Ph.D. proposal, Department of Computer Sciences,
University of Texas at Austin, March 1993

Learning Search-Control Heuristics for Logic Programs: Applications to Speedup Learning and Language Acquisition

John M. Zelle
Department of Computer Sciences
University of Texas
Austin, TX 78712
(512) 471-9589
zelle@cs.utexas.edu

March 5, 1993

Abstract

This paper presents a general framework, learning search-control heuristics for logic programs, which can be used to improve both the efficiency and accuracy of knowledge-based systems expressed as definite-clause logic programs. The approach combines techniques of explanation-based learning and recent advances in inductive logic programming to learn clause-selection heuristics that guide program execution. Two specific applications of this framework are detailed: dynamic optimization of Prolog programs (improving efficiency) and natural language acquisition (improving accuracy). In the area of program optimization, a prototype system, Dolphin is able to transform some intractable specifications into polynomial-time algorithms, and outperforms competing approaches in several benchmark speedup domains. A prototype language acquisition system, Chill is also described. It is capable of automatically acquiring semantic grammars, which uniformly incorprate syntactic and semantic constraints to parse sentences into case-role representations. Initial experiments show that this approach is able to construct accurate parsers which generalize well to novel sentences and significantly outperform previous approaches to learning case-role mapping based on connectionist techniques. Planned extensions of the general framework and the specific applications as well as plans for further evaluation are also discussed.

1 Introduction

A general goal of machine learning is automating the construction of efficient knowledge-based systems. Research has typically concentrated on either acquiring basic domain knowledge (concept learning) or improving the efficiency of a problem solver (speedup learning) (Shavlik and Dietterich, 1990). A second dichotomy has grown up around the techniques generally considered appropriate to these tasks. Research in concept learning has tended to focus on techniques of inductive, or similarity-based learning, (SBL) while speedup is often achieved through explanation-based learning (EBL). However, there is no clear separation between these goals and methods. Inductive learning can be used to improve efficiency (Mitchell et al., 1983) and analytical methods such as EBL can sometimes be used to learn new domain knowledge that improves accuracy (Flann and Dietterich, 1989).

One framework that cleanly unifies these goals and techniques is that of learning search-control rules for a problem solver. The notion of intelligence as controlled search pervades AI. Learning rules to control search may improve both the efficiency and accuracy of a problem solver. Efficiency is improved by eliminating search paths that do not lead to solutions. Such control rules act as heuristics to allow more efficient search in future problems. Accuracy can be improved by learning control rules which prune the search along paths leading to incorrect solutions. These control rules may be viewed as preconditions that were previously omitted from the search operators. Adding these preconditions results in a more accurate theory. Furthermore, control-rule learning can make good use of both SBL and EBL techniques. SBL methods can be used to learn conditions for the effective application of an operator by performing induction over the situations in which the use of the operator led to a favorable outcome. In applications where a strong domain theory is available to explain operator success, EBL techniques may be used to generalize the explanation and extract sufficient conditions for future applications of the operator.

The learning of search-control knowledge has been previously investigated primarily in the context of STRIPS-like planners (Minton, 1988) and forward-chaining production systems (Langley, 1985; Laird et al., 1986). Recently, Cohen (1990) has argued some advantages of extending the search-control learning framework into the domain of logic programming. This proposal embraces such an approach and offers a general framework for learning search-control heuristics in logic programs. The motivation for this research is three-fold.

First, logic programming provides a very well-understood representational and computational platform upon which to build. Logic programming has a firm theoretical foundation (Lloyd, 1984) and a practical instantiation in the programming language, Prolog. The value of the former is well-known. The value of the latter should not be underestimated. Efficient commercial Prolog implementations are readily avaiable, and the language is considered a viable tool for the production of real-world systems. Extending the usefulness of learning techniques into this domain has the potential of bringing these techniques more into the mainstream of application development.

Second, learning mechanisms within the logic programming framework are becoming well understood. Logic programs have been recognized as a fruitful domain for EBL techniques virtually since their inception, since the notion of "explanation" may be usefully equated with the structure of a proof. Techniques for generalizing these structures and compiling macro-rules are well known(Mitchell et al., 1986; DeJong and Mooney, 1986; Prieditis and Mostow, 1987). On the SBL side, there has been an explosion of recent research in inductive logic programming (ILP) which addresses the induction of Horn-clause concept definitions (Muggleton, 1992; Quinlan, 1990).

Third, expanding the search control framework into this new domain opens up a whole new range of potential learning applications. One immediate example is the use of speedup learning for Prolog program optimization. One of the original motivations for the logic programming paradigm was the promise of declarative programming. Unfortunately, in existing languages such as Prolog, abstract, declarative program specifications often execute too slowly to be generally useful. One possible remedy is to employ speedup learning, which involves dynamically improving the efficiency of a system as it gains experience solving problems in a domain (Tadepalli, 1992), to enhance the efficiency of programs based on experience gained from solving sample problems. In this way, declarative specifications might be dynamically transformed into efficient procedural programs.

This proposal shows how speedup may be achieved by learning search-control rules that determine which clauses in a Prolog program to use to solve particular subgoals. A combination of explanation-based generalization (EBG) (Mitchell et al., 1986; DeJong and Mooney, 1986) and ILP (FOIL (Quinlan, 1990)) techniques are used to learn these control rules. An initial version of this approach has already been implemented and tested and is capable of automatically converting an $O(n!)$ generate-and-test sorting program into an $O(n^2)$ insertion sort based on experience gained from a single example.

Other potential learning applications are those that can usefully exploit the power of a first-order, relational framework. One example of such a domain is that of natural language processing (NLP). Understanding natural language requires many sources of knowledge relating abstract entities such as words and clauses to each other and to world knowledge and "meanings." It seems at its core an inherently relational endeavor. In fact, Prolog itself evolved from an NLP framework and is still a major tool for work in computational linguistics (Gazdar and Mellish, 1989; Pereira and Shieber, 1987). Despite the power of languages such as Prolog, however, designing computer systems to "understand" natural language input is a difficult task. The laboriously hand-crafted computational grammars supporting natural language applications are often inefficient, incomplete and ambiguous. The difficulty in constructing natural language systems is yet another example of the "knowledge acquisition bottleneck" which motivates machine learning research.

This proposal shows how language acquisition can be viewed as a problem of control-rule learning. In particular, semantic grammars, which uniformly incorporate both syntactic and semantic constraints to parse sentences and produce semantic analyses (Allen, 1987; Brown and Burton, 1975), can be automatically acquired in the control-rule learning framework. Constructive induction (Michalski, 1983; Kijsirikul et al., 1992) is employed to generate syntactic and semantic word- and phrase-classes that support the parsing process. Experiments with an initial implementation demonstrate that the method is capable of learning accurate parsers that generalize well to novel sentences.

These two applications, program optimization and natural language acquisition, demonstrate the usefulness of this learning framework for improving both efficiency and accuracy. Individually, the promise of automatically converting declarative specifications into efficient, procedural programs, or substantially automating the aquisition of sophisticated natural language parsers provide significant justification for this research. Taken together, they illustrate the potential of the learning architecture to solve many practical machine learning problems.

The body of the proposal is organized as follows. Section 2 describes a basic framework for inducing control rules for logic programs. Section 3 discusses its application to speedup learning and initial results in this area. Section 4 discusses its application to language acquisition and initial results in this area. Section 5 discusses planned research for extending both the basic framework and its application in these two areas. Section 6 concludes and discusses the potential impact of the research.

2 Learning Search-control in Logic Programs

2.1 The Learning Problem

A logic program is expressed using the definite clause subset of first- order logic [1]. Computation is performed using a resolution proof strategy on an existentially quantified goal. For example, a program to sort lists might be expressed as a collection of definite clauses expressing a logical definition of the two-place predicate, `sort`. A goal of the form `sort(X,Y)` is taken to be true exactly when `Y` is a sorted version of the list represented by `X`. When using a logic program to do useful computation, the arguments of the top-level goal are typically partitioned into *input* and *output* argument sets. The program is executed by providing a goal which has its input arguments instantiated. A theorem prover contructively proves the existence of a goal meeting these constraints and produces bindings for the output arguments. In our sorting example, the first argument may be considered an input, and the second an output. Given a correct definition of sort, proving the goal `sort([9,1,5,3,4],Y)` produces the output binding: `Y = [1,3,4,5,9]`.

Prolog provides a particular implementation of logic programming using a very simple control strategy. Depth-first search with simple backtracking is used to search for a proof. Multiple solutions may be found for a given goal by asking the prover to backtrack and find alternative bindings for the output arguments. It is often useful to distinguish between programs that are *deterministic* and those that are *nondeterministic*. A deterministic program is one that computes a (partial) function; that is, it finds at most a single solution for each unique instantiation of its input arguments. A nondeterministic program computes a relation, admitting multiple solutions.

Search-control in a logic program can be viewed as a *clause selection* problem (Cohen, 1990). Clause selection is the process of deciding which of several applicable program clauses should be used to reduce a particular subgoal during the course of a proof. If program clauses are always applied appropriately, the program executes deterministically (without backtracking) and produces only correct solutions. A framework for control-rule learning in Prolog programs is illustrated in Figure 1. The input to the learning system is a Prolog program and a set of training examples, which are fully instantiated examples of the top-level program goal. The output is a modified program that incorporates learned clause-selection heuristics. In the discussion that follows, it is assumed that the set of training examples is *output complete*. By this it is meant that the set of training examples includes every correct solution for each unique instantiation of input arguments appearing in the set.

