancy. Let B be a portion of a search space that is known to be redundant with another portion A. Suppose that a is a complete strategy for the portion A, and b is a complete strategy for the portion B. For the strategy ab, the likelihood of success for the second step b is reduced to zero since B is redundant with A. However, for the strategy ba, the likelihood of success for both strategies is unaffected by the redundancy. The matter is more complicated for strategies that interleave searching of the two portions. In this case, any actions from B that follow all actions from A will have their probability of success reduced to zero.

Accurately dealing with redundancy seems to require an iterative approach to calculating worth. As before, we could start from database facts and work upward. If a block is constructed that has successors that are redundant with it (or have parts that are redundant with it) we could change the worth of the redundant portion to zero for the purposes of all subsequent worth calculations. We would then have to recompute the worth of any blocks containing the redundant portions for the purposes of all subsequent worth calculations. In addition, the same incremental worth update would have to be done at run-time whenever steps were executed that had redundant cousins.

A less expensive (heuristic) approach to dealing with redundancy is to ignore it for purposes of calculating worth, but remove redundant steps at run-time once their subsuming steps have been processed. Although this strategy is not always optimal, it is at least as good as the optimal strategy if the redundancy had not been recognized. This is because the run-time recognition and elimination of redundant steps can only improve the performance of a strategy.

The second situation where the independence assumption is violated is when caching of intermediate solutions is used, and similar subgoals appear in different portions of the inference space. If answers to a subgoal g are cached, and another similar subgoal g' appears elsewhere in the search space, the overall cost of finding answers to g' has been reduced. More precisely, the probability of finding answers to g' in the database has been improved, and some of the inference steps for finding answers to g' become redundant. It is relatively easy to construct examples where the addition of caching can drastically change the optimal strategy.

To deal with caching accurately, we would have to adopt an iterative approach like that suggested above for reduncancy; whenever a block is created that has successors containing an identical subgoal, the worth for the identical subgoal would have to be updated to reflect the cached information. As with the treatment of redundancy, run-time updating of the worth network must also be done. Whenever a derived fact is cached, the worth network must be incrementally updated to reflect the addition of the new fact.

As with the treatment of redundancy, a simpler approach is to ignore caching in computing the initial worth network, but still update the network incrementally as caching takes place. The initial strategy may not be optimal, but will be at least as good as the optimal strategy if no caching were used.

5.1.3. *Forward inference*

Although this paper concentrates on the evaluation and selection of backward inference steps, there is no fundamental reason why similar evaluations could not be made for forward (data-directed) inference steps. For a forward inference step, instead of computing the probability that a goal expression would match a fact in the database, or the right hand side of a rule, we could compute the probability that a fact would match a premise of a particular rule or one of the goal expressions. Likewise, the expected cost would be the expected cost of using a particular forward inference path to find a solution to a goal. The "worth" of a forward inference step would therefore be an evaluation of how useful the forward inference step is likely to be. As with step selection in backward inference, it should be possible to incrementally update the worth information for forward inference steps as new goals get added to the system.

The application of this technique to both forward and backward inference opens up an intriguing possibility; it should be possible to maintain worth

information for both forward and backward inference steps and to interleave forward and backward inference by choosing the step with the greatest worth. In doing this, whenever a new goal or subgoal is generated, the worth information for forward inference steps would have to be incrementally updated. Likewise, whenever a new fact was added to the database, or is derived by forward inference, the worth information for backward inference steps would have to be incrementally updated.

This method for interleaving forward and backward inference is very dynamic, but also may be computationally expensive because of the amount of incremental updating of the worth networks that would take place. It would be interesting to compare this possibility with the weaker, but computationally less expensive compile-time methods developed by Treitel [16–18].

5.2. Related work

The idea of using elementary decision theory in the selection of problem solving strategies appears to have its origin in the works of Simon and Kadane [11], and Sproull [2, 15]. In [11], Simon and Kadane prove theorems similar to those in Section 2.4, but for a simpler case where the local probability of problem solving steps is always one. They suggest the use of this result for control of general search problems, but give no indication of how to estimate the probability of success for individual steps. Much of Simon and Kadane's analysis is based on earlier work appearing in the operations research literature. In particular, many of their results and the results appearing in Section 2.5 are similar to results of Garey [3] for job shop scheduling problems. Recently, Barnett [1] has investigated the efficacy of this sort of analysis on some simple examples and has considered the sensitivity of the analysis of Simon and Kadane to errors in the estimation of expected cost and probability. It is not yet clear whether this analysis will apply to the work described here.

In [2, 15], Sproull makes use of decision theory to help find optimal plans in a travel planning system. He assumes that expected cost and probability estimates are available, a priori, for each action and goal considered by the planner. He then uses this information to determine the probability of success and expected cost for different possible plans. As with the work of Simon and Kadane, Sproull does not suggest any way of automatically assessing the expected cost and probability of success for achieving goals. Sproull also does not provide any means of ordering conjunctive goals, or of assessing the expected cost or probability of achieving conjunctive goals. In [12] we have extended the work of Sproull to allow evaluation of the expected cost and probability of achieving conjunctive goals, and have provided an $n \log n$ algorithm for choosing the order in which to work on conjunctive goals.

The work closest in spirit to that described in this paper is recent work by Natarajan. In [6, 8] Natarajan uses decision theoretic analysis to help reorder

rules for PROLOG programs. Because of the concentration on PROLOG programs, Natarajan limits the possible search strategies being considered to those that can be achieved with a depth-first search procedure (rather than a best-first search procedure). Many of his theorems are similar to those found in Section 2, but the proofs are simpler because of the restriction to depth-first search.

Another difference between Natarajan's work and our own is that, rather than computing probability and expected cost information from primitive data about the distribution of atomic facts in the database, Natarajan gathers this data empirically while the problem solver is running. The advantage of this approach is that the resulting control strategies are not bound by the uniformity assumption and can, therefore, be more specific and more accurate for the mix of problems that will actually occur. The disadvantage is that such a system cannot provide search guidance for problems of a form not yet encountered. What this suggests is that the approach described in this paper and an empirical or learning approach are complementary. The approach described in this paper can provide initial control information for types of problems not seen before. A learning approach could then be used to further refine that control information.

In the work described above, Natarajan also considers the problem of ordering conjunctions. However, he only considers the case where all conjuncts are independent of each other. This leads to a particularly simple result that such conjuncts can be ordered by decreasing utility. This result could be used with the best-first search procedure from Section 4.3 to speed up the search for the best ordering when the conjuncts remaining do not share any variables.

In other work, [7, 9], Natarajan considers the problem of finding the best ordering for conjunctions when all answers to the problem are sought. In particular, he proves a more general version of the *adjacency restriction* found in [13]. This result allows a reduction in the number of conjunct orderings that need to be considered in the search for the optimal ordering. Although this theorem only applies to the case where all solutions are sought for a conjunction, a variant of it may turn out to be applicable in the single answer case.

Other recent work using decision theoretic techniques for inference control has been done by Treitel [16–18]. Treitel has used this kind of analysis to evaluate, on average, whether it is better to use each particular rule in the forward or backward direction. For a class of strategies called *coherent strategies*, Treitel has shown that these evaluations can be performed efficiently using integer programming techniques.

