An Analytic Learning System for Specializing Heuristics

Steven Minton
Sterling Software
AI Research Branch
NASA Ames Research Center, MS 269-2
Moffett Field, CA 94035-1000, U. S. A.
minton@ptolemy.arc.nasa.gov

Abstract

This paper describes how meta-level theories are used for analytic learning in MULTI-TAC. MULTI-TAC operationalizes generic heuristics for constraint-satisfaction problems, in order to create programs that are tailored to specific problems. For each of its generic heuristics, MULTI-TAC has a meta-theory specifically designed for operationalising that heuristic. We present examples of the specialisation process and discuss how the theories influence the tractability of the learning process. We also describe an empirical study showing that the specialised programs produced by MULTI-TAC compare favorably to hand-coded programs.

1 Introduction

MULTI-TAC (Multi-Tactic Analytic Compiler) is a learning system for constraint-satisfaction problems (CSPs). The system operationalises generic heuristics[ll], producing problem-specific versions of these heuristics, and then attempts to find the most useful combination of these heuristics on a set of training problems.

This paper focuses on the knowledge that MULTI-TAC uses in order to operationalize generic heuristics, and how this approach differs from previous work in "analytic" (or knowledge-based) speed-up learning. Previous analytic speed-up methods, such as EBL, chunking, and derivational analogy, have been used primarily for caching problem-solving experience[9]. Typical EBL systems, for example, learn from problem-solving successes and/or failures by caching a knowledge structure summarising the experience (e.g., a chunk) and then reusing that knowledge during subsequent problem solving. In retrospect, relatively little attention has been paid to the *theories* (i.e., the knowledge) used in the learning process. However, this subject deserves more attention since the choice of theories determines what is learned.

MULTI-TAC employs *meta-level* theories in the learning process. These enable the system to reason *about* the problem solver's base-level theory, as opposed to simply caching the generalised results of the problem solver's search. We argue that this approach is particularly appropriate when solving combinatorial problems, such as scheduling problems.

The system employs a rich variety of meta-level theories. This reflects a shift in research focus from "learning as a caching process" to "learning as an inferential process". The key is to find tractable meta-theories for generating useful search control knowledge. We outline two such theories, and discuss how the representation of the meta-level theories and the representation of the underlying constraint-satisfaction task influences the utility of the learning process. We also describe an empirical study in which MULTI-TAC compared favorably with hand-coded programs.

2 Theories and Analytic Learning

Informally, analytic learning systems are characterized by a "theory-driven" component that generates hypotheses by analyzing a domain. Several analytic approaches have been used for *speed-up* learning, in which the goal is to improve problem-solving efficiency. For example, one approach is to apply EBL to generate possible search control rules[9]. These rules serve as "hypotheses" which are tested by evaluating their utility on a set of problem instances. We will use EBL to illustrate our argument here, although a variety of analytic approaches have been employed for speed-up learning.

The simplest type of EBL normally utilized by a problem solver involves learning from success. In this case, the theory that is used in the explanation process is typically the same theory that the problem solver uses in its search. For example, in macro-operator learning, a simple form of EBL, an explanation is simply an operator sequence which solved a problem.

Learning from failure is slightly more complex, but again, can often be accomplished by using essentially the same theory that the problem solver employs. For instance, if the base-level theory includes the axiom "*P Q1* v Q2", a system can reason about the failure to derive "P" by using an equivalent form of the axiom, ^{U}^*P* *-+ -**Q1* A -**Q2**.

In this paper we explore a more sophisticated approach in which the learning system is given additional "meta-level* knowledge. Meta-level theories enable the learning system to reason *about* the base-level theory that the problem solver employs, so that the learning system can do more than simply generate inferences from the

base-level theory. For example, we show how a domain-independent heuristic, such as the CSP heuristic "prefer the least-constraining-value" can be automatically specialised for a particular problem by reasoning about what "least-constraining" means in the context of that problem. This process involves incorporating information that is not normally used by the base-level problem solver. Although some forms of meta-reasoning have been employed in previous EBL systems (*e.g.,* learning from goal-interactions [9]) this approach remains largely unexplored.