When performing pure speedup learning, the initial program is correct, but potentially inefficient. The set of training examples can be generated by using the initial program to enumerate the solutions to a set of prototypical problems. For a deterministic program, each solved problem will generate a single (fully instantiated) training example. In the case

[1] A definite clause is a disjunction of literals having exactly one unnegated literal, called the *head*. The negated literals comprise the clause *body*

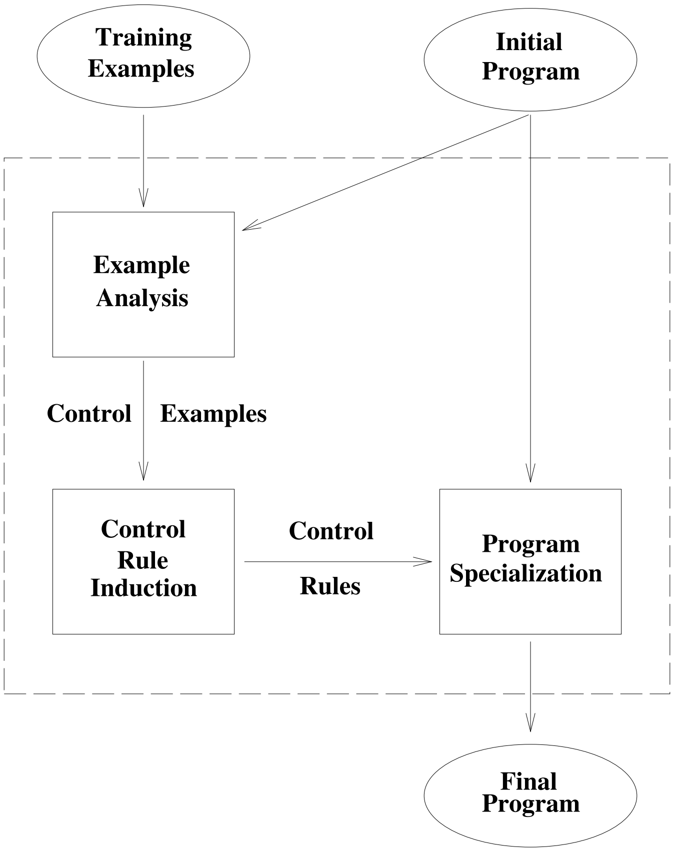


Figure 1: Learning Clause Selection Heuristics

of a nondeterministic program, each problem may give rise to multiple training examples. Typically, applications of speedup learning are only concerned with finding a single solution to each problem, even though the program may admit multiple solutions. In this case, the first solution found may be considered the "correct" output, and the other solutions ignored, effectively rendering the program deterministic. In any case, the initial program is specialized to solve similar problems more efficiently.

When learning to improve accuracy the input program must be appropriately over-general. Presented with a goal having only its input arguments instantiated, the program must be capable of producing the correct output(s) for that input, but may produce many incorrect solutions as well. The program is specialized to produce only correct solutions.

The three phases of the algorithm, *example analysis*, *control rule induction*, and *program specialization*, are explained and illustrated by way of a simple example in the following subsections.

2.2 Example Analysis

During example analysis, the training examples are utilized to identify clauses of the program that give rise to unwanted backtracking or incorrect solutions. Examples of correct and incorrect clause applications are extracted as *control examples* for subsequent control rule induction. A control example for a clause is a (partially) instantiated subgoal to which the

```
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|Ys0],[Y|Ys]) :-  insert(X,Ys0,Ys).

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

Figure 2: Naive Sorting Program

clause was applied at some point during the search for a proof of a training example.

Positive control examples are generated by finding the first proof for each training example using the initial program. The clause applications actually used in the completed proof are retraced, but with the goal having only its input arguments instantiated. For each clause application in the proof, a "snapshot" of the subgoal to which it is applied is saved as a positive control example for that clause. These positive control examples reflect the subgoals to which the clause should be applied when the program is actually being used to generate the expected output from the given input.

The set of positive control examples along with the assumption of output completeness in the training examples also implicity defines a set of negative control examples. Knowing the set of clauses which should be applied to a given subgoal identifies other clauses as those that should *not* be applied. The exact mechanism for generation of negative control examples is dependent on whether the output program is intended to be nondeterministic (generate multiple solutions). For deterministic programs, the positive examples for a clause are considered negative examples for all *prior* clauses which do not have the same positive example. Since only one solution is being computed, subsequent clauses will not have a chance to match against this particular subgoal, and it need not be included in their negative example sets. For nondeterministic programs, a positive example for one clause is considered a negative example for *all* matching clauses which do not also have that subgoal as a positive example. This is necessary because subsequent clauses may be matched against this subgoal when backtracking for more solutions. These subsequent clauses should not be applied unless they lead to a proof of an alternative solution (in which case they will also have this subgoal as a positive control example extracted from the proof of a different training example).

As an example, consider the naive sorting program in Figure 2, 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, `insert(Item, List1, List2)`, holds when `List2` is `List1` with `Item` inserted at an arbitrary location. In `naivesort`, `insert` is called with its first and second arguments instantiated to insert items into the permutation, which is returned in the third argument.

Table 1: Control Examples for First `insert` Clause

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)`

Although the top-level goal, `naivesort`, is intended to be determinisitic, computation proceeds nondeterministically by generating successive permutations. The nondeterminism of the `permutation` predicate actually arises from the definition of `insert`. Either clause of `insert` may be used to reduce any `insert` subgoal. This nondeterminism could be eliminated by learning a control rule for the first clause that accurately predicts the situations in which an item should be placed at the front of the list. Given the training example, `naivesort([9,1,5,3,4],[1,3,4,5,9])`, the example analysis phase discovers five examples of correct uses of the clause and six failed attempts. These control examples, shown in Table 1, represent the concept `useful_insert_1`, that is, subgoals to which the first clause of `insert` should be applied. The positive control examples are the subgoals which were reduced by the first `insert` clause. The negative control examples for this clause are the subgoals which were solved using the second clause of `insert`.

2.3 Control Rule Induction

The goal of control-rule induction is to produce a set of control rules specifying the contexts in which it is useful to apply clauses of the initial program. For each program clause, C, a definition of the concept, "subgoals for which C is useful," is needed. Thus, control rule learning is viewed as relational concept learning over the control examples. Given the positive and negative control examples for a clause, the specific induction task is to find a Horn-clause definition which covers all the positive examples, and as few negatives as possible. There is a growing body of research in inductive logic programming (ILP) which addresses this problem (Muggleton, 1992; Quinlan, 1990). The specific induction algorithm used may be tailored to the goals of the learning system and the available knowledge about the domain task. Two substantially different induction algorithms are employed in the speedup and natural language acquisition applications described below.

Continuing with the naivesort example, a suitable induction algorithm might produce a control rule such as:

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

This simple definition correctly classifies the examples in Table 1. It states that the first clause of `insert` should be used to insert an element into a list that is empty or has a head

```
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 3: Improved Insert Predicate

at least as large as the element being inserted.

2.4 Program Specialization

In the program specialization phase, the initial Prolog program is modified using the learned control rules so that attempts to use a clause inappropriately fail immediately. In this way, the search space of the Prolog solver is pruned to efficiently produce correct solutions, effectively utilizing the control information without incurring the overhead of a separate interpreter.

For non-disjunctive (single clause) control rules, the learned conditions are simply placed into the 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 clauses of the learned control 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 control information has made the program clause deterministic. If the learned control rules cover no negative control examples 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` shown in Figure 3. In effect, `permutation` has been modified to produce ordered permutations. Careful inspection shows that this is a version of the insertion sort algorithm, and an $O(n!)$ sort has been "optimized" into an $O(n^2)$ version by learning and incorporating suitable control rules.

3 Application: Program Speedup with DOLPHIN

3.1 The Learning Problem

One application of this learning framework is Prolog program optimization. The naivesort example above illustrates how the learning of clause selection heuristics effects a form of *test* *incorporation* (Dietterich, 1986), which can sometimes dramatically enhance the efficiency of an algorithm. Our specific learning problem then is: given a Prolog program and a set of training problems, produce a new program which incorporates clause selection heuristics to solve the training problems more efficiently.

Standard EBL methods can be applied to learn clause selection heuristics for Prolog programs; 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 (1990) handles this problem by combining EBL with induction to learn a small set of "approximate" control rules with reduced match cost. His method, AxA-EBL, was shown to out-perform standard EBL control rules across a number of problem solving domains.

The DOLPHIN, (Dynamic Optimization of Logic Programs through Heuristics INduction) system can be viewed as an extension of the AxA-EBL approach. DOLPHIN improves on AxA-EBL in two significant ways. First, it employs a more powerful induction algorithm, namely Quinlan's FOIL (Quinlan, 1990). Second, DOLPHIN performs this induction in a more expansive proof context, allowing it to learn a wider variety of control heuristics.

3.2 Control Rule Induction in DOLPHIN

Control rule learning in DOLPHIN amounts to finding an *operational* (i.e. efficiently executable) definition for the class of subgoals to which a given clause should be applied. DOLPHIN adopts a FOIL-like (Quinlan, 1990) framework to perform this induction.

```
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 4: FOIL Covering Algorithm

FOIL may be viewed as a simple covering algorithm which has the basic form shown in Figure 4. 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 that maximizes an information-based gain heuristic. DOLPHIN employs the FOIL framework but focuses the clause construction by replacing the exponential search through all possible literals with a search through literals in the generalized proofs of training examples. Standard EBG techniques (DeJong and Mooney, 1986; Mitchell et al., 1986) are used to generalize the proof trees of the training examples, removing those elements of a proof that are dependent on the specific facts of the example while maintaining its overall structure. These generalized proofs are then used in a modified clause construction step.

Consider 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. In this way, successive clause specializations are always motivated by examining successful control decisions from the training examples.

The clause specialization search algorithm is detailed in Figure 5. SPEC-PAIRS is a list

```
Let PC be the original program clause
Let PROOFS be the set of generalized proofs
Let C be the (control rule) 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 5: Clause Specialization Algorithm

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.

