Constraint processing for optimal maintenance scheduling

Daniel Frost and Rina Dechter*
Dept. of Information and Computer Science,
University of California, Irvine, CA 92697-3425
{frost, dechter}@ics.uci.edu

Abstract

A well-studied problem in the electric power industry is that of optimally scheduling preventative maintenance of power generating units within a power plant. We show how these problems can be cast as constraint satisfaction problems and provide an "iterative learning" algorithm which solves the problem in the following manner. In order to find an optimal schedule, the algorithm solves a series of CSPs with successively tighter cost-bound constraints. For the solution of each problem in the series we use constraint learning, which involves recording additional constraints that are uncovered during search. However, instead of solving each problem independently, after a problem is solved successfully with a certain cost-bound, the new constraints recorded by learning are used in subsequent attempts to find a schedule with a lower cost-bound.

We show empirically that on a class of randomly generated maintenance scheduling problems iterative learning reduces the time to find a good schedule. We also provide a comparative study of the most competitive CSP algorithms on the maintenance scheduling benchmark.

*The authors thank the Electric Power Research Institute for its support through grant RP 8014-06.

Contents

1 Introduction

1.1 Overview of the Research

This report describes our research on a well-studied problem of the electric power industry: optimally scheduling preventative maintenance of power generating units within a power plant. Our goal was to investigate the efficacy of both standard and new constraint processing techniques on these problems. We took the approach of defining a formal model of maintenance scheduling problem, casting the model in the constraint satisfaction framework, and then evaluating some of the most powerful constraint solving algorithms on the resulting constraint satisfaction problems (CSPs).

Constraint problems are usually treated as decision problems: the algorithm can return any acceptable solution, or a proof that no acceptable solution exists. Because maintenance scheduling is an optimization problem, we developed the use a series of CSPs in which the constraints are tightened until an optimal schedule is found. Our empirical results showed that when applying this optimization scheme, a constraint learning algorithm significantly reduced the CPU time required to find an optimal schedule.

Our research focused on the problem of devising a schedule for the preventative maintenance of power generating units. A typical power plant consists of one or two dozen power generating units which can be individually scheduled for preventive maintenance. Both the required duration of each unit's maintenance and a reasonably accurate estimate of the power demand that the plant will be required to meet throughout the planning period are known in advance. The general purpose of determining a *maintenance schedule* is to determine the duration and sequence of outages of power generating units over a given time period, while minimizing operating and maintenance costs over the planning period, subject to various constraints. The schedule is influenced by many factors, including the length of the maintenance period for each unit, restrictions on when maintenance can be performed, the anticipated power demand for the entire plant, and the cost of maintenance and fuel at various times of the planning period. A maintenance schedule is often prepared in advance for a year at a time, and scheduling is done most frequently on a week-by-week basis.

We propose an approach to maintenance scheduling based on the constraint satisfaction problem framework [5]. In this model, there are a finite number of variables, and associated with each variable is a finite domain of values. A solution to a CSP assigns to each variable a value from its domain, subject to a set of constraints that specify that some combinations of assignments are not allowed. Algorithms for CSPs usually find one or more solutions, or report that no solution exists. Many CSP search algorithms are based on backtracking, or depth-first, search. The general constraint satisfaction problem is NP-complete.

We report the results of applying some of the most powerful constraint processing techniques developed in recent years [11, 10, 12, 4, 24] to the maintenance scheduling problem. Most of the empirical evaluation of constraint algorithms was done with purely random binary CSPs. Applying the algorithms to maintenance scheduling-based CSPs (MSCSPs) provides a testbed of problem instances that have an interesting structure and non-binary constraints. Our empirical results indicate that algorithms which are superior on random uniform binary CSPs are also superior on maintenance scheduling problems, thus providing some validation to the empirical approach based on pure random problems.

The constraint framework consists entirely of so-called *hard* constraints, those which must be satisfied for a solution to be valid. Optimization problems have a cost function as well. To avoid explicitly representing the cost function, or objective function as it is called in the Operations Research literature, we approach optimization as solving a series of related CSPs, each consisting solely of hard constraints. The CSPs in the series differ in that a hard constraint (or group of constraints) corresponding to the objective function with a particular cost-bound is tighter in each succeeding problem in the series. The tighter constraints result from a reduced cost-bound in the function being optimized. An optimal solution is found by determining the lowest cost-bound for which the corresponding constraint satisfaction problem has a solution. A similar approach was used recently to find a shortest plan using satisfiability and CSP techniques [15, 2].

We present experiments with five algorithms that have proven most useful when tested on random problems. In general, when an algorithm is applied to a maintenance problem instance, it solves each of the corresponding CSPs independently. For the new "iterative learning" procedure, an algorithm that learns new constraints during the search is used, and constraints learned during one instance of the series are applied on later instances. This approach was particularly beneficial for the optimization task.

1.2 Structure of the Report

This report consists of six sections. Following the Introduction, the second section describes the Constraint Satisfaction framework and several algorithms that can be used to solve Constraint Satisfaction Problems. Constraint satisfaction techniques have been used to solve a wide variety of scheduling and configuration problems. The development of new algorithms for these problems is an area of intense research in the field of Artificial Intelligence, and in this report we report on the performance of several state of the art algorithms.

Section 3 of the report describes the maintenance scheduling problem and how we have encoded it as a constraint problem. Our approach generally follows that of several other authors, except that we consider how best to interpret the problems as CSPs. The flexibility of the CSP framework means that this procedure can be resolved in many different ways. We present one formulation that is effective.

In section 4 we address the issue of optimization. The constraint framework consists entirely of so-called *hard* constraints, those which must be satisfied for a solution to be valid. Optimization problems can be viewed as having also *soft* constraints, which can be partially satisfied. The problem then is to find the "best" partially satisfied solution. We show how this can be done in the CSP framework.

The results of experimental comparisons of various CSP algorithms are given in Section 5. Our experimental procedure is distinguish by a reliance on a random problem generator, which is described. Using the generator allows us to run algorithms on a large number of problems with defined characteristics, thus reducing the chance that our results will be unduly swayed by the unusual characteristic of one sample instance. We also describe in detail the application of constraint algorithms to a single maintenance scheduling problem.

In the final section we present our conclusions and a summary of our results.

1.3 Related Work

Computational approaches to maintenance scheduling have been intensively studied since the mid 1970's. Dopazo and Merrill [6] formulated the maintenance scheduling problem as a 0-1 integer linear program. Zurm and Quintana [30] used a dynamic programming approach. Egan [7] studied a branch and bound technique. More recently, techniques such as simulated annealing, artificial neural networks, genetic algorithms, and tabu search have been applied [16].

2 Constraint Satisfaction

In this section we define the constraint satisfaction framework, and describe several algorithms that have been developed for it.

2.1 Definition of CSPs

A *constraint satisfaction problem* (CSP) consists of a set of n variables, $X_1, \ldots, X_n$, and a set of constraints. For each variable X_i a *domain* $D_i = \{x_{i1}, x_{i2}, \ldots, x_{id}\}$ with d elements is specified; a variable can only be assigned a value from its domain. A *constraint* specifies a subset of the variables and which combinations of value assignments are allowed for that subset. A constraint is a subset of the Cartesian product $D_{i_1} \times \ldots \times D_{i_j}$, consisting of all tuples of values for a subset $(X_{i_1}, \ldots, X_{i_j})$ of the variables which are compatible with each other. A constraint can also be represented in other ways which may be more convenient. For instance, if X_1, X_2, and X_3 each have a domain consisting of the integers between 1 and 10, a constraint between them might be the algebraic relationship $X_1 + X_2 + X_3 > 15$.

A *solution* to a CSP is an assignment of values to all the variables such that no constraint is violated. A problem that has a solution is termed *satisfiable* or *consistent*; otherwise it is *unsatisfiable* or *inconsistent*. Sometimes it is desired to find all solutions; in this thesis, however, we focus on the task of finding one solution, or proving that no solution exists. A *binary* CSP is one in which each constraint involves at most two variables. A constraint satisfaction problem can be represented by a *constraint graph* that has a node for each variable and an arc connecting each pair of variables that are contained in a constraint.

A variable is called *instantiated* when it is assigned a value from its domain. A variable is called *uninstantiated* when no value is currently assigned to it. Reflecting the backtracking control strategy of assigning values to variables one at a time, we sometimes refer to instantiated variables as *past* variables and uninstantiated variables as *future* variables. We use "$X_i = x_j$" to denote that the variable X_i is instantiated with the value x_j, and "$X_i \leftarrow x_j$" to indicate the act of instantiation.

The variables in a CSP are often given an order. We denote by $\vec{x}_i$ the instantiated variables up to and including X_i in the ordering. If the variables were instantiated in order $(X_1, X_2, \ldots, X_n)$, then $\vec{x}_i$ is shorthand for the notation $(X_1{=}x_1, X_2{=}x_2, \ldots, X_i{=}x_i)$.

A set of instantiated variables $\vec{x}_i$ is *consistent* or *compatible* if no constraint is violated, given the values assigned to the variables. Only constraints which refer exclusively to instantiated variables X_1 through X_i are considered; if one or more variables in a constraint have not been assigned values then the status of the constraint is indeterminate. A value x for a single variable X_{i+1} is consistent or compatible relative to $\vec{x}_i$ if assigning $X_{i+1} = x$ renders $\vec{x}_{i+1}$ consistent.

