Technion - Computer Science Department - Technical Report CIS9812 - 1998

Controlled Utilization of Control Knowledge for Speeding up Logic Inference

Oleg Ledeniov olleg@cs.technion.ac.il
Shaul Markovitch shaulm@cs.technion.ac.il
Computer Science Department
Technion – the Israel Institute of Technology
32000 Haifa Israel

Abstract

The utility problem occurs when the cost of the acquired knowledge outweighs its benefit. When the learner acquires control knowledge for speeding up a problem solver, the benefit is the speedup gained due to the better control, and the cost is the added time required by the control procedure due to the added knowledge. Previous work in this area was mainly concerned with the cost of matching control rules. The solutions to this kind of utility problem involved some kind of selection mechanism to reduce the number of control rules (or, generally, they involved filtering of control knowledge).

In this work we consider a control mechanism that carries very high cost regardless of the particular knowledge acquired, therefore filtering of control knowledge does not reduce the execution time. We propose to use in such cases explicit reasoning about the economy of the control process. A "control of control" module supervises the work of the control procedure, invoking and stopping it selectively. For its decisions, this module needs some "control of control" knowledge, which can be learned from experience.

We have implemented this framework within the context of a program for speeding up logic inference by subgoal ordering. We conducted a series of experiments that showed the usefulness of the proposed framework.

1. Introduction

Speedup learning is a sub-area of machine learning where the goal of the learner is to acquire knowledge for accelerating the speed of a problem solver (Minton, 1988; Tadepalli & Natarajan, 1996). Several works in speedup learning concentrated on acquiring control knowledge for controlling the search performed by a problem solver. When the cost of using the acquired control knowledge outweighs its benefits, we face the so called *Utility Problem* (Minton, 1988; Gratch & DeJong, 1992; Markovitch & Scott, 1993a).

Existing works dealing with the utility of control knowledge are based on a model of control rules whose main associated cost is the time it takes to match their preconditions. Several solutions has been proposed for this instance of the utility problem. Most of the solutions are based on *filtering out* control rules that are estimated to be of low utility (Minton, 1988; Markovitch & Scott, 1993a; Gratch & DeJong, 1992). Others try to restrict the complexity of the preconditions (Tambe, Newell, & Rosenbloom, 1990).

In this work we consider a different setup, where the control procedure has potentially very high complexity, regardless of the specific control knowledge acquired. In this type of setup, the utility problem can become very significant, since the cost of using the control

knowledge can be higher than the time it saves on search. Filtering is not useful in such cases, because deleting control knowledge will not necessarily reduce the complexity of the control process.

For dealing with this problem, we propose to use explicit reasoning about the economy of the control process. For example, consider the following three-step framework:

1. Convert the control procedure into an *anytime* one (so that it can be stopped at any moment).
2. Learn a *resource investment function* for the anytime procedure. This function predicts the expected savings in search time, given the resources that are invested in control decision.
3. Run the anytime control procedure so that the control time plus the expected search time is minimal.

In this paper we demonstrate this framework as applied to a learning system for speeding up logic inference. The control procedure is a subgoal-ordering algorithm. The learning system learns costs and numbers of solutions of subgoals to be used by the ordering algorithm. A "good" ordering of subgoals will increase the efficiency of the logic interpreter. However, the ordering procedure has high complexity. We employ the above framework by first converting our ordering algorithm into an anytime procedure. We then learn a resource-investment function for each goal pattern. The functions are then used to terminate the ordering procedure before its cost becomes too high. We demonstrate experimentally how the ordering time decreases without harming significantly the quality of the resulting ordering.

The paper is organized as follows. Section 2 presents the LASSY1 system, on which we shall illustrate our ideas. The system is aimed for subgoal ordering in logic programs. Section 3 shows empirically that the utility problem is present in the LASSY1 system. In Section 4 we propose solutions to the problem through controlling of the control procedure. One of the solutions was briefly described above – we learn resource-investment functions that tell the ordering procedure to stop at the optimal moment. Another solution involves caching of generalized control decisions (in the case of subgoal ordering, this means caching the ordering results that were made in the past). We show that the proposed methods, separately and in combination, succeed to reduce the utility problem. Section 5 concludes.

2. Learning Control Knowledge for Speeding up Logic Inference

In this section we describe the architecture of our system, which we shall use to illustrate our methods. The structure of the system is to the large extent based on the LASSY system of Markovitch and Scott (1993b) (more exactly, on its inductive part, without the lemma learning), therefore we do not introduce new terminology and call our system LASSY1. The architecture is depicted in Figure 1. The architecture is divided into a learning system and a performance system. The performance system is an interpreter for logic programs. We assume that the interpreter receives a domain and a sequence of user queries (drawn from a fixed distribution). The goal of the learning system is to learn control knowledge for speeding up the performance of the Prolog interpreter with respect to the given query

Technion - Computer Science Department - Technical Report CIS9812 - 1998

Technion - Computer Science Department - Technical Report CIS9812 - 1998

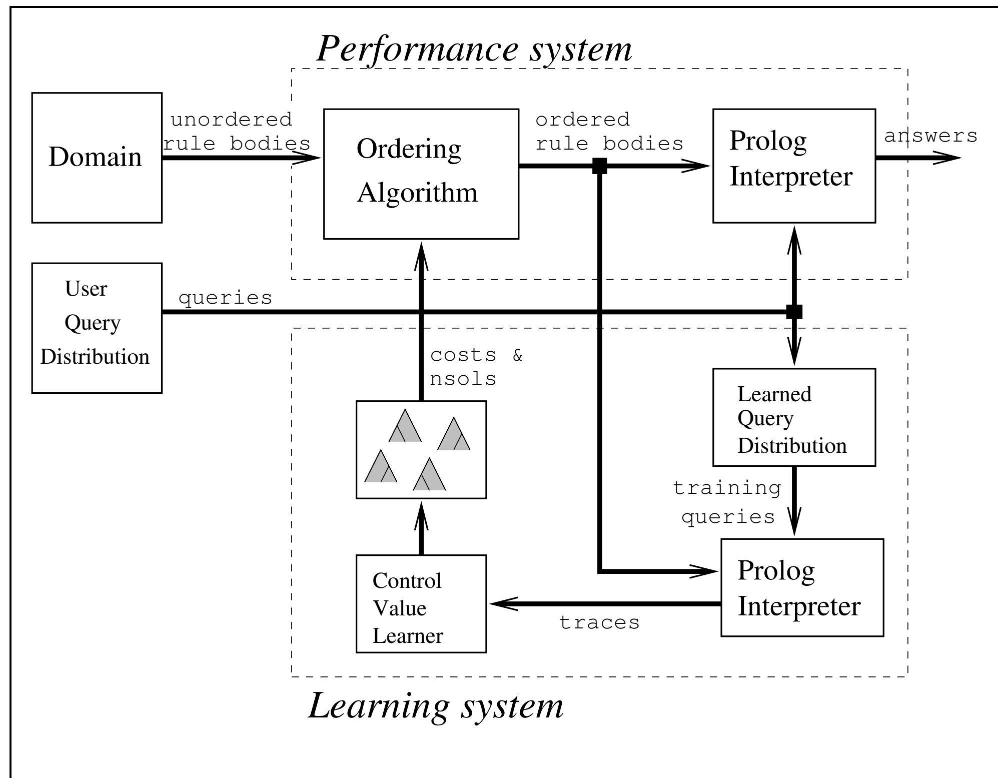


Figure 1: The initial architecture of the LASSY1 system.

distribution. In this paper we present an off-line version of the learning system. However, our learning methodology can be used in on-line fashion as well.

The control procedure of the performance system is a subgoal ordering algorithm. The order in which subgoals are executed affects significantly the efficiency of the logic inference process (Warren, 1981; Nie & Plaisted, 1990; Naish, 1985). Before the Prolog interpreter reduces a subgoal by a rule body, it calls the ordering algorithm which provides it with a low-cost ordering of this rule body. The cost reflects the amount of resources spent during the proof – in this paper we define it as the number of unifications performed. The operation of the control procedure depends on average costs and numbers of solutions of various predicates under various bindings. The goal of the learning system is to acquire this control knowledge from the problem-solving experience.

The learning system maintains a query distribution which is inferred from past user queries. This distribution is used to generate training queries. The learning system applies to these queries the Prolog interpreter (augmented by the subgoal ordering algorithm). The proof traces of the training queries are passed to the learning module, which extracts the control knowledge and generalizes it, yielding a set of regression trees – a special case of decision trees that classify to continuous values. The ordering procedure can then use these trees for estimating the cost and the number of solutions of literals during the ordering process.

Technion - Computer Science Department - Technical Report CIS9812 - 1998

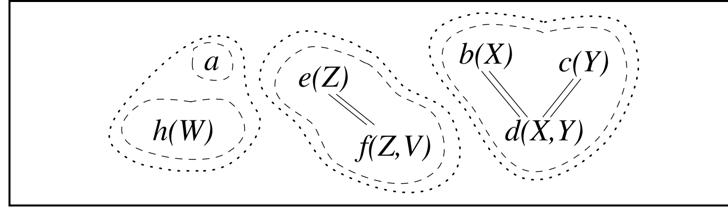


Figure 2: An example of a graph representing a set of subgoals {a, b(X), c(Y), d(X,Y), e(Z), f(Z,V), h(W)}. Directly dependent subgoals are connected by edges. Independent subgoals and indivisible subsets are equivalent to connected components (surrounded by dashed lines). The divisibility partition (under empty binding set) is shown by dotted lines.

In the subsections that follow we describe the control component and the learning component in more details.

2.1 The Control Component: the Subgoal Ordering Algorithm

In this subsection we define briefly the Divide-and-conquer subgoal ordering algorithm which serves as the control procedure in the LASSY1 architecture. A more detailed presentation appears in (Ledeniov & Markovitch, 1998).

