NASA-TM-109251

IN-63-TM
185442
ORGCON
7P

Efficient Dynamic Optimization of Logic Programs

Phil Laird
Artificial Intelligence Research Branch
MS 269-2
NASA Ames Research Center
Moffett Field, CA 94035

(NASA-TM-109251) EFFICIENT DYNAMIC OPTIMIZATION OF LOGIC PROGRAMS (NASA) 7 p

N94-12889

Unclas

G3/63 0185442

NASA Ames Research Center

Artificial Intelligence Research Branch

Technical Report FIA-92-20

June, 1992

To appear in Workshop on Knowledge Compilation and Speedup Learning, July 1992.

Efficient Dynamic Optimization of Logic Programs

Philip Laird[*]
AI Research Branch
M.S. 269-2
NASA Ames Research Center
Moffett Field, California 94035, U.S.A.

Abstract

A summary is given of the dynamic-optimization approach to speedup learning for logic programs. The problem is to restructure a recursive program into an equivalent program whose expected performance is optimal for an unknown but fixed population of problem instances. We define the term "optimal" relative to the source of input instances and sketch an algorithm that can come within a logarithmic factor of optimal with high probability. Finally we show that finding high-utility unfolding operations (such as EBG) can be reduced to clause reordering.

Purpose

This paper presents an outline of the motivation, problem, methods, and results of some recent work on dynamic optimization of programs. An earlier paper (Laird, 1992b) contains details, experimental results, and more complete references to related work. In addition, this paper discusses some new results, particularly the efficient handling of unfolding transformations.

Dynamic Optimization

"Speedup learning" refers generally to the problem of learning to perform more efficiently with practice (a form of skill learning). A particular case of speedup learning, called *dynamic optimization*, is the problem of improving the average-case performance of a program without affecting its the correctness. Unlike static optimization, dynamic optimization requires sample runs or other experience with the distribution of problem instances to be solved.

One approach—known variously as *memoization* or *caching*—is to retain results of solved problems for subsequent reuse in order to reduce the amount of redundant computation. Explanation-based generalization is a familiar example of this method. Another approach is to formulate and refine useful rules about a search process. SOAR and PRODIGY have successfully exploited this idea.

The approach taken here, however, is different: the structure of the program is modified in order to improve the average-case performance; and instead of making a priori assumptions about the "average case", we learn what we need to know about it by statistical sampling.

As a generic task, automated program speedup has great potential for commercial return. Data processing programs are very complex because they must handle correctly every contingency, but most of these contingencies, occur rarely, if ever, during the lifetime of the program. Restructuring the program so that it runs fastest on the kinds of problems that it encounters most often is a sensible approach, one that is best accomplished by automation: not only is restructuring code a difficult and error-prone process, humans users can seldom supply more than qualitative understanding of the statistical properties of the data that the program will be processing.

Formal Problem Definition

The goal of this work is to find practical optimization methods, not just to prove abstract or asymptotic properties of the problem. Still, a formal statement of the dynamic-optimization problem is useful.

Fix a computational language $\mathcal{L}$. Let τ be a family of correctness-preserving program transformations for programs over $\mathcal{L}$. We are given as input three things: (1) a program P in $\mathcal{L}$; (2) an unknown stochastic process $\mathcal{S}$ that can be invoked to generate problem instances for the program; and (3) a computable cost function $\mathcal{C}$ that assigns a real-valued cost to any computation.[1] In cases where there may be multiple

[*]Email: laird@pluto.arc.nasa.gov

[1]We also need some reasonability conditions on the cost

solutions, the cost of running a program on an input instance is taken to be the cost of finding the *first* solution. The task is to find a program P' such that (1) P' is equivalent to P in the sense that there is a finite sequence of transformations in τ that maps P into P', and (2) P' is optimal with respect to $\mathcal{C}$ and $\mathcal{S}$, i.e., the expected cost of solving problems drawn from $\mathcal{S}$ is minimal (as measured by $\mathcal{C}$) among all programs equivalent to P.

