Effective Generalization of Relational Descriptions

Larry Watanabe and Larry Rendell

Beckman Institute and Department of Computer Science
University of Illinois at Urbana–Champaign
1304 W. Springfield Avenue, Urbana, Illinois 61801 U.S.A.
watanabe@cs.uiuc.edu rendell@cs.uiuc.edu

Abstract

The problem of computing maximally–specific generalizations (MSCG's) of relational descriptions can be modelled as tree search. We describe several transformations and pruning methods for reducing the complexity of the problem. Based on this analysis, we have implemented a search program (X–search) for finding the MSCG's. Experiments compare the separate and combined effects of pruning methods on search efficiency. With effective pruning methods, full–width search appears feasible for moderately sized relational descriptions.

Introduction

Since Hayes–Roth's (1977) SPROUTER, several systems have been designed that manipulate structural descriptions. Michalski's (1983) INDUCE learns concepts expressed using both attributes and predicates, preferring simpler structural descriptions. VanLehn's (1989) Sierra also learns concepts, efficiently updating the set of minimally general hypotheses. Whereas these systems input examples for supervised learning, Quinlan's (1989) FOIL inputs a set of descriptions for unsupervised learning. Instead of generalizing descriptions, Falkenhainer, Forbus, and Gentner's (1989) SME (structure mapping engine) was designed to find analogies by matching descriptions between a target and base domain.

Although one purpose of this paper is to compare algorithms that manipulate structural descriptions, its main purpose is to explore a new algorithm X–search. First, we detail the basic problem and some assumptions, then develop and explains the X–search algorithm. Next, we describe an empirical analysis of X–search. We review other programs, and compare them to X–search. The last section summarizes our work.

Problem and Assumptions

The problem of matching two structural descriptions is often expressed in terms of first order predicate logic. Given such a representation, certain assumptions and constraints may simplify algorithms.

1. This research was supported in part by grant IRI 8822031 from the National Science Foundation.

Representation and Matching

A relation is a k–ary predicate P^k. Since any k–ary predicate can be represented using a combination of unary and binary predicates, we assume a fixed set R of m unary relations $P^1_1, \ldots, P^1_m$ and n binary relations $P^2_1, \ldots, P^2_n$ (Haussler 1989). The atomic formulae over R are the ***literals*** $P^1_i(x)$ $(i \leq n)$ and $P^2_j(x_1, x_2)$ $(j \leq l)$, where each (subscripted) x is a variable.

To simplify notation we drop the variables in literals and the superscripts of literals indicating unary versus binary predicates. An existential conjunctive expression is a formula $F = \exists\ x_1, \ldots, x_r$: P_1 & P_2 &...& P_n, where $n \geq 1$. These descriptions can be viewed as graphs, in which the nodes are the variables or constants, and the edges are the binary relations. Unary relations may be drawn as reflexive edges.

We can view the problem of structure matching from two perspectives. One is pictorial: a set of relations is a graph, and the problem is to superimpose two candidate graphs so that their nodes and edges agree. The other view is logical: a set of relations is an existential conjunctive expression, and the problem is to unify two such expressions by performing suitable substitutions of constants for variables in the two candidate expressions. The following section develops the logical view of matching.

The Mechanics of Matching

We briefly review the notions from logic needed to create or to match existential conjunctive expressions. Then we define the problem of matching.

Substitution and consistency. A ***substitution*** $\theta = \{c_1/x_1, \ldots, c_n/x_m\}$ is set of correspondences between constants c_i and variables x_i. Under the 1:1 mapping assumption (Hayes–Roth 1977), every constant in a ***consistent*** substitution corresponds to exactly one variable, and every variable corresponds to exactly one constant.

A substitution instance of a literal P is the result of replacing each of the variables in P by the corresponding constants in θ. This is denoted by $P \circ \theta$.

A description F ***matches*** a set of literals L if for some substitution θ, $F \circ \theta \subseteq L$. A description is consistent with a set of positive examples if it matches all of them.