The problem solver is a Prolog interpreter for logic programs, and the control procedure orders subgoals in AND nodes of AND-OR proof trees (Lloyd, 1984). When the current goal is unified with a rule head (a *reduction* is performed), the set of subgoals of the rule body, under the current binding of variables, is given to the ordering algorithm. The algorithm produces candidate orderings and estimates their cost using the following equation:

$$\begin{aligned} Cost(\langle A_1, A_2, \ldots A_n \rangle) &= \overline{cost}(A_1) + \overline{nsols}(A_1) \times \overline{cost}(A_2) + \ldots + \left(\prod_{j=1}^{n-1} \overline{nsols}(A_j) \right) \times \overline{cost}(A_n) \\ &= \sum_{i=1}^{n} \left[\left(\prod_{j=1}^{i-1} \overline{nsols}(A_j)|_{\{A_1 \ldots A_{j-1}\}} \right) \times \overline{cost}(A_i)|_{\{A_1 \ldots A_{i-1}\}} \right], \end{aligned} \tag{1}$$

where $\overline{cost}(A_i)|_{\mathcal{B}}$ is the *average cost* of proving a subgoal A_i under all the solutions of a set of subgoals $\mathcal{B}$ and $\overline{nsols}(A_i)|_{\mathcal{B}}$ is the *average number of solutions* of A_i under all the solutions of $\mathcal{B}$. Here $\mathcal{B}$ is called the *binding set* of A_i. For each subgoal A_i, its average cost is multiplied by the total number of solutions of all preceding subgoals. We also define the *cn* function which operates on subgoals:

$$cn(A)|_{\mathcal{B}} = \frac{\overline{nsols}(A)|_{\mathcal{B}} - 1}{\overline{cost}(A)|_{\mathcal{B}}}. \tag{2}$$

The main idea of the Divide-and-conquer algorithm is to create a special AND-OR tree, called the *divisibility tree*, which represents the partition of the given set of subgoals into subsets, and to perform a traversal of this tree. The partition is performed based on *subgoal dependence*:

Technion - Computer Science Department - Technical Report CIS9812 - 1998

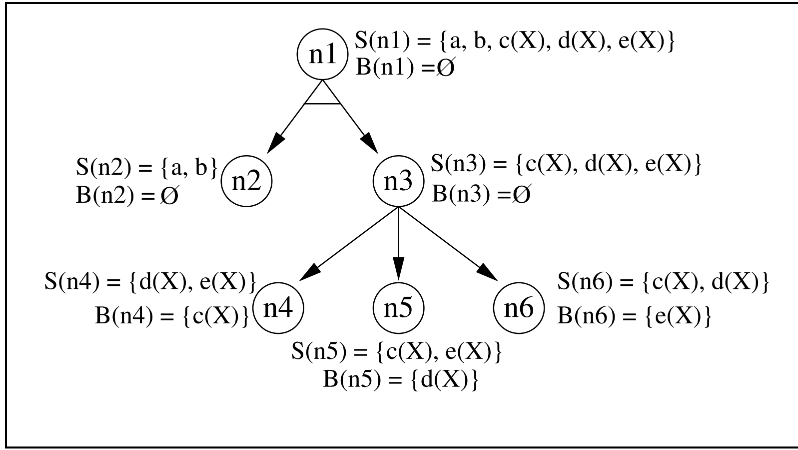


Figure 3: The divisibility tree of $\{a, b, c(X), d(X), e(X)\}$. The set associated with node $n1$ is divisible, and is represented by an AND-node. Its children correspond to its divisibility subsets – one independent, $S(n2) = \{a, b\}$, and one indivisible, $S(n3) = \{c(X), d(X), e(X)\}$. $n3$ is an OR-node, whose children correspond to its three subgoals. The sets $S(n2)$, $S(n4)$, $S(n5)$ and $S(n6)$ are independent under the corresponding binding sets, and their nodes are leaves. Here we assumed that X is bound after it appears in an argument of a subgoal.

1. Two subgoals are *directly dependent* under the binding set $\mathcal{B}$, if they share a variable, not bound by a subgoal of $\mathcal{B}$ (i.e., after we prove all subgoals from $\mathcal{B}$, this variable remains unbound). Dependence is the transitive closure of direct dependence.

2. A set of subgoals is *independent* under the binding set $\mathcal{B}$, if all its subgoal pairs are independent under $\mathcal{B}$, and is *dependent* otherwise. A dependent set of subgoals is *indivisible* if all its subgoal pairs are dependent under $\mathcal{B}$, and is *divisible* otherwise. If we represent subgoals as graph vertices and direct dependence as graph edges, then the set is indivisible iff the graph is connected.

3. A *divisibility partition* of $\mathcal{S}$ under $\mathcal{B}$, $DPart(\mathcal{S}, \mathcal{B})$, is a partition of $\mathcal{S}$, whose elements are subsets of $\mathcal{S}$ that are mutually independent and indivisible under $\mathcal{B}$, except at most one element which contains all the subgoals independent of $\mathcal{S}$ under $\mathcal{B}$. If we continue the parallel with graphs, then the divisibility partition is the partition of a graph into connected components, with all the "lonely" nodes collected together, in one component – see Figure 2 for illustration.

4. The *divisibility tree* of a set of subgoals $\mathcal{S}_0$ is an AND-OR tree. Each node N in it has an associated set of subgoals $\mathcal{S}(N)$ and an associated binding set $\mathcal{B}(N)$ (for the root node, $\mathcal{S}(N) = \mathcal{S}_0$ and $\mathcal{B}(N) = \emptyset$). If $\mathcal{S}(N)$ is *independent* under $\mathcal{B}(N)$, then N is a leaf. If $\mathcal{S}(N)$ is *indivisible* under $\mathcal{B}(N)$, then N is an OR-node, and each subgoal B_i in $\mathcal{S}(N)$ defines a child node whose set of subgoals is $\mathcal{S}(N) \setminus \{B_i\}$ and the binding set is $\mathcal{B}(N) \cup \{B_i\}$. If $\mathcal{S}(N)$ is *divisible* under $\mathcal{B}(N)$, then N is an AND-node, and each subset $\mathcal{S}_i$ in the divisibility partition $DPart(\mathcal{S}, \mathcal{B})$ defines a child node with associated

Technion - Computer Science Department - Technical Report CIS9812 - 1998

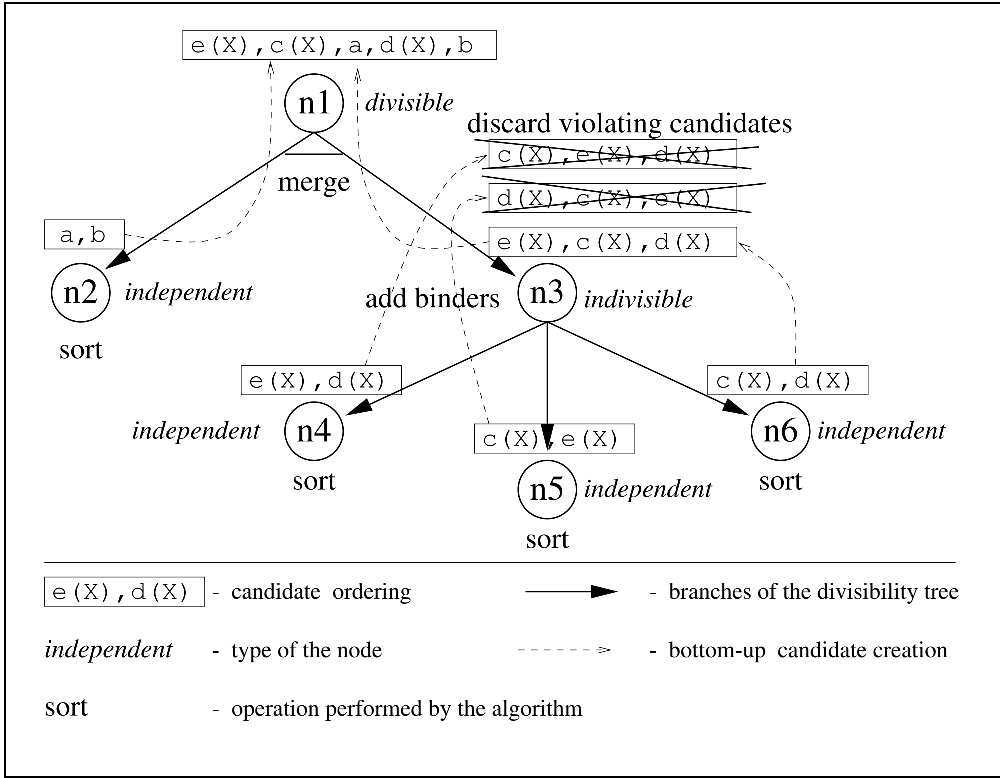


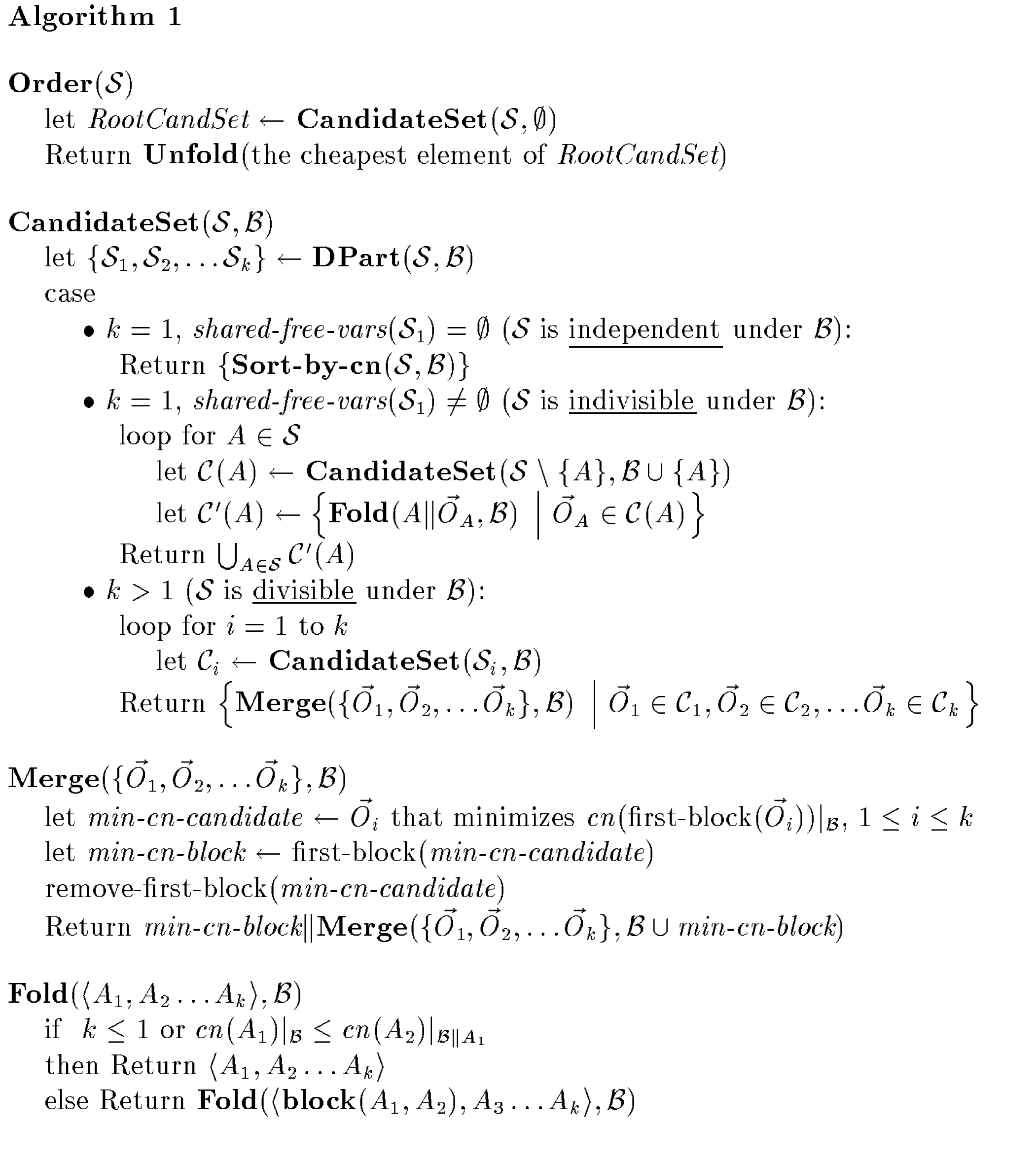
Figure 4: Illustration of the work of the Divide-and-conquer algorithm on the divisibility tree of Figure 3. Dashed lines show the bottom-up creation of candidate orderings. We do not show here the exact control values that lead to such results.

set of subgoals $\mathcal{S}_i$ and binding set $\mathcal{B}(N)$. Figure 3 shows the divisibility tree of the set $\mathcal{S}_0 = \{a, b, c(X), d(X), e(X)\}$.

