Integrating Heuristics for Constraint Satisfaction Problems: A Case Study

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

Abstract

This paper describes a set of experiments with a system that synthesizes constraint satisfaction programs. The system, MULTI-TAC, is a CSP "expert" that can specialize a library of generic algorithms and methods for a particular application. MULTI-TAC not only proposes domain-specific versions of its generic heuristics, but also searches for the best combination of these heuristics and integrates them into a complete problem-specific program. We demonstrate MULTI-TAC's capabilities on a combinatorial problem, "Minimum Maximal Matching", and show that MULTI-TAC can synthesize programs for this problem that are on par with hand-coded programs. In synthesizing a program, MULTI-TAC bases its choice of heuristics on the instance distribution, and we show that this capability has a significant impact on the results.

Introduction

AI research on constraint satisfaction has primarily focused on developing new heuristic methods. Invariably, in pursuing new techniques, a tension arises between efficiency and generality. Although efficiency can be gained by designing very specific sorts of heuristics, there is little to be gained scientifically from devising ever more specialized methods. One attractive option is to devise generic algorithms that can be made efficient by incorporating additional information. A good example of this is AC-5 [Van Hentenryk *et al.*, 1992a], a generic arc-consistency method that can be specialized for functional, anti-functional or monotonic constraints to yield a very efficient algorithm.

The idea of specializing generic algorithms for a particular application can be carried much further. In this paper we evaluate a system, MULTI-TAC (Multi-Tactic Analytic Compiler), that can specialize a library of generic algorithms and heuristics to synthesize programs for constraint-satisfaction problems (CSPs). MULTI-TAC not only proposes specialized versions of its generic heuristics, but also searches for a good combination of these heuristics, and integrates them into a complete application-specific search program.

The issues we explore are quite different from those traditionally investigated in the CSP paradigm; our focus is on the specialization and integration of well-known heuristics for a given application rather than the development of new heuristics. For instance, few authors have explicitly considered that many CSP applications require solving repetitive instances of a problem (such as scheduling a factory) and are not "1-shot" problems. However, in our work we take this into account when selecting heuristics, because, as we will show, the relative utility of different heuristics can depend on the population of instances encountered.

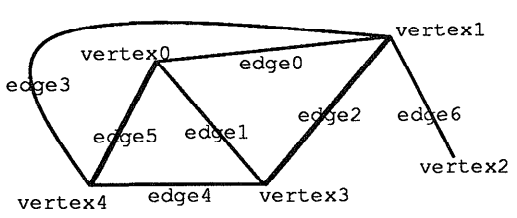
We begin this paper with an overview of the MULTI-TAC system. We then present a case study showing the system's performance on a particular problem and illustrating the utility of automatically tailoring a program to an instance distribution.

The Problem

The input to MULTI-TAC consists of a problem specification and an instance generator for that problem. The instance generator serves as a "black box" that generates instances according to some distribution. The system is designed for a scenario where some combinatorial search problem must be solved routinely, such as a staff scheduling application where each week jobs are assigned to a set of workers.

MULTI-TAC outputs a Lisp program that is tailored to the particular problem and the instance distribution. The objective is to produce as efficient a program as possible for the instance population. In practice, our goal is to do as well as competent programmers, as opposed to algorithms experts. Achieving this level of performance on a wide variety of problems could be quite useful; there are many relatively simple applications that are not automated because programming time is expensive.

For our case study we chose an NP-complete problem, "Minimum Maximal Matching" (MMM), described in [Garey and Johnson, 1979]. The problem is simple to specify, but interesting enough to illustrate the system's performance. An instance of MMM consists of a




```
(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.