There are several other approaches to controlling inference that are related, but different than the approach taken in this paper. One such approach is that of connection graph theorem proving. Connection graph theorem provers maintain explicit links between rules that can be resolved with each other, and between rules and atomic facts that will resolve with them. A given inference

step is only attempted if there is an inference path beginning with that step, and leading to a collection of facts in the database. This approach can be seen as a special case of the decision theoretic approach taken here; the cost of the inference path is effectively ignored, and no distinction is made between different positive probabilities, only between zero and nonzero probability. A similar technique is used by Klahr [4] in a backward production system.

A second approach (sometimes used in the database community) is to flatten the entire search space, and order the resulting rules. If one were to use the techniques discussed in Section 2 for ordering the resulting rules, the strategy chosen by this approach would, in fact, be identical to the strategy found by following worth information. The difference is primarily in the amount of storage required. In the case of flattening, for every possible goal expression we would have to store a potentially complex rule for every path from the goal to a terminal node in that search space. For example, suppose we have a collection r of rules, and for simplicity assume that each rule only contains a single clause in its premise. Suppose that we can divide these rules up into sets $r_1, r_2, \ldots, r_n$ so that the relations in the premises of the rules in r_i are those found in the consequences of the rules in r_{i-1}. Suppose that there are k_i rules in the set r_i and suppose that, on average there are d_i rules in each set r_i for each conclusion relation represented in r_i. If we were to flatten this rule set, we would need to have k_1 rules for the conclusion relations from r_1, $k_2 d_1$ rules for the relations in r_2, $k_3 d_2 d_1$ rules for the conclusions in r_3, and so on. Thus, the total number of rules required is

$$k_1 + k_2 d_1 + k_3 d_2 d_1 + \cdots + k_n d_{n-1} \cdots d_1 .$$

The number of rules would be even larger if the rules had more than one conjunct. As a result, the amount of storage required becomes intractable for any large rule set where the d_i are greater than one, or the number of conjuncts in the premises of rules is generally greater than one. In contrast, the original number of rules is only $k_1 + \cdots + k_n$, and the approach described in this paper only requires a constant amount of storage for each rule.

5.3. Final remarks

In this paper, we have presented a domain-independent means of controlling backward inference that is sensitive to global information about the facts and rules available in a system's database. It involves: (1) compile-time computation of worth information for all inference steps sanctioned by the rules in a system, (2) keeping this information stored with the rule corresponding to each inference step, and (3) having the inference procedure choose the step with the greatest worth at each stage of the problem solving process.

The computation of worth information is based on an assumption that facts

and queries for each relation will be uniformly distributed over the domain of the relation. Simple decision theory is used to evaluate the probability and expected cost for different strategies. The theorems developed in Section 2.4 then allow efficient computation of the worth of each inference step. As discussed in Section 2.4.3 the worth information for a collection of rules can be computed in time that is effectively linear in the number of rules. This worth information can be updated in an incremental fashion as the database changes.

The approach described in this paper is in some sense half way in between a domain-independent control strategy and a domain-dependent strategy. The equations and mechanism for computing worth information are domain-independent, but the actual assessments, the worth numbers, are domain-dependent, since the calculations make use of information about the number of each type of atomic fact in the database.

Despite the mathematical nature of many of the calculations done in this paper, the general approach has a great deal of intuitive appeal. When solving problems, people seem to have a reasonably good idea of the merit of each different approach to solving a problem. This is what worth information is all about: an evaluation of the merit of each approach.

Appendix A. Proofs of Theorems

A.1. Proof of the best strategy theorem

The proof of the best strategy theorem requires a number of lemmas, with rather complex proofs:

Lemma A.1.

$$\min\{U(x), U(y)\} \leq U(xy) \leq \max\{U(x), U(y)\} .$$

Furthermore, if $U(x) \neq U(y)$ *the inequalities are strict.*

Proof.

$$\begin{aligned} U(xy) &= \frac{P(xy)}{E(xy)} = \frac{P(x) + P_0(x, y)P(y)}{E(xy)} \\ &= \frac{E(x)U(x) + P_0(x, y)E(y)U(y)}{E(xy)} \\ &\leq \frac{E(x) + P_0(x, y)E(y)}{E(xy)} \max\{U(x), U(y)\} = \max\{U(x), U(y)\} . \end{aligned}$$

The proofs for the other inequalities are similar. □

Lemma A.2. *If b and c are independent strategies, then*

$$E(abcd) < E(acbd) \leftrightarrow U(abcd) > U(acbd) \leftrightarrow U(b) > U(c) .$$

Proof. Since $P(abcd) = P(acbd)$ we know that

$$U(abcd) > U(acbd) \leftrightarrow E(abcd) < E(acbd) ,$$

so it is only necessary to show that

$$E(abcd) < E(acbd) \leftrightarrow U(b) > U(c) .$$

We have

$$\begin{aligned} &E(acbd) - E(abcd) \\ &\quad = (E(acb) + P_0(acb, d)E(d)) - (E(abc) + P_0(abc, d)E(d)) \\ &\quad = E(acb) - E(abc) \\ &\quad = (E(a) + P_0(a, c)E(c) + P_0(ac, b)E(b)) \\ &\qquad - (E(a) + P_0(a, b)E(b) + P_0(ab, c)E(c)) \\ &\quad = E(c)(P_0(a, c) - P_0(ab, c)) - E(b)(P_0(a, b) - P_0(ac, b)) . \end{aligned}$$

Now consider the terms $P_0(ab, c)$ and $P_0(ac, b)$.

$$\begin{aligned} P_0(ab, c) &= L((ab)_c)P_0((ab)_{\bar{c}}) = L(a_c)P_0(a_{\bar{c}}b) \\ &= L(a_c)(P_0(a_c) - P_0(a_{\bar{c}}, b)P(b)) \\ &= P_0(a, c) - L(a_c)P_0(a_{\bar{c}}, b)P(b) \\ &= P_0(a, c) - L(a_c)L((a_{\bar{c}})b)P_0((a_{\bar{c}})_{\bar{b}})P(b) \\ &= P_0(a, c) - L(a_{bc})P_0(a_{\overline{bc}})P(b) \\ &= P_0(a, c) - P_0(a, bc)P(b) . \end{aligned}$$

Likewise,

$$P_0(ac, b) = P_0(a, b) - P_0(a, bc)P(c) .$$

So,

$$\begin{aligned} E(acbd) - E(abcd) &= E(c)P_0(a, bc)P(b) - E(b)P_0(a, bc)P(c) \\ &= P_0(a, bc)(E(c)P(b) - E(b)P(c)) \\ &= P_0(a, bc)E(b)E(c)(U(b) - U(c)) . \end{aligned}$$

Thus,

$$E(abcd) < E(acbd) \leftrightarrow U(abcd) > U(acbd) \leftrightarrow U(b) > U(c) . \qquad \square$$

Lemma A.3. *If $s = ab$ is an indivisible block, then $U(b) \geq U(s) \geq U(a)$.*

Proof. First note that any indivisible block s will be composed of a sequence of smaller indivisible blocks $s_0 s_1 s_2 \ldots s_n$ such that s_0 is the initial step of the block, and none of the other s_i are part of any larger blocks in s. For convenience let $s_{i,j}$ refer to the subsequence $s_i \ldots s_j$. By the definition of an indivisible block, the block s must have been formed by first adding s_1 onto s_0, then s_2 onto $s_{0,1}$, etc. Thus each $s_{0,i}$ is also an indivisible block and $U(s_{i+1}) \geq U(s_{0,i})$. This combined with Lemma A.1 gives:

$$U(s_{i+1}) \geq U(s_{0,i+1}) \geq U(s_{0,i}) .$$

By transitivity on the last half of this inequality we get:

$$U(s_{0,n}) \geq U(s_{0,i}) .$$

Again applying Lemma A.1 we get:

$$U(s_{i+1,n}) \geq U(s_{0,n}) \geq U(s_{0,i}) .$$

Thus, for the simple case where $b = s_{i+1,n}$ the result is proved.