A variable X_i is a *dead-end* when no value in its domain is consistent with $\vec{x}_{i-1}$. We distinguish two types of dead-ends. X_i is a *leaf* dead-end if there are constraints prohibiting each value in D_i, given $\vec{x}_{i-1}$. X_i is found to be an *interior* dead-end when some values in D_i are compatible with $\vec{x}_{i-1}$, but the subtree rooted at X_i does not contain a solution. Different algorithms may define or test for consistency in different ways. The term dead-end comes from analogy with searching through a maze. At a dead-end in a maze, one cannot go left, right, or forward, and must retrace one's steps.

The most basic consistency enforcing algorithm enforces *arc-consistency*. A constraint satisfaction problem is arc-consistent, or *2-consistent*, if every value in the domain of every variable is consistent with at least one value in the domain of any other selected variable [20, 17, 8]. In general, i-consistency algorithms guarantee that any consistent instantiation of $i-1$ variables can be extended to a consistent value of any ith variable.

An individual constraint among variables $(X_{i_1}, \ldots, X_{i_j})$ is called *tight* if it permits a small number of the tuples in the Cartesian product $D_{i_1} \times \ldots \times D_{i_j}$, and *loose* if it permits a large number of tuples. For example, assume variables X_1 and X_2 have the same domain, with at least three elements in it. The constraint $X_1 = X_2$ is a

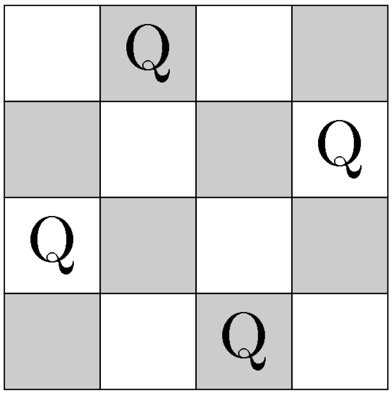


Figure 1: A solution to the 4-Queens problem. Each "Q" represents a queen. No two queens share the same row, column, or diagonal.

tight constraint. Once one variable is assigned a value, only one choice exists for the other variable. On the other hand, the constraint $X_1 \neq X_2$ is a loose constraint, as instantiating one variable prohibits only one possible value for the other.

2.2 An Example of a CSP

As a concrete example of a CSP, consider the N-Queens puzzle. An illustration of the 4-Queens puzzle is shown in Fig. 1. The desired result is easy to state: place N chess queens on an N by N chess board such that no two queens are in the same row, column, or diagonal. In comparison to this short statement of the goal, a specification of a computer program that solves the N-Queens puzzle would be quite lengthy, and would deal with data structures, looping, and possibly function calls and recursion. The usual encoding of the N-Queens problem as a CSP is based on the observation that any solution will have exactly one queen per row. Each row is represented by a variable, and the value assigned to each variable, ranging from 1 to N, indicates the square in the row that has a queen. A constraint exists between each pair of variables. Fig. 2 shows a constraint satisfaction representation of the 4-Queens problem, using this scheme. The four rows are represented by variables R1, R2, R3, R4. The four squares in each row, on one of which a queen must be placed, are called c1, c2, c3 and c4. The constraints are expressed as relations, that is, tables in which each row is an allowable combination of values. The task of a CSP algorithm is to assign a value from {c1, c2, c3, c4} to each variable R1, R2, R3, R4, such that for each pair of variables the respective pair of values can be found in the corresponding relation. The constraint graph of the N-Queens puzzle is fully connected, for any value of N, because the position of a Queen on one row affects the permitted positions of Queens on all other rows.

Another example of a constraint satisfaction problem is Boolean satisfiability (SAT). In SAT the goal is to determine whether a Boolean formula is satisfiable.

Variables: R1, R2, R3, R4. (rows)					
Domain of each variable: {c1, c2, c3, c4} (columns)					
Constraint relations (allowed combinations):					
R1 R2	R1 R3	R1 R4	R2 R3	R2 R4	R3 R4
c1 c3	c1 c2	c1 c2	c1 c3	c1 c2	c1 c3
c1 c4	c1 c4	c1 c3	c1 c4	c1 c4	c1 c4
c2 c4	c2 c1	c2 c1	c2 c4	c2 c1	c2 c4
c3 c1	c2 c3	c2 c3	c3 c1	c2 c3	c3 c1
c4 c1	c3 c2	c2 c4	c4 c1	c3 c2	c4 c1
c4 c2	c3 c4	c3 c1	c4 c2	c3 c4	c4 c2
	c4 c1	c3 c2		c4 c1	
	c4 c3	c3 c4		c4 c3	
		c4 c2			
		c4 c3			

Figure 2: The 4-Queens puzzle, cast as a CSP.

A Boolean formula is composed of Boolean variables that can take on the values *true* and *false*, joined by operators such as $\vee$ (and), $\wedge$ (or), $\neg$ (negation), and "()" (parentheses). For example, the formula

$$(P \vee Q) \wedge (\neg P \vee \neg S)$$

is satisfiable, because the assignment (or "interpretation") (P=*true*, Q=*true*, S=*false*) makes the formula true.

2.3 CSP Algorithms

Over the last 20 years many algorithms and heuristics have been developed for constraint satisfaction problems. Our CSP solver system includes backtracking [3], backmarking [13, 14], forward checking [14], and a version of backjumping [13, 4] proposed in [24] and called there *conflict-directed* backjumping. Space does not permit more than a brief discussion of these algorithms. All are based on the idea of considering the variables one at a time, during a *forward* phase, and instantiating the current variable V with a value from its domain that does not violate any constraint either between V and all previously instantiated variables (backtracking, backmarking, and backjumping) or between V and the last remaining value of any future, uninstantiated variable (forward checking). If V has no such non-conflicting value, then a *dead-end* occurs, and in the *backwards* phase a previously instantiated variable is selected and re-instantiated with another value from its domain. With backtracking, the variable chosen to be re-instantiated after a dead-end is always the most recently

Backtracking with DVO

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable, selected according to a VARIABLE-ORDERING-HEURISTIC (see Fig. 4).
2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) (Forward checking style look-ahead) Examine the future variables $X_i, cur < i \leq n$, For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i; if D'_i is now empty, go to 1 (without examining other X_i's).
3. (Backtrack.) If there is no previous variable, exit with "inconsistent." Otherwise, set cur equal to the index of the previous variable. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 3: The BT+DVO algorithm.

instantiated variable; hence backtracking is often called *chronological* backtracking. Backjumping, in contrast, can in response to a dead-end identify a variable U, not necessarily the most recently instantiated, which is connected in some way to the dead-end. The algorithm then "jumps back" to U, uninstantiates all variables more recent than U, and tries to find a new value for U from its domain. The version of backjumping we use is very effective in choosing the best variable to jump back to.

In the following sections we describe in more detail the algorithms that we used in our experimental evaluation of applying the constraint processing framework to maintenance scheduling problems.

2.4 BT+DVO

We first describe a relatively simple CSP algorithm. This algorithm, called BT+DVO, combines simple backtracking with a dynamic variable ordering scheme called DVO. The algorithm is described in Fig. 3.

The main idea of the variable ordering heuristic, described in Fig. 4, is to select the future variable with the smallest remaining domain as the one which will be instantiated next. This idea was proposed by Haralick and Elliot [14] under the rubric "fail first." We have found that augmenting the fail-first strategy with the

VARIABLE-ORDING-HEURISTIC

1. If no variable has yet been selected, select the variable that participates in the most constraints. In case of a tie, select one variable arbitrarily.
2. Let m be the size of the smallest D' set of a future variable.
   (a) If there is one future variable with D' size $= m$, then select it.
   (b) If there is more than one, select the one that participates in the most constraints (in the original problem), breaking any remaining ties arbitrarily.

Figure 4: The dynamic variable ordering heuristic used by DVO.

tie-breaking rules described in step 1 and step 2 (b) of Fig. 4 produces a 10% to 30% improvement in performance, when compared to BT+DVO without the tie-breakers. The guiding intuition behind the tie-breakers is to select the variable that is the most constraining, and thus most likely to reduce the size of the D' sets of those variables selected after it.

2.5 BT+DVO+IAC

Another algorithm which does more work at each instantiation, by integrating an AC-3 based arc-consistency procedure [18] is BT+DVO+IAC (Fig. 5. Several other algorithms which enforce arc-consistency during search have been proposed [27, 13, 21, 26]. In all versions, values for future variables are removed not only if they are inconsistent with the current partial assignment, but also if they are not compatible with at least one value in the remaining domain of each other future variable. At the cost of more processing per node, BT+DVO+IAC increases the likelihood of detecting early that a partial assignment cannot lead to a solution.

2.6 BJ+DVO

The remaining algorithms are based on backjumping. Backjumping is a variant of backtracking, but after a dead-end it can return to an earlier variable than the immediatedly previous one. The version of backjumping we use is called conflict-directed backjumping [24]. Combining backjumping with dynamic variable ordering is called BJ+DVO.

2.7 BJ+DVO+LVO

The processing of future variables which provides information for dynamic variable ordering can also be used to priorize the values of the current variables, using a tech-

Backtracking with DVO and Integrated Arc-consistency.

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable, selected according to a VARIABLE-ORDERING-HEURISTIC (see Fig. 4).
2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) (Forward checking style look-ahead) Examine the future variables $X_i, cur < i \leq n$, For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i; if D'_i is now empty, go to (e) (without examining other X_i's).
   (d) (Arc-consistency.) Perform AC-3(i).
   (e) Go to 1.
3. (Backtrack.) If there is no previous variable, exit with "inconsistent." Otherwise, set cur equal to the index of the previous variable. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 5: The BT+DVO+IAC algorithm.

nique called look-ahead value ordering (LVO). LVO ranks the values of the current variable, based on the number of conflicts each value has with values in the domains of future variables. Experiments in [9] show that look-ahead value ordering can be of substantial benefit, especially on hard constraint satisfaction problems. Although the LVO heuristic does not always correctly predict which values will lead to solutions, it is frequently more accurate than an uninformed ordering of values.