The problem and context of matching. Matching appears in several guises. One problem is to find a maximally specific common generalization MSCG of two or more objects. Another problem is to unify two or more descriptions of object classes. In these and other cases, the matching problem may be reduced to pairwise matching. For example, if we want to find a generalization of positive examples, we can first form an existential conjunctive expression from one example, then iteratively match each of the remaining examples to the composite. This version of the matching problem is concept learning.

In Mitchell's (1978) version space method of concept learning, the set of candidate concepts is represented by two boundary sets G and S. G is the set of all maximally general expressions consistent with the examples, and S is the set of all maximally specific expressions consistent with the examples. When a positive example p is encountered by the system, S is minimally generalized so that each expression in S is consistent with p. Similarly, when a negative example n is encountered, G must be minimally specialized to become consistent with n. Van Lehn (1989) gives an efficient algorithm for updating the G set.

In theory, however, G can grow exponentially even for the propositional case (Haussler 1989). In contrast, a propositional S never contains more than one hypothesis (Bundy, Silver, & Plummer 1985). But for structural domains, Haussler shows that S and G can grow exponentially.

Yet Haussler (1989) notes that many of the techniques developed for learning in structural domains appear to work well in applications of practical interest. This is our motivation in trying to improve current techniques despite the intractability of the problem. In particular, one-sided methods using only the S set might be more efficient than two-sided methods in domains of practical interest.

Our Approach to the Problem

Generalizing specific expressions. In the one-sided version space method, the set S must be modified to make its members consistent with a new positive example. This leads to the following generalization problem:

Given:	s, a member of S, and
	p, a positive example
Find:	a set of existential descriptions C such that:
	every concept in C matches p,
	every concept in C is a generalization of s, and
	no concept in C is a generalization of any other concept in C.

The set C is simply the set of MSCG's of s and p. As Van Lehn (1989) notes, S may have multiple members so the above problem must be solved for each member of S, and the results merged. In the one-sided approach, S is replaced by the merged C at each step.

Hayes-Roth (1977) proposed an ***interference matching algorithm*** for computing C. In the version space framework, interference matching involves the following steps:

```
for every literal $l$ in $s$
  Add $\{l\}$ to $C$
Repeat
  Choose an unchosen literal $l$ from $s$
  For each $c$ in $C$
    if $c + \{l\}$ matches $p$
      Add $c + \{l\}$ to $C$
  Prune $C$ according to heuristics
until all literals from $s$ have been chosen
```

Interference matching uses some sophisticated heuristics to make a good choice of literal for specialization and to prune C. However, pruning may prevent some elements of C from being found.

Decomposition into connected components. Our method uses the fact that graph matching is computationally less complex if only disconnected subgraphs are matched. This is is related to factoring of version spaces (Genesereth & Nilsson 1988). Similar methods are used by Falkenhainer, Forbus and Gentner (1989) for analogical mapping.

A graph represents a conjunctive description $F = \exists\ x_1, \ldots, x_r\colon P_1\ \&\ P_2\ \& \ldots \&\ P_n$. The existential expression is ***connected*** iff for every x_i, x_j, either x_i and x_j occur in the same literal P_i or there is an x_k such that x_i and x_k are connected and x_k and x_j are connected.

An existential conjunctive expression can be normalized by finding its connected components and creating a new existential conjunctive expression for each component. The result is a ***set*** of existential conjunctive expressions. The set of expressions matches another expression E iff every element of the set matches E.

As shown in Figure 1., the S set of a version space can be normalized by replacing each of its elements by the element's connected components, and removing the non-maximally specific components. Although some

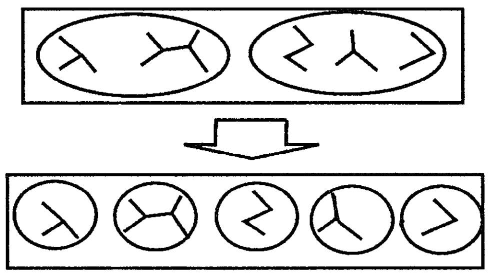
Figure 1. Normalized S set of version space

or all of the new elements of S may be less specific than the old elements of S, together they define the same boundary of the version space as previously.