The general proof is by induction on the depth of the strategy s. We define the depth of a strategy as the number of levels in the search space contained in the strategy. For the case where s is of depth 1, each s_i will be an individual step. The strategy b must therefore be a subsequence $s_{i,n}$ of s. The previous argument holds and the theorem follows for this case.

Now assume that the theorem holds for all strategies s' of depth less than k, and suppose that s is of depth k. In general b is some arbitrary tail of s. Suppose that b begins with some step in s_i. Let r and t be defined such that $a = s_{0,i-1} r$, $b = t s_{i+1,n}$, and $s_i = rt$. Note that the depth of s_i is necessarily less than that of s. By our induction hypothesis, $U(t) \geq U(s_i) \geq U(r)$. By our previous argument we also know that $U(s_i) \geq U(s_{0,i-1})$. Putting these two together we get $U(t) \geq U(s_{0,i-1})$. Since the utility of t is greater than that of both $s_{0,i-1}$ and r, by Lemma A.1 it is greater than the utility of their concatenation.

$$U(t) \geq U(s_{0,i-1} r) = U(a) .$$

Again applying Lemma A.1 we get

$$U(t) \geq U(s_{0,i}) \geq U(a) .$$

By our previous argument we also know that $U(s_{i+1,n}) \geq U(s) \geq U(s_{0,i})$. Now, since both t and $s_{i+1,n}$ have utilities greater than that of $s_{0,i}$ by Lemma A.1 their concatenation will have greater utility:

$$U(b) = U(ts_{i+1,n}) \geq U(s_{0,i}) \geq U(a) .$$

Using Lemma A.1 once more on this last equation we get

$$U(b) \geq U(s) \geq U(a) .$$

By induction the theorem holds for strategies of arbitrary depth. □

Lemma A.4. *If $s = ab$ is a best strategy, then $U(b) \geq U(s) \geq U(a)$.*

Proof. Suppose that $U(a) > U(b)$. By Lemma A.1, we known that

$$U(s) < \max\{U(a), U(b)\} = U(a) .$$

But this means that a alone would be a better strategy than ab, violating our assumption. Therefore $U(b) \geq U(a)$. Again using Lemma A.1 we know that

$$\max\{U(a), U(b)\} \geq U(s) \geq \min\{U(a), U(b)\} .$$

But

$$\max\{U(a), U(b)\} = U(b) \quad \text{and} \quad \min\{U(a), U(b)\} = U(a) ,$$

so $U(b) \geq U(s) \geq U(a)$. □

Lemma A.5. *Every step s_i in a best strategy s must be followed by the remaining steps of a maximal indivisible block for s_i.*

Proof (by induction). We first prove the assertion for the final step in the best strategy, then assume it for all steps following the ith step and prove it for the ith step.

Let s_n be the final step in the best strategy. Suppose that every maximal indivisible block, s_n^*, for s_n contains additional steps, i.e. $s_n^* = s_n t$. Since the steps in t depend on s_n they cannot precede s_n and therefore do not belong to s. By Lemma A.3, $U(t) \geq U(s_n)$. Furthermore, since we are assuming that s_n alone is not a maximal indivisible block, the inequality must be strict $U(t) > U(s_n)$. But by Lemma A.4 we also know that $U(s_n) \geq U(s)$. So $U(t) > U(s)$. By Lemma A.1 this implies that $U(st) > U(s)$. But this violates our assumption that s is the best strategy. s_n^* must therefore be the single step s_n.

Now assume that the lemma is true for all steps beyond s_i and suppose that s_i is not followed by all of the steps from any of its maximal indivisible blocks, s_i^*. We divide s up into three segments $s = abc$ such that

- a is the initial block preceding s_i,
- b is the largest indivisible block beginning with s_i and contained in s,
- c is the remainder of s.

We further divide c up into pieces $c_0, c_1, \ldots, c_k$ such that all of the even pieces $c_0, c_2, c_4, \ldots$ consist only of steps that are dependent on steps in b, and the odd pieces $c_1, c_3, c_5, \ldots$ are independent of b (c_0 may be empty). First note that the even and odd pieces are independent of each other. By Lemma A.2 the pieces must therefore be arranged in order of decreasing utility,

$$U(c_0) \geq U(c_1) \geq \cdots \geq U(c_k) .$$

Let $t = t_1 \ldots t_m$ refer to the remaining steps in some maximal indivisible block s_i^* containing the segment b. Thus $s_i^* = bt$. We now show that $U(t_1^*) > U(c_j)$. There are two cases to consider:

Case 1. Suppose that c_0 is not empty. By the definition of a maximal indivisible block $U(t_1^*)$ must be greater than or equal to that of all other descendants of b. As a result, $U(t_1^*) \geq U(c_0)$. But if the utilities were equal bc_0 would form an indivisible block. This violates our assumption that b was the largest indivisible block beginning with s_i contained in s. Therefore, in this case

$$U(t_1^*) > U(c_0) \geq U(c_j) .$$

Case 2. Suppose that c_0 is empty. Now suppose that $U(c_1) > U(b)$. According to Lemma A.2 the strategy could be improved by interchanging b and c_1, violating our assumption that s is a best strategy. Consequently, $U(c_1) \leq U(b)$. By Lemma A.3 we know that $U(t) \geq U(b)$. In fact, since we are assuming that b alone is not a maximal indivisible block the inequality is strict, $U(t) > U(b)$. By the definition of a maximal indivisible block $U(t_1^*)$ is greater than or equal to that of all other descendants of bt_1^*. By Lemma A.1, we get that $U(t_1^*) \geq U(t)$. As a result, we get $U(t_1^*) > U(b)$. This fact, together with the fact that $U(c_1) \leq U(b)$ gives

$$U(t_1^*) > U(c_1) \geq U(c_j) .$$

Using the fact that $U(t_1^*) > U(c_j)$ we can now perform the final step in the proof. Again there are two cases to consider.

Case 1. Suppose that t_1 is not contained in s. Then t_1^* is not contained in s. By Lemma A.4 we know that $U(c) \geq U(s)$ and therefore that $U(t_1^*) > U(s)$. By

Lemma A.1 $U(st_1^*) > U(s)$, which violates our assumption that s is a best strategy.

Case 2. Suppose that t_1 is in s. By our induction hypothesis t_1 is followed by the remaining steps in t_1^*. Suppose t_1^* is contained in c_i, i.e. $c_i = dt_1^* e$. Since $U(t_1^*)$ is greater than or equal to that of all other descendants of bt_1^* we know that $U(t_1^*) \geq U(d)$. If $U(t_1^*) > U(d)$ the strategy could be improved by interchanging d and t_1^*. Therefore $U(d) = U(t_1^*)$, which implies that $U(dt_1^*) = U(t_1^*)$. But by our earlier argument we know that $U(t_1^*) > U(c_i)$, for every i. As a consequence, the strategy can be improved by interchanging c_{j-1} and dt_1^*, violating our assumption that s is a best strategy.

