# Selecting Task Decompositions for Constrained Heuristic Search*

**Elise H. Turner and Roy M. Turner**
Department of Computer Science
University of Maine
Orono, ME 04469-5752
E-Mail: eht | rmt@umcs.maine.edu

From: AAAI Technical Report WS-00-02. Compilation copyright © 2000, AAAI (www.aaai.org). All rights reserved.

## Abstract

There is a large class of problems that can be represented by task decomposition trees which specify alternative sets of variables that must be given values. Any one of these alternatives can be represented as a constraint satisfaction problem. We describe a technique for selecting a task decomposition that results in a constraint problem that can be solved efficiently. This technique takes advantage of the constraint graph textures suggested by work on constrained heuristic search. First, texture-based heuristics developed for the task decomposition tree are used to select alternative subtasks. As each subtask is selected, its variables, and any constraints associated with them, are added to the constraint graph and constraints are propagated. In addition, variables in the constraint graph may be assigned a value when suggested by texture-based heuristics. When a complete decomposition has been selected, constrained heuristic search is used to finish solving the constraint problem. Experimental results suggest that the use of texture-based heuristics for selecting the task decomposition and for deciding when to assign values in a partial constraint graph result in a significant improvement in efficiency and time needed for solving the problem.

## Introduction

There is a large class of problems which can be characterized by task decomposition trees (TDTs). A TDT is an AND-OR tree representing alternative ways of accomplishing a task. Children of AND-nodes represent subtasks, while children of OR-nodes are *alternatives* for achieving a subtask. In many cases, the leaves of the TDT contain variables which must be assigned without violating some set of constraints, creating a constraint satisfaction problem (CSP) that must be solved for the selected task decomposition.

In this paper, we describe how the flexibility in choosing a task decomposition can be exploited to create a CSP that can be solved efficiently. We use *constrained heuristic search* (CHS) (Fox, Sadeh, & Baykan 1989) to solve the CSP so that we can exploit texture-based heuristics to estimate how alternatives within the task decomposition will affect the CSP. However, we do not attempt to represent the TDT in the CHS formalism. Consequently, our approach can be used on TDTs that are generated from a planner or specified by some other means without reference to the CHS formalism. After a more detailed discussion of our approach and an example, we present experimental results which suggest that texture-based heuristics can be used successfully to create CSPs directly from a TDT.

## Using CHS with Task Decomposition Trees

CHS combines constraint satisfaction and heuristic search by treating constraint graphs as search states. The constraint graph contains three types of nodes: variable nodes, constraint nodes and satisfiability specifications. Variable nodes are adjacent to the constraints in which they are involved, and constraint nodes are adjacent to the variables which they relate. Satisfiability specifications are adjacent to constraint nodes connected by AND, OR or XOR. Operators are applied to the constraint graph to generate a new search state. Constraints are then propagated within the new search state, and a new operator is selected to create another search state. Operators include adding or deleting constraints or variables and reducing the domains of variables, possibly by making an assignment. Features of the topology of the constraint graph are described by *textures*. Heuristics, based on these textures, are created for a domain and used to select operators.

*This material is based upon work supported under contract N0001-14-98-1-0648 from the U.S. Office of Naval Research. The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred. Portions of this work also appear in (Turner & Turner 1999) and are used by permisson of the *International Journal of Applied Intelligence*.

We extend CHS for use with a TDT as follows:

- Texture-based heuristics are used to determine whether a new subtask should be selected or a value should be assigned to some variable in the constraint graph.
- Texture-based heuristics are used to select the alternative that will be used to satisfy each required subtask.

Our extensions can be viewed as suggestions for how heuristics can be used to select the operators to be applied to the constraint graph. However, it is important to note that the TDT remains separate from the constraint graph. Keeping the TDT separate from the constraint graph is advantageous because it allows the TDT to be represented outside of the CHS formalism. Although satisfiability specifications allow alternative decompositions to be represented easily in some domains (e.g., (Baykan & Fox 1991)), converting a TDT to a CHS graph incurs some cost. It can also limit the options available for selecting the decomposition by forcing a value to be prematurely associated with the variable or by forcing an entire task decomposition to be selected, precluding feedback from one alternative selection from influencing the next.