As a concrete example, control rule induction for the naivesort problem is straight-forward. The generalized proof of the single training example is shown in Figure 6. The

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

generalized proof is created by "retracing" the proof steps on a goal having an uninstantiated input argument.

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. 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 inis 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 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 exactly the definition of `useful_insert_1` which was presented earlier.

3.3 Preliminary Experimental Results

The DOLPHIN system has been evaluated on five problem domains: naivesort, 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]. This makes the program 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).

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 of (Fikes et al., 1972). BW is generated from the blocks world operators of (Nilsson, 1980). 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.

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 that 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, the strategy used by (Cohen, 1990) of retaining the original clauses is adopted. In testing, the problem is first attempted 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).

The results of the experiments on these problems are graphically summarized in Figure 7 and Figure 8. In all domains tested, 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 test set.

It is not surprising that DOLPHIN produces programs for the sorting problem that are significantly faster. It has already been 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 DOLPHIN learned rules that were more heuristic. As can be seen from the graph, these heuristics were very effective in pruning the search space of the larger problems.

The timing graphs for μLEX, RW and BW compare DOLPHIN to three other approaches to speedup: 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.

The curves labeled "AxA-EBL" and "EBL-Control" in these graphs are taken from (Cohen, 1990) and represent only a rough comparison. The overall times reported there are considerably longer than those of our implementation; the values shown here have been 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.

On the μLEX problem Cohen found that AxA-EBL and EBL-Control performed essentially identically. Our results suggest that the overall performance level achieved by DOLPHIN is similar, but that DOLPHIN converges to an efficient program more quickly (i.e., having seen fewer examples). The lack of utility for the macro approach on this problem is probably due to the relative simplicity of the problems in the testing set; the match cost of the macros

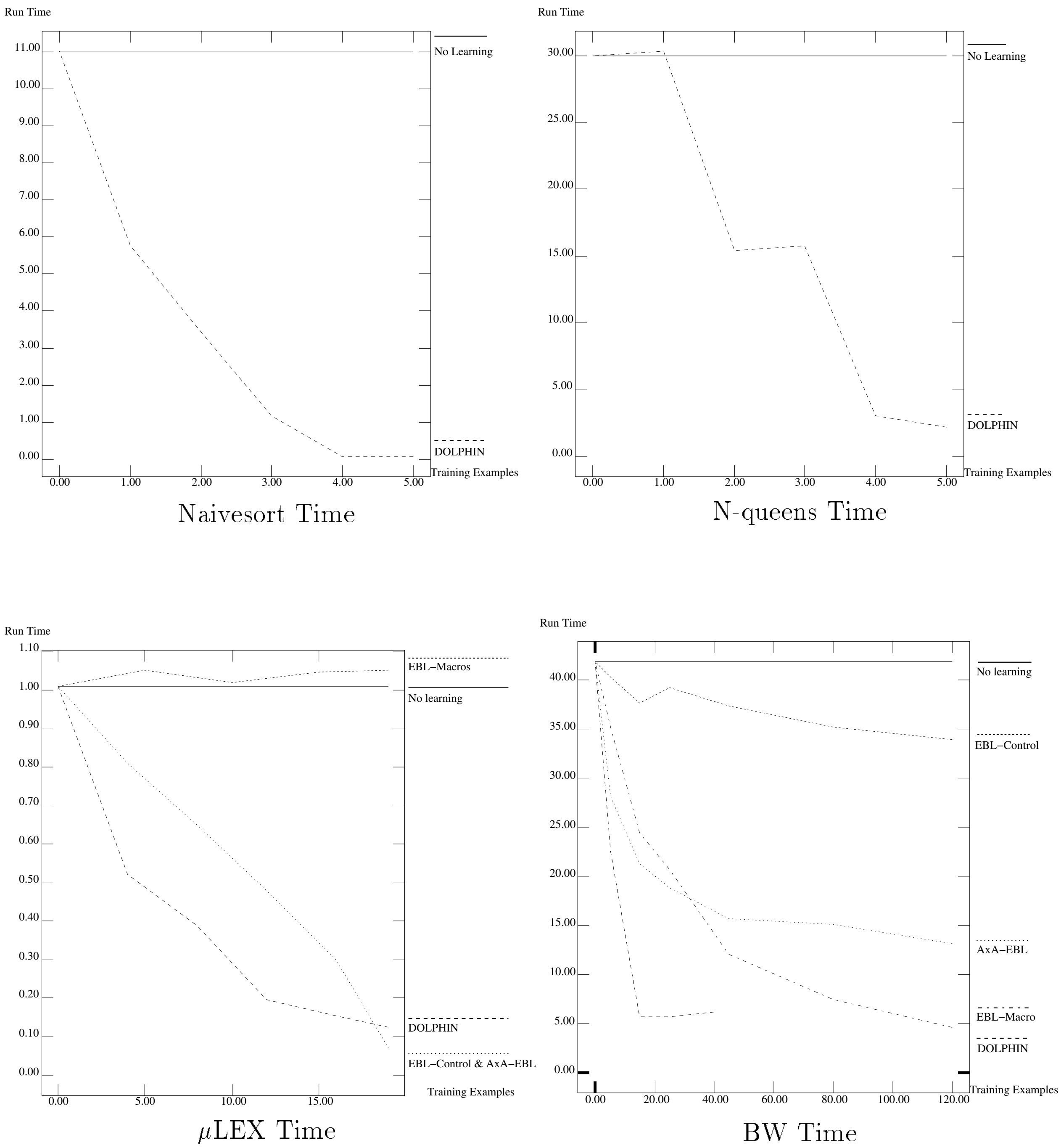
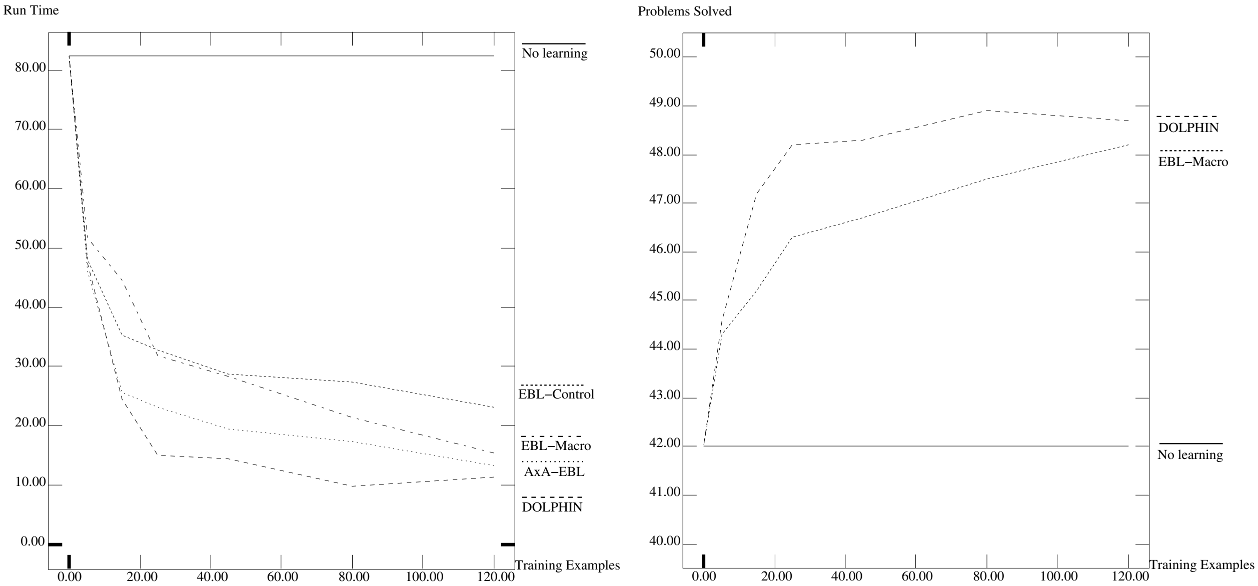


Figure 7: Performance Curves for Naivesort, N-Queens, μLEX and BW



RW Time RW Coverage

Figure 8: Performance and Coverage Curves for RW

was not significantly lower than the cost of proving an example from scratch.

In the planning problems, EBL-Macro fared surprisingly well, bettering all but DOLPHIN in BW and doing only slightly worse than AxA-EBL on RW. This demonstrates that there is a fair amount of regularity in the space of problems generated for these domains. This is probably an artifact of the limits imposed on the size of the generated problems. In both planning domains, DOLPHIN achieved results at least as good as those of the other systems, and reached maximum speedup with fewer training problems. One problem encountered in these tests was that the current implementation of DOLPHIN is memory intensive, and training sets larger than 40 were not runnable on BW due to limitations of the Prolog system. Nevertheless, the speedup achieved at 15 training problems is already significantly better than the reported results for AxA-EBL training on 120 examples.

In the RW domain, Cohen used a time bound of 60 seconds for each problem with the results being that approximately 1/7th of the generated problems could not be solved by the original planner. In order to make our results comparable, the time bound was reduced to 8 seconds to reflect the increased speed of our solver. Most of the speedup achieved in this domain comes from solving initially unsolvable problems in the testing set. To force DOLPHIN to learn control rules that generalized to larger problems, it was necessary to turn off the time limit during training. The righthand graph in Figure 8 shows the relative performance of DOLPHIN and EBL-Macro in terms of test problems solved. Similar overall coverage results were not reported for the other systems.

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. 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 `insert`.

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` 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.

3.4 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) which used a simple technique for removing conditions. Most other recent investigations have not focussed on learning control-rules (Ellman, 1988; Tadepalli, 1989; Chien, 1989) or have not employed induction (Chase et al., 1989).