As a result, s_i must be followed by the remaining steps in one of its maximal indivisible blocks. By induction on i the lemma holds. □

Using these lemmas, we can now prove the best strategy theorem.

Theorem 2.9 (Best strategy). *A best strategy for a step is a maximal indivisible block beginning with that step.*

Proof. By Lemma A.5, we know that the best strategy for a step a begins with a maximal indivisible block, a^*. Suppose that the strategy contains additional steps t. Note that t must consist of a sequence of maximal indivisible blocks that are descendants of the block a^*. By the definition of a maximal indivisible block we know that all descendants of the block must have utility lower than that of the block itself. Therefore $U(a^*) \geq U(t)$. But by Lemma A.4 we know that $U(t) \geq U(a^*)$. Together these imply that $U(a^*) = U(t)$, which means that $a^* t$ is also a maximal indivisible block for a. □

A.2. Proof of the optimality theorem

In order to prove Theorem 2.11 another lemma is required. The lemma and proof are quite similar to that of Lemma A.5 except that we are dealing with complete strategies rather than best strategies.

Lemma A.6. *Every step s_i in an optimal complete strategy s must be followed by the remaining steps of a maximal indivisible block for s_i.*

Proof (by induction). Let s_n be the final step in the strategy. Since s is a complete strategy, s_n cannot have any successors. Therefore s_n is a maximal indivisible block.

Now we assume that the lemma is true for all steps beyond s_i, and prove that it holds for s_i. Suppose that s_i is not followed by all of the steps from any of its

maximal indivisible blocks, s_i^*. We divide s up into three segments $s = abc$ such that

- a is the initial block preceding s_i,
- b is the largest indivisible block beginning with s_i and contained in s,
- c is the remainder of s.

We further divide c up into pieces $c_0, c_1, \ldots, c_k$ such that all of the even pieces $c_0, c_2, c_4, \ldots$ consist only of steps that are dependent on steps in b, and the odd pieces $c_1, c_3, c_5, \ldots$ are independent of b (c_0 may be empty). First note that the even and odd pieces are independent of each other. By Lemma A.2 the pieces must therefore be arranged in order of decreasing utility,

$$U(c_0) \geq U(c_1) \geq \cdots \geq U(c_k) .$$

Let $t = t_1 \ldots t_m$ refer to the remaining steps in some maximal indivisible block s_i^* containing the segment b. Thus $s_i^* = bt$. We now show that $U(t_1^*) > U(c_j)$. There are two cases to consider:

Case 1. Suppose that c_0 is not empty. By the definition of a maximal indivisible block $U(t_1^*)$ must be greater than or equal to that of all other descendants of b. As a result, $U(t_1^*) \geq U(c_0)$. But if the utilities were equal bc_0 would form an indivisible block. This violates our assumption that b was the largest indivisible block beginning with s_i contained in s. Therefore, in this case

$$U(t_1^*) > U(c_0) \geq U(c_j) .$$

Case 2. Suppose that c_0 is empty. Now suppose that $U(c_1) > U(b)$. According to Lemma A.2 the strategy could be improved by interchanging b and c_1, violating our assumption that s is a best strategy. Consequently, $U(c_1) \leq U(b)$. By Lemma A.3 we know that $U(t) \geq U(b)$. In fact, since we are assuming that b alone is not a maximal indivisible block the inequality is strict, $U(t) > U(b)$. By the definition of a maximal indivisible block, $U(t_1^*)$ is greater than or equal to that of all other descendants of bt_1^*. By Lemma A.1, we get that $U(t_1^*) \geq U(t)$. As a result, we get $U(t_1^*) > U(b)$. This fact, together with the fact that $U(c_1) \leq U(b)$ gives

$$U(t_1^*) > U(c_1) \geq U(c_j) .$$

We can now perform the final step in the proof. Since s is a complete strategy we know that t_1 is in s. By our induction hypothesis t_1 is followed by the remaining steps in t_1^*. Suppose t_1^* is contained in c_j, i.e. $c_j = dt_1^*e$. Since $U(t_1^*)$ is greater than or equal to that of all other descendants of bt_1^*, we know that $U(t_1^*) \geq U(d)$. If $U(t_1^*) > U(d)$ the strategy could be improved by inter-

changing d and t_1^*. Therefore $U(d) = U(t_1^*)$, which implies that $U(dt_1^*) = U(t_1^*)$. But by our earlier argument we know that $U(t_1^*) > U(c_i)$, for every i. As a consequence, the strategy can be improved by interchanging c_{i-1} and dt_1^*, violating our assumption that s is an optimal strategy.

As a result, s_i must be followed by the remaining steps in one of its maximal indivisible blocks. By induction on i the lemma holds. □

The proof of the optimality theorem is now straightforward.

Theorem 2.11 (Optimality). *The complete strategy with the lowest expected cost for solving a problem consists of the steps in the space ordered by decreasing worth (respecting the allowed partial ordering on steps).*

Proof (by contradiction). Suppose that the optimal strategy is *abcd* where b and c are independent steps and that $W(c) > W(b)$. Since *abcd* is assumed to be optimal, by Lemma A.6 the steps in a maximal indivisible block c^* for c must follow c in s. By our assumption

$$W(c) = U(c^*) > W(b) \geq U(b) .$$

Therefore, by Lemma A.2 the strategy *abcd* can be improved by interchanging b and c^*. Therefore *abcd* is not optimal, violating our assumption. □

Appendix B. A Program for Worth Calculation

As discussed in Section 2.4.3 the basic process involves starting with those relations that do not appear in the conclusion of any rule and working upwards through the DAG of rules, building indivisible blocks and storing worth information with each rule.