However, when the TDT is not represented in the CHS formalism, heuristics developed to select an alternative cannot simply measure a texture, but must predict the effect that possible alternatives will have on the textures of the constraint graph. In addition, processing before the decomposition is selected may differ from processing when the constraint graph is completed. For example, different heuristics may be used to determine the next operator to be applied to the constraint graph.

Our algorithm distinguishes between processing which occurs as the task decomposition is being selected and that which occurs after the constraint graph is completed. Before an entire task decomposition has been selected, our algorithm decides whether to select an alternative to add to the constraint graph or to assign a value in the existing constraint graph. In this paper, we will refer to any constraint graph that does not contain an entire task decomposition as a *partial constraint graph*. If the algorithm decides to select an alternative, it chooses an alternative for satisfying a single subtask and adds the variables and constraints to the constraint graph. This constitutes a new state for CHS, and constraints are propagated. Once an entire task decomposition has been selected, our algorithm performs CHS as described in (Fox, Sadeh, & Baykan 1989).

It is important that the appropriate amount of feedback reaches the variables of the TDT without overconstraining the search. This means that the selection of one alternative should be allowed to influence the next, yet the search should not commit to values too early. The former is handled by selecting the alternative for one subtask at a time and propagating constraints when that value is added to the constraint graph. For example, if a variable's domain is restricted, the contention for values in the domains of the variables remaining in the TDT may change. This new information may influence the texture-based assessment of the TDT and cause the alternative that best fits the existing constraint graph to be selected.

In general, we delay committing to values for variables in the constraint graph as long as possible. This allows the decision about which value to select to take advantage of as much global information as possible. In the task decomposition tree itself the only information available to use to estimate textures of the resulting constraint graphs is information about the variables' domains. This is because variables may not belong to all task decompositions, so not all variables in the TDT will be used in the solution. As a result, information that depends on the selection of an alternative cannot be easily and reliably calculated. For example, the number of constraints involving a variable, needed for the variable tightness texture (Fox, Sadeh, & Baykan 1989), is not available. However, variables present in a state's partial constraint graph represent commitments to part of the structure that will be present in the complete constraint graph. We can take advantage of information about the constraints in the partial constraint graph to later help select a value for the variable. Thus, there is some advantage for delaying value selection until a variable is inserted into the constraint graph so that this information can be used.

However, value selection need not wait until the complete constraint graph is built. There are cases when values should be assigned in partial constraint graphs to help inform the alternative selection process. For example, if we can identify variables which compete for particular values and assign them to another value, then we reduce future competition for that value in the complete constraint graph. By propagating the assignment's effects back into the TDT, possibly reducing the domains of some variables, we can use this information to select alternatives.

## An Example Domain

In our example domain, a set of components, each of which may have several capabilities, must be assigned subtasks so that all subtasks in some task decompo-

