Appears in *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93),* pp.1106-1111, Chambery, France, 1993

Combining FOIL and EBG to Speed-up Logic Programs

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 is shown to be an improvement over competing EBL approaches in several domains. Additionally, the algorithm 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 [Minton, 1988; Laird *et al.*, 1986]. Recently, Cohen [Cohen, 1990] has argued the advantages of learning search control rules for the *clause selection problem* in logic programming. Clause selection is the process of deciding which of several applicable clauses to use in reducing a particular subgoal in the course of a proof. Incorporating a set of accurate control rules for clause selection into a Prolog program has the potential to reduce or eliminate backtracking thereby producing significant speed-up.

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, handles this problem by combining EBL with induction to learn a small set of "approximate" control rules with reduced match cost. AxA-EBL was shown to out-perform standard EBL control rules across a number of problem solving domains.

Our system, DOLPHIN, (Dynamic Optimization of Logic Programs through Heuristics INduction) can be viewed as an extension of the AxA-EBL approach. DOLPHIN improves on AxA-EBL in two significant ways. First, we employ a more powerful induction algorithm, namely Quinlan's FOIL [Quinlan, 1990]. Second, whereas AxA-EBL "explains" clause selection decisions for subgoals without considering the structure of the surrounding proof, DOLPHIN explicitly considers the surrounding proof during induction. This is beneficial since conditions that cause the application of a clause to eventually fail may lie outside of the proof of the specific subgoal to which the clause is applied.

Empirical results in five problem domains show that our approach produces approximate control rules of high utility and consistently 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 by learning from 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 constitutes the "top-level" goal and a set of training problems. The output is a modified program that incorporates learned clause selection heuristics. The algorithm proceeds in three phases: *example analysis*, *control rule induction*, and *program specialization*. The algorithm is explained in depth and illustrated by way of a simple example in the following subsections.

2.1 Example Analysis

In the example analysis phase, the training problems are solved using the existing program, and traces of the proofs are analyzed to produce two outputs: a set of correct and incorrect clause selection decisions for each clause of the program, and a generalized proof of each training example.

A selection decision for a clause is a partially instantiated subgoal to which the clause was applied at some point in the proof process of a training problem. A correct decision is one that appears in a successful proof. A correct decision for a given clause is an incorrect decision for any prior program clause whose head also unifies with it; since we are considering only the first proof of each training problem, the prior clause must have been

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

permutation([],[]).
permutation([X|Xs],Ys) :-  permutation(Xs,Ys0),
                           insert(X,Ys0,Ys).

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

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

Figure 1: Naive Sorting Program

tried unsuccessfully by the top-down, left-to-right Prolog prover.

Standard EBG techniques [DeJong and Mooney, 1986; Mitchell *et al.*, 1986] are used to generalize the proof trees of the training problems. 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 that 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.

Table 1: Examples of `useful_insert_1`

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

The only nondeterminism in this program comes in the definition of the predicate, `insert/3`. This nondeterminism could be eliminated by learning a control rule for the first clause that 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 five examples of correct uses of the clause and six failed attempts. These selection decision examples, shown in Table 1, represent positive and negative examples of the concept, `useful_insert_1` (i.e., subgoals to which it is useful to apply the first rule of `insert/3`).