The code given in Fig. 17 is divided up into several different procedures for clarity. The top level procedure, ProcessDB, merely loops through all relations represented in the database and calls ProcessRel on each one. This guarantees that no portion of the DAG is omitted in case it consists of several unconnected pieces. The procedure ProcessRel is responsible for calculating the worth of a database lookup of the given relation, and for calculating the worth of all rules containing that relation in their consequent. In order to calculate the worth of a rule ProcessRel must first make sure that the worth is already calculated for all successors of the rule. As a result it first calls itself recursively on the premise relation of the rule before calling ProcessRule to do the actual worth calculation for the rule. The procedure ProcessRule does the actual work of building the maximal indivisible block for a given rule. (When it is called, the worth will already be computed for all of the rules successors in the DAG.) Note that it doesn't actually store the indivisible block, but just keeps a list of

```
ProcessDB:                                              ; compute worth for a DB
    For each p : Indb(p(x⃗)) do: ProcessRel(p)

ProcessRel(q):                                          ; compute worth for predicate q
    If P(q,0) then Return                               ; if already done then return
    For each b ∈ {0,...,2^Arity(q) − 1} do:             ; compute DB probabilities
        P(q,b) ← DBProbability(q,b)
    For each r,p : Indb(r) ∧ r = "p(y⃗) → q(x⃗)" do:      ; take care of successors
        ProcessRel(p)
        ProcessRule(r,p,q)

ProcessRule(r,p,q):                                     ; compute worth for a rule
    For each b ∈ {0,...,2^Arity(q) − 1} do:
        b' ← Regress(r,b)                               ; find the bindings in the premise
        L ← LocalProb(r,b)                              ; initialize the block
        P ← P(p,b')
        BP(r,b) ← L * P
        BE(r,b) ← If P = 0 then 0 else I + L * D
        W(r,b) ← BP(r,b) / BE(r,b)
        BSuccessors(r,b) ← {r',b' : Indb(r') ∧ r' = "φ → p(x⃗)"}
        For each r' ∈ BSuccessors(r,b) do:
            BP0(r,b,r') ← L * (1 − P)
        While BSuccessors(r,b) do:                      ; build indivisible blocks
            r',b' ← r',b' ∈ BSuccessors(r,b) : Max W(r',b')
            If W(r',b') < W(r,b) then Exit
            Combine(r,b,r',b')

Combine(r,b,r',b'):                                     ; combine two blocks
    BSuccessors(r,b) ← (BSuccessors(r,b) − r',b') ∪ BSuccessors(r',b')
    BP(r,b) ← BP(r,b) + BP0(r,b,r') * BP(r',b')         ; compute probability
    BE(r,b) ← BE(r,b) + BP0(r,b,r') * BE(r',b')         ; compute expected cost
    W(r,b) ← BP(r,b) / BE(r,b)                          ; compute worth
    For each z ∈ BSuccessors(r',b') do:                 ; compute P0 for new successors
        BP0(r,b,z) ← BP0(r,b,r') * BP0(r',b',z)
    Discard BP0(r,b,r')
```

Fig. 17. A program for worth calculation.

the current block's successors. The procedure Combine is called to do the actual dirty work of putting two blocks together once it is determined that they form an indivisible block.

There is one additional subtlety to the ProcessRule and Combine procedures. In order to do the expected cost and probability calculations when combining two blocks s and t we need to know $P_0(s, t)$. In the vast majority of cases the rules involved will have only variables in their conclusions. Thus $L(s_t)$ will be one, which implies that $P_0(s_t) = P_0(s)$ and therefore $P_0(s, t) = P_0(s) = 1 - P(s)$,

which is a computable quantity. However, when $L(s_t)$ is not one, $P_0(s, t)$ is impossible to compute without searching through the block s. This search would be expensive, so we store $P_0(s, t)$ for those successors of a block where $L(s_t) \neq 1$. When blocks are combined this information can be updated using the equation

$$P_0(st, z) = P_0(s, t)P_0(t, z) .$$

In the procedures that follow, the bound arguments for a predicate p are indicated by a number between 0 and $2^{\mathsf{Arity}(p)} - 1$. A 1 in the corresponding bit vector indicates that the argument position is bound, and a zero indicates that it is not. Thus p^0 indicates the relation p with all arguments free, whereas p^3 indicates p with its final two arguments bound.

The function $P(p, b)$ is used to refer to the probability that a database lookup will succeed for the predicate p and set of bindings b. $\mathsf{BP}(r, b)$ and $\mathsf{BE}(r, b)$ refer to probability of success and expected cost for the indivisible block associated with a rule r and set of bindings b. $\mathsf{BSuccessors}(r, b)$ is the list of successors for the block associated with the rule r and bindings b.

Appendix C. Computing $P_{\geq n}$ and E_n

C.1. Computing $P_{\geq n}$ for database lookup steps

As in the development of Section 2.2 let $R(\boldsymbol{B})$ be the goal expression of interest, and let

$m \stackrel{\text{def}}{=}$ the number of facts in the database having the relation R,

$g \stackrel{\text{def}}{=}$ the number of different possible bindings $\boldsymbol{b}$ that have constants in the same positions as $\boldsymbol{B}$.

$h \stackrel{\text{def}}{=}$ the average number of solutions (in theory) for goals of the form $R(\boldsymbol{b})$ over all possible bindings $\boldsymbol{b}$ that have constants in the same positions as $\boldsymbol{B}$.

Theorem C.1. *Let $D_{R(\boldsymbol{B})}$ refer to the step of finding solutions in the database for a query of the form $R(\boldsymbol{B})$. The probability that this step will find n or more solutions $(n \leq m)$ is*

$$P_{\geq n}(D_{R(\boldsymbol{B})}) = 1 - \sum_{i=0}^{n-1} \binom{h}{i} \binom{(g-1)h}{m-i} \Big/ \binom{gh}{m} . \tag{C.1}$$

Here we assume that the set of bindings $\boldsymbol{B}$ lie within the appropriate domains for the relation R, i.e. $B_i \in \mathsf{Domain}(R, i)$. This assumption is often true for

subgoals generated internally. When this is not true, the total probability must be multiplied by the probability that variable bindings lie within the appropriate domains, as suggested in [13].

Proof. The probability that there are n or more facts in the database matching a given goal expression is one minus the probability that there are fewer than n such propositions in the database.

$$P_{\geq n}(D_{R(\boldsymbol{B})}) = 1 - P_{<n}(D_{R(\boldsymbol{B})}) = 1 - \sum_{i=0}^{n-1} P_i(D_{R(\boldsymbol{B})}) .$$

Assuming that the query has n answers, there are $\binom{h}{n}$ different ways in which those n answers could be distributed over the h potential solutions to the query.[10] Likewise, there are $\binom{(g-1)h}{m-n}$ different ways of placing the remaining $m-n$ facts containing the relation R over the remaining $g-1$ possible queries of the same form. There are $\binom{gh}{m}$ different ways in which all m of the facts containing the relation R could be distributed over all theoretically possible R-tuples. Thus the probability that there are exactly n answers for a particular query of the form $R(\boldsymbol{B})$ is

$$P_n(D_{R(\boldsymbol{B})}) = \binom{h}{n}\binom{(g-1)h}{m-n} \Big/ \binom{gh}{m} .$$

The probability that there are at least n solutions (for $n \leq m$) is therefore

$$P_{\geq n}(D_{R(\boldsymbol{B})}) = 1 - \sum_{i=0}^{n-1} \binom{h}{i}\binom{(g-1)h}{m-i} \Big/ \binom{gh}{m} . \qquad \square$$

For $n = 1$ (i.e. only one answer is needed), this expression reduces to the equation given in Theorem 2.3:

$$P_{\geq 1}(D_{R(\boldsymbol{B})}) = 1 - \binom{(g-1)h}{m} \Big/ \binom{gh}{m} .$$

There are several other useful special cases of this theorem.

