Appears in the
Proceedings of the Ninth Canadian Conference on Artificial Intelligence (CSCSI-92),
Vancouver, May 1992.

A Formal Analysis of Solution Caching

Vinay K. Chaudhri*
Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4
vinay@ai.toronto.edu

Russell Greiner†
Siemens Corporate Research
Princeton, NJ 08540
greiner@learning.siemens.com

Abstract

Many inference management systems store and maintain the conclusions found during a derivation process in a form that allows these conclusions to be used during subsequent derivations. As this approach, called "solution caching", allows the system to avoid repeating these derivations, it can reduce the system's overall cost for answering queries. Unfortunately, as there is a cost for storing these conclusions, it is not always desirable to cache every solution found — this depends on whether the savings achieved by performing fewer inference steps for these future queries exceeds the storage overhead incurred. This paper formally characterizes this tradeoff and presents an efficient algorithm, FOCL, that produces an optimal caching strategy: *i.e.*, given an inference graph of a knowledge base, anticipated frequencies of queries and updates of each node in this graph, and various implementation-dependent cost parameters, FOCL determines which of these nodes should cache their solutions to produce a system whose overall cost is minimal. The paper also presents empirical results that indicate that a system based on FOCL can significantly outperform one based on any of the standard approaches to solution caching.

1 Introduction

A "solution caching"[1] system will store and maintain the conclusions found during a derivation process, in a form that allows the system to simply retrieve and reuse these stored solutions to answer subsequent queries. As this avoids the cost of repeating these derivations, it can significantly improve the response time for repeated queries. These savings are especially important in very large and complex knowledge bases and in applications where the response time is critical, such as real time process control. As caching does incur the cost of storing the derived conclusions, it may not be useful when the storage cost is very high or when the queries are not repeated a large number of times.

As an example, consider the knowledge base, KB_1, shown in Figure 1.[2] If we ask for all `living` objects (*i.e.*, find all `X` satisfying the `living(X)` query), the inference engine will backward chain, traversing the inference graph down to the ground facts. Here, the solutions are:

$$\Delta \;=\; \left\{ \begin{array}{lll} \text{living(slime2)} & \text{living(george)} & \text{living(john)} \\ \text{living(fred)} & \text{living(fido)} & \text{living(roxy)} \\ \text{living(apple1)} & \text{living(orange3)} & \text{living(bean7)} \end{array} \right\} \qquad (1)$$

An inference management system IMS[3] that can cache its answers could then store these derived solutions, in effect forming a larger knowledge base $KB_1' \leftarrow KB_1 \cup \Delta$ that includes all of KB_1 as well as these nine propositions, Δ. If the same query `living(X)` is posed again, this IMS will find these solutions by a single lookup rather than by re-deriving them by backward chaining. Notice the IMS is spending additional time in processing the initial query to store this information, in the hope that it will save time later when addressing this same query for the second and subsequent times. Notice, however, that these cached entries can become "dirty" if the fact set is changed — *e.g.*, if we delete the literal `man(john)`, or add a new fact `dog(shep)`. Hence, we must consider how often each type of fact is updated

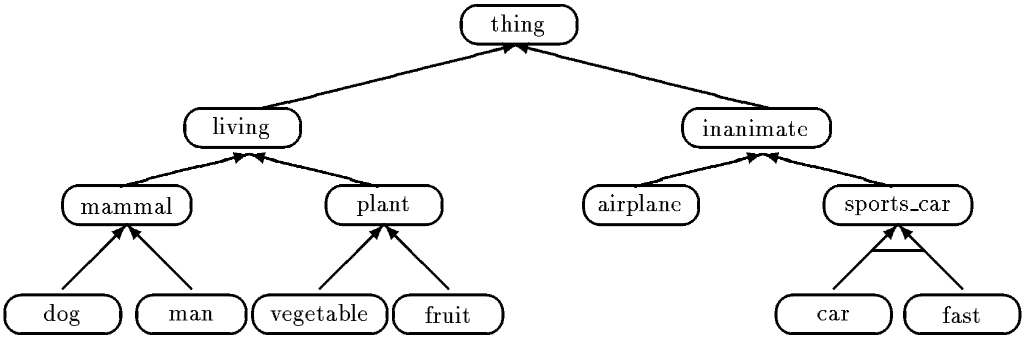
*Supported by the Department of Computer Science at the University of Toronto and by the Information Technology Research Centre of Ontario. Both authors gratefully acknowledge the help from Professors Charles Elkan and M. W. Carter, as well as from Tom Petsche, Wen-Ling Hsu and the anonymous referees.

†Much of this work was performed at the University of Toronto where it was supported by the Institute for Robotics and Intelligent Systems and by an operating grant from the National Science and Engineering Research Council of Canada.

[1] We will use the term "caching" as a shorthand for this form of "solution caching". *N.b.*, it does *not* refer to the technique of moving information from secondary to primary storage.

[2] Following Prolog conventions, names that begin with a capital letter (*e.g.*, "`X`") are variables. Unbound variables that appear in assertions are assumed universally quantified; those in queries, existentially quantified.

[3] This IMS can be a knowledge representation system, a deductive database, a logic programming system, an object-oriented system, etc.



Rule Base

```
thing(X)     :- living(X).    thing(X)     :- inanimate(X).
living(X)    :- mammal(X).    living(X)    :- plant(X).
mammal(X)    :- dog(X).       mammal(X)    :- man(X).
plant(X)     :- vegetable(X). plant(X)     :- vegetable(X).
inanimate(X) :- vehicle(X).   inanimate(X) :- airplane(X).
sports_car(X) :- car(X), fast(X).
```

Fact Set

```
living(slime2)    mammal(george)
man(john)         man(fred)
dog(fido)         dog(roxy)
fruit(apple1)     vegetable(bean7)
fruit(orange3)    airplane(p17)
```

Figure 1: Knowledge Base KB_1 — Ground facts + Rules

- Structure of the inference graph
- Distribution of queries and updates
- Costs of each inference step, fact-set retrieval, ...
- Incremental cost of additional storage
- Number of solutions to be cached (at each node)

Table 1: Factors Affecting Caching Performance

and the cost of propagating this change to insure the cached information remains accurate.