```
(iff (satisfies Edge_i Val)
   (and (implies (equal Val 1)
          (forall Vrtx suchthat (endpoint Edge_i Vrtx)
            (forall Edge_j suchthat (endpoint Edge_j Vrtx)
              (or (equal Edge_j Edge_i)
                  (assigned Edge_j 0)))))
        (implies (equal Val 0)
          (exists Vrtx suchthat (endpoint Edge_i Vrtx)
            (exists Edge_j suchthat (endpoint Edge_j Vrtx)
              (and (not (equal Edge_j Edge_i))
                   (assigned Edge_j 1)))))
        (exists Solset (set-of Edge suchthat
                         (assigned Edge 1))
          (exists Solsize suchthat (cardinlty Solset Solsize)
            (leq Solsize K)))))
```

Figure 2: Description of MMM Constraints

graph $G = (V, E)$ and an integer $K \leq | E |$. The problem is to determine whether there is a subset $E' \subseteq E$ with $| E' | \leq K$ 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'.

In order to present a problem to MULTI-TAC, it must be formalized as a CSP. Our CSP specification language is relatively expressive. Variables have an integer range. A *problem specification* defines a set of types (*e.g.*, vertex and edge) and a set of constraints described in a predicate logic. An *instance specification* (see Figure 1) instantiates the types and relations referred to in the problem description.

To formulate MMM as a CSP, we employ a set of boolean variables, one for each edge in the graph. If a variable is assigned the value 1, this indicates the corresponding edge is a member of the subset E'. A value of 0 indicates the corresponding edge is not a member of E'. Figure 2 shows how the constraints are stated in the MULTI-TAC problem specification. The statement specifies the conditions under which a value Val satisfies the constraints on a variable $Edge_i$. We can paraphrase these conditions as follows:

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

The constraint language includes two types of relations, problem-specific *user-defined relations* such as **endpoint**, and built-in *system-defined relations*, including **assigned**, **equal**, **leq**, and **cardinality**. The **assigned** relation has special significance since the system uses it to maintain its state. During the search for a solution, variables are assigned values. Thus, for every variable *Var* there is at most one value *Val* such that (**assigned** *Var Val*) is true at any time. A *solution* consists of an assignment for each variable such that the constraints are satisfied.

MULTI-TAC's specification language enables us to represent a wide variety of combinatorial problems, including many scheduling problems and graph problems. There are, of course, limitations imposed by the specification language. Currently only decision problems are specifiable, since one cannot state "optimization criteria". (We plan to include this in the future.) Furthermore, the system's expertise is limited in many respects by the pre-defined relations. So, for example, geometric concepts such as "planarity" present difficulties.

The MULTI-TAC Architecture

MULTI-TAC rests on the supposition that intelligent problem solving can result from combining a variety of relatively simple heuristics. At the top level, the synthesis process is organized by an "algorithm schema" into which the heuristics are incorporated. Currently only a backtracking schema is implemented (although we also plan to include an iterative repair [Minton *et al.*, 1992] schema), so the remainder of the paper assumes a backtracking search. As in the standard backtracking CSP search[Kumar, 1992] variables are instantiated sequentially. Backtracking occurs when no value satisfies the constraints on a variable.

Associated with an algorithm schema are a variety of generic heuristics. For backtracking, these can be divided roughly into two types: heuristics for variable/value selection and heuristics for representing and

propagating information. Let us consider variable and value selection first. Below we list five generic variable/value ordering heuristics used by MULTI-TAC:

- 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.

Specialized versions of these heuristics are generated by refining a meta-level description of the heuristic, as described in [Minton, 1993]. The refinement process incorporates information from the problem description in a manner similar to partial evaluation. Alternative approximations are generated by dropping conditions. For example, one of the specializations of Most-Constraining-Variable-First that MULTI-TAC produces for MMM is: Prefer the edge that has the most adjacent, unassigned edges. An approximation of this is: Prefer the edge that has the most adjacent edges.

MULTI-TAC also employs heuristic mechanisms for maintaining and propagating information during search. These include:

- Constraint propagation: MULTI-TAC selects whether or not to use forward checking. If forward checking is used, then for each unassigned variable the system maintains a list of *possible values* – the values that currently satisfy the constraints on the variable.
- Data structure selection: MULTI-TAC chooses how to represent user-defined relations. For example, it chooses between list and array representations.
- Constraint simplification: The problem constraints are rewritten (as in [Smith, 1991] and [Minton, 1988]) so they can be tested more efficiently.
- Predicate invention: MULTI-TAC can "invent" new relations in order to rewrite the constraints, using test incorporation [Braudaway and Tong, 1989; Dietterich and Bennett, 1986] and finite-differencing [Smith, 1991]. For example, in MMM the system can decide to maintain a relation over edges which share a common endpoint (i.e., the "adjacent" relation).

We are currently working on a variety of additional heuristic mechanisms, such as identifying semi-independent subproblems.

Compiling a target program