Yoo and Fisher (Yoo and Fisher, 1991) combine induction and explanation to improve performance in a problem-solving framework. They enhance the utility of EBL-macros by clustering them in a COBWEB-style classification tree maintaining explanations at various levels of detail. By contrast, DOLPHIN uses supervised learning methods to acquire explicit search-control rules.

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. 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 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 by considering the entire proof of a top-level goal as the explanation for the successful application of a clause to a particular subgoal.

Several researchers in knowledge compilation and logic programming have examined analytical methods for test incorporation as an optimization technique (Braudaway and Tong, 1989; Bruynooghe et al., 1989). 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.

4 Application: Language Acquisition with CHILL

4.1 The Learning Problem

4.1.1 Acquisition as learning mappings

Most computational research in language acquisition has concentrated on learning the syntax of a language from sample sentences (Anderson, 1977; Berwick, 1985). In practice, natural language systems are typically more concerned with extracting the meaning of a sentence, usually expressed as a case-role structure. Semantic grammars, which uniformly incorporate both syntactic and semantic constraints to parse sentences and produce semantic analyses, have proven extremely useful in constructing natural language interfaces for limited domains (Allen, 1987; Brown and Burton, 1975). An interesting question for machine learning is whether such grammars can be automatically constructed from an analysis of examples in a given domain. The language acquisition problem for our purposes is: Given a corpus of sentences paired with semantic representations of the sentence meaning, construct a small, efficient parser that is capable of mapping sentences into meanings.

This perspective on language acquisition differs somewhat from typical approaches. The fundamental question in human language acquisition research is the "projection problem." That is, how is it that all humans develop an ability to communicate in their native tongue given the relatively impoverished, unsystematic learning inputs available to children. In particular, how can errors in a grammar be fixed without the presence of negative examples? Research in generative linguistics has focused on abstract characterizations of the phonemic and syntactic structure of natural languages in an attempt to indentify innate "universals" which constrain the types of grammatical hypotheses entertained by children during acquisition.

Rather than focusing on the structure of language per se, the alternative approach taken here is to consider the implications of acquiring of an efficient, deterministic parser. Shift-reduce parsing has proven to be an efficient mechanism for natural language analysis (Tomita, 1986), and deterministic variants have been argued to accurately model certain aspects of human language processing (Marcus, 1980). A reasonable beginning, then, is to posit an underlying shift-reduce parsing mechanism capable of mapping input sentences onto appropriate representations and to learn control rules that guide this mapping. In this respect our approach is similar to others, notably (Berwick, 1985; Simmons and Yu, 1992). However, our research differs in that it applies new learning techniques and concentrates on semantic rather than syntactic analyses.

The approach offers several advantages. Selection of an appropriate language processing mechanism allows for the generation of implicit negative examples (see Section 2.2) which can guide grammar induction and prevent over-generalization. Making use of these negative examples allows the lifting of many "hard-wired" constraints based on specific theories of grammar (e.g., the X-bar assumptions of (Berwick, 1985)). Further benefit can be gained from the use of modern induction techniques, particularly those incorporating constructive induction. Appropriate word-classes can be automatically acquired rather than being given to the system *a priori*. Finally, concentrating on semantic representations produces structures likely to be useful in natural language applications, and allows syntactic and semantic knowledge to be learned and incorporated in a uniform way; there is no arbitrary distinction between syntactic and semantic levels of analysis.

In this view, meaning and communication are paramount, and language structure arises from the need for efficient methods of conveying meaning. This alternative approach to acquisition is not necessarily conflicting. To the extent that the underlying principles of efficient parsability and tractable induction produce parsers which obey the constraints of the so called "universal" grammar, they may be considered as deeper explanations for these constraints. In essence, typical linguistic theories may eventually be seen as epiphenomena of the underlying principles of information and learning interacting with simple underlying language processing mechanisms.

4.1.2 A specific application: case-role mapping

One example of the type of analysis that might be desired is simple case-role mapping. Traditional case theory (Fillmore, 1968) decomposes a sentence into a proposition represented by the main verb and various arguments such as agent, patient, and instrument, represented by noun phrases. The basic mapping problem is to decide which sentence constituents fill which roles. Though case analysis is only a part of the overall task of sentence interpretation, the problem is nontrivial even in simple sentences.

Consider these sentence/case-analysis examples from (McClelland and Kawamoto, 1986):

1.	The boy hit the window.	[hit agent:boy patient:window]
2.	The hammer hit the window.	[hit insrument:hammer patient:window]
3.	The hammer moved.	[moved patient:hammer]
4.	The boy ate the pasta with the cheese.	[ate agent:boy patient:[pasta accomp:cheese]]
5.	The boy ate the pasta with the fork.	[ate agent:boy patient:pasta instrument:fork]

In the first sentence, the subject, `boy`, is an agent. In the second, the subject, `hammer`, is an instrument. The role played by the subject must be determined on the grounds that boys are animate and hammers are not. In the third sentence, the subject, `hammer`, is interpreted as a patient, illustrating the importance of the relationship between the surface subject and the verb. In the last two sentences, the prepositional phrase could be attached to the verb (making `fork` an instrument of `ate`) or the object (`cheese` is an accompaniment of `pasta`). Semantic knowledge is required to make the correct assignment.

4.1.3 Previous approaches to learning case-role mapping

Recent research in learning the case-role assignment task has taken place under the connectionist paradigm (Miikkulainen and Dyer, 1991; St. John and McClelland, 1990; McClelland and Kawamoto, 1986). They argue that proper case-role assignment is a difficult task requiring many independent sources of knowledge, both syntactic and semantic, and therefore well-suited to connectionist techniques.

The work of Miikkulainen and Dyer (1991), who used the case-role mapping task to demonstrate their FGREP method, is illustrative. Their model employs a recurrent network which allows words of an input sentence to be processed sequentially. Following (McClelland and Kawamoto, 1986), the network output has fixed slots for verb, agent, instrument, patient and modifier (a slot for the modifier of a patient, such as "cheese" in "pasta with cheese"). The network is trained using a modification of backpropagation that automatically develops distributed word encodings during the training process. Words are presented to, and read out of, the network using these learned encodings. The model was demonstrated using a set of 1475 sentence/case-structure pairs originally from (McClelland and Kawamoto, 1986) (hereafter referred to as the M & K corpus). The corpus was produced from a set of 19 sentence templates generating sentences/case-structure pairs of the type illustrated above.

Connectionist models face a number of difficulties in handling natural language. Since the output structures are "flat" (nonrecursive) it is unclear how the embedded propositions in more sophisticated analyses can be handled. The models are also limited to producing a single output structure for a given input. If an input sentence is truly ambiguous, the system produces a single output that appears as a weighted average of the possible analyses, rather than enumerating the consistent interpretations. The symbolic techniques advocated here overcome these deficiencies. In addition, empirical results demonstrate that the system trains faster and generalizes to novel inputs better than its neural counterparts.

4.2 Control Rule Induction in CHILL

Our system, CHILL, (Constructive Heuristics Induction for Language Learning) is a general approach to semantic-grammar acquisition. The input to the system is a set of training instances consisting of sentences paired with the desired case representations. The output is a shift-reduce parser (in Prolog) which maps sentences into case representations. The parser may produce multiple analyses (on backtracking) for a single input sentence, allowing for ambiguous sentences in the training set.

Applying the clause selection framework to this problem requires automating the construction of an initial program. Thus, the CHILL algorithm consists of two distinct tasks. First, the training instances are used to formulate an overly-general parser which is capable of producing case structures from sentences. The initial parser is overly-general in that it also produces many spurious analyses for any given input sentence. This initial program is then specialized to produce only correct parses. The mechanism for inducing control rules employs constructive induction to invent word and phrase categories which support the parsing process. The next two subsections detail these two tasks.

4.2.1 Creating an Overly General Parser

Our system adopts a simple shift-reduce framework for case-role mapping (Simmons and Yu, 1992). The process is best illustrated by way of example.

Consider the sentence: "The man ate the pasta." Parsing begins with an empty stack and an input buffer containing the entire sentence. At each step of the parse, either a word is shifted from the front of the input buffer onto the stack, or the top two elements on the stack are popped and combined to form a new element which is pushed back onto the stack. The sequence of actions and stack states for our simple example is shown in Figure 9. The action notation *(x label)*, indicates that the stack items are combined via the role, *label*, with the item from stack position, x, being the head. An advantage of assuming such a constrained parsing mechanism is that the form of structure building actions is limited, and

Action	Stack Contents
	[]
(shift)	[the]
(shift)	[man, the]
(1 determ)	[[man, determ:the]]
(shift)	[ate, [man, determ:the]]
(1 agent)	[[ate, agent:[man, determ:the]]]
(shift)	[the, [ate, agent:[man, determ:the]]]
(shift)	[pasta, the, [ate, agent:[man, determ:the]]]
(1 determ)	[[pasta, determ:the], [ate, agent:[man, determ:the]]]
(2 patient)	[[ate, patient:[pasta, determ:the], agent:[man, determ:the]]]

Figure 9: Shift-Reduce Parsing of "The man ate the pasta."

the operations required to construct a given case representation are directly inferable. In general, a structure building action is required for each unique case-role that appears in the analysis.

A shift-reduce parser is easily represented as a logic program. The state of the parse is reflected by the contents of the stack and input buffer. Each distinct parsing action becomes an operator that takes the current stack and input and produces new ones.

Figure 10 shows the overly-general program to parse the above example. The `parse` predicate takes a list of words representing a sentence and returns a case structure. The `maps_to` predicate maps a stack and input buffer in its first two arguments into a new stack and buffer in the third and fourth arguments. The mapping is performed by zero or more applications of simple actions represented by `op`. For example, the first clause of `op` implements the *(1 agent)* action. The final predicate, `combine`, simply attaches a value to a head via some label to produce a new structure.