Our goal is to use solution caching as a way of producing an efficient IMS — one that requires a minimal amount of time to deal with the anticipated distribution of queries and updates.

There are two standard ways of dealing with solution caching. Many systems, including PROLOG [CM81], never cache any solutions. Others (*e.g.*, [Mos83]) will cache every solution found — *e.g.*, at every node in the graph shown in Figure 1. This paper shows that neither of these two simple approaches leads to an optimally efficient system, and so proposes a third approach: of selectively caching only at certain nodes. Section 2 first develops a quantitative model that formalizes the interaction amongst the different factors affecting the caching performance, summarized in Table 1. Section 3 then uses this model to define an algorithm, FOCL (for "Find Optimal Cache Label") that determines which literals should cache their solutions. Section 4 then presents a set of performance experiments to demonstrate that the optimal caching scheme produced by FOCL can significantly outperform the systems that use either obvious approach, of caching everywhere or not caching anywhere.

Due to space restrictions, this short article cannot provide a comprehensive survey of the related research; instead, we refer the interested reader to our extended paper [CG92]. That paper also discusses how our results can be used by a wide variety of systems, including knowledge representation, deductive databases, logic programming and object-oriented systems; and presents both the relevant proofs and a more comprehensive discussion of the FOCL algorithm.

2 Framework and Cost Model

This work deals with definite clause knowledge bases (*i.e.*, a set of clauses, where each clause has exactly one positive literal); we call each ground atomic literal a "fact", and each non-atomic clause, a "rule". We can arrange the rules into an inference graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{A} \rangle$, where each node $n \in \mathcal{N}$ corresponds to an atomic literal, and each hyper-arc $a \in \mathcal{A}$ corresponds to a rule, leading from (the nodes representing) its set of antecedents to (the node representing) its conclusion; see, *e.g.*, Figure 1.[4] As discussed above, given a query (*e.g.*, "`living(Y)`"), the IMS will search through the graph seeking all solutions, both those corresponding to an immediate database retrieval (which finds `living(slime2)`) as well as the solutions found by backward chaining — *i.e.*, following the various rules to their subgoals, to obtain the other eight solutions shown in Equation 1. The IMS will then return all of these answers.

The IMS may also decide to "cache" these solutions — *e.g.*, store all of the Δ set associated with the `living(X)` node. It may also store the solutions found at any intermediate node — *e.g.*, store the facts {`mammal(george)`, `mammal(john)`, `mammal(fred)`, `mammal(fido)`, `mammal(roxy)`} associated with the `mammal(X)` node, or the facts {`plant(bean7)`, `plant(apple1)`, `plant(orange3)`} associated with the `plant(X)` node, etc.

Our IMS has the option of caching at any of the nodes in the graph — if so, it will store *all* of the derived solutions associated with each selected node. As an example,

[4] We will often identify each node $n \in \mathcal{N}$ with its associated literal.

the first time the IMS encounters the `living(X)` query (perhaps as a subquery of the "higher" `thing(X)` query) it will compute the bindings shown in Equation 1. If it has decided to cache at this node, it will then store the subset of these solutions that are not already explicitly present — that is, all but `living(slime2)`. The second, and subsequent times IMS encounters this `living(X)` query, it will simply retrieve all nine solutions (the eight newly stored values, and the one originally stored) and so will not need to backward chain.

The retrieval time required to find these answers is clearly much less when these answer have been cached. There is, however, a cost to storing these solutions initially, and there is also an additional cost each time there is an update to any of the relations used in any of `living(X)`'s "children" — *i.e.*, if we add `dog(shep)`, etc. Hence, it is not obvious whether we should cache at any of the nodes.

We can formally define our task in terms of the following definition:

Defn#1. A Caching Label of an inference graph is a function which assigns a label of either "C^+" or "C^-" to each node in the graph. The label "C^+" (resp., "C^-") means that solutions should (resp., should not) be cached at this node.
We let $\mathcal{LL}(\mathcal{G})$ refer to the set of all labels for the inference graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{A} \rangle$ — *i.e.*,

$$\mathcal{LL}(\langle \mathcal{N}, \mathcal{A} \rangle) \overset{def}{=} \{ \mathcal{L}_\ell \,|\, \mathcal{L}_\ell \colon \mathcal{N} \mapsto \{C^+, C^-\} \};$$

and let $\mathcal{L}_\ell[n_k]$ refer to the value the labelling $\mathcal{L}_\ell \in \mathcal{LL}(\mathcal{G})$ assigns to the node $n_k \in \mathcal{N}$.

The Overall Cost of a caching label $\mathcal{L}_\ell$, written $E[\mathcal{L}_\ell]$, is the total cost required to perform all anticipated queries at all nodes, assuming the IMS uses the labelling $\mathcal{L}_\ell$. This value is the sum of the costs associated with each node in the graph, and includes the costs of performing all retrieving inferencing, storing and updating steps. (The particular formula for these costs will depend on the type of inference graph and cost model involved; see the next subsection, especially Equation 3.)

Optimal Cache Labelling Task:
Instance: An inference graph $\mathcal{G} = \langle \mathcal{N}, \mathcal{A} \rangle$ with its space of cache labellings $\mathcal{LL}(\mathcal{G})$, and cost function $E \colon \mathcal{LL}(\mathcal{G}) \mapsto \Re$.

Problem: Identify a cache labelling $\mathcal{L}_* \in \mathcal{LL}(\mathcal{G})$ whose cost is minimal over all labellings — *i.e.*, such that

$$\forall \mathcal{L} \in \mathcal{LL}(\mathcal{G}) \;\; E[\mathcal{L}_*] \;\leq\; E[\mathcal{L}]. \qquad \Box$$

The rest of this subsection sketchs a particular concrete cost model (*i.e.*, a specific $E[\cdot]$ function) for a particular class of inference graphs; [CG92] provides a more exact specification.[5]

We assume that the total cost of a database retrieval varies linearly with the number of solutions obtained — *e.g.*, the cost of retrieving the 2 facts matching `man(Y)` is twice the cost of retrieving the 1 fact matching `vegetable(Z)` (*viz.*, `vegetable(bean7)`). In general, it will cost "nL" to retrieve the n facts matching a proposition, where L is a constant that is independent of the particular proposition considered; *i.e.*, independent of whether the query was `man(Y)` or `vegetable(Z)`.