For each node of the tree, a set of *candidate orderings* is created, and the orderings of an internal node are obtained by combining orderings of its children. For different types of nodes in the tree, the combination is performed differently. We proved several sufficient conditions that allow us to discard a large number of possible ordering combinations, therefore the obtained sets of candidate orderings are generally small:

1. If a subset of subgoals is independent, then the node has one candidate, whose subgoals are *sorted* by their cn values (Equation 2).
2. If a subset is indivisible, then each candidate of the node is obtained by adding one of its subgoals to the left end of a candidate of its child, that corresponds to this subgoal.
3. If the subset of a node is divisible, then each its candidate is obtained by merging candidates of its children. For each combination of candidates of children one candidate of the AND-node is created. We detect *blocks* – sequences of subgoals that

Technion - Computer Science Department - Technical Report CIS9812 - 1998



Algorithm 1

Order($\mathcal{S}$)
let *RootCandSet* $\leftarrow$ **CandidateSet**($\mathcal{S}, \emptyset$)
Return **Unfold**(the cheapest element of *RootCandSet*)

CandidateSet($\mathcal{S}, \mathcal{B}$)
let $\{\mathcal{S}_1, \mathcal{S}_2, \ldots \mathcal{S}_k\} \leftarrow$ **DPart**($\mathcal{S}, \mathcal{B}$)
case
- $k = 1$, *shared-free-vars*$(\mathcal{S}_1) = \emptyset$ ($\mathcal{S}$ is independent under $\mathcal{B}$):
  Return $\{$**Sort-by-cn**$(\mathcal{S}, \mathcal{B})\}$
- $k = 1$, *shared-free-vars*$(\mathcal{S}_1) \neq \emptyset$ ($\mathcal{S}$ is indivisible under $\mathcal{B}$):
  loop for $A \in \mathcal{S}$
  let $\mathcal{C}(A) \leftarrow$ **CandidateSet**$(\mathcal{S} \setminus \{A\}, \mathcal{B} \cup \{A\})$
  let $\mathcal{C}'(A) \leftarrow \left\{ \mathbf{Fold}(A \| \vec{O}_A, \mathcal{B}) \;\middle|\; \vec{O}_A \in \mathcal{C}(A) \right\}$
  Return $\bigcup_{A \in \mathcal{S}} \mathcal{C}'(A)$
- $k > 1$ ($\mathcal{S}$ is divisible under $\mathcal{B}$):
  loop for $i = 1$ to k
  let $\mathcal{C}_i \leftarrow$ **CandidateSet**$(\mathcal{S}_i, \mathcal{B})$
  Return $\left\{ \mathbf{Merge}(\{\vec{O}_1, \vec{O}_2, \ldots \vec{O}_k\}, \mathcal{B}) \;\middle|\; \vec{O}_1 \in \mathcal{C}_1, \vec{O}_2 \in \mathcal{C}_2, \ldots \vec{O}_k \in \mathcal{C}_k \right\}$

Merge$(\{\vec{O}_1, \vec{O}_2, \ldots \vec{O}_k\}, \mathcal{B})$
let *min-cn-candidate* $\leftarrow \vec{O}_i$ that minimizes $cn(\text{first-block}(\vec{O}_i))|_{\mathcal{B}}$, $1 \leq i \leq k$
let *min-cn-block* $\leftarrow$ first-block(*min-cn-candidate*)
remove-first-block(*min-cn-candidate*)
Return *min-cn-block*$\|$**Merge**$(\{\vec{O}_1, \vec{O}_2, \ldots \vec{O}_k\}, \mathcal{B} \cup$ *min-cn-block*$)$

Fold$(\langle A_1, A_2 \ldots A_k \rangle, \mathcal{B})$
if $k \leq 1$ or $cn(A_1)|_{\mathcal{B}} \leq cn(A_2)|_{\mathcal{B} \| A_1}$
then Return $\langle A_1, A_2 \ldots A_k \rangle$
else Return **Fold**$(\langle \mathbf{block}(A_1, A_2), A_3 \ldots A_k \rangle, \mathcal{B})$

Figure 5: The Divide-and-conquer ordering algorithm.

cannot be broken. In the AND-node candidates, blocks of the children candidates must be sorted by cn. The blocks grow with time (we call this process *folding*, since we maintain each block as a separate entity, like an ordinary subgoal).

The candidate orderings are propagated up the divisibility tree. The last step of the algorithm is to return the cheapest candidate of the root according to Equation 1. The execution of the algorithm on the divisibility tree of Figure 3 is illustrated in Figure 4.

Technion - Computer Science Department - Technical Report CIS9812 - 1998

The Divide-and-conquer algorithm is listed in Figure 5. The code of the **DPart**, **Unfold** and **Sort-by-cn** procedures is not listed, due to its straightforwardness. The merging procedure recursively extracts from the given folded orderings blocks that are minimal by cn. The folding procedure joins two leading blocks of a sequence into a larger block, as long as the first block has larger cn value than the second one. The correctness proof of the algorithm can be found in (Ledeniov & Markovitch, 1998).

The Divide-and-conquer algorithm has exponential complexity in the worst case: $O(n!)$ if we work with a divisibility tree, or $O(n^2 \cdot 2^n)$ if we pass to *divisibility graphs*, where all identical nodes of a divisibility tree are collided together. However, in most practical cases its time complexity is polynomial. For a small number v of shared free variables, the complexity of the algorithm is roughly bounded by $O(n^{v+1} \cdot \log n)$. Usually, the number of shared free variables in a rule body is relatively small (note that here we refer not to all the free variables that are written in the program text, but to those of them that remain free after the rule head unification). For example, if all the subgoals are independent under the current variable binding ($v = 0$), the complexity is $O(n \log n)$. For more details about the Divide-and-conquer ordering algorithm see (Ledeniov & Markovitch, 1998).

2.2 The Learning Component: Acquisition of Control Knowledge Using Regression Trees

The learning system performs off-line learning by generation and processing of training queries. The training queries are generated according to the distribution of the user queries that were seen in the past. The ordering algorithm described in the previous subsection assumes the availability of correct values of average cost and number of solutions for various literals. The learning component acquires this control knowledge while solving training queries.

Storing a separate unit of control values for each literal is not practical, for two reasons. The first is the large space required by this approach. The second reason is the lack of generalization: the ordering algorithm is quite likely to encounter literals which were not seen before, and whose real control values are unknown.

The learner therefore generalizes the control knowledge: it acquires control values for *classes* of literals rather than for separate literals. One easy way to define classes is by *modes* or *binding patterns* (Debray & Warren, 1988; Ullman & Vardi, 1988): for each argument we denote whether it is free or bound. For example, for the predicate `father` the possible classes are `father(free,free)`, `father(bound,free)`, `father(free,bound)` and `father(bound,bound)`.

The more refined the classes, the smaller is the variance of real control values inside each class, and the more precise are the $\overline{cost}$ and $\overline{nsols}$ estimations that the classes assign to their members. Consequently, the Divide-and-conquer algorithm produces better (cheaper) orderings. Class refinement can be obtained by using more sophisticated tests on the predicate arguments than the simple "bound-unbound" test. For this purpose we can use *regression trees* – a sort of decision trees that classify to continuous numeric values (Breiman et al., 1984; Quinlan, 1986). Two separate regression trees are stored for every program predicate, one for its $\overline{cost}$ values, and one for the $\overline{nsols}$. For each literal whose $\overline{cost}$ or $\overline{nsols}$ is required, we address the corresponding tree of its predicate and perform

Technion - Computer Science Department - Technical Report CIS9812 - 1998

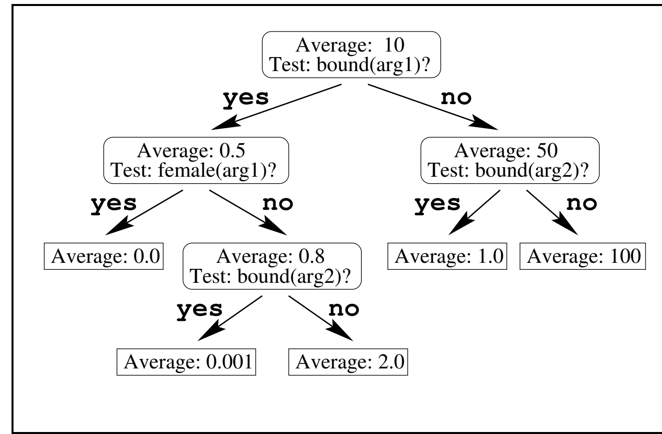


Figure 6: A possible regression tree for estimated number of solutions for `father(arg1,arg2)`.

recursive descent in it, starting from the root. Each tree node contains a test which applies to the arguments of the literal. A possible regression tree for estimated number of solutions for predicate `father` is shown in Figure 6.

The tests used in tree nodes can be *syntactic* (not depending on the current domain), like the following examples:

- Is the first argument bound?
- Is the first argument the constant `c1`?
- Is the first argument greater than 80?
- Is the first argument greater than the second one?

They can also be *semantic* (depending on the current domain), for example:

- Is the first argument female?
- Is the first argument the father of the second one?
- Is the first argument married to some relative of the second argument?
- Is the first argument older than the second one?

If we only use the test *"is argument i bound?"*, then the classes of literals defined by regression trees coincide with the classes defined by binding patterns. Semantic tests employ logic inference. For example, the first one of the semantic tests above invokes the predicate `female` on the first argument of the literal. Therefore these tests must be as "cheap" as possible, otherwise the retrieval of control values can take too much time.