The specialization and approximation processes may produce many candidate heuristics that can be incorporated into the algorithm schema. For MMM, MULTI-TAC generates 52 specialized variable- and value-selection heuristics. In order to find the best combination of candidate heuristics, MULTI-TAC's evaluation module searches through a space of *configurations*. A configuration consists of an algorithm schema together with a list of heuristics. Multiple heuristics of the same type are prioritized by their order on the list. For example, if two variable-ordering heuristics are included, then the first heuristic has higher priority; the second heuristic is used only as a tie-breaker when the first heuristic does not determine a unique "most-preferred" candidate. A configuration can be directly compiled into a LISP target program by the MULTI-TAC code generator and then tested on a collection of instances.

As mentioned earlier, MULTI-TAC's objective is to find the most efficient target program for the instance distribution. For the experiments reported here, "most efficient" is defined as the program that solves the most instances given some fixed time bound per instance. If two programs solve the same number of instances, the one that requires the least total time is preferred. Although this is a simplistic criterion, it is sufficient for our purposes.

The evaluation module carries out a parallel hill-climbing search (essentially a beam search) through the space of configurations, using a set of training instances, a per-instance time bound and an integer B, the "beam width". The evaluator first tests each heuristic individually by compiling a program with only the single heuristic; the system records the number of problems solved within the time bound and the total time for the training instances solved. The top B configurations are kept. For each of these configurations, the system then evaluates all configurations produced by appending a second heuristic, and so on, until either the system finds that none of the best B configurations can be improved, or the process is manually interrupted. In our experiments with MMM, the search took anywhere from one hour to several hours, depending on B, the time-limit and the difficulty of the training instances.

One drawback to this scheme is that occasionally two heuristics may interact synergistically, even though they are individually quite poor, and MULTI-TAC's evaluator will never discover such pairs. This is a general problem for hill-climbing utility-evaluation schemes [Minton, 1988; Gratch and DeJong, 1992], which Gratch calls the *composability problem* [Gratch and DeJong, 1991]. To overcome this, we have experimented with a preprocessing stage in which MULTI-TAC evaluates pairs of heuristics in order to find any positive interactions, and then groups these pairs together during the hill-climbing search. Unfortunately, this can be time-consuming if there are several hundred candidate heuristics. We believe that meta-knowledge can be used to control the search, so that only the pairs most likely to interact synergistically are evaluated.

Experimental Results

This section describes a set of experiments illustrating the performance of our current implementation, MULTI-TAC1.0, on the MMM problem and contrasting the synthesized programs to programs written by NASA computer scientists. In our initial experiment, two computer scientists participated, one a MULTI-TAC project member (PM) and the other, a Ph.D. working on an unrelated project (Subject1). We also evaluated a simple, unoptimized CSP engine for comparative purposes.

The subjects were asked to write the fastest programs they could. Over a several-day period Subject1 spent about 5 hours and PM spent 8 hours working on their respective programs. The subjects were given access to a "black box" instance generator. The instance generator randomly constructed solvable instances (with approximately 50 edges) by first generating E' and then adding edges. MULTI-TAC employed the same instance generator, and required approximately 1.5 hours to find its "best" configuration.

Table 1, Experiment1, shows the results on 100 randomly-generated instances of the MMM problem, with a 10-CPU-second time bound per instance. The first column shows the cumulative running time for all 100 instances and the second column shows the number of unsolved problems. The results indicate that PM (the project member) wrote the best program, followed closely by MULTI-TAC and then by Subject1. The unoptimized CSP program was by far the least efficient. These conclusions regarding the relative efficiencies of the four programs can be justified statistically using the methodology proposed by Etzioni and Etzioni [1993]. Specifically, any pairwise comparison of the four programs' completion times using a simple sign test is statistically significant with $p \leq .05$. In fact, we note that for the rest of the experiments summarized in Table 1, a similar comparison between MULTI-TAC's program and any of the other programs is significant with $p \leq .05$, so we will forgo further mention of statistical significance in this section.

We also tried the same experiment on another problem selected from [Garey and Johnson, 1979], K-Closure, with very similar results (see [Minton, 1993]). We were encouraged by our experiments with these two problems, especially since MULTI-TAC performed well in comparison to our subjects. It is easy to write a learning system that can improve its efficiency; it is more difficult to write a learning system that performs well when compared to hand-coded programs. [1] (Published MMM algorithms would provide a further basis for comparison, but we are not aware of any such algorithms.) It is also notable that none of the MULTI-TAC project members was familiar with either MMM or K-closure prior to the experiments. Problems that are "novel" to a system's designers are much more interesting benchmarks than problems for which the system was targeted [Minton, 1988].

[1] However, we should note that MMM and K-Closure were not "randomly" chosen from [Garey and Johnson, 1979], but were selected because they could be easily specified in MULTI-TAC's CSP language and because they appeared amenable to a backtracking approach.

Outline of Subject1's algorithm:

```
Procedure Solve(E', EdgesLeft, K)
if the cardinality of E' is greater than K return failure
  else if EdgesLeft = ∅ return solution
       else for each edge e in EdgesLeft
                 if Solve(E' ∪ {e},
                          EdgesLeft −{e} − {AdjcntEdges e},
                          K)
                    return solution
            finally return failure
```

Outline of PM's algorithm:

```
Procedure Solve(E', SortedEdgesLeft, SolSize, K)
if SortedEdgesLeft = ∅ return solution
  else if SolSize = K return failure
       else for each edge e in SortedEdgesLeft
                 SortedEdgesLeft ← SortedEdgesLeft −{e}
                 if Solve(E' ∪ {e},
                          SortedEdgesLeft −{AdjcntEdges e},
                          1 + SolSize,
                          K)
                    return solution
            finally return failure
```

Figure 3: Subject1's algorithm and PM's algorithm

One of the interesting aspects of the experiment is that it demonstrates the critical importance of program-optimization expertise. Consider Subject1's algorithm shown in Figure 3 (details omitted). The recursive procedure takes three arguments: the edges in the subset E', the set of remaining edges $E - E'$, and the parameter K. The algorithm adds edges to E' until either the cardinality of E' exceeds K or a solution is found. PM's algorithm, also shown in Figure 3, improves upon Subject1's algorithm in several respects:

- **Pre-sorted Edges:** The most significant improvement involves sorting the edges in a preprocessing phase, so that edges with the most adjacent edges are considered first. Interestingly, Subject1 reported trying this as well, but apparently he did not experiment sufficiently to realize its utility.
- **No redundancy:** Once PM's algorithm considers adding an edge to E', it will not reconsider that edge in subsequent recursive calls. (This is a source of redundancy in Subject1's program. For example, in Subject1's program, if the first edge in `EdgesLeft` fails it may be reconsidered on each recursive call.)
- **Size of E' incrementally maintained:** PM's program uses the counter *SolSize* to incrementally track the cardinality of E', rather than recalculating it on each recursive call.
- **Efficiently updating SortedEdgesLeft:** Al-

	Experiment1		Experiment2		Experiment3	
	CPU sec	unsolved	CPU sec	unsolved	CPU sec	unsolved
MULTI-TAC	4.6	0	27.8	1	449	7
Project member (PM)	3.4	0	76.8	2	1976	33
Subject1	166	6	-	-	-	-
Subject2	-	-	8.9	0	-	-
Subject3	-	-	-	-	1035	7
Simple CSP	915	83	991	98	4500	100

Table 1: Experimental results for three distributions

though Figure 3 does not show it, PM used a bit vector to represent SortedEdgesLeft. This vector can be efficiently updated via a bitwise-or operation with a stored adjacency matrix.

- **Early failure:** PM's program backtracks as soon as E' is of size K, which is one level earlier than Subject1's program.

MULTI-TAC's program behaves similarly to the hand-coded programs in many respects. The program iterates through the edges, assigning a value to each edge and backtracking when necessary. In synthesizing the program, MULTI-TAC selected the variable-ordering heuristic: "prefer edges with the most neighbors". This is a specialization of "Most-constrained-Variable-First". The system also selected a value-ordering rule: "try 1 before 0", so that in effect, the program tries to add edges to E'. This rule is a specialization of "Least-Constraining-Value-First", since the value 1 is least-constraining with respect to the second constraint (see Section 2). MULTI-TAC also included many of the most significant features of PM's algorithm:

- **Pre-sorted Edges:** MULTI-TAC's code generator determined that the variable-ordering heuristic is *static, i.e.,* independent of variable assignments. Therefore, the edges are pre-sorted according to the heuristic, so that edges with the most adjacent edges are considered first.
- **No redundancy:** MULTI-TAC's program is free of redundancy simply as a result of the backtracking CSP formalization.
- **Size of E' incrementally maintained:** This was accomplished by finite differencing the third constraint (which specifies that $|E'| \leq K$).