Meta-level analysis is particularly appropriate when the base-level theory is intractable and there is no obvious skew in the distribution of solutions. For example, consider an NP-hard problem such as scheduling or timetabling. It is unlikely that simply caching generalized solutions will prove very useful unless the distribution of instances is extremely biased so that some special cases arise repeatedly. In contrast, applying heuristics that are specialized to the problem is quite likely to be useful. Such heuristics can improve average problem-solving performance on intractable problems without depending on a skewed distribution.

3 MULTI-TAC

MULTI-TAC a speed-up learning system for solving combinatorial problems, such as shop scheduling, timetabling, and bin packing. Such problems abound in industry and government. The system is designed for a scenario where instances of some NP-hard problem must be solved routinely, such as a factory where a schedule must be devised each week. While all NP-complete problems can be reduced to a single problem type, such as Satisfiability, and then solved by some heuristic algorithm, this is generally a poor approach since the relative utility of many heuristics is problem-dependent [6]. Instead, one typically writes a program specifically for the problem at hand by modifying some "off-the-shelf" algorithm and adding appropriate heuristics. Although NP-complete problems are intractable in the worst case, relatively efficient programs can often be designed for specific applications. Indeed, for certain NP-complete problems, such as graph-coloring, simple heuristic methods can produce solutions in polynomial time with high probability even for "random" distributions[l5].

In this paper, we adopt the standard terminology of computer science and use the term "problem" to refer to a generic problem class and "instance" to refer to a particular problem instance. For example, "Graph-3-Colorability" is a problem that requires that each node in a graph be assigned one of three possible colors, such that no two neighbors have the same color. An instance of Graph-3-Colorability would consist of a specific graph and a specific set of colors.

MULTI-TAG'S input consists of a problem specification plus an instance generator for that problem. The instance generator is a "black box" that generates instance specifications according to some distribution. MULTI-TAGS' output is a Lisp program that is tailored to the problem and the instance distribution. Our goal is to synthesize programs that are as efficient as those written by competent programmers (not algorithms experts). Doing so requires the system to have considerable expertise. However, it primarily requires expertise that would be "commonsense" to a programmer, as opposed to a knowledge base of operations research algorithms. (We are not opposed to the latter approach, it is simply not the focus here).

In order to present a problem to MULTI-TAC it must be formalized as an integer CSP, that is, as constraints over a set of integer variables. The problem are described using a first-order, sorted logic, a relatively expressive language for CSPs. A solution exists when all the variables are assigned a value such the constraints on each variable are satisfied. Consider the problem "Graph-3-Colorability". The nodes can be represented by variables whose value can range from 0 to 2. The constraints on variables are specified as follows:

(iff (satisfies *Var Vat)*
  (forall *Neigh-var* such-that (edge *Neigh-var Var)*
    (not (assigned *Neigh-var Vat)))))*

That is, a value satisfies the constraints on a variable iff all neighboring variables have not been assigned that value. The constraint language includes two types of relations, problem-specific *user-defined* relations such as edge and built-in *system-defined* relations such as satisfies and assigned. User-defined relations are defined by explicitly listing their extension in the instance specification. There are variety of built-in relations, such as sum, greater-than, max (of a set), and union.

The graph coloring constraint is particularly simple. However, a wide variety of problems can be specified in the language. Consider another example, Bin Packing. Each object is represented by a variable and each bin by a value. The user-defined relations object-size and bin-capacity relate each variable to a "size" and each bin to a "capacity". The bin-packing constraint is represented as follows (paraphrasing in English): a variable (object) can be assigned a value (bin) iff the bin's capacity minus the object's size is greater than or equal to the sum of the sizes of the other objects assigned to the bin.

The program synthesis process employed by MULTI-TAG is hierarchically organized. At the top level, the system chooses one of a set of generic constraint satisfaction search algorithms, including backtracking and iterative repair [10]. Currently only the backtracking strategy is implemented, so the remainder of the paper assumes a backtracking search. As in the standard CSP backtracking search[7], the system selects a variable and then chooses a value for that variable. Backtracking occurs when all values for a variable fail to satisfy the constraints. Thus, two obvious points for search control knowledge are the choice of which variable to instantiate next and the choice of which value to assign. Associated with the backtracking schema are generic heuristics for choosing variables and values, such as:

- Most-Constrained-Variable-First: This variable ordering heuristic prefers the variable with the fewest possible values left.
- Most-Constraining-Variable-First: A related variable ordering heuristic, this prefers variables that constrain the most other variables.