For concreteness we apply our work to the Prolog language[2] with three transformations (defined below): predicate unrolling, clause reordering, and unfolding of pairs of clauses. The cost functions I used in the experiments were CPU time and and the number of atomic unifications. The problem instances have been independently selected with replacement from some fixed distribution, often with unbounded support (i.e., the number of possible problems is infinite).

The transformations we admit are as follows: (see Figure 1)

- (Unrolling) Copy all clauses of a predicate p and assign the predicate a new name, e.g., pcopy. Some references to the old predicate may be changed to the new name. In practice, we shall change all references in the tails of the unrolled clauses to refer to the new name.
- (Reordering) Reorder the clauses of a particular predicate.
- (Unfolding) Unfold two clauses C_1 and C_2 by resolving a premise goal from C_2 with the head of C_1. The result is a new clause to be added to the program.[3]

These are not the only possible semantics-preserving transformations for logic programs, but they are sufficient to obtain good results in practice and simple enough to understand mathematically.

Properly, the dynamic optimization problem defined above is ill-posed if one can construct a Prolog program and an input source such that there exists an infinite sequence of equivalent programs each of which has lower expected cost than its predecessor. In fact one can do just this. But in practice this technicality is removable by requiring only that we construct a program whose expected cost is, with high probability, within ϵ of optimum, for arbitrary $\epsilon > 0$.

More than a technicality is the fact that the (corresponding decision) problem is NP-hard. Even if we restrict the transformations to clause reordering, the problem of determining whether or not there is a reordering such that the expected cost is no greater than C is NP-complete, as demonstrated by a reduction from the minimal set cover problem. We are, therefore, unlikely to find a polynomial time solution to the problem, but we are certainly free to look for a polynomial-time approximation algorithm with some performance guarantee. Indeed, this is our approach.

Note:In the programs below only the predicates are shown, without their arguments—e.g., p instead of p(...).

(a) Initial program:

```
[C1]:  p <- p.
[C2]:  p.
```

(b) After an unrolling: (Clauses C_3 and C_4 are copies of C_1 and C_2, resp., but with the predicate renamed.

```
[C'1]:  p <- pcopy.
[C2]:   p.
[C3]:   pcopy <- pcopy.
[C4]:   pcopy.
```

(c) After reordering the pcopy predicate:

```
[C'1]:  p <- pcopy.
[C2]:   p.
[C4]:   pcopy.
[C3]:   pcopy <- pcopy.
```

(d) Let θ be a unifier for the underscored literals above; after unfolding C'_1 through C_4, we obtain:

```
[C'1.1]:  pθ.
[C2]:     p.
[C'1.2]   p <- pcopyrest.
[C4]:     pcopy.
[C3]:     pcopy <- pcopy.
[C3.1]:   pcopyrest <- pcopy.
```

Here, $C_{3.1}$ is a copy of C_3; note that the case C_4 is covered by the unfolded clause $C'_{1.1}$.

Figure 1: Basic Prolog transformations

The Learn/Optimize Cycle

Our main point of departure from the caching approach is to separate the learning phase from the program transformation phase. A similar approach has been employed by Greiner and Orponen (1991) for non-recursive query languages. During the learning phase one collects statistics about the probabilities and costs of success and failure of specific clauses at specific points (contexts) in the proof. The transformation phase uses these statistics to select transformations that are likely to improve performance. Applying these transformations results in a revised program

function, e.g., if one computation extends another, the cost increases. As a minimum the cost should be a Blum measure.

[2]In this paper we treat only "pure" Prolog, but the techniques extend to most impure constructs as well.

[3]The standard EBG algorithm unfolds the program through all the resolution steps used in solving a single problem instance (4).

P' equivalent to the original but with lower expected cost for the same source of problem instances.

Ours is not a one-shot learning and transforming process, however: we repeat this learn/optimize process starting with P', deriving yet another optimized program P'', and so forth. To see why this is necessary, imagine that we subject the program in Figure 1(b) to the learn cycle and thereupon decide to reorder the clauses C_1' and C_2. As a result of this change the expected costs and probabilities for C_3 and C_4 will also change; hence any decision about the optimal order of C_3 and C_4 should be deferred until C_1' and C_2 have been reordered and the statistics revised.