Whereas the other two programs are recursive, MULTI-TAC's program is iterative, which is typically more efficient in LISP. There are additional minor efficiency considerations, but space limitations prevent their discussion.

A followup study was designed with a different volunteer, another NASA computer scientist (Subject2). We intended to make the distribution harder, but unfortunately, we modified the instance generator rather naively – we simply made the instances about twice as large, and only discovered later that the instances were actually not much harder. Table 1, Experiment2, shows the results, again for 100 instances each with a ten-CPU-second time limit. The program submitted by PM was the same as that in Experiment1, since PM found that his program also ran quickly on this distribution, and he didn't think any futher improvements would be significant. MULTI-TAC also ended up with essentially the same program as in Experiment1. Our second subject rather quickly (3-4 hours) developed a program which performed quite well. The program was similar to PM's program but simpler; the main optimizations were the same, except that no bit array representation was used. Surprisingly, Subject2's program was the fastest on this experiment, and MULTI-TAC's program finished second. PM's program was slower than the others, apparently because it copied the state inefficiently, an important factor with this distribution because of the larger instance size.

Finally, we conducted a third experiment, this time being careful to ensure that the distribution was indeed harder. We found (empirically) that the instances were more difficult when the proportion of edges to nodes was decreased, so we modified the instance generator accordingly. The results for 100 instances, this time with a 45-second time bound per instance, are shown in Table 1, Experiment3.

Our third subject spent about 8 hours total on the task. His best program used heuristic iterative repair [Minton *et al.*, 1992], rather than backtracking. The edges in E' are kept in a queue. Let us say that an edge is *covered* if it is adjacent to any edge in E'. If E' is not a legal solution, then the last edge in the queue is removed. The program selects a new edge that is adjacent to the most uncovered edges (and not adjacent to any edge in E') and puts it at the front of the queue.

PM spent approximately 4 hours modifying his program for this distribution. The modified program uses iteration rather than recursion, and instead of presorting the edges, on each iteration the program selects the edge that has the most adjacent uncovered edges and adds it to E'. Other minor changes were also included.

For this distribution, MULTI-TAC's program is, interestingly, quite different from that of the previous distributions. MULTI-TAC elected to order its values

	Experiment2 Instances		Experiment3 Instances	
	secs	unslvd	secs	unslvd
MULTI-TAC prgrm2	27.8	1	1527	22
MULTI-TAC prgrm3	804	69	449	7

Table 2: Results illustrating distribution sensitivity

so that 0 is tried before 1. Essentially, this means that when the program considers an edge, it first tries assigning it so it is *not* in E'. Thus, we can view this program as incrementally selecting edges to include in the set $E - E'$. For variable ordering, the following three rules are used:

1. Prefer edges that have no adjacent edges along one endpoint. Since this rule is static, the edges can be pre-sorted according to this criterion.
2. Break ties by preferring edges with the most endpoints such that all edges connected via those endpoints are assigned. (I.e, an edge is preferred if all the adjacent edges along one endpoint are assigned, or even better, if all adjacent edges along both endpoints are assigned).
3. If there are still ties, prefer an edge with the fewest adjacent edges.

Each of the above heuristics is an approximation of "Most-Constrained Variable First". Intuitively speaking, these rules appear to prefer edges whose value is completely constrained, or edges that are unlikely to be in E' (which makes sense given the value-ordering rule).