Extending the program to parse further examples is accomplished by adding additional clauses to the `op` predicate. Each clause is a direct translation of a required parsing action. As was mentioned earlier, the identification of the necessary actions is straight-forward. A particularly simple approach is to include two actions (e.g., *(1 agent)* and *(2 agent)*) for each role used in the training examples; any unnecessary operator clauses will be removed from the program during the subsequent specialization process.

4.2.2 Control Induction Algorithm

The overly-general parser produces a great many spurious analyses for the training sentences because there are no conditions specifying when it is appropriate to use the various operators. Incorporating appropriate control heuristics can yield a correct, efficient parser. For example, the *(1 agent)* clause is typically modified to:

```
op([A,[B,det:the]],C,[D],C) :- animate(B), combine(A,agent,B,D).
animate(boy). animate(girl). animate(man). animate(lion) . . .
```

```
parse(S, Parse) :- maps_to([], S, [Parse], []).

maps_to(Stack, Input, Stack, Input).
maps_to(Stack0, In0, Stack, In) :-
    op(Stack0, In0, Stack1, In1), maps_to(Stack1, In1, Stack, In).

op([Top,Second|Rest],In,[NewTop|Rest],In) :- combine(Top,agent,Second,NewTop).
op([Top,Second|Rest],In,[NewTop|Rest],In) :- combine(Top,determ,Second,NewTop).
op([Top,Second|Rest],In,[NewTop|Rest],In) :- combine(Second,patient,Top,NewTop).
op(Stack,[Word|Words],[Word|Stack],Words).  % Shift operation.

combine([Head|Pairs], Label, Value, [Head,Label:Value|Pairs]) :- !.
combine(Head, Label, Value, Struct) :- combine([Head], Label, Value, Struct).
```

Figure 10: Overly-General Parser for "The man ate the pasta."

Here, a new predicate has been invented representing the concept "animate."[2] This rule may be roughly interpreted as stating: "If the stack contains two items, the second of which is a completed noun phrase whose head is animate, then attach this phrase as the agent of the top of stack."

The CHILL induction algorithm may be viewed as a composite of proven techniques from other systems, notably GOLEM (Muggleton and Feng, 1992), FOIL (Quinlan, 1990), and CHAMP (Kijsirikul et al., 1992). The intuition is that we want to find a small (hence general) definition which discriminates between the positive and negative examples. We start with a most specific definition (the set of positive examples) and introduce generalizations which make the definition more compact (as measured by a CIGOL-like size metric (Muggleton and Buntine, 1988)). The search for more general definitions is carried out in a hill-climbing fashion. At each step, a number of possible generalizations are considered; the one producing the greatest compaction of the theory is implemented, and the process repeats. The basic algorithm is outlined in Figure 11.

The heart of the algorithm is the `Find_Generalization` procedure. It takes two clauses in the current definition and constructs a new clause that subsumes them and does not cover any negative examples. `Reduce_Definition` proves the positive examples using the current definition augmented with the new generalized clause. Preferential treatment is given to the new clause (it is placed at the top of the Prolog definition) and any clauses which are no longer used in proving the positive examples are deleted to produce the reduced definition.

The `Find_Generalization` algorithm is shown in Figure 12. The first step is to construct the least general generalization (LGG) (Plotkin, 1970) of the input clauses. If the LGG does not cover any negative examples, no further refinement is necessary. If the clause is too general, an attempt is made to refine it using a FOIL-like mechanism which adds literals derivable either from background or previously-invented predicates. If the resulting clause is still too general, it is passed to `Invent_Predicate` which invents a new predicate to

[2]Invented predicates actually have system generated names. They are renamed here for clarity.

```
Let Pos := Positive Examples
Let Neg := Negative Examples
Let Def := Positive examples considered as unit clauses.
Repeat
  Let OldDef := Def
  Let S be a sampling of pairs of clauses in OldDef,
  Let OldSize := TheorySize(OldDef)
  Let CurrSize := OldSize
  For each pair of clauses <C1, C2> in S
        Find_Generalization(C1, C2, Pos, Neg, NewClause, NewPredicates)
        Reduce_Definition(Pos, OldDef, NewClause, NewPredicates, NewDef)
        if TheorySize(NewDef) < CurrSize then
            CurrSize := TheorySize(NewDef)
            Def := NewDef
Until CurrSize = OldSize % No compaction achieved
Return Def
```

Figure 11: CHILL Induction Algorithm

```
Lgg = clause_lgg(C1, C2)
if not(covers_some_example(Lgg, Neg)) then
        NewClause := Lgg
        NewPredicates = {}
else % attempt to specialize Lgg
   PosToCover := examples in Pos provable by C1 or C2
   SpecLgg := add_antecedents(Lgg, PosToCover, Neg)
   if not(covers_some_example(SpecLgg, Neg)) then
        NewClause := SpecLgg
        NewPredicates := {}
   else % New predicate needed
        CoveredNeg := examples in Neg provable by SpecLgg
        Invent_Predicate(SpecLgg, PosToCover, CoveredNeg, NewPredicates, Ante)
        NewClause := SpecLgg + Ante
```

Figure 12: Algorithm for `Find_Generalization`

discriminate the positive examples from the negatives which are still covered.

Predicate invention is carried out in a manner analogous to CHAMP. The first step is to find a minimal-arity projection of the clause variables such that the set of ground tuples generated by the projection when using the clause to prove the positive examples is disjoint with the ground tuples generated in proving the (still covered) negative examples. These ground tuple sets form the positive and negative example sets for the new predicate. The top-level induction algorithm is recursively invoked with these examples to create a definition of the predicate. The method differs from CHAMP in that it uses a greedy search to find a near-minimum projection of variables that separates the examples and minimizes the set of positive examples for the new predicate.

The actual induction algorithm employed by CHILL is slightly more complicated than so far presented. As the above discussion indicates, the process of of finding a generalization of two clauses involves three steps: form an LGG, specialize, and invent a predicate. These three steps involve increasingly more computational effort. In order to conserve this effort, the outer compaction loop is first performed only attempting the LGG construction step. When no more compaction is achieved, the system resorts to compaction via LGG and specialization. Finally, compaction is attempted using predicate invention. With this "iterative-deepening" approach, significant compaction may be achieved with minimal computational effort, reducing the size of the theory on which subsequent (more intensive) processing is required.

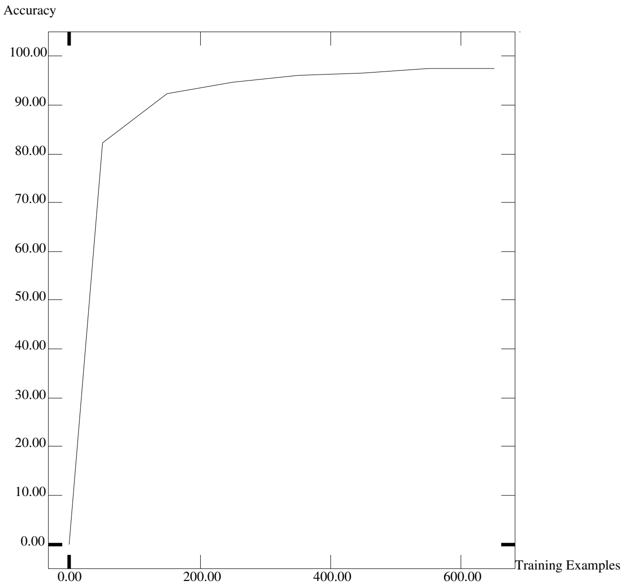
4.3 Preliminary Experimental Results

4.3.1 Experiments with the M & K corpus

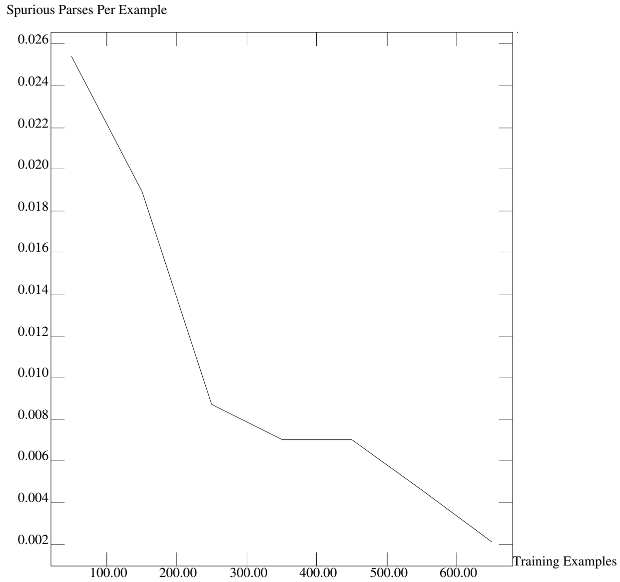
A prototype version of CHILL has been tested on several case-role assignment tasks. In the first experiment, CHILL was tried on the baseline task reported in (Miikkulainen and Dyer, 1991) using the 1475 sentence/case-structure examples from the M & K corpus. The sample actually comprises 1390 unique sentences, some of which allow multiple analyses. Since the parser is capable (through backtracking) of generating all legal parses for an input, training was done considering each unique sentence as a single example. If a particular sentence was chosen for inclusion in a training or testing set, the pairs representing all correct analyses of the sentence were included in that set. This guaranteed that the training examples were output complete.