In general, the cycle of learning and optimizing is repeated, with transformations occurring at successively deeper levels of the program, until some kind of convergence is achieved. Unrolling is followed by reordering or unfolding in order to effect optimizations at specific points in the computation. As the program is unrolled, the total number of clauses increases, but in practice the physical size of the program is a negligible part of the run time. Note that unrolling has the effect of separating out a finite, non-recursive initial part of the computation from the later, recursive parts. For example:

- In the transformation from Figure 1(a) to 1(b), the initial call (to p) is distinguished from the recursive calls to pcopy. Thus our learning phase can gather different statistics for the initial program call to p and for the subsequent recursive calls to pcopy.
- If, in Figure 1(a) the subgoal p from clause C_1 is solved more efficiently using a different clause order (C_2 before C_1) from that of the main goal p, then the program will be unrolled as in Figure 1(b) and reordered as in Figure 1(c).

An unfolding transformation may increase by one the number of clauses in a procedure (e.g., p in Figure 1(c-d)). The "catchall" clause ($C_{1.2}'$) invokes a different version of the unfolded predicate, one that omits the clause that was unfolded (here, C_4 is omitted in `pcopyrest`).

Unrolling the program in this way is another distinguishing feature of our approach. A reasonable question is: instead of unrolling, why not just optimize the program by reordering the existing clauses and unfolding some of them for some number of steps? The reason is that advice obtained early in a long search has exponentially greater benefit than advice obtained later, since more of the search space will be avoided. After initial experiments in which no unrolling was done, I found that the additional steps of unrolling and optimizing the initial calls in the program provided substantially greater improvement.

To summarize, the learn/optimize cycle gradually unrolls the program and optimizes it for calls first at depth 0, then at depth 1, and so on. If on some cycle no optimizations can be found and depth d, the cycle still continues by optimizing depth $d+1$. The procedure halts, not because it runs out of transformations, but because of finite limits on the accuracy of the learning algorithm. At this point the final optimization is to reorder the clauses of the recursive procedures (without unrolling).

Below, we discuss several specific issues concerning the learn/optimize cycle.

Well-foundedness

A problem can arise when some some of the clauses of a procedure overlap in the cases that they cover, i.e., more than one clause can successfully resolve certain goals. Suppose, for example, that in Figure 1(b) C_3 and C_4 each cover almost all provable instances of the pcopy predicate, with equal expected cost. Then with the clause order shown, we will charge C_3 with almost all the cost of solving pcopy goals. C_3 thus appears to be more expensive than C_4, and the transformation phase will therefore place C_4 before C_3 in the revised program. But after reordering and repeating the learning phase, C_4 is now charged for the cost of pcopy, and the original order appears preferable. By continuing to reverse the order of these two clauses we could get stuck in a loop that never improves the program. As noted above, finding an optimal ordering of a set of clauses is NP-complete, so no exact algorithm is likely to be significantly faster than testing every possible ordering.

A simple solution to this problem is that once the clause order of a predicate is chosen, we should never change it again, even if subsequent learn/optimize cycles make a different order seem better. It turns out that this "greedy" procedure for clause reordering yields an approximation whose expected cost is within a logarithmic factor of optimal

What about unfolding transformations? For these there is no inverse folding transformation in the admissible set, so once carried out they cannot be undone. However the order of the unfolded clauses (e.g., $C_{1.1}'$, $C_{1.2}'$, and C_2 in Figure 1(d)) may be changed on the next cycle. Below we shall show that the question of whether the statistics can mislead us into performing a sub-optimal unfolding transformation can be finessed, and the search for unfolding transformations reduces to one of finding good clause orderings.

Convergence

As discussed above, it may happen that as a result of a transformation the measured expected cost of the program actually *increases* on the next cycle; even so, we continue the cycle even after a temporary increase, because transformations on subsequent cycles can reduce the mean cost of the program substantially. Consequently, this is *not* a hill-climbing procedure.

