Appears in *Proceedings of the 1992 Machine Learning Workshop on Knowledge Compilation and Speedup Learning,* Aberdeen, Scotland, July, 1992

Speeding-up Logic Programs by Combining EBG and FOIL

John M. Zelle and **Raymond J. Mooney**
Department of Computer Sciences
University of Texas
Austin, TX 78712
zelle@cs.utexas.edu, mooney@cs.utexas.edu

Abstract

This paper presents an algorithm that combines traditional EBL techniques and recent developments in inductive logic programming to learn effective clause selection rules for Prolog programs. When these control rules are incorporated into the original program, significant speed-up may be achieved. The algorithm produces not only EBL-like speed up of problem solvers, but is capable of automatically transforming some intractable algorithms into ones that run in polynomial time.

1 INTRODUCTION

Explanation-based learning (EBL) research in logic programming has generally focussed on learning *macros* (compiled rules) [Mitchell et al., 1986; DeJong and Mooney, 1986; Prieditis and Mostow, 1987], while EBL work in planning and production systems has tended to focus on learning search-control rules [Mitchell et al., 1983; Minton, 1989; Laird et al., 1986]. Recently, Cohen [Cohen, 1990] has argued the advantages of learning search control rules to solve the *clause selection problem* in logic programming. Standard EBL methods can be applied to clause selection; however, they tend to produce control rules that are accurate but highly complex. The complexity of the rules makes them costly to use and can often degrade overall performance rather than improving it [Minton, 1988]. Cohen's system, AxA-EBL, deals with this problem by using induction to learn a small set of "approximate" control rules with reduced match cost. AxA-EBL "explains" correct uses of a clause by compiling out a generalized macro for the subgoal to which the clause was applied. A pool of candidate control rules is formed by considering all *k-bounded approximations* of these macros. A k-bounded approximation is formed by dropping all but j conditions from the macro for some $j < k$. AxA-EBL then searches this pool for a small set of rules which maximizes coverage of positive examples and minimizes the coverage of negatives. AxA-EBL was shown to out-perform standard EBL control rules in a number of problem solving domains.

Although quite successful, AxA-EBL suffers from a number of weaknesses. First, only very small values of k can be used since the number of k-bounded approximations grows exponentially in k. A second problem is that explanantions for subgoals are created without considering the context of the surrounding proof. Often the conditions which cause the application of a clause to eventually fail lie outside of the proof of the specific subgoal to which the clause was applied.

Our system DOLPHIN (Dynamic Optimization of Logic Programs through Heuristics INduction) can be viewed as an extension of the AxA-EBL approach that attempts to solve these two problems. We apply a more powerful induction algorithm, Quinlan's FOIL [Quinlan, 1990], to the problem of constructing approximate control rules in the context of the surrounding proof structure. We present empirical results in three domains showing that this approach produces approximate control rules of high utility and generally outperforms competing approaches. In particular, we show that, unlike existing methods, DOLPHIN can transform an $O(n!)$ generate-and-test sorting program into an $O(n^2)$ insertion sort after solving only a single problem.

2 THE DOLPHIN ALGORITHM

The DOLPHIN algorithm attempts to optimize a Prolog program by learning clause selection heuristics. The input to the learning system is a Prolog program, a specification of which predicate consitutes the "top-level" goal and a set of training problems. DOLPHIN uses the examples to induce a set of control heuristics which are then incorporated into the original program to produce a modified program as output. The algorithm proceeds in three phases: example analysis, control rule induction, and program specialization.

```
naivesort(X,Y) :-
        permutation(X,Y),
        ordered(Y).

permutation([],[]).
permutation([X|Xs],Ys) :-
        permutation(Xs,Ys1),
        select(X,Ys,Ys1).

select(X,[X|Xs],Xs).
select(X,[Y|Ys],[Y|Ys1]) :-
        select(X,Ys,Ys1).

ordered([_X]).
ordered([X,Y|Ys]) :-
        X =< Y,
        ordered([Y|Ys]).
```

Figure 1: Naive Sorting Program

2.1 EXAMPLE ANALYSIS

In the example analysis phase, training examples are solved using the existing program. The structure of each proof is recorded, and the proofs are used to extract examples of correct and incorrect clause selection decisions in a manner similar to [Cohen, 1990; Mitchell et al., 1983]. A selection decision for a clause is a partially instantiated subgoal to which the clause was applied. A correct decision is one that occurs in a successful proof tree, while an incorrect decision is an application that was tried and subsequently backtracked over. Any given example may produce numerous positive and negative examples of clause selection decisions.

Once the examples have been solved, standard EBG techniques [DeJong and Mooney, 1986; Mitchell et al., 1986] are used to generalize the resulting proof trees. The goal of this generalization is to remove those elements of the proof that are dependent on the specific facts of the example while maintaining the overall structure of the proof. In DOLPHIN, this is done by "retracing" the proof steps on a top-level goal which has uninstantiated input arguments.