Training and testing followed the standard paradigm of first choosing a random set of test examples (in this case 740) and then creating parsers using increasingly larger subsets of the remaining examples. All reported results reflect averages over five trials. During testing, the parser was used to enumerate all analyses for a given test sentence. Parsing of a sentence can fail in two ways: an incorrect analysis may be generated, or a correct analysis may not be generated. In order to account for both types of inaccuracy, a metric was introduced to calculate the "average correctness" for a given test sentence as follows: $Accuracy = (\frac{C}{P} + \frac{C}{A})/2$ where P is the number of distinct analyses produced, C is the number of the produced analyses which were correct, and A is the number of correct analyses possible for the sentence.

CHILL performs very well on this learning task as demonstrated by the learning curve



Accuracy



Number of Spurious Analyses

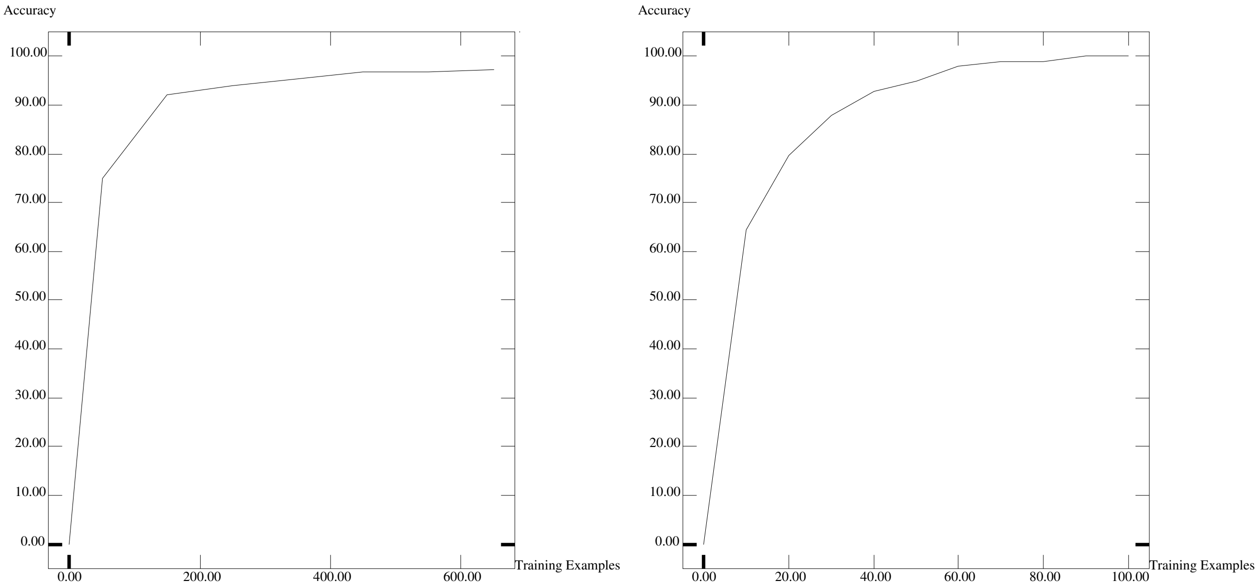
Figure 13: CHILL Experiment 1: M & K corpus

shown in the left half of Figure 13. The system achieves 92% accuracy on novel sentences after seeing only 150 training sentences. Training on 650 sentences produces 98% accuracy. The system also exhibits the desirable property that it tends to produce very few inaccurate parses. The right half of Figure 13 shows the number of spurious parses per sentence as a function of training set size.

This initial experiment, following the example in (Miikkulainen and Dyer, 1991), did not use distinct tokens for different senses of ambiguous words. However, one of the original motivations for connectionist approaches was the ability to handle lexical ambiguity (McClelland and Kawamoto, 1986; St. John and McClelland, 1990). The M & K corpus was explicitly designed with the ambiguous lexical items, "bat" (flying vs. baseball) and "chicken" (live-animal vs. dead-food). A second experiment was carried out using distinct tokens for different word senses in the case representations. The parsing model was extended to include independent shift operators for each sense of ambiguous words. Using this modification, the first experiment was repeated. The results are shown in the left-hand graph of Figure 14. The curve is virtually identical to that of the first experiment showing that CHILL can successfully incorporate lexical disambiguation within the case-role mapping task.

It is also worth noting that CHILL consistently invented interpretable word classes. One example, the invention of `animate`, has already been presented. This concept is implicit in the analyses presented to the system, since only animate objects are assigned to the agent role. Other invented classes clearly picked up on the distribution of words in the input sentences. The system regularly invented semantic classes such as `human`, `food`, and `possession` which were used for noun generation in the M & K corpus.

Phrase classes useful to making parsing distinctions were also invented. For example, the structure `instrumental_phrase` was invented as:



M & K corpus with Disambiguation   Active/Passive Corpus

Figure 14: CHILL Experiments 2 and 3

```
instr_phrase([]).
instr_phrase([with, the, X]) :- instrument(X).
instrument(fork). instrument(spoon). instrument(bat). ...
```

It was not necessary in parsing the M & K corpus to distinguish between instruments of different actions, hence instruments of various verbs such as "hit" and "ate" are grouped together. Where the semantic relationship between words is required to make parsing distinctions, such relationships can be learned. CHILL created one such relation: `can_possess(X,Y) :- human(X), possession(Y)`; which reflects the distributional relationship between humans and possessions present in the M & K corpus. Notice that this invented rule itself contains two invented word categories.

Although there is no *a priori* reason to suppose CHILL must invent interpretable categories, the naturalness of the invented concepts supports the empirical results indicating that CHILL is making the "right" generalizations.

4.3.2 Comparison with previous results

Connectionist learning curves for the case-role mapping problem tend to be expressed in terms of low level measures such as "number of correct output bits," whereas the result important to us is whether the system generates correct parses for novel examples. The closest comparison on these experiments can be made with the results in (Miikkulainen and Dyer, 1991) where an accuracy of 95% was achieved at the "word level" training with 1439 of the 1475 pairs from the M & K corpus. Since the output contains five slots, assuming independence of errors gives an estimate of 0.95^5 or only 78% completely correct parses. One must be cautious in these comparisons as the types of inaccuracies differ substantially. Neural networks always produce an output many of which contain minor errors, whereas

```
Show me the two star hotels in downtown LA with double rates below 65 dollars.

[show, theme:[hotels, det:the,
                      type:[star, mod:two],
                      loc:[la, casemark:in, mod:downtown],
                      attr:[rates, casemark:with, mod:double,
                                 less:[nbr(65), casemark:below,
                                                unit:dollars]]]
       dative:me]
```

Figure 15: Case-structure Example from Tourist Domain

CHILL tends to produce a correct output or none at all. From an engineering standpoint, it seems advantageous to have a system which "knows" when it fails; connectionists might be more interested in failing "reasonably."

With respect to training time, the induction algorithm employed by CHILL is a prototype implemented in Prolog. Running on a SparcStation 2, the creation of the parsers for the examples in this paper required from a few minutes to half an hour of CPU time. This compares favorably with backpropagation training times usually measured in hours or days.

As a further comparison to connectionist approaches, CHILL was used to duplicate a generalization experiment reported in (St. John and McClelland, 1990). This experiment used an artificial corpus created from ten reversible actions and ten people, yielding sentences such as: "John saw Mary." These sentences could appear in either active or passive voice resulting in a total of 2000 such sentences. When trained with 1750 of the 2000 sentences, their neural model processed the remaining 250 sentences with 97% accuracy. An average learning curve over five trials for CHILL is shown on the right-hand graph of Figure 14. The testing set in this case consisted of 1900 sentences. As can be seen from the graph, CHILL performs significantly better than its neural counterpart, achieving 100% accuracy after training on only 90 examples.

4.3.3 A "realistic" example

These initial experiments were run with small corpora specifically designed to illustrate the case mapping problem. As such, they do not necessarily reflect the true difficulty of semantic grammar acquisition for natural language applications. A fourth experiment was designed to test CHILL on a more realistic task. A portion of a semantic grammar was "lifted" from an extant prototype natural language database designed to support queries concerning tourist information (Ng, 1988). The portion of the grammar used recognized over 150,000 distinct sentences. A simple case grammar, which produced labellings deemed useful for the database query task, was devised to generate a sample of sentence/case-structure analyses. The example pair shown in Figure 15 illustrates the type of sentences and analyses used.

A learning curve for this experiment is shown in Figure 16. The curve depicts a five trial average of generalization results for 500 sentences which differed from any used in training.

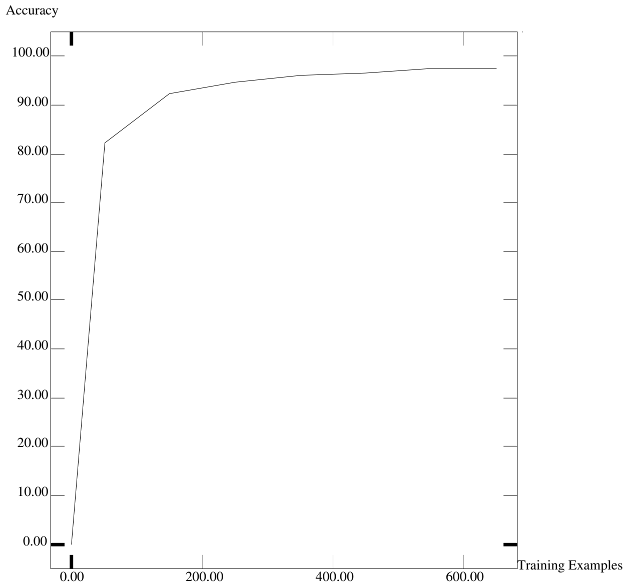


Figure 16: Accuracy on Tourist Queries

The results are very encouraging. With only 50 training examples, the resulting parser achieved 93% accuracy on novel sentences. With 300 training examples, accuracy is 99%. This suggests that the relative lack of ambiguity in this sample task tends to compensate for the larger example space, resulting in excellent generalization to unseen cases.