- Least-Constraining-Value-First: A value ordering heuristic, this heuristic prefers values that constrain the fewest other variables.
- Dependency-Directed Backtracking: If a value choice is independent of a failure, backtrack over that choice without trying alternatives.

To synthesize a program, MULTI-TAC first operationalizes the generic variable and value-ordering heuristics, producing a set of candidate search control rules. This set may be large (typically between 10 and 100 rules in our experiments), since there are a variety of heuristics and each heuristic may be specialized and/or approximated in several different ways. A *system configuration* consists of a combination of control rules and a variety of flag settings controlling other heuristic mechanisms *(e.g.,* a flag indicates whether or not to use forward checking [7]). The system carries out a *utility evaluation* process in which it searches for the most effective system configuration using a hill-climbing search. During this search, the system evaluates the utility of a given configuration by compiling a Lisp program that implements the configuration, and then "experimenting" with the program by running it on a set of instances. (The compilation process also includes a variety of additional optimization techniques, such a finite differencing[l3] and constraint simplification).

The learning process impacts the efficiency of the target code in two ways. First, the generic heuristics are re-expressed as problem-specific rules. Second, the utility evaluation process searches for a combination of these heuristics that yields the best overall performance. The resulting program may include several heuristics, which can act synergistically.

Unfortunately, space does not allow a complete discussion of the architecture. In the next section, we describe how the generic variable- and value-selection heuristics are specialized, the main subject of this paper.

4 Specializing Generic Heuristics

MULTI-TAC is based on the supposition that expert problem solving can result from combining a variety of relatively simple, generic heuristics. The key lies in specializing those heuristics to a given problem and then selecting the most useful combination.

As in the EBL paradigm [9], the learning process requires a target concept and a theory describing the target concept. The result is a specialization of the target concept *(i.e.,* a sufficient condition for the target concept).

In MULTI-TAC each generic heuristic, such as Least-Constraining-Value-First, is a target concept. To specialize the heuristic, we use a meta-theory which describes the heuristic. (For some heuristics there are several meta-theories, each one representing a different tactic for operationalising the heuristic.) The specialization method is very similar to the BBS method used in PRODIGY/EBL [9], except that the entire set of possible specializations is generated (as in Etzioni's STATIC [3]), rather than using an example to guide the specialization process. As discussed in section 6, the meta-theories are designed so that the set of possible specializations is not prohibitively large. Below we present two examples illustrating the specialization process.

4.1 Least-Constraining-Value-First

Our first example illustrates the theory MULTI-TAC employs to operationalize Least-Constraining-Value-First.

The backtracking CSP search is modeled as a series of state changes. In a given state, some variables have a value *assigned,* and the rest of the variables are *unassigned.* Formally, (holds (assigned *Var Vat) S)* if and only if variable *Var* is assigned value *Val* in state 5. In general, for any given state 5 and any *Statement* in the constraint language (holds *Statement* 5) if and only if *Statement* is true in *S.*

Assigning a value to variable *Varl constrains* another variable *Varl* iff the number of values that satisfy the constraints on *Varl* is reduced. A value is a least-constraining value if it would constrain the fewest number of other variables. (This is just one way to formalize the intuitive notion of "least constraining"). The generic control rule which expresses this notion (paraphrased for readability) is:

```
If the current variable to be assigned is Varl
choose a value with the minimum score, where
   the score of Val1 is the number of Varl, such that
      there exists a Val2 such that
         (and (satisfies VarS Val2
                (is-false-upon-assignment Varl Val1
                      "(satisfies Var2 Val2)")
```