As an example, consider the naive sorting program in Figure 1 which sorts a list by generating permutations until it finds one that is ordered. Permutations are generated by permuting the tail of the input list and inserting the head somewhere in the permuted tail. The predicate, `select(Item, List1, List2)`, holds when `Item` is a member of `List1`, and `List2` is `List1` with an occurrence of `Item` removed. In `naivesort`, `select/3` is actually called with its first and third arguments instantiated to insert items into the permutation, which is returned in the second argument.

Notice that the only nondeterminism in this program comes in the definition of the predicate, `select/3`. This nondeterminism could be eliminated by learning a control rule for the first clause which accurately predicts when the item should be placed at the front of the list. Given the top-level example, `naivesort([9,1,5,3,4],X)`, the example analysis phase discovers 5 examples of correct uses of the clause and 6 failed attempts. These selection decision examples, shown in Table 1, represent positive and negative examples of the concept, `useful_select_1`.

```
naivesort([A,B,C,D,E], [B,D,E,C,A])
   permutation([A,B,C,D,E], [B,D,E,C,A])
      permutation([B,C,D,E], [B,D,E,C]
         permutation([C,D,E], [D,E,C])
            permutation([D,E], [D,E])
               permutation([E], [E])
                  permutation([],[])
*1*               select(E, [E], [])
               select(D, [D,E], [E])
            select(C, [D,E,C], [D,E])
               select(C, [E,C], [E])
                  select(C, [C], [])
*2*      select(B, [B,D,E,C], [D,E,C])
      select(A, [B,D,E,C,A], [B,D,E,C])
         select(A, [D,E,C,A], [D,E,C])
            select(A, [E,C,A], [E,C])
               select(A, [C,A], [C])
                  select(A, [A], [])
   ordered([B,D,E,C,A])
*3*   B =< D
      ordered([D,E,C,A])
         D =< E
         ordered([E,C,A])
            E =< C
            ordered([C,A]
               C =< A
               ordered([A])
```

Figure 2: Generalized Proof of naivesort([9,1,5,3,4], X)

The generalized proof extracted from the trace of this example is shown in Figure 2. The proof provides a context which "explains" the success of correct decisions. Generalized proofs are used along with the selection decision examples to do control rule induction.

Table 1: Examples of `useful_select_1`

Positives	Negatives
`select(9,A,[])`	`select(9,A,[5])`
`select(1,A,[3,4,5])`	`select(9,A,[4,5])`
`select(5,A,[])`	`select(9,A,[3,4,5])`
`select(3,A,[4])`	`select(9,A,[1,3,4,5])`
`select(4,A,[])`	`select(5,A,[4])`
	`select(5,A,[3,4])`

2.2 CONTROL RULE INDUCTION

The goal of the control rule induction phase is to produce an operational definition for when it is useful to apply each clause of the original program. Given a program clause, C, we desire a definition of the concept "subgoals for which C is useful." Thus, control rule learning in this context can be viewed as relational concept learning in a logic programming framework. A number of recent systems [Quinlan, 1990; Muggleton and Feng, 1990] have tackled this problem. Dolphin employs a modification of Quinlan's Foil algorithm to perform control rule induction.

The choice of a Foil-like framework was motivated by a number of factors. First, Foil is relatively easy to implement and has proven efficient in a number of domains. Second, the general Foil algorithm has a "most general" bias which tends to produce simple definitions. This is important in creating classification rules with low match cost. Third, it is relatively easy to bias Foil with prior knowledge [Pazzani and Kibler, Forthcoming]. In our case, we can take advantage of the information provided by the generalized proofs produced in the example analysis phase.

2.2.1 Basic Foil Algorithm

```
positives-to-cover = {positive examples}.
While positives-to-cover is not empty
   Find a clause, C,  that covers some
      examples in positives to cover, but
      covers no negative examples.
   Add C to the developing definition.
   Remove examples covered by C
         from positives-to-cover.
```

Figure 3: Foil Covering Algorithm

Foil attempts to learn a definition of a concept in terms of some given background predicates. The definition comprises a set of function-free definite clauses that cover all of the positive examples of the concept, and none of the negative examples.

Foil may be viewed as a simple covering algorithm which has the basic form shown in Figure 3. The "find a clause" step is implemented by a general-to-specific hill-climbing search. Foil adds antecedents to the developing clause one at a time. At each step Foil evaluates all possible literals that might be added and selects the one which maximizes an information-based gain heuristic.

Since Foil learns clauses which are function-free, the generation of candidate literals to add to the developing clause essentially consists of trying each background predicate with all possible combinations of variables currently in the clause and new variables. Any function symbols in the theory must be introduced by the background predicates. This means, for instance, that programs operating on lists need a background predicate like `components(List, Head, Tail)` that can decompose a list structure.

