Small is Beautiful: A Brute-Force Approach to Learning First-Order Formulas

Steven Minton and **Ian Underwood**
Recom Technologies
NASA Ames Research Center
Mail Stop 269-2
Moffett Field, CA 94035
{minton,ian}@ptolemy.arc.nasa.gov

Abstract

We describe a method for learning formulas in first-order logic using a brute-force, smallest-first search. The method is exceedingly simple. It generates all irreducible well-formed formulas up to a fixed size and tests them against a set of examples. Although the method has some obvious limitations due to its computational complexity, it performs surprisingly well on some tasks. This paper describes experiments with two applications of the method in the MULTI-TAC system, a program synthesizer for constraint satisfaction problems. In the first application, axioms are learned, and in the second application, search control rules are learned. We describe these experiments, and consider why searching the space of small formulas makes sense in our applications.

Introduction

Most machine learning systems prefer smaller, simpler hypotheses to larger, more complex ones. This bias is a form of Occam's Razor. While Occam's razor has obvious aesthetic appeal, some researchers have attempted to justify Occam's razor on more principled grounds by showing that it produces more accurate hypotheses; for instance, Blumer et al. (1987) show formally that one version of Occam's razor produces hypothesis that are likely to be predictive of future observations.

However, in some applications, the *utility* of the learned information depends on more than just prediction accuracy. Utility considerations often provide us with an additional reason for applying Occam's Razor, a point which has received scant attention. In our application, which involves automated problem solving, utility considerations place strong requirements on the learning process. In particular, the learned theories must consist of small sets of simple first-order formulas. Complex formulas, or large numbers of formulas, can significantly degrade system performance.

Yet another reason for preferring simpler hypotheses is that they can be relatively straightforward to find, particularly if we equate "simplest" with "smallest". In our system, the learning component systematically generates well-formed formulas, in order of size, beginning with the smallest. It tests each formula against a set of training examples, attempting to find formulas that are adequate for the needs of the performance component. This *brute force, smallest-first* (BFSF) search often produces theories that are more efficient and more comprehensive than those entered by hand.

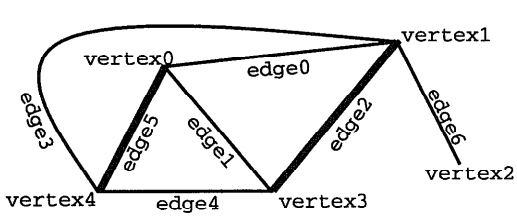
In this paper we describe two applications of BFSF inductive learning in the MULTI-TAC system. In one application, the system learns axioms that are then used by a theorem prover to reason about generic constraint satisfaction problems (CSPs), such as graph coloring and bin packing. In the second application, the system learns search control rules to guide a constraint satisfaction engine. We were initially surprised that a brute force approach worked so well for our applications. In retrospect, we can identify several reasons why searching the space of small formulas is appropriate in our domains.

An Overview of the Multi-TAC system

MULTI-TAC (Multi-Tactic Analytic Compiler) is designed for a scenario where a combinatorial search problem must be solved routinely, as in a scheduling application where each week manufacturing tasks are assigned to workers. The system takes as input a specification of the generic problem and a set of problem instances. The objective is to synthesize an efficient program for the instance population. In practice, our goal is to do as well as competent programmers, as opposed to algorithms experts. Attaining this level of performance on a wide variety of problems would very useful; many relatively simple applications are not automated because programmers are unavailable.

The system starts with a set of domain-independent heuristics. When it encounters a new domain, it creates problem-specific approximations of these heuristics, and then searches for the combination that performs best on the instance population. It returns as output a Lisp program that incorporates this combination of heuristics.

In order to present a problem to MULTI-TAC it must be formalized as an integer CSP, that is, as a set of constraints over a set of integer variables. A solution




```
(declare-parameter 'K 2)
(declare-type-size 'edge 7)
(declare-type-size 'vertex 5)
(declare-relation-data
  '((endpoint edge0 vertex0)
    (endpoint edge0 vertex1)
    (endpoint edge1 vertex0)
    (endpoint edge1 vertex3)...))
```

Figure 1: An instance of MMM with $K = 2$. A solution $E' = \{\text{edge2 edge5}\}$ is indicated in boldface. The instance specification is on the right.

exists when all the variables are assigned a value such the constraints on each variable are satisfied.