Figure 7 schematically shows an hierarchy of literal classes. The averages of larger classes make less precise estimations for the class elements, but are easier to obtain. The lower do classes stand in this hierarchy, the better control estimations we obtain, and the

Technion - Computer Science Department - Technical Report CIS9812 - 1998

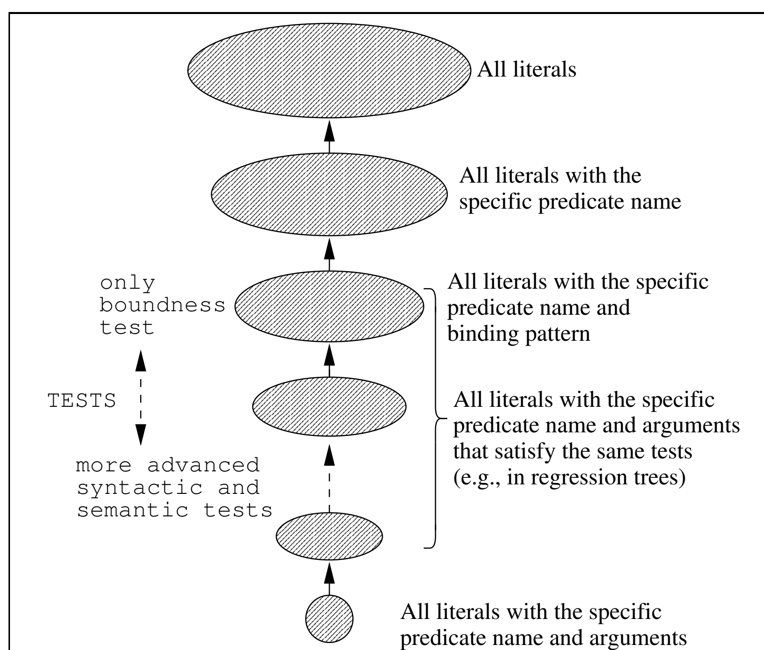


Figure 7: Hierarchy of literal classes. The estimations of larger classes are less precise, but are easier to obtain.

better is the quality of the produced orderings. On the other hand, work with more refined classes requires increased time and memory expenses.

Unlike some other regression mechanisms (e.g. Nearest-Neighbor systems), a regression tree may take much time to be built, but is very efficient in ensuing use. On account of this it ideally suits for off-line learning: the expensive tree-building process is performed during the training phase, when we are not restricted in time; during the execution phase we only use the prepared trees. Since tree building is expensive, we can perform it not after each new instance of some predicate, but after we saw a certain number of new instances: until then we use the old regression tree of the predicate and store the new instances aside. When the required number of new instances is seen, the tree is updated or rebuilt.

Before we have learned enough control knowledge to order training cases well, some proofs may be non-terminating. Also, proofs of certain literals may be inherently infinite, and we must learn this, in order to avoid execution of these literals under the specific bindings. In order to prevent unlimited investigation of infinite trees, we can employ the strategy of *censoring* (Etzioni & Etzioni, 1992): the proof is interrupted when it reaches a pre-defined limit on the tree depth, or on the number of unifications performed. In effect, after such interruption we can only assert that the cost of the literal is "very large, probably infinite", but in most cases this suffices to distinguish between finite and infinite computations.

Technion - Computer Science Department - Technical Report CIS9812 - 1998

3. Subgoal Ordering and the Utility Problem

To test the effectiveness of our ordering algorithm, we experimented with it on various domains, and compared its performance to other ordering algorithms. All experiments described below consist of a training session (when the learner acquires the control knowledge for literal classes), followed by a testing session (when the problem solver proves the queries of the testing set using different ordering algorithms). The goal of ordering is to reduce the time spent by the Prolog interpreter when it proves queries of the testing set. This time is the sum of the time spent by the ordering procedure (*ordering time*) and the time spent by the interpreter itself (*inference time*).

In order to ensure statistical significance of comparison of different ordering algorithms, we experimented with many different domains. For this purpose, we created a set of artificial domains, each with a small set of predicates, but with random number of clauses in each predicate, and with random rule lengths. Predicates in rule bodies, and arguments in both rule heads and bodies are randomly drawn from fixed distributions. Each domain has its own training and testing sets (these two sets do not intersect). Since the domains and the query sets are generated randomly, we repeated each experiment 100 times, each time with a different domain. The following ordering methods were tested:

- **Random:** Each time we address a rule body, we order it randomly (to reduce the effect of chance, we ran each experiment 10 to 20 times).
- **Best-first search**: over the space of prefixes. Out of all prefixes that are permutation of each other, only the cheapest one is retained. A similar algorithm was employed by Markovitch and Scott (1989).
- **Adjacency:** A best-first search with adjacency restriction test added. The adjacency restriction requires that two adjacent subgoals always stands in the cheapest order. A similar algorithm was described by Smith and Genesereth (1985).
- **The Divide-and-conquer algorithm**, as described in Section 2.1.

All ordering methods, apart from the Random method, use learned control knowledge for ordering. As was already pointed out in Section 2.2, one important decision is how to define classes of literals for which the control knowledge is accumulated. We tested two possible class definitions: by *binding patterns* and by *regression trees*, with simple syntactic tests (like those listed in Section 2.2). Using regression trees means spending more time for retrieval of control values, but promises more accurate control value estimations.

Another important decision is the amount of training needed: the more training examples we process, the more correct are the predictions of control values. On the other hand, after we saw enough training examples, additional training may be undesirable: it just wastes time, without much gain. We have built learning curves to detect the point when training can be safely stopped. The left graph in Figure 8 is the mean of 100 learning curves of 100 artificial domains: for each domain, the curve shows the total time of running the system on the testing set, as a function of the number of control values learned. Here we used regression trees for literal class definitions; a similar graph is built if we use binding patterns. "Heavy" domains, with large time values, have more impact on the form of

Technion - Computer Science Department - Technical Report CIS9812 - 1998

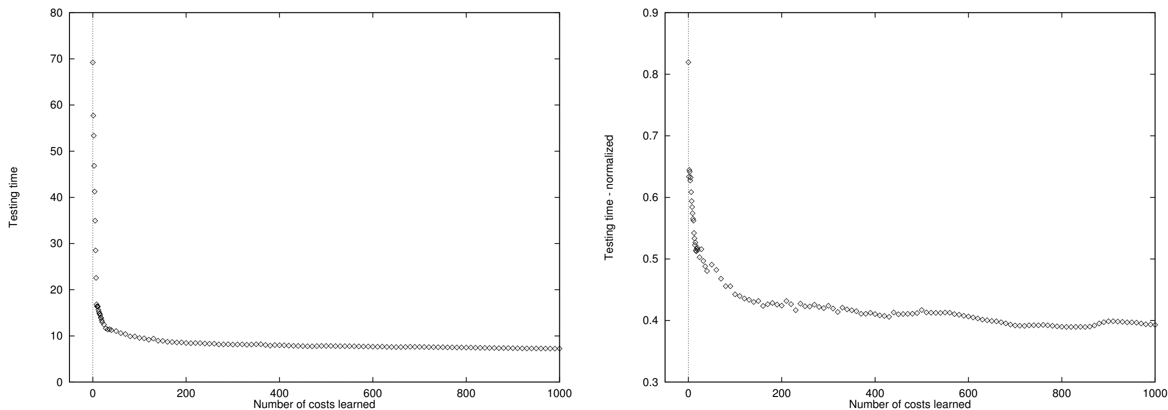


Figure 8: The left graph is the average learning curve of 100 domains, using regression trees for literal classes. The right graph shows the normalized average.

Ordering Method	Unifications	Reductions	Ordering Time	Inference Time	Total Time	Ord.Time / Reductions
Random	86052.06	27741.52	8.191	27.385	35.576	0.00029
binding patterns:						
Best-first	8526.42	2687.39	657.973	2.933	660.906	0.24
Adjacency	8521.32	2686.96	470.758	3.006	473.764	0.18
Divide-and-conquer	8492.99	2678.72	8.677	2.895	11.571	0.0032
regression trees:						
Best-first	2829.00	978.59	347.313	1.178	348.491	0.35
Adjacency	2525.34	881.12	203.497	1.137	204.634	0.23
Divide-and-conquer	2454.41	859.37	2.082	1.030	3.112	0.0024

Table 1: The effect of ordering algorithm on the tree sizes and the CPU time (mean results over 100 artificial domains).

the summary graph than their "lighter" colleagues. To avoid such bias, we *normalized* the learning curves of the domains: each curve is normalized (multiplyed by a factor), so that its maximal value becomes 1. In this way, all curves have equal impact on the summary graph. The right graph in Figure 8 shows the mean of 100 normalized learning curves. From both graphs follows that for our artificial domains training can be safely stopped after approximately 300 cost values were learned. In the subsequent experiments we stopped training after more than 600 cost values were learned (since each learned cost value means one reduction performed, the overall training time was very small).

Table 1 shows the results of this experiment, when a training session is followed by a testing session. The results clearly show the advantage of the Divide-and-conquer algorithm over other ordering algorithms. It produces much shorter inference time than the random ordering method. It requires much shorter ordering time than the other deterministic

Technion - Computer Science Department - Technical Report CIS9812 - 1998

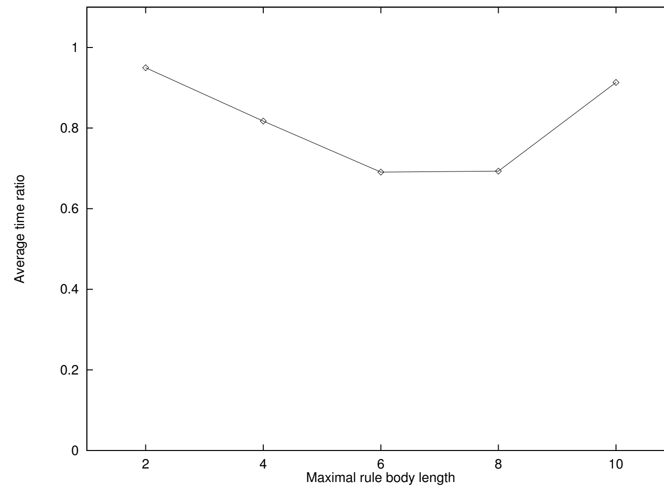


Figure 9: The graph of average utility ratios for various maximal rule body lengths.

ordering algorithms. Therefore, its total time is the best. The results with regression trees are better than the results with binding patterns. This is due to the better accuracy of the control knowledge which is accumulated for more refined classes.

It is interesting to note that the random ordering method performs better than the *best-first* and the *adjacency* methods. The inference time that these method produce is much better than the inference time when using random ordering. However, they require very long ordering time which outweighs the inference time gain. This is a clear manifestation of the utility problem where the time required by the control procedure outweighs its benefit. The Divide-and-conquer algorithm has much better ordering performance. However, its complexity is $O(n!)$ in the worst case. Therefore, it is quite feasible to encounter the utility problem even when using the efficient ordering procedure.