While the function-free limitation makes the search for literals in Foil relatively efficient, it presents numerous difficulties in using Foil to learn control heuristics for Prolog programs including:

- Prolog programs often have functions which are not introduced into the program with primitive "constructor" predicates.
- Introducing constructors tends to create definitions with greater match cost, since discriminations that could be made immediately via unification are pushed into calls to more primitive antecedents. This is particularly wasteful when an antecedent shares a variable with the head of a clause, and delaying unification may affect how the clause is indexed by the Prolog system.
- Forcing definitions to be function-free may exacerbate the incompleteness of hill-climbing. Pattern matching in unification can be used to test a value in a deeply nested structure. Decomposing the structure via constructor antecedents may require the addition of arbitrarily many antecedents which do not individually help discriminate between positive and negative examples. An extension to Foil, determinate literals, has been proposed to partially address this problem [Quinlan, 1991] at the cost of complicating the simple Foil algorithm.

Dolphin uses the generalized proofs of examples to suggest which literals might be added to the current clause. Our approach has the twin benefits of focusing the Foil search and learning clauses which may include functions.

2.2.2 Using Proofs to Specialize Clauses

The generalized proofs of top-level examples can be seen as giving the context for the appropriate application of clauses used within the proof. The operational nodes of the proof represent all of the primitive conditions that had to be satisfied for the proof to succeed. Dolphin employs induction in an attempt to identify a small set of simple tests that provide significant guidance in determining whether the application of a given clause is likely to be part of a complete proof. It is important to note that the conditions being sought may not necessarily lie in the proof of the specific subgoal to which the clause is applied. A decision to use a clause may turn out to be wrong not because the clause itself could not succeed, but rather, because it did succeed and produced bindings which could not be used in completing the surrounding proof. This is especially common in programs such as the naivesort which implement a generate-and-test paradigm.

DOLPHIN employs the same general covering algorithm as FOIL but modifies the clause construction step. Successive specializations of a clause are generated by considering the uses of the program clause for which the heuristic is being learned. Suppose we are learning a definition of the concept "subgoals for which clause, $A \leftarrow B$, is useful." The most general clause covering the examples is simply `useful(`A'`)`$\leftarrow$ `true`, where A' is a "copy" of A having uninstantiated arguments; call this clause, C. C can be specialized by (partially) instantiating some of its variables, or by adding antecedents to its body. The former is achieved by unifying A' with a (generalized) subgoal which was solved by the original clause, $A \leftarrow B$. The latter is done by unifying A' with a subgoal and adding an operational literal from the proof which shares some variables with that subgoal. The clause specialization search algorithm is detailed in Figure 4.

```
Let PC be the original program clause
Let PROOFS be the set of generalized proofs
Let C = be the clause to be specialized
SPEC-PAIRS = {}
for each PROOF in PROOFS
   SUBGOALS = subgoals in PROOF solved by PC
   for each SUBGOAL in SUBGOALS
      add (SUBGOAL,true) to SPEC-PAIRS
      for each LITERAL in PROOF
         if operational(LITERAL) and
               share_vars(LITERAL,SUBGOAL)
**          add generalize((SUBGOAL,LITERAL))
               to SPEC-PAIRS
REPEAT
   SPECIALIZATIONS = {}
   for each (SUBGOAL,ANTE) in SPEC-PAIRS
      add to SPECIALIZATIONS the clause
         created by unifying C's argument
         with SUBGOAL and appending ANTE
         to C's body
   C = simplest clause among those tied for
       maximal information-gain in
       SPECIALIZATIONS
UNTIL no specialization has
      positive information-gain
```

Figure 4: Clause Specialization Algorithm

The `generalize((SUBGOAL,LITERAL))` call on the line labelled ** is included so that specializations created by adding a literal only instantiate variables to the extent dictated by the literal. Sub-terms appearing in SUBGOAL which are not in LITERAL are generalized to unique (uninstantiated) variables. Thus, instantiating head variables (via unification with a subgoal) and adding a literal are independent specializations.

A couple of things are worth noting about the algorithm. Variables in a newly added antecedent are connected with the existing clause by unification of the predicate's argument with a subgoal sharing variables with the antecedent. Thus, the current version of the algorithm only adds literals which directly share some variables with the head of the clause under construction. This restriction is not strictly necessary. It may be that arbitrary chains of literals from a proof are useful in the developing definition; new literals could be added by matching a chain in a proof with antecedents already included in the clause being constructed. Enforcing the "no-chaining" restriction makes the algorithm easier to implement, improves the efficiency of the induction (since fewer possible specializations are considered) and lowers the match-cost of learned heuristics by avoiding extended chains of literals which must be evaluated.

It should be noted that the no-chaining restriction may prevent the learning of an accurate control rule. Specialization terminates when no further improvement is found in the information-gain metric. This means it is possible to produce clauses in the concept definition which still cover some negative instances, permitting the learning of approximate definitions.