Corollary C.2. *If $m = 1$ (i.e. only one fact in the database has the desired relation),*

[10] It may be helpful to think about this in terms of egg cartons and marbles. Given g egg cartons, each of which has h holes in it, and m marbles distributed randomly over all g times h holes (no hole can hold more than one marble), what is the probability that a given egg carton will contain exactly n marbles?

$$P_{\geqslant n}(D_{R(B)}) = \begin{cases} 1/g\,, & \text{if } n = 1\,, \\ 0\,, & \text{otherwise}\,. \end{cases}$$

Corollary C.3. *If* $h = 1$ (*i.e. the query is a ground clause or functional expression*),

$$P_{\geqslant n}(D_{R(B)}) = \begin{cases} m/g\,, & \text{if } n = 1\,, \\ 0\,, & \text{otherwise}\,. \end{cases}$$

Corollary C.4. *If* $g = 1$ (*i.e. none of the variables in R are bound*),

$$P_{\geqslant n}(D_{R(B)}) = \begin{cases} 1\,, & \text{if } n \leqslant m\,, \\ 0\,, & \text{otherwise}\,. \end{cases}$$

Example. In the kinship example, the database contains three fatherhood facts ($m = 3$), so there is some chance that certain fatherhood queries will produce more than one answer. In the case of $D_{F_{\mathrm{bb}}}$ all variables are bound, so $h = 1$ and there can be only one answer. As we calculated in Section 2.2

$$P(D_{F_{\mathrm{bb}}}) = m/g = 3/(5 \times 5) = 0.12\,.$$

For the case of $D_{F_{\mathrm{bf}}}$ the query is a functional expression, so $h = 1$ again holds and there can be only a single answer. As before

$$P(D_{F_{\mathrm{bf}}}) = m/g = 3/5 = 0.6\,.$$

For the case of $D_{F_{\mathrm{fb}}}$ the computation is more complex. For our database $g = 5$ and $h = 3$. Using the equation derived above, we get

$$P_0(D_{\mathrm{fb}}) = \binom{3}{0}\binom{4 \times 3}{3} \Big/ \binom{5 \times 3}{3} = 0.484\,,$$

$$P_1(D_{F_{\mathrm{fb}}}) = \binom{3}{1}\binom{4 \times 3}{2} \Big/ \binom{5 \times 3}{3} = 0.435\,,$$

$$P_2(D_{F_{\mathrm{fb}}}) = \binom{3}{2}\binom{4 \times 3}{1} \Big/ \binom{5 \times 3}{3} = 0.079\,,$$

$$P_3(D_{F_{\mathrm{fb}}}) = \binom{3}{3}\binom{4 \times 3}{0} \Big/ \binom{5 \times 3}{3} = 0.002\,.$$

So

$$P(D_{F_{\mathrm{fb}}}) = 1 - P_0 = 0.516\,,$$

$$P_{\geqslant 2}(D_{F_{\mathrm{fb}}}) = 1 - P_0 - P_1 = 0.081\,,$$

$$P_{\geqslant 3}(D_{F_{\mathrm{fb}}}) = 1 - P_0 - P_1 - P_2 = 0.002\,.$$

Table 3
Probability that a database lookup will succeed for the four different ways of binding the variables in mother and father queries; omitted entries are zero

	P_0	P_1	P_2	P_3	$P_{\geq 1}$	$P_{\geq 2}$	$P_{\geq 3}$
$D_{F_{ff}}$				1	1	1	1
$D_{F_{bf}}$	0.4	0.6			0.6		
$D_{F_{fb}}$	0.484	0.435	0.079	0.002	0.516	0.081	0.002
$D_{F_{bb}}$	0.88	0.12			0.12		
$D_{M_{ff}}$		1			1		
$D_{M_{bf}}$	0.8	0.2			0.2		
$D_{M_{fb}}$	0.8	0.2			0.2		
$D_{M_{bb}}$	0.96	0.04			0.04		

Finally, for the case $D_{F_{ff}}$, $P = P_{\geq 2} = P_{\geq 3} = 1$. A summary of the probability data for queries of the father and mother relations is shown in Table 3.

C.2. Computing E_n for database lookup steps

In Section 2 we assumed that the expected cost, E, was given for individual steps. When more than one answer is sought, the cost of database lookup can vary, depending on the number of answers available and the number of answers sought. We will assume that we are given $C_i(s)$, defined as the additional cost of finding the ith answer (in a database lookup) after the $(i-1)$st has already been found. In computing E_n from the C_i we will assume that the system is smart enough that it will not look for an $(i+1)$st answer if it fails to find an ith answer. Likewise we assume that it will not look for the ith answer if the probability, $P_{\geq i}$, of finding one is zero.

Assume, for a moment, that $P_{\geq n}(s) > 0$. We know that in looking for n answers we will always incur at least the cost $E = C_1$. If one answer is found (probability P) the additional cost of looking for a second answer, C_2, will be incurred. If two answers are actually found (probability $P_{\geq 2}$) the additional cost C_3 will be incurred. And so forth, on up to the number of answers desired, n.

$$E_n(s) = C_1(s) + P(s)C_2(s) + P_{\geq 2}(s)C_3(s) + \cdots + P_{\geq n-1}(s)C_n(s)$$
$$= \sum_{i=1}^{n} P_{\geq i-1}(s)C_i(s) .$$

Alternatively, if $P_{\geq i}(s) = 0$ for all $m < i \leq n$ then the process will stop at m. In this case $E_n(s) = E_m(s)$.

Table 4
Expected cost of looking for solutions in the database for the different types of motherhood and fatherhood queries, $E_i = E_3$ for $i > 3$ since a step is never attempted if the probability of success is zero

	$D_{M_{ff}}$	$D_{M_{bf}}$	$D_{M_{fb}}$	$D_{M_{bb}}$	$D_{F_{ff}}$	$D_{F_{bf}}$	$D_{F_{fb}}$	$D_{F_{bb}}$
E_1	D	D	D	D	D	D	D	D
E_2	D	D	D	D	$2D$	D	$1.5D$	D
E_3	D	D	D	D	$3D$	D	$1.6D$	D

Example. For our example, we assume that $C_i(s) = D$. Thus we get

$$E_n(s) = \begin{cases} D \sum_{i=0}^{n-1} P_{\geqslant i}(s), & \text{if } P_{\geqslant n}(s) > 0, \\ E_m(s), & \text{if } P_{\geqslant i}(s) = 0 \text{ for all } m < i \leqslant n. \end{cases}$$

Using the probability information from Table 3 it is straightforward to compute the expected cost information for the kinship example. The results are summarized in Table 4. Note that for the mother relation all entries are D since $P_{\geqslant 2} = 0$. Likewise for the functional and ground forms of the father relation.

C.3. Computing $P_{\geqslant n}$ and E_n for disjunctive strategies

We next show how to compute E_n and $P_{\geqslant n}$ for strategies. Consider a strategy st consisting of two substrategies s and t that are *independent*, as illustrated in Fig. 18. In this case, s will be used first to try to find n answers to the problem. The expected cost of this is $E_n(s)$. If s produces only $n - i$ answers t will be used to try to find i additional answers. The expected cost of this is $E_i(s)$, and the probability that it will happen is $P_{n-i}(s)$. Thus the expected cost for finding n answers using st will be the sum over all $1 \leqslant i \leqslant n$ of the product of this probability and expected cost.

$$E_n(st) = E_n(s) + \sum_{i=1}^{n} P_{n-i}(s) E_i(t). \tag{C.2}$$

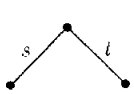

Fig. 18. Combining independent strategies.