The X–Search Algorithm

The X-search algorithm computes the set of MSCG's for a given description s and a positive example p. This computation can be viewed as search in a tree T, where nodes correspond to descriptions, and a branch corresponds to a literal that is used to specialize the description. Every description is more general than s, and is constrained to match p. For example, interference matching searches the tree shown in Figure 2.

From this perspective, a branch (or its literal) can be viewed as specializing a node (or its description) of the search tree. Our algorithm uses a connectivity constraint: a literal can specialize a node only if it is connected to its description. The empty description, when at the root node, is defined to be connected to every literal in s.

The search tree can be viewed in yet another perspective. Suppose n is a node in the search tree with description d. If a branch specializes n to a node n' with a literal l, then the subtree rooted at n' searches for the MSCG's that are supersets of $d + \{l\}$. The i-th sibling of n', n_i, searches for the MSCG's that are supersets of $d + \{l_i\}$. Since any two MSCG's must differ by at least one literal, no node in the subtree rooted at n_i needs to be specialized by $l_1, l_2, \ldots l_{i-1}$. This avoids an inefficiency in the interference matching algorithm: an MSCG will be found at many leaf nodes of the search tree, once for every permutation of its literals. We refer to these constraints as ***literal constraints.*** The tree searched by a depth-first strategy with literal constraints is shown in Figure 3.

Previous systems have used some form of connectivity and literal constraints. The next two constraints appear to be new.

The observation that any two MSCG's must differ by at least one literal motivates another pruning

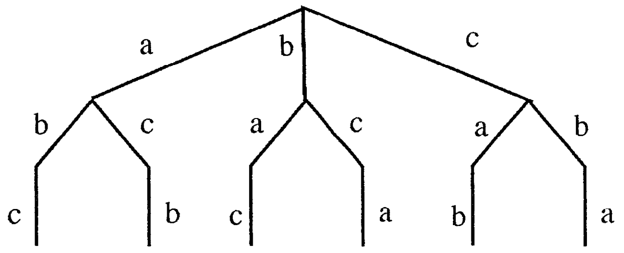

Figure 2. Tree searched by interference matching

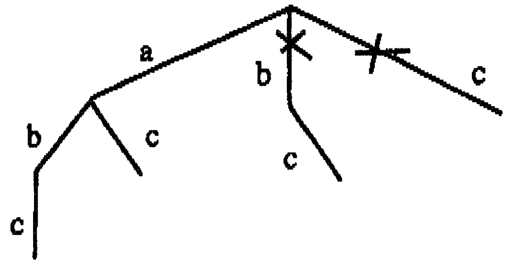

Figure 3. Tree searched with literal constraints.

method. This method, called ***root pruning***, prunes the branches leading from the root. Specifically, when a MSCG is found, we can prune all of the branches from the root that are labelled by a literal in the MSCG. Because this pruning technique can only be applied once during the search, X–search waits until it finds a reasonably "large" MSCG before applying root pruning. The branches that are pruned by root pruning are marked by "X" in Figure 3. We will discuss the final pruning method, ***substitution pruning,*** after presenting the main procedures of the X–search algorithm.

Figure 4 shows the X–search algorithm. X–search is given s, a connected structural description, and p, a positive example, and returns a superset of the MSCG's of s and p. Each literal in s is matched against p to initialize its substitutions list, used later in CreateNode (Fig. 5).

The search stack is initialized to contain the root node corresponding to the empty description. A node has a marked list to keep track of which literals have been tried at the node, and a literal–constraint list to keep track of which literals are not allowed to

```
X-search($s$, $p$)
  for each literal $l \in s$
    Init-Substitutions($l$);
  stack := NULL; root = {};
  Push(root, stack);
  while stack $\neq$ NULL;
    top = Pop(stack)
    while $\exists$ $l \in s$ s.t. Expandable(top, $l$)
      Mark(top, $l$);
      next := CreateNode(top, $l$)
      if next $\neq$ FAIL
        Push(top, stack);
        top := next
    if top is a leaf then
        Push(top, Saved)
        if top is "large" apply root pruning;
  remove subsumed elements from Saved;
  return(Saved)
```

Figure 4. Basic MSCG algorithm X-search.

specialize a node because of the literal constraint. A literal l is added to the marked list of a node n by a call to Mark(n, l), and to the literal–constraint list by Add–Literal–Constraint(n, l). This information is used to determine which literals can be used to specialize a literal. The root node can also have literals marked by by root pruning.