Combining BJ+DVO with LVO results in BJ+DVO+LVO; a description of this algorithm appears in Fig. 9. The algorithm is essentially the same as BJ+DVO; the differences are in steps 1A, 2 (b), and 2 (c).

Step 1A of BJ+DVO+LVO is where the algorithm's look-ahead phase takes place. The current variable is tentatively instantiated with each value x in its domain D'_{cur}. BJ+DVO+LVO looks ahead, in a forward checking style manner, to determine the impact each x will have on the D' domains of uninstantiated variables. Specifically, the domain value which is in conflict with the fewest values in the domain of future variables is selected first.

In step 2 (b), the current variable is instantiated with the highest ranking value. If the algorithm returns to a variable because of a backjump, the highest ranked

AC-3(d)

$Q \leftarrow \{\text{arc}(i,j) | i > d, j > d\}$
repeat
  select and delete any arc(p,q) in Q
  REVISE(p,q)
  if $D'_p = \emptyset$
  then return
  if REVISE removed a value from D'_p
  then $Q \leftarrow Q \cup \{\text{arc}(i,p) | i > d\}$
until $Q = \emptyset$

Figure 6: Algorithm AC-3.

REVISE(i,j)

for each value $y \in D'_i$
  if there is no value $z \in D'_j$ such that $(\vec{x}_{cur}, X_i{=}y, X_j{=}z)$ is consistent
  then remove y from D'_i

Figure 7: The Revise procedure.

remaining value in its domain is selected. If the variable is re-instantiated after earlier variables have changed, then the ranking of the values has to be repeated in step 1A.

Step 2 (c) essentially disappears in BJ+DVO+LVO; once a value is actually selected, it would not make sense to repeat the look-ahead that has already been done. To avoid repeating consistency checks, our implementation saves in tables the results of step 1A. After a value is chosen in 2 (b), the appropriate D's and Ps of future variables are copied from these tables instead of being recomputed in step 2(c). The space required for these tables is not prohibitive. BJ+DVO uses $O(n^2 d)$ space for the D' sets, where d is the size of the largest domain: n levels in the search tree (D' is saved at each level so that it does not have to be recomputed after backjumping) $\times$ n future variables $\times$ d values for each future variable. Our implementation of BJ+DVO+LVO uses $O(n^2 d^2)$ space. There is an additional factor of d because at each level in the search tree up to d values are explored by look-ahead value ordering. Similarly, the space complexity for the P sets increases from $O(n^2)$ in BJ+DVO to $O(n^2 d)$ for

Backjumping with DVO

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$. Set $P_i \leftarrow \emptyset$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur equal to the index of the next variable, selected according to a VARIABLE-ORDERING-HEURISTIC (see Fig. 4). Set $P_{cur} \leftarrow \emptyset$.
2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) Examine the future variables $X_i, cur < i \leq n$. For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i and add X_{cur} to P_i; if D'_i becomes empty, go to (d) (without examining other X_i's).
   (d) Go to 1.
3. (Backjump.) If $P_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent." Otherwise, set $P \leftarrow P_{cur}$; set cur equal to the index of the last variable in P. Set $P_{cur} \leftarrow P_{cur} \cup P - \{X_{cur}\}$. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 8: The BJ+DVO algorithm.

BJ+DVO+LVO. To solve a typical problem instance described in the next section, BJ+DVO required 1,800 kilobytes of random access memory, and BJ+DVO+LVO required 2,600 kilobytes. On our computer the additional space requirements of LVO had no discernable impact.

2.8 BJ+DVO+LRN

Learning in CSPs, also known as constraint recording, involves a during-search transformation of the problem representation into one that may be search more effectively. This is done by enriching the problem description by new constraints, also called *no-goods*, which do not change the set of solutions, but make the problem more explicit. Learning comes into play at dead-ends; whenever a dead-end is reached a constraint explicated by the dead-end is recorded. Learning during search has the potential for reducing the size of the search space, since additional constraints may cause unfruitful branches of the search to be cut off at an earlier point. The cost of learning is that the computational effort spent recording and then consulting the additional constraints may overwhelm the savings. The kind of learning employed here takes advantage of processing already performed by the backjumping algorithm to identify

Backjumping with DVO and LVO

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set cur to be the index of the next variable, according to a VARIABLE-ORDERING-HEURISTIC. Set $P_{cur} \leftarrow \emptyset$.

1A. (Look-ahead value ordering.) Rank the values in D'_{cur} as follows: For each value x in D'_{cur}, and for each value v of a future variables $X_i, cur < i \leq n$, determine the consistency of $(\vec{x}_{cur-1}, X_{cur}{=}x, X_i{=}v)$. Rank of x based on the number of conflicts it has with future values v.

2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop the highest ranked value x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) (This step can be avoided by caching the results from step 1A.) Examine the future variables $X_i, cur < i \leq n$. For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i and add X_{cur} to P_i; if D'_i becomes empty, go to (d) without examining other X_i's.
   (d) Go to 1.
3. (Backjump.) If $P_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent." Otherwise, set $P \leftarrow P_{cur}$; set cur equal to the index of the last variable in P. Set $P_{cur} \leftarrow P_{cur} \cup P - \{X_{cur}\}$. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 9: Backjumping with DVO and look-ahead value ordering (LVO).

the new constraint to be learned. It can be combined with backjumping and dynamic variable ordering and is called BJ+DVO+LRN. See Fig. 10.

Backjumping with DVO and Learning
Input: *order*

0. (Initialize.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. (Step forward.) If X_{cur} is the last variable, then all variables have value assignments; exit with this solution. Otherwise, set *cur* to be the index of the next variable, according to a VARIABLE-ORDERING-HEURISTIC. Set $P_{cur} \leftarrow \emptyset$.
2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$, go to 3.
   (b) Pop x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) Examine the future variables $X_i, cur < i \leq n$. For each v in D'_i, if $X_i = v$ conflicts with $\vec{x}_{cur}$ then remove v from D'_i and add X_{cur} to P_i; if D'_i becomes empty, go to (d) (without examining other X_i's).
   (d) Go to 1.
3. Learn, then backjump.
   (a) If $P_{cur} = \emptyset$ (there is no previous variable), exit with "inconsistent."
   (b) If X_{cur} was reached by a backjump, go to (dg).
   (c) Perform LEARNING(*order*).
   (d) Set $P \leftarrow P_{cur}$; set *cur* equal to the index of the last variable in P. Set $P_{cur} \leftarrow P_{cur} \cup P - \{X_{cur}\}$. Reset all D' sets to the way they were before X_{cur} was last instantiated. Go to 2.

Figure 10: The BJ+DVO+LRN algorithm.

The learning procedure uses as its conflict-set the parent set P that is explicated by the backjumping algorithm itself. Recall that BJ+DVO examines each future variable X_i and includes X_{cur} in the parent set P_i if X_{cur}, as instantiated, conflicts with a value of P_i that previously did not conflict with any variable. Since the conflict set needed for learning is already assembled by the underlying backjumping algorithm, the added complexity of computing the conflict set is constant. To achieve constant time complexity at each dead-end the parent set must be modified to include not only the parent variables but also their current values.

2.9 BJ+DVO+LRN+LVO

We have combined many of the techniques discussed above into a single algorithm, called BJ+DVO+LRN+LVO. Specifically, this algorithm can be viewed as a merger between BJ+DVO+LRN (Fig. 11) and the look-ahead value ordering heuristic described in Fig. 10. The algorithm uses backjumping, dynamic variable ordering, learning, and look-ahead value ordering.

3 The Maintenance Scheduling Problem

The problem of scheduling off-line preventative maintenance of power generating units is of critical interest to the electric power industry. A typical power plant consists of one or two dozen power generating units which can be individually scheduled for preventive maintenance. Both the required duration of each unit's maintenance and a reasonably accurate estimate of the power demand that the plant will be required to meet throughout the planning period are known in advance. The general purpose of determining a *maintenance schedule* is to determine the duration and sequence of outages of power generating units over a given time period, while minimizing operating and maintenance costs over the planning period, subject to various constraints. A maintenance schedule is often prepared in advance for a year at a time, and scheduling is done most frequently on a week-by-week basis. The power industry generally considers shorter term scheduling, up to a period of one or two

```
LEARNING(order)

CS ← P_cur
if size of CS ≤ order
then RECORD(CS)
```

Figure 11: The jump-back learning procedure.