Likewise, the probabilities are

$$P_{\geq n}(st) = P_{\geq n}(s) + \sum_{i=1}^{n} P_{n-i}(s)P_{\geq i}(t) ,$$

$$P_n(st) = P_{\geq n}(st) - P_{\geq n+1}(st) = \sum_{i=0}^{n} P_{n-i}(s)P_i(t) .$$

Alternatively, suppose that all steps in a strategy t depend upon all steps in the strategy s, i.e. t cannot be performed unless s is performed. This situation is illustrated in Fig. 19. In this case the strategy s will always be performed. The cost of this is $E(s)$. If s is possible then the strategy t will be used to try to find n answers to the problem (note that s cannot produce any answers itself since it consists entirely of inference actions). The cost of this is $E_n(t)$, and the probability that it will happen is $L(s) = \Pi_{x \in s} L(x)$. Thus

$$E_n(st) = E(s) + L(s)E_n(t) . \tag{C.3}$$

Similarly, the probability of success for st is

$$\begin{aligned} P_{\geq n}(st) &= L(s)P_{\geq n}(t) , \\ P_n(st) &= L(s)P_n(t) . \end{aligned} \tag{C.4}$$

Although the expected cost and probability of success for many strategies can be built up using only these two basic cases, in general a more powerful result is required. Extending the previous notation, let $P_n(s, t)$ refer to the probability that the strategy s produces exactly n answers and that the commonly-rooted strategy t will be possible after executing s.

$$P_n(s, t) \stackrel{\text{def}}{=} L(s_t)P_n(s_{\bar{t}}) .$$

Using this notion, the expected cost and probability equations can be generalized to arbitrary strategies.

Fig. 19. Combining dependent strategies.

Theorem C.5. *If t is a singly rooted strategy,*

$$E_n(st) = E_n(s) + \sum_{i=1}^{n} P_{n-i}(s, t)E_i(t) ,$$

$$P_{\geqslant n}(st) = P_{\geqslant n}(s) + \sum_{i=1}^{n} P_{n-i}(s, t)P_{\geqslant i}(t) ,$$

$$P_n(st) = P_n(s) - P_n(s, t)P(t) + \sum_{i=1}^{n} P_{n-1}(s, t)P_i(t) .$$

Proof. The strategy s is used first to try to find the desired n answers. The expected cost of this is $E_n(s)$. If n answers are found, no part of t will be executed. If only $n - i$ answers are found using s, the strategy t will be used to try to find i additional answers, but only if all of the steps that t depends on were possible. The probability of this is $P_{n-i}(s, t)$. Thus, in that fraction of the cases there will be the additional cost $E_i(t)$. The expected cost is therefore the sum over all $1 \leqslant i \leqslant n$ of the product of this probability and expected cost:

$$E_n(st) = E_n(s) + \sum_{i=1}^{n} P_{n-i}(s, t)E_i(t) .$$

The argument for $P_{\geqslant n}(st)$ is similar.

Using this result, the formula for $P_n(st)$ follows:

$$\begin{aligned} P_n(st) &= P_{\geqslant n}(st) - P_{\geqslant n+1}(st) \\ &= P_{\geqslant n}(s) + \sum_{i=1}^{n} P_{n-i}(s, t)P_{\geqslant i}(t) - P_{\geqslant n+1}(s) \\ &\quad - \sum_{i=1}^{n+1} P_{n+1-i}(s, t)P_{\geqslant i}(t) \\ &= P_n(s) + \sum_{i=1}^{n} P_{n-i}(s, t)P_{\geqslant i}(t) - \sum_{i=0}^{n} P_{n-i}(s, t)P_{\geqslant i+1}(t) \\ &= P_n(s) - P_n(s, t)P(t) + \sum_{i=1}^{n} P_{n-i}(s, t)(P_{\geqslant i}(t) - P_{\geqslant i+1}(t)) \\ &= P_n(s) - P_n(s, t)P(t) + \sum_{i=1}^{n} P_{n-i}(s, t)P_i(t) . \qquad \square \end{aligned}$$

The special case results previously developed for independent and dependent strategies, and for $n = 1$ all follow as corollaries of these general equations.

C.4. Example

Using the information in Tables 3 and 4, and the equations of Theorem C.5, $P_{\geqslant n}$ and E_n can be computed for strategies in the kinship example.

For illustration purposes we show the computation for the strategy

$$S_{P_{\mathrm{bf}}} \stackrel{\mathrm{def}}{=} I_{F_{\mathrm{bf}}} D_{F_{\mathrm{bf}}} I_{M_{\mathrm{bf}}} D_{M_{\mathrm{bf}}} .$$

For convenience let

$$S_F \stackrel{\mathrm{def}}{=} I_F D_F \quad \text{and} \quad S_M \stackrel{\mathrm{def}}{=} I_M D_M .$$

The probabilities for $S_{P_{\mathrm{bf}}}$ are then:

$$P(S_{P_{\mathrm{bf}}}) = P_0(S_{F_{\mathrm{bf}}})P(S_{M_{\mathrm{bf}}}) + P(S_{F_{\mathrm{bf}}}) = 0.4(0.2) + 0.6 = 0.68 ,$$

$$\begin{aligned} P_{\geqslant 2}(S_{P_{\mathrm{bf}}}) &= P_0(S_{F_{\mathrm{bf}}})P_{\geqslant 2}(S_{M_{\mathrm{bf}}}) + P_1(S_{F_{\mathrm{bf}}})P(S_{M_{\mathrm{bf}}}) + P_{\geqslant 2}(S_{F_{\mathrm{bf}}}) \\ &= 0.4(0) + 0.6(0.2) + 0 = 0.12 , \end{aligned}$$

$$\begin{aligned} P_{\geqslant 3}(S_{P_{\mathrm{bf}}}) &= P_0(S_{F_{\mathrm{bf}}})P_{\geqslant 3}(S_{M_{\mathrm{bf}}}) + P_1(S_{F_{\mathrm{bf}}})P_{\geqslant 2}(S_{M_{\mathrm{bf}}}) + P_2(S_{F_{\mathrm{bf}}})P(S_{M_{\mathrm{bf}}}) \\ &\quad + P_{\geqslant 3}(S_{F_{\mathrm{bf}}}) \\ &= 0.4(0) + 0.6(0) + 0(0.2) + 0 = 0 . \end{aligned}$$

For expected cost we get:

$$\begin{aligned} E(S_{P_{\mathrm{bf}}}) &= E(S_{F_{\mathrm{bf}}}) + P_0(S_{F_{\mathrm{bf}}})E(S_{M_{\mathrm{bf}}}) = (I + D) + 0.4(I + D) \\ &= 1.4(I + D) , \end{aligned}$$

$$\begin{aligned} E_2(S_{P_{\mathrm{bf}}}) &= E_2(S_{F_{\mathrm{bf}}}) + P_1(S_{F_{\mathrm{bf}}})E(S_{M_{\mathrm{bf}}}) + P_0(S_{F_{\mathrm{bf}}})E_2(S_{M_{\mathrm{bf}}}) \\ &= (I + D) + 0.6(I + D) + 0.4(I + D) = 2(I + D) , \end{aligned}$$

$$\begin{aligned} E_3(S_{P_{\mathrm{bf}}}) &= E_3(S_{F_{\mathrm{bf}}}) + P_2(S_{F_{\mathrm{bf}}})E(S_{M_{\mathrm{bf}}})P_1(S_{F_{\mathrm{bf}}})E_2(S_{M_{\mathrm{bf}}}) \\ &\quad + P_0(S_{F_{\mathrm{bf}}})E_3(S_{M_{\mathrm{bf}}}) \\ &= (I + D) + 0(I + D) + 0.6(I + D) + 0.4(I + D) = 2(I + D) . \end{aligned}$$

The complete probability and expected cost data for the strategy S is summarized in Tables 5 and 6.