If the learning algorithm were to continue to unroll clauses and collect statistics on them to arbitrary accuracy, the learn/optimize cycle might never halt: ever deeper transformations might be found that continue reducing the expected cost of the program, ultimately by very small amounts. In practice, however, it is the learning algorithm, specifically its finite sample-size limits, that bounds the cycle. The deeper one goes in the search tree, the lower the probability that the nodes will be encountered on any given problem, and hence the larger will be the number of instances that must be solved in order to estimate the likelihoods and costs accurately.

Slightly more formally, one can characterize the program to which the process converges by defining a program to be d-optimized if all its calls at depths d and below are optimal. Then if the learning algorithm provides correct statistics (in the PAC sense), the optimized program will be (probably approximately) d-optimized to some depth d.

Order of Transformations

Suppose that for a subgoal at some point in the program, an unfolding transformation *and* a reordering tranformation would each be effective, according to the results of the most recent learning phase. Which transformation(s) should we perform? My intuition tells me that the clauses should be reordered before any of them should be unfolded with subsequent clauses, but initially I had difficulty justifying this fact. Below I'll argue why it is in fact true.

The Learning Algorithm

The statistics collected during the learning phase are driven by the need to predict the efficiency of program transformations during the subsequent phase. Consider the predicate p in Figure 1(b) and the decision about the ordering of the clauses C_1' and C_2. The solution to this problem is well known: we need to estimate for each clause the *a priori* probability p_i that the clause will succeed (independently of whether the other clause leads to a solution) and of the expected cost C_i of applying the clause (regardless of success or failure). Then if we change the program by placing these clauses in decreasing order of p_i/C_i, the ordering of the p clauses is optimal—*provided* no two clauses cover (solve) the same problem instance. If multiple clauses cover some instances, *we still adopt the same ordering*: although this may not be an optimal ordering, it is (as noted above) a greedy approximation to optimal. On subsequent passes of the cycle, the ordering of the clauses C_1' and C_2 will not be altered, so statistics on them need not be collected. Instead, clause C_3 will be unrolled (C_4 has no subgoals to unroll) and statistics p_i and C_i will be collected for clauses C_3 and C_4.

Efficient statistical methods for estimating p and C for a clause to within any given accuracy and confidence are well known, and a sufficient sample size is easily computed. For some clauses, p may be fall below the accuracy limit and thus be estimated as zero: such clauses are not optimized further. Typically the deeper one gets in the program (i.e., the more unrolling transformations have been performed), the smaller the likelihood that any particular problem instance will invoke that clause, and the larger the sample size required to collect the needed statistics. Consequently we impose a lower bound on the *absolute* likelihood that a clause will be invoked and refuse to optimize clauses whose likelihood falls below that threshold. It turns out (Laird, 1992b) that this lower bound is not enough to guarantee termination of the cycle: we must also impose an absolute upper bound on the depth of the unrolling. In practice, however, this absolute depth will probably not be reached.

Finally we are left with optimizing the recursive clauses (like **pcopy** in Figure 1(d)). Unlike the clauses that precede it in the computation, such clauses are invoked multiple times. A Markov-tree learning algorithm (called the TDAG algorithm) is used to compute the probabilities of success for these recursive procedures. Unfortunately there is no polynomial time bound on the sample size needed to estimate the probabilities for such clauses, since for Markov processes successive events are not statistically independent. The sample size needed to ensure the accuracy of the statistics is highly dependent upon the structure of the underlying call graph and the transition probabilities determined by the input source. As a practical approach, I have determined this sample size heuristically, with good results.

Finding Unfolding Transformations

We have seen that the necessary statistics and the corresponding procedure for selecting good clause-reordering transformations are straightforward, and that we can quantify the relationship between the optimal order and our approximation to it. What about unfolding transformations? In a previous paper (Laird, 1992a) I outlined a rather complicated algorithm for finding unfolding transformations, with special statistics collected for the purpose during the learning phase. But it turns out that *virtually the same algorithm as for reordering transformations applies.* In fact, unfolding can be viewed as a special case of reordering.

The simple program in Figure 1(c) represents the SLD search procedure for resolving goals with predicate p. In Figure 2 a portion of the depth-first search tree for this program is shown explicitly. The SLD-resolution procedure searches this tree in pre-order left-to-right.