We assume a uniform cost for caching any proposition; *i.e.*, it costs the same to cache `dog(fido)` as to store `between(a P b)`; call this value "S". It will also cost this same amount to perform any *update* to a cache — whether by adding or deleting a literal. We likewise assume a uniform cost "R" for reducing any goal along any one rule, to that rule's (appropriately instantiated) antecedents. For example, it costs R to reduce `thing(X)` to `living(X)`; and costs the same R to reduce `sports_car(X)` to `car(X)` and `fast(X)`.

We assume, as given, the values of these parameters:

L – Cost of any one lookup, per unit fact found
R – Cost of reducing any one goal (along any one rule)
S – Cost of caching/adding/deleting any one fact

In addition, for each node n_i in the inference graph, we must know

Defn#2. $s(n_i)$ is the current number of propositions explicitly in the fact set that match the goal associated with this node. As an example based on the graph in Figure 2 (taken from the far left side of Figure 1's KB_1), $s(n_4)$ is the number of propositions in the fact set that match `dog(X)`. Assuming the associated fact set contains (all and only) the facts { `dog(fido)`, `dog(roxy)`, `mammal(lulu)` }, then $s(n_4) = 2$, $s(n_3) = 1$ and $s(n_1) = s(n_2) = 0$.

Defn#3. $d(n_i)$ is the number of direct queries posed at the node n_i. *E.g.*, we will ask the question `mammal(X)` a total of $d(n_3)$ times.

Defn#4. $u(n_i)$ is the rate of updates to the node n_i, where each update is either adding or deleting a literal to the extension of the node n_i. As an example, if we plan to add in two new literals — *e.g.*, `dog(shep)` and `dog(phydeau)` — and delete one literal `dog(fido)` over a period of time during which the number of queries was 100, then $u(n_4) = (2 + 1)/100 = 0.03$.

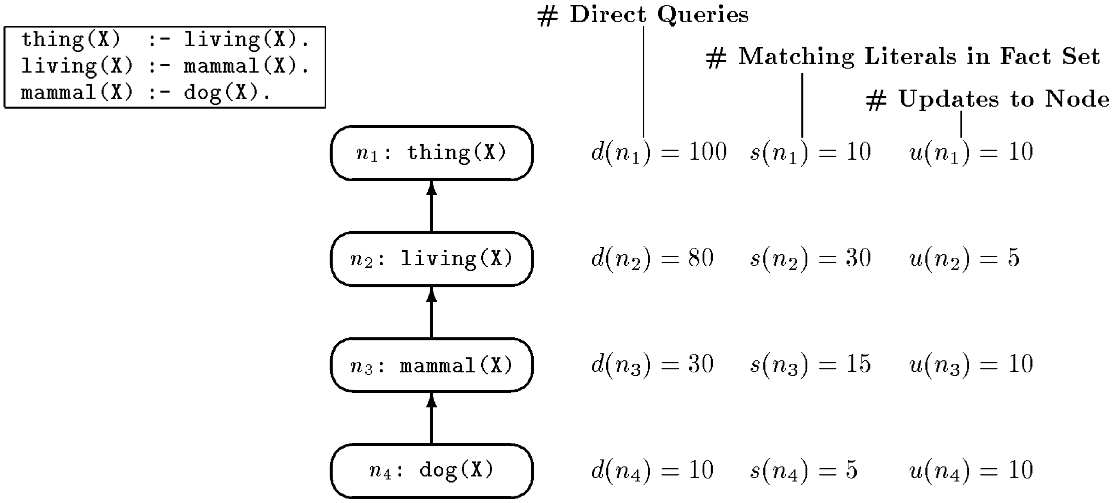
(The values of $d(\cdot)$ and $u(\cdot)$ are with respect to some interval of time; see Subsection 3.4. That subsection also discusses how to estimate the values of these parameters.)

Given the values of these implementation-dependent parameters, we can compute the cost $E[\mathcal{L}_\ell]$ of any given $\mathcal{L}_\ell \in \mathcal{LL}(\mathcal{G})$. We need the following terms:

Defn#5. Given any node $n \in \mathcal{N}$, let
$Ch(n) = \{ n_i \,|\, \langle n, n_i \rangle \in \mathcal{A} \}$ refer to n's immediate children; and
$\mathcal{U}(n)$ refer to the set of nodes in the graph strictly strictly "under" n, at any depth: *i.e.*,

$$\mathcal{U}(n) \;=\; Ch(n) \,\cup \bigcup_{n_i \in Ch(n)} \mathcal{U}(n_i).$$

[5] This paper uses a very simple model for purely pedagogical reasons; we are aware that sophisticated PROLOG systems require a much more elaborate model [Debray, personal communication]. The analysis in this paper *does* apply to those models as well; see [CG92].



thing(X) :- living(X).
living(X) :- mammal(X).
mammal(X) :- dog(X).
# Direct Queries
# Matching Literals in Fact Set
# Updates to Node
n_1: thing(X)
n_2: living(X)
n_3: mammal(X)
n_4: dog(X)
$d(n_1) = 100$ $s(n_1) = 10$ $u(n_1) = 10$
$d(n_2) = 80$ $s(n_2) = 30$ $u(n_2) = 5$
$d(n_3) = 30$ $s(n_3) = 15$ $u(n_3) = 10$
$d(n_4) = 10$ $s(n_4) = 5$ $u(n_4) = 10$

Figure 2: Knowledge Base KB_2; and Parameters

(Notice $n \notin \mathcal{U}(n)$; and if n is a leaf, then $Ch(n) = \mathcal{U}(n) = \{\}$.)

