Maintenance scheduling problems as benchmarks for constraint algorithms

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

Abstract

The paper focuses on evaluating constraint satisfaction search algorithms on application based random problem instances. The application we use is a well-studied problem in the electric power industry: optimally scheduling preventive maintenance of power generating units within a power plant. We show how these scheduling problems can be cast as constraint satisfaction problems and used to define the structure of randomly generated non-binary CSPs. The random problem instances are then used to evaluate several previously studied algorithms. The paper also demonstrates how constraint satisfaction can be used for optimization tasks. To find an optimal maintenance schedule, a series of CSPs are solved with successively tighter cost-bound constraints. We introduce and experiment with an "iterative learning" algorithm which records additional constraints uncovered during search. The constraints recorded during the solution of one instance with a certain cost-bound are used again on subsequent instances with tighter cost-bounds. Our results show that on a class of randomly generated maintenance scheduling problems, iterative learning reduces the time required to find a good schedule.

1 Introduction

The last three decades have seen the development of many algorithms and heuristics for solving constraint satisfaction problems (CSPs). Determining which algorithms are superior to others remains difficult. Theoretical analysis provides worst-case guarantees which often do not reflect average performance. For instance, a backtracking-based algorithm that incorporates features such as variable ordering heuristics will

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

often in practice have substantially better performance than a simpler algorithm without this feature [12, 8], and yet the two share the same worst-case complexity. Similarly, one algorithm may be better than another on problems with a certain characteristic, and worse on another category of problem. Ideally, we would be able to identify this characteristic in advance and use it to guide our choice of algorithm.

Algorithms and heuristics have often been compared by observing their performance on benchmark problems, such as the 8-queens puzzle, or on suites of random instances generated from a simple, uniform distribution. The advantage of using a benchmark problem is that if it is an interesting problem (to someone), then information about which algorithm works well on it is also interesting. The drawback is that if algorithm A beats algorithm B on a single benchmark problem, it is hard to extrapolate from this fact. An advantage of using random problems is that there are many of them, and researchers can design carefully controlled experiments and report averages and other statistics. A drawback of random problems is that they may not reflect any real life situations.

In this paper we demonstrate another method for comparing CSP search algorithms, by applying them to random problems that have been generated with a particular structure. The structure, in the present case, was derived from a well-studied problem of the electric power industry: optimally scheduling preventive maintenance of power generating units within an electric power plant. Our approach was to define a formal model which captures most of the interesting characteristics of maintenance scheduling, and then to cast the model as a constraint satisfaction problem. A program was written to create specific problem instances which adhered to this model, had certain parameters (such as number of time periods to be scheduled) fixed in advance, and used pseudo-random numbers to determine other characteristics of the instances (such as the power output of individual generating units). With this effectively unlimited supply of structured random CSP instances, we have a new technique for evaluating the performance of CSP algorithms in the context of a specific problem domain.

Within this general methodology of evaluating algorithms on realistic problems, while obtaining statistically significant results, we present a case-study. We chose for our comparison five backtracking based constraint techniques that have been developed in recent years: backtracking with dynamic variable ordering and integrated arc-consistency (BT+DVO+IAC), conflict-directed backjumping with dynamic variable ordering (BJ+DVO), BJ+DVO with constraint learning (BJ+DVO+LRN), BJ+DVO with look-ahead value ordering (BJ+DVO+LVO), and a combination of the last two called BJ+DVO+LRN+LVO. Each algorithm has shown itself to be "best of breed," in that it is superior to other similar algorithms [10, 9, 11, 4, 15]. However, no clear dominance relationship between these algorithms has been revealed in the past. We restricted our attention to comparisons of complete, backtracking based CSP algorithms; this study does not extend either to algorithms based on stochastic local search (such as GSAT or WSAT), or to algorithms designed specifically for optimization problems (such as branch and bound). An experimental comparison of backtracking based search algorithms on many instances with a real-life structure has not, to our knowledge, been carried out previously. We found that two algorithms stood out. BJ+DVO, which among the algorithms we experimented with does the least work per node in the search tree, was the fastest on large problems and the worst on small problems. BT+DVO+IAC, which performs extensive look-ahead processing at each node, was best on small problems and worst on large problems.