The function Expandable(d, l) returns true iff l can be used to specialize d. For the root node, this is true iff l is not on the marked or literal–constraint lists of d. For all other nodes, Expandable(d, l) returns true iff l is not on the marked or literal–constraint lists of d, and l is connected to d. The main procedure of X–search conducts a depth–first search for a superset of the MSCG's, which is stored on *Saved*. Afterwards, the non–maximally specific elements of *Saved* are removed.

CreateNode forms a new node, if possible, from a node d, and a literal l. CreateNode's main function is to find the matches from d and l to p, and to store these on the substitutions list of d and l. The matches are generated by multiplying the substitutions of d by the substitutions of l, and saving the consistent ones. If there are no consistent ones, CreateNode returns FAIL. If there are some consistent ones, CreateNode creates a literal constraint for that literal and the node d, and returns the specialization $d + \{l\}$. The literal constraints of the parent node d are inherited here by the new node $d + \{l\}$.

CreateNode also calls Substitution–Prune (Fig. 6), which implements the final pruning method. Substitution pruning attempts to prune the generation of substitutions at a new node. This is important for efficiency because the number of substitutions at a node can grow exponentially in the depth of the tree. Substitution pruning has a second advantage: nodes without any substitutions are pruned. Thus, pruning substitutions can indirectly lead to pruning of nodes.

Substitution pruning is based on an analysis of the necessity for backtracking. First, backtracking might be necessary when a node with description d matches several different parts of p. For each match, it may be possible to specialize d with one of the literals.

```
CreateNode($d$, $l$)
   for $\theta \in$ Substitutions($d$)
      if $\neg$Prune($d, \theta, l$) or $d$ = root
         for $sub' \in$ Substitutions($l$)
            if Consistent($\theta$, $sub'$)
                Add–Substitution($d+\{l\}$, $\theta + sub'$)
   if Substitutions($d + \{l\}$) = NULL then return(FAIL)
   else
      for $l' \in s$
         if Literal–Constraint($d$, $l'$) = TRUE then
                Add–Literal–Constraint($d+\{l\}$, $l'$)
   return($d + \{l\}$)
```

Figure 5. Procedure CreateNode

However, it may not be possible to specialize d with any pair of these literals, because the matching literals may be in different parts of p.

Even if d has only one match to p, backtracking may be needed because of the 1:1 mapping assumption. Suppose description d is matched to p with substitution θ. Specializing d with a literal l may require assigning a constant c from a literal in p to a variable x from l. Specializing d with another literal l' may require assigning the same constant c to a different variable x' from l'. Because of the 1:1 mapping assumption, we cannot assign c to both x and x', so d cannot be specialized with both l and l'.

If the graph is acyclic, then two literals are incompatible only if they have the same predicate name, and their corresponding variables are bound to the same constants by θ. This defines an equivalence class of literals, any of which may map to the same subset of literals in p. If there are fewer literals in the equivalence class than corresponding literals in p, then the substitution can be pruned.

Figure 6 shows the implementation of substitution pruning. The equivalence class is initialized to the first literal that matches p using the substitution θ (plus some other assignments of constants to variables). The next time Prune is called with d and θ, the substitution is pruned if l is not in the equivalence class.

Empirical Results

Three experiments were run. The first experiment measured the cpu times, real times, and substitution counts for X–search as a function of the size of the examples. The second experiment compared compared the effects of X–search's pruning methods by running X–search without each of the pruning methods in turn. The third experiment compared X–search against a random beam search.