Defn#6. The Number of Indirect Queries at the node n_k with respect to a labelling $\mathcal{L}_\ell$, designated "$I_\ell(n_k)$", is the total number of queries that the user can ask at any of n_k's ancestor nodes and that will cause the inference process to retrieve values at n_k. Notice this value depends on the cache labelling.

If we cache at the parent node n_k, then the child n_{k+1} receives only one indirect query — only for the first derivation. (*E.g.*, if we cache at living(·), then its child mammal(·) will receive only a single indirect query.) If we do not cache, the number of indirect queries that n_{k+1} receives is the sum of direct and indirect queries at the parent. Hence,

$$I_\ell(n_{k+1}) \;=\; \begin{cases} I_\ell(n_k) + d(n_k) & \text{if } \mathcal{L}_\ell[n_k] = \mathrm{C}^- \\ 1 & \text{if } \mathcal{L}_\ell[n_k] = \mathrm{C}^+ \end{cases} \qquad (2)$$

As the root node n_1 does not have any parents, we have $I_\ell(n_1) = 0$ for every $\mathcal{L}_\ell$.

To illustrate this: using Figure 2, let $\mathcal{L}_{\langle ----\rangle}$ denote the labelling that does not cache at any node, and $\mathcal{L}_{\langle -+--\rangle}$, the labelling that caches only at the node n_2 and nowhere else. Then the number of indirect queries at n_3 is $I_{\langle ----\rangle}(n_3) = d(n_1)+d(n_2)$; and $I_{\langle -+--\rangle}(n_3) = 1$.

For any node $n \in \mathcal{N}$, define $E[\mathcal{L}_\ell,\, n]$ to be the cost of using the label $\mathcal{L}_\ell$ to deal with the nodes including and below n — *i.e.*, with the nodes $\{n\} \cup \mathcal{U}(n)$. Notice $E[\mathcal{L}_\ell] = E[\mathcal{L}_\ell, \langle\texttt{root}\rangle]$, where $\langle\texttt{root}\rangle$ is the root node (here n_1). The incremental cost of dealing with the node n, above the cost of its children, involves the expense of retrieving n's complete extension a total of $d(n) + I_\ell(n)$ times. If we cache, then we must add in the cost of caching the additional $I_\ell(n)$ answers after answering the query for the first time, and also the cost of returning these cached solutions during each subsequent retrieval. We also have the additional cost of the processing the subsequent updates. Hence,

$$E[\mathcal{L}_\ell, n] \;=\; \sum_{n_i \in Ch(n)} E[\mathcal{L}_\ell, n_i] \;+\; \begin{cases} [(L\cdot s(n)) + (R\cdot|Ch(n)|)](I_\ell(n) + d(n)) & \text{if } \mathcal{L}_\ell[n] = \mathrm{C}^- \\ [(L\cdot s(n)) + (R\cdot|Ch(n)|)] & \text{if } \mathcal{L}_\ell[n] = \mathrm{C}^+ \\ \quad + [d(n) + I_\ell(n) - 1]\cdot(L\cdot[s(n) + s(\mathcal{U}(n))]) & \\ \quad + S\cdot s(\mathcal{U}(n)) \;+\; S\cdot u(\mathcal{U}(n)) & \end{cases} \qquad (3)$$

where $s(\mathcal{U}(n)) = \sum_{n_i \in \mathcal{U}(n)} s(n_i)$ and $u(\mathcal{U}(n)) = \sum_{n_i \in \mathcal{U}(n)} u(n_i)$. Notice the value of $E[\mathcal{L}_\ell, n]$ depends on both the labels of the nodes *below* n (as it involves $E[\mathcal{L}_\ell, n_i]$ for each $n_i \in Ch(n)$), and the labels of the nodes *above* n (as it involves $I_\ell(n)$).

The precise characterization of the cost, shown in Equation 3, is one of the important contributions of our work. Notice it extends previous work (*e.g.*, [Sel89, SJGP90]) which assumes that this cost is given and is independent of the structure of the knowledge base.

3 Optimal Labelling Algorithm

For pedagogical reasons, Subsection 3.1 first describes the FOCL algorithm for a simple class of inference graphs; Subsections 3.2 and 3.3 then discuss how FOCL generalizes to cover other classes, enabling FOCL to handle any "tree structured" inference graph; *i.e.*, any graph that includes at most one directed path between any pair of nodes. (Figure 1 is an example.) FOCL depends on various input values; Subsection 3.4 discusses ways of obtaining or estimating these values.

3.1 Using FOCL for Linear KBs

This subsection deals only with the particular class of "linear knowledge bases", where each clause can have at most one negative literal and the conclusion of at most one rule can match any given proposition. (Hence, each "rule" can have only one antecedent, meaning there are no conjunctions; and any goal can be reduced to at most one subgoal, so $|Ch(n)| \leq 1$ for all nodes n.) The graph in Figure 2 suggests such a knowledge base.

$$\begin{array}{lcll} P(n_4,0) &=& \{\,\mathcal{L}_\ell \mid \mathcal{L}_\ell[n_1]=\mathrm{C}^- \ \&\ \mathcal{L}_\ell[n_2]=\mathrm{C}^- \ \&\ \mathcal{L}_\ell[n_3]=\mathrm{C}^-\,\} & \langle\, - - - ?\,\rangle \\ P(n_4,1) &=& \{\,\mathcal{L}_\ell \mid \mathcal{L}_\ell[n_1]=\mathrm{C}^+ \ \&\ \mathcal{L}_\ell[n_2]=\mathrm{C}^- \ \&\ \mathcal{L}_\ell[n_3]=\mathrm{C}^-\,\} & \langle\, + - - ?\,\rangle \\ P(n_4,2) &=& \{\,\mathcal{L}_\ell \mid \qquad\qquad\qquad\quad \mathcal{L}_\ell[n_2]=\mathrm{C}^+ \ \&\ \mathcal{L}_\ell[n_3]=\mathrm{C}^-\,\} & \langle\, ?\ + - ?\,\rangle \\ P(n_4,3) &=& \{\,\mathcal{L}_\ell \mid \qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathcal{L}_\ell[n_3]=\mathrm{C}^+\,\} & \langle\, ?\ ?\ + ?\,\rangle \end{array}$$

Figure 3: Values of $P(n_i,\,j)$

One naïve way to find the optimal labelling is to enumerate all of the possible labellings, compute the cost of each and select the one that is minimum. This approach is not computationally feasible even for this simple class of knowledge bases, as there are $2^{|\mathcal{N}|}$ labellings.