For example, consider the NP-complete problem, "Minimum Maximal Matching" (MMM), described in (Garey & Johnson 1979). An instance of MMM consists of a graph $G = (V, E)$ and an integer $K \leq \mid E \mid$. The problem is to determine whether there is a subset $E' \subseteq E$ with $\mid E' \mid \leq K$ such that no two edges in E' share a common endpoint and every edge in $E - E'$ shares a common endpoint with some edge in E'. See Figure 1 for an example.

To formulate MMM as a CSP, we represent each edge in the graph with a variable. If an edge is chosen to be in E', it is assigned the value 1, otherwise it is assigned the value 0. The constraints can be stated as follows:

1. If $edge_i$ is assigned 1, then for every $edge_j$ that shares a common endpoint with $edge_i$, $edge_j$ must be assigned 0.
2. If $edge_i$ is assigned 0, then there must exist an $edge_j$ such that $edge_i$ and $edge_j$ share a common endpoint, and $edge_j$ is assigned 1.
3. The cardinality of the set of edges assigned 1 must be less than or equal to K.

A *problem specification* describes the types (*e.g.*, **vertex** and **edge**) and relations (*e.g.*, **endpoint**) and specifies the constraints in a typed predicate logic. An *instance specification* (Figure 1) instantiates the types and relations referred to in the problem specification. Our constraint language is relatively expressive, as it allows for full first-order quantification and the formation of sets and bags. Below we show how the first constraint above is specified, for some edge $Edge_i$:

```
(or (not (assigned Edge_i 1))
    (∀ Vrtx : (endpoint Edge_i Vrtx)
      (∀ Edge_j : (endpoint Edge_j Vrtx)
        (or (equal Edge_j Edge_i)
            (assigned Edge_j 0))))
```

The notation (∀x: (endpoint x y)...) should be read as "forall x such that (endpoint x y)...".

The constraint language includes two types of relations, problem-specific *user-defined relations* such as **endpoint**, and built-in *system-defined relations*, such as **assigned**, **equal** and **less-than**. (There are no functions; instead, we use two-place relations.) The **assigned** relation has special significance since it represents the state during the search process. In MMM, for example, the search proceeds by assigning each edge a value. In a solution state, every edge must be assigned a value such that the constraints are satisfied.

Enumerating Formulas

We now describe how the MULTI-TAC systematically generates hypotheses of increasing size. A formula is defined to be of size s if it contains s atomic formulas. The system first generates formulas of size 1, then formulas up to size 2, and so on, until it exceeds a pre-determined bound on either the computation time or the number of formulas. The generation process is based on a recursive grammar[1] for the language:

wff = (∀ *var* : *atomic wff*) | (∃ *var* : *atomic wff*) |
(**and** *wff* ... *wff*) | (**or** *wff* ... *wff*) |
(**not** *atomic*) | *atomic*
atomic = (*predicate term* ... *term*)
term = *var* | *constant*

Formulas of size $s > 1$ are generated by existentially or universally quantifying formulas of size $s-1$, or conjoining or disjoining sets of formulas whose size sums to s. In MULTI-TAC a recursive procedure, GENERATE-FORMULAS accomplishes this. GENERATE-FORMULAS takes a size s and a set "free variables", and generates all formulas of size s defined over one or more of the free variables.

Atomic formulas are generated using the user-defined types and relations (from the problem specification) and the system-defined relations. Each argument either is a variable or is a constant mentioned in the problem specification (i.e., arbitrary integers are *not* used). The MMM specification, for example, mentions the constants 1 and 0, and the relation **endpoint**.

Our implementation uses several simple techniques to improve the efficiency of the generation process. The most significant of these is that only "irreducible" formulas are generated. MULTI-TAC uses a simplifier to check each (sub)formula returned by GENERATE-FORMULAS; any formula which can be reduced to a smaller equivalent formula is discarded since the smaller formula will have been generated as well.

[1]The grammar shown here is simplified, since it does not include set/bag generators.

After the candidate formulas have been generated, MULTI-TAC employs training examples to identify which formulas are useful. In the following sections we describe two applications, and for each we describe how the useful formulas are identified.

Inducing axioms

MULTI-TAC includes a resolution theorem-prover that can be used during program synthesis for several different reasoning tasks. For example, the prover can be used to verify that a given value will necessarily violate a constraint, or that the antecedent of a search control rule will be satisfied in a given situation.