Table 5
Probability data for the strategy S for each of the four different ways of binding the variables to parent queries; omitted entries are zero

	P_0	P_1	P_2	P_3	P_4	$P_{\geq 1}$	$P_{\geq 2}$	$P_{\geq 3}$	$P_{\geq 4}$
$S_{P_{\mathrm{ff}}}$					1	1	1	1	1
$S_{P_{\mathrm{bf}}}$	0.32	0.56	0.12			0.68	0.12		
$S_{P_{\mathrm{fb}}}$	0.3965	0.43	0.1522	0.02084	0.00086	0.6034	0.1739	0.0217	0.00086
$S_{P_{\mathrm{bb}}}$	0.8448	0.1504	0.0048*			0.1552	0.0048*		

Table 6
Expected cost data for the strategy S for each of the four different ways of binding the variables to parent queries; note that $E_i = E_4$ for $i > 4$

	E_1	E_2	E_3	E_4
$S_{P_{\mathrm{ff}}}$	$I + D$	$I + 2D$	$I + 3D$	$2I + 4D$
$S_{P_{\mathrm{bf}}}$	$1.4(I + D)$	$2(I + D)$	$2(I + D)$	$2(I + D)$
$S_{P_{\mathrm{fb}}}$	$1.5(I + D)$	$1.91I + 2.41D$	$2I + 2.6D$	$2I + 2.6D$
$S_{P_{\mathrm{bb}}}$	$1.88(I + D)$	$1.88(I + D)$	$1.88(I + D)$	$1.88(I + D)$

C.5. Computing $P_{\geq n}$ and E_n for conjunctions

In Section 4.2 we only developed equations for computing $P_{\geq 1}$ and E_1 for conjunctions. In general, the equations for $P_{\geq n}$ and E_n are quite nasty:

$$
\begin{aligned}
P_{\geq n}(ab') = \sum_{i=0}^{\infty} P_{\geq i-1}(a)\Bigg[& P_0(b')^i P_{\geq n}(b) \\
& + iP_0(b')^{i-1} \sum_{j=1}^{n-1} P_j(b') P_{\geq n-j}(b') \\
& + \binom{i}{2} P_0(b')^{i-2} \sum_{j-1}^{n-2} P_j(b') \sum_{k=1}^{n-j-1} P_k(b') P_{\geq n-j-k}(b') \\
& + \binom{i}{3} P_0(b')^{i-3} \sum_{j-1}^{n-3} P_j(b') \sum_{k=1}^{n-j-2} P_k(b') \\
& \qquad \times \sum_{l=1}^{n-j-k-1} P_l(b') P_{\geq n-j-k-l}(b') \\
& \vdots \\
& + \binom{i}{n-2} P_0(b')^{i-(n-2)} [P_1(b')^{n-2} P_{\geq 2}(b') \\
& \qquad + (n-2) P_1(b')^{n-3} P_2(b') P(b')] \\
& + \binom{i}{n-1} P_0(b')^{i-(n-1)} P_1(b')^{n-1} P(b') \Bigg].
\end{aligned}
$$

In fact, close inspection shows that the number of terms in this expression is combinatorial in n.

As an alternative, it is possible to use the approximation

$$P_{\geq n} \approx P^n$$

for conjunctions. This approximation is not a very good one for purely disjunctive strategies because the probability distribution may look more like a step function than a bell-shaped curve. For example, if we consider the strategy $S_{P_{bf}}$ for finding the parents of a given individual, the probability distribution plunges to zero for $n > 2$ (people only have two parents), whereas P^n trails off gradually.

However, the approximation seems to be much better for conjunctions, because they have the property that the probability distribution is much smoother than that for any of the individual conjuncts.

ACKNOWLEDGMENT

Narinder Singh gave me the original idea for how to do compile-time analysis of the shape of a search space. Matt Ginsberg helped with some of the mathematics, and gave me the ideas for some of the algorithms in Section 2.4.3. Discussions with Tom Dean, Mike Genesereth and Richard Treitel led to the ideas in Section 5.1.3 on applying the results to forward inference and resolution. Larry Birnbaum, Drew McDermott, and Yehosha Sagiv pointed out the similarities to flattening of the inference space and Ray Reiter brought connection graph theorem proving to my attention. Many members of the Logic Group at Stanford provided constructive criticism on the organization and presentation of this material. Thanks to Cindy Chow, Russ Greiner and Joe Likuski for thorough checking of the equations and proofs.

This work was supported by ONR contract N00014-81-K-0004.

REFERENCES

1. Barnett, J.A., How much is control knowledge worth?: A primitive example, *Artificial Intelligence* **22** (1984) 77–89.
2. Feldman, J.A., and Sproull, R.F., Decision theory and Artificial Intelligence II: The hungry monkey, *Cognitive Sci.* **1** (1977) 158–192.
3. Garey, M.R., Optimal task sequencing with precedence constraints, *Discrete Math.* **4** (1973) 37–56.
4. Klahr, P., Planning techniques for rule selection in deductive question-answering, in: D.A. Waterman and F. Hayes-Roth (Eds.), *Pattern-Directed Inference Systems* (Academic Press, New York, 1978) 223–239.
5. Korf, R., Planning as search: A quantitative approach, *Artificial Intelligence* **33** (1) (1987) 65–88.
6. Natarajan, K.S., Adaptive search: An approach to optimize search effort for problem solving, Research Rept. 12228, IBM (1986).
7. Natarajan, K.S., On optimizing backtrack search for all solutions to conjunctive problems, Research Rept. 11982, IBM (1986).
8. Natarajan, K.S., Optimizing depth-first search of AND-OR trees, Research Rept. 11842, IBM (1986).

9. Natarajan, K.S., Optimizing backtrack search for all solutions to conjunctive problems, in: *Proceedings IJCAI-87*, Milan, Italy (1987) 955–958.
10. Raiffa, H., *Decision Analysis: Introductory Lectures on Choices under Uncertainty* (Addison-Wesley, Reading, MA, 1970).
11. Simon, H.A., and Kadane, J.B., Optimal problem-solving search: All-or-none solutions, *Artificial Intelligence* **6** (1975) 235–247.
12. Smith, D.E., A decision theoretic approach to the control of planning search, Tech. Rept. LOGIC-87-11, Stanford University, Stanford, CA (1988).
13. Smith, D.E., and Genesereth, M.R., Ordering conjunctive queries, *Artificial Intelligence* **26** (1985) 171–215.
14. Smith, D.E., Genesereth, M.R., and Ginsberg, M.L., Controlling recursive inference, *Artificial Intelligence* **30** (1986) 343–389.
15. Sproull, R., Strategy construction using a synthesis of heuristic and decision-theoretic methods, Ph.D. Thesis, Stanford University, Stanford, CA (1977).
16. Treitel, R., Sequentialization of logic programs, Ph.D. Thesis, Tech. Rept. STAN-CS-86-1135, Stanford University, Stanford, CA (1986).
17. Treitel, R.J., and Genesereth, M.R., Choosing directions for rules, *J. Autom. Reasoning* **3** (1987) 395–431.
18. Treitel, R.J., and Smith, D.E., Effects of changing inference direction on the optimal ordering of a rule's antecedents, Tech. Rept. LOGIC-87-12, Stanford University, Stanford, CA (1987).

Received March 1987; revised version received March 1988