Another approach is to label each node one at a time, by traversing the entire inference graph in one direction — either top down or bottom up. Unfortunately, the decision of whether to cache at a node depends on the cost of answering all queries at that node, which in turn depends on the labels of both the ancestor and the descendant nodes: the total number of queries that reach a node depend on which of its ancestors are cached, and the cost of obtaining the complete extension of a node will depend on which of its descendants are cached. This rules out a single traversal in either direction, as either requires quantities that would not be available. Fortunately, however, we can capture this interdependence using a dynamic programming technique to obtain a solution that is provably optimal [Nem66].

The basic idea involves two traversals of the inference graph; see Figure 4. Equation 3 shows that the value of $E[\mathcal{L}_\ell,\, n_k]$ depends on $E[\mathcal{L}_\ell,\, n_{k+1}]$, $I_{\mathcal{L}_\ell}(n_k)$ and various input parameters. Fortunately, the values of $E[\mathcal{L}_\ell,\, n_{k+1}]$ and $I_\ell(n_k)$ can be decoupled: given any class of labellings that share a common $I_\ell(n_k)$ value, the best labelling will be the one with the smallest $E[\mathcal{L}_\ell,\, n_{k+1}]$ value.

FOCL's first pass [Figure 4's Line 1] works from the root down to the leaf node (here from n_1 to n_4), partitioning the set of possible labellings into equivalence classes that share a common value of $I_\ell(n_k)$. That is, define

$$\begin{array}{rl} P(n_k,0) = & \{\ \mathcal{L}_\ell \in \mathcal{LL}(\mathcal{G}) \mid \forall\, 0<j<k.\ \mathcal{L}_\ell[n_j]=\mathrm{C}^-\ \} \\ P(n_k,i)\ = & \{\ \mathcal{L}_\ell \in \mathcal{LL}(\mathcal{G}) \mid \forall\, i<j<k.\ \mathcal{L}_\ell[n_j]=\mathrm{C}^- \ \& \\ & \qquad \mathcal{L}_\ell[n_i]=\mathrm{C}^+\ \} \qquad \text{for } i=1..(k-1) \end{array} \quad (4)$$

Notice $P(n_i,j)$ is a set of labels. As an example, $P(n_1,0)=\mathcal{LL}(\mathcal{G})$ is the set of all labellings.

Figure 3 describes the values of $P(n_4,j)$ for the allowed values of j. Its right side encodes each $P(n_i,j)$ as a sequence of the form $\langle \pm_1, \pm_2, \ldots \pm_k\rangle$ where, for each $\mathcal{L}_\ell \in P(n_i,j)$, $\pm_m = +$ means $\mathcal{L}_\ell[n_m]=\mathrm{C}^+$, $\pm_m = -$ means $\mathcal{L}_\ell[n_m]=\mathrm{C}^-$, and $\pm_m = ?$ means $\mathcal{L}_\ell[n_m]$ is arbitrary. In general,

$$P(n_i,j) \quad \equiv \quad \langle\ \underbrace{?\ ?\ \cdots\ ?}_{\text{irrelevant}}\ \ \overset{\overset{i^{th}}{\downarrow}}{+}\ \ \underbrace{-\ -\ \cdots\ -}_{\text{all }-\text{'s}}\ \ \overset{\overset{j^{th}}{\downarrow}}{\underbrace{?\ ?\ \cdots\ ?}_{\text{irrelevant}}}\ \rangle$$

for $i=1..(j-1)$. Notice $P(n_i,j)$ does not restrict labels on the basis of their values for nodes n_m where $m<i$ or $m \geq j$.

The k sets $\{P(n_k,i)\}_{i=0}^{k-1}$ partition the set of all labellings. By construction, the value of $I_\ell(n_k)$ is the same

```
Algorithm FOCL($\mathcal{G}$, $\{d(n_i),\, s(n_i),\, u(n_i)\}_i$, $R, L, S$)
  1. For each $n_k$ := $\langle$root$\rangle$..$\langle$leaf$\rangle$
       Compute  $\{V(n_k,i)\}_{i=0}^{l-1}$  based on Equation 4, ...
  2. For each $n_k$ := $\langle$leaf$\rangle$..$\langle$root$\rangle$
       Compute  $\{M(n_k,i)\}_{i=0}^{k-1}$  based on Equation 5, ...
  3. Return the optimal labelling, based on the decisions
     made in determining $M(\langle$root$\rangle, 0)$ ...
```

Figure 4: Code for FOCL

for each label $\mathcal{L}_\ell \in P(n_k,i)$; call this value $V(n_k,i)$. (Here, $V(n_1,0)=1$, $V(n_2,0)=100$, $V(n_2,1)=1$, ..., $V(n_3,0)=180$, $V(n_3,1)=81$, $V(n_3,2)=180$, etc.) Observe that FOCL can compute these values efficiently using a single top-down pass as the values of $V(n_k,i)$ can be computed based on the values of $\{V(n_{k-1},j)\}_j$. *N.b.,* FOCL *will only deal with these* $V(n_i,j)$ *values; it never needs to explicitly construct the* $P(n_k,j)$ *sets.*

FOCL's second pass [Figure 4's Line 2] works from the bottom up (here from n_4 up to n_1). At each stage, when dealing with n_k, FOCL determines the labellings in each equivalence class that are best from "here down": that is, it computes $M(n_k,i)$, defined to be the smallest value of $E[\mathcal{L}_\ell,\, n_k]$ over all labelling $\mathcal{L}_\ell \in P(n_k,i)$; *i.e.*,

$$M(\,n_k,\, i) \quad \overset{def}{=} \quad \min\{\ E[\mathcal{L}_\ell,\, n_k] \mid \mathcal{L}_\ell \in P(n_k,i)\ \} \quad (5)$$

