Learning Investment Functions for Controlling the Utility of Control Knowledge

Oleg Ledeniov and Shaul Markovitch
Computer Science Department
Technion, Haifa, Israel
{olleg,shaulm}@cs.technion.ac.il

Abstract

The utility problem occurs when the cost of the acquired knowledge outweighs its benefits. 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 costs 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. In this work we deal with a control mechanism that carries very high cost regardless of the particular knowledge acquired. We propose to use in such cases explicit reasoning about the economy of the control process. The solution includes three steps. First, the control procedure must be converted to anytime procedure. Second, a resource-investment function should be acquired to learn the expected return in speedup time for additional control time. Third, the function is used to determine a stopping condition for the anytime procedure. We have implemented this framework within the context of a program for speeding up logic inference by subgoal ordering. The control procedure utilizes the acquired control knowledge to find efficient subgoal ordering. The cost of ordering, however, may outweigh its benefit. Resource investment functions are used to cut-off ordering when the future net return is estimated to be negative.

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 (Tadepalli & Natarajan 1996). Several works in speedup learning concentrated on acquiring *control* knowledge for controlling the search performed by the problem solver. When the cost of using the acquired control knowledge outweighs its benefits, we face the so called *Utility Problem* (Minton 1988; Markovitch & Scott 1993). 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. Most of the existing solutions for this problem involve *filtering* out control rules that are estimated to be of low utility (Minton 1988; Markovitch & Scott 1993; Gratch & DeJong 1992). Others try to restrict the complexity of the preconditions (Tambe, Newell, & Rosenbloom 1990).

In this work we deal with a different setup where the control procedure has potentially very high complexity regardless of the specific control knowledge acquired. In this setup, the utility problem can become significant since the cost of using the control knowledge can be higher than the time it saves on search. Filtering is not useful for such cases, since deleting control knowledge will not necessarily reduce the complexity of the control process. We propose to use a three step framework for dealing with this problem. First, convert the control procedure into an *anytime* procedure. Then learn the *resource investment function* which predicts the saving in search time for given resources invested in control decision. Finally, run the anytime control procedure minimizing the sum of the control time and the expected search time.

This framework is demonstrated through a learning system for speeding up logic inference. The control procedure is the Divide-and-Conquer (DAC) subgoal ordering algorithm (Ledeniov & Markovitch 1998). The learning system learns costs and number 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 making the DAC algorithm anytime. We then learn a resource investment function for each goal pattern. The functions are used to stop the ordering procedure before its costs become too high. We demonstrate experimentally how the ordering time decreases without harming significantly the quality of the resulting order.

Learning Control Knowledge for Speeding up Logic Inference

In this section we describe our learning system which performs off-line learning of control knowledge for

speeding up logic inference. The problem solver is a Prolog interpreter for pure Prolog. Thus the goal of the learning system is to speed up the SLD-resolution search of the AND-OR tree.

The Control Procedure

The control procedure orders AND nodes of the AND-OR search tree. When the current goal is unified with a rule head, the set of subgoals of the rule body, under the current binding, is given to our DAC algorithm to find a low-cost ordering.

The algorithm produces candidate orderings and estimates their cost using the equation (Smith & Genesereth 1985):

$$\begin{array}{l} Cost(\langle A_1, A_2, \ldots A_n\rangle) = \\ \sum_{i=1}^{n} \sum_{b \in Sols(\{A_1, \ldots A_{i-1}\})} Cost(A_i|_b) = \\ \sum_{i=1}^{n} \left[\left(\prod_{j=1}^{i-1} \overline{nsols}(A_j)|_{\{A_1 \ldots A_{j-1}\}} \right) \right. \\ \left. \times \overline{cost}(A_i)|_{\{A_1 \ldots A_{i-1}\}} \right], \end{array} \quad (1)$$

where $\overline{cost}(A)|_{\mathcal{B}}$ is the *average cost* (number of unifications) of proving a subgoal A under all the solutions of a set of subgoals $\mathcal{B}$, $\overline{nsols}(A)|_{\mathcal{B}}$ is the *average number of solutions* of A under all the solutions of $\mathcal{B}$. For each subgoal A_i, its average cost is multiplied by the total number of solutions of all the preceding subgoals.

The main idea of the DAC algorithm is to create a special AND-OR tree, called the *divisibility tree* (DT), 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 dependency information. We call two subgoals that share a free variable *dependent*. A leaf of the tree represents an independent set of subgoals. An AND node represents a subset that can be partitioned into subsets that are mutually independent and each of the AND branches corresponds to the DT of one of the partitions. An OR node represents a dependent set of subgoals. Each OR branch corresponds to an ordering where one of the subgoals in the subset is placed first. The selected first subgoal binds some of the variables in the remaining subgoals. 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 prove 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. Candidate orderings are propagated up the DT. A candidate of an OR-node is generated from a candidate of *one* of its children, while a candidate of an AND-node is constructed by merging candidates of *all* its children. The last step of the algorithm is to return the lowest cost candidate of the root according to equation 1. In most practical cases the new algorithm works in polynomial time. For more details about the DAC algorithm see (Ledeniov & Markovitch 1998).