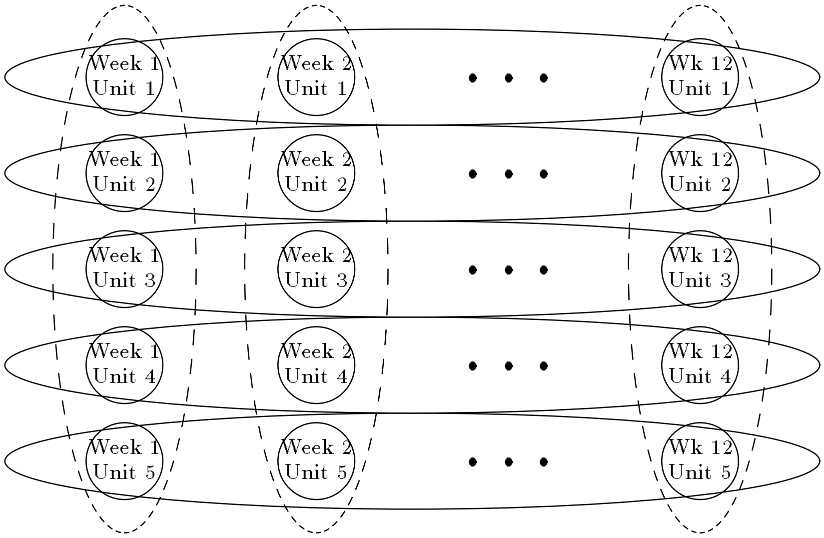


Figure 12: A diagrammatic representation of a maintenance scheduling constraint satisfaction problem. Each circle stands for a variable representing the status of one unit in one week. The dashed vertical ovals indicate constraints between all of the units in one week: meeting the minimum power demand and optimizing the cost per week. The horizontal ovals represent constraints on one unit over the entire period: scheduling an adequate period for maintenance.

weeks into the future, to be a separate problem called "unit commitment."

As a problem for an electric power plant operator, maintenance scheduling must take into consideration such complexities as local holidays, weather patterns, constraints on suppliers and contractors, national and local laws and regulations, and other factors that are germane only to a particular power plant. Our simplified model is similar to those appearing in most scholarly articles, and follows closely the approach of Yellen and his co-authors [1, 28]. The maintenance scheduling problem can be represented by a rectangular matrix (see Fig. 12). Each entry in the matrix represents the status of one generating unit for one week. We will use the terms week and time period interchangeably. A unit can be in one of three states: ON, OFF, or MAINT.

3.1 Parameters

A specific maintenance scheduling problem, in our formulation, is defined by a set of parameters, which are listed in Fig. 13. Parameters U, the number of units, and W, the number of weeks, control the size of the schedule. Many power plants have a fixed number of crews which are available to carry out the maintenance; therefore the parameter M specifies the maximum number of units which can be undergoing maintenance at any one time.

	Input:
U	number of power generating units
W	number of weeks to be scheduled
M	maximum number of units which can be maintained simultaneously
m_{it}	cost of maintaining unit i in period t
c_{it}	operating cost of unit i in period t
k_i	power output capacity of unit i
e_i	earliest maintenance start time for unit i
l_i	latest maintenance start time for unit i
d_i	duration of maintenance for unit i
N	set of pairs of units which cannot be maintained simultaneously
D_t	energy (output) demand in period t
	Output:
x_{it}	status of unit i in period t: ON, OFF or MAINT

Figure 13: Parameters which define a specific maintenance scheduling problem.

In this paragraph and elsewhere in the paper we adopt the convention of quantifying the subscript i over the number of units, $1 \leq i \leq U$, and the subscript t over the number of weeks, $1 \leq t \leq W$. Several parameters specify the characteristics of the power generating units. The costs involved in preventative maintenance, m_{it}, can vary from unit to unit and from week to week; for instance, hydroelectric units are cheaper to maintain during periods of low water flow. The predicted operating cost of unit i in week t is given by c_{it}. This quantity varies by type of unit and also in response to fuel costs. For example, the fuel costs of nuclear units are low and change little over the year, while oil-fired units are typically more expensive to operate in the winter, when oil prices often increase.

Parameter k_i specifies the maximum power output of unit i. Most formulations of maintenance scheduling consider this quantity constant over time, although in reality it can fluctuate, particularly for hydro-electric units.

The permissible window for scheduling the maintenance of a unit is controlled by parameters e_i, the earliest starting time, and l_i, the latest allowed starting time. These parameters are often not utilized (that is, e_i is set to 1 and l_i is set to W) because maintenance can be performed at any time. The duration of maintenance is specified by parameter d_i.

Sometimes the maintenance of two particular units cannot be allowed to overlap, since they both require a particular unique resource, perhaps a piece of equipment or a highly trained crew member. Such incompatible pairs of units are specified in the set $N = \{(i_1, i_2), \ldots, (i_{n-1}, i_n)\}$.

The final input parameter, D_t, is the predicted power demand on the plant in each week t. The parameters x_{it} are the output of the scheduling procedure, and define the maintenance schedule. x_{it} can take on one of three values:

- ON: unit i is on for week t, can deliver k_i power for the week, and will cost c_{it} to run;
- OFF: unit i is off for week t, will deliver no power and will not result in any cost;
- MAINT: unit i is being maintained for week t, will deliver no power, and will cost m_{it}.

3.2 Constraints

A valid maintenance schedule must meet the following constraints or domain requirements, which arise naturally from the definition and intent of the parameters.

First, the schedule must permit the overall power demand of the plant to be met for each week. Thus the sum of the power output capacity of all units not scheduled for maintenance must be greater than the predicted demand, for each week. Let $z_{it} = 1$ if $x_{it} = \text{ON}$, and 0 otherwise. Then the schedule must satisfy the following inequalities.

$$\sum_i z_{it} k_i \geq D_t \quad \text{for each time period } t \tag{1}$$

The second constraint is that maintenance must start and be completed within the prescribed window, and the single maintenance period must be continuous, uninterrupted, and of the desired length. The following conditions must hold true for each unit i.

$$\begin{array}{rll}
\text{(start)} & \text{if } t < e_i \text{ then } x_{it} \neq \text{MAINT} & (2) \\
\text{(end)} & \text{if } t \geq l_i + d_i \text{ then } x_{it} \neq \text{MAINT} & (3) \\
\text{(continuous)} & \text{if } x_{it_1} = \text{MAINT and } x_{it_2} = \text{MAINT and } t_1 < t_2 & \\
 & \text{then for all } t, t_1 < t < t_2,\ x_{it} = \text{MAINT} & (4) \\
\text{(length)} & \text{if } t_1 = \min_t(x_{it} = \text{MAINT}) \text{ and } t_2 = \max_t(x_{it} = \text{MAINT}) & \\
 & \text{then } t_2 - t_1 + 1 = d_i & (5) \\
\text{(existence)} & \exists t \text{ such that } x_{it} = \text{MAINT} & (6)
\end{array}$$

The third constraint is that no more than M units can be scheduled for maintenance simultaneously. Let $y_{it} = 1$ if $x_{it} = \text{MAINT}$, and 0 otherwise.

$$\sum_i y_{it} \leq M \quad \text{for each time period } t \tag{7}$$

The final constraint on a maintenance schedule is that incompatible pairs of units cannot be scheduled for simultaneous maintenance.

$$\text{if } (i_1, i_2) \in N \text{ and } x_{i_1 t} = \text{MAINT then } x_{i_2 t} \neq \text{MAINT} \qquad \text{for each time period } t \tag{8}$$

After meeting the above constraints, we want to find a schedule which minimizes the maintenance and operating costs during the planning period. Let $w_{it} = m_{it}$ if $x_{it} = $ MAINT, c_{it} if $x_{it} = $ ON, and 0 if $x_{it} = $ OFF.

$$\text{Minimize } \sum_i \sum_t w_{it} \tag{9}$$

Objective functions other than (9) can also be used. For example, it may be necessary to reschedule the projected maintenance midway through the planning period. In this case, a new schedule which is as close as possible to the previous schedule may be desired, even if such a schedule does not have a minimal cost.

3.3 Formalizing Maintenance Problems as CSPs

There are several ways to encode the maintenance scheduling problem in the constraint satisfaction framework. The formulation involves a tradeoff between the number of variables, the number of values per variable, and the constraints arity.

We defined the problem's output variables as variables in the CSP, and specified the problem's constraints in a relational manner in order to allow our general purpose CSP algorithms to be applied with minimal modification.

We encode maintenance scheduling problems as CSPs with $3 \times U \times W$ variables. The variables can be divided into a set of $U \times W$ visible variables, and two $U \times W$ size sets which we call hidden variables. Of course the distinction between visible and hidden variables is used for explanatory purposes only; the CSP solving program treats each variable in the same way. Each variable has two or three values in its domain. Both binary and higher arity constraints appear in the problem. The visible variables X_{it} correspond directly to the output parameters x_{it} of the problem definition, having the corresponding domain values {ON, OFF, MAINT}.

Because i ranges from 1 to U and t ranges from 1 to W, there are $U \times W$ visible variables. Each X_{it} has the domain {ON, OFF, MAINT}, corresponding exactly to the values of x_{it}.

The first set of hidden variables, Y_{it}, signifies the maintenance status of unit i during week t. The domain of each Y variable is {FIRST, SUBSEQUENT, NOT}. $Y_{it} =$ FIRST indicates that week t is the beginning of unit i's maintenance period. $Y_{it} =$ SUBSEQUENT indicates that unit i is scheduled for maintenance during week t and for at least one prior week. $Y_{it} =$ NOT, indicates no maintenance. Binary constraints between each X_{it} and Y_{it} are required to keep the two variables synchronized (we list the compatible value combinations):

X_{it}	Y_{it}
ON	NOT
OFF	NOT
MAINT	FIRST
MAINT	SUBSEQUENT

The second set of hidden variables, Z_{it}, are boolean variables having domains {NONE, FULL}, which indicate whether unit i is producing power output during week t. The binary constraints between each X_{it} and the corresponding Z_{it} variable are as follows, again listing the legal combinations:

X_{it}	Z_{it}
ON	FULL
OFF	NONE
MAINT	NONE

The hidden variables triple the size of the CSP. The reasons for creating them will become clear as we now discuss how the constraints are implemented.