Working bottom up, FOCL will know the values of $\{M(n_{k+1},j)\}_j$ when dealing with n_k. It can use the appropriate value from this set in the role of "$E[\mathcal{L}_\ell,\, n_{k+1}]$", together with the value $V(n_k,i)$ for "$I_\ell(n_k)$" in Equation 3, and then compute the two candidate values for $M[\mathcal{L}_\ell,\, n_k]$: one based on mapping n_k to C^+, and the other to C^-. FOCL sets $M(n_k,j)$ to be the smaller of these two values. It can then use this information to compute $M(n_{k-1},i)$, and so on. On reaching the root node, FOCL explicitly has the value of $M(\langle\texttt{root}\rangle,0)$, which by construction is the minimal value of $E[\mathcal{L}_\ell] = E[\mathcal{L}_\ell,\, \langle\texttt{root}\rangle]$ value over all $\mathcal{L}_\ell \in P(\langle\texttt{root}\rangle,0) = \mathcal{LL}(\mathcal{G})$, as desired.

At each stage, FOCL also records whether the preferred label within each $P(n_k,i)$ mapped n_k to C^+ or C^-; it can assemble these mappings to form the optimal labelling. Notice the runtime of the FOCL process is $O(|\mathcal{N}|^2)$.[6]

[6] Given the graph and values shown in Figure 2 and the parameters $R=2$, $L=1$ and $S=10$, the optimal labelling is $\mathcal{L}_{\langle -+-- \rangle}$. Its cost is 29% (resp., 45%) better than the alternative approach of not caching at all (resp., indiscriminately caching everywhere). Section 4 provides a more comprehensive set of examples.

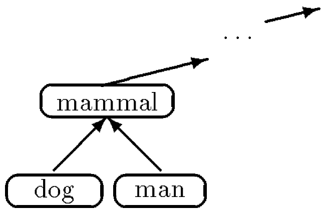


Figure 5: Multiple rules matching a goal

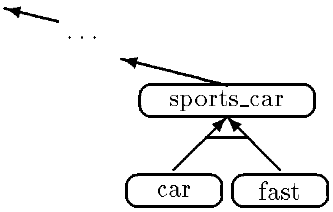


Figure 6: Conjunction in the Inference Graph

3.2 Multiple Rules matching a (Sub)Goal

This subsection deals with the situation where multiple rules can match a goal. Consider Figure 5, taken from the left side of Figure 1's inference graph), and let n_a (resp., n_d, n_m) represent the node whose literal is `mammal(X)` (resp., `dog(X)`, `man(X)`). In this case the nodes in each of the two diverging branches will receive the same number of indirect queries:

$$I_\ell(n_m) \;=\; I_\ell(n_d) \;=\; \begin{cases} I_\ell(n_a) + d(n_a) & \text{if } \mathcal{L}_\ell[n_a] = \mathrm{C}^- \\ 1 & \text{if } \mathcal{L}_\ell[n_a] = \mathrm{C}^+ \end{cases}$$

As each of these two branches, separately, is a linear knowledge base, we can use the analysis from the previous section. During the upward traversal, notice that Equation 3 continues to hold, even though $Ch(n_a) = \{n_d, n_m\}$ is not a singleton. (As the first term of Equation 3 is a summation over all the node's children, it will incorporate the cost of both the branches.) Thus the cost equation easily generalizes to the case of trees. The other computations remain the same as in the simple linear knowledge base case.

3.3 Conjunctions in Rules

Each rule with a conjunctive antecedent (*i.e.*, each clause with more than one negative literal) corresponds to a more complicated hyper-arc in the inference graph. While computing the cost function for such nodes, we must deal with the extra overhead of finding solutions that satisfy all literals. This process is equivalent to evaluating a join in the relational database [Ull88]. There are various methods of evaluating joins, including selection on an attribute, sort join, multiway merge-sort, join using index, etc. [Ull89]. As sort join seems to work well in general, we will base our discussion on this approach.

To explain the working of the sort join, consider the inference graph shown in Figure 6 (taken from the far right side of Figure 1). To answer the query `sports_car(X)`, we first find all the solutions to `car(X)` and `fast(X)` individually, and then sort each of them independently. Then, in a single traversal of the sorted lists, we find the values that are common to both `car` and `fast`, giving the set of answers to the `sports_car` query.

Now does this affect our formulation? For each query at `sports_car`, there will be one query at each of `car` and `fast`. Thus the expressions for the basic terms remain similar to the previous subsection. The cost for sorting a list of length m is $O(m \log m)$ [AHU87], and of a simple traversal, is $O(m)$. Thus, in the above example if there are m_1 solutions to `fast` and m_2 solutions to `car`, the asymptotic cost of evaluating the conjunction is bounded by $K(m_1 \log m_1 + m_2 \log m_2 + m_1 + m_2)$ where K is a constant dependent on the implementation. This value needs to be added to the cost expression Equation 3.

When there are more than two subgoals in a conjunction, we need to use multi-way joins [Ull89]. The basic process, however, remains the same.

3.4 Computing the Parameters

In any given environment, these values of the three implementation dependent parameters, L, R and S, are standard and should be known from the supplier's data. The three functions, $u(n_k)$, $d(n_k)$ and $s(n_k)$, are specific to an application and have to be obtained by the user of the FOCL algorithm.

Fortunately, we can estimate these values based on the statistics that are maintained by several commercial database systems [SAC+79]. We can periodically collect these statistics (*e.g.*, each time the system is "re-compiled"), and use these values as (estimates of) $s(n_k)$, $u(n_k)$ and $d(n_k)$ when computing the appropriate caching label. We can also use statistical measures to bound our confidence in these estimates; see [Gre92].

Notice that both $d(n_i)$ and $u(n_i)$ are with respect to some interval of time — either the "lifetime" of the overall system, or the time during which this caching strategy is "in effect", which can be the interval of time between a pair of re-compilations.