In addition to a general methodology of using application based random problems and a specific case-study, the third contribution of this paper is a demonstration of optimization using constraint algorithms. The constraint framework consists entirely of hard constraints, which must be satisfied for a solution to be valid. The maintenance scheduling problem is fundamentally an optimization problem, which has both hard constraints and a cost function to be minimized. We converted the optimization problem to a constraint satisfaction problem by fixing a maximum value for the cost function and then searched for a schedule at or below that cost. When we wanted to find an optimal or near-optimal schedule, we transformed the problem into one of searching a series of CSPs with ever-tightening constraints until an optimal schedule was found by proving that no schedule exists with a tighter cost-bound. This approach inspired the development of a new variant of CSP learning, called "iterative learning," that is particularly suited towards optimization problems.

In section 2 we describe the maintenance scheduling problem in detail. Section 3 provides a brief description of the constraint satisfaction problem framework and describes a method for casting maintenance scheduling problems as CSPs. The algorithms we experimented with, including the new iterative learning algorithm, are described in section 4. Section 5 covers the program that generated random maintenance scheduling type problem instances. In section 6, we report the results of several experiments. Section 7 describes related work, and in section 8 we summarize our results and our conclusions.

2 Maintenance Scheduling Problems

The problem of scheduling off-line preventive maintenance of power generating units is of substantial 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

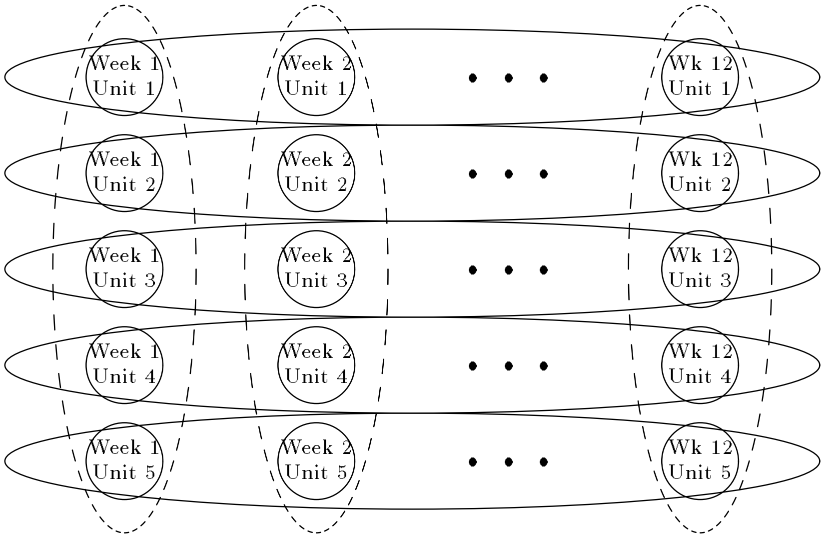

Figure 1: A diagrammatic representation of a maintenance scheduling constraint satisfaction problem. Each circle stands for a variable representing the status of one unit in one week. The dashed vertical ovals indicate constraints between all of the units in one week: meeting the minimum power demand and optimizing the cost per week. The horizontal ovals represent constraints on one unit over the entire period: scheduling an adequate period for maintenance.

considers shorter term scheduling, up to a period of one or two 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, 16].

2.1 Parameters

The maintenance scheduling problem can be visualized as a rectangular matrix (Fig. 1). 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. A specific maintenance scheduling problem, in our formulation, is defined by a set of parameters, which are listed in Fig. 2. 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 available to carry out maintenance; therefore, the parameter M specifies the maximum number

	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 2: Parameters which define a specific maintenance scheduling problem.

of units which can be undergoing maintenance at any one time.

In this paragraph and elsewhere in the paper we adopt the convention of quantifying the subscript i over the units, $0 \leq i \leq U-1$, and the subscript t over the weeks, $0 \leq t \leq W-1$. Several parameters specify the characteristics of the power generating units. The costs involved in preventive 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 $N = \{(j_1, j_2), \ldots, (j_{n-1}, j_n)\}$.

The final input parameter, D_t, is the predicted power demand on the plant in each week t. The variables 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}.

2.2 Schedule Requirements

A valid maintenance schedule must meet the following 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 scheduled to be on must be not less than the predicted demand, for each week. Let $z_{it} = 1$ if $x_{it} =$ ON, and 0 otherwise. Then the schedule must satisfy the following inequalities:

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

Additional requirements are 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.

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

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

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

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

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

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 $x_{it} = \textsc{maint}$, c_{it} if $x_{it} = \textsc{on}$, and 0 if $x_{it} = \textsc{off}$.

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

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 Formalizing Maintenance Problems as CSPs

The constraint satisfaction problem framework is a well-studied model that formalizes many real-life problems [5]. A constraint satisfaction problem consists of a finite number of variables. Associated with each variable is a finite domain of values. The problem also has a set of constraints. Each constraint pertains to a subset of the variables and specifies which combinations of assignments of values to variables are permitted. The "arity" of a constraint is the number of variables to which it refers. A solution to a CSP assigns to each variable a value from its domain, such that no constraint is violated. 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.

Formalizing a maintenance scheduling problem as a constraint satisfaction problem entails deciding on the variables, the domains, and the constraints which will represent the requirements of the problem, as described in the previous section. The goal, of course, is to develop a scheme that is conducive to finding a solution – a schedule – speedily. The formulation must trade off between the number of variables, the number of values per variable, and the arities of the constraints. In general, problems having fewer variables, smaller value domains, and constraints of smaller arity will tend to be easier to solve. Since real-life problems are often large, these three conditions cannot be met simultaneously, and compromises must be made to achieve a satisfactory representation as a constraint satisfaction problem. How to best make these compromises and how the choices made affect the performance of an algorithm are important directions for further research. In this section we describe the representation used in the case-study, without making any claim that it is the best possible. We specify most of the constraints in a relational manner in order to allow our implementations of 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. (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, and have the domain {ON, OFF, MAINT}.

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 during week t. 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 variables with the domain {NONE, FULL}. They indicate whether unit i is producing power output during week t. The obvious binary constraints are defined between each X_{it} and the corresponding Z_{it}.

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 compatible value combinations, or as a table of incompatible combinations or tuples, 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 is $D_5 = 800$. 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 for an implementation that stores constraints as a list of incompatible value combinations.

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.

Constraint (4) – continuous maintenance period

To encode this domain constraint in our formalism, we enforce three conditions:

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 by a subsequent week of maintenance.

Each of these conditions can be enforced by unary or binary constraints on the Y variables.

Constraint (5) – length of maintenance period

A maintenance period of the correct length should not be too short or too long. For each unit i, each time period t, and every $\tau, t < \tau < t + d_i$, the following binary constraint prevents a short maintenance period (disallowed tuple listed):

Y_{it}	$Y_{i\tau}$
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_1t}	...	X_{i_Mt}
MAINT	MAINT	MAINT

The number of no-goods will be exponential in M. If M is big, it may be beneficial to express this constraint in a procedural (rather than relational) form. However, as stated earlier we preferred a relational representation whenever possible.

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_1t}	Y_{i_2t}
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 minimize the maximum cost per week, instead of minimizing the sum of costs over all weeks. Clearly, an optimal solution to the more restricted cost function may not optimize the original function. Our motivation for using weekly cost instead of total cost is our knowledge that CSP algorithms can more effectively exploit more local constraints.

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 CW, a new problem parameter (not referenced in Fig. 2) which specifies the maximum cost allowed in any period. This is the only constraint in our formulation that is implemented procedurally.

4 Constraint Algorithms

We investigated the efficacy of various constraint satisfaction algorithms on maintenance scheduling CSPs. This section describes several algorithms which have been reported earlier and which we used in our experiments. We then describe iterative learning, which takes advantage of our approach to optimization problems in the CSP framework.

4.1 Algorithms for CSPs

The algorithms used in this study are all variants of backtracking [3]. Backtracking is based on the idea of considering the problem's variables one at a time, and instantiating the current variable V with a value from its domain that does not violate any constraints. If no non-conflicting value is available, then a *dead-end* occurs, and the algorithm *backtracks* to the previously instantiated variable and selects for it a new value. The backtracking algorithm traverses the search space of partial assignments in a depth-first manner.

Four of the five algorithms used in the experiments are based on a backtracking variant called backjumping with dynamic variable ordering (BJ+DVO) [10], which is described in Fig. 3. Backjumping is a refinement of backtracking. In response to a dead-end, backjumping identifies a variable U, not necessarily the most recently instantiated, which is in some way connected to the dead-end. The algorithm "jumps back" to U, uninstantiates all variables more recent than U, and tries to find a new, compatible value for U from U's domain. Our implementation is based on the version of backjumping generally accepted to be the most effective, called conflict-directed backjumping [15]. When a backtracking or backjumping algorithm uses a dynamic variable ordering heuristic, the order of variable instantiation is decided at run time, and may vary at different points in the search. The DVO portion of the

Algorithm BJ+DVO
Input: A set of n variables $X_1 \dots X_n$; for each variable X_i a domain D_i; and a set of constraints.
Output: Either an assignment of one value to each variable, or "inconsistent," indicating no such assignment is possible.

0. (Initialize internal variables.) Set $D'_i \leftarrow D_i$ for $1 \leq i \leq n$.
1. (Step forward.) If all variables have value assignments, then exit with this solution. Otherwise, set cur equal to the index of the variable X_i for which $|D'_i|$ is smallest (breaking ties randomly). Set $P_{cur} \leftarrow \emptyset$.
2. Select a value $x \in D'_{cur}$. Do this as follows:
   (a) If $D'_{cur} = \emptyset$ (deadend), go to 3.
   (b) Pop x from D'_{cur} and instantiate $X_{cur} \leftarrow x$.
   (c) Remove values incompatible with $X_{cur} = x$, as follows.
   Let $U \subset \{X_1, \dots, X_n\}$ be the variables not yet assigned values.
   For each $X_u \in U$,
   for each v in D'_u,
   if $X_u = v$ conflicts with the current partial assignment,
   then remove v from D'_u,
   add X_{cur} to P_u,
   if D'_u is now empty,
   then go to (d) (without examining other X_u'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 3: Pseudo-code for the BJ+DVO algorithm.

BJ+DVO algorithm includes a "look-ahead" component which filters the domains of uninstantiated variables, removing values that are incompatible with the current partial assignment. This look-ahead is equivalent to that of the forward checking algorithm [12]. The variable ordering heuristic selects the uninstantiated variable with the smallest remaining domain. This idea was proposed by Haralick and Elliot under the rubric "fail first" [12]. In [10] we showed that BJ+DVO dominates backtracking and conflict-directed backjumping without DVO and DVO without backjumping.

In addition to experimenting with BJ+DVO, we also used two extensions to this algorithm, individually and together. Look-ahead value ordering (LVO) is a heuristic for ordering the values in the domain of the current variable; combined with BJ+DVO it yields BJ+DVO+LVO [11]. LVO ranks the values of the current variable, using a formula based on the number of conflicts each value has with values in the domains of uninstantiated variables. Experiments in [8] show that the variant of look-ahead value ordering used in the case-study can be of substantial benefit, especially on hard constraint satisfaction problems. Of course, the LVO heuristic does not always correctly predict which values will lead to solutions, but it is frequently more accurate than an uninformed ordering of values.

We refer to BJ+DVO with learning [4, 9, 2] as 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 searched 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 remaining search to be cut off at an earlier point. The risk is that the computational effort spent recording and then consulting the additional constraints may overwhelm the savings. When only constraints with i or fewer variables are recorded by learning, the result is called ith-order learning. The kind of learning employed in this paper takes advantage of processing already performed by the backjumping algorithm to identify the new constraint to be learned; it was shown in experiments with structureless random problems to be better than other versions of learning tested in [9, 8]. As LVO and learning are orthogonal techniques for improving CSP search, they can be combined in a straight forward manner. We refer to the combination as BJ+DVO+LRN+LVO [8].

The fifth algorithm, called BT+DVO+IAC [8], uses backtracking instead of backjumping and does more work at each instantiation by integrating an AC-3 based arc-consistency procedure [14]. With IAC (for "integrated arc-consistency"), values for uninstantiated 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 uninstantiated variable. At the cost of more processing per node, BT+DVO+IAC increases the likelihood of detecting early on that a partial assignment cannot lead to a solution.

For more details about these and other constraint processing algorithms, see [12, 10, 9, 11, 8, 4, 15, 5].

4.2 Optimizing with CSPs

The constraint satisfaction framework is a decision procedure; any solution that can be found is equally good. We can find an optimal schedule by treating the

Solution Procedure for Optimization
Input: A CSP with both hard constraints and an objective function that is compared to a cost-bound.
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. (a) Add a constraint (or set of constraints) to the CSP specifying that the value of the objective function must be less than the cost-bound.
   (b) Solve the CSP using a constraint algorithm.
   (c) If a solution was found, decrement the cost-bound and go to 2 (a).
3. Return the last solution found, and the corresponding cost-bound.

Figure 4: The solution procedure for optimization.

maintenance scheduling problem as a series of CSPs. The procedure is described in Fig. 4. Initially, a schedule is found with a very high cost-bound. The cost-bound is then gradually lowered, with a new solution found each time. Eventually, the cost-bound is so low that no solution exists which meets it, and the last schedule found is optimal, within the tolerance 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.

4.3 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. 4. The idea is to use a learning algorithm, such as BJ+DVO+LRN, to solve the maintenance scheduling CSPs. 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 CSP with a certain cost-bound is also a valid solution to the same problem with a higher cost-bound. If 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.

5 Problem Instance Generator

We propose evaluating the efficacy of various CSP algorithms and heuristics when applied to maintenance scheduling CSPs (MSCSPs). Performing an experimental average-case analysis requires a source of many MSCSPs. We therefore developed an MSCSP generator, which can create any number of problems that adhere to a set of input parameters.

The "maintgen" instance generator is a program that reads in a text file and creates in an output file one or more MSCSP instances to be solved by the CSP solver. The input to the generator is a file containing the problem parameters, either by explicit enumeration, by listing particular values that are then interpolated, or by specifying the parameters of a normal distribution from which the scheduling problem parameters are randomly drawn. 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 initial maximum cost per week, and the amount it is to be decremented after each successful search for a schedule, are also specified.

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 input file to maintgen, and are always set to minimum and maximum values in the output file. Maintenance costs are specified by the standard deviation and by a sample of weekly demands per unit. As with demand, values for weeks that are not given explicitly are interpolated. Operating costs are defined with exactly the same structure. The last piece of information is the number of incompatible pairs of units. The requested number of pairs is selected randomly from a uniform distribution of the units. We have not studied the ramifications of the particular structure of the maintgen program, for example using linear interpolation at some points and drawing from a normal distribution elsewhere. We do not claim it to be better or in any way more realistic than another similarly designed procedure would be.

Here is an example of an input file to the maintgen generator followed by a specification of one 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
#
# 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 lines have 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 line has initial max cost per week, and decrement amount
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.

6 Experimental Results

We 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. These problems are probably somewhat smaller than those typically encountered in industry, which may have a few dozen units and, often, 52 one-week time periods. The implementation was written in C, and the processor used was a Sun SparcStation 4, running at 110 MHz, with 32 megabytes of main memory.

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.

6.1 Optimization with Learning

In the first experiment, 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 BJ+DVO+LRN. The results are shown in Fig. 5 and Fig. 6.

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. It required less CPU time on average, but the improvement over BJ+DVO was much less than on the smaller problems.

6.2 Comparison of Constraint Algorithms

The second experiment 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 Table 1, where the column labeled "CC" reports the average number of consistency checks made, "Nodes" is the average number of times a value was assigned to a variable, and "CPU" is average CPU time in seconds.

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. On the other hand, 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. Previous results [8] have indicated that more look-ahead, such as is performed by integrated arc-consistency, is effective on problems with tight constraints, and detrimental on problems with loose constraints. Nevertheless,

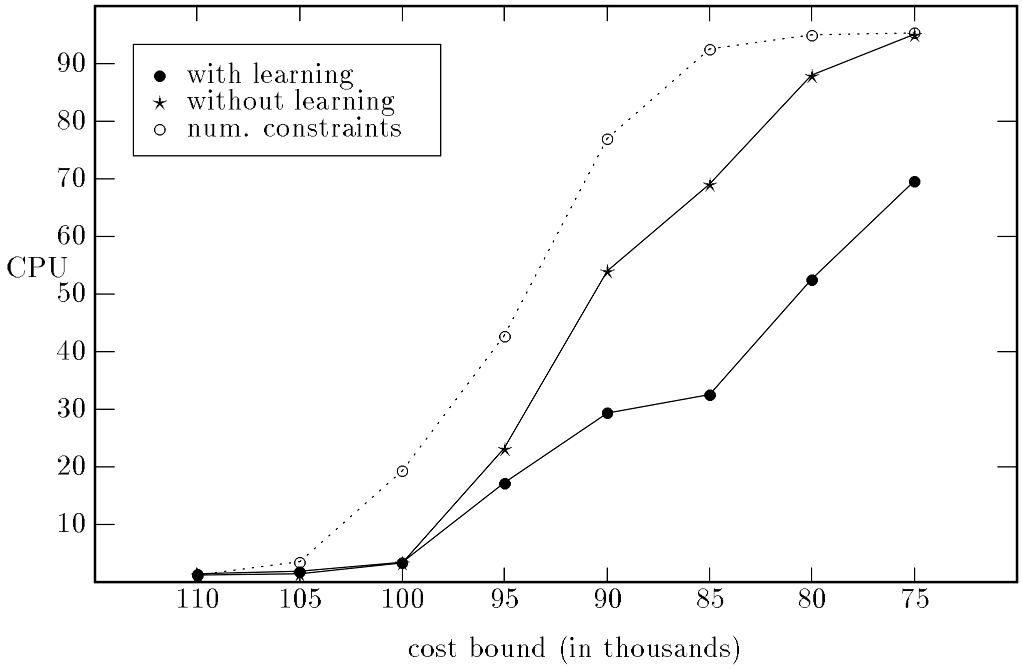

Figure 5: Average CPU seconds on 100 small problems (15 units, 13 weeks) to find a schedule meeting the cost-bound on the y-axis, using BJ+DVO with iterative learning ($\bullet$) and without learning ($\star$). Cumulative number of constraints learned corresponds to right-hand scale.

backjumping remains an effective technique on the larger problems. Another explanation for the inferiority of BT+DVO+IAC on large problems, when compared to BJ+DVO, is that the naive backtracking technique (not utilizing backjumping) really penalizes BT+DVO+IAC on this suite of problems. As is evident from the number of nodes expanded and the number of consistency checks, BJ+DVO not only had a smaller overhead per node expansion, it also explored a smaller search space (28,540 nodes instead of 32,105). This, of course, calls for augmenting IAC with backjumping rather than backtracking. However, a BJ+DVO+IAC combination requires a careful analysis of the jumpback strategies which we have not yet implemented. It is also interesting to note that a comparison of BJ+DVO and BJ+DVO+LRN indicates that the overhead of learning does not pay off on these problems. However, when embedded in interative learning the learning procedure is beneficial (see Figs. 5 and 6), due to iterative learning's ability to take advantage of the learned constraints over several problem instances. 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.