Constraint (1) – weekly power demand

Each demand constraint involves the U visible variables that relate to a particular week. The basic idea is to enforce a U-ary constraint between these variables which guarantees that enough of the variables will be ON to meet the power demand for the week. This constraint can be implemented as a table of either compatible or incompatible combinations, or as a procedure which takes as input the U variables and returns TRUE or FALSE. Our implementation uses a table of incompatible combinations. For example, suppose there are four generating units, with output capacities k_1=100, k_2=200, k_3=300, k_4=400. For week 5, the demand D_5=800. The following 4-ary constraint among variables $(Z_{1,5}, Z_{2,5}, Z_{3,5}, Z_{4,5})$ is created (incompatible tuples are listed).

$Z_{1,5}$	$Z_{2,5}$	$Z_{3,5}$	$Z_{4,5}$	comment (output level)
NONE	NONE	NONE	NONE	0
NONE	NONE	NONE	FULL	400
NONE	NONE	FULL	NONE	300
NONE	NONE	FULL	FULL	700
NONE	FULL	NONE	NONE	200
NONE	FULL	NONE	FULL	600
NONE	FULL	FULL	NONE	500
FULL	NONE	NONE	NONE	100
FULL	NONE	NONE	FULL	500
FULL	NONE	FULL	NONE	400
FULL	FULL	NONE	NONE	300
FULL	FULL	NONE	FULL	700
FULL	FULL	FULL	NONE	600

Because the domain size of the Z variables is 2, a U-ary constraint can have as many as $2^U - 1$ tuples. If this constraint were imposed on the X variables directly, which have domains of size 3, there would be 79 tuples ($3^4 - 5$) instead of 13 ($2^4 - 3$). This is one reason for creating the hidden Z variables: to reduce the size of the demand constraint.

A relation such as that in the above table may be *projected* onto a subset of its variables, by listing the combinations of values which are restricted to this subset. The relation's projection onto $(Z_{1,5}, Z_{2,5})$, for example, is

$Z_{1,5}$	$Z_{2,5}$
NONE	NONE
NONE	FULL
FULL	NONE
FULL	FULL

Tuples in the new, projected relation which appear with all possible combinations of the remaining variables in the original relation may be recorded as smaller constraints. That is, the binary constraint over the pair $(Z_{1,5}, Z_{2,5})$, allowing ($Z_{1,5}$=NONE, $Z_{2,5}$= NONE) while the remaining tuples are not allowed, is implied by the 4-ary constraint. It is clearly desirable to recognize these smaller constraints prior to search, and our system does so. In effect, the system notices that if $Z_{1,5}$ is NONE and $Z_{2,5}$ is NONE, then the demand constraint for week 5 cannot be met, whatever the status of the other units.

Constraints (2) and (3) – earliest and latest maintenance start date

These constraints are easily implemented by removing the value FIRST from the domains of the appropriate Y variables. The removal of a domain value is often referred to as imposing a *unary* constraint.

Constraint (4) – continuous maintenance period

To encode this domain constraint in our formalism, we enforce three conditions using binary relational constraints over the Y's:

1. There is only one first week of maintenance.
2. Week 1 cannot be a subsequent week of maintenance.
3. Every subsequent week of maintenance must be preceded by a first week of maintenance or a subsequent week of maintenance.

Each of these conditions can be enforced by unary or binary constraints on the Y variables. To enforce condition 1, for every unit i and pair of weeks t_1 and $t_2, t_1 \neq t_2$, we add the following binary constraint to the CSP (disallowed tuple listed):

Y_{it_1}	Y_{it_2}
FIRST	FIRST

Condition 2 is enforced by a unary constraint removing SUBSEQUENT from the domain of each Y_{i1} variable. Condition 3 is enforced by the following constraint for all $t > 1$ (disallowed tuple listed):

Y_{it-1}	Y_{it}
NOT	SUBSEQUENT

Constraint (5) – length of maintenance period

A maintenance period of the correct length cannot be too short or too long. If unit i's maintenance length d_i=1, then too short is not possible (constraint (6) prevents non-existent maintenance periods); otherwise, for each unit i, each time period t, and every $t_1, t < t_1 < t + d_i$, the following binary constraint prevent a short maintenance period (disallowed tuple listed):

Y_{it}	Y_{it_1}
FIRST	NOT

To ensure that too many weeks of maintenance are not scheduled, it is only necessary to prohibit a subsequent maintenance week in the first week that maintenance should have ended. This results in the following constraint for each i and t, letting $t_1 = t + d_i$ (disallowed tuple listed):

Y_{it}	Y_{it_1}
FIRST	SUBSEQUENT

Constraint (6) – existence of maintenance period

This requirement is enforced by a high arity constraint among the Y variables for each unit. Only the weeks between the earlist start week and the latest start week need be involved. At least one $Y_{it}, e_i \leq t \leq l_i$, must have the value START. It is simpler to prevent them from all having the value NOT, and let constraints (4) and (5) ensure that a proper maintenance period is established. Thus the $(l_i - e_i + 1)$-arity constraint for each unit i is (disallowed tuple listed):

Y_{il_i}	...	Y_{ie_i}
NOT	NOT	NOT

Constraint (7) – no more than M units maintained at once

If M units are scheduled for maintenance in a particular week, constraints must prevent the scheduling of an additional unit for maintenance during that week. Thus the CSP must have $(M + 1)$-ary constraints among the X variables which prevent any $M + 1$ from having the value of MAINT in any given week. There will be $\binom{U}{M+1}$ of these constraints for each of the W weeks. They will have the form (disallowed tuple listed):

$X_{i_1 t}$	...	$X_{i_M t}$
MAINT	MAINT	MAINT

We see that this requires an exponential number in M of no-goods. If M is big, it may be beneficial to leave this constraint in a procedural (rather than relational) form.

Constraint (8) – incompatible pairs of units

The requirement that certain units not be scheduled for overlapping maintenance is easily encoded in binary constraints. For every week t, and for every pair of units $(i_1, i_2) \in N$, the following binary constraint is created (incompatible pair listed):

$X_{i_1 t}$	$Y_{i_2 t}$
MAINT	MAINT

Objective function (9) – minimize cost

To achieve optimization within the context of our constraint framework, we create a constraint that specifies that the total cost must be less than or equal to a set amount. In order to reduce the arity of the cost constraint, we introduce a simplification to the problem: we optimize cost by week instead of over the entire planning period. Therefore, the algorithm achieves an optimal solution to a more restricted cost function which may not optimize the original one. Further study is need to assess the trade-offs between constraint size and global optimality.

We implemented the cost constraint as a procedure in our CSP solving program. This procedure is called after each X type variable is instantiated. The input to the procedure is the week, t, of the variable, and the procedure returns TRUE if the total cost corresponding to week t variables assigned ON or MAINT is less than or equal to C_t, a new problem parameter (not referenced in Fig. 13) which specifies the maximum cost allowed in period t. This is the only constraint in our formulation that is implemented procedurally.

4 Solution Procedure

In this section we show a novel use of CSP algorithms to solve an optimization problem and in particular how constraint learning can be exploited for optimization.

4.1 Finding an Optimal CSP

A constraint optimization problem is a CSP augmented with a cost function, defined as follows. Let $X = x_1, \ldots, x_n$, and let $f_1, \ldots, f_l$ be real-valued functions defined over subsets of variables $S_{i_1}, \ldots, S_{i_l}, S_{i_j} \subseteq X$, such that for $\bar{x} = (x_1, \ldots, x_n)$

$$C(\bar{x}) = \sum_{j=1}^{l} f_j(x_j).$$

The optimization task is to find $\bar{x}^O = (x_1, \ldots, x_n)$ satisfying all constraints, such that

$$C(\bar{x}^O) = \min_x C(x).$$

In general, we can find a solution with optimal cost by solving a series of CSPs. Each problem P^i is augmented with a constraint

$$\sum_{j=1}^{l} f_j(x_j) \leq C^i,$$

where $C^1 \geq C^2 \geq \ldots \geq C^j \geq \ldots$. If problem P^i has a solution and problem P^{i+1} does not, we know that the solution obtained for P^i is optimal.

The procedure as implemented is described in Fig. 14. Initially, a schedule is found with a very high cost-bound. For the maintenance scheduling problems, this is C_t, the maximum cost per week. The cost-bound is then gradually lowered, with a new schedule found each time. Eventually, the cost-bound is so low that no schedule exists which meets it, and the last schedule found is optimal, within the limit of the amount by which the cost-bound was lowered. A more sophisticated control algorithm, based on a binary search approach, can be envisioned. In the experiments reported below, we used the simple decrement only technique. Another enhancement would be to permit different cost-bounds for different weeks.

4.2 Optimization with Learning

To make the optimization process more efficient, we introduce the notion of a memory that exists between successive iterations of step 2 in Fig. 14 The idea is to use a learning algorithm, such as BJ+DVO+LRN, to solve the maintenance scheduling CSPs (MSCSPs), and the new constraints introduced by learning are retained for use in later iterations. We call this approach *iterative learning.*

Retaining a memory of constraints is safe because as the cost-bound is lower the constraints become tighter. Any solution to an MSCSP with a certain cost-bound is also a valid solution to the same problem with a higher cost-bound. If the the cost-bound were both lowered and raised, as suggested in the previous section with a binary search approach, then some learned constraints would have to be "forgotten" when the cost-bound was raised.