In order to use the theorem prover, we require axioms that describe the problem domain. Some axioms can be derived from the problem specification. For example, the problem description explicitly specifies the argument types for each relation. So, for MMM, an axiom stating that (**endpoint** x y) implies (**edge** x) can be created directly from the problem specification. However, other information may be left implicit in the instances. For example, in MMM (or any graph) it is necessarily true that for every edge there is at least one vertex that is its endpoint, but this is not stated in the problem specification.

We need these additional axioms for the theorem prover to operate effectively. In addition, for the theorem prover to operate efficiently, it helps significantly if there are only a few small axioms. Having too many axioms, or very complex axioms, may greatly impair a resolution theorem prover's performance because the branching factor will be higher.

These axioms have to come from somewhere. One possibility is to require the user to enter them along with the problem specification. However, experience shows that axioms entered by users tend to be incomplete, unnecessarily complicated, and just plain incorrect. One contributing factor is that people are generally not facile with predicate calculus. (In fact, the authors have noticed that we ourselves make many mistakes.) But even if users are asked to enter the axioms in English they tend to neglect relevant information. For example, for the MMM problem, one might forget to mention that every edge has an endpoint since it is so obvious. (It may also seem obvious that every edge actually has two unique endpoints, but in fact this is not true, since some edges may be connected to the same vertex at both ends. This illustrates the difficulty of writing axioms.)

Alternatively, we have found that a suitable set of axioms can often be found using BFSF induction. Our training examples are problem instances randomly selected from the instance distribution. (The instance distribution only provides positive examples, so we do not use negative examples.) An iterative approach is used to produce axioms. On the nth iteration, the system generates formulas of size n and tests them against the training examples, retaining only those formulas that are consistent with all of the examples. Then the system filters this set further, eliminating formulas that can be proved using smaller formulas as axioms (i.e., those found on the previous iterations). The remaining formulas are reduced to an independent set by trying to prove each formula, using the other formulas as axioms. This set is minimal, in the sense that each axiom in the set is not provable in terms of the others. (This is time-consuming, but necessary, since redundant formulas degrade the theorem prover's performance.) The system then proceeds with the next iteration, until a resource bound is exceeded.

Of course, there is always a chance that the system may induce incorrect axioms. We can either ask the user for assistance in eliminating incorrect axioms or accept the entire set and take the chance that our proofs will be incorrect in some cases.

Here we consider only the former approach. After each iteration, the user is asked to approve the proposed axioms. If the user chooses not to accept an axiom, he can provide an example that is inconsistent with the proposed axiom. The example is then added to the training set. For instance, the system may propose that "there are at least two edges in every graph". The user can then enter a graph containing a single edge as a counterexample. Often a single counterexample will rule out a whole class of potential axioms; we have found that a few counterexamples often suffice to rule out all incorrect axioms.

Table 1 summarizes a set of experiments with several combinatorial problems described in (Garey & Johnson 1979): BIN PACKING, DOMINATING SET, GRAPH 3-COLORABILITY, NOT ALL EQUAL 3-SAT, PARTITION-INTO-TRIANGLES and MMM. To make the experiment more challenging, the task was to produce axioms for distributions consisting only of solvable instances. Axioms characterizing only solvable instances can be used to quickly screen out unsolvable instances, and in addition are useful for many other tasks.[2] For purposes of comparison, we asked two human volunteers to do the same task. (Both were familiar with predicate calculus, and one was a MULTI-TAC project member who had experience with the theorem prover.)

For each problem, all formulas up to size 4 were generated. To test the formulas, fifteen training examples were generated. As explained above, the user (one of the authors) could enter additional examples by hand, and in our experiments between 2 and 4 additional examples were entered per problem in response to incorrect axioms. Columns 2,3 and 4 show the number of formulas of each size that were generated. The last column shows the number of axioms finally retained.

[2]These axioms are essentially a superset of the axioms describing arbitrary instances (solvable or not). For example, for **Graph 3-Colorability**, the axiom set would include the axioms describing graphs in general, but it might also include the axiom "No vertex has an edge to itself", since this is true of solvable instances.

