Appears in the
Proceedings of the Second International Conference on Knowledge Representation and Reasoning (KR-91),
Boston, May 1991.

Probably Approximately Optimal Derivation Strategies

Russell Greiner
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4

Pekka Orponen*
Department of Computer Science
University of Helsinki
SF-00510 Helsinki, Finland

Abstract

An inference graph can have many "derivation strategies", each a particular ordering of the steps involved in reducing a given query to a sequence of database retrievals. An "optimal strategy" for a given distribution of queries is a complete strategy whose "expected cost" is minimal, where the expected cost depends on the conditional probabilities that each requested retrieval succeeds, given that a member of this class of queries is posed. This paper describes the PAO algorithm that first uses a set of training examples to approximate these probability values, and then uses these estimates to produce a "probably approximately optimal" strategy — *i.e.*, given any $\epsilon, \delta > 0$, PAO produces a strategy whose cost is within ϵ of the cost of the optimal strategy, with probability greater than $1-\delta$. This paper also shows how to obtain these strategies in time polynomial in $1/\epsilon$, $1/\delta$ and the size of the inference graph, for many important classes of graphs, including all and-or trees.

1 Introduction

General problem solving (a.k.a. deduction) is expensive. There can be an exponential number of potential "solution paths" for a given query/goal — as there can be many rules/operations that each reduce the goal to a new set of subgoals, and each of these subgoals can, itself, have many possible reductions, etc. [GN87]. It is obviously advantageous to explore these different paths in a good order, one that will lead to a solution quickly. This "best ordering" depends on the probabilities that each path will succeed; unfortunately, this information is usually unavailable. In addition, the task of finding the optimal strategy, even given these probabilities, can be very difficult; *c.f.*, [Gre91b]. This report presents an efficient technique for finding good approximations to these probabilities, and then using them to produce near-optimal derivation systems.

The rest of this section provides the framework for this research, and discusses how it relates to the work on satisficing search strategies, effective use of control information, and explanation-based learning. Section 2 then describes our underlying task: designing efficient algorithms that first use samples to approximate an unknown distribution, and then use this estimate to produce good strategies, with provably high probability. The following two sections present such algorithms, for various classes of situations: Section 3 first presents a relatively simple version, PAO_0, that is sufficient for one class of knowledge bases. Section 4 then presents various modifications to this algorithm, producing systems that are more efficient and are able to handle other situations.

Framework: The objective of a derivation process (DP) is to determine whether a given query follows from a given knowledge base (KB). This work focuses on "backward chaining derivation processes", atomic literal queries and non-recursive "definite clause KBs".[1] Here, an atomic formula σ (*e.g.*, the initial query) follows from the KB if a matching fact is in the KB (*i.e.*, if the "database retrieval" for σ succeeds) or if there is a rule whose conclusion matches σ and whose antecedents all follow from the KB. Our DP can, therefore, backward-chain from the query through the rules, to reach a *satisficing set* of retrievals — *i.e.*, the query follows whenever each of these retrievals succeeds. (*E.g.*, consider the trivial knowledge base KB_C, whose rules appear in Figure 1.

*This research was performed while this author was visiting the Department of Computer Science at the University of Toronto.

[1] That is, every clause in the KB's CNF form contains exactly one positive literal. The atomic clauses are called "facts", and the others, "rules".

Here, the singleton $\{ D_c \}$ set of retrievals (corresponding to the `Cheap(C1)` literal) forms a *satisficing set* for the `BuyCar(C1)` query, based on the R_{bc} rule.)

In general, the DP can use the KB's rules to reduce the query to various satisficing sets of retrievals. A satisficing set of retrievals is a *solution* if each member of that set succeeds — *i.e.*, if each of these literals is present in the KB. Notice that different satisficing sets will be solutions for different queries. (*E.g.*, as $\{$ `Cheap(C1)`, `Pretty(C2)` $\}$ are all of KB_C's facts, $\{ D_c \}$ represents a solution for the `BuyCar(C1)` query, but not for either `BuyCar(C2)` or `BuyCar(C19)`.)

It is convenient to arrange these rules into an "inference graph", where each node corresponds to an atomic formula that appears in some rule, and each arc, to a rule or a database retrieval; the tree in Figure 1 is an example. Derivation can then be viewed as a search through this graph, trying to find some solution.

We can therefore reduce our task of building an efficient derivation process to a graph theoretic task, of finding a good "strategy", where a strategy is a fixed ordering on these arcs, specifying when to use each rule and when to perform each retrieval. (For example, given the query `BuyCar(C3)`, the sequence $\Theta_1 = \langle R_{bc}\, D_c\, R_{bp}\, D_p \rangle$ represents the strategy that first follows the R_{bc} rule down to the `Cheap(C3)` database retrieval, D_c. If that retrieval succeeds, the derivation process returns "success" and terminates. If not, it then pursues the strategy's other sub-path: following the R_{bp} rule down to the `Pretty(C3)` database retrieval, D_p. The derivation process then returns either success or failure, based on the success of this second retrieval.) An alternative to this strategy, $\Theta_2 = \langle R_{bp}\, D_p\, R_{bc}\, D_c \rangle$ first follows R_{bp} to D_p and then, if needed, R_{bc} to D_c. Which of the two strategies is better?

A "good" strategy is one that performs well in practice. To quantify this, we assume there is some distribution of queries that our derivation system will be asked to solve. We can then define a strategy's *expected cost* as the average cost[2] required to find one solution for a query (or to determine that no solution exists) averaged over the distribution of anticipated queries, $\mathcal{Q}$. This value depends on the computational cost of each reduction and retrieval, and on the *a priori* likelihoods that each particular retrieval will succeed — *i.e.*, the conditional probability that a specific requested proposition will be in the KB, given that the query is taken from $\mathcal{Q}$. (Continuing with the same KB_C, assume that each reduction and retrieval costs 1 unit, and imagine we knew that 60% of the queries would be `BuyCar(C1)`, 25% would be `BuyCar(C2)`, and 15%, `BuyCar(C3)`. Here, the D_c retrieval will succeed 60% of the time, and D_p, 25%. We can now compute Θ_1's expected cost, $E[\Theta_1]$: it is the cost of the initial $\langle R_{bc}\, D_c \rangle$ subpath, plus the probability that this path will fail $(1 - 0.6 = 0.4)$ times the cost of $\langle R_{bp}\, D_p \rangle$; hence, $E[\Theta_1] = (1+1) + 0.4(1+1) = 2.8$.) In general, a strategy's expected cost is the weighted sum of the costs of each path to its set of retrievals, weighted by the probability that we will need to pursue this path (*i.e.*, that none of the prior paths succeeded).[3]

We can now state our objective: to find a graph traversal strategy that is complete (*i.e.*, is guaranteed to find an solution if there is one) and has the minimal expected cost of all such complete strategies [Gre91a]. We consider such strategies to be *optimal*.

There are several known algorithms for computing an optimal strategy for various classes of KBs (or for the related, more abstract task of finding the optimal *satisficing search strategy* for various general search structures [OG90]). Most, including [Gar73, Bar84, SK75, Nat86, Gre91a], assume that the success probabilities are given initially. Unfortunately, these probabilities are, in general, not known initially (*e.g.*, we usually do *not* know that 60% of the queries will be `BuyCar(C1)`, etc.). [Smi89] presents one way of approximating their values, based on the (questionable) assumption that these probabilities are correlated with the distribution of facts in the database. For example, assume that KB'_C includes the same rule base as KB_C together with 1,000 facts of the form `Cheap(`κ`)` (each using a different constant κ) and 10 facts of the form `Pretty(`κ`)`. Given any goal of the form `BuyCar(`κ`)` for some constant κ — *e.g.*, `BuyCar(C19)` — that system assumes that we are 100 times more likely to find the corresponding `Cheap` fact (here "`Cheap(C19)`") than the `Pretty` fact ("`Pretty(C19)`"). This means [Smi89]'s algorithm would claim that Θ_1 is the optimal strategy (*i.e.*, it is better than the other legal candidate, Θ_2).