The data were generated from the structural examples used in Hoff, Stepp, and Michalski (1983). Each

```
Prune–Substitutions($d$, $\theta$, $l$)
   if $d + \{l\}$ matches $p$ using $\theta$ then
      if EquivClassLiteral($d$, $\theta$) is undefined then
         set EquivClassLiteral($d$,$\theta$) to $l$
         return FALSE
      else if $l$ is not in the same equivalence class as
         EquivClassLiteral($d$, $\theta$) under $\theta$
         return TRUE
      else if Competitors($d$, $\theta$, $l$) <
         Resources($s$, $\theta$, $l$) then
         return TRUE
      else
         return FALSE
   else return TRUE
```

Figure 6. Procedure Prune–Substitutions

data set consisted of a pair of positive examples. The examples of the i-st pair were created by merging i examples of i different classes from Hoff et al.'s data. Additional literals were added to the examples to connect their descriptions.

The real times, cpu times, and substitution counts for X-search are shown in Table 1. The example size is the average number of literals in an example.

The substitution counts for X-search in several configurations is given in Table 2. The configurations correspond to the single pruning constraint that was not used in the test run. These are: LC = literal constraints, RP = root pruning, SP = substitution pruning, and CC = connectivity constraints. Where no table entry is given, the job was killed after taking excessive time.

For the third experiment, X-search and the beam search were compared along two dimensions. The first dimension measured how many nodes beam search expanded to find the full MSCG set. This was determined by rerunning beam search with incrementally increasing beam width until the full MSCG set was found. Only the number of nodes expanded during the last run was counted. The second test measured how many descriptions were found by beam search when it expanded the same number of nodes as needed by X-search. The test data consisted of about 33 pairs of examples. For the first test, beam search expanded about four times as many nodes as X-search. For the second test, beam search found 70% of the descriptions found by X-search.

Table 1. CPU time and substitution counts.

Example Size	Real Time (secs)	CPU Time (secs)	Number of Subs.
11	0.2	0.0	235
20	0.7	0.0	581
31	2.1	0.0	1313
45	22.8	0.0	6355
63	135.6	0.1	25616
86	226.2	0.4	29089

Table 2. Effect of removing pruning methods

Example Size	LC	RP	SP	CC
11	547	235	425	925
20	1672	581	1184	108802
31	3165	1313	2439	–
45	13234	16087	349176	–
63	56842	69286	–	–
86	67070	73639	–	–

Discussion

X-search was able to handle moderately sized descriptions, up to 86 binary predicates per example, in .4 seconds of CPU time and of 226.2 seconds of real time, on a Sun4 workstation. These results indicate that full-width search may be feasible for moderate-sized examples. Previously, full-width search methods have been avoided because they are expensive even for small problems. An alternative to full-width search is a beam search, typically based on information-theoretic criteria or an evaluation function.

An information-theoretic or evaluation function based approach considers the number of positive and negative examples when evaluating partial descriptions. For example, one might prune a node if it corresponded to a description that covered many negative examples and few positive ones. This is an important source of information that methods such as version spaces ignore. Instead, methods such as version spaces are guided by their inductive bias. The advantage of the version space approach is that when there are few examples, and the examples are carefully chosen, a great deal of information about the concept can be extracted. Thus, computing MSCG's is important for problems such as analogy, or learning from a helpful teacher. In contrast, an information-theoretic approach will probably perform better when doing unsupervised learning in a complex and noisy domain with many examples.

Although the pruning methods used by X-search seem to give significant increases in efficiency, the problem is still intractable in the worst case. However, efficient pruning methods can delay the point at which full-width search becomes infeasible. X-search must also be viewed from the perspective of its intended use. Typically, the learned descriptions are used as classification rules for an expert system. The expert system is limited by the same matching combinatorics, as X-search, and cannot efficiently use extremely large and complex rules.

The results show that removing even one of the methods leads to a substantial increase in the number of substitutions. The most important constraint is connectivity, followed by substitution pruning, literal constraints, and root pruning.

Although substitution pruning is an effective constraint, it requires that the descriptions be acyclic. This is a strong requirement, so we will be generalizing the method to handle cyclic descriptions in future research.