4.4 Related Work

Most computational research in language acquisition has focused on the learning of syntax rather than search control or semantic parsing (Wirth, 1989; Berwick and Pilato, 1987; VanLehn and Ball, 1987; Berwick, 1985; Wolff, 1982; Liu and Soo, 1992). However, A number of language acquisition systems may be viewed as the learning of search control heuristics.

Langley and Anderson (Langley, 1982; Anderson, 1983) have independently posited acquisition mechanisms based on learning search control in production systems. These systems were cognitively motivated and addressed the task of language generation rather than the case-role analysis task examined here.

Berwick's LPARSIFAL (Berwick, 1985) acquired parsing rules for a type of shift-reduce parser. His system was linguistically motivated and incorporated many constraints specific to the theory of language assumed. In contrast, CHILL relies on implicit negative examples and first-order induction techniques to avoid commitment to any specific model of grammar. Perhaps the simplest example of this difference is the treatment of word classes. Berwick's system requires the pre-defined word classes of input words to be known, whereas CHILL automatically induces whatever word-classes are necessary to support parsing. CHILL also differs in that it produces semantic analyses and can uniformly learn syntactic and semantic constraints for sentence interpretation.

More recently, an exemplar-based acquisition system for the style of case grammar used in CHILL is described in (Simmons and Yu, 1992). Their system depends on an analyst to provide word classifications appropriate to the parsing task and requires detailed interaction to guide the parsing of training examples. CHILL employs constructive induction and requires no interaction during training.

5 Research Plans

The results achieved so far are quite encouraging, but there are many remaining problems which will be addressed in further research. This includes work on the general framework as well as enhancing the inductive mechanisms for both the program optimization and language acquisition applications. In addition, the resulting systems will be tested on more realistic problems to adequately assess their strengths and weaknesses. This section elaborates on each of these aspects.

5.1 Research on the General Framework

The work so far has uncovered some important problems with the general control-rule learning framework outlined in Section 2. This section describes these problems and plans for addressing them.

In some cases, the information needed to determine the appropriate clause to solve a subgoal is not contained in the subgoal literal itself, and is therefore not available for use by a control rule. Consider the following simplified specification of a state-space search:

```
sss(Goal, State, State) :- holds_in(Goal, State).
sss(Goal, State0, State) :-
     op(Op), apply(Op, State0, State1), sss(Goal,State1,State).

op(<operator-1>). op(<operator-2>). ... op(<operator-n>).
```

Directing this search requires learning heuristics for when the various clauses of `op` should be used. Unfortunately, every call to `op` is identical, having a single uninstantiated variable. There is no way of distinguishing among subgoals to which a particular clause should or should not be applied.

In order to solve this problem, additional arguments containing the information required to make intelligent control decisions must be added to the `op` predicate. Also, the rest of the program must be changed to appropriately pass this information down to the subgoal. If the set of examples for a control concept are inconsistent, [3] it is an indication that additional information is needed. By examining the arguments of other predicates used in the proofs of the training examples, the system may attempt to find a small set of them whose values allow it to discriminate between the positive and negative control choices for the subgoal. This process is analogous to determining a small set of arguments for newly constructed predicates and can be efficiently performed using greedy search (see Section 4.2.2 and (Kijsirikul et al., 1992)). In the state-space search example, the current goal and state should provide sufficient

[3] A set of examples is inconsistent if it contains the same example description labeled as both positive and negative.

information to unambiguously choose an operator. Using this approach, the program might be modified as follows:

```
sss(Goal, State0, State) :-
     op(Goal, State, Op), apply(Op, State0, State1), sss(Goal,State1,State).

op(G,S,<operator-1>) :- useful_op_1(G,S).
op(G,S,<operator-2>) :- useful_op_2(G,S).
<etc.>
```

A second problem with the current implementation is that control rules for different clauses of a predicate are learned in isolation. It is often the case the concepts which are useful in making control decisions for a clause are also useful in making decisions for similar clauses.

For example, the concept of `animate` is potentially useful in making a number of decisions during natural language parsing. The current system is forced to re-invent this concept in all the places where it might be useful. This results not only in duplicated effort, but may affect how well the resulting parser generalizes to new inputs. Different control rules for `animate` may "see" slightly different sets of examples. For instance, if the word, "dog" never appears as the agent of a sentence, then the resulting `animate` concept for the agent rule will not include "dog." If a novel sentence uses "dog" as an agent, it may not parse correctly. This is despite the fact that "dog" could have appeared as animate in another control rule such as for the patient of certain verbs like "kill." A single concept for `animate` would include all of the words that were used in either place, thus creating a more general parser. In general, "pooling" examples to learn a shared concept results in more accurate rules. The difficulty is in knowing when and how to do such pooling.

In a FOIL-like inductive learner, it is relatively easy to make a newly constructed concept available for re-use by simply adding it to the list of predicates that can be used to specialize a clause. However, the definition of a constructed concept may be incomplete or incorrect. Allowance must be made for the possibility of extending the definition. This suggests the following algorithm. If, during clause specialization, the predicate found to yield the highest information gain is a previously constructed concept, this provides evidence that the concept is shared. If the current examples are consistent with the previous examples, a new definition may be constructed by merging the two sets and recursively inducing clauses to cover all of the examples. In the previous example, learning the control rule for "patients of kill" would add "dog" to the concept `animate`, thus avoiding the parsing failure when a dog is subsequently seen as the agent of an action.

5.2 Research on Speedup Learning

5.2.1 Algorithm and System Development

Further enhancement of the inductive mechanism in DOLPHIN in a number of ways is planned. First is the incorporation constructive induction to invent new predicates so as to broaden the range of programs which are optimizable. For example, consider the following alternative definition of `permutation`:

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

Here `insert` is being used to select an element to put at the front of the permutation and its tail is computed recursively. Given this definition, the current system cannot convert `naivesort` into an efficient sort. In order to do this, it must learn a control rule for the first clause of `insert` which verifies that the first item is the smallest element remaining in the list (yielding a version of selection sort). This requires the invention of a new recursive predicate for "least element of a list". A method similar to that used in CHILL should be able to invent such predicates. Incorporation of such a method in DOLPHIN and exploration of even more powerful constructive inductive methods suitable for speedup learning is planned.

Another problem is that programs often use a single predicate for several different purposes. The definition of such a predicate may need to be optimized differently for each of its functions. For example, suppose a program used `permutation` to sort some lists into ascending order and others into descending order. The different uses of the predicate will be reflected as conflicting control examples. This problem may be addressed by analyzing the conflicting examples to determine if they arise from distinct calls to the predicate. If so, a duplicate version of the predicate definition should be created for each competing caller and separately optimized for its particular function.

Finally, another avenue of research which might be pursued is the development of more sophisticated techniques for finding control rules which are optimal in the sense of maximizing speedup. Although 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. Incorporating a measure of operationality into the hill-climbing mechanism to explicitly bias the search toward more efficient control heuristics may significantly improve the resulting programs. Such a metric could make use of user-specified information about the efficiency of various predicates, or empirical data on their actual execution times.

5.2.2 Experimental Evaluation

In order to evaluate the efficacy of this framework for speedup learning, experimental testing on larger problems is planned. The experimental methodology will involve producing speedup learning curves like those shown in Section 3 demonstrating performance improvement and comparing our approach to other benchmark methods such as macro learning and traditional EBL control learning.

With respect to program optimization, the approach holds the promise of making Prolog a truly declarative programming language. As the naive-sort example illustrates, our approach can take a strictly logical definition of a problem which is computationally intractable and use training examples to automatically convert it into an efficient program. We believe the ability of the system to perform interesting types of test-incorporation on generate-and-test programs gives it a wide range of applicability in this regard. We plan to test the system on optimizing more realistic declarative program specifications of the type often found in textbooks. These programs are usually presented in a declarative style which emphasizes clarity over efficiency. One possibility is to apply DOLPHIN to natural language applications presented in texts such as (Gazdar and Mellish, 1989; Pereira and Shieber, 1987).

Further testing in more traditional speedup-learning domains such as planning is also envisioned. Most of the existing benchmarks in speedup learning are "toy problems," such as blocks-world planning. However, realistic problem domains are beginning to be explored. The PRODIGY group at Carnegie Mellon University has recently assembled domain theories and problem sets for process planning (Gil, 1991) and logistics transportation (Veloso, 1992). A simple front-end should be able to automatically translate the STRIPS-operator definitions used by PRODIGY into clauses for a means-ends Prolog planner allowing DOLPHIN to be tried on one of these more realistic domains.

5.3 Research on Natural Language Acquisition

5.3.1 Algorithm and System Development

Although the inductive mechanism for language acquisition is fairly well developed, it needs to be completely reimplemented to improve its efficiency. This will require some additional changes to the algorithm, including the addition of pruning heuristics to eliminate investigation of generalizations which cannot produce greater compaction than ones already found and caching previously attempted generalizations to prevent duplicated effort.

The method for generating the initial overly-general parser is also an area for significant development. The current parser produces only simple case structures whose head and slot-fillers are words (or word-senses) from the input sentence. The method could be extended to create deeper semantic representations which actually infer additional information which is not explicit in the sentence. For example, one may want to parse the sentence "Jack went home" into the structure:

```
[ptrans agent:[person name:jack sex:male]
        patient:[person name:jack sex:male]
        destination:[residence resident:[person name:jack sex:male]]].
```

This requires the ability to add slot-filler pairs containing the additional information. If the proper operators are available, the inductive mechanism should be able to determine the contexts in which they are to be applied. The main problem is inferring the necessary set of operators from the training corpus in a tractable fashion.