This, of course, need not be true; there is no reason to assume the distribution of the user's queries will be correlated with the distribution of facts in the KB. The user may, for example, only ask questions whose answers just happen to come from the `Pretty` facts. (For example, imagine that all of the queries dealt with RollsRoyces — *i.e.*, were of the form `BuyCar(`κ`)`, where κ referred to a not-cheap Rolls Royce.) What we really need is the *conditional probability that a given retrieval will succeed, given that a query from a particular class is asked* — *i.e.*, $Pr[$ `Cheap(`κ`)` retrieval succeeds $|$ query `BuyCar(`κ`)` is asked $]$. That is, imagine the user asks a total of K `BuyCar(`κ`)` queries over the lifetime of the sys-

[2] While we usually think of this "cost" as time, it could also be number of external file reads, or any other positive measure.

[3] It is actually more complicated, as we need only consider the cost of the *additional* steps, to go from some already-visited nodes down to the retrievals. See [Gre91a].

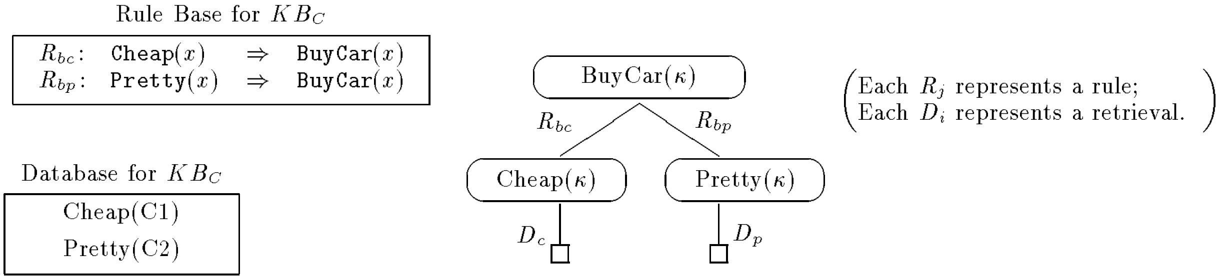

Figure 1: The KB_C knowledge base and its associated inference graph, G_C

tem (*e.g.*, `BuyCar(C1)`, `BuyCar(C19)`, `BuyCar(C2)`, ...); and of these, the related `Cheap`(κ) retrieval (if requested) would succeed a total of $\#c(K)$ times. Then the value we need, the real value of $Pr[$ `Cheap`(κ) succeeds | `BuyCar`(κ) asked], is the ratio of these numbers — $\frac{\#c(K)}{K}$.

While there is no empirical way of finding such probabilities exactly (short of running all K trials), there is a meaningful way of estimating their values, based on the observation that the past can be a good predictor of the future. This basic observation is the key insight underlying "explanation-based learning" systems [MKKC86, DM86, MCK+89]. Each of these systems analyzes the solutions found to certain previous problems, and many use this information to suggest new strategies. Most of these systems, however, use only a single training example. Empirical studies have confirmed that the resulting strategies are not always good [Min88].

By contrast, this research uses a *set* of examples, and can guarantee that the resulting strategy will usually be arbitrarily close to optimal. In particular, this paper presents an efficient process that determines the number of training examples needed to approximate these probability values with the property that, with high probability, these estimates will be good enough to produce an approximately optimal derivation strategy. It assumes only that there is a set of queries that our DP will encounter, and that these queries are selected at random, according to some arbitrary but stationary distribution. (Notice this distribution will depend on the particular task our DP is addressing; *n.b.*, we do not assume that it is a uniform distribution over all possible problems [Gol79], nor that it will necessarily correspond to any particular collection of "benchmark challenge problems" [Kel87]. Hence, we are using the same weak "distribution-free assumption" that underlies the current work in PAC-learning [Val84].) We can translate this query distribution into a set of probability values that specify the chance that each retrieval will succeed. We assume, also, that these success probabilities (of the different retrievals) are independent of one another.

2 The PAO Task

This section defines our task in very abstract terms; the next two sections present concrete algorithms that achieve this task, in various situations.

As mentioned above, a derivation process, DP, is a program that answers queries based on the information in a knowledge base. The cost it requires to produce that answer will depend on the particular *strategy* it uses. We assume there is a class of strategies associated with any given knowledge base (*e.g.*, Θ_1 and Θ_2 are the two strategies associated with KB_C) and define DP_Θ to be the process that uses the strategy Θ.

Each strategy will perform various "probabilistic experiments" (*e.g.*, attempted database retrievals), which each succeed for certain queries, and fail for others. (*E.g.*, the two experiments associated with KB_C are D_c and D_p, corresponding to the `Cheap`(κ) and `Pretty`(κ) attempted database retrievals; notice D_c succeeds when the query is `BuyCar(C1)`, but fails when the the query is `BuyCar(C2)`.) We can, in general, identify each query with the subset of the experiments that succeed in that context: *e.g.*, using the $\mathcal{L}$ function that maps each query to the associated set of successful probabilistic experiments, $\mathcal{L}($`BuyCar(C1)`$) = \{D_c\}$ and $\mathcal{L}($`BuyCar(C19)`$) = \{\}$.

We define $e[\Theta, q]$ to be the cost for DP_Θ to answer the query q. (Notice this depends only on which experiments succeed; *i.e.*, on the value of $\mathcal{L}(q)$.)

We assume that the queries are drawn from a set $\mathcal{Q}$ at random, according to some stationary distribution. This means we can consider the set of $e[\Theta, q_i]$ to be independent, identically-distributed random variables, and then define $E[\Theta]$ to be their common expected value; *i.e.*,

$$E[\Theta] = \mathrm{average}_{q \in \mathcal{Q}}\, e[\Theta, q]$$

As we further assume that the experiments are independent, we can view $E[\Theta]$ as a function of their success probabilities; to make this explicit, we will write $E_P[\Theta]$, where P is the vector of the probability values for the various experiments. (*E.g.*, $P = \langle p_c, p_p \rangle$ where $p_c = Pr[$ `Cheap`(κ) succeeds | `BuyCar`(κ) asked] and $p_p = Pr[$ `Pretty`(κ) succeeds | `BuyCar`(κ) asked].)

We can then define the *optimal strategy* as the complete strategy whose expected cost is minimal, with respect to the distribution (*i.e.*, probability vector, P):

$$\Theta_{opt} \text{ is } optimal \iff \forall \Theta.\, E_P[\Theta_{opt}] \leq E_P[\Theta]$$

We can now define the PAO task: Each problem instance takes

- $\mathcal{S}$, the set of possible derivation processes, all based on a given KB
  (e.g., the derivation processes DP_{Θ_1} and DP_{Θ_2} associated with KB_C, based on the Θ_1 and Θ_2 strategies, respectively)
- $\mathcal{Q}$, a (stationary) distribution of the queries
  (e.g., {`BuyCar(C1)`, `BuyCar(C3)`, `BuyCar(C7)`, ...})
- $\mathcal{O}$, an oracle that generates appropriate queries from $\mathcal{Q}$ according to this fixed (but unknown) distribution,
- $\epsilon \in \Re^+$ (*i.e.*, a bound on the allowed excess), and
- $\delta \in (0, 1]$ (*i.e.*, the required confidence).