The use of a random beam search is a weak basis for comparison. However, it was quite difficult to come up with a good evaluation function for beam search for this problem. When given only two positive examples, an information-theoretic evaluation function is of little value. Evaluation functions that score descriptions according to some desirability metric have the effect of concentrating the beam in the same region of the search space. Thus, the same MSCG was rederived by many elements of the beam. Several of our initial attempts at evaluation functions produced worse results than the random strategy, and others were insignificantly better. Although the results of this experiment were inconclusive, our difficulties have led us to conjecture that a good evaluation function for beam search must evaluate the beam as a whole, rather than any particular element of it.

The empirical evaluation of X–search gives some indication of how well it performs. However, these indicators are based only on a few examples with similar structure, namely pseudo–chemical molecules. We will be conducting more extensive empirical evaluation and comparisons in future research.

Other Approaches

Earlier algorithms share some similarities with each other and with our program. Our approach also differs in some ways.

SPROUTER

Hayes–Roth's (1977) interference matching algorithm SPROUTER is one of the earliest and most widely imitated method for learning structural descriptions. Hayes–Roth makes the 1:1 mapping assumption in early versions of his system, although he notes the inadequacy of this assumption for learning many classes of interesting concepts. The interference matching algorithm matches literals in a member s of the specific set S against literals in p, checking the consistency of the bindings, and extending the generalized description if possible. Only the best w descriptions are kept at each step. Hayes–Roth's (1977) paper also discusses many issues in structural learning that we have not addressed here.

INDUCE

Michalski's (1983) INDUCE is one of the earliest structural learning algorithms. INDUCE is similar to interference matching in using a beam search through the space of possible abstractions, but differs in searching for a maximally general description that is consistent with the negative examples, rather than a maximally specific description that is consistent with the postive examples. INDUCE also uses a two–space search method: first a description is found in structure only space, and then attribute–based learning methods are used to specialize the structural description. INDUCE uses an evaluation function that evaluates the completeness and consistency of the descriptions with respect to the positive and negative example. The evaluation function gives INDUCE an information–theoretic flavor, in contrast to version spaces which relies more heavily on the characteristics of a particular example. INDUCE's two space search method, in which attributes are ignored when first finding a structural description, is similar to part of Falkenhainer, Forbus and Gentner's Structure Mapping Engine (1989), which ignores attributes when finding an initial set of analogical mappings. Michalski's representation is more expressive than ours, allowing attributes to be combined with operators such as $<, \leq, =, \geq$, and $>$, to permit expressions such as [distance(x,y) $<$ 33]. INDUCE can also learn disjunctive descriptions, unlike version–space based methods.

SUBDUE

Holder's (1989) SUBDUE uses clustering methods to construct features from the examples. The examples are simplified by replacing parts of their initial descriptions by constructed features. These modified examples are given to INDUCE which performs the actual learning. Holder's two–step approach to learning structural descriptions is more efficient than a single–step approach.

The Structure Mapping Engine

Falkenhainer, Forbus and Gentner's (1989) SME addresses the problem of structural matching in the context of analogical reasoning. Analogical reasoning has some rather different properties from learning relations, as predicates can be matched against other predicates. Falkenhainer et al.'s approach is interesting in its use of extensive knowledge to guide the matching process. The initial construction of matchings, is similar to VanLehn's (1989) method in that it enumerates the possible correspondences of objects in the target and base domain. SME is efficient when knowledge about the base and target domains is available to guide the mapping process.

Sierra

VanLehn (1989) gives an efficient method for updating the G set when a negative example is encountered that matches G. VanLehn enumerates the substitutions, and uses an efficient bit–representation for the set of substitutions. He reduces the problem of updating G to a series of cover problems, and uses Well's algorithm (1971, sec. 6.4.3) for finding irredundant covers to solve the problem.

The length of the bit representations is $c!/(c-N)!$, where N is the number of variables in s and c is the number of constants in n, the negative example. Although this number is very large, the efficiency of the bit operations is sufficient to produce some impressive results. In addition, VanLehn enforces the equivalent of type constraints on substitutions.

FOIL

Quinlan's (1989) FOIL uses a full first-order predicate representation to learn relations. An interesting feature of FOIL is its ability to learn relations in either supervised or unsupervised mode. FOIL uses an information-theoretic evaluation function to perform an heuristic search for a single description. Unlike most other structural learning systems, FOIL is capable of learning disjunctive, recursive descriptions, with embedded skolem functions.