To study this problem, we performed another experiment where we varied the *maximal length of rules* in our randomly generated domains and tested the effect of the maximal rule length on the utility of learning. Rules with longer bodies require much longer ordering time, but also carry a greater potential benefit.

Earlier we assumed that our aim was the minimizing of *total* execution time. This approach was heavily biased to the favor of effective ordering of "heavy" domains, that added much to the total time. The fate of more "light" domains was neglected: their ordering, good or bad, did not affect the summary result. This bias is undesirable: we want our algorithm to work good with different domains, not only with the "heavy" ones. To remove this effect, we shall now compute for each domain the ratio of its total ordering time with and without ordering, and the quality of the algorithm will be measured by the sum (or average) of these ratios.

The graph in Figure 9 plots the average time saving of the ordering algorithm: for each domain we calculate the ratio of its total testing time with the Divide-and-conquer algorithm and with the random method. For each maximal body length, a point on the

Technion - Computer Science Department - Technical Report CIS9812 - 1998

graph shows the average of ratios over 50 artificial domains. For each domain, testing with the random method was performed 20 times, and the average result was taken.

The following observations can be made:

1. For short rule bodies, the Divide-and-conquer ordering algorithm performs only a little better than the random method. When bodies are short, little permutations are possible, thus the random method often finds good orderings.

2. For middle-sized rule bodies, the utility of the Divide-and-conquer ordering algorithm grows. Now the random method errs more frequently (there are more permutations of each rule body, and less chance to pick a cheap permutation). At the same time, the ordering time is not too large yet.

3. For long rule bodies, the utility decreases again. Although the tree sizes are now reduced more and more (compared to the sizes of the random method), the additional ordering time grows.

These results show that risk of encountering the utility problem exists even with our sophisticated ordering algorithm. In the following section we present a methodology for controlling the cost of the control mechanism by explicit reasoning about its expected utility.

4. The Solution: Controled Utilization of Control Knowledge

The preceding section showed an instance of the utility problem which is quite different from the one caused by the matching time of control rules. There, the cost associated with the usage of control knowledge could be reduced by filtering out rules with low utility. Here, the high cost is an inherent part of the *control procedure* and is not a direct function of the *control knowledge*.

Thereupon, we see the need to *control the control procedure*. For this purpose, we introduce a new module, which supervises the control procedure and sees that the latter does not take unnecessarily much time. This supervision can take different forms and use different algorithms. In this section we present two such methods, applied to our specific problem of subgoal ordering. One of them turns the control procedure (in our case – the ordering algorithm of Section 2.1) into an *anytime algorithm*, and commands it to stop the execution when it becomes wasteful. The other method caches the past control decisions (in our case – ordering results) in a generalized way, and then uses these cached results, if they are reliable enough (without invoking the control procedure). We performed experiments on the resulting variation of the LASSY1 system. The experiments showed a notable improvement of the ordering time, without loss of the quality of the produced orderings.

4.1 Anytime Divide-and-conquer Algorithm

The first proposed method for controlling the control procedure is an instance of the following general methodology:

1. Convert the control procedure into an anytime algorithm (Boddy & Dean, 1989).

Technion - Computer Science Department - Technical Report CIS9812 - 1998

2. Acquire a resource-investment function, which predicts the expected reduction in search time as a result of investing more control time.

3. Execute the anytime control procedure with a termination criterion based on the resource-investment function.

In this subsection we show how this three-step methodology can be applied to our Divide-and-conquer algorithm.

The basic version of the Divide-and-conquer algorithm (Algorithm 1) propagates the set of all candidate orderings in a bottom-up fashion to the root of the divisibility tree. Then it uses Equation 1 to estimate the cost of each candidate and finally returns the cheapest candidate. The anytime version of the algorithm works differently:

- Find first candidate of the root, compute its cost.
- Loop until a termination condition holds (and while there are untried candidates):
  - Find next candidate, compute its cost.
  - If it is cheaper than the current cheapest one – update the current cheapest candidate.
- Return the cheapest candidate seen.

In the new framework we do not generate all candidates exhaustively (unless the termination condition never holds). This algorithm is an instance of *anytime algorithms* (Boddy & Dean, 1989; Boddy, 1991): it always has a "current" answer ready, and at any moment can be stopped and return its current answer. The new algorithm visits each node of the divisibility tree several times: it finds its first candidate, then passes this candidate to higher levels (where it participates in mergings and concatenations). Then, if the termination condition permits, the algorithm re-enters the node, finds its next candidate and exits with it, and so on.

Algorithm 2 which implements the proposed framework is shown in Figure 10 and continued in Figure 11. To each node, apart from the associated set of subgoals and the binding set, we added several fields that serve for memorization of the state where we left the node on our last visit. In Algorithm 1, the notation of "nodes" and "children" was merely useful for better visualisation of the execution: the algorithm itself worked with sets of subgoals, and after a set was completely treated, it was discarded; a divisibility tree never existed as a data structure. Now, we must keep the lower levels of the divisibility tree when we return to the root. For this purpose, the algorithm maintains the divisibility tree as a data structure, and its work can be described as a tree traversal. Note that all nodes of a divisibility tree never physically co-exist, since in every OR-node only one child is maintained at every moment of time. Thus, if the termination condition occurs early in the course of the execution, many OR-branches are not created.

The implementation of the functions **Fold**, **Unfold**, **Merge**, **DPart** and **Sort-by-cn** is the same as in Algorithm 1 and thus is not specified here. A minor difference is that **Merge** now works with arrays and not with sets of orderings. In Algorithm 2, we call by the name *projections* the candidates of the child nodes that are interleaved to create a candidate of a node.

Technion - Computer Science Department - Technical Report CIS9812 - 1998

```
Algorithm 2

Order(S)
   Root ← MakeNode(S, ∅)   /* See below */
   min-candidate ← FirstCandidate(Root)
   min-cost ← Cost(candidate)   /* Equation 1 */
   loop  until TerminationCondition   /* discussed in Section 4.1.1 */
      or until no more candidates exist
         candidate ← NextCandidate(Root)
         cost ← Cost(candidate)   /* Equation 1 */
         if cost < min-cost
            then
               min-cost ← cost
               min-candidate ← candidate
   Return Unfold(min-candidate)

FirstCandidate(N)
   case (N.NodeType)   /* this field was set by MakeNode() */
      leaf:
         Return Sort-by-cn(N.Subgoals, N.BindingSet)
      OR:
         N.UntriedBinders ← N.Subgoals
         b ← pop(N.UntriedBinders)
         N.CurrentBinder ← b
         N.CurrentChild ← MakeNode(N.Subgoals\{b}, N.BindingSet∪{b})
         Return Fold(b || FirstCandidate(N.CurrentChild))
      AND:
         {S_1, S_2, ... S_k} ← DPart(N.Subgoals, N.BindingSet)
         N.Child ← new(array[k])
         N.Projection ← new(array[k])
         loop for i = 1 to k
            N.Child[i] ← MakeNode(S_i, N.BindingSet)   /* the array of children nodes */
            N.Projection[i] ← FirstCandidate(N.Child[i])   /* each child has one candidate */
         Return Merge(N.Projection, N.BindingSet)   /* merge the entire array */

                                                                        continued...
```

Figure 10: The anytime ordering algorithm.

Technion - Computer Science Department - Technical Report CIS9812 - 1998

```
NextCandidate(N)
  case (N.NodeType)
    leaf:
      Return ∅  /* an independent set has only one candidate – the first one */
    OR:
      cand ← NextCandidate(N.CurrentChild)
      if cand ≠ ∅  /* another candidate of the same child node */
        then Return Fold(N.CurrentBinder || cand)
        else  /* must try another binder */
          if N.UntriedBinders ≠ ∅
            then
              b ← pop(N.UntriedBinders)
              N.CurrentBinder ← b
              N.CurrentChild ← MakeNode(N.Subgoals\{b}, N.BindingSet∪{b})
              Return Fold(b || FirstCandidate(N.CurrentChild))
            else Return ∅  /* no untried binders remain */
    AND:
      loop for i = 1 to |N.Subgoals|
        N.Projection[i] ← NextCandidate(N.Child[i])
        if N.Projection[i] = ∅  /* no "fresh" candidates */
          then
            N.Projection[i] ← FirstCandidate(N.Child[i])
            Return Merge(N.Projection, N.BindingSet)
      /* if did not leave earlier: */
      Return ∅  /* no more candidates */

MakeNode(S, B)
  N ← new(Node)
  N.Subgoals ← S
  N.BindingSet ← B
  case (S under B)
    independent: N.NodeType ← leaf
    indivisible: N.NodeType ← OR
    divisible: N.NodeType ← AND
  Return N
```

Figure 11: The anytime ordering algorithm – continued.

Technion - Computer Science Department - Technical Report CIS9812 - 1998

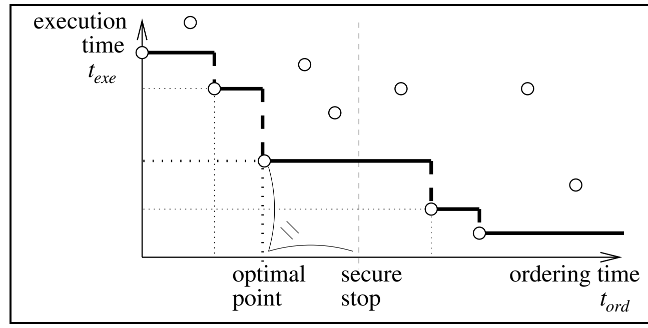


Figure 12: A resource-investment function.

Some improvements can be made to this algorithm for greater efficiency (we did not include them into the notation of Algorithm 2 for the sake of simplicity):

1. When we perform merging in an AND-node, several projections remain unchanged from the last merging. Since cn values of their blocks do not change, the interleaving of those projections cannot change. We can save time by not computing anew the interleaving of these unchanged projections, and just taking it from the last merging.

2. After a node produces all its candidates once, it is often asked to produce them again (for merging with other candidates on higher levels). To avoid this multiple work, we can store the candidate set after we produce it for the first time, and afterwards use the list of stored candidates whenever we need them.

4.1.1 Using Resource-Investment Functions For Termination Control

The only undefined part of the algorithm is the termination condition which dictates when the algorithm should stop exploring new candidates and return the currently best candidate. Note that if this condition is removed or is always false then all candidates are created, and the anytime version becomes equivalent to the original Divide-and-conquer algorithm (Algorithm 1).

The more time we dedicate to ordering, the more candidates we create, and the cheaper becomes the best current candidate. This functional dependence can be expressed by a *resource-investment function* (*RIF* for shorthand).