For each instance, we seek a strategy $\Theta_{pao} \in \mathcal{S}$, whose expected cost is, with high probability, close to the cost of the optimal strategy. Stated more precisely, given the real distribution (here, the $P = \langle p_c, p_p \rangle$ defined above), there is an optimal strategy, $\Theta_{opt} \in \mathcal{S}$, whose expected cost is $E_P[\Theta_{opt}]$. With high probability (*i.e.*, $\geq 1 - \delta$), the cost of the result of this algorithm, $E_P[\Theta_{pao}]$, will be no more than ϵ over $E_P[\Theta_{opt}]$ — *i.e.*,

$$Pr[\ E_P[\Theta_{pao}] \leq E_P[\Theta_{opt}] + \epsilon\ ] \ \geq\ 1 - \delta\ .$$

Notes: [1] We insist that the algorithm for this task be efficient — *i.e.*, run in time that is polynomial in $1/\epsilon$, $1/\delta$ and the "size" of knowledge base.
[2] Each PAO algorithm will be responsible for computing the $\mathcal{L}$ function — *i.e.*, it must use the knowledge base to determine which experiments are successful, for each observed query.
[3] The oracle can simply be the *user* of the DP system, who is going about his usual business of asking the system relevant queries. (That is, he can ask his questions during the "training mode", while PAO is computing the pao strategy, as well as afterwards.) Of course, the overall DP+PAO system would have to return the solution found (as a side-effect, wrt PAO).

3 Simple PAO_0 Algorithm

This section presents the (relatively simple but general) PAO_0 algorithm that, with high probability, identifies approximately optimal strategies for a certain specified class of KBs (*viz.*, non-recursive definite class KBs whose inference graphs are hyper-trees; see below). After Subsection 3.1 presents the necessary definitions, Subsection 3.2 describes the algorithm itself. Subsection 3.3 sketches the proofs that PAO_0 works correctly and is efficient. The next section presents several extensions to this basic algorithm, designed to provide answers more efficiently and to handle other situations.

3.1 Definitions

The definitions below define the particular $\mathcal{S}$ used, and constrain $\mathcal{Q}$:

- We can identify any non-recursive definite clause KB with a particular **inference graph** $G = \langle N, A, S, B, c \rangle$, where N is a set of nodes, each node corresponds to an atomic literal that must be proven towards achieving some goal; $A \subset N \times 2^N$ is the set of directed hyper-arcs, each arc corresponds either to an invocation of a rule or a database retrieval.[4] The set $S \subset 2^N$ corresponds to the set of "success node-sets"; reaching each element of some member (*i.e.*, reaching each $n \in s$ for some $s \in S$) means the overall derivation is successful. (For the G_C inference graph associated with the KB_C knowledge base, these sets are the singletons $\{D_c\}$ and $\{D_p\}$, as we need only find a single successful database retrieval to establish the given goal in this simple context.[5]) The set $B \subset A$ corresponds to the "blockable" arcs — each $b \in B$ is an arc that can be "blocked" (*i.e.*, not succeed) for some query. (*E.g.*, D_c is blockable as it does not succeed for the `BuyCar(C2)` query.) The *cost function* $c\colon N \times N \mapsto \Re^+$ gives the positive real valued cost of each part of a hyper-arc in G.[6]

  PAO_0 deals only with inference graphs that are "hyper-trees"; *i.e.*, connected directed graphs, each with a unique "root" node (having no parents), where every other node has a single parent. (See Subsection 4.2.)

  As the KB_C knowledge base is "disjunctive" (*i.e.*, it is definite clause and each of its CNF clauses includes at most one negative literal), its associated inference graph, G_C, relatively easy — in particular, each node descends to a single child node. Appendix A provides an example of a less trivial hyper-tree, based on non-disjunctive KB.

- Given any such inference hyper-tree G, we let $\mathcal{S}(G) = \{\mathrm{DP}_\Theta\}$ represent the set of derivation

[4] The graph only includes an arc corresponding to a database retrieval if the corresponding proposition *might* be in the fact set. Notice, for example, that the G_C inference graph does not include an arc corresponding to the `BuyCar(`κ`)` retrieval, as the KB's fact set does not include any such propositions.

[5] S includes only the *minimal* success sets; *n.b.*, any superset of any member of S is also sufficient. Hence, the set $\{D_c,\ D_p\}$ is sufficient.

[6] Notice these values are not used when answering queries; they are, however, essential for parts of our PAO algorithms.

processes, where each DP_Θ uses its own **strategy** Θ to specify how it will search this inference graph to find an answer to a given query. In the case of simple disjunctive KBs, each strategy corresponds to a sequence of the elements of A;[7] DP_Θ follows these arcs, in this order, stopping as soon as it reaches a success node. (*E.g.*, the DP_{Θ_1} process, based on the $\Theta_1 = \langle R_{bc}\, D_c\, R_{bp}\, D_p \rangle$ strategy, will stop if the D_c retrieval is successful, ignoring the remaining $\langle R_{bp}\, D_p \rangle$.) Notice this means that our DPs are seeking only a single solution; this type of search is called a "satisficing search" [SK75].[8]

Strategies are considerably more complicated when dealing with non-disjunctive KBs; see Appendix A.

- Each **query** (*i.e.*, each member of Q) corresponds to an instantiation of the root node of G.

Hence, each DP_Θ is given a query, q, as its input. It traverses the inference graph G, beginning at the root node (corresponding to q), in the order described by Θ, trying to reach a satisfying set of success nodes (*i.e.*, all nodes in some member of S). DP_Θ is only allowed to use the arcs that are not blocked for this query, *i.e.*, the members of $\mathcal{L}(q)$.

In general, the expected cost of a strategy, $E[\Theta]$, can be expressed as a function of the success probabilities of the various experiments, and the values of the c cost function; see Appendix A.

3.2 Basic PAO_0 Algorithm

The PAO_0 algorithm presented in this subsection deals with satisficing searches through any KB whose inference graph is a hyper-tree. It consists of three subroutines, $\textsc{Get-Number}_0$, $\textsc{Find-Probs}_0$ and $\textsc{Find-OptDS}_0$, that are called in sequence. The $\textsc{Get-Number}_0$ procedure takes as input an inference graph, G, and the two "error terms", ϵ and δ, and returns an integer

$$M \;=\; \mathcal{GN}_{\mathcal{AOG}}(G,\, \epsilon,\, \delta) \;\stackrel{def}{=}\; \left\lceil 2 \left(\frac{nC}{\epsilon}\right)^2 \ln \frac{2n}{\delta} \right\rceil \quad (1)$$

where $C = \sum_{a \in A} c(a)$ is the sum of the costs of all of the arcs in the inference graph G, and $n = |B|$ is the total number of probabilistic arcs.

The $\textsc{Find-Probs}_0$ procedure then calls the $\mathcal{O}$ oracle M times to obtain M queries from the class Q. For each of these $q_i \in Q$, it must determine the value of $\mathcal{L}(q_i)$ — *i.e.*, which of the probabilistic arcs are blocked. (*E.g.*, it must determine, for the `BuyCar(C1)` query, that the D_c retrieval succeeds but the D_p retrieval does not.) This involves searching through the entire inference graph. We implement it using the $\mathrm{DP}^{(1)}$ derivation process, that finds *all* solutions to a given query; *i.e.*, it goes down all possible paths, and attempts each probabilistic arc.[9]

After all M queries, $\textsc{Find-Probs}_0$ tallies how many times each probabilistic arc succeeded, and translates this number into a frequency value. (Continuing the above example, assume that $\textsc{Get-Number}_0$ returns the value $M_{G_C} = 30$ for the G_C graph and some values of ϵ and δ. $\textsc{Find-Probs}_0$ would then pose $M_{G_C} = 30$ queries, each of the form `BuyCar(`κ`)` for some appropriately-distributed set of ground κs, to the KB_C knowledge base and observe that the associated `Cheap(`κ`)` query (resp., `Pretty(`κ`)` query) succeeded 10 times (resp., 18 times). It would then produce the frequency values of $\hat{p}_c = \frac{10}{30}$ and $\hat{p}_p = \frac{18}{30}$.)