As shown in Table 1, MULTI-TAC's program solved the same number of problems as Subject3's program, and had by far the best run time in this experiment. Interestingly, MULTI-TAC also synthesized a configuration similar to PM's program, which was rejected during the evaluation stage.

One of the interesting aspects of this experiment is that none of our human subjects came up with an algorithm similar to MULTI-TAC's. Indeed, MULTI-TAC's algorithm initially seemed rather mysterious to the author and the other project members. In retrospect the algorithm seems sensible, and we can explain its success as follows. In a depth-first search, if a choice is wrong, then the system will have to backtrack over the entire subtree below that choice before it finds a solution. Thus the most critical choices are the early choices – the choices made at a shallow depth. We believe that MULTI-TAC's algorithm for the third distribution is successful because at the beginning of the search its variable-ordering heuristics can identify edges that are very unlikely to be included in E'. We note that the graphs in the third distribution are more sparse than the graphs in the other two distributions; so, for instance, the first rule listed above is more likely to be relevant with the third distribution.

An underlying assumption of this work is that tailoring programs to a distribution is useful. This is supported by Table 2, which shows that MULTI-TAC's program2, which was synthesized for the instances in the second experiment, performs poorly on the instances from the third experiment, and vice-versa. (The hand-coded programs show the same trends, but not as strongly.) This appears to be due to two factors. First, there is a relationship between heuristic power and evaluation cost. The programs tailored to the easy distribution employ heuristics that are relatively inexpensive to apply, but less useful in directing search. For example, pre-sorting the edges according to the number of adjacent edges is less expensive than picking the edge with the most adjacent uncovered edges on each iteration, but the latter has a greater payoff on harder problems.

Second, some heuristics may be qualitatively tailored to a distribution, in that their advice might actually mislead the system on another distribution. There is some evidence of this in our experiments. For example, the third variable-ordering rule used by MULTI-TAC in Experiment3 degrades the program's search on the second distribution! The program actually searches fewer nodes when the rule is left out.

Discussion

MULTI-TAC's program synthesis techniques were motivated by work in automated software design, most notably Smith's KIDS system [Smith, 1991], and related work on knowledge compilation [Mostow, 1991; Tong, 1991] and analytical learning [Minton *et al.*, 1989; Etzioni, 1990]. There are also a variety of proposed frameworks for efficiently solving CSP problems that are related, although less directly, including work on constraint languages [Guesgen, 1991; Lauriere, 1978; Van Hentenryk *et al.*, 1992b], constraint abstraction [Ellman, 1993], and learning [Day, 1991]. Perhaps the closest work in the CSP area is Yoshikowa and Wada's [1992] approach for automatically generating "multi-dimensional CSP" programs. However, this work does not deal with the scope of heuristic optimizations we deal with. In addition, we know of no previous CSP systems that employ learning techniques to synthesize distribution-specific programs.

In the future, we hope to use the system for real applications, and also to characterize the class of problems for which MULTI-TAC is effective. Currently we have evidence that MULTI-TAC performs on par with human programmers on two problems, MMM and K-closure. (MULTI-TAC is also capable of synthesizing the very effective Brelaz algorithm [Turner, 1988] for graph-coloring). However, we do not yet have a good characterization of MULTI-TAC's generality, and indeed it is unclear how to achieve this, short of testing the program on a variety of problems as in [Lauriere, 1978].

This study has demonstrated that automatically specializing heuristics is a viable approach for synthesizing CSP programs. We have also shown that the