An example of a RIF is shown in Figure 12: the x-axis (t_{ord}) corresponds to the estimated ordering time spent, small circles depict the candidate orderings found, and the y-axis (t_{exe}) shows the estimated execution time of the cheapest candidate seen till now (the execution time includes the inference time of the rule itself, and all the inference and ordering time that will be spent on the lower levels of the proof tree).

For each point (candidate) on the RIF, we can define the total computation time which is the sum of the ordering time it takes us to reach this point plus the estimated time it takes to execute the currently cheapest candidate:

$$t_{all} = t_{ord} + t_{exe} \tag{3}$$

Technion - Computer Science Department - Technical Report CIS9812 - 1998

There exists a point (candidate) for which this sum is minimal ("optimal point" in Figure 12). This is the best point to terminate the anytime ordering algorithm: if we stop earlier, execution time of the chosen candidate will be large, and if we stop later, the ordering time will be large. In each case the total time spent on this rule will increase. Thus, the termination condition of our anytime algorithm can be expressed as follows:

$$\text{TerminationCondition} :: \text{OptimalPoint(candidate)}$$

But how can we know that the current point on the RIF is optimal? We have only seen the points to the left of it, and cannot predict how the RIF will behave further. If we continue to explore the current RIF until we see all the candidates (then we surely can detect the optimal point), all the advantages of an early stop are lost. We cannot just stop at the point where t_{all} starts to grow, since there may be several local minima of t_{all}.

However, there is a condition that guarantees that optimal point cannot occur in the future. If the current ordering time t_{ord} becomes greater than the currently minimal t_{all}, there is no way that the optimal point will be found in later stage: since t_{ord} can only grow thereafter, and t_{exe} is positive, t_{all} cannot become smaller than the current minimum. So using this termination condition guarantees that the current known minimum is the global minimum. It also guarantees that, if we stop at this point, the total execution time will be:

$$t_{ord}^{opt} + 2 \cdot t_{exe}^{opt}$$

where $(t_{ord}^{opt}, t_{exe}^{opt})$ are the coordinates of the point with minimal t_{all} (the "optimal point" in Figure 12). Now the total time is less than twice the optimal one.

We can maintain a variable t_{all}^{opt} and update it after the cost of a new candidate is computed. The termination condition therefore becomes:

$$\text{TerminationCondition} :: t_{ord} \geq t_{all}^{opt} \tag{4}$$

where t_{ord} is the time spent in the current ordering. The point where these two value become equal is shown as the "secure stop" point in Figure 12.

Although the secure-point-stop strategy promises us that no more than twice the minimal effort is spent, we would surely prefer to stop at the optimal point. A possible solution is to *learn* the optimal point, basing on RIFs produced on the previous orderings of this rule. The RIF learning is performed in parallel with the learning of control values. We learn and store a RIF for each rule and for each binding pattern of the rule head, as a set of (t_{ord}, t_{exe}) pairs accumulated during training. Instead of binding patterns we can use classification trees, where attributes are rule head arguments, and tests are the same as for regression trees that learn control knowledge. Before we start ordering a rule, we use the learned tree to estimate the optimal stop point for this rule. Assume that this point is at time t_{opt}. Then the termination condition for the anytime algorithm is

$$\text{TerminationCondition} :: t_{ord} \geq t_{opt} \tag{5}$$

where t_{ord} is the time spent in the current ordering.

One problem with both formulas 4 and 5 is that they deal with *CPU time*. It is much more convenient and safe to work with discrete measurements. In the same way as we introduced *cost units* as a discrete analogue of the execution time (e.g., the number of unifications

Technion - Computer Science Department - Technical Report CIS9812 - 1998

Ordering Method	Unifications	Ordering Units	Ordering Time	Inference Time	Total Time	Ord.Time Reductions
complete ordering	2467.95	12817.29	1.668	1.039	2.707	0.001931
current RIF	2338.15	2586.66	0.686	0.999	1.685	0.000833
learned RIFs	2340.37	1279.63	0.569	1.027	1.595	0.000690

Table 2: Comparison of the uncontrolled and controlled ordering methods.

performed), we can define *ordering units* that reflect the number of steps performed by the ordering procedure (e.g., the number of nodes visited or candidates created). In addition, we cannot know exactly how long it will take us to execute a candidate ordering – we can only estimate its cost according to our control knowledge (and Equation 1). Therefore, we need two transformation coefficients:

- one to translate the number of ordering units (from the start of ordering and until now) into estimated ordering time that passed,
- one to translate estimated cost of an ordering into estimated execution time.

These two values can be learned from experience. We can assume that for a given program and a given query distribution these coefficients do not change drastically in the course of execution.

4.1.2 Experimentation

We have implemented Algorithm 2 with both terminal conditions 4 and 5. In the following experiments, the ordering method which uses the former definition of the terminal condition (Equation 4) is called the *current-RIF* method, since it makes its decisions based on the current RIF that was seen till now. The second method, which used the terminal condition in latter form (Equation 5), is called the *learned-RIF* method.

We first repeat the first experiment of Section 3 with the following three conditions:

1. No limitations are set on the ordering algorithm (termination condition is always false).
2. *Current-RIF*: The ordering algorithm stops at the secure stop point (termination condition defined by Equation 4).
3. *Learned-RIF*: The ordering algorithm stops at the estimated optimal point (termination condition defined by Equation 5).

The results are shown in Table 2. As we see, the tree sizes did not change significantly. But the number of ordering units is decreased drastically (divided by 5 if we explore current RIF, and by 10 if we use learned RIFs). The average time of ordering one rule (the right-most column of the table) is also strongly reduced, which shows that much less ordering is performed, and this does not lead to worse ordering results (large tree sizes).

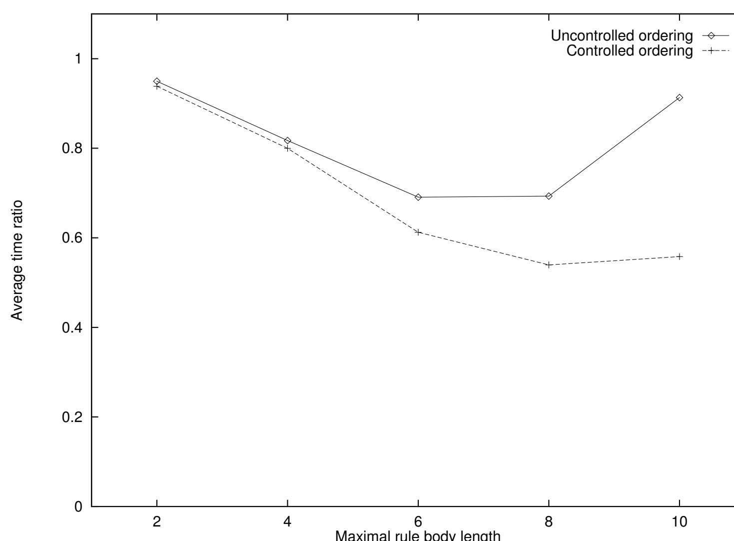


Figure 13: The graph of average utility for various maximal rule body lengths.

Now we repeat the second experiment described in Section 3, distributing domains by their maximal body lengths, and computing the average utility of the ordering algorithm separately for each maximal body length. The upper graph in Figure 13 is the same as in Figure 9 from Section 3, where we performed complete ordering. The new graph is shown in dotted line, and corresponds to ordering with anytime algorithm (using learned RIFs for estimating the optimal stop point). We can see that using the resource-sensitive ordering algorithm reduced the utility problem by controlling the costs of ordering long rules.

4.2 Caching Generalized Control Decisions

In this subsection we present another method of controlling the control procedure, different from that described in the previous subsection. The idea is *to cache* the control decisions made in the past (in the case of the subgoal ordering problem, this means caching the orderings that were made in the past for the given rule body). Any time we must make a new control decision, we first look in the cache, and if the result stored there is reliable enough, we use it; otherwise, we invoke the usual control procedure. Note that the treatment of cache is not performed by the control procedure, but by the "control of control" module.

Evidently, the control decisions must be generalized in some way, when cached: using only exact matches would lead to large memory expenses and frequent failures (exactly as it is with the accumulation of control values – see discussion in Section 2.2). Another important decision that must be made is the degree of reliability of the cached information.

4.2.1 Classification Trees With Orderings As Classes

We now apply the methodology described above to the subgoal ordering problem. With different arguments in the rule head, the subgoals of the rule body can have different control

Technion - Computer Science Department - Technical Report CIS9812 - 1998

Technion - Computer Science Department - Technical Report CIS9812 - 1998

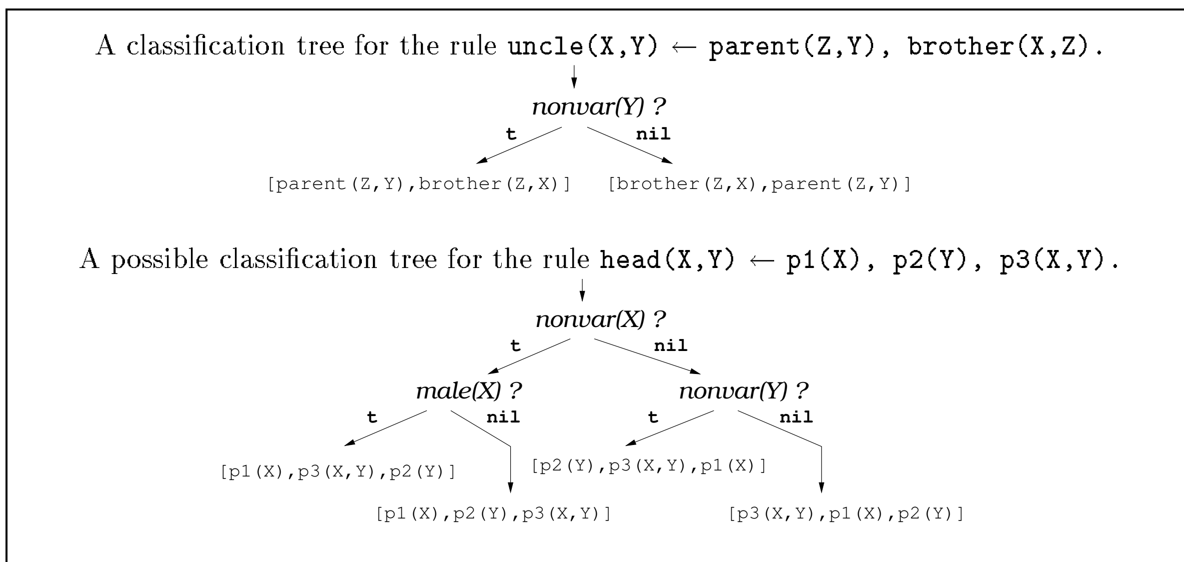


Figure 14: Examples of classification trees that learn rule body orderings.

values, and thus different orderings can be minimal. So, there is a dependence between the values of the arguments in the rule head and the minimal ordering of the rule body, and we can *learn* such dependences – for example, by *classification trees* (Quinlan, 1986). For each rule we store a classification tree. The classification attributes are the arguments of the rule head, since they are the only objects whose values can be known on the ordering stage, before the rule body is executed. The classes are the different orderings. If there are n subgoals in the rule body, $n!$ orderings (classes) are possible, but most likely only few will appear in the tree. We can use the same tests on the arguments as in the regression trees for cost computations (Section 2.2). Two examples of classification trees appear in Figure 14.