$\textsc{Find-Probs}_0$ gives these approximate probability values and the inference graph to the $\textsc{Find-OptDS}_0$ procedure, that produces the *ϵ, δ-probably-approximately-optimal* derivation strategy, Θ_{pao}. $\textsc{Find-OptDS}_0$ is simply the $\Upsilon_{\mathcal{AOT}}(G, Q)$ algorithm (defined in [Gre91a]) that takes G, a general hyper-tree, and Q, a vector of (approximations to) the success probabilities of the retrievals, and returns the strategy that would be optimal, if these Q values were accurate; *i.e.*, $\Theta_Q = \Upsilon_{\mathcal{AOT}}(G, Q)$ satisfies $\forall \Theta \in \mathcal{S}(G)\ E_Q[\Theta] \geq E_Q[\Theta_Q]$. PAO_0 then returns this value, $\Theta_{\hat{P}} = \Upsilon_{\mathcal{AOT}}(G, \hat{P})$.

3.3 Proof Sketches

The critical claim underlying the PAO_0 algorithm is the formula in Equation 1, which is proven in [GO91, Theorem 1]. The basic ideas follow.

Let G be an inference graph with n probabilistic arcs; and $P = \langle p_1, \ldots, p_n \rangle$ be the real probability vector — *i.e.*, p_i is the (real) success probability of the i^{th} probabilistic arc. The real optimal strategy for the inference graph G, based on these probability values, is $\Theta = \Upsilon_{\mathcal{AOT}}(G, P)$. The expected cost of this strategy (again based on the real probability vector P) is $E_P[\Theta]$. Imagine, however, that we only have an approximation to P, called $\hat{P}$; the "optimal" strategy, based on this vector, is $\hat{\Theta} = \Upsilon_{\mathcal{AOT}}(G, \hat{P})$. How bad is this $\hat{\Theta}$ — that is, how much more might this strategy cost, over the optimal one? This depends on how close each p_i is to the corresponding component of $\hat{P}$ (called $\hat{p}_i$). The answer appears below:

[7] Of course, not every sequence qualifies; in particular, we only consider sequences where each arc descends from a node that has been already been reached. Hence, $\langle D_p, R_{bp}, \ldots \rangle$ is not a legal strategy as its first step, the D_p arc, descends from a node that has not yet been reached.

[8] Notice it applies to ground queries as well as existentially quantified queries, when the user is seeking only a single binding list for those variables.

[9] Notice this $\mathrm{DP}^{(1)}$ differs from the DP_Θs, as each DP_Θ will stop as soon as it finds a solution. Hence, DP_{Θ_1} will stop if the D_c retrieval succeeds, and will not consider the $\langle R_{bp},\, D_p \rangle$ path. By contrast, $\mathrm{DP}^{(1)}$ will always try both D_c and D_p, even if the initial D_c retrieval succeeds.

Lemma 3.1 (Sensitivity Analysis of $\Upsilon_{\mathcal{AOT}}(G, P)$) *Let $G = \langle N, A, S, B, c\rangle$ be an inference graph with n probabilistic arcs $B = \langle b_1, \ldots, b_n\rangle$, and let $P = \langle p_1, \ldots, p_n\rangle$ and $\hat{P}_i = \langle \hat{p}_1, \ldots, \hat{p}_n\rangle$ be two n-element probability vectors whose respective i^{th} components differ by at most $\lambda_i \in \Re$; i.e., $\forall i.\ |p_i - \hat{p}_i| \le \lambda_i$. Then*

$$E_P[\Upsilon_{\mathcal{AOT}}(G, \hat{P})] \ \le\ E_P[\Upsilon_{\mathcal{AOT}}(G, P)] + 2\sum_i \lambda_i \cdot C_{\neg}[b_i]$$

where $C_{\neg}[b_i]$ is the sum of the costs of the arcs in G that are not *on the path from the G's root, through b_i, to a success node.*

(To illustrate the $C_{\neg}[\cdot]$ function: In the G_C graph, the value of $C_{\neg}[D_p]$ is the sum of the costs of the R_{bc} rule and the D_c retrieval; *i.e.*, $C_{\neg}[D_p] = c(R_{bc}) + c(D_c)$. Likewise, $C_{\neg}[D_c] = c(R_{bp}) + c(D_p)$. Notice each $C_{\neg}[b_i] \le C$, where $C = \sum_{a\in A} c(a)$ is the sum of the costs of all of the arcs in G.)

Given any ϵ, therefore, we need to sample each experiment enough times that we are confident that each $\hat{p}_i$ is within $\lambda_i = \frac{\epsilon}{2nC}$ of the real value, p_i. To show that $\mathcal{GN}_{\mathcal{AOG}}(G, \epsilon, \delta)$ trials of each experiment is sufficient, we need only the assertion that these experiments are independent, and the following simple form of Chernoff's bound:

Lemma 3.2 (from [Bol85, p. 12]) *If M independent experiments are performed to obtain an estimate $\hat{p}$ for a probability p, and $\lambda \ge 0$, then*

$$Pr[\,|\hat{p}_i - p_i| > \lambda\,] \ \le\ 2e^{-2M\lambda^2}$$

Finally, this algorithm is efficient:

Lemma 3.3 (PAO_0's Efficiency) *The PAO_0 algorithm runs in time polynomial in $\frac{1}{\epsilon}$, $\frac{1}{\delta}$ and $|G|$, the size of the inference graph (assuming a constant time oracle $\mathcal{O}$ and constant time database retrieval process).*

(The proof involves the observations that $\textsc{Get-Number}_0$ requires only $O(|G|)$ to compute the values of n and C; $\textsc{Find-Probs}_0$ requires only $O(\frac{1}{\epsilon}^2 \frac{1}{\delta} |G|^3)$ (*i.e.*, $O(|G|)$ to traverse the inference graph to perform the n retrievals for each of its $M = O(\frac{1}{\epsilon}^2 \frac{1}{\delta} |G|^2)$ trials); and $\textsc{Find-OptDS}_0$ is $O(|G|^2)$ [Gre91a].)

4 Extensions to the PAO Algorithm

For pedagogical reasons, the previous section presented a very simple algorithm, one that is both relatively restricted (able to handle only inference hypertrees[10]) and relatively inefficient. This subsection presents several alternate PAO_i algorithms that are more efficient or can process other situations (*e.g.*, other classes of inference graphs, etc.). ([GO91] discusses yet other extensions to the basic PAO_0 algorithm.)

4.1 Fewer Total Number of Retrievals

The $\textsc{Find-Probs}_0$ part of the PAO_0 system must perform $M \cdot n$ retrievals for a general and-or inference tree — *i.e.*, it must perform all n retrievals on each of M trials. We show below two modifications that allow $\textsc{Find-Probs}$ to perform a smaller total number of retrievals.

Use Different Formulae for Different Classes of Graphs: The $\mathcal{GN}_{\mathcal{AOG}}$ formula is an upper bound, that applies to any and-or inference tree.[11] Lower values apply to certain subclasses; for example, we need only

$$\mathcal{GN}_{2o}(\ G,\ \epsilon,\ \delta\ ) \ \stackrel{def}{=}\ \lceil 2\left(\frac{c_r}{\epsilon}\right)^2 \ln\frac{2}{\delta}\rceil$$

trials for any inference graph of the form shown in Figure 1 — *i.e.*, involving exactly two simple (non-hyper) paths, each leading to a singleton solution set through a single probabilistic arc, each path of equal cost, called c_r (*e.g.*, for G_C, this is the cost of each rule plus its associated retrieval; *i.e.*, $c_r = c(R_{bc}) + c(D_c) = c(R_{bp}) + c(D_p)$).[12] Here, $\textsc{Find-Probs}$ would have to perform only $\mathcal{GN}_{2o}(\cdots) < \mathcal{GN}_{\mathcal{AOG}}(\cdots)$ trials.