	$s=2$	$s=3$	$s=4$	Final
Bin Packing	6	42	450	4
Dominating Set	8	70	1242	4
Graph 3-Color	8	48	472	5
Not-All-Eq 3-Sat	6	40	474	35
Part. Into Triangles	8	42	472	6
Min Max Matching	6	32	250	5

Table 1: Axiom Learning, Experimental Results

One striking aspect of these experiments was how few training instances were required. In fact, the set of formulas generally stabilized after the third example – relatively few formulas were eliminated by subsequent examples (except for the user-provided counterexamples). This is somewhat surprising, since a PAC analysis reveals that for 1000 hypotheses, approximately 200 examples are required just to be .95 confident that the error is less than .05. Of course, this is a worst-case analysis, and it appears that the worst case assumptions are violated in at least two ways. First, many of the formulas are equivalent (or almost equivalent), since there are many ways to state the same fact. Second, most of the formulas appear to be either true, false, almost always true or almost always false. Thus, after just a few examples, the formulas that are left are true or almost always true. We conjecture that this will occur for many problems. An analogous situation has been identified in the CSP literature – for some well-known problems, almost all instances are easy to solve because they are either over-constrained or under-constrained (Cheeseman, Kanefsky, & Taylor 1991; Mitchell, Selman, & Levesque 1992).

If we compare the learned axioms to those produced by our human subjects, the results are as expected. Often the humans neglected to state relevant axioms or stated them incorrectly. The machine-generated axiom sets were more complete, and in some cases, the axioms were stated more concisely. (In fact, the machine did a better job than the authors for MMM.) On the other hand, for one problem, NOT ALL EQUAL 3SAT, the system generated many redundant axioms because it could not prove they were redundant within the time limit. Finally, the humans also identified a couple of axioms of size 5 and 6 that the system obviously did not generate. (However, these axioms appeared to be useless for the system's purposes.)

As we have argued, one advantage of the BFSF method is that it produces small sets of small axioms. To illustrate the benefits, we tried reversing the smallest-first bias, producing MMM axioms of size four before looking for smaller axioms. This did indeed result in a poorer set of axioms, including:

$(\forall\ E_1 : (\text{edge}\ E_1)$
$\quad (\exists\ E_2 : (\text{edge}\ E_2)$
$\quad\quad (\exists\ V : (\text{endpoint}\ E_1\ V)$
$\quad\quad\quad (\text{endpoint}\ E_2\ V)$

This states that "for every edge E_1, there is an edge E_2 that shares an endpoint with E_1. Although this appears to be false, it is in fact true, since E_1 and E_2 can refer to the same edge. Thus, it is actually an obscure way to state the axiom, "every edge has an endpoint". The problem with this, aside from obscurity, is that it is much less efficient for resolution theorem-proving; larger axioms translate into more (and larger) clauses, and thus increase the prover's branching factor.

Inducing Search Control Rules

In MULTI-TAC, search control rules are used to control the choices made during the constraint satisfaction process, such as variable and value-ordering choices. For example, we can implement the generic variable-ordering heuristic "prefer the most-constrained variable" by using a rule which, at each choice point, selects the variable with the fewest possible remaining values. Unfortunately, using this generic rule can be costly since it requires that the system maintain the possible values for each variable during the search.

Minton (1993a) has described an analytic approach for automatically generating control rules. This method operationalizes generic variable and value-ordering heuristics by incorporating information from the problem specification. For MMM this process produced 52 candidate control rules, including the those shown below. (Recall that in MMM the CSP search proceeds by assigning each edge either 0 or 1.)

- Prefer an edge with the most neighbors (i.e., adjacent edges).
- Prefer an edge with the most neighbors that have been assigned values.
- Prefer an edge that has a neighbor that has been assigned the value 1.

These rules were produced by operationalizing and approximating the generic "most constrained variable first" heuristic. The rules vary in their application cost and their effectiveness in reducing search. For example, the first rule is relatively inexpensive, since the ordering can be precomputed and the edges sorted accordingly before the CSP search process begins. (The compiler has been specially crafted to make efficient use of such rules.) The third rule is more powerful but more costly to apply since the ordering cannot be precomputed. MULTI-TAC allows such rules to be used in combination. The system's utility evaluation module carries out a beam search for the best combination of rules; different combinations are evaluated by running them on representative problem instances.