4.3 Problem Instance Generator

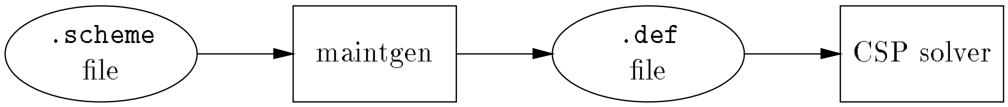
One of our goals is to be able to determine the efficacy of various CSP algorithms and heuristics when applied to Maintenance Scheduling CSPs. To perform an experimental average-case analysis, we need a source of many MSCSPs. We have therefore developed an MSCSP generator, which can create any number of problems that adhere to a set of input parameters.

Solution Procedure for Optimization
Input: A MSCSP with hard constraints, and an objective function.
Output: The lowest cost-bound for which a solution was found, and a solution with that cost-bound.

1. Set the cost-bound to a high value.
2. Until no solution can be found,
   (a) Add a constraint (or set of constraints) to the MSCP specifying that the value of the objective function is less than the cost-bound.
   (b) Solve the MSCSP using a constraint algorithm.
   (c) Decrement the cost-bound.
3. Return the last solution found, and the corresponding cost-bound.

Figure 14: The solution procedure for optimization.

A flowchart of the overall system is below:



A "scheme" file is an ACII file (with a name usually ending in ".scheme") that defines a class or generic type of MSCSP. Here is an example of a .scheme file:

The input to the generator is a file containing most of the basic parameters, either explicitly enumerated or a kernel for generating all the necessary parameters by interpolation or by some parameterized distribution. The generator generates as many problem instances as necessary using the input parameters and then the problem instance is solved by the various algorithms. The maintgen program generator reads in a file and creates one or more MSCSP instances which can be solved by the CSP solver. The maintgen program uses a random number generator seed and a number indicating how many individual problems should be generated. The parameters given to the generator specify the fundamental size parameters: the number of weeks W, the number of generating units U, and the number of units which can be maintained at one time M. Also, the demand for some number of weeks is specified. The demand for weeks that are not explicitly specified is computed by a linear interpolation between the surrounding specified weeks. The process is shown diagrammatically in Fig. 16. There is no randomness in the demand "curve" that is created based on a scheme file. Note that the weeks are numbered starting from 0, so that in this example the last of the 25 weeks is week #24.

The following line in the scheme file specifies the initial maximum cost per week,

Backjumping with learning

1. If all variables have been assigned values, then return this solution. Otherwise, select a variable using the dynamic variable ordering heuristic.
2. Find a compatible value for the current variable. If successful, go to 1. Otherwise, go to 3.
3. (Dead-end.) Find a subset of the variables with values assigned that are responsible for the dead-end. Add a new constraint which prohibits that combination from reoccuring. Select the latest variable in that subset to be the current variable, and go to 2.

Figure 15: Sketch of the BJ+DVO+LRN algorithm.

and the amount it is to be decremented after each successful search for a schedule.

The characteristics of the units, that is, their output capacities and required maintenance times, are not specified individually. Instead, these values are randomly selected from normal distributions whose means and standard deviations are specified. Currently the earliest and latest maintenance start dates are not specified in the scheme file, and are always set to 0 and $W - 1$ in the .def file.

Maintenance costs are specified by the standard deviation (1,000 in the example), and by a sample of weekly demands per unit. As with demand, values for weeks that are not given explicitly are interpolated. However, for maintenance costs there is a random element; the interpolated value is used as the mean, together with the specified standard deviation. Operating costs are defined in the lines following the maintenance costs, with exactly the same structure. The last piece of information is the number of incompatible pairs of units. The requested number of pairs is created randomly from a uniform distribution of the units.

Here is an example of an input file to the generator followed by a specification of the problem instance that was generated.

```
# lines beginning with # are comments
# first line has weeks, units, maximum simultaneous units
4 6 2
#
# next few lines have several points on the demand curve,
# given as week and demand.  Other weeks are interpolated.
0   700
3  1000
# end this list with EOL
EOL
```

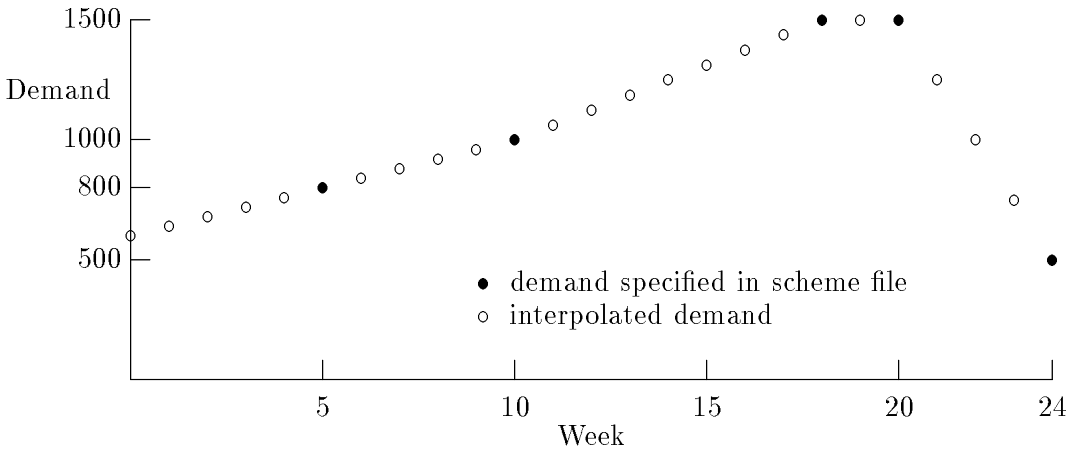


Figure 16: Weekly demand generated by the maintgen program when the following (week, demand) points are specified: (5, 800), (10, 10,000), (18, 1,500), (20, 1,500), (24, 500).

```
#
# next line has initial max cost per week, and decrement amount
60000 3000
#
# next line has average unit capacity and standard deviation
200 25
#
# next line has average unit maintenance time and std. dev.
2 1
#
# next line has standard deviation for maintenance costs
1000
#
# next few lines have some points on the maintenance cost curve,
# first number is week, then one column per unit
0 10000 10000 10000 10000 10000 10000
3 13000 16000 19000 10000  7000 10000
#
# next number is standard deviation for operating costs
2000
# next few lines have some points on the operating cost curve,
# first number is week, then one column per unit
0 5000 5000 5000 5000 5000 5000
# the next line specifies the number of incompatible pairs
2
```

```
# and that's it!
```

Below is a corresponding generated problem instance.

```
# comments begin with #
# first line has weeks W, units U, max-simultaneous M
4 6 2
# demand, one line per week
700
800
900
1000
# next few lines has maximum cost per week.
# Cost must be <= max.
60000 3000
# one line per unit:
# capacity  maint length  earliest maint start  latest maint start date
194 1 0 3
171 3 0 3
209 1 0 3
166 1 0 3
219 2 0 3
217 2 0 3
# maintenance costs, one line per week, one column per unit
11085 10034 9374 8945 10858 10045
11056 11988 13670 10465 9301 10625
12745 14625 15422 10422 8099 7629
12534 15394 21098 9841 6748 9364
# operating costs, one line per week, one column per unit
4284 6857 3847 5050 5145 4998
5987 7352 1967 4635 6152 4635
3746 6475 5151 3988 8172 4131
6152 3436 5475 5600 4366 6070
# incompatible pairs of units (numbering starts from 0)
1 3
2 3
EOL
# and that's it!
```

The output problem instance is in a format which is recognized by our CSP solver.

5 Experimental Results

In this section we describe in detail an experiment with a single maintenance scheduling problem, and then summarize the results of several experiments based on sets of 100 randomly generated problem instances.

5.1 A Single Problem

To demonstrate the effectiveness of the constraint processing framework, we will show how it operates on a small test example, drawn from [1, 28]. The problem has five units and a 12 week time horizon. Table 1 shows the characteristics of the units. Table 2 shows the load demand for each week. The operating and maintenance costs are specified in Tables 3 and 4. There are two maintenance crews, and units 1, 2 and 3 are to be maintained by a single crew.

Because there are five units and 12 time periods, this problem has 60 visible variables, each indicating whether a unit is being maintained, or if not what its utilization level for the week is. To solve this example problem we used BT+DVO. This is a relatively simple and unsophisticated approach; it does not take advantage of any special structure of the problem.

Table 1: Unit Operating Characteristics

		.....Maintenance.....	
Unit	Capacity	Window	Duration
1	150	1—11	2
2	150	1—11	2
3	130	1—11	2
4	90	1—11	2
5	50	1—11	2

Optimizing the schedule to minimize cost is an iterative cost-bounded procedure. We start with an unoptimized schedule which meets extremely loose cost bounds. The cost bounds specify the maximum allowable cost (summing operating and maintenance costs) per week. The cost bounds are modified iteratively using a schedule (for instance, reducing the cost in fixed amounts or binary search) until we reach an optimal or near-optimal solution.

For the problem under consideration, we first generated a schedule with no cost bounds; the resulting unoptimized schedule had a total cost of $179,673, and generated much more power each week than the minimum demand required (as specified in Table 2). In fact, in this initial schedule every unit was set to be operating at full power when it was not being maintained. Finding this schedule took less than one second of processing time. We then reduced the weekly cost bounds to 90% of the cost for each week in this unoptimized schedule. This forced the algorithm to reduce the power for some units, or to move maintenance times to weeks when the maintenance cost was less. We repeated 8 times this cycle of generating a schedule, noting the resulting costs, and then creating a new schedule with lower costs. Eventually no lower-cost schedule could be found. The optimum schedule we generated is shown in Table 5. The total computer processing time was about an hour.