The $\textsc{Get-Number}_1$ algorithm[13] can exploit such formulae. This process first identifies the given graph with the appropriate formula, $\mathcal{GN}_{\mathcal{G}}$ (where some appropriate set of formulae, $\{\mathcal{GN}_{\mathcal{G}}\}$, is known in advance), then evaluates that $\mathcal{GN}_{\mathcal{G}}$ formula on the appropriate values, corresponding to various numbers associated with this graph (*e.g.*, the number of probabilistic arcs, particular costs of the arcs, etc.).

The classification part is relatively straightforward, usually based on certain properties of the KB, such as whether it is propositional (rather than in general predicate calculus), or definite, or disjunctive, etc. We consider only simple tests that can be run in time poly-

[10] Technically, we should state "KBs whose inference graphs are hyper-trees". To simplify our descriptions, we will often identify a class of KBS with their class of inference graphs.

[11] In fact, this formula applies to any and-or directed acyclic hyper-graphs, not just hyper-trees. We can therefore use it for the class of DAGs discussed in Appendix B.

[12] This is proven in [GO91, Lemma 3].

[13] In general, each $\textsc{Get-Number}_i$ (resp., $\textsc{Find-Probs}_i$, $\textsc{Find-OptDS}_i$) is a variant of the $\textsc{Get-Number}_0$ (resp., $\textsc{Find-Probs}_0$, $\textsc{Find-OptDS}_0$) algorithm discussed above. Each PAO_i algorithm is composed on $\textsc{Get-Number}_i$, $\textsc{Find-Probs}_i$ and $\textsc{Find-OptDS}_i$. Notice that some of these modified versions will accept slightly different sets of inputs, or return slightly different results, than the originals.

nomial in the size of the inference graph; notice all three of the tests mentioned above qualify.

Different Number of Samples for Different Experiments: Both PAO_0 and PAO_1 (the version that uses GET-NUMBER$_1$) assume that FIND-PROBS will perform the same number of trials for each probabilistic arc (*e.g.*, for each retrieval). However, we see from the lemmata above that certain arcs require fewer trials than others. In particular, for general hyper-trees, each $d_i \in D$ retrieval requires only

$$m_i = \mathcal{GN}'_{\mathcal{AOG}}(d_i, G, \epsilon, \delta) \quad \stackrel{def}{=} \quad \lceil 2\left(\frac{n\, C_\neg[d_i]}{\epsilon}\right)^2 \ln\frac{2n}{\delta}\rceil \tag{2}$$

trials, using $C_\neg[\cdot]$ function defined in Lemma 3.1. Notice that

$$\mathcal{GN}'_{\mathcal{AOG}}(d_i, G, \epsilon, \delta) \quad < \quad \mathcal{GN}_{\mathcal{AOG}}(G, \epsilon, \delta) = M$$

We now use this observation in two ways. First, we can continue to use the GET-NUMBER$_1$ algorithm, but identify the class of and-or hyper-trees with the

$$\tilde{M} \;=\; \tilde{\mathcal{GN}}_{\mathcal{AOG}}(G,\, \epsilon,\, \delta) \;\stackrel{def}{=}\; \max_d\{\mathcal{GN}'_{\mathcal{AOG}}(d,\, G,\, \epsilon,\, \delta)\}$$

formula, rather than $\mathcal{GN}_{\mathcal{AOG}}$. (Here, we would need only $\tilde{M} \cdot n$ total samples, which is strictly less than $M \cdot n$.)

Second, we can define the PAO_2 algorithm (that uses modified GET-NUMBER$_2$ and FIND-PROBS$_2$ subroutines) to perform an even smaller total number of retrievals. Here, GET-NUMBER$_2$ returns a *vector* of numbers $\vec{M} = \langle m_1, \ldots m_n\rangle$, where each m_i is the number of trials required of the d_i database retrieval, defined in Equation 2. The FIND-PROBS$_2$ algorithm would then perform only m_i retrievals, for each d_i. Hence, this requires only $\sum_{i=1}^{n} m_i < M \cdot n$ retrievals.

Of course, FIND-PROBS$_2$ cannot simply use $DP^{(1)}$ (as FIND-PROBS$_0$ had), as that would force FIND-PROBS$_2$ to perform the same number of samples for each retrieval. Therefore, FIND-PROBS$_2$ uses the different (though related) $DP^{(2)}$ procedure that only pursues a path if necessary — *e.g.*, it would only follow a path (*e.g.*, $\langle R_{bp}, D_p\rangle$) if it contain a probabilistic arc (*e.g.*, D_p) for which we still need more trials. (*E.g.*, imagine we needed $\mathcal{GN}'_{\mathcal{AOG}}(D_c, G_C, \epsilon, \delta) = 25$ trials of the D_c retrieval, but only $\mathcal{GN}'_{\mathcal{AOG}}(D_p, G_C, \epsilon, \delta) = 15$ for D_p. Then $DP^{(2)}$ would test both D_c and D_p for the first 15 queries given by the oracle; afterwards, for the next 10 queries, it would only test D_c. Hence, it would perform a total of $(15 + 15) + 10 = 40$ retrievals.[14])

(Another variant involves the $DP^{(3)}$ procedure that, for each query, stops as soon as it knows the answer. *E.g.*, after seeing that the D_c retrieval succeeded for KB_C, $DP^{(3)}$ would return success and stop; by contrast, both $DP^{(1)}$ and $DP^{(2)}$ might continue exploring the graph, to determine whether D_p was blocked or not. Of course, $DP^{(3)}$ would have to vary its strategy over time, to make sure each retrieval gets enough samples — *i.e.*, to avoid having 1000 samplings of D_c but 0 samplings of D_p. Hence, the user could use this $DP^{(3)}$ system to answer his queries, efficiently, during PAO's "training mode". This process is discussed in detail in [GO91].)

[14]For comparison, observe that $DP^{(1)}$ would, instead, have performed $\max\{25, 15\} = 25$ trials of both D_c and D_p, for a total of 50 retrievals.

4.2 Other Classes of KBs

While many standard derivation systems do correspond to and-or hyper-trees, there are exceptions. We can use the GET-NUMBER$_1$ algorithm (defined above) to deal with such systems, provided the collection of $\{\mathcal{GN}_{\mathcal{G}}\}$ formulae includes these other classes of graphs.

The problem with most graphs that are more general than and-or trees is in computing the optimal strategy: It is NP-hard to compute the optimal strategy from a context whose structure is a DAG (rather than a tree) and probability vector [Sah74, Gre91b]. Hence, the FIND-OPTDS subroutine cannot simply find the optimal strategy for the graph and the frequency vector (as FIND-OPTDS$_0$ does, using $\Upsilon_{\mathcal{AOT}}$ for and-or trees).

Fortunately, there are efficient "approximation algorithms" for certain classes of such contexts. Each such algorithm, $\tilde{\Upsilon}_{\mathcal{G}}$, defined for a class of graphs $\mathcal{G}$, takes a graph $G \in \mathcal{G}$ and probability vector Q (whose i^{th} component is the success probability value of the i^{th} probabilistic arc in G), and returns a complete strategy whose cost is guaranteed to be no more than $\lambda = \lambda(G, Q)$ over the cost of the truly optimal strategy for this graph and probability vector. (Appendix B presents an example of such an algorithm for one class of graphs.) Here, we would first determine whether the given ϵ value is large enough; *i.e.*, if $\epsilon > \lambda$. If so, GET-NUMBER$_3$ could produce a number, $M' = \mathcal{GN}_{\mathcal{G}}(G, \epsilon - \lambda, \delta)$ that is "$\epsilon - \lambda$" accurate. (*I.e.*, consider the strategy obtained using the (intractable) $\Upsilon_{\mathcal{G}}$ algorithm on this graph and the success frequencies obtained from M' trials of each retrieval. Its cost is within $\epsilon - \lambda$ of the cost of the optimal strategy, with probability $\geq 1 - \delta$.) The FIND-OPTDS$_3$ algorithm would hand these observed frequencies, together with the context, to the approximation algorithm $\tilde{\Upsilon}_{\mathcal{G}}$. With probability $\geq 1 - \delta$, the cost of the resulting $\Theta_{pao} = \tilde{\Upsilon}_{\mathcal{G}}(G, \hat{P})$ will be within $(\epsilon - \lambda) + \lambda = \epsilon$ of optimal, as desired.