Although the analytic learning method has performed well, producing control rules that are comparable or better than hand-coded ones (Minton 1993b; 1993a), there is a significant drawback. For each generic heuristic, the system's designers must write a meta-level theory that can be operationalized to produce control rules. This involves a sophisticated "theory-engineering" process where the designers guess

	Analytic Learning		Inductive Learning			
	Solved	Time	Solved	Time	Rules Gen	Rules Kept
Bin Packing	46	.58	66	.48	539	17
Dominating Set	19	.93	45	.73	2931	70
Graph 3-Colorability	96	.33	100	.25	805	30
Not-All-Equal 3-Sat	84	.66	81	.75	1084	68
Partition Into Triangles	100	.73	100	.54	805	38
Minimum Maximal Matching	92	.10	38	.73	535	22

Table 2: Search Control Learning, Experimental Results

what operationalizations will produce useful rules, without knowing exactly what problems the system will eventually be tried on.

An alternative approach is to use BFSF induction for generating control rules. (Actually, as we will see, it makes sense to use both learning methods for robustness!) Minton (1990) and Etzioni & Minton (1992) have argued that smaller control rules tend to be more efficient and more general. We will not review these arguments here, but we will show empirically that BFSF induction produces good control rules.

We begin by considering how candidate control rules approximating the "most-constrained variable" heuristic can be learned. In MULTI-TAC, variable ordering preference rules[3] take the form "(Prefer v) if (P v)" where (P v) is an arbitrary formula containing v. The BFSF method generates all candidate variable-ordering rules up to size s, and tests them using examples that illustrate the most-constrained heuristic. To find examples we run our CSP problem solver (without any ordering heuristics) on randomly selected problem instances and periodically stop the solver at variable selection choice points. Each example consists of a pair of variables and a state, such that one variable is a most-constrained variable in that state and the other variable is not. A variable is "most-constrained" if no other variable has fewer possible values.

We test each rule on each example by seeing if the antecedent holds for the most-constrained variable and does not hold for the other variable. In this case we say the rule was correct on the example. Since we do not expect our rules to be one-hundred percent correct, we simply retain all rules which are correct more often than they are incorrect. The utility evaluation module then finds the best combination of these rules.

MULTI-TAC can learn other types of control rules similarly, such as rules that prefer the "least constraining value". The main requirement is an inexpensive way of generating examples. Unfortunately, for some generic heuristics, we have not yet found an inexpensive way of producing examples. For instance, we would like to induce rules that recognize problem symmetries (Minton 1993a) but we do not know an inexpensive way of generating examples of symmetries.

We experimentally evaluated the inductive method by comparing to the analytic method on the 6 problems introduced earlier. This is a significant test for the inductive method because in previous experiments (Minton 1993b; 1993a) the analytic method produced very good results – in some cases, MULTI-TAC's programs were faster than those of human programmers. To compare the inductive and analytic approaches on each problem, both methods were used to produce most-constrained-first variable-ordering rules and least-constraining-first value-ordering rules. The BFSF method generated rules up to a size limit of 4. Thirty examples were used in the BFSF test phase.

In our experiments, the rules produced by the inductive approach resulted in superior programs on four of the six problems. To evaluate the programs, we used one hundred randomly-generated instances as in (Minton 1993b). The columns labeled "Solved" in Table 2 show the primary performance indicator: how many of the 100 test instances were solved within a preset time limit. The columns labeled "Time" show the fraction of the total available CPU time actually used. (This is relevant primarily for GRAPH 3-COLORABILITY and PARTITION-INTO-TRIANGLES where both methods solved most of the instances). The remaining two columns show the total number of rules generated during BFSF search and the number of rules retained.

The inductive approach proved superior on four problems, but its performance was particularly good on BIN PACKING and DOMINATING SET, where substantially more instances were solved. There was also one problem where the inductive approach was substantially inferior, MMM. We analyzed why the inductive approach performed poorly on MMM, and found that the "good" rules (those learned by the analytic approach) were generated, but they did not do well in the test phase. We believe the problem arises because our examples are produced during un-informed search, which in some respects does not mirror the situations that occur when control rules are used.

Finally, we note that, as in our axiom learning experiments, very few examples were necessary for our experiments. As can be seen from the last two columns, it was relatively rare for a rule to be retained.

[3] There is also an alternative syntax for preference rules that allows individual candidates to be numerically scored. We also generate rules of this form, but since the syntax involves the set generator construct which we avoided discussing in the last section, we will not describe it here.