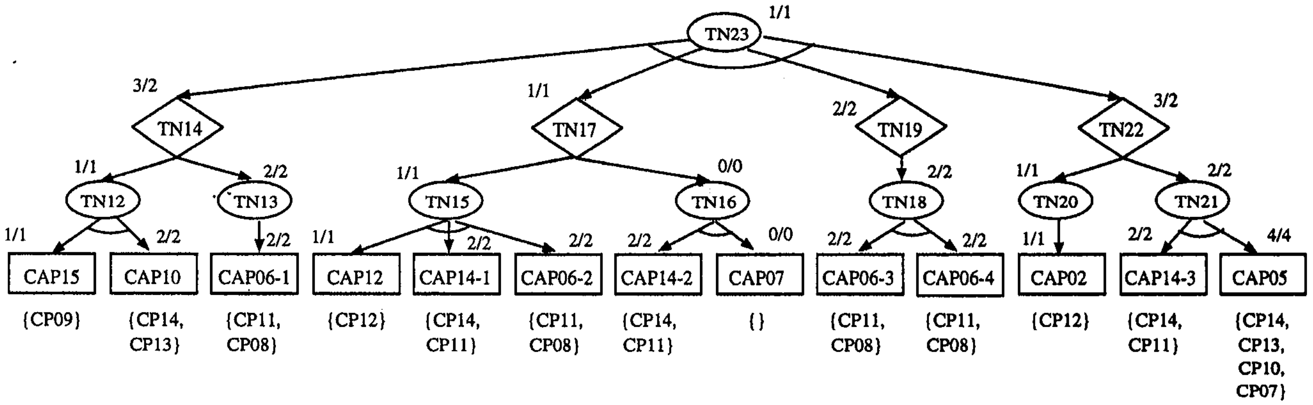

Figure 1: A task decomposition tree, showing the domains of variables. Circles are AND-nodes, diamonds are OR-nodes, and boxes are variables. CAPxx is a variable requiring capability xx, CPyy is a component. Numbers associated with nodes are in the form: AND-rating/OR-rating.

sition have been assigned. Examples of components of this sort are robots that have multiple sensors and effectors that can be operated simultaneously, or computers that can run multiple programs at the same time. Components are assumed to be assigned to a set of tasks for the duration of the execution of the root task.

We assume that the problem is given to the program in the form of a TDT and a description of the set of available components. Primitive tasks appear at the level above the leaves and are represented by AND-nodes whose children represent the capabilities required to carry out the subtask. A leaf specifies the capability required, the number of resource units that will be consumed in performing the capability for the task, and the list of components that can perform the capability (i.e., that have the capability and that have the requisite resource units). The leaves are called *variables*, since they will be added to the constraint graph, and the lists of components are the variables' domains. Figure 1 shows a representative task decomposition tree. Component descriptions include the capabilities that the component can perform and the number of resource units it has.

The constraint graph that corresponds to a decomposition is constructed from the variables in that decomposition of the TDT. Assignment of values to variables are constrained in two ways: the component assigned to the variable must have the capability needed, and no component can be assigned tasks in excess of its total resource units. Currently, constraints cannot be relaxed to produce a solution. We assume that the domains of the variable nodes contain no component that does not have the requisite capability or that does not have enough resource units to perform the capability for that task in isolation. That is, these are the graph's unary constraints, and we assume the constraint graph is node consistent (Mackworth 1977). This can be assured in a simple preprocessing step.

The more interesting kind of constraint is the second type. These are resource constraints on the components and the variables that potentially use them. They are represented as $n$-ary constraints, with the arity dependent on the number of variables in the constraint graph that have the component in their domain. These are the constraints that actually appear in constraint graphs in our approach. When a new variable is added to a constraint graph, a new constraint of arity 1 is added for any component that has a capability needed by that variable, if no other variable in the constraint graph has that value in its domain. If a component has a capability needed by the new variable and one or more capabilities needed by other variables, then a corresponding constraint for it will already exist in the constraint graph. In this case, the existing constraint is linked to the new variable, increasing the constraint's arity by one.

Search begins with an initial state consisting of a TDT and an empty constraint graph. At each state in which there are remaining alternatives in the TDT, the algorithm must decide whether to add an alternative to the constraint graph or assign a value to a variable already in the graph.