4.3 Handling Other Situations

Range of Queries: So far, we have insisted that each of the queries (*i.e.*, each $q \in \mathcal{Q}$) is an instantiation of the root node of the inference tree. We can, however, allow different queries to correspond to different nodes — *e.g.*, allow some queries to be instantiations of the `BuyCar(`κ`)` node and others, of `Pretty(`κ`)`, etc.

We must define the expected cost of each strategy appropriately: For each strategy Θ and each node n in the inference graph, we first define $E^n[\Theta]$ to be the expected cost of using Θ to solve the queries that are instantiations of n; this is obviously relative to this distribution of queries.[15] We can then define Θ's overall expected cost to be the weighted average of these "relative" expect costs, weighted by their respective frequencies of occurrence.

Probabilistic Intermediate Arcs: So far, the only probabilistic arcs we have considered correspond to database retrievals (*e.g.*, to D_c, corresponding to `Cheap(`κ`)`). In general, however, arcs based on rules can also be "probabilistic". As an example, consider the arc based on the rule R_{mm} : `Man(`x`)` $\Rightarrow$ `Status(`x`,Mortal)`, that descends from the `Status(` κ`, Mortal )` node to `Man(` κ `)`. Notice this arc is blocked if the query is `Status(` κ`, Immortal )`, as this rule would not apply in that situation. ([GO91] discusses solutions to some of the subtle problems that arise from the precedence constraints among the experiments.)

Different Fact Sets for Different Queries: So far, we have assumed that the set of facts associated with the knowledge base is invariant over time. We can, however, allow the database of ground atomic propositions to vary from query to query. (*N.b.*, we are *not* changing the rules that form the inference graph; *e.g.*, R_{bc} and R_{bp} are kept.) For example, the first query (*e.g.*, `BuyCar(C1)`) can be in the context of a knowledge base that contains the two facts { `Cheap(C1)`, `Pretty(C2)` }; the second query (*e.g.*, `BuyCar(C8)`), in the context of the single fact {`Pretty(C8)`}; the third query, (*e.g.*, `BuyCar(C8)`), in the first context; etc etc etc.

In general, we can consider the input to each DP be a a pair $\langle q, DB\rangle$, where q is a query corresponding to a node of G and DB is a database of ground atomic propositions. Notice each such pair specifies which probabilistic arcs are blocked. All of the analyses shown above will hold, provided there is some fixed probability that each particular arc will be blocked, over the distribution of such pairs, and these events are independent.

[15] Of course, this DP_Θ will only use the portions of Θ that is relevant to this query — *i.e.*, that corresponds to the subgraph that is "under" n.

Not Just Derivation Systems: While this work is worded in terms of finding the probably approximately optimal *derivation strategy*, this analysis also applies to any situation that involves finding a single satisficing answer within a search space. In general, this analysis pertains to any directed hyper-tree where [1] the task involves reducing the root down to some subset of the leaf-nodes; [2] various hyper-arcs can be blocked, with some unknown probability; and [3] each arc has some fixed cost.

Hence, this analysis can be used to deal with a set of Operators that can be used to reduce a Goal down to a set of Primitive Actions, each of which might or might not succeed, etc.

5 Conclusion

The results presented in this paper borrow from, and extend, various other lines of research. It takes its framework, viewing a derivation process as a graph search, from the work on effective use of control information ([SG85], [TG87], [Smi89], etc.). Those works (like the work on satisficing search strategies from which they arose [Gar73, SK75, Bar84, Nat86, Gre91a]) require that the user supply specific probability values for the probabilistic arcs, or use methods for computing these values that are based on unlikely assumptions. Our work fills this gap, by providing an effective, efficient technique for finding good estimates of these values. In addition, most of their techniques insist on finding the "optimal" solution; this becomes problematic when that task is intractable. Our work implements the obvious way of sidestepping this limitation, by allowing near-optimal solutions.

Our approach also resembles the work on EBL ("explanation-based learning") [DeJ88], as it uses previous solutions to suggest an improved derivation system. Most EBL systems use only a single example to suggest a new strategy; we extend those works by showing how to use a *set* of samples and by describing, further, the exact number of samples required. While most EBL systems are purely heuristic in nature, we use techniques from mathematical statistics to guarantee that our new strategies will (usually) be close to optimal.

Finally, this work borrows those statistical methods (as well as its title) from the field of "probably approximately correct learning" [Val84]. We provide a concrete application of these theoretical techniques.

To recapitulate: this work describes a technique for improving the efficiency of a derivation system. This process, PAO, uses a computed number of examples of the derivation process's behavior to determine which derivation paths are likely to succeed, then uses this information to identify a good strategy: one whose expected cost will be arbitrarily close to optimal, with

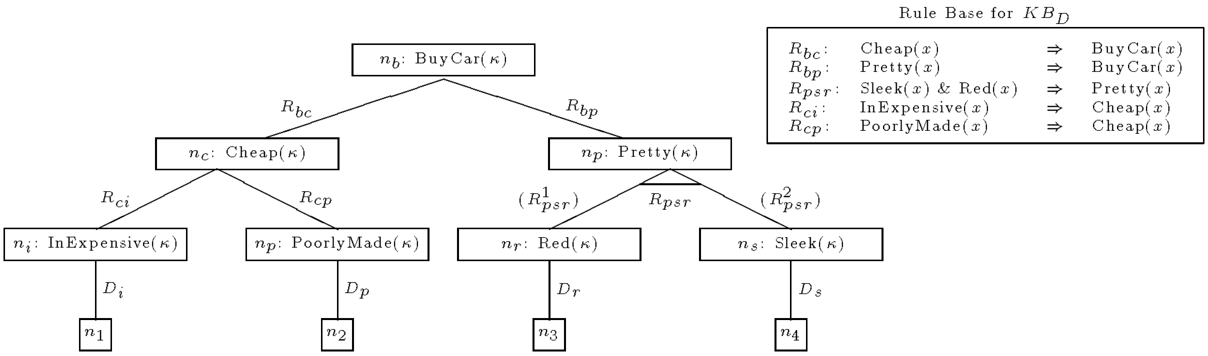

Figure 2: And-Or Inference Graph G_D

arbitrarily high probability. Within this framework, it completely solves several important, general classes of knowledge bases, including those whose rules form and-or hyper-trees.

A And-Or Hyper-Tree Inference Graphs

This paper has focused on disjunctive KBs for pedagogical reasons, as they are much easier to describe. This appendix uses an example of a non-disjunctive definite clause KB to illustrate a non-degenerate hyper-tree, and a strategy for this structure.

Consider the KB_D knowledge base, that includes the rules shown in Figure 2. As above, we can arrange the rules into an inference graph, $G_D = \langle N_D, A_D, S_D, B_D, c_D \rangle$. Each $n \in N_D$ refers to an atomic proposition to be proven. Here, $A_D \subset N_D \times 2^{N_D}$ is a set of *directed hyper-arcs*; meaning an arc can lead from one node to a *set* of children nodes, where these nodes, conjunctively, imply their common parent. As an example, the R_{psr} arc connects n_p to $\{n_r, n_s\}$. Each member $s \in S$ is a set of nodes; *i.e.*, $S_D \subset 2^{N_D}$. (*E.g.*, S_D is $\{ \{n_1\}, \{n_2\}, \{n_3, n_4\} \}$.) A derivation process is successful if it reaches each member of some $s \in S$.