4 Empirical Results

This section compares the performance of three IMS systems: IMS_{opt} that uses the *optimal caching scheme* obtained by FOCL, $\text{IMS}_\forall$ that uses the *cache everywhere scheme* of [SM91], and $\text{IMS}_\neg$ that uses the *no cache scheme*.

Experimental Setup: We determined the values of $E[\text{IMS}_{opt}]$, $E[\text{IMS}_\forall]$ and $E[\text{IMS}_\neg]$ in 54 different contexts. Each context is defined in terms of a particular knowledge base and specific distribution of queries, specified below. All simulations use the same cost parameters $R = 2$, $L = 1$ and $S = 10$, obtained from experimental data [Deb90]. The depth of each knowledge base is set at 6, which is considered typical for real applications. Furthermore, the only atomic database facts that match a node are at the leaves; *i.e.*, $s(n_i) = 0$ for all internal nodes n_i. We also specify that each node $n \in \mathcal{N}$ is updated at the rate of $u(n) = 0.01$, *i.e.*, once every one hundred queries.

We considered three clusters of experiments, which differ in terms of the number of ground instances that match each leaf predicate. Figure 7 (resp., Figure 8, Figure 9) describes 18 contexts in which each leaf predicate matches exactly 1 (resp., 5, 10) database literals.

Each cluster of $18 = 6 \times 3$ contexts is formed as the cross-product of 6 different knowledge bases, depending on whether the branching factor from goal to subgoals was 1..6,[7] and three different query distributions: In distribution `Dist1`, the root receives 10 queries (*i.e.*, $d(\langle \texttt{root} \rangle) = 10$), each node at the next level receives 20, then 30 for each node at level 3, etc. In `Dist2`, this is reversed — each leaf node receives 10 queries, and the root, 60, as there are 6 levels. In `Dist3`, every node at every level receives same number of queries. ([CG92] specifies this data more precisely.)

Each graph plots $\frac{E[\,\mathrm{IMS}_{\forall}\,]-E[\,\mathrm{IMS}_{opt}\,]}{E[\,\mathrm{IMS}_{opt}\,]}$ and $\frac{E[\,\mathrm{IMS}_{\neg}\,]-E[\,\mathrm{IMS}_{opt}\,]}{E[\,\mathrm{IMS}_{opt}\,]}$, versus the branching factor, for each of its 18 contexts. Notice that better IMS systems have *smaller* $\frac{E[\,\mathrm{IMS}\,]-E[\,\mathrm{IMS}_{opt}\,]}{E[\,\mathrm{IMS}_{opt}\,]}$ values. As the optimal IMS_{opt} system has the uniform value 0, we did not plot values for the IMS_{opt} system in these graphs.

Experiment Cluster 1: 1 **Ground Instance per Leaf node:** These results appear in Figure 7. For small values of ground instances (*i.e.*, for branching factor $\approx$ 1), the best scheme is to cache everywhere and therefore the cost obtained by $\mathrm{IMS}_{\forall}$ is the same as the cost of IMS_{opt}. This does *not* hold for larger branching factors, however; for branching factor is 6, IMS_{opt}'s cost is 55% better than $\mathrm{IMS}_{\forall}$. The opposite is true for $\mathrm{IMS}_{\neg}$ as the degradation in cost is more pronounced for low values of the branching factor. (It is 700% when the branching factor is 1. This actual value is too high to be shown in Figure 7.) While the actual improvements vary with the specific query distribution, the improvements are significant in all cases — an average of 30%. However, it seems that `Dist2`, with more queries at "higher" nodes, favours $\mathrm{IMS}_{\forall}$, which makes sense as there can be more saving in the inference cost, regardless of the query distribution.

Experiment Cluster 2: 5 **Ground Instances per Leaf node:** As shown in Figure 8, $\mathrm{IMS}_{\forall}$'s cost is far worse than IMS_{opt}'s in most of these cases. In fact, $\mathrm{IMS}_{\neg}$ is closer to optimum, even though it can be worse by as much as 50%. $\mathrm{IMS}_{\forall}$ performs well only when the size of knowledge base is small (five rules, one ground instance) or when the query distribution is uniform (`Dist3`). Since $\mathrm{IMS}_{\forall}$ caches everywhere, it fails to respond to the variations in query distribution. This effect was less pronounced when the number of instances was lower, as in the previous experiment.

Experiment Cluster 3: 10 **Ground Instances per Leaf node:** As shown in Figure 9, $\mathrm{IMS}_{\neg}$ seems to be