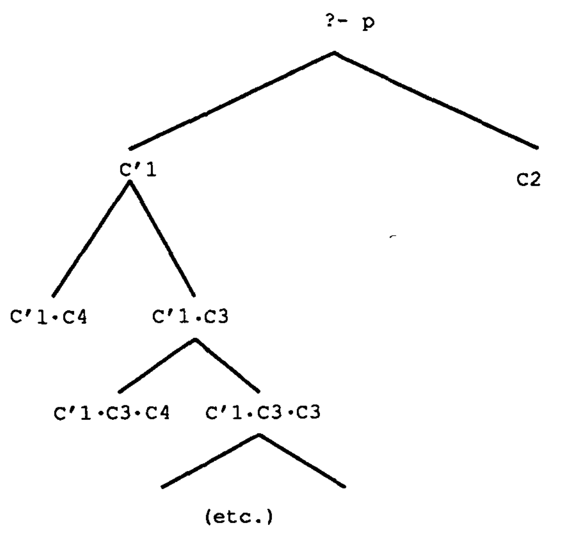

Figure 2: Search tree for the program of Fig. 1(c).

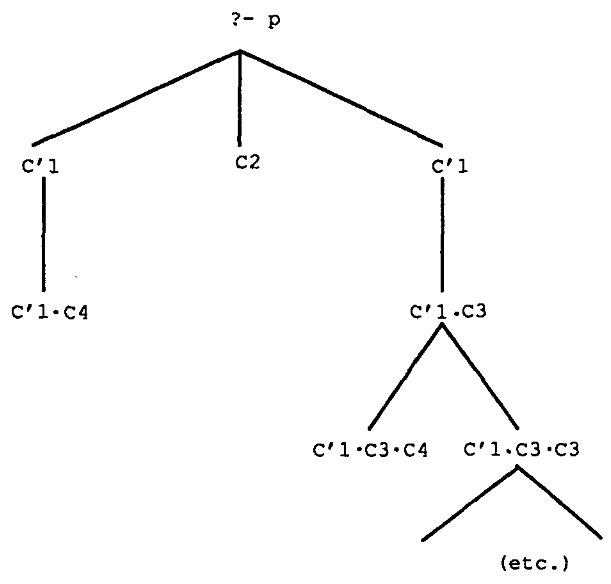

Figure 3: Reordered Search tree.

```
[C'1C4]:   pθ.
[C2]:      p.
[C'1C3]:   pφ <- pcopyφ.
[C3]:      pcopy <- pcopy.
[C4]:      pcopy.
```

Figure 4: The program of Fig. 1(b) reordered to depth two, according to Fig. 3. The substitution θ is that shown in Fig. 1(d), unifying the **pcopy** consequent of clause C_1' with the head of C_4. The substitution ϕ unifies the **pcopy** term in clause C_1' with the head of clause C_3.

In the figure each node is labeled with the sequence of clauses used to resolve subgoals. For example, to reach the node labeled C'1·C3, one resolves the p-goal using clause C_1', tries to resolve the resulting pcopy-subgoal using clause C_4 but fails, backtracks, and tries again using clause C_3.

The order of the nodes C_1' and C_2 at depth 1 is established, as described above, by placing them in order left-to-right with decreasing values of p/C. The order of the clauses C_3 and C_4 below C_1' is determined similarly.

Note that the option to consider node C2 *after* node C'1· C4 but *before* C'1· C3 is not available if we may only reorder clauses for a single predicate. It is, however, if we consider reordering all the nodes at depths ≤ 2—i.e., the nodes C'1· C4, C'1· C3, and C2. To do so we need to estimate the p/C values for the *pairs of steps* that lead to the depth-two nodes—e.g, we need to find the likelihood that a successful solution will be found using clause C'1 followed by C4, in comparison to the other two possibilities of C'1 followed by C3 or C2. Suppose we do this and we discover that the p/C value for C2 is between that of the two nodes at depth two; then we should reorder the tree as in Figure 3. The corresponding program is shown in Figure 4. Note that there are now three clauses for the predicate p, since there are three nodes of depth ≤ 2 in the tree; the first and third clauses have each been unfolded one level.