As above, a *strategy* specifies the order in which to consider the various possible arcs. These strategies are much more complicated to describe than the ones for disjunctive KBs (*e.g.*, KB_C). There, a strategy could terminate, successfully, after performing a single successful database retrieval. This is not necessarily true here — *i.e.*, we may have to perform several other successful retrievals before returning `Yes`. (For example, it is not sufficient for the D_r retrieval alone to succeed; the D_s retrieval must succeed as well.) Hence, each strategy must specify the action to take both when a retrieval succeeds, and when it fails.

The Θ_D "strategy tree" shown in Figure 3 is an example of a strategy for traversing the G_D graph. The

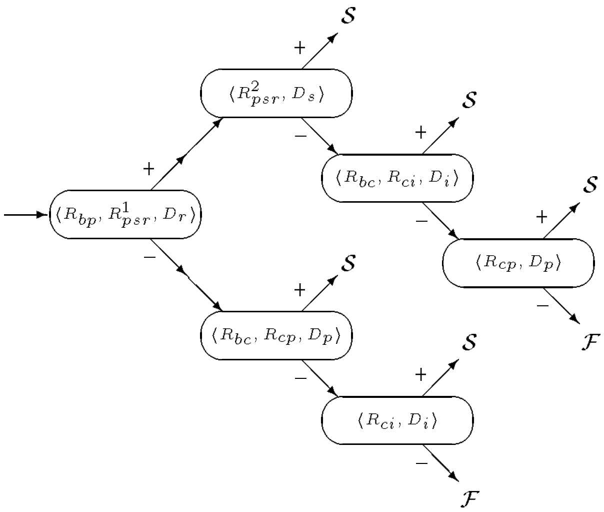

Figure 3: Strategy Tree Θ_D for the And-Or Graph G_D

derivation process that uses this strategy, DP_{Θ_D}, will first follow the path given in Θ_D's initial (*i.e.*, leftmost) node, $\langle R_{bp}, R^1_{psr}, D_r \rangle$. Notice the final step, D_r, is probabilistic; if it succeeds, then DP_{Θ_D} will follow the +-labeled arc to the $\langle R^2_{psr}, D_s \rangle$ node in the Θ_D strategy tree; otherwise, it will follow the −-labeled arc to $\langle R_{bc}, R_{cp}, D_p \rangle$ node. Assume that D_r was successful (*i.e.*, that the `Red(κ)` retrieval succeeded); DP_{Θ_D} then follows the instructions on the $\langle R^2_{psr}, D_s \rangle$ node, meaning it follows R^2_{psr} and attempts the D_s retrieval. If D_s succeeds, DP_{Θ_D} stops, returning success. (This is indicated by the $\mathcal{S}$ node, at the end of +-labeled arc ascending from Θ_D's $\langle R^2_{psr}, D_s \rangle$ node.) If not, DP_{Θ_D} follows the −-labeled arc to $\langle R_{bc}, R_{ci}, D_i \rangle$, and so pursues these steps in the G_D tree. Etc etc etc. (See [GO91] for further details of this process.)

We can compute the expected cost of a strategy, given the c cost function and the success probability of each probabilistic arc. We define it recursively, in terms of the cost of the strategy subtree whose root is some node, m. If m is a leaf node (*i.e.*, either $\mathcal{S}$ for "success" or $\mathcal{F}$ for "failure"), then the cost of m is 0. Otherwise, let m's arc sequence be $\langle a_1, \ldots, a_q \rangle$, where a_q is a probabilistic arc, with success probability $Pr[a_q] = p_q$.

(Notice these a_is are arcs in the inference graph, not in the strategy tree.) Let m^+ (resp., m^-) be the node in the strategy tree at the end of the +-labeled (resp., −-labeled) arc from m. The cost of a (sub)strategy rooted in m is the sum of the costs of the arcs in the m node (*i.e.*, $\sum_{i=1}^{q} c(a_i)$) plus p_q times the cost of the sub-strategy headed by m^+, plus $1-p_q$ times the cost of the sub-strategy headed by m^-.

(Notice: we need to use this same type of "strategy tree" structure even when considering simple (*i.e.*, non-hyper) inference trees, when intermediate arcs are probabilistic.)

B Efficient Algorithm for finding an Approximately Optimal Strategy

[Gre91b] proves that the task of finding a minimal cost strategy for an arbitrary graph is NP-hard. This reduction proof uses the particular class of DAGs suggested by Figure 4. This appendix provides an efficient algorithm that produces an approximately optimal strategy for this class of structures.

We consider the class of simple 4-deep non-hyper graphs, that have only a single "layer" of nodes with multiple parents — the $\{\mathtt{B}_j\}$ nodes. Let $c_b \in \Re^+$ be the cost of each $\vec{\mathtt{A}_i\mathtt{B}_j}$ arc[16]; $c_s \in \Re^+$ be the cost of each $\vec{\mathtt{B}_j\mathtt{S}_j}$ arc; and $c(i) \in \Re^+$ be the cost of each $\vec{\mathtt{G}\mathtt{A}_i}$ arc. The only probabilistic arcs are the ones from $\mathtt{B}_j$ to the associated $\mathtt{S}_j$ success nodes; let p_j be the success probability of the $\vec{\mathtt{B}_j\mathtt{S}_j}$ arc.

Notice first that there is an obvious poly time algorithm for computing the optimal strategy if any $p_j = 1$, as the optimal strategy here is simply the minimal cost path from $\mathtt{G}(\kappa)$ to the associated $\mathtt{S}_j$ success node. (If there are more than one such p_j, we use the smallest of these values.) We can, furthermore, ignore any $\mathtt{S}_j$ if its corresponding $p_j = 0$. We can, therefore, deal only with the case where $0 < p_j < 1$ holds for j.

[Gre91b] proves, in general, that the optimal strategy for any DAG is an ordering of the arcs in some embedded tree. It also proves that, for any strategy Θ,

$$\rho \cdot C_T(T[\Theta]) \;\leq\; E[\Theta] \;\leq\; C_T(T[\Theta])$$

where $T[\Theta]$ is the tree associated with the strategy Θ (*i.e.*, the tree formed by the arcs used by Θ); $C_T(T)$ (read "the *tree cost* of the tree T") is the sum of the costs of T's arcs; and $\rho = \prod_j (1-p_j)$ is the product of the failure probabilities of Θ's probabilistic arcs.

This allows us to bound the cost of any strategy: $E[\Theta] \;\geq\; \rho \cdot C_T(T_{opt})$, where T_{opt} is the minimal cost embedded tree. We can use this as a bound for the

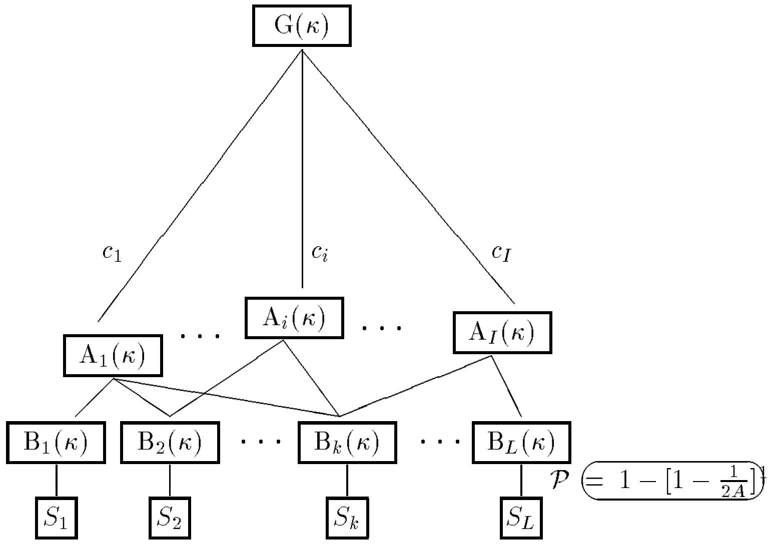

Figure 4: Class of Inference Graphs used in [Gre91b]

cost of the optimal strategy Θ_{opt};[17] *i.e.*,

$$C_T(T_{opt}) \;\leq\; \frac{E[\Theta_{opt}]}{\rho}$$

Now consider the following algorithm: $\tilde{\Upsilon}_{\mathcal{DAG}}(G,\; P)$

Step 1: Let $K^* \;=\;$ GREEDY(G) be a subset of $\mathcal{A} = \{\mathtt{A}_i\}$

Step 2: Let T^* be the tree built using K: T^*'s nodes are the elements of K, G and all of the $\{\mathtt{B}_j\}$ and $\{\mathtt{S}_j\}$ nodes. T^*'s arcs are the $\vec{\mathtt{G}\mathtt{A}_i}$ arcs for each $\mathtt{A}_i \in K^*$; one $\vec{\mathtt{A}_i\mathtt{B}_j}$ arc for each $\mathtt{B}_j$; and all of the $\vec{\mathtt{B}_j\mathtt{S}_j}$ arcs.