The generalized proof extracted from the trace of this problem is shown in Figure 2 (the numbered lines are discussed in a later section). The proof provides a context that "explains" the success of the correct decisions. Generalized proofs are used along with the selection decision examples to do control rule induction.

```
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*                                 insert(E, [], [E])
                              insert(D, [E], [D,E])
                        insert(C, [D,E], [D,E,C])
                              insert(C, [E], [E,C])
                                    insert(C, [], [C])
*2*               insert(B, [D,E,C], [B,D,E,C])
            insert(A, [B,D,E,C], [B,D,E,C,A])
                  insert(A, [D,E,C], [D,E,C,A])
                        insert(A, [E,C], [E,C,A])
                              insert(A, [C], [C,A])
                                    insert(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)

2.2 Control Rule Induction

Control rule learning in DOLPHIN amounts to finding an operational definition for the class of subgoals to which a given clause should be applied. A number of recent systems have addressed the issue of relational concept learning in a logic programming framework [Quinlan, 1990].

The choice of a FOIL-like framework was motivated by a number of factors. First, FOIL is relatively simple 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 easy to bias FOIL with prior knowledge [Pazzani and Kibler, 1992]; we use the generalized proofs produced in the example analysis phase to guide the induction.

2.2.1 Basic FOIL 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 basic covering algorithm. FOIL attempts to find a clause which covers some of the positive examples, but none of the negative. The positive examples covered by the clause are removed from consideration, and the process repeats to find additional clauses until all of the positive examples are covered. Each successive clause is constructed 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 that maximizes an information-based gain heuristic. Since FOIL learns clauses that 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.

The informed FOIL algorithm employed by DOLPHIN is more focused and efficient because it replaces the exponential search through all variablizations of each predicate with a search through literals in the generalized proofs produced in the example analysis phase. It has the additional advantage of being able to learn definitions containing function symbols, which are necessary in control rules for arbitrary Prolog programs.

2.2.2 Using the Results of EBG to Guide FOIL

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 that could not be used in completing the surrounding proof. This is especially common in programs that 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 successful uses of the program clause for which the heuristic is being learned. Suppose we are learning a definition of the concept "subgoals for which the clause, `A←B`, is useful." The most general clause covering the examples is simply `useful(A')←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 some (generalized) subgoal that was solved by the original clause, `A←B`. The latter is done by unifying `A'` with a subgoal and adding an operational literal from the proof that shares some variables with that subgoal. The clause specialization search algorithm is detailed in Figure 3.

`SPEC-PAIRS` is a list of specializations analogous to all literals that would be considered by FOIL for extending a clause. Each pair consists of a "template" to instantiate variables in the head of the clause, and a literal to add to the clause body. A pair with the literal, `true`, represents specializing by variable instantiation only. The `generalize(<SUBGOAL,LITERAL>)` call on the line labeled ** is included so that specializations adding an operational literal only instantiate variables to the extent required; sub-terms appearing in `SUBGOAL` that are not in `LITERAL` are generalized to unique (uninstantiated) variables. The algorithm repeatedly selects the best specialization and applies it to the developing clause. Specialization terminates when no further improvement is found in the information-gain metric. The clause may still cover negative instances, permitting the learning of approximate definitions.

```
Let PC be the original program clause
Let PROOFS be the set of generalized proofs
Let C be (control rule) clause to specialize
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 := clause with best info-gain in SPECIALIZATIONS
until no specialization has positive information-gain
```

Figure 3: Clause Specialization Algorithm

2.2.3 Control Rules for Naivesort

Control rule induction for the naivesort problem is straight-forward. Initially the set of clauses for the concept, `useful_insert_1`, is empty, and the covering algorithm attempts to find a clause that covers some of the positive examples from Table 1. The initial clause,

```
useful_insert_1(insert(A,B,C))← 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:

```
<insert(E,[],[E]),true>
<insert(B,[D|X],[B,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 yields the clauses:

```
useful_insert_1(insert(A,[],[A]))← true
useful_insert_1(insert(A,[B|C],[A,B|C])← A =< B
```

These along with other possible specializations, will be evaluated on the examples in Table 1 to determine which produces the most gain. In this case, the first clause covers more positive examples and is preferred. Since the clause covers no negative examples, no further specialization is needed. This clause is picked as the first clause of the concept definition. The positive examples covered by the clause are removed 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 `insert/3`.

```
insert(A, B, [A|B]) :- useful_insert_1(A,B,[A|B]),!.
insert(A, [B|C], [B|D]) :- !, insert(A, C, D).