In other words, each candidate value *Vail* for variable *Varl* is scored by counting the number of other variables that are constrained when *Vail* is assigned to *Varl,* where variable *Varl* is constrained if for some value *Val2,* (satisfies *Varl Val2)* changes from True to False.[1] The semantics of "Is-false-upon-assignment" can be formally characterized as follows. Let (next-state *S2 S1 Varl Val1l)* be true if *S2* is the state that results from *S1* after *Val1* is assigned to *Varl.* Then (is-false-upon-assignment *Varl Val1 Statement)* is true if:

```
V51,52,
   [(holds Statement S1) A (next-state S2 S1 Varl Val1)}
   => (not (holds Statement S2))]
```

MULTI-TAC derives a specialized control rule by operationalizing "is-false-upon-assignment". The system employs a theory designed specifically for this task. Since space does not permit us to describe the complete theory, below we list the sequence of specializations carried out for our Graph-Colorability example. (We have taken a few liberties for readability).

```
Initial statement to be specialized:
 (is-false-upon-assignment Varl Val1
     "(satisfies Varl Val2)")
```

[1]This is a *local* definition of "constrains", since we only consider the case where assigning a value to one variable directly constrains another variable. For example, if assigning a value to *Varl* constrains *Varl,* which in turn constrains *Var3,* we do *not* say that assigning a value to *Varl* constrains *VarS.*

Expands into:

```
(is-false-upon~as«ignaent  Varl Vail
    "(forall neigh-var such-that (edge neigh-var VarS)
         (not (assigned neigh-var Val2)))")
```

Specializes to:

```
(exists neigh-var such-that (edge neigh-var Var2)
    (is-falsa-upon-assigxunent Varl Val1
         "(not (assigned neigh-var Val2))"))
```

Specializes to:

```
(exists neigh-var such-that (edge neigh-var VarS)
    (and (equal Varl neigh-var)(equal Val1 Val2)))
```

Simplifying, we have:

```
(and (edge Varl Var2)(equal Val1 Val2))
```

As the example shows, the analysis proceeds by recursively propagating "is-false-upon-assignment" inward through the constraint. The meta-theory which guides this analysis describes how each type of statement should be specialized. For example, below we show an axiom for specializing a universally quantified formula. (This is used in the first specialization step above).

```
(only-if
 (is-false-upon-assignment Var Vol
   "(forall x such-that gen-statement test-statement)")
 (exists x such-that gen-statement
   (is-false-upon-assignment Var Val"test-statement)))
```

This axiom indicates that a universally quantified statement over a set will become false if the statement becomes false for at least one member of the set. There are similar axioms for conjunctions, disjunctions, simple predicates, etc.

Incorporating the specialized result in our example back into the generic control rule and then simplifying gives us the following specialized version of least-constraining-value-first:

```
If the current variable to be assigned is Varl
choose a value with the minimum score, where
  the score of Val1 is the number of Var2, such that
     (and (edge Var2 Varl)
          (satisfies Var2 Val1)
```

To determine the score for the color red, for example, this rule counts the number of neighbors which themselves can be colored red (i.e., the number of neighbors which do not themselves have a red neighbor). To understand why this makes sense, consider the extreme case where all neighbors already have a red neighbor. In this case, choosing red does not constrain any neighbor since none could be colored red in any event. Thus the score for red will be 0, and so red will be a preferred value.

Of what value is this specialization process? Consider the generic, unspecialized rule that would be the alternative. This rule would simply count, for each value, how many variables would be constrained by this value. For Graph Colorability the complexity would be $0(k^2nd)$, where k is the number of colors, n is the number of nodes in the graph, and d is the maximum node degree (i.e., number of edges per node) since for every value, each unassigned node in the graph would be examined to see if any of its k possible values would be eliminated, at a cost of d per constraint check. In comparison, the cost of evaluating the specialized rule is $0(kd^2)$, since for each value, each neighbor is examined and a constraint check of cost d is carried out. (Note that d < n, so the savings is at least a factor of k.)

For another illustration of the utility of this process, consider the Bin Packing problem described earlier. Choosing a value to assign to a variable corresponds to choosing a bin for an object. If we simply try each bin in turn, this gives us the "first-fit" heuristic. Specializing Least-Constraining-Value-First gives us (a version of) the well-known "best-fit" heuristic, which prefers the bin with the least remaining capacity that will hold the object. To see why the "best-fit" heuristic is a specialization of Least-Constraining-Value-First, consider that the bin with the least remaining capacity is a "possible bin" for fewer objects than any other bin. Thus, putting the object in this bin constrains the fewest other objects. (Note that there is an alternate argument that the least-constraining bin is the one with the "most remaining capacity". In fact, this results from an alternative specialization. The relative utility of these alternatives depends on the instance distribution.)