Commonalities

Many programs, such as SME, X-search, Sierra, and SPROUTER are incremental, conjunctive learners that rely on the conjunctive bias, their current hypothesis, and the current example to form a hypothesis. Other programs, such as INDUCE and FOIL are non-incremental, disjunctive learners that use information-theoretic methods or an evaluation function to form a hypothesis.

Haussler (1989) shows a number of interesting results about the difficulty of learning existential conjunctive descriptions. He proves that this problem is NP-complete, even when no binary relations are defined, attributes are Boolean valued, and each example contains exactly two objects. However, he notes that heuristic methods for learning existential concepts can be effective, if not always efficient.

Haussler (1989) also presents a subset query method for pac-learning learning existential conjunctive descriptions that have a fixed number of variables. The main technique he uses is matching positive examples with each other and with intermediate hypotheses to form MSCG's. This is the basis of many learning algorithms for existential conjunctive concepts (Dietterich 1983).

Conclusions

This paper describes a new search method, X-search, for computing the maximally specific common generalizations (MSCG's) of a description and a positive example. Finding MSCG's is important for learning relational descriptions and plays a role in many structural learning programs. This is the main problem in updating the specific (S) set in a version space algorithm for structured descriptions. The main weakness of X-search is that under the 1:1 mapping assumption, it is restricted to descriptions corresponding to acyclic graphs. Our preliminary results indicate that X-search is a fast and effective method for computing the MSCG (S) set.

Acknowledgements

We want to thank Ryszard Michalski for supporting the first author during part of this research, and Robert Stepp, Larry Holder, Diane Cook, and Brad Whitehall for helpful discussions about structural learning.

References

Bundy, Alan, A. Silver, and D. Plummer, An Analytical Comparison of Some Rule-Learning Programs, *Artificial Intelligence*, vol 27, pp 137-181, 1985.

Dietterich, Thomas G. and Ryszard S. Michalski, A Comparative Review of Selected Methods for Learning from Examples, in *Machine Learning: An Artificial Intelligence Approach* ed. R. S. Michalski et al, pp. 41-81, Tioga, 1983.

Falkenhainer, Brian, Kenneth D. Forbus, and Dedre Gentner, The Structure-Mapping Engine: Algorithm and Examples,*Artificial Intelligence*, vol. 41, no. 1, pp. 1-63, 1989.

Genesereth, Michael R. and Nils J. Nilsson, *Logical Foundations of Artificial Intelligence*, Morgan Kaufman, 1988.

Haussler, David, Learning Conjunctive Concepts in Structural Domains, *Machine Learning*, vol. 4, no. 1, pp. 7-40, 1989.

Hayes-Roth, Frederick and John McDermott, Knowledge Acquisition from Structural Descriptions, *Proc. fifth International Joint Conference on Artificial Intelligence*, pp. 356-362, Morgan Kaufman Publishers, Inc., Cambridge, Massachussetts, August, 1977.

Hoff, William A., Ryszard S. Michalski, and Robert E. Stepp, *INDUCE 3: A Program for Learning Structural Descriptions from Examples*, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, Illinois.

Holder, Lawrence B., Empirical Substructure Discovery, *Proceedings of the Sixth International Workshop on Machine Learning*, pp. 133-136, Morgan Kaufman Publishers, Inc., Ithaca, New York, June, 1989.

Michalski, R. S., A Theory and Methodology of Inductive Learning, in *Artificial Intelligence*, vol. 20, 2, pp. 111-161, 1983.

Mitchell, Thomas M., *Version Spaces: An Approach to Concept Learning*, Stanford University, 1978. Ph.D. Thesis

Quinlan, J. R., Learning Relations: Comparison of a Symbolic and a Connectionist Approach, *University of Sydney Technical Report*, no. TR-346, Basser Department of Computer Science, University of Sydney, Sydney, Australia, May, 1989.

VanLehn, Kurt, Efficient Specialization of Relational Concepts, *Machine Learning*, vol. 4, no. 1, pp. 99-106, 1989.