Discussion

In the preceding sections we outlined a relatively simplistic, brute force approach for generating axioms and search control knowledge in MULTI-TAC. Initially, we were surprised that the approach worked so well. In retrospect, we can identify several important factors that contribute to its success.

First, computers are very good at brute force search. They can quickly examine large numbers of candidate formulas. But even so, brute force would be out of the question without a strong bias for small formulas. Thus, a second contributing factor is the "smallest-first" bias. This bias is clearly appropriate for our applications. Both the theorem prover and the search control module are much more likely to be effective if the learned formulas (the axioms and search control rules) are concise.

A third reason for the success is that certain characteristics of our problem domains help keep the number of small candidate formulas manageable. Most importantly, each problem specification generally includes only a few user-defined predicates. This is similar to having only a few "features" per domain. Similarly, the problem specification generally only mentions a few constants. (Recall that the generator uses only those constants mentioned in the problem specification, essentially a form of language bias.)

These justifications are still insufficient to explain our results, however. The situation reminds us of the story of the drunk who was looking for his keys by the lamppost. (The drunk looks for his keys by the light, even though he thinks he may have dropped them somewhere else.) While we know that small formulas are preferable for our applications and furthermore, that we can easily search the space of small formulas, do we really have any confidence that our target formulas (either axioms or control rules) will actually BE small? Clearly there is no guarantee. But we note that the language has been designed so that users can easily and concisely specify the constraints on a large variety of combinatorial problems. Furthermore, when the user formulates a given problem domain, he will tend to define relations that will make the constraints simple to specify. Therefore, because we believe that the problem constraints will usually be concisely specified, we also have some confidence that the axioms and search control rules will be concisely specifiable as well. This is hard to formalize or quantify, but we believe that it is a significant factor.

Finally, we have already remarked upon the fact that very few training examples were required for our applications. On a related note, we should mention that the issue of noise is not an important consideration in our applications. The theorem-proving application is noise-free, and the search control application does not involve noise in the traditional sense.

Interestingly, the factors we have identified as important to the success of our approach should hold for a variety of other tasks besides automated software synthesis. In particular, BFSF induction might prove useful in other design tasks where a user creates a problem specification and the design task is broken down into small subcomponents.

Limitations and Future work

The most obvious limitation of our approach is that the number of generated formulas grows exponentially as the size bound is increased.[4] In our experiments, it typically took under a minute for a Sparc2 to generate all formulas of size 4. Generating formulas of size 5 can usually be accomplished in several minutes. For the axiom-learning application, the number of size 5 formulas ranges from 3450 for MMM to almost 34000 for DOMINATING SET. In practice, however, generating the formulas is not the bottleneck. In order to test whether a formula is consistent with an example, MULTI-TAC converts each formula to an Lisp procedure which is then compiled. The most time-consuming aspect of the induction process is running the Common-Lisp compiler on these procedures.[5] For 5000 formulas, this takes more than an hour. Once the formulas have been compiled, testing the examples is relatively quick.

Thus, the most profitable improvement would be to reduce the number of formulas generated. Currently, GENERATE-FORMULAS often produces "nonsensical" subformulas that are unsatisfiable. In principal these could be identified by a theorem-prover if it had the appropriate domain axioms. We are currently investigating a much more practical approach that uses examples to identify subformulas that are probably unsatisfiable. Instead of generating all formulas and testing them on the examples, we interleave the generation and test processes. Any subformula that is false on all examples is eliminated during the generation phase.

Another improvement involves the invention of new predicates to reformulate the constraints. As we explained previously, the success of our approach depends on whether the target concepts can be expressed concisely, given the constraint language and the user-defined predicates. In some cases, a concise representation may require the invention of new predicates. How can we invent appropriate predicates? One way is to look for transformations that rewrite the problem constraints in a more concise form. For example, the MMM problem constraints could be rewritten more concisely given a "neighbor" predicate, such that

[4] Assume the domain has p predicates and k constants, where r is the maximum arity of any predicate. Then, since a formula of size s has at most s atomic formulas, there are at most sr distinct variables in any formula. Consequently, if α is the number of literals that could be generated, then $\alpha \leq 2p(k+sr)^r$. Since a formula of size s may include no more than s quantifiers and connectives (of which there are 4), the number of possible formulas is $O(4^s\alpha^s)$.