The Learning Component

The learning system either performs on-line learning using the user queries, or performs off-line learning by generating training queries based on a distribution learned from past user queries. The ordering algorithm described above 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 the training queries.

Storing control values separately for each literal is not practical, for several 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 thus unknown.

The learner therefore acquires control values for *classes* of literals rather than for separate literals. The more refined are the classes, the smaller is the variance of real control values inside each class, the more precise are the $\overline{cost}$ and $\overline{nsols}$ estimations that the classes assign to their members, and the better orderings we obtain. One easy way to define classes is by *modes* or *binding patterns* (Debray & Warren 1988): for each argument we denote whether it is free or bound.

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 recursive descent in it, starting from the root. Each tree node contains a test which applies to the arguments of the literal. Since we store separate trees for different predicates, we do not need tests that apply to the predicate names. A possible regression tree for estimation of number of solutions for predicate `father` is shown in Figure 1.

The tests used in the nodes can be *syntactic*, such as "is the first argument bound?", or *semantic*, such as "is the first argument male?". 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 require logic inference. For example, the first one of the semantic tests above invokes the predicate `male` on the first argument of the literal. Therefore these tests must be as "cheap" as possible, otherwise the retrieval of control values will take too much time.

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. Most experiments were performed on randomly created

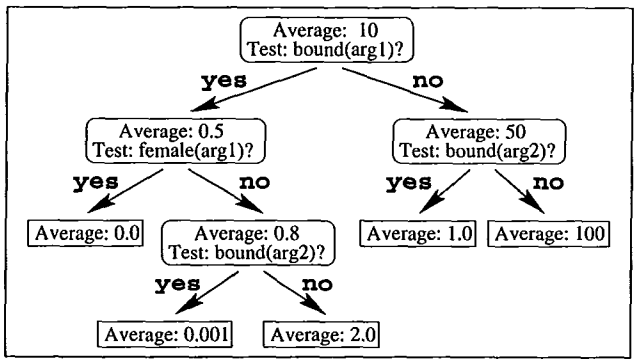

Figure 1: A regression tree for estimation of number of solutions for `father(arg1,arg2)`.

artificial domains. We also tested the performance of the system on several real domains.

Most experiments described below consist of a training session, followed by a testing session. Training and testing sets of queries are randomly drawn from a fixed distribution. During the training session the learner acquires the control knowledge for literal classes. During the testing session 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 (*inference time*).

In order to ensure statistical significance of results 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 fixed 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.
- *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 used 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* DAC *algorithm* using binding patterns for learning.
- *The* DAC *algorithm* using regression trees with syntactic tests.

Table 1 shows the results obtained. The results clearly show the advantage of the DAC 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 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 that is accumulated for a 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 DAC 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 large potential benefit.

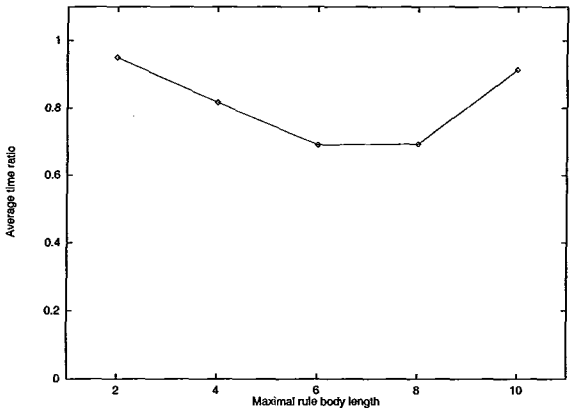

Figure 2: The effect of rule body length on utility.

The graph in Figure 2 plots the average time saving of the ordering algorithm: for each domain we calculate the ratio of its total testing time with the DAC algorithm and with the random method. For each maximal body length, a point on the graph shows the 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
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
DAC - binding patterns	8492.99	2678.72	8.677	2.895	11.571	0.0032
DAC - regression trees	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).

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 DAC ordering algorithm performs only a little better than the static 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 DAC 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 again decreases, and the average time ratio nearly returns to the 1.0 level. Although the tree sizes are now reduced more and more (compared to the sizes of the random method), the additional ordering time grows notably, and the overall effect of ordering becomes almost null.

These results show that risk of encountering the utility problem exists even with our efficient 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.

Controlling the Utilization of Control Knowledge

The last 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.

For cases such as the above, we propose to use a a methodology with the following three steps. First, make the control procedure anytime. Then acquire the resource-investment function that predicts the expected reduction in search time as a result of investing more control time. Finally, execute the anytime control procedure with a termination criterion based on the resource investment function. In this section, we will show how this three-step methodology can be applied to our DAC algorithm.

Anytime Divide-and-Conquer Algorithm

The DAC algorithm propagates the set of all candidate orderings in a bottom-up fashion to the root of the DT. 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, 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 minimal one – update the current cheapest candidate.
- Return the current cheapest candidate.