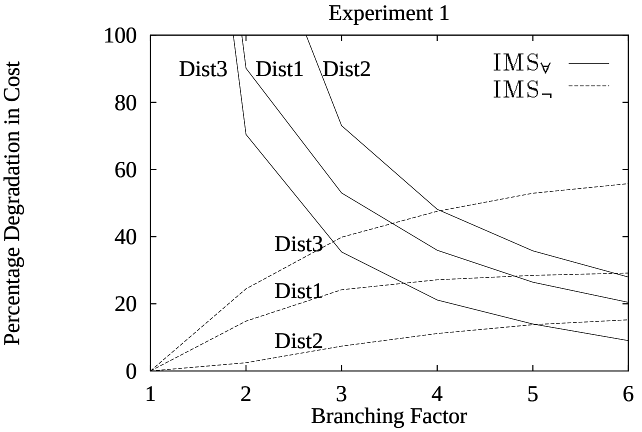


Figure 7: Results for Experiment Cluster 1

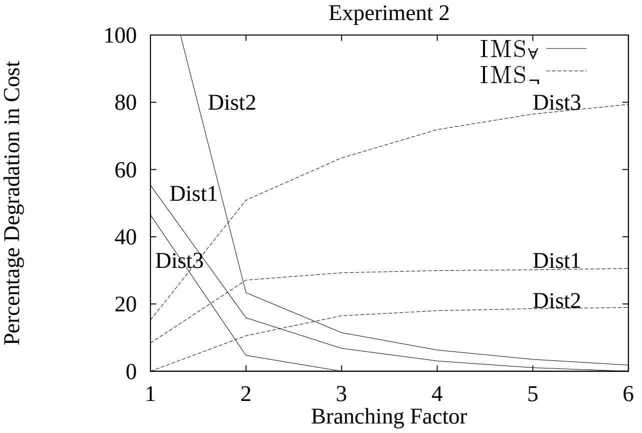


Figure 8: Results for Experiment Cluster 2

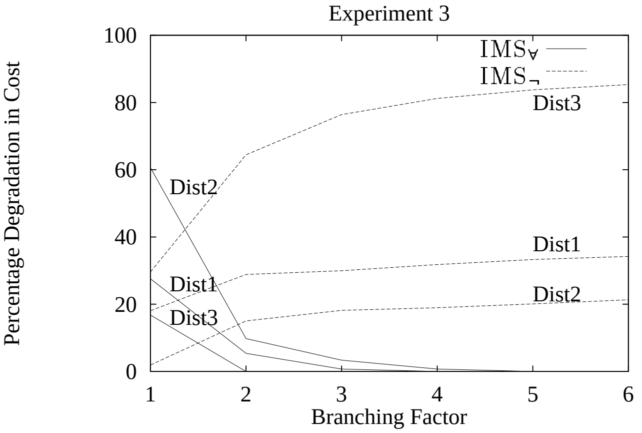


Figure 9: Results for Experiment Cluster 3

[7] Hence, the knowledge bases ranged from 5 rules and 1 ground instance (in Figure 7's framework, when the branching factor is 1) to about 10,000 rules and approximately 100,000 ground instances (in Figure 9's framework, when the branching factor is 6). Note that these are precisely the kind of knowledge bases envisaged to be required in the future applications [MCPT91].

almost optimal here; in most cases within 10%. $\mathrm{IMS}_\forall$ does worse in general, especially with larger branching factors.

Summary of Experiments: These experiments suggest that $\mathrm{IMS}_\forall$ is appropriate only for small number of ground instances; this explains the results obtained by [SM91]. However, this approach does not hold for larger knowledge bases. Given a large number of ground instances, it may be better to use $\mathrm{IMS}_\neg$ rather than $\mathrm{IMS}_\forall$. Of course, neither of these can be better than IMS_{opt}, which is guaranteed to have the best performance in all cases.

5 Conclusions

This work can be extended in a few directions: To deal with inference graphs that are not tree-shaped (*e.g.*, redundant [Gre91], recursive [SGG86], etc.); to estimate the number of solutions that can be cached at each node without actually running the inference process; to use sampling to approximate the distribution of queries and updates [LN90, Gre92]; and to deal with different cost models — *e.g.*, to include the storage cost of maintaining cached values, or to allow the values of the various parameters (*e.g.*, R, L, S) to vary with the size of knowledge base, or the number of variables involved, etc.

To recap: this paper first presents a formal definition of caching and a quantitative model that formalizes the interactions among the various parameters that affect caching performance. We use this model to design FOCL, an efficient algorithm that computes the optimal cache labelling for any "tree shaped" knowledge base; *i.e.*, FOCL determines which literals should cache their solutions to obtain an IMS system whose overall cost (for answering a given distribution of queries, given a specific distribution of updates, etc.) is minimal. The paper also presents a set of experiments to illustrate that the FOCL-based IMS_{opt} can outperform systems based on either of the standard caching strategies (no caching or universal caching), across a broad range of contexts.

References

[AHU87] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structure and Algorithms.* Addison-Wesley Publishing Company, 1987.

[CG92] Vinay K. Chaudhri and Russell Greiner. A formal analyis of caching in backward chaining systems. Technical Report forthcoming, University of Toronto, 1992.

[CM81] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog.* Springer-Verlag, New York, 1981.

[Deb90] S. K. Debray. Personal Communication, 1990.

[Gre91] Russell Greiner. Finding the optimal derivation strategy in a redundant knowledge base. *Artificial Intelligence*, 50(1):95–116, 1991.

[Gre92] Russell Greiner. Learning efficient query processing strategies. In *Proceedings of Eleventh ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, 1992.

[LN90] R.J. Lipton and J.F. Naughton. Query size estimation by adaptive sampling. In *Proceedings of Ninth ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, pages 40–46, 1990.

[MCPT91] John Mylopoulos, Vinay K. Chaudhri, Dimitris Plexousakis, and Thodoros Topaloglou. Four Obvious Steps Towards Knowledge Base Management. In *Proceedings of the Second International Workshop on Intelligent and Cooperative Information Systems: Core Technology for Next Generation Information Systems*, pages 28–30, Como, Italy, October 1991.

[Mos83] M. G. Moser. An Overview of NIKL, the new implementation of KL–ONE. In C. L. Sidner, editor, *Research in Knowledge Representation and Natural Language Understanding – Annual Report, 1 September 1982 - 31 August 1983*, pages 7–26. Bolt Beranek and Newman, 1983.

[Nem66] George L. Nemhauser. *Introduction to Dynamic Programming.* John Wiley and Sons, Inc., New York, 1966.

[SAC+79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access Path Selection in a Relational DBMS. In *Proceedings of the 1979 SIGMOD Conference*, 1979.

[Sel89] Timos K. Sellis. Efficiently supporting procedures in a relational database system. In *Proceedings of the 1987 SIGMOD Conference*, Detroit, August 1989.

[SGG86] D. E. Smith, M. R. Genesereth, and M. L. Ginsberg. Controlling recursive inference. *Artificial Intelligence*, 30(3):343–89, 1986.

[SJGP90] Michael Stonebraker, Anant Jhingaran, Jeffrey Goh, and Spyros Potamianos. On rules, procedures, caching and views in databases. Technical Report UCB/ERL M90/36, University of California, Berkeley, April 1990.

[SM91] G. Schmolze and William S. Mark. The NIKL Experience. *Cognitive Science*, 6(2):48–69, February 1991.

[Ull88] Jeffrey Ullman. *Principles of Data Base and Knowledge Base Systems*, volume 1. Addison Wesley, 1988.

[Ull89] Jeffrey Ullman. *Principles of Data Base and Knowledge Base Systems*, volume 2. Addison Wesley, 1989.