4.2 Identifying Symmetric Values

We now describe a meta-theory that enables MULTI-TAC to recognize certain types of symmetries and thereby eliminate unnecessary search. The result is a control rule that carries out a specialized form of dependency-directed backtracking. Once again we will use Graph Colorability as an illustration. Suppose that we are choosing the color for a node, and the possible values include red and green. If no node in the graph is yet colored either red or green, then these two colors can be considered equivalent. If coloring the node red results in failure (i.e. the remainder of the graph cannot be colored consistently), then it makes no sense to try green since it is guaranteed to fail as well. To see why this is true, consider that any solution where the node is colored green could be transformed into a solution where the node is colored red merely by interchanging the labels green and red on all nodes.

This is an example of reasoning by symmetry [5; l]. In the general case, we will consider symmetries where two values Val1 and Val2 are swapped, such that all the variables that are assigned Val1 are re-assigned Val2, and vice versa. A generic control rule for utilizing this information is (paraphrased):

```
If assigning value Val1 to variable Var resulted in failure
    and value Val2 is a possible value for variable Var
    and no variable is assigned value Val1
    and no variable is assigned value ValS
    and interchanging Val1 and ValS
        preserves the solution property
then eliminate Val2 as a possible value for Var
```

We can formalize the last antecedent as follows. Let (interchange S1 S2 Val1 ValS) be true if S2 is the state that results from interchanging Val1 and and Val2 in state S1. Then, interchanging Val1 and Val2 preserves the solution property if:

VS1,s2 [(interchange *S1 St Val1 Valt)*
⇒ V Var_9 [(holds (satisfies Var_S *Valt) S1)*
⇒ (holds (satisfies Var_x *Val1) St)]]*

MULTI-TAC'S task is to operationally express the condition under which *Val1 and Val2 can* be interchanged. As we will show, for Graph Colorability the analysis reveals that this condition simplifies to TRUE, since interchanging *Val1* and *Valt* in *any* solution preserves the solution property. In other problems, *Val1* and *Valt* can be interchanged only in certain circumstances. For instance, in the Bin Packing problem, interchanging two values is equivalent to swapping all the objects assigned to one bin with all the objects assigned to another bin. In this case, the specialisation process reveals that the two bins can be interchanged only if they have equal capacities. Thus, for Bin Packing, the resulting search control rule indicates that if putting an object in a bin fails, then putting it in another bin will also fail provided that the bins have the same capacity and both are empty.

In order to operationally express the conditions under which two values can be interchanged, MULTI-TAC specialises the statement below, using a theory specifically designed for this task.

(interchange-preserves-truth *(Val)*
"(satisfies *Var Vat)")*

The meta-predicate "interchange-preserves-truth" is a two-place relation. The second argument, e.g., "(Satisfies *Var Val)"* is the statement whose truth must be preserved by the interchange. The first argument, e.g., *"(Val)** is a list of the free (logical) variables in the statement that are affected by the interchange. (That is, if *Val = Val1* prior to the exchange then *Val = Val2* afterwards, and if *Val = Val2* prior to the interchange then *Val = Val1* afterwards, otherwise Val is unaffected by the interchange.)

For each type of statement in MULTI-TAC'S language, the meta-theory determines the conditions under which a value interchange will preserve the truth of the statement. MULTI-TAC recursively analyzes the expression, keeping track of the terms to which the interchange applies. Below we show the series of specializations used in the graph coloring example, after having expanded the definition of "satisfies" in the statement above.

(interchange-preserves-truth (Val)
"(forall *Neigh-var* such-that (edge *Neigh-var Var)*
(not (assigned *Neigh-var Vol)))")*

specializes to:
(forall *Neigh-var* such-that (edge *Neigh-var Var)*
(interchange-preserves-truth *(Val)*
"(not (assigned *Neigh-var Vat))"))*

specializes to:
(forall *Neigh-var* such-that (Edge *Neigh-var Var)*
(true))

simplifies to:
(true)

The key step in this analysis is the last specialisation, where it is determined that the truth of "(not (assigned *Neigh-var Vat))"* will be preserved when *Val* is affected by the interchange, e.g., if (not (assigned *neighbor Val1))* is true prior to the interchange, then (not (assigned *neighbor* Vai2)) will be true after the interchange.