In the new framework, we do not generate all candidates exhaustively (unless the termination condition never holds). This algorithm is an instance of **anytime algorithm** (Boddy & Dean 1989): 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 DT several times: it finds its first candidate, then pass 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. The algorithm stores for each node some information about the last candidate created. Note that all the nodes of a DT never physically co-exist simultaneously, since in every OR-node only one child is maintained at every moment of time. Thus, if the termination condition occurs in the course of the execution, some OR-branches of the DT are not created.

Using Resource-Investment Functions

The only part of the above algorithm that remains undefined 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 DAC algorithm.

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).

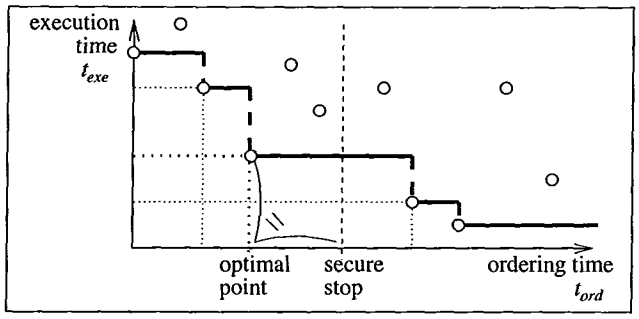

Figure 3: A resource-investment function.

An example of a RIF is shown in Figure 3: the x-axis (t_{ord}) corresponds to the ordering time spent, small circles on the graph are candidate orderings found, and the y-axis (t_{exe}) shows the execution time of the cheapest candidate seen till now.

For each point (candidate) on the RIF, we can define $t_{all} = t_{ord} + t_{exe}$, the total computation time which is the sum of the ordering time it takes us to reach this point and the time it takes to execute the currently cheapest candidate. There exists a point (candidate) for which this sum is minimal ("optimal point" in Figure 3). This is the best point to terminate the anytime ordering algorithm: if we stop before it, execution time of the chosen candidate will be large, and if we stop after it, the ordering time will be large. In each of the two cases the total time spent on this rule will increase.

But how can we know that the current point on the RIF is optimal? We have only seen the points before it (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 can not 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 $t^{opt}_{ord} + 2 \cdot t^{opt}_{exe}$, where $(t^{opt}_{ord}, t^{opt}_{exe})$ are the coordinates of the point with minimal t_{all} (the "optimal point" in Figure 3). Now the total time is less than twice the optimal one.

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

$$\text{TerminationCondition} :: t_{ord} \geq t^{opt}_{all} \qquad (2)$$

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 3.

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

$$\text{TerminationCondition} :: t_{ord} \geq t_{opt} \qquad (3)$$

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

Experimentation

We have implemented the anytime algorithm, with both terminal conditions 2 (the *current-RIF* method) and 3 (the *learned-RIF* method). The results are shown in Table 2. As we see, the tree sizes did not change significantly, but the ordering time decreased. The average time of ordering one rule (the rightmost column of the table) also decreased strongly, which shows that much less ordering is performed, and this does not lead to worse ordering results.

We then repeated the second experiment, distributing domains by their maximal body lengths, and computing the average utility of ordering separately for each maximal body length. The upper graph in Figure 4 is the same as in Figure 2. The new graph for the anytime algorithm with learned RIFs is shown in dotted line. We can see that using the resource-sensitive ordering algorithm reduced the utility problem by controlling the costs of ordering long rules.

Conclusions

This paper presents a technique for dealing with the utility problem when the complexity of the control procedure that utilizes the learned knowledge is very high regardless of the particular knowledge acquired. We propose to convert the control procedure into an anytime algorithm and perform explicit reasoning about

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

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

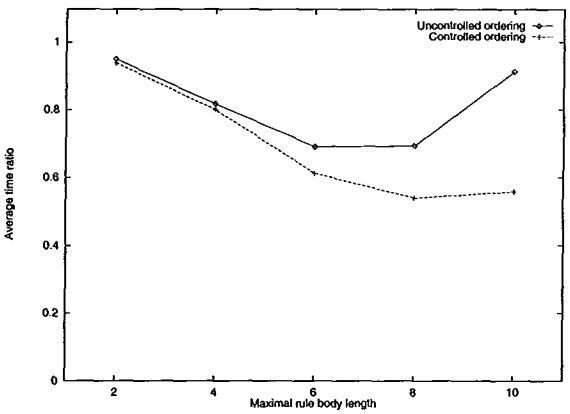

Figure 4: The effect of rule body length on utility.

the utility of investing additional control time. This reasoning uses the *resource investment function* of the control process which can be either learned, or built during execution.

We show an example of this type of reasoning in 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 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 to learn and use resource investment functions. The methodology described here can be used also for other tasks 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.

References

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

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

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

Gratch, J., and 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*, 235–240. San Jose, California: American Association for Artificial Intelligence.

Ledeniov, O., and Markovitch, S. 1998. The divide-and-conquer subgoal-ordering algorithm for speeding up logic inference. Technical Report CIS9804, Technion.

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

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

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

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

Smith, D. E., and Genesereth, M. R. 1985. Ordering conjunctive queries. *Artificial Intelligence* 26:171–215.

Tadepalli, P., and 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.; and Rosenbloom, P. 1990. The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning* 5(3):299–348.