To do this, we apply a heuristic based on Fox *et al.*'s *variable contention texture* (Fox, Sadeh, & Baykan 1989) to the partial constraint graph to see how likely it is that a particular value must be assigned to a particular variable. We can estimate the variable contention by taking advantage of the fact that all variables competing for a value are adjacent to the constraint governing the resource units of that value. Each constraint in the graph is given a contention value according to the formula:

$$C_c = \frac{\sum \frac{R_v}{|D_v|}}{R_r}$$

where the summation is over the variables adjacent to the constraint in the graph. $R_v$ is the number of resource units needed by variable $v$, $|D_v|$ is the size of variable $v$'s domain, and $R_r$ is the number of resource units the constraint's component has that are currently unassigned to any variable. This estimates the number of variables contending for the component. Each variable in the graph is given a contention value that is the minimum of all contention values of the constraints on the variable. If any variable has a contention over some threshold, a value is assigned to that variable to relieve contention. The first value tried is the one associated with the variable's constraint that has the *least* contention. If that does not work (i.e., if search backtracks to this point), then the variable's other values are tried in an arbitrary order. If no contention at any variable in the partial constraint graph is over the threshold, then the algorithm selects an alternative to add to the constraint graph instead.

Alternative selection proceeds by selecting variables from the TDT to add to the constraint graph, generating a new state. This process begins at the root of the TDT and makes a selection at each node until a primitive task is reached. At that point, all the primitive tasks' variables are added to the constraint graph.

Two types of selections must be made in this process: AND-selections and OR-selections. Since all the children of an AND-node must be satisfied if the task is to be achieved, all of its children will ultimately be selected. However, it is important to first add the variables to the constraint graph that will be the most difficult to assign.

To select children of AND-nodes, we use a heuristic based on *constraint reliance* (Fox, Sadeh, & Baykan 1989). In a constraint graph, constraint reliance measures the need for a single constraint to be satisfied. We estimate this for the structure of the task decomposition tree based on the number of alternative ways to achieve the subtask. The constraints associated with the subtask must only be satisfied if that alternative is selected. The more alternatives for a subtask, the less we depend on one particular alternative being achieved, and, hence, the less we depend on its associated constraints being satisfied. The goal is to select the subtask with the fewest alternatives to be added to the tree first. To do this, an "AND-rating" is computed for each node in the TDT. This is computed bottom-up. A variable's AND-rating is the length of its domain. An OR-node's AND-rating is the sum of its children's, and an AND-node's AND-rating is the minimum of its children's. When selecting a child at an AND-node, the child with the lowest AND-rating is chosen.

At OR-nodes, choices are made between alternatives that will be placed in the constraint graph, so the alternative that will give the constraint graph the best value for some texture should be selected. We base our selection on the *value goodness* texture (Fox, Sadeh, & Baykan 1989), which is used in selecting operators in CHS. In the constraint graph, this texture measures the likelihood that a particular value assignment leads to a solution (Fox, Sadeh, & Baykan 1989). The corresponding heuristic for the TDT selects the task alternative with the greatest number of values. We calculate this heuristic by computing an "OR-rating" for each node in the TDT. This is done bottom-up, as for the AND-rating, and, in fact, can be done at the same time. Variables have OR-ratings that are the length of their domains. An AND-node has an OR-rating that is the minimum of its children's rating. An OR-node has an OR-rating that is the maximum of its children's. The algorithm selects children of OR-nodes in order of decreasing OR-rating.

When selections have been made for alternatives for all subtasks, a complete constraint graph representing one decomposition will have been created. At this point, our algorithm uses constrained heuristic search to solve the constraint graph. In this phase of the algorithm, we consider only CHS operators which select an assignment for a variable. A new state is created when a value is selected, then constraints are propagated in that state. A solution is reached when a consistent assignment has been made for all the variables.

Two heuristics are used to assign values to variables. The variable tightness texture is used to select the variable, and the value goodness texture is used to select the value which will be assigned.

Consider the example represented by the TDT in Figure 1. The initial state contains the TDT and an empty constraint graph. The first task to work on is selected using the AND-ratings of the children of the top-level node, which estimates the constraint reliance texture in a constraint graph that would include that alternative. TN17 is selected because it has the minimum AND-rating. At OR-nodes, the child with the maximum OR-rating is selected to estimate the value of the value goodness texture in the resulting constraint graph. In this case, the alternative TN15 is selected to achieve TN17. A new state is created containing a constraint graph with the three variables CAP12, CAP14-1, and CAP06-2. The entire branch of the TDT leading to TN17 is pruned in the TDT of the new state, indicating the selection. Four constraints are added to the constraint graph to enforce the resource limits on the new variables and their values, one each for components CP12, CP14, CP11, and CP08. The resulting constraint graph is shown in Figure 2.

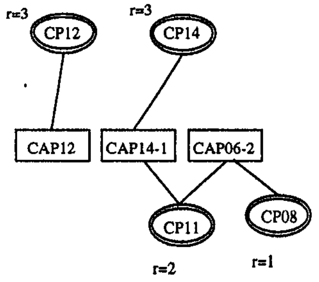

Figure 2: Constraint graph resulting from the TDT in Figure 1 (see text). *r* values are resources a component has available.

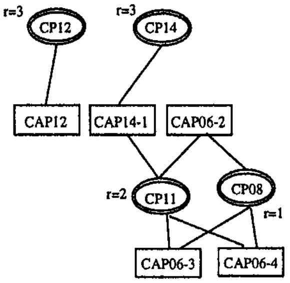

Figure 3: Constraint graph resulting from the TDT in Figure 1 (see text). *r* values are resources a component has available.

From this state, the program again starts at the top-level AND-node. This time, it selects TN19 and its only child, TN18. Two variables for the two instances of CAP06 are added to the constraint graph in a new state, and the branch is pruned. The resulting constraint graph is shown in Figure 3. At this point, the program notices three variables whose contentions are above the threshold: all of the variables requiring CAP06 are contending for CP08, which has a contention value of $\frac{3}{2}$. The program selects one of the variables, setting its value to be the component in the domain associated with the constraint having the *lowest* contention, in this case, CP11 (contention = $\frac{3}{4}$). This reduces contention for CP08. The information is propagated back into the state's TDT to update the domains of other variables that this decision may affect, thus potentially reducing future search.

The program continues in this fashion until a completed alternative is in the constraint graph, at which time values are selected and constraints propagated by CHS.[1]

## Experiments

Two programs, *Random* and *Alt-int*, were written to test the hypothesis that our approach results in increased efficiency and decreased run times. *Random*, the control for this experiment, first builds a complete constraint graph for one task decomposition, without selecting any values for variables. It then uses CHS to solve the constraint graph. Textures are not used during alternative selection. Instead, random selections are made at both AND-nodes and OR-nodes in the task decomposition tree to decide which variables to add to the constraint graph. Textures *are* used during CHS, as described above.

*Alt-int* implements our approach. The selection of variables to add to the constraint graph is guided by textures, as described previously. Value selection is interleaved with alternative selection. The variable contention-based heuristic is applied, with a threshold of 1, to determine when values should be selected. When no more choices remain for a decomposition, the constraint graph is solved using CHS as in *Random*.

Both programs generate a new search state when variables are to be added to the constraint graph and when values are selected. In both programs, backtracking occurs when no operator can be applied or when constraint propagation fails in a state. The current versions of the programs use chronological backtracking. In future work, we will explore using backjumping (Sadeh, Sycara, & Xiong 1995; Dechter 1990) or other dependency-directed backtracking to reduce backtracking and to eliminate the need to check variable contention.

Both programs were run on 1750 randomly-generated problems. Of these, 503 had solutions. *Alt-int* was run once on each problem, while *Random* was run 10 times and the results averaged. Each problem had a set of 5 components (i.e., potential values for variables), each with 4 randomly-assigned capabilities drawn from a pool of 20 possible capabilities. Each component was assigned 2 resource units. A task decomposition tree was generated for each problem by randomly selecting the following parameters: number

[1] As an illustration of the power of selecting values during the construction of the constraint graph, a version of the program that did not select values until the constraint graph was completed backtracked over 5000 times while solving this problem, with an efficiency (see below) of 0.0006. Using contention-based value selection, the program did not backtrack and had an efficiency of 1.

| | Run Time | Efficiency |
|---|---|---|
| *Random* | 29.7ms<br>(65.9) | 0.876<br>(0.178) |
| *Alt-int* | 23.1ms<br>(26.6) | 0.924<br>(0.189) |
| $t$-test | $p < 0.005$ | $p < 0.001$ |

Table 1: Mean run times and efficiencies for the experiment. Numbers in parentheses are standard deviations.

of AND node branches (from 1–5); number of OR node branches (1–7); and resource units required by each variable (1–2). Capabilities needed by the variables were randomly drawn from the same pool of 20 possible capabilities. Each task decomposition tree had the same depth (3, not counting the root), corresponding to tasks with 1–5 subtasks, each having 1–7 alternative ways of being accomplished, and each alternative requiring 1–5 capabilities. For the trees with solutions, the number of possible decompositions (possible constraint graphs) ranged from 1 to 12,348, with the size of the solution space ranging up to 56,376.

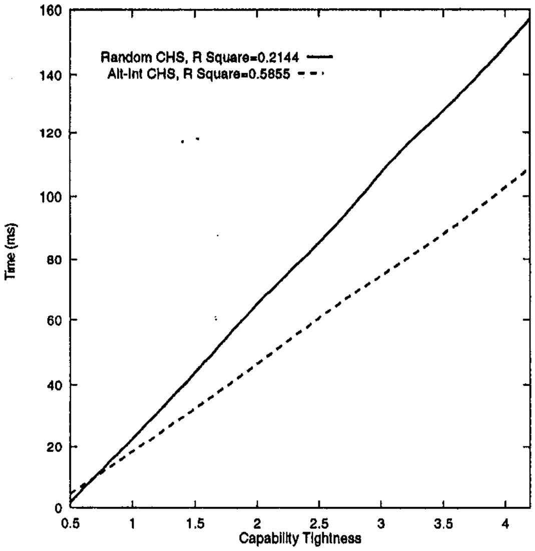

Figure 4: Linear regression of run times for the two programs versus capability tightness.

To estimate the difficulty of solving each tree, a measure called the capability tightness was computed as:

$$\mathrm{CT} = \frac{|V|}{\sum |C_r|}$$

where $|V|$ is the total number of variables in the TDT and $|C_r|$ is the number of capabilities component $r$ has. The intuition behind this measure is that as the number of variables rises relative to the total number of capabilities available, the likelihood will increase that variables will be in contention for components. Consequently, as the capability tightness increases, the problem becomes more difficult to solve, and it becomes more important to make good decisions at all choice points.

For each problem, time and efficiency were measured. Efficiency was computed as the ratio of work needed to work done. Work needed depended on the solution found by the program and was computed as the number of value selections and alternative selections that would have been done had the program made the correct choice at each choice point. Value selections were counted for explicit applications of an operator or when constraint propagation narrowed a variable's domain to a single value. Each time a set of variables was added to a constraint graph, the count of alternative selections was increased. For *Random*, efficiency was computed as work needed over the average work done over all 10 runs.

Runs were performed on a Sun UltraSPARC 140 using Allegro Common Lisp. Statistics were calculated using the CLASP software package (Anderson *et al.* 1995) and Microsoft Excel.

Table 1 shows the results of the runs. The $p$ values[2] were determined using the paired-sample $t$-test (Cohen 1995). *Alt-int* performed significantly better than *Random* with respect to both time and efficiency.

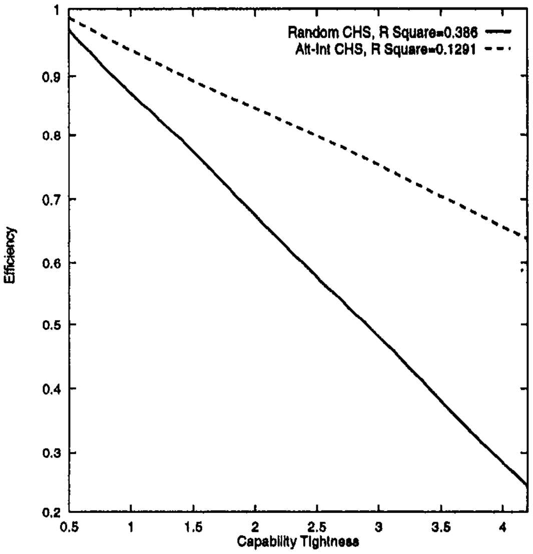

Figure 5: Linear regression of efficiencies of the two programs versus capability tightness.

Figure 4 and Figure 5 show linear regressions of

[2] A measure statistical significance that indicates the probability that the difference in means resulted from chance alone.

time and efficiency, respectively, against tree capability tightness for the two programs. Though the $R^2$ values are low, the results suggest that run time increases and efficiency decreases with increasing capability tightness for both programs, as expected. However, run time of *Random* seems to increase faster than that of *Alt-int*, and its efficiency seems to decrease faster.

## Conclusion

This research focuses on solving problems represented by task decomposition trees using constrained heuristic search. Constrained heuristic search has been integrated with task decomposition problems in the past (Sathi *et al.* 1992; Baykan & Fox 1991). In CORAL (Sathi *et al.* 1992), a task decomposition tree similar to the ones described in this paper is used to specify possible configurations of orders that must be filled by a warehouse, but textures are not used to select alternatives from the task decomposition tree. Wright (Baykan & Fox 1991) does use textures to choose alternative layouts for spatial planning problems. However, Wright relies on hierarchical constraints to represent task decompositions entirely within the CHS formalism. This works well for their domain, but as suggested above, it is less than ideal when the task decomposition tree will be provided by a planner or other problem solver, including a human who is unfamiliar with the formalism.

Our technique both exploits heuristics for selecting alternatives and allows alternative ways of solving the problem to be represented in a standard task decomposition tree. We have presented a technique for selecting a task decomposition by applying heuristics based on textures. These heuristics allow us to make predictions about which alternatives will lead to the most favorable textures in the resulting constraint graph. Experimental results have shown that texture-based heuristics for selecting alternatives and for deciding when to select values in a partial constraint graph lead to significant improvements in run time and efficiency of search.

## References

Anderson, S. D.; Westbrook, D. L.; Schmill, M.; Carlson, A.; Hart, D. M.; and Cohen, P. R. 1995. *Common Lisp Analytical Statistics Package: User Manual.* Department of Computer Science, University of Massachusetts.

Baykan, C. A., and Fox, M. S. 1991. Constraint techniques for spatial planning. In ten Hagen, P., and Veerkamp, P. J., eds., *Intelligent CAD Systems III: Practical Experience and Evaluation.* New York: Springer-Verlag. 187–204.

Cohen, P. R. 1995. *Empirical Methods in Artificial Intelligence.* Cambridge, Massachusetts: The MIT Press.

Dechter, R. 1990. Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *Artificial Intelligence* 41:273–312.

Fox, M. S.; Sadeh, N.; and Baykan, C. 1989. Constrained heuristic search. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89).*

Mackworth, A. 1977. Consistency in networks of relations. *Artificial Intelligence* 8(1):99–118.

Sadeh, N.; Sycara, K.; and Xiong, Y. 1995. Backtracking techniques for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence* 76:455–480.

Sathi, N.; Fox, M. S.; Goyal, R.; and Kott, A. S. 1992. Resource configuration and allocation: A case study of constrained heuristic search. *IEEE Expert* 7(2):26–35.

Turner, E. H., and Turner, R. M. 1999. A constraint-based approach to assigning system components to tasks. *International Journal of Applied Intelligence* 10(2/3):155–172.