Minimum Maximal Matching

	Total CPU Sec	Number Unsolved
Project member	3.4	o
MULTI-TAC	4.6	o 1
Subject 1	165.9	6 1
Simple CSP	915.0	83

K-Closure

	Total CPU Sec	Number Unsolved
Project member	3.4	o 1
MULTI-TAC	4.8	o 1
Subject2	482.2	42
Simple CSP	932.4	75

Figure 1: Performance results on two problems

We are also implementing two related meta-theories to carry out more sophisticated analyses. The first considers interchanging values *Val1* and *Valt* in a solution, except that those variables already assigned at the decision point are left unchanged. For graph coloring, this analysis reveals that if value *Val1* fails, value *Valt* will also fail provided no node presently assigned *Val1* or *Valt* is next to an unassigned node. For Bin Packing, this analysis reveals that if bin *Val1* fails, bin *Valt* will fail provided both bins have the same remaining capacity.

The second meta-theory considers interchanging value *Val1* with Val1+/, where I is some integer. This is not useful for either graph coloring or bin-packing, but is useful for Traveling Salesman, Hamiltonian Circuit, and some types of scheduling problems. Consider the Traveling Salesman problem where the variables correspond to cities, and their values indicate the order in which the cities are visited. The analysis reveals that it does not matter which city is chosen to start the tour.

5 Initial Results

Although MULTI-TAC is still under development, the present version, MULTI-TACI.O, is capable of synthesizing very good algorithms on some problems. For example, on graph coloring, MULTI-TACI.O synthesizes the well-known "Brelaz" algorithm[15]. However, this is not very surprising since the author's previous familiarity with this algorithm influenced the design of MULTI-TAC. A more interesting question is how the system performs on problems that are unfamiliar to the author and other project members.

In order to gauge the system's current effectiveness, we selected two problems from the book "Computers and Intractability" [6], Minimal Maximal Matching and K-Closure. These problems could be easily expressed in MULTI-TAC'S language and, in addition, they appeared amenable to a backtracking approach (based on a cursory examination). Two Ph.D.-level computer scientists working on unrelated projects volunteered to write pro-

grams that we could use for comparison. In addition, one of the MULTI-TAC project members was asked to write programs for both problems. The humans were given access to the same instance generators as MULTI-TAC for the purpose of testing their programs.[2] The instance generators used simple random techniques to generate instances; we did not attempt to construct "tricky" instances. The humans were asked to write the fastest program they could, given their busy schedules, and they spent between 5 and 12 hours coding their programs.

For both problems, Figure 1 compares target code generated by MULTI-TAC with the hand-coded programs and with an unoptimized CSP engine (with no heuristics). The figures show the cumulative running time on 100 instances of both problems. The programs were allowed to run for a maximum of 10 CPU seconds per instance, so the figures also report for each program the number of instances that were not solved within this time bound. The results clearly indicate that MULTI-TAC'S target programs were more efficient than those of our two volunteers (Subject 1 and Subject2), although the project member was able to produce even faster code on both problems. Also, it is clear that the unoptimized CSP engine performed poorly.

Interestingly, the project member's programs were significantly faster than our two subjects' programs. We believe that this illustrates the importance of expertise in writing heuristic algorithms. Whereas the project member has become something of an expert on heuristics for combinatorial problems through his work on MULTI-TAC, the other two subjects do not work in this area.

Obviously, the results are extremely promising. Few, if any, previous speed-up learning systems have performed favorably against hand-coded programs. However, we note that these results do not address the generality of the system, since the problems were not randomly selected from "Computers and Intractability". Thus, while the results indicate the potential of our approach, a more thorough evaluation must eventually be carried out. Of course, MULTI-TAC is still under development and with further work its efficiency and robustness will both improve. (See [8] for a complete description of these experiments, as well as followup experiments.)