useful_insert_1(A, [], [A]) :- !.
useful_insert_1(A, [B|C], [A,B|C]) :- A =< B, !.
```

Figure 4: Improved Insert Predicate

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, forcing the clause to fail quickly on subgoals to which it should not be applied.

For non-disjunctive (single clause) control rules, the learned conditions are simply placed into the original program clause preceding the original conditions, and the clause head is unified with the argument of the control rule. For disjunctive control rules, 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 set of learned control rules 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 control information has made the program clause deterministic. If the learned control 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 committing to the program clause once it has been selected as useful.

Returning to the sorting example, folding the clause selection rules back into the program as described produces a new definition of `insert/3` shown in Figure 4. 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 example. 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.

3 Experimental Results

3.1 Experimental Design

The DOLPHIN system has been evaluated on five problem domains: Two generate-and-test programs, naivesort and N-queens, and three "standard" EBL problems μLEX, RW, and BW borrowed from [Cohen, 1990]. 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].

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]. RW and BW are planning domains utilizing means-ends analysis planners, which were automatically generated from operator definitions. RW is based on the STRIPS robot world. BW is generated from the blocks world operators. Control rules were learned for a single recursive predicate having seven arguments and averaging about 5 conjuncts per clause. This predicate consisted of 18 and 12 clauses in RW and BW respectively. Since problems in these domains can be quite difficult, the planner provides for bounds on both plan length and CPU time.

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 and the results averaged.

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.

The examples for the naivesort problem were drawn from randomly generated lists of size three to eight. The testing set contained 100 such lists. The data for the N-queens domain consisted of the nine problems corresponding to the 4-queens through 12-queens problems. The four 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]. In the planning domains, problems were generated by taking a random walk of bounded length in the state space of the planner in a manner identical to [Cohen, 1990].

3.2 Results

In all domains tested, DOLPHIN was able to significantly improve the performance of the initial program. Figure 5 shows an average learning curve for the naivesort. It is not surprising that DOLPHIN produces programs for the sorting problem that are significantly faster. 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. The points on the graph represent averages over ten trials, and the intermediate averages reflect the proportion of those trials for which insertion-sort was successfully learned. The graph shows that two examples are usually sufficient to learn insertion-sort, and four examples virtually guarantee success.

In the N-queens problem, there is no local condition that can simply be moved into the permutation portion of the program to render it deterministic. In this case

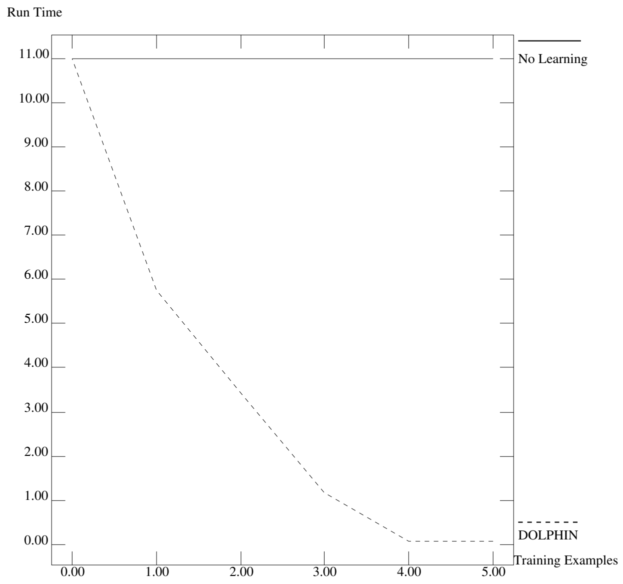

Figure 5: Naivesort Results

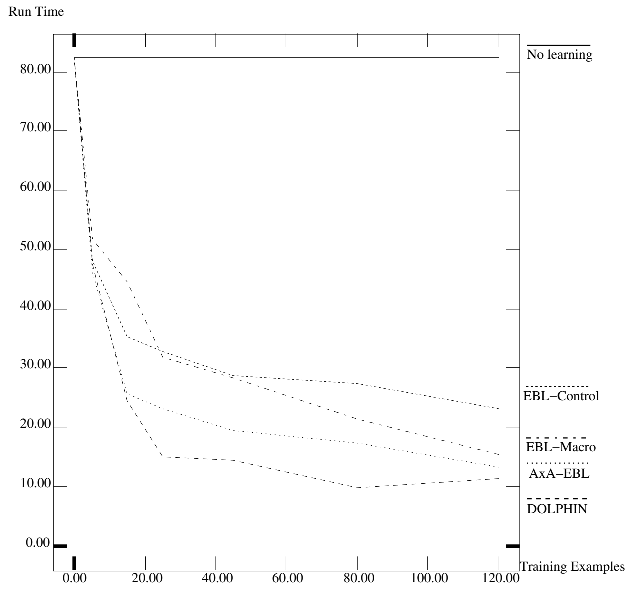

Figure 6: RW Results

DOLPHIN learned rules that were more heuristic. Nevertheless, the results were similar to those for naivesort with training on the 4-8 queens problems yielding more than six-fold speedup for solving the larger problems in the testing set.

In the EBL domains, DOLPHIN was compared to three alternative learning strategies: EBL-macro, EBL-control, and AxA-EBL. EBL-macro used the EBG mechanism from DOLPHIN to learn macro-rules for the top-level goal. The learning mechanism first tried to prove an example using it's learned rules. If no learned rule applied, the problem was solved using the normal solver and the subsequent proof was generalized to produce a new macro, which was added to the end of the list of learned rules. During testing, each problem was first attempted using only the learned macros; if no macro matched the problem, it was solved using the original solver without the macros. The results for AxA-EBL and EBL-Control were taken from [Cohen, 1990]. 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.

While the relative effectiveness of the three alternatives varied across domains, DOLPHIN always produced speedup as good or better than the best of the competing approaches and reached maximum speedup with fewer training examples. The average learning curve for the RW domain, shown in Figure 6, is representative. Although one must be cautious when making comparisons across implementations, the consistency of the results in these domains supports the conclusion that DOLPHIN produces better speed-up than any of the competing approaches and converges to an efficient program more rapidly.