A classification tree for a rule is built in the standard way: given a set of examples of past orderings of this rule, we select a test that splits the classes (orderings) better (with smaller error, or larger information gain). Then we create the children recursively, splitting the examples according to the chosen test. If the node has zero variance (all instances belong to the same class – all goals have the same ordering), or if no test can split the examples sufficiently well, we leave this node to be a leaf. Every node has its "representative" ordering – this value will be returned as the classification answer if the tree search stops in this node. In our experiments, we defined the representative ordering of a node as its modal (most frequently repeated) instance, and computed the error of a test as the fraction of instances that disagree with the representative (so the error is always between 0 and 1). An interesting question for further research concerns selecting the representative ordering of multi-class nodes. For example, it could be defined as the "average" ordering of the class (then we must define a distance measure for orderings).

When using a classification tree, we perform recursive descent according to test outcomes. The descent starts at the root and terminates at a leaf node. If the test error in the

Technion - Computer Science Department - Technical Report CIS9812 - 1998

Ordering Method	Unifications	Ordering Time	Inference Time	Total Time	Ord.Time Reductions
complete ordering	2467.95	1.668	1.039	2.707	0.001931
current RIF	2338.15	0.686	0.999	1.685	0.000833
learned RIFs	2340.37	0.569	1.027	1.595	0.000690
learned orderings	2994.19	0.389	1.213	1.602	0.000375

Table 3: Learning orderings by classification trees, compared with other ordering methods.

leaf does not exceed the maximal permitted error, we can use the representative ordering of this leaf as the cached result; otherwise we must perform the whole ordering process.

By changing the maximal permitted error limit, we can change the degree of our confidence in learned orderings: if this limit is set to 0, we use learned orderings only when the class is uniform, i.e. in the past all 100% instances of this class agreed on the same ordering. If this limit is set to 1.0, we always take the cached orderings, even if the error is large. By increasing the confidence level, we gain in ordering time, but lose in execution time (since more incorrect orderings are accepted).

Note that when we use regression trees for computation of control values (Section 2.2), and perform recursive descent in these trees, some test outcomes may be unknown: the tests are applied to arguments of *rule body subgoals*, and some bindings will be known only in the course of execution. But here we work with *rule head* arguments, whose values are known at the moment when we enter the rule, thus all test outcomes are known.

4.2.2 Experimental Results

The three first rows of Table 3 repeat the results shown on Table 2. Its last row depicts the result of caching orderings. The error limit here was set to 1.0 – that means that every time there was a cached ordering, we used it, even if its error was large. We did not prune classification trees, because we saw that in our domains the produced trees were small.

As we can see from the table, the ordering time of the new method is significantly less than in other methods – we perform very little ordering, thus the ordering time is mostly the time of search in classification trees. But large error limit leads to larger tree sizes, and to an increase in inference time. The overall effect is positive: the learning method with error limit 1.0 performs on our artificial domains better than the simple unrestricted Divide-and-conquer algorithm, and than the anytime algorithm with the current-RIF strategy. The advantage of the learned-RIF method over it is not statistically significant.

Not always the leaf nodes in the classification trees are uniform (with their examples belonging all to one class). It often happens that examples that reach a node are ordered differently, but no available test can divide this set better. When we descend a classification tree and reach a leaf, we can take its representative ordering and use it as the cached result. If the error of the leaf is zero, we can do it safely – even if we now start a complete ordering, it will produce the same result as in the previous times (assuming that control values did

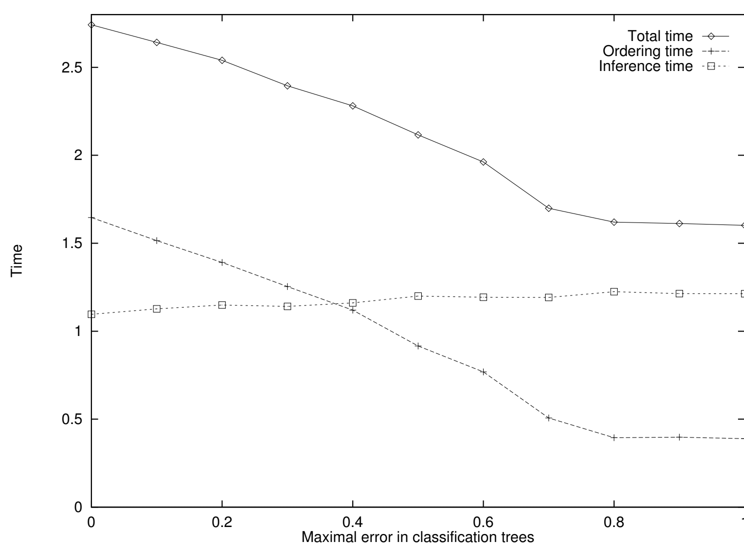


Figure 15: Total, ordering and inference times, depending on the maximal permitted error for classification trees.

not change drastically since then). But if the error is non-zero, doubts arise – whether it is safe to use this ordering, or it is better to start a "normal" ordering process.

If we use classification trees that learn orderings, we can define the maximal permitted error of a tree node, under which we still trust cached orderings. If the current error exceeds this limit (and we cannot descend further from the node), we reject the proposed ordering and invoke the ordering procedure.

Figure 15 shows the experimental results, when the error limit was changed from 0.0 (we use only the 100% secure results) to 1.0 (we use any cached result), by a step of 0.1. The domains and their training and testing sets are the same as in the first experiment of Chapter 3. As can be seen from the graphs, the larger the error bound, the larger is the inference time, and the smaller is ordering time. Both results are intuitively explainable. The more confidence we have in the cached knowledge, the less "new" ordering must be performed – hence the decrease in the number of ordering units. On the other hand, when the error limit grows, we have more chances to select a suboptimal ordering, which leads to larger tree sizes. As we can see from the experimental results, the decrease in ordering time pays off rather well for the increase in inference time. This means that in the given artificial domains it is worth the risk to trust the classification trees more, and to order less.

4.2.3 Using Classification Trees For Program Transformation

It appears that the classification trees, introduced in Section 4.2.1, can find additional use. If there are enough examples, and all the leaves of the classification tree have small error values, then we can include our ordering knowledge into the Prolog program itself, and rewrite its clauses accordingly. For example, the two rules of Figure 14 can be rewritten as

Technion - Computer Science Department - Technical Report CIS9812 - 1998

Technion - Computer Science Department - Technical Report CIS9812 - 1998

Algorithm 3

Rewrite(*tree*)
  Rewrite-with-prefix(**Root**(*tree*), $\emptyset$)

Rewrite-with-prefix(*node, prefix*)
  if **IsLeaf**(*node*)
  then **Output**(*prefix* || **RepresentativeOrdering**(*node*))
  else
    let *test* ← **ClassificationTest**(*node*)
    Rewrite-with-prefix(**LeftChild**(*node*), *prefix* || *test*)
    Rewrite-with-prefix(**RightChild**(*node*), *prefix* || not(*test*))
  Return

Figure 16: Rewriting a rule from a classification tree.

follows:

```
uncle(X,Y) ← nonvar(Y), parent(Z,Y), brother(Z,X).
uncle(X,Y) ← var(Y), brother(Z,X), parent(Z,Y).

head(X,Y) ← nonvar(X), male(X), p1(X), p3(X,Y), p2(Y).
head(X,Y) ← nonvar(X), not(male(X)), p1(X), p2(Y), p3(X,Y).
head(X,Y) ← var(X), nonvar(Y), p2(Y), p3(X,Y), p1(X).
head(X,Y) ← var(X), var(Y), p3(X,Y), p1(X), p2(Y).
```

Essentially, we compose a rule body from a path in the classification tree descending from the root to a leaf, and the representative ordering of this leaf. The complete framework of static ordering can be defined as follows:

1. Training (obtaining control knowledge).
2. Building classification trees.
3. Rewriting rules using classification trees.
4. Executing the rewritten program under the standard left-to-right computation rule.

The process of rewriting a rule is performed by a simple traversal of the classification tree of the rule – see algorithm in Figure 16. It can be regarded as a *program transformation* technique (Pettorossi & Proietti, 1994, 1996). Note that our method has an advantage in that it considers the distribution of queries received by the program, therefore concentrating on relevant cases only. Also, it has error-controlling abilities, since every leaf in a classification tree can estimate the error of its prediction.

Technion - Computer Science Department - Technical Report CIS9812 - 1998

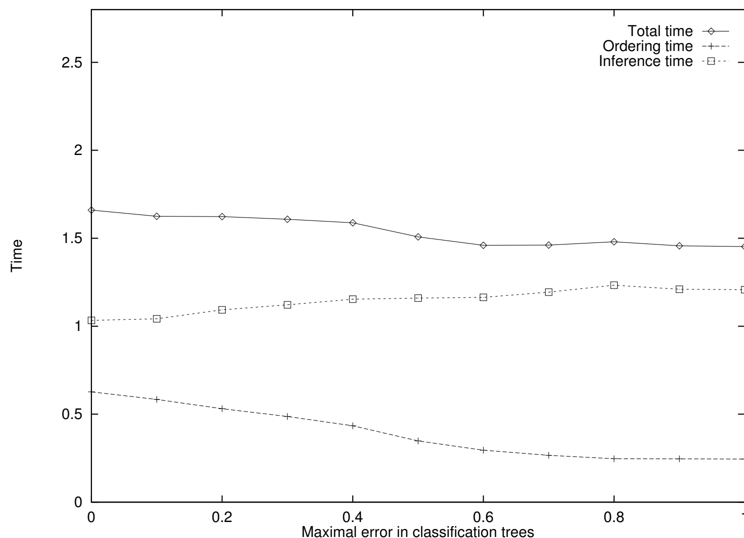


Figure 17: Total, ordering and inference times, depending on the maximal permitted error for classification trees. Anytime ordering was used with the secure-stop termination strategy.