[5] We also note that eliminating redundant axioms is time consuming, since the theorem prover is used.

(**neighbor** $edge_1$ $edge_2$) if $edge_1$ and $edge_2$ share a common endpoint. Currently, MULTI-TAC is capable of inventing new predicates (through finite-differencing and related techniques), but these are used only to rewrite the problem constraints for efficiency purposes. We plan to investigate whether the invented predicates can improve the induction process as well.

Related Work

Most the the empirical work on first-order learning has been in the context of inductive logic programming (ILP), a paradigm which is quite different from ours (Muggleton 1992). In ILP the target language is usually restricted to horn clauses, while we allow full first-order formulas. On the other hand, ILP methods can be used to learn recursive programs, while we are interested only in simple formulas. Furthermore, ILP methods are generally concerned with optimizing accuracy in an environment that is typically noisy. In contrast, our applications have led us to focus primarily on efficiency. Finally, perhaps the most significant difference is that ILP systems usually operate by generalizing or specializing hypotheses. In our approach, the hypothesis space is explicitly enumerated.

All in all, there has been surprising little applied work using brute force enumeration of formulas, perhaps because the approach seems so simplistic. Weiss et al.(1990) describe an algorithm that looks for the best logical expression of a fixed length or less that covers a sample population, but they search the space heuristically. Systematic enumeration techniques have received a bit more attention from researchers interested in more restricted languages. Riddle, Segal and Etzioni (1994) report good results with an exhaustive, depth-bounded algorithm for learning decision trees, and Schlimmer (1993) describes an exhaustive, but efficient, method for learning determinations.

Methodologically, we were influenced by Holte's (1993) study showing that one-level decision trees perform well on many commonly used datasets. Holte advocates exploring algorithms that have small hypothesis spaces, a methodology he refers to as "simplicity-first". If a simple algorithm works, then one can analyze why it worked, otherwise the hypothesis space can be expanded to rectify specific deficiencies. We view our work as an instance of this methodology.

Conclusion

We have shown that brute force induction is surprisingly useful for learning axioms and control rules in MULTI-TAC. Our approach relies on a bias in favor of small formulas. One reason that this bias is appropriate is that small formulas tend to have much higher *utility* in our applications, a point rarely discussed in the induction literature.

The success of our approach also depends on certain aspects of the domain. In particular, we limit the predicates and terms in our generalization language to those mentioned in the problem specification, a type of a language bias. We pointed out that our approach will be successful only if the target concepts can be represented concisely. Since presumably the language allows the problem constraints to be represented concisely, we conjectured that the language is also sufficient to allow the target concepts to be represented concisely.

We expect that our approach will prove useful for many of the problems that MULTI-TAC encounters. It also seems plausible that the approach will work for other types of design problems, and thus may be a promising avenue for further research.

References

Blumer, A.; Ehrenfeucht, A.; Haussler, D.; and Warmuth, M. K. 1987. Occam's razor. *Information Processing Letters* 24:377–380.

Cheeseman, P.; Kanefsky, B.; and Taylor, W. 1991. Where the *really* hard problems are. In *IJCAI-91.*

Etzioni, O., and Minton, S. 1992. Why EBL produces overly-specific knowledge: A critique of the prodigy approaches. In *Proc. Ninth International Machine Learning Conference.*

Garey, M., and Johnson, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co.

Holte, R. 1993. Very simple classification rules perform well on most commonly used datasets. *Machine Learning* 1(11):63–90.

Minton, S. 1990. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence* 42.

Minton, S. 1993a. An analytic learning system for specializing heuristics. In *Proc. IJCAI-93.*

Minton, S. 1993b. Integrating heuristics for constraint satisfaction problems: A case study. In *Proc. AAAI.*

Mitchell, D.; Selman, B.; and Levesque, H. 1992. Hard and easy distributions of SAT problems. In *Proc. AAAI-92.*

Muggleton, S. 1992. *Inductive Logic Progamming.* Academic Press.

Riddle, P.; Segal, R.; and Etzioni, O. 1994. Representation design and brute-force induction in a Boeing manufacturing domain. *Applied Artificial Intelligence* 8:125–147.

Schlimmer, J. 1993. Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In *Proc. Tenth International Machine Learning Conference.*

Weiss, S.; Galen, R.; and Tadepalli, P. 1990. Maximising the predictive value of production rules. *Artificial Intelligence* 45.