6 Discussion: Tract ability of Analysis

MULTI-TAC's approach to generating control knowledge is motivated in part by the PRODIGY/EBL system [9] and Etzioni's subsequent work with STATIC [3]. Like PRODIGY/BBL, MULTI-TAC produces control knowledge by specializing meta-level concepts. However, PRODIGY'S EBL module generates explanations from "first principles" in the following sense: the meta-theories used to explain the examples are essentially descriptions of the problem solver that are interpreted during the explanation process. As a result, the explanations are long and complex, and the explanation process must be guided by heuristics so that useful explanations can be generated tractably.

In contrast, although STATIC is designed to learn the same type of control knowledge as PRODIGY/BBL, STATIC'S meta-theories are much simpler. By "simpler", we mean that the search tree of possible specializations is much smaller. As a result a much higher proportion of the possible specializations can be examined (all of them, in fact). Even though the number of possible specializations is considerably smaller, the useful specializations tend to be included. To a large extent, this observation motivated the development of MULTI-TAC.

How is it that STATIC'S theories can be simpler, yet still include the useful specializations? There are at least two contributing factors that we have identified. The first is that STATIC'S theories are written in a more abstract language than PRODIGY/EBL'S. Although STATIC' theories describe the same target concepts as PRODIGY/BBL (success, failure, goal-interaction), the theories describe the target concepts directly in terms of the planning operators in the base-level theory (what Etzioni called a "problem-space graph"). In comparison, PRODIGY/EBL'S theories have an extra layer of indirection, since the meta-theories describe the target concepts in terms of how the problem solving architecture operates on the base-level theory. The second factor is that the complexity of STATIC'S analyses is depth-bounded in accordance with Etzioni's hypothesis that "recursive" explanations are unlikely to be useful.

In designing MULTI-TAC, we borrowed both of these ideas in an effort to make the analyses as simple as possible. More precisely, our goal is for the search tree of possible specializations to be small enough so that we can tractably generate the entire tree. Like STATIC, the metarlevel analyses in MULTI-TAC operate directly on the base-level theory, *i.e.,* on the constraints in the problem specification. We have not attempted to axiomatize the problem solver, as in PRODIGY/BBL. Instead, the meta-theories used in MULTI-TAC are each designed for specializing a particular generic heuristic, as illustrated by the two theories described in the last section. Compared to the "first principles" approach used in PRODIGY/BBL, MULTI-TAC'S approach is more like that of an "expert system".

In addition, MULTI-TAC'S analyses are depth-limited, reminiscent of the way STATIC'S analyses are likewise limited. Specifically, the meta-theories in MULTI-TAC are not used to analyze chains of constraints. The system only analyzes direct constraints between variables. In fact, because the specialization process operates by recursively processing the constraint specification (as illustrated in the previous section), the depth of the tree of specializations is limited by the fixed size of the constraint specification. For example, at compile time MULTI-TAC can analyze the conditions under which two variables are arc-consistent, or (by analogy) arc-independent, but cannot analyze the conditions under which two variables are path-consistent, or path-independent. The only way for the system to make use of these latter concepts is for the problem solver to transitively apply the "arc" properties at runtime.

The guiding principle (perhaps "hypothesis" would be

[2]Unfortunately, due to a misunderstanding, Subject2 did not make use of the instance distribution and thus undoubtedly was at a disadvantage.

more accurate) underlying MULTI-TAC'S specialisation process is that relatively simple analyses can produce useful control knowledge. Indeed, our previous experience with PRODIGY/EBL and STATIC indicates that we are much more likely to be successful if the analyses are simple. As discussed by Etsioni and Minton[4], as the proofs become more complex, EBL is more likely to generate specialisations that are overspecific in that they include irrelevant conditions. The complexity of finding a "good" specialisation, or what Etsioni and Minton call a minimal sufficient condition grows with the size of the theory.[3] In MULTI-TAC we have reduced the size of the meta-theories using the techniques outlined above, such that a complete static evaluation typically produces between 10 and 100 search control rules. In contrast, PRODIGY/EBL is capable of producing an unlimited number of search control rules as the size of the instances increases, most of which are useless.