4.3 Combination of Controlling-Control Methods

The anytime algorithm of Section 4.1 can find a good ordering of a rule body rather efficiently. But this ability seems useless when we have a reliable cached result. On the other hand, if we only use cached orderings, we sometimes may chose suboptimal orderings, because the leaf whose representative we took had a large error. In the view of this, the following combination of methods seems promising:

- Look in the cached orderings, and if there is a result with error lower than the current error limit – use it.
- Otherwise (no result, or high error) – invoke the anytime ordering algorithm, using the learned RIFs or exploring the current RIF to control the ordering process.

Figure 17 shows the experimental results of the combined method. Again, the maximal error limit of classification trees ranges from 0.0 to 1.0, and the secure-stop strategy is used to stop the anytime algorithm. One can observe the same tendency: with increasing error limit, the tree sizes (and the inference time) increase, and the ordering time decreases. The total time also decreases. A very similar graph can be drawn if we use learned RIFs to stop the anytime algorithm. Figure 18 plots only the graphs of the total time for the three methods:

1. Using cached orderings, and performing complete ordering when no cached result is available.
2. Using cached orderings, and exploring the current RIF when no cached result is available.

Technion - Computer Science Department - Technical Report CIS9812 - 1998

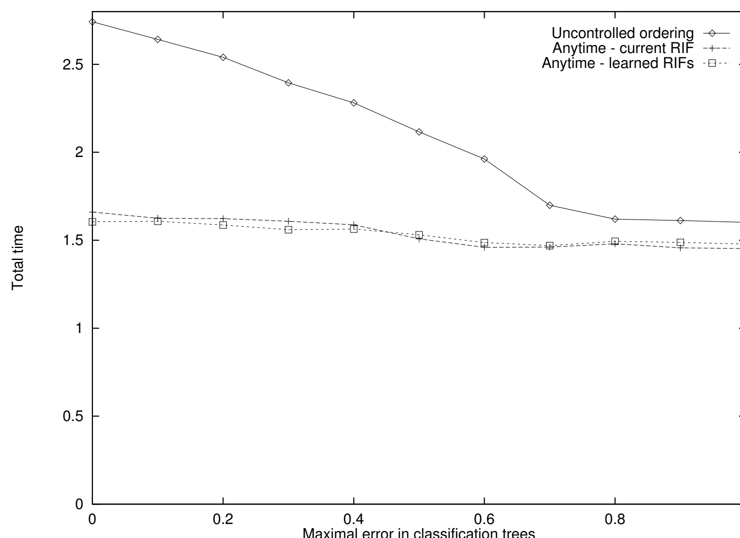


Figure 18: Total time for the three methods, depending on the maximal permitted error for classification trees.

3. Using cached orderings, and exploring learned RIFs when no cached result is available.

As one can see from Figure 18, there is little difference between the two anytime methods: any one of them reduces the time significantly, when the maximal permitted error is small. If the error limit is large, very little ordering is performed (most results are taken directly from classification trees), and the effect of anytime ordering is minor.

By combining cached orderings with anytime algorithm, we have succeeded to curtail the ordering time by a factor of 6, which leads to decrease in total time by a factor of 1.8.

5. Conclusion

In this paper we propose a methodology for dealing with the utility problem in the case when the complexity of the control procedure that utilizes the learned knowledge is very high regardless of the particular knowledge acquired. We propose to augment the control procedure by a "control of control" (C^2) module which will supervise the execution of the control procedure. One possibility is to stop the execution when it becomes wasteful. For this purpose, the control procedure can be converted into an anytime procedure, and the C^2 module performs explicit reasoning about the utility of investing additional control time. This reasoning uses the *resource investment function* of the control process – such functions can be either learned, or built during execution. Another possibility of "controlling control" is to invoke the control procedure selectively – for example, by caching control decisions that were made in the past. Of course, these two methods do not exhaust the specter of all possibilities, but a profound exploration of all the specter was not the goal of this work.

We show an application of the proposed methodology to a learning system for speeding up logic inference. The system orders sets of subgoals for increased inference efficiency. The costs of the ordering process, however, may exceed its potential gains. We describe a way

Technion - Computer Science Department - Technical Report CIS9812 - 1998

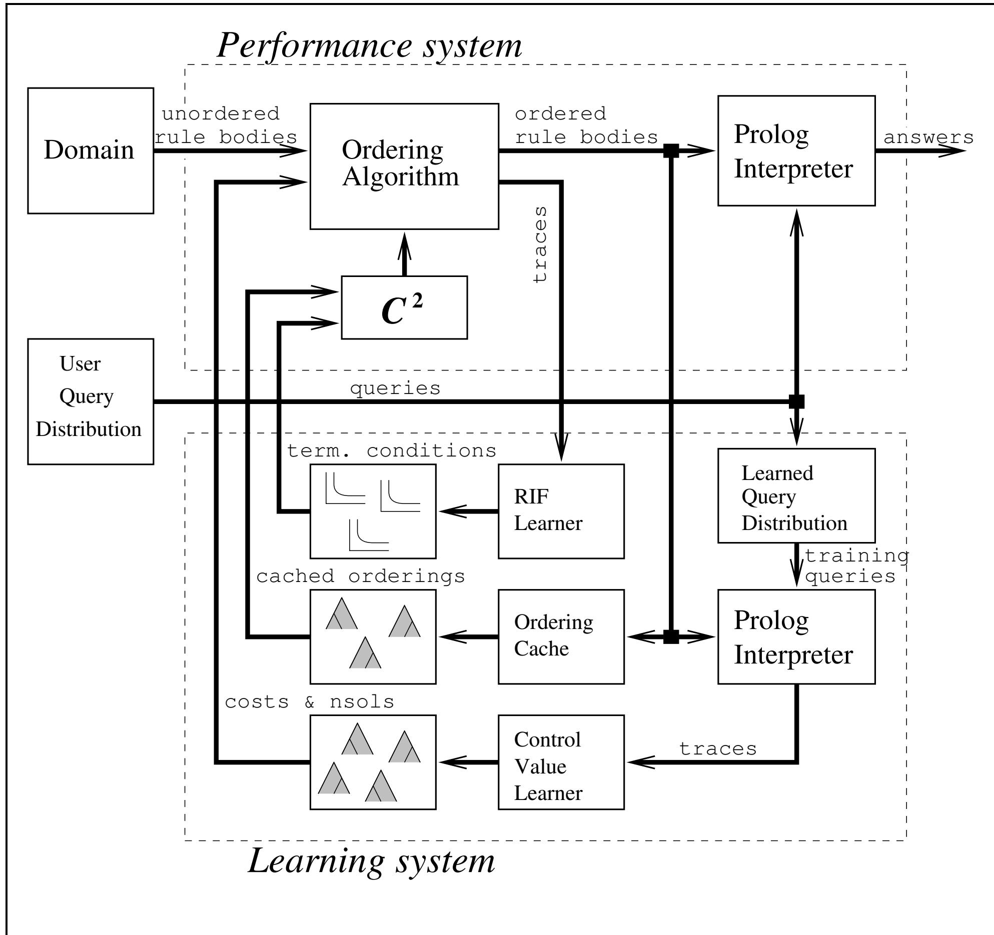


Figure 19: The revised architecture of the LASSY1 system.

to convert the ordering procedure to be anytime. We then show how to reason about the utility of ordering using a resource investment function during the execution of the ordering procedure. We also show how the control (ordering) decisions can be cached, and how the two methods (anytime ordering and caching) can be combined.

Figure 19 shows the revised architecture of the LASSY1 system (the initial architecture was shown in Figure 1 of Section 2). We have added the C^2 module, supervising over the control procedure. The learning system now stores three separate kinds of control knowledge: one of them is the control values used by the ordering algorithm, the other two are used by the C^2 module: the termination conditions of RIFs, and the cached orderings.

The methodology described here can be used also for other domains such as planning. There, we want to optimize the total time spent for planning and execution of the plan. Learning resource investment functions in a way similar to the one described here may increase the efficiency of the planning process.

Technion - Computer Science Department - Technical Report CIS9812 - 1998

References

Boddy, M. (1991). Anytime problem solving using dynamic programming. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, Vol. II, pp. 738–743 Menlo Park. AAAI Press/MIT Press.

Boddy, M., & Dean, T. (1989). Solving time-dependent planning problems. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 979–984 Los Altos, CA. Morgan Kaufmann.

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group.

Debray, S. K., & Warren, D. S. (1988). Automatic mode inference for logic programs. *The Journal of Logic Programming*, *5*, 207–229.

Etzioni, O., & Etzioni, R. (1992). Statistical methods for analyzing speedup learning experiments. Research note, Department of Computer Science, University of Washington.

Gratch, J., & DeJong, D. (1992). COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pp. 235–240 San Jose, California. American Association for Artificial Intelligence.

Ledeniov, O., & Markovitch, S. (1998). The divide-and-conquer subgoal-ordering algorithm for speeding up logic inference. *Journal of Artificial Intelligence Research*, *?* to appear.

Lloyd, J. W. (1984). *Foundations of Logic Programming*. Springer-Verlag, Berlin.

Markovitch, S., & Scott, P. D. (1989). Automatic ordering of subgoals – a machine learning approach. In *Proceedings of North American Conference on Logic Programming*, pp. 224–240 Ithaca, NY.

Markovitch, S., & Scott, P. D. (1993a). Information filtering: Selection mechanisms in learning systems. *Machine Learning*, *10*(2), 113–151.

Markovitch, S., & Scott, P. D. (1993b). Information filtering: Selection mechanisms in learning systems. *Machine Learning*, *10*, 113–151.

Minton, S. (1988). *Learning Search Control Knowledge: An Explanation-Based Approach*. Kluwer, Boston, MA.

Naish, L. (1985). Automatic control for logic programs. *The Journal of Logic Programming*, *3*, 167–183.

Nie, X., & Plaisted, D. A. (1990). Experimental results on subgoal ordering. In *IEEE Transactions On Computers*, Vol. 39, pp. 845–848.

Pettorossi, A., & Proietti, M. (1994). Transformation of logic programs: Foundations and techniques. *The Journal of Logic Programming*, *19 & 20*, 261–320.

Pettorossi, A., & Proietti, M. (1996). Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, *28*(2), 360–414.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

Smith, D. E., & Genesereth, M. R. (1985). Ordering conjunctive queries. *Artificial Intelligence*, *26*, 171–215.

Tadepalli, P., & Natarajan, B. K. (1996). A formal framework for speedup learning from problems and solutions. *Journal of Artificial Intelligence Research*, *4*, 419–443.

Tambe, M., Newell, A., & Rosenbloom, P. (1990). The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning*, *5*(3), 299–348.

Ullman, J. D., & Vardi, M. Y. (1988). The complexity of ordering subgoals. In *Proceedings of the Seventh ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pp. 74–81 Austin, TX.

Warren, D. H. D. (1981). Efficient processing of interactive relational database queries expressed in logic. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pp. 272–281. IEEE Computer Society Press.

Technion - Computer Science Department - Technical Report CIS9812 - 1998