Optimizing the schedule to minimize cost is an iterative cost-bounded procedure. We start with an unoptimized schedule which meets extremely loose cost bounds. The cost bounds specify the maximum allowable cost (summing operating and maintenance costs) per week. The cost bounds are modified iteratively using a schedule (for instance, reducing the cost in fixed amounts or binary search) until we reach an optimal or near-optimal solution.

For the problem under consideration, we first generated a schedule with no cost bounds; the resulting unoptimized schedule had a total cost of \$179,673, and generated much more power each week than the minimum demand required (as specified in Table 2). In fact, in this initial schedule every unit was set to be operating at full power when it was not being maintained. Finding this schedule took less than one second of processing time. We then reduced the weekly cost bounds to 90% of the
cost for each week in this unoptimized schedule. This forced the algorithm to reduce the power for some units, or to move maintenance times to weeks when the maintenance cost was less. We repeated 8 times this cycle of generating a schedule, noting the resulting costs, and then creating a new schedule with lower costs. Eventually no lower-cost schedule could be found. The optimum schedule we generated is shown in Table 5. The total computer processing time was about an hour.

Table 2: Total Energy Demand for Each Week

Week	Demand
1	36170
2	38398
3	36170
4	38398
5	36855
6	37707
7	36427
8	37250
9	38379
10	36170
11	36248
12	36855

Table 3: Weekly Operating Cost (/MWh)

Week	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5
1	8.2	9.3	10.6	11.9	12.0
2	8.0	9.5	10.7	11.4	12.5
3	8.8	9.7	10.7	11.1	12.0
4	8.0	9.6	10.5	11.2	12.3
5	8.7	9.8	10.6	11.3	12.3
6	8.6	9.7	10.1	11.2	12.1
7	8.7	9.0	10.3	11.2	12.4
8	8.5	9.3	10.2	11.1	12.4
9	8.8	9.4	10.5	11.6	12.9
10	8.7	9.6	10.8	11.8	12.8
11	8.0	9.9	10.0	11.3	12.9
12	8.0	9.9	10.4	11.4	12.7

Table 4: Weekly Maintenance Cost

Week	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5
1	8,700	9,100	9,300	9,350	9,550
2	8,700	9,100	9,300	9,350	9,550
3	8,700	9,100	9,300	9,350	9,550
4	8,700	9,100	9,300	9,350	9,550
5	8,700	9,100	9,300	9,350	9,550
6	8,700	9,100	9,300	9,350	9,550
7	8,900	9,300	9,600	9,650	9,850
8	8,900	9,300	9,600	9,650	9,850
9	8,900	9,300	9,600	9,650	9,850
10	8,700	9,100	9,300	9,350	9,550
11	8,700	9,100	9,300	9,350	9,550
12	8,700	9,100	9,300	9,350	9,550

The schedule found by our scheme is shown in Tables 5(a) and 5(b). Because the operating cost in $/MWh for Unit 1 are always lower than the operating cost for any other unit, that unit was always selected to be running at full power when it was not undergoing maintenance. Unit 2, the second most efficient unit, was always selected next, running at full or half power, as controlled by the demand for the week. Note that since the weekly demand for Week 2, 38,398 MWh, is slightly higher than the demand during Week 1, 36,170 MWh, it is necessary during Week 2 to schedule Unit 2 to run at full power and not half power, as is sufficient for Week 1. Because the demand is not high relative to the generating capacity of the units, it is not necessary to schedule Units 3, 4 and 5 to run except when Units 1 or 2 are being maintained.

It is also interesting to note that the schedule avoids any maintenance during weeks 7, 8 and 9. According to the schedule of maintenance costs in Table 4, the costs are higher in these weeks.

Table 5(a): Optimum Schedule Computed by Constraint Processing
First 6 weeks

	Week					
Unit	1	2	3	4	5	6
Status per Unit per Week:						
1	F	F	F	F	M	M
2	H	F	H	F	F	F
3	M	M	O	O	F	F
4	M	M	O	O	O	O
5	O	O	O	O	O	O
Cost per Unit per Week:						
1	1,377	1,344	1,478	1,344	8,700	8,700
2	781	1,596	814	1,612	1,646	1,629
3	9,300	9,300	0	0	1,780	1,696
4	9,350	9,350	0	0	0	0
5	0	0	0	0	0	0
Total	20,808	21,590	2,292	2,956	12,126	12,025
Total for all 12 weeks: 143,662						

Table 5 (b): Optimum Schedule Computed by Constraint Processing
Last 6 Weeks

Unit	Week 7	8	9	10	11	12
Status per Unit per Week:						
1	F	F	F	F	F	F
2	H	H	F	H	M	M
3	O	O	O	O	F	F
4	O	O	O	O	O	O
5	O	O	O	M	M	O
Cost per Unit per Week:						
1	1,461	1,428	1,478	1,461	1,344	1,344
2	756	781	1,579	806	9,100	9,100
3	0	0	0	0	1,680	1,747
4	0	0	0	0	0	0
5	0	0	0	9,550	9,550	0
Total	2,217	2,209	3,057	11,817	21,674	12,191
Total for all 12 weeks: 143,662						

The schedule generated by our system is similar to, and compares favorably with, the schedule reported in [28]. A direct comparison is not entirely possible, since we have used a slightly different objective function and made some different assumptions about costs. Nevertheless, if the schedule in [28] were evaluated according to our objective function, the cost would be \$151,337, compared to our cost of \$143,662.

5.2 Random Problems

We now present the results of experiments with two sets of 100 MSCSPs each. The smaller problems had 15 units and 13 time periods, resulting in 585 variables. The larger problems had 20 units and 20 time periods, resulting in 1200 variables. We conducted two experiments with each set of 100 problems. In the first we used the algorithms BJ+DVO and iterative learning based on BJ+DVO+LRN to solve the optimization task. In the second we compared the performance of all five algorithms described in section 4, using a fixed cost-bound that is close to the lowest feasible one.

Determining whether a potential value for a variable violates a constraint with another variable is called a *consistency check*. Because consistency checking is performed so frequently, it constitutes a major part of the work performed by all of these algorithms. Hence a count of the number of consistency checks is a common measure of the overall work of an algorithm.

In the first experiment on random problems, we tried to find an optimal schedule for each MSCSP in the smaller and larger sets, using BJ+DVO and iterative learning. Iterative learning used 6th-order BJ+DVO+LRN. The results are shown in Fig. 17 and Fig. 18.

For the 100 smaller problems, the cost-bound was set initially at 110,000 per week, and then reduced by 5,000 for each iteration. All 100 MSCSPs had schedules at cost-bound 85,000 and above. Only 38 had schedules within the 80,000 bound; at 75,000 only four problems were solvable. On the set of 100 larger MSCSPs, the cost-bound started at 150,000 per week and was reduced by 5,000. Schedules were found for all instances at cost-bound 120,000 and above. 97 instances had schedules at cost- bound 115,000 and 110,000; 11 at cost-bound 105,000; and two at cost-bound 100,000 and 95,000.

Iterative learning performed better, on average, than BJ+DVO on these random maintenance scheduling problems. For instance, on the set of smaller problems, after finding a schedule with cost-bound 95,000 the average number of learned constraints was 214. Tightening the cost-bound to 90,000 resulted in over twice as much CPU time needed for BJ+DVO (54.01 CPU seconds compared to 23.28), but only a 71% increase for iterative learning (29.41 compared to 17.20). Iterative learning was less effective on the larger MSCSPs. Although it required less CPU time on average, the improvement over BJ+DVO was much less than on the smaller problems.

The second experiment with random problems utilized the same sets of 100 smaller MSCSP instances and 100 larger instances, but we did not try to find an optimal schedule. For the smaller problems we set the cost-bound at 85,000 and for the larger problems we set the cost-bound at 120,000. Each bound was the lowest level at which schedules could be found for all problems. We used the five algorithms described earlier to find a schedule for each problem. The results are summarized in Fig. 19.

Among the five algorithms, BJ+DVO performed least well on the smaller problems and best on the larger problems, when average CPU time is the criterion. BT+DVO+IAC was the best performer on the smaller problems and the worst on the larger problems. This reversal in effectiveness may be related to the increased size of the higher arity constraints on the larger problems. The high arity constraints, such as those pertaining to the cost-bound, the weekly power demand, and the existence of a maintenance period, become looser as the number of units and number of weeks increase. Earlier results [9] have indicated that more look-ahead is effective on problems with tight constraints, and detrimental on problems with loose constraints. Nevertheless backjumping remains an effective technique on the larger problems. Further experiments are required to determine how the relative efficacy of different algorithms is influenced by factors such as the size of the problem (number of weeks and units) and characteristics such as the homogeneity of the units.