If the construction of deeper case representations proves successful, a further avenue of research might be to produce even more difficult mappings. In many applications, the desired final output of a natural language system is a logical representation or a database query rather than a case structure. Given an appropriate overly-general parser, CHILL could potentially learn to map sentences directly into such representations; however, the "inductive leap" could be too wide to bridge with a reasonable amount of training data and execution time. An alternative approach which we plan to explore is to bridge this gap in multiple steps by first learning a parser that translates from sentences to case structures and then learning another parser that translates from case structures to database queries or logic.

5.3.2 Experimental Evaluation

Evaluation of this framework for language acquisition will involve experimental tests on various natural language corpora. The experimental method will involve the production of learning curves similar to those in Section 4 demonstrating accuracy in parsing novel sentences.

Initial testing will take place on a wider range of artificial corpora that illustrate interesting natural language issues such as question formation, lexical disambiguation, ellipsis completion, and building deeper semantic representations. This will allow for additional comparisons to connectionist approaches.

An increasing amount of research in natural language involves *corpus-based* methods, which use large repositories of text to build parsing or translation systems. A few corpora of parsed sentences (*tree banks*) have been assembled to support this work (Simmons and Yu, 1992; Black et al., 1992). Some of the data from (Simmons and Yu, 1992) will be used to evaluate CHILL on a larger corpus. It is anticipated that this will present difficulties as the text is drawn from a sample of newswire stories and may not contain sufficient regularity upon which CHILL can construct a parser. A more limited domain such as children's stories, will also be used to evaluate CHILL with a simpler, but nonetheless real, corpus. Such an experiment will involve finding a sample of simple text and hand-crafting a set of analyses for its sentences. One possibility might be the use of several books from a related series where training could be done on the initial books, and testing performed on a previously unseen volume.

It is also hoped that the CHILL approach might be evaluated in an application other than strict parsing. There are numerous possibilities which might be tried incuding generation of database queries, generating text from meanings and translation. In the area of database queries, potential data sets include constructing a corpus from the tourist domain used in previous experiments or, more ambitiously, using an existing corpus such as the ATIS sample which was extracted using "Wizard of Oz" techniques (Samuelsson and Rayner, 1991). An existing system has produced semantic analyses of subsets of this corpus and we could attempt to automatically build such a system from samples of its input and output. Similar experiments could be carried out using samples i/o pairs from existing text-generation or machine translation systems.

6 Conclusions and Significance

The learning of search-control heuristics for logic programs is a very general learning architecture that has important applications to improving the efficiency and competence of knowledge-based systems. Continued development of this architecture and its applications to speedup learning and natural language acquisition is planned. The ultimate goal is the production and systematic testing of efficient, robust systems that may be applied to solve real-world problems. This research integrates and extends previous methods in search-control learning, explanation-based learning, and inductive logic programming under one coherent framework that is capable of both speedup and knowledge-level learning.

In the area of speedup learning, it holds the promise of truly declarative programming in which a logical specification and a set of sample problems can be automatically and dynamically transformed into an efficient procedural program. This method also has the potential of improving the performance of planning and problem solving systems to a greater degree than traditional explanation-based learning methods by learning approximate control rules of high utility. Dolphin incorporates a novel integration of EBL and SBL techniques to effect speedup of Prolog programs. In particular, Dolphin has successfully transformed an intractable specification into a polynomial-time algorithm, and has been shown to outperform competing approaches in several benchmark speedup learning domains. Planned enhancements will further broaden the range of Prolog programs which can be successfully optimized.

In the area of natural language acquisition, these techniques hold the promise of automating the development of complex natural language parsers that map sentences into semantic representations by dynamically integrating syntactic and semantic constraints. It is possible that domain-specific natural language interfaces could be automatically constructed from training sets of pre-parsed sentences. An initial prototype of this approach, Chill, implements a novel combination of first-order induction techniques to automatically acquire semantic grammars. Chill has already demonstrated the capability to learn parsers which generalize well to novel sentences and significantly outperform previous approaches based on connectionist techniques. Planned enhancements will allow its application to even more realistic and difficult corpora.

References

Allen, J. F. (1987). *Natural Language Understanding.* Menlo Park, CA: Benjamin/Cummings.

Anderson, J. R. (1977). Induction of augmented transition networks. *Cognitive Science*, 1:125–157.

Anderson, J. R. (1983). *The Architecture of Cognition.* Cambridge, MA: Harvard University Press.

Berwick, B. (1985). *The Acquisition of Syntactic Knowledge.* Cambridge, MA: MIT Press.

Berwick, R. C. and Pilato, S. (1987). Learning syntax by automata induction. *Machine Learning*, 2(1):9–38.

Black, E., Lafferty, J., and Roukaos, S. (1992). Development and evaluation of a broad-coverage probabilistic grammar of English-language computer manuals. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics*, pages 185–192. Newark, Delaware.

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

Braudaway, W. and Tong, C. (1989). Automatic synthesis of constrained generators. In *Proceedings of the Eleventh International Joint conference on Artificial intelligence.* Detroit, MI.

Brown, J. S. and Burton, R. R. (1975). Multiple representations of knowledge for tutorial reasoning. In Bobrow, D. and Collins, A., editors, *Representation and Understanding.* New York: Academic Press.

Bruynooghe, M., Schreye, D. D., and Krekels, B. (1989). Compiling control. *Journal of Logic Programming*, 6:135–243.

Chase, M., Zweban, 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 intractable 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. F. and Mooney, R. J. (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 the Seventh National Conference on Artificial Intelligence*, pages 564–569. St. Paul, MN.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288.

Fillmore, C. J. (1968). The case for case. In Bach, E. and Harms, R. T., editors, *Universals in Linguistic Theory*. New York: Holt, Reinhart and Winston.

Flann, N. S. and Dietterich, T. G. (1989). A study of explanation-based methods for inductive learning. *Machine Learning*, 4(2):187–226.

Gazdar, G. and Mellish, C. (1989). *Natural Language Processing in Prolog*. New York: Adison-Wesley Publishing Company.

Gil, Y. (1991). A specification of process planning for PRODIGY. Technical Report CMU-CS-91-179, Pittsburgh, PA: School of Computer Science, Carnegie Mellon University.

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.

Kijsirikul, B., Numao, M., and Shimura, M. (1992). Discrimination-based constructive induction of logic programs. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 44–49. San Jose, CA.

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

Langley, P. (1982). Language acquisition through error recovery. *Cognition and Brain Theory*, 5.

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

Liu, R.-L. and Soo, V.-W. (1992). Augmenting and efficiently utilizing domain theory in explanation-based natural language acquisition. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 475–479. Aberdeen, Scotland.

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

Marcus, M. (1980). *A Theory of Syntactic Recognition for Natural Language*. Cambridge, MA: MIT Press.

McClelland, J. L. and Kawamoto, A. H. (1986). Mechnisms of sentence processing: Assigning roles to constituents of sentences. In Rumelhart, D. E. and McClelland, J. L., editors, *Parallel Distributed Processing, Vol. II*, pages 318–362. Cambridge, MA: MIT Press.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., editors, *Machine Learning: An Artificial Intelligence Approach*, pages 83–134. Tioga.

Miikkulainen, R. and Dyer, M. G. (1991). Natural language processing with modular PDP networks and distributed lexicon. *Cognitive Science*, 15:343–399.

Minton, S. (1988). 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.

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

Muggleton, S. and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 339–352. Ann Arbor, MI.

Muggleton, S. and Feng, C. (1992). Efficient induction of logic programs. In Muggleton, S., editor, *Inductive Logic Programming*, pages 281–297. New York: Academic Press.

Muggleton, S. H., editor (1992). *Inductive Logic Programming*. New York, NY: Academic Press.

Ng, H. T. (1988). A computerized prototype natural language tour guide. Technical Report AI88-75, Austin, TX: Artificial Intelligence Laboratory, University of Texas.

Nilsson, N. (1980). *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga.

Pereira, F. and Shieber, S. (1987). *Prolog and Natural Language Analysis*. Stanford, CA: Center for the Study of Language and Information.

Plotkin, G. D. (1970). A note on inductive generalization. In Meltzer, B. and Michie, D., editors, *Machine Intelligence (Vol. 5)*. New York: Elsevier North-Holland.

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

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

Samuelsson, C. and Rayner, M. (1991). Quantitative evaluation of explanation-based learning as an optimization tool for a large-scale natural language system. In *Proceedings of the Twelfth International Joint Conference on Artificial intelligence*, pages 609–615. Sydney, Australia.

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

Shavlik, J. W. and Dietterich, T. G., editors (1990). *Readings in Machine Learning*. San Mateo,CA: Morgan Kaufmann.

Simmons, R. F. and Yu, Y. (1992). The acquisition and use of context dependent grammars for english. *Computational Linguistics*, 18(4):391–418.

St. John, M. F. and McClelland, J. L. (1990). Learning and applying contextual constraints in sentence comprehension. *Artificial Intelligence*, 46:217–257.

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

Tadepalli, P., editor (1992). *Proceedings of the ML92 Workshop on Knowledge Compilation and Speedup Learning*. Aberdeen, Scotland.

Tomita, M. (1986). *Efficient Parsing for Natural Language*. Boston: Kluwer Academic Publishers.

VanLehn, K. and Ball, W. (1987). A version space approach to learning context-free grammars. *Machine Learning*, 2(1):39–74.

Veloso, M. (1992). Learning by analogical reasoning in general problem solving. Technical Report CMU-CS-92-174, Pittsburgh, PA: School of Computer Science, Carnegie Mellon University.

Wirth, R. (1989). Completing logic programs by inverse resolution. In *Proceedings of the European Working Session on Learning*, pages 239–250. Montpelier, France: Pitman.

Wolff, J. G. (1982). Language acquisition, data compression, and generalization. *Language and Communication*, 2:57–89.

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.