This example illustrates how appropriate unfolding transformations can be found using the same learning statistics as for reordering. Actually, there is a minor difference between reordering one-step nodes and two-step nodes: in the event of failure, the program of Figure 4 must unify the p goal with the head of a p-clause three times, instead of two in Figure 1(c); this small additional expense can be estimated and taken into account in the decision about whether to change the order of the tree in Figure 2 to that of Figure 3.

Although we have treated only a special case, the general procedure for determining unfolding transformations is the same: Determine the best ordering for *pairs* of resolution steps instead of single steps; if in the resulting order a clause at the lower level has only one immediate child node (e.g., the nodes labeled `C'1` in Figure 4), go ahead and unfold. By iterating this procedure during the learn/optimize cycle, one can end up unfolding multiple steps into a single clause, without the need to estimate statistics for more than two steps at a time.

Conclusion: Where do we go from here?

The work described here builds upon and extends the work of many researchers, including Smith and Genesereth (1985), Prieditis and Mostow (1987), Gooley and Wah (1989) , and Greiner and Orponen (1991). The principal contributions are as follows:

- We have shown that recursive programs *can* be dynamically optimized efficiently and effectively. Previous work has reported difficulties speeding up recursive programs.
- The nature of that optimization can be quantified better than previous heuristic methods were able to.
- We have shown how search control can be embedded in the program instead of being added on in the form of a time-consuming meta-theory that must be evaluated outside the actual program.
- By averaging over several runs we have removed the dependency of the resulting optimized program on the order of the examples. This has been a consistent problem with caching methods.
- We have integrated the utility estimates into the learning procedure: no transformations are even generated unless their utility justifies it with high probability. The learning data and the utility evaluation data are one and the same.
- We have eliminated all vestiges of the "generalization-to-N" anomalies and tricky "operationality" decisions that occur with EBG-based methods. In fact, unfolding itself has been reduced to a minor transformation that occurs after consecutive pairs of clauses have been reordered.

This work is still in progress. Future plans call for redoing the previous dynamic optimizer for Prolog to incorporate the improvements described here and to handle many of the so-called impure constructs in Prolog, including negation-as-failure, `call`, and and `or`. Mathematical analysis of this approach is still incomplete, and we cannot yet argue that a strategy different from unrolling and reordering will not provide superior optimization.

Finally, for dynamic optimization of programs to be of commercial value, we must be able to optimize programs written in commercial programming languages. Prolog is not a commercial data-processing language and, in my estimation, is not likely to become one. The method described here does not apply to procedural languages like C and Fortran, where order of decisions cannot be changed. One could add nondeterministic search primitives to such languages, but such new constructs are unlikely to gain wide acceptance. Constraint-logic programming, on the other hand, is growing in popularity as a programming language, and it is quite likely that these optimization methods are applicable. Also, very similar dynamic-optimization problems occur in database query languages, where further opportunities for commercialization may be found.

References

[1] Gooley, M. and Wah, B. 1989. Efficient reordering of Prolog programs. *IEEE Trans. on Knowledge and Data Engineering* 1:470–482.

[2] Greiner, R. and Orponen, P. 1991. Probably approximately optimal derivation strategies. In *Proceedings 2nd International Conference, Knowledge Representation and Reasoning.* 277–288.

[3] Greiner, R. 1989. Towards a formal analysis of EBL. In *Proc. Sixth Int. Machine Learning Workshop.* Morgan Kaufmann. 450–453.

[4] Laird, P. and Gamble, E. 1990. Extending EBG to term-rewriting systems. In *Proceedings AAAI-90.* American Association for Artificial Intelligence.

[5] Laird, P. 1992a. Discrete sequence prediction and its applications. In *Proc., 9th National Conference on Artificial Intelligence.* AAAI.

[6] Laird, P. 1992b. Dynamic optimization. In *Proc., 9th International Machine Learning Conference.* Morgan Kaufmann.

[7] Prieditis, A. and Mostow, J. 1987. Prolearn: Towards a Prolog interpreter that learns. In *Proceedings of AAAI-87.* Morgan Kauffman.

[8] Smith, D. E. and Genesereth, M. R. 1985. Ordering conjunctive queries. *Artificial Intelligence* 26.