Unfortunately, designing tractable theories requires considerable expertise. In the future, we hope to address this issue. One possibility is to start with intractable theories and to incrementally refine them.

MULTI-TAC bears some resemblance to automatic programming systems that refine a high-level specification by applying correctness-preserving transformations. We were motivated particularly by Smith's KIDS system [13] and related work on knowledge compilation (e.g.,[12; 14]). Our approach is primarily distinguished from these systems by the use of machine learning, in that we generate alternative search control rules and then test them on examples. This approach enables the system to be completely autonomous, in contrast to transformational systems that require the user to direct the transformational process. Nevertheless, the approach we employ for representing and reasoning about constraints could also be employed in a transformational system.

Recently, Ellman [2] and Yoshikawa and Wada [16] have proposed new methods for improving CSP search. In the future we hope to incorporate these into MULTI-TAC, giving it a broader range of possible optimizations.

7 Conclusion

This paper has advocated the use of meta-level theories for analytic learning. We illustrated this approach with two meta-level theories used for speeding up constraint satisfaction in MULTI-TAC. Each met a-theory is designed for operationalizing a specific heuristic in such a way that the number of specializations is limited. In this sense, MULTI-TAC can be considered an expert system for operationalizing the generic heuristics. This approach appears quite promising; in an empirical study, target code produced by MULTI-TAC compared favorably with hand-coded programs.

8 Acknowledgements

I am indebted to several colleagues for their significant contributions to MULTI-TAC: Jim Blythe, Gene Davis, Andy Philips, Ian Underwood and Shawn Wolfe. Peter Cheeseman, Oren Etsioni, Rich Keller, Phil Laird, and Mike Lowry commented on drafts of this paper. And last, but not least, Bernadette Kowalski Minton helped to generate the ideas behind MULTI-TAC.

[3] Unfortunately, a single example, as used in EBL, does not necessarily help the system discriminate between useful and useless specialisations during the explanation process.

References

[1] J.M. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In AAAI Workshop on Tractable Reasoning, 1992.

[2] T. Ellman. Abstraction via approximate symmetry. In IJCAI Proceedings, 1993.

[3] O. Etzioni. A Structural Theory of Explanation-Based Learning. PhD thesis, Carnegie-Mellon, 1990.

[4] O. Etzioni and S. Minton. Why EBL produces overly-specific knowledge: A critique of the PRODIGY approaches. In Machine Learning Conference Proceedings, 1992.

[5] E.C. Freuder. Eliminating interchangeable values in constraint satisfaction. In AAAI Proceedings, 1991.

[6] M.R. Garey and D.S. Johnson. Computers and Intractability. W.H. Freeman and Co., 1979.

[7] V. Kumar. Algorithms for constraint satisfaction problems. AI Magazine, 13, 1992.

[8] S. Minton. Integrating heuristics for constraint satisfaction problems: A case study. In AAAI Proceedings, 1993.

[9] S. Minton, J.G. Carbonell, C.A. Knoblock, D.R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem solving perspective. Artificial Intelligence, 40:63-118, 1989.

[10] S. Minton, M. Johnston, A.B. Philips, and P. Laird. Solving large scale constraint satisfaction and scheduling problems using a heuristic repair method. In Proceedings AAAI-90, 1990.

[11] J. Mostow. Machine transformation of advice into a heuristic search procedure. In Machine Learning, An Artificial Intelligence Approach. Tioga Press, 1983.

[12] J. Mostow. A transformational approach to knowledge compilation. In M.R. Lowry and R.D. McCartney, editors, Automating Software Design. AAAI Press, 1991.

[13] D.R. Smith. KIDS: A knowledge-based software development system. In M.R. Lowry and R.D. McCartney, editors, Automating Software Design. AAAI Press, 1991.

[14] C. Tong. A divide and conquer approach to knowledge compilation. In M.R. Lowry and R.D. McCartney, editors, Automating Software Design. AAAI Press, 1991.

[15] J.S. Turner. Almost all k-colorable graphs are easy to color. Journal of Algorithms, 9:63-82, 1988.

[16] M. Yoshikawa and S. Wada. Constraint satisfaction with multi-dimensional domain. In The First International Conference on Planning Systems, 1992.