Step 3: Let Θ^* be the optimal strategy for T^*, found using [Smi89]'s polynomial algorithm.

This uses the GREEDY algorithm shown below, (a variant of Chvatal's algorithm for finding an approximately minimal set covering [Chv79]) to find a good enough subset, $K^* \subset \mathcal{A}$. Notice K^* must "cover" the entire $\{\mathtt{B}_j\}$ set — *i.e.*, for each $\mathtt{B}_j$, K^* must include at least one $\mathtt{A}_i$ such that $\vec{\mathtt{A}_i\mathtt{B}_j}$ is in the original DAG.

GREEDY(G)

Step 0: Set $K^* = \{\}$
For each $\mathtt{A}_i$, let
$R_i \;=\; R(\mathtt{A}_i) = \{\mathtt{B}_j \,|\, \mathtt{A}_i$ leads to $\mathtt{B}_j\}$.

Step 1: If $R_j \;=\; \{\}$ for each j, then stop, returning K^*.
Otherwise, find subscript k maximizing the ratio $|R_i|/c_i$.

Step 2: Add k to K^*, replace each R_j by $R_j - R_i$ and return to Step 1.

Let m be the largest value of $|R(\mathtt{A}_i)|$, over all $\mathtt{A}_i$ nodes. From [Chv79], we see that this algorithm returns a tree

[16] We let "$\vec{ab}$" refer to the arc from node a to node b.

[17] Notice we are *not* claiming that $T[\Theta_{opt}] = T_{opt}$.

whose tree cost is within a factor of

$$H(m) \stackrel{def}{=} \sum_{i=1}^{m} \frac{1}{j} \quad = \quad O(\ln m)$$

of the cost of the optimal subtree.[18] We prove below that this means the cost of $\tilde{\Upsilon}_{\mathcal{DAG}}(G,\ P)$'s answer, Θ^*, is within the additive constant $\lambda = \left(\frac{H(m)}{\rho} - 1\right) E[\Theta^*]$ over the cost of the optimal strategy. While we can efficiently compute the value of $E[\Theta^*]$, we may prefer to use the (easier to compute) larger bound $\lambda = \left(\frac{H(m)}{\rho} - 1\right) C$, where $C = L(c_b + c_s) + \sum_i c(i)$ is the sum of the costs of all of the arcs in the complete DAG.

Proof: Let Θ_{opt} be the optimal strategy for this DAG; T_{opt} be the minimal cost embedded subtree; and Θ^* be the strategy returned by $\tilde{\Upsilon}_{\mathcal{DAG}}(G,\ P)$, based on the $T^* = T[\Theta^*]$ subtree.

$$\begin{aligned}
E[\Theta^*] &\leq C_T(T[\Theta^*]) \\
&\leq H(m) \cdot C_T(T_{opt}) \\
&\leq H(m) \cdot \frac{E[\Theta_{opt}]}{\rho} \\
&\leq E[\Theta_{opt}] \quad + \quad \left(\frac{H(m)}{\rho} - 1\right) E[\Theta_{opt}] \\
&\leq E[\Theta_{opt}] \quad + \quad \left(\frac{H(m)}{\rho} - 1\right) E[\Theta^*]
\end{aligned}$$

The final step uses the fact that $E[\Theta_{opt}] \leq E[\Theta]$ for any strategy Θ, and in particular, for Θ^*. $\Box$.

Acknowledgements

This work was supported by an Operating Grant from Canada's Natural Science and Engineering Research Council to the first author, and by the Academy of Finland to the second. We also greatly appreciate the comments we received from William Cohen, Gerhard Lakemeyer, Dave Mitchell, Dale Schuurmans and Bart Selman.

References

[Bar84] J. A. Barnett. How much is control knowledge worth?: A primitive example. *Artificial Intelligence: An International Journal*, 22:77–89, 1984.

[Bol85] B. Bollobás. *Random Graphs*. Academic Press, 1985.

[Chv79] V. Chvatal. A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, 4:233–35, 1979.

[DeJ88] Gerald DeJong. AAAI workshop on Explanation-Based Learning. Sponsored by AAAI, 1988.

[DM86] Gerald DeJong and Raymond Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–76, 1986.

[Gar73] M. R. Garey. Optimal task sequencing with precedence constraints. *Discrete Mathematics*, 4, 1973.

[GN87] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[GO91] Russell Greiner and Pekka Orponen. Probably approximately optimal satisficing strategies. Technical report, University of Toronto, 1991.

[Gol79] A. Goldberg. An average case complexity analysis of the satisfiability problem. In *Proceedings of the 4th Workshop on Automated Deduction*, pages 1–6, Austin, TX, 1979.

[Gre91a] Russell Greiner. Finding an optimal satisficing strategy in an and-or tree. Technical report, University of Toronto, forthcoming 1991.

[Gre91b] Russell Greiner. Finding the optimal derivation strategy in a redundant knowledge base. *Artificial Intelligence: An International Journal*, forthcoming 1991.

[Kel87] Richard M. Keller. Defining operationality for explanation-based learning. In *AAAI-87*, pages 482–87, Seattle, July 1987.

[MCK+89] Steven Minton, Jaime Carbonell, C.A. Knoblock, D.R. Kuokka, Oren Etzioni, and Y. Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence: An International Journal*, 40(1-3):63–119, September 1989.

[Min88] Steven Minton. Quantitative results concerning the utility of explanation-based learning. In *AAAI-88*, pages 564–69, San Mateo, CA, August 1988. Morgan Kaufmann Publishers, Inc.

[MKKC86] Thomas M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Example-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[Nat86] K. S. Natarajan. Optimizing depth-first search of AND-OR trees. Technical report, Research report RC-11842, IBM T. J. Watson Research Center, January 1986.

[OG90] Pekka Orponen and Russell Greiner. On the sample complexity of finding good search strategies. In *Proceedings of COLT-90*, pages 352–58, Rochester, August 1990.

[18] We can actually get a tighter bound: $C_T(T^*) - C_{bs} \leq H(m)[C_T(T_{opt}) - C_{bs}]$, where $C_{bs} = L(c_b + c_s)$ is the cost of the tree from the set of $\mathtt{A}_i$s down to the leaf $\mathtt{S}_j$s.

[Sah74] S. Sahni. Computationally related problems. *SIAM Journal on Computing*, 3(4):262–279, 1974.

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

[SK75] H. A. Simon and J. B. Kadane. Optimal problem-solving search: All-or-none solutions. *Artificial Intelligence: An International Journal*, 6:235–247, 1975.

[Smi89] David E. Smith. Controlling backward inference. *Artificial Intelligence: An International Journal*, 39(2):145–208, June 1989.

[TG87] Richard J. Treitel and Michael R. Genesereth. Choosing orders for rules. *Journal of Automated Reasoning*, 3(4):395–432, December 1987.

[Val84] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–42, 1984.