It is also worth noting that EBL-macro, AxA-EBL and EBL-control all fail on the naivesort and N-queens problems, although the reasons differ. EBL-macro suffers from the recursive, unbounded nature of the problems. This strategy is forced to acquire generalized proofs of all permutations of various sized lists with little hope of achieving significant coverage on novel examples. AxA-EBL fails 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 `insert/3` has no antecedents. Hence, the only learnable condition is `true`, which is not a useful heuristic. These two problems nicely illustrate the advantage of considering the surrounding proof context when explaining a successful 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.*, 1986; 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. Approximating control rules was also investigated in [Chase *et al.*, 1989]; however, their system, ULS, does not employ induction and is therefore limited to conservative approximations, producing relatively modest improvements in efficiency.

The most closely related work is AxA-EBL [Cohen, 1990] mentioned above. AxA-EBL "explains" correct uses of a clause by compiling out a generalized macro for the subgoal to which the clause was applied. AxA-EBL searches a pool of approximations of these macros to find a small set of rules that maximizes coverage of positive examples and minimizes the coverage of negatives. DOLPHIN improves on AxA-EBL by using a more powerful inductive learning mechanism (FOIL) and considering the entire top-level proof as the explanation for the successful application of a clause to a subgoal.

Researchers in knowledge compilation and logic programming have examined test incorporation as an optimization technique [Braudaway and Tong, 1989; Bruynooghe *et al.*, 1989]. Research has concentrated on analytically identifying sound program transformations. The test incorporation necessary to optimize the particular programs presented in this paper is either unsound (N-queens) or would require information external to the program (transitivity of $\leq$ in naivesort) to demonstrate soundness. DOLPHIN which relies on empirical techniques, performs a wider range of optimizations automatically and can "tune" the performance of a program to the distribution of examples.

5 Future Research

Experiments with DOLPHIN have raised a number of issues that require further investigation. One shortcoming of the current approach is its critical dependence on the form of the program that is being optimized. Using an alternative definition of `permutation/2` which selects an item to be the head of the permutation and then recursively computes a permutation to be the tail would prevent DOLPHIN from learning 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. Inventing a recursive control-rule of this sort requires augmenting DOLPHIN with constructive induction techniques.

Other changes to the inductive mechanism might also prove useful. While the simplicity of the control rules learned by DOLPHIN tends to increase their utility, there is no explicit use of match-cost or operationality. An interesting avenue of investigation would be to incorporate notions of operationality into the hill-climbing mechanism to bias the search toward more efficient heuristics.

Finally, programs often use a single predicate in different ways. It these cases it might be useful to specialize the different uses of a predicate independently.

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 kind of test incorporation on logic programs. Test incorporation allows DOLPHIN to transform some intractable algorithms into ones that run in polynomial time.

Acknowledgments

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 William Cohen who provided code for the planning domains and helpful comments on an earlier version of this paper. Thanks also to Je-Nien Ernst who provided some of the data on EBL-macro.

References

[Bratko, 1990] I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison Wesley, Reading:MA, 1990.

[Braudaway and Tong, 1989] Wesley Braudaway and Chris Tong. Automatic synthesis of constrained generators. In *Proceedings of the Eleventh International Joint conference on Artificial intelligence*, Detroit, MI, 1989.

[Bruynooghe *et al.*, 1989] Maurice Bruynooghe, Danny De Schreye, and Bruno Krekels. Compiling control. *Journal of Logic Programming*, 6:135–243, 1989.

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

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

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

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

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

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

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

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

[Mitchell *et al.*, 1986] Tom M. Mitchell, Richard M. Keller, and Smadar T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[Pazzani and Kibler, 1992] M. Pazzani and D. Kibler. The utility of background knowledge in inductive learning. *Machine Learning*, 9:57–94, 1992.

[Prieditis and Mostow, 1987] A. Prieditis and J. Mostow. Prolearn: Towards a prolog interpreter that learns. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, Seattle, WA, Jul 1987.

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