utility of heuristics can be sensitive to the distribution of instances. Since humans may not have the patience to experiment with different combinations of heuristics, these results suggest that the synthesis of application-specific heuristic programs is a promising direction for AI research.

In our experiments, we also saw that MULTI-TAC can be "creative", in that the system can take combinatorial problems that are unfamiliar to its designers and produce interesting, and in some respects unanticipated, heuristic programs for solving those problems. This is purely a result of the system's ability to specialize and combine a set of simple, generic building blocks. By extending the set of generic mechanisms we hope to produce a very effective and general system. At the same time, we plan to explore the issues of organization and tractability that arise in an integrated architecture.

Acknowledgements

I am indebted to several colleagues for their contributions to MULTI-TAC. Jim Blythe helped devise the specialization theories and search control mechanism, Gene Davis worked on the original CSP engine and language, Andy Philips co-developed the code generator, Ian Underwood developed and refined the utility evaluator, and Shawn Wolfe helped develop the simplifier and the utility evaluator. Furthermore, Ian and Shawn did much of the work running the experiments reported in this paper. Thanks also to my colleagues at NASA who volunteered for our experiments, to Oren Etzioni for his advice, and to Bernadette Kowalski-Minton for her help revising this paper.

References

Braudaway, W. and Tong, C. 1989. Automated synthesis of constrained generators. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence.*

Day, D.S. 1991. Learning variable descriptors for applying heuristics across CSP problems. In *Proceedings of the Machine Learning Workshop.*

Dietterich, T.G. and Bennett, J.S. 1986. The test incorporation theory of problem solving. In *Proceedings of the Workshop on Knowledge Compiliation.*

Ellman, T. 1993. Abstraction via approximate symmetry. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence.*

Etzioni, O. and Etzioni, R. 1993. Statistical methods for analyzing speedup learning experiments. *Machine Learning.* Forthcoming.

Etzioni, O. 1990. *A Structural Theory of Explanation-Based Learning.* Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University.

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

Gratch, J and DeJong, G. 1991. A hybrid approach to guaranteed effective control strategies. In *Proceedings of the Eighth International Machine Learning Workshop.*

Gratch, J and DeJong, G. 1992. An analysis of learning to plan as a search problem. In *Proceedings of the Ninth International Machine Learning Conference.*

Guesgen, H.W. 1991. A universal progamming language. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence.*

Kumar, V. 1992. Algorithms for constraint satisfaction problems. *AI Magazine* 13.

Lauriere, J.L. 1978. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence* 10:29–127.

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

Minton, S.; Johnston, M.; Philips, A.B.; and Laird, P. 1992. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 58:161–205.

Minton, S. 1988. *Learning Search Control Knowledge: An Explanation-based Approach.* Kluwer Academic Publishers.

Minton, S. 1993. An analytic learning system for specializing heuristics. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence.*

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

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

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

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

Van Hentenryk, P.; Deville, Y.; and Teng, C-M. 1992a. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence* 57:291–321.

Van Hentenryk, P.; Simonis, H.; and Dincbas, M. 1992b. Constraint satisfaction using constraint logic programming. *Artificial Intelligence* 58:113–159.

Yoshikawa, M. and Wada, S. 1992. Constraint satisfaction with multi-dimensional domain. In *Proceedings of the First International Conference on Planning Systems.*