2.2.3 Control Rules for Naivesort

Control rule induction for the naivesort problem is straight-forward. Initially the set of control rules is empty, and the covering algorithm attempts to find a clause which covers some of the positive examples from Figure 1. The initial clause,

useful_select_1(select(A,B,C))$\leftarrow$ true

covers all of the positive and negative examples. DOLPHIN will attempt to specialize it.

In performing specialization, `SPEC-PAIRS` will contain, among others, the pairs:

```
(select(E,[E],[]),true))
(select(B, [B,D|X],[D|X]), B =< D)
```

The first is created by unification with the subgoal labeled *1* in the generalized proof (Figure 2). The second is created from the subgoal labeled *2* and the operational literal labeled *3*. The sub-term [E,C] from *2* has been generalized to a new variable, X, since [E,C] does not appear in the operational literal and is therefore unnecessary for connecting it to the subgoal.

Applying these two pairs to produce specializations produces the clauses:

useful_select_1(select(A,[A],[]))$\leftarrow$ true
useful_select_1(select(A, [A,B|C],[B|C])$\leftarrow$ A =< B

These along with other possible specializations, will be evaluated on the examples in Table 1 to determine which produces the most gain (loosely, greatest accuracy). The first clause covers three positive examples and no negatives, the second covers two positives and no negatives, so the former is preferred. Since the clause covers no negative examples, no further improvement is possible via specialization. This clause is picked as the first clause of the control rule definition. The examples covered by the clause are removed from positives-to-cover and the process repeats. On the second iteration, the winning specialization is the second clause shown above. At this point, all of the positive examples are covered, and we have found a definition for when it is useful to apply the first clause of `select/3`.

2.3 PROGRAM SPECIALIZATION PHASE

Once clause selection rules have been learned, they are passed to the program specialization phase which "folds" this control information into the original program. The basic approach is to guard the body of each clause with the selection information. This forces the clause to fail quickly on subgoals to which it should not be applied.

For non-disjunctive (single clause) selection rules, the learned conditions are simply placed into the original program clause preceding the conditions already present on the clause, and the clause head is unified with the argument of the selection rule. For disjunctive selections, a single new literal is added at the front of the program clause. This new literal has the same arguments as the clause head. The definition of the new literal comprises the clauses of the selection rule with the heads modified so that what were originally arguments of the subgoal are made direct arguments of the predicate. A cut ("!") is appended to the body of each clause of this definition since there is no reason to consider multiple proofs of the usefulness of the original program clause.

A decision is also made as to whether the selection information has made the program clause deterministic. If the learned selection rules cover no incorrect decisions in the training data, then it is assumed that the modified clause is deterministic and a cut is placed after the added condition(s). This has the effect of commiting us to the program clause once it has been selected as useful.

Returning to the sorting example, folding the clause selection rules back into the theory as described produces a new definition of `select/3` shown in Figure 5.

In effect, `permutation/2` has been modified to produce ordered permutations. Careful inspection shows that this is a version of the insertion sort algorithm, and we have actually made an $O(N!)$ sort into an $O(N^2)$ version by learning from a single top-level ex-

```
select(A, [A|B], B) :-
   useful_select_1(A, [A|B], B), !.
select(A, [B|C], [B|D]) :-
   !, select(A, C, D).

useful_select_1(A, [A], []) :- !.
useful_select_1(A, [A,B|C], [B|C]) :-
   A =< B, !.
```

Figure 5: Improved Select Predicate

ample. By inserting conditions from the testing portion of a generate-and-test program into the generating portion, DOLPHIN is performing *test incorporation* [Dietterich, 1986] which, as illustrated here, can sometimes dramatically enhance the efficiency of an algorithm.

2.4 IMPLEMENTATION

DOLPHIN is implemented in Quintus Prolog on a SPARCstation 2. Little attempt has been made to optimize the code, but the design choices discussed above have yielded a relatively efficient system. The construction of optimized programs for all of the examples discussed in this paper were accomplished in less than a minute of system run-time.

3 EXPERIMENTAL RESULTS

3.1 EXPERIMENTAL DESIGN

The DOLPHIN system has been developed and tested on three problems: naivesort, n-queens, and μLEX. For each of these problems, experiments were run to determine the effectiveness of DOLPHIN in optimizing inefficient programs.

The n-queens problem is adapted from a Prolog program given in [Bratko, 1990]. The problem is to find a placement of N queens on an NxN chessboard such that no queen is attacking another. The program implements a generate and test strategy where a configuration is represented by a permutation of the list, [1..N]. This makes the program very similar to naivesort which also permutes a list and tests.

μLEX is a simplified symbolic integration solver using state-space search with iterative deepening. The actual Prolog code for the solver is the same as that used in [Cohen, 1990].

In each domain, a set of testing problems was chosen as a benchmark. DOLPHIN was then run on independently derived training sets of various sizes to produce "optimized" versions of the programs. These programs were then run on the examples in the testing set to evaluate their performance. For each training set size, 10 trials were run. The results presented here represent an average over the 10 trials.

The examples for the naivesort problem were drawn from randomly generated lists of size 3 to 8. The testing set contained 100 such lists. The data for the N-queens domain consisted of the 9 problems corresponding to the 4-queens through 12-queens problems. The 4 largest problems were used as the test set, and training was done on successively larger subsets of the smaller problems. The μLEX training and testing problems are from [Keller, 1987].

Since Dolphin only specializes clauses of the original program, the optimized program is guaranteed to be sound with respect to the computations performed by the original program. That is, any solution found by the optimized program must also have been a solution of the original. Unfortunately, the optimized program may not be complete. There may be problems which the original program can solve, but whose solutions have been pruned from the search space of the optimized version. In order to guarantee the completeness of the final program, we adopt the strategy used by [Cohen, 1990] and retain the original clauses. In testing, we first attempt to solve the problem using the optimized program. If this fails, the original program is then used to find a solution to the problem.

It is clear that the speed-up achieved in this two-tiered approach is critically dependent not only on the efficiency of evaluating the learned rules, but also on the extent to which the optimized program covers novel examples. For each of the three problems we evaluate the effect of training set size on both the run-time and the coverage achieved by the optimized program.

3.2 RESULTS

The results of the experiments on these three problems are graphically summarized in Figure 6 and Figure 7. In all three, Dolphin was able to significantly improve the performance of the initial program. The times shown in the timing graphs represent the number of seconds required to solve the problems in the training set. All times are for compiled Quintus Prolog programs running on a SPARCstation 2.

It is not surprising that Dolphin produces programs for the sorting problem that are significantly faster than the original permutation-based sorting algorithm. We have already shown how the $O(N!)$ sort can be "optimized" into a polynomial version. What is, perhaps, surprising is how few examples are necessary on average to learn the enhanced program. As can be seen from the coverage graph, 2 examples is ususally sufficient to learn insertion-sort, and 4 examples virtually guarantee success.

In the n-queens problem, there is no local condition (known to the authors) which allows for a fundamental improvement in the order of the algorithm. In this case Dolphin learned rules which were more "heuristic". Still, the rules learned by Dolphin were very effective in pruning the search space of the larger problems. The coverage curve for this problem is not very interesting as there were only four problems in the test set, and all four were covered by all training sets which included the 4 and 5-queen examples. It is interesting to note that the performance of the optimized programs continues to improve even after coverage has stopped improving. This is possible because the presence of more examples guides the inductive component to construct rules with greater utility.

On the μLEX domain which is a more "standard" EBL problem, Dolphin compared favorably with other approaches to learning in the logic programming framework. The plot labelled EBL-macro reflects the result of applying the EBG mechanism used in Dolphin to learn macro-rules for the top-level goal. The learning mechanism in EBL-macro first tried to prove an example using it's learned rules. If no learned rule applied, the problem was proved using the normal solver and the subsequent proof was generalized to produce a new macro. The new macro was then added to the end of the list of learned rules. The lack of utility for the macro approach on μLEX is probably due to the relative simplicity of the problems in the testing set. The match cost of the macros was not significantly lower than the cost of proving the example from scratch.

The curve labeled "AxA-EBL & EBL-Control" in Figure 7 is taken from [Cohen, 1990] and represents only a rough comparison. The times reported there range from over 26 seconds to under 2 seconds. The values shown here were interpolated off a graph and scaled appropriately. AxA-EBL is Cohen's integration of induction and explanation discussed earlier. EBL-control is a "rational reconstruction" of standard EBL control rule learning applied to the clause selection problem. Cohen found that these two performed essentially identically on the μLEX problem.

Although one must be cautious when making comparisons across systems, the curves seem to suggest that Dolphin converges to an efficient program more quickly than AxA-EBL or EBL-control. For μLEX, this difference is probably not due to the induction mechanism, but rather to the differing strategies used in folding the learned rules back into the program. Dolphin is conservative in committing to a rule choice and only inserts a cut after the learned control conditions if the conditions did not cover any negative training examples. AxA-EBL (and EBL-control) always insert the cut. This enhances the relative coverage of the Dolphin-optimized program allowing it to backtrack from mistakes and continue solving in the more efficient program rather than immediately defaulting to the less efficient solver whenever a selected clause fails

It is also worth noting that EBL-macro, AxA-EBL

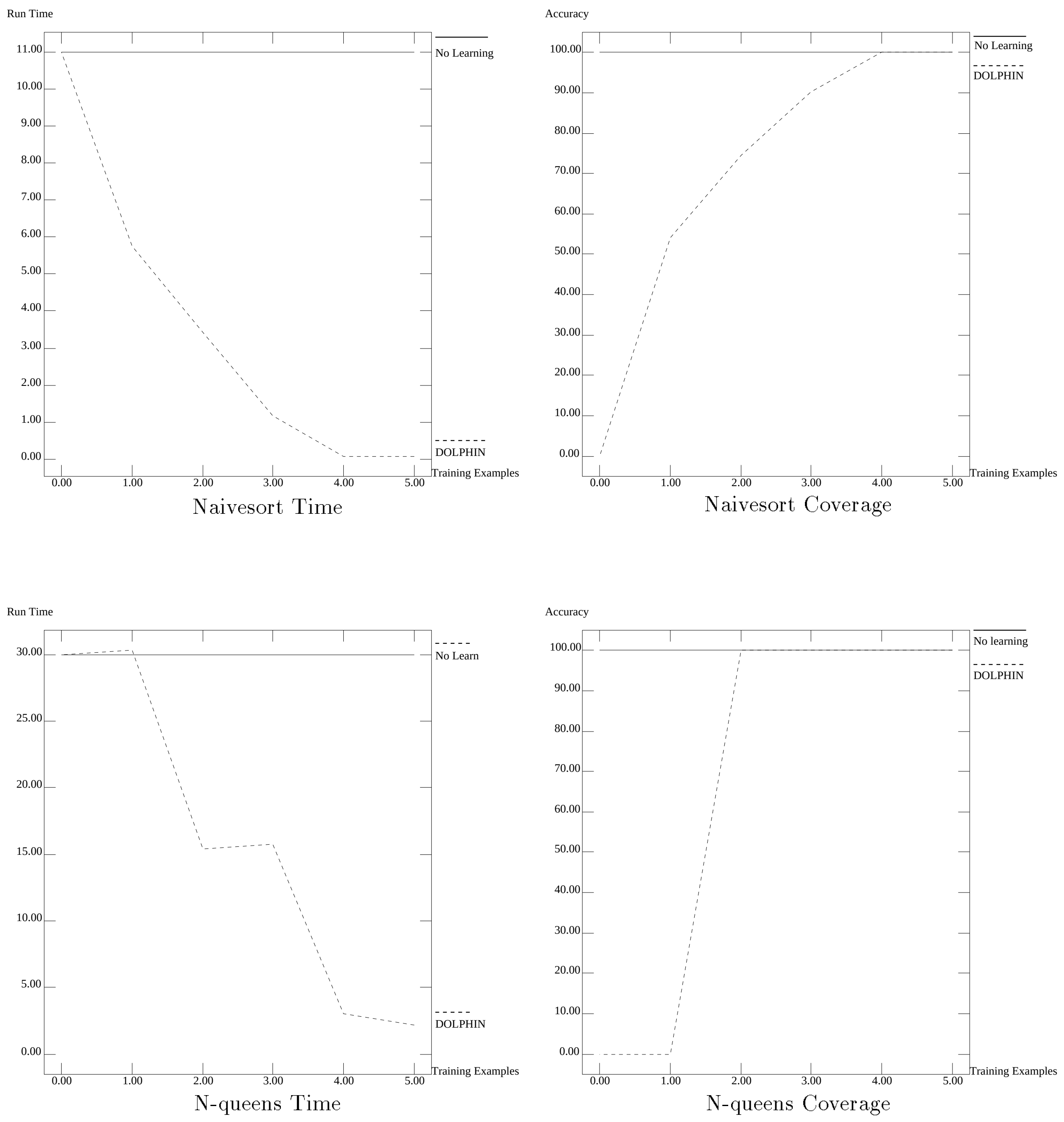
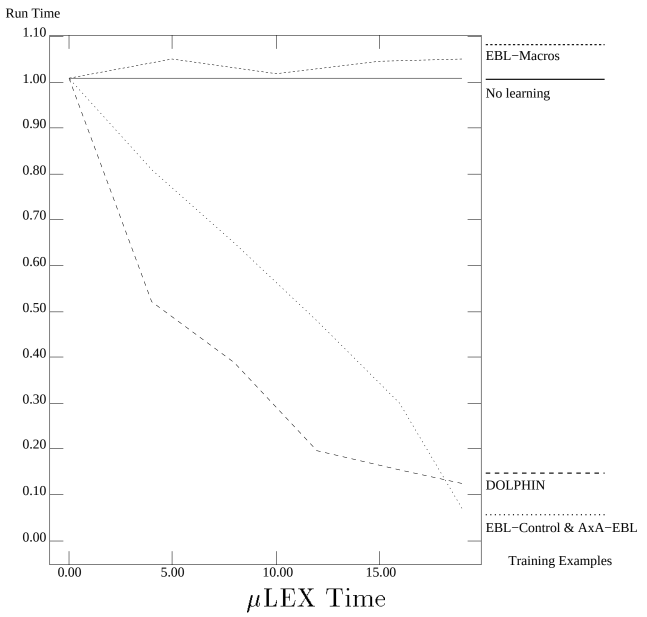


Figure 6: Performance and Coverage Curves for Naivesort and N-Queens



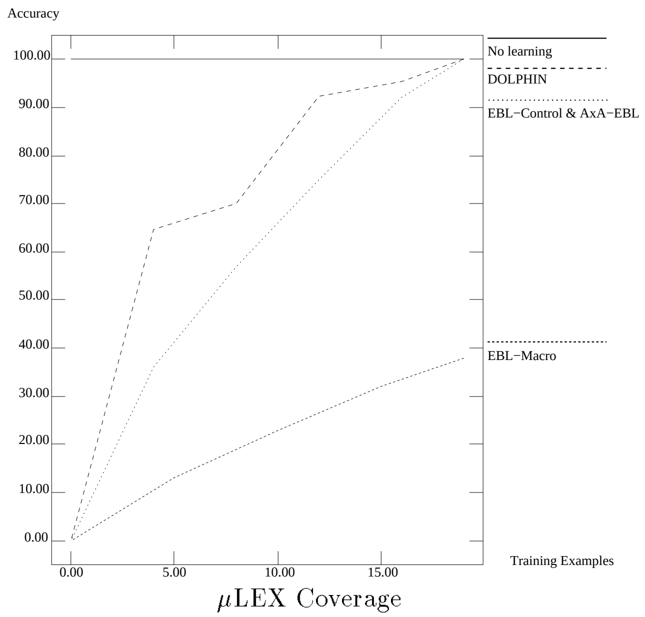


Figure 7: Performance and Coverage Curves for μLEX

and EBL-control would all fare very poorly on the naivesort and n-queens problems, although the reasons would differ. EBL-macro would suffer from the recursive, unbounded nature of the problems. This strategy would be forced to acquire generalized proofs of all permutations of various sized lists with little hope of achieving significant coverage on novel examples. Techniques such as "generalizing to N" [Shavlik, 1990] would not help significantly since there is no regular recursive structure in the correct explanations. Each ordering of the input list produces a different sequence of applications of the first and second clauses of `select/3`.

AxA-EBL would fail to learn any control rules for these programs at all. AxA-EBL only considers explanations of the immediate subgoal to which a clause was applied. The proof of the subgoal in this case is just `true`, because the first clause of `select/3` has no antecedents. Hence, the only learnable condition is `true`, which is not a useful heuristic. These two problems are perfect illustrations of the need to consider the larger context of the surrounding proof in explaining the success of a clause application. Finally, EBL-control shares the shortcomings of both EBL-macro and AxA-EBL.

4 RELATED WORK

Early research in learning control rules [Mitchell et al., 1983; Langley, 1985] did not focus on the utility problem, approximation, or application to logic programming. LEX-2 combined induction and EBL by inducing over complete explanation-based generalizations. DOLPHIN on the other hand, uses induction to select the most useful pieces of EBL generalizations. Also, DOLPHIN takes advantage of recent progress in relational learning, namely, FOIL.

The first use of approximations in learning control rules was probably MetaLEX [Keller, 1987]. However, it used a fairly simple method of simplifying learned rules by removing conditions. Most other recent investigations in learning approximations [Ellman, 1988; Tadepalli, 1989; Chien, 1989] have not focussed on search-control heuristics. Approximating control rules was investigated in [Chase et al., 1989]; however, their system, ULS, does not employ induction and is therefore limited to conservative approximations. Reported improvements in efficiency for ULS were relatively modest.

Recently, Yoo and Fisher [Yoo and Fisher, 1991] have used induction over explanations [Flann and Dietterich, 1989] to improve performance in a problem solving framework. They attempt to increase the utility of EBL-macros by clustering them in a COBWEB-style classification tree that maintains explanantions at various levels of detail. By contrast, DOLPHIN uses supervised learning methods to acquire explicit search-control rules.

FOCL [Pazzani and Kibler, Forthcoming] also combines EBG and FOIL by integrating induction into the operationalization process. FOCL improves the accuracy of a theory rather than it's efficiency. Consequently, unlike DOLPHIN, it does not use integrated learning to acquire control rules.

The most closely related work is AxA-EBL [Cohen, 1990]. As discussed above, Dolphin improves on AxA-EBL by using a more-powerful inductive learning mechanism and considering the entire proof of a top-level goal as the explanation for the successful application of a clause to a particular subgoal.

5 FUTURE RESEARCH

Experiments with Dolphin have raised a number of issues which require further investigation. One shortcoming of the current approach is it's critical dependence on the form of the program which is being optimized. Using the alternative definition of `permutation/2` shown in Figure 8 would prevent Dolphin from being able to learn the insertion-sort heuristic. In order to achieve significant performance enhancement on this program, the system would have to learn to choose the minimum element from a list when performing a permutation. In this particular case, we might be able to turn the naivesort into a selection sort if the system is allowed to learn a "recursive" control rule which uses the predicate for which control is being learned. A more general solution might be to use constructive induction to directly invent a suitable test (in this case "minimum of list").

```
permutation([], []).
permutation(Xs, [H|T]) :-
        select(H, Xs, Xs1),
        permutation(Xs1, T).
```

Figure 8: Alternative Definition, `Permutation/2`

Other changes to the inductive mechanism might also prove useful. While the simplicity of the control rules learned by the inductive component of Dolphin tends to increase their utility, there is no explicit use of match-cost or operationality. In fact, the Foil information metric actually tends to favor conditions having multiple proofs, which seems harmful to utility since many instantiations of a condition may have to be tried before it fails completely. One interesting avenue of investigation would be to incorporate notions of operationality into the hill-climbing mechanism to bias the search toward more efficient control heuristics.

Another problem is the need to deal with an even more extended notion of proof context. Programs often use a single predicate in different ways. Suppose a program used `permutation/2` to sort some lists into ascending order and others into descending order. The conflicting uses of the permutation predicate will be reflected as "noise" in the selection decision examples over which Dolphin is attempting to do the induction. Resolving this problem requires that the selection decisions themselves be treated as proof-context dependent.

6 CONCLUSIONS

Dolphin is the first system to combine recent developments in inductive logic programming with EBG in order to improve the efficiency of logic programs. By using the Foil algorithm to select the most useful portions of generalized explanations, Dolphin can learn approximate control rules of high utility. In particular, by examining the entire proof when generating control rules for any subgoal, Dolphin can perform a form of test incorporation on logic programs. Test incorporation allows Dolphin to transform some intractable algorithms into ones that run in polynomial time.

Acknowledgements

This research was supported by the National Science Foundation under grant IRI-9102926, the NASA Ames Research Center under grant NCC 2-629, and the Texas Advanced Research Program under grant 003658114. We wish to thank Je-Nien Ernst who provided the data on EBL-macro.

References

Bratko, I. (1990). *Prolog Programming for Artificial Intelligence.* New York , NY: Addison-Wesley.

Chase, M., Zweben, M., Piazza, R., Burger, J., Maglio, P., and Hirsh, H. (1989). Approximating learned search control knowledge. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 40–42. Ithaca, NY.

Chien, S. (1989). Using and refining simplifications: Explanation-based learning of plans in intractible domains. In *Proceedings of the Eleventh International Joint conference on Artificial intelligence.* Detroit, MI.

Cohen, W. W. (1990). Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 268–276. Austin, TX.

DeJong, G. and Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176.

Dietterich, T. (1986). Learning at the knowledge level. *Machine Learning*, 1:287–316.

Ellman, T. (1988). Approximate theory formation: An explanation-based approach. In *Proceedings of National Conference on Artificial Intelligence*, pages 564–569. St. Paul, MN.

Flann, N. and Dietterich, T. (1989). A study of explanation-based methods for inductive learning. *Machine Learning*, 4:187–226.

Keller, R. (1987). *The Role of Explicit Contextual Knowledge in Learning Concepts to Improve Performance.* PhD thesis, New Brunswick, N: Rutgers University. Also appears as tech. report ML-TR-7.

Laird, J., Rosenbloom, P., and Newell, A. (1986). Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1).

Langley, P. (1985). Learning to search: From weak methods to domain specific heuristics. *Cognitive Science*, 9(2):217–260.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. In *Proceedings of National Conference on Artificial Intelligence*, pages 564–569. St. Paul, MN.

Minton, S. (1989). Explanation-based learning: A poblem solving perspective. *Artificial Intelligence*, 40:63–118.

Mitchell, T., Utgoff, T., and Banerji, R. (1983). Learning problem solving heuristics by experimentation. In Michalski, R., Mitchell, T., and Carbonell, J., editors, *Machine Learning: An Artificial Intelligence Approach*. Palo Alto, CA: Morgan Kaufmann.

Mitchell, T. M., Keller, R., and Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80.

Muggleton, S. H. and Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First International Conference on Algorithmic Learning Theory*, pages 368–381. Tokyo.

Pazzani, M. and Kibler, D. (Forthcoming). The utility of background knowledge in inductive learning. *Machine Learning*.

Prieditis, A. and Mostow, J. (1987). Prolearn: Towards a prolog interpreter that learns. In *Proceedings of National Conference on Artificial Intelligence*. Seattle, WA.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, 5(3):239–266.

Quinlan, J. R. (1991). Determinate literals in inductive logic programming. In *Proceedings of the Eighth International Workshop on Machine Learning*. Evanston, IL.

Shavlik, J. (1990). Generalizing number in explanation based learning. *Machine Learning*, 5:39–70.

Tadepalli, P. (1989). Lazy explanation-based learning: A solution to the intractible theory problem. In *Proceedings of the Eleventh International Joint conference on Artificial intelligence*. Detroit, MI.

Yoo, J. and Fisher, D. (1991). Concept formation over problem-solving experience. In Fisher, D., Pazzani, M., and Langley, P., editors, *Concept Formation: Knowledge and Experience in Unsupervised Learning*. Palo Alto, CA: Morgan Kaufmann.