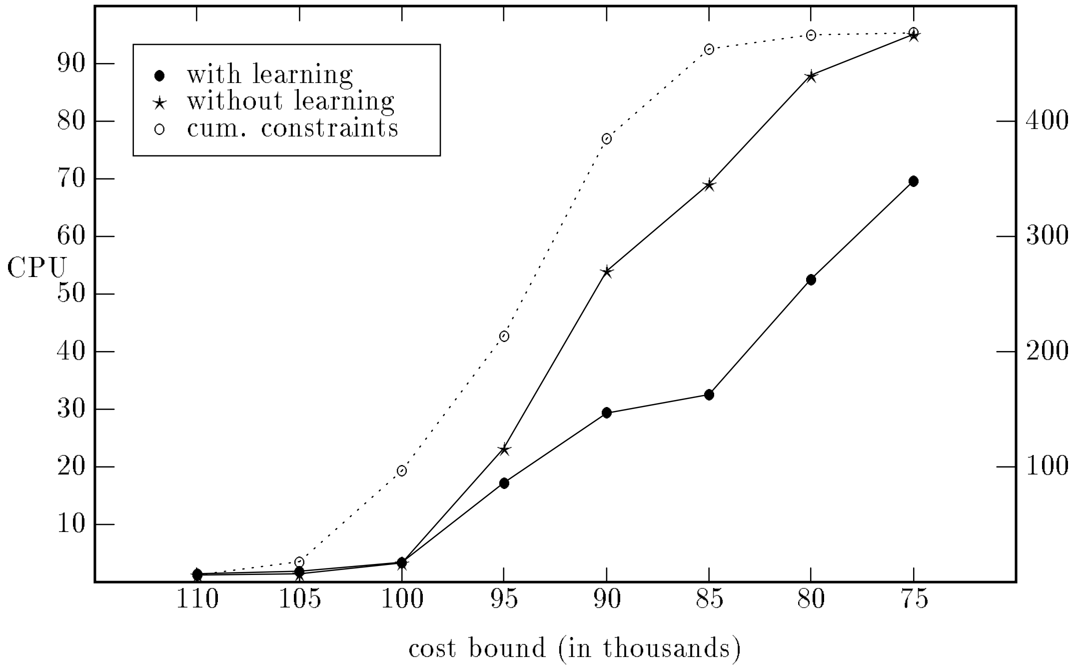


Figure 17: Average CPU seconds on 100 small problems (15 units, 13 weeks) to find a schedule meeting the cost-bound on the y-axis, using BJ+DVO with learning ($\bullet$) and without learning ($\star$). Cumulative number of constraints learned corresponds to right-hand scale.

6 Conclusions

The research presented in this report has demonstrated that maintenance scheduling problems can be successfully addressed using the constraint satisfaction framework. We have shown that a variety of general-purpose algorithms and heuristics for CSPs work well on these problems.

The constraint satisfaction problems derived from the maintenance scheduling needs of the electric power industry are an interesting testbed for CSP algorithms. The problems have a mixture of tight binary constraints, such as those that bind the X and Y variables together, and loose high arity constraints, such as those that ensure that at least one maintenance period is scheduled for each unit. The most promising algorithm for these problems is iterative learning. Further studies on larger maintenance scheduling CSPs is required to determine whether one algorithm dominates the others as problem size increases.

A challenging problem that is difficult to formalize is to find the best way to encode the requirements of a problem such as maintenance scheduling into constraints of a CSP. In section 3 we discussed some of the trade-offs involved in, for example,

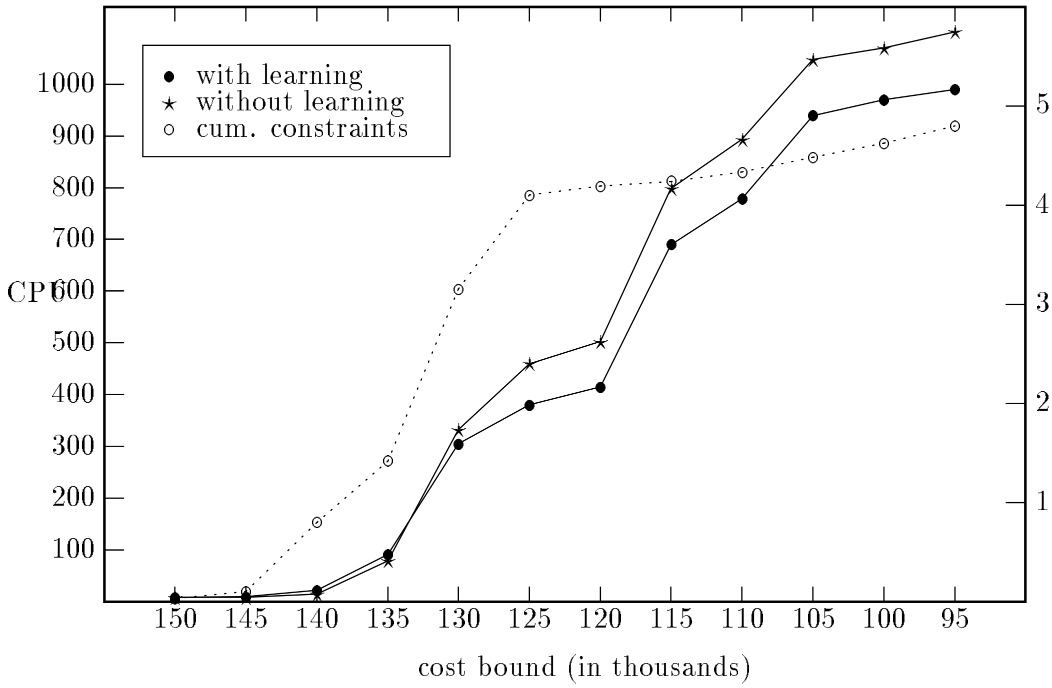


Figure 18: Average CPU seconds on 100 large problems (20 units, 20 weeks) to find a schedule meeting the cost-bound on the y-axis, using BJ+DVO with learning ($\bullet$) and without learning ($\star$). Cumulative number of constraints learned corresponds to right-hand scale.

adding "hidden" variables in return for a smaller number of tuples in high arity constraints. This is an important area for future research that has the potential of greatly impacting the applicability of the constraint satisfaction framework to problems from science and industry.

References

[1] T. M. Al-Khamis, S. Vemuri, L. Lemonidis, and J. Yellen. Unit maintenance scheduling with fuel constraints. *IEEE Trans. on Power Systems*, 7(2):933–939, 1992.

[2] Roberto Bayardo and Daniel Mirankar. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304, 1996.

[3] James R. Bitner and Edward M. Reingold. Backtrack Programming Techniques. *Communications of the ACM*, 18(11):651–656, 1975.

Algorithm	Average CC	Nodes	CPU
100 smaller problems:			
BT+DVO+IAC	315,988	3,761	51.65
BJ+DVO	619,122	8,981	70.07
BJ+DVO+LVO	384,263	5,219	54.48
BJ+DVO+LRN	671,756	8,078	67.51
BJ+DVO+LRN+LVO	476,901	5,085	57.45
100 larger problems:			
BT+DVO+IAC	7,673,173	32,105	694.02
BJ+DVO	2,619,766	28,540	460.42
BJ+DVO+LVO	6,987,091	26,650	469.65
BJ+DVO+LRN	5,892,065	27,342	521.89
BJ+DVO+LRN+LVO	6,811,663	26,402	475.12

Figure 19: Statistics of five algorithms on MSCSPs.

[4] Rina Dechter. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[5] Rina Dechter. Constraint networks. In *Encyclopedia of Artificial Intelligence*, pages 276–285. John Wiley & Sons, 2nd edition, 1992.

[6] J. F. Dopazo and H. M. Merrill. Optimal Generator Maintenance Scheduling using Integer Programming. *IEEE Trans. on Power Apparatus and Systems*, PAS-94(5):1537–1545, 1975.

[7] G. T. Egan. An Experimental Method of Determination of Optimal Maintenance Schedules in Power Systems Using the Branch-and-Bound Technique. *IEEE Trans. SMC*, SMC-6(8), 1976.

[8] E. C. Freuder. A sufficient condition for backtrack-free search. *JACM*, 21(11):958–965, 1982.

[9] Daniel Frost. *Algorithms and Heuristics for Constraint Satisfaction Problems.* PhD thesis, University of California, Irvine, CA 92697-3425, 1997.

[10] Daniel Frost and Rina Dechter. Dead-end driven learning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 294–300, 1994.

[11] Daniel Frost and Rina Dechter. In search of the best constraint satisfaction search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 301–306, 1994.

[12] Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 572–578, 1995.

[13] John G. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 1979.

[14] R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[15] Henry Kautz and Bart Selman. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 1194–1201, 1996.

[16] Hyunchul Kin, Yasuhiro Hayashi, and Koichi Nara. An Algorithm for Thermal Unit Maintenance Scheduling Through Combined Use of GA SA and TS. *IEEE Trans. on Power Systems*, 12(1):329–335, 1996.

[17] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[18] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[19] Alan K. Mackworth. The logic of constraint satisfaction. *Artificial Intelligence*, 58:3–20, 1992.

[20] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[21] Bernard A. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In L. Kanal and V. Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer, 1988.

[22] Bernard A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.

[23] Patrick Prosser. BM + BJ = BMJ. In *Proceedings of the Ninth Conference on Artificial Intelligence for Applications*, pages 257–262, 1993.

[24] Patrick Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[25] Paul Walton Purdom. Search Rearrangement Backtracking and Polynomial Average Time. *Artificial Intelligence*, 21:117–133, 1983.

[26] Daniel Sabin and Eugene C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. In *Principles and Practice of Constraint Programming*, pages 10–20, 1994.

[27] David Waltz. Understanding Line Drawings of Scenes with Shadows. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, pages 19–91. McGraw-Hill, 1975.

[28] J. Yellen, T. M. Al-Khamis, S. Vemuri, and L. Lemonidis. A decomposition approach to unit maintenance scheduling. *IEEE Trans. on Power Systems*, 7(2):726–731, 1992.

[29] Ramin Zabih and David McAllester. A Rearrangement Search Strategy for Determining Propositional Satisfiability. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 155–160, 1988.

[30] H. H. Zurm and V. H. Quintana. Generator Maintenance Scheduling Via Successive Approximation Dynamic Programming. *IEEE Trans. on Power Apparatus and Systems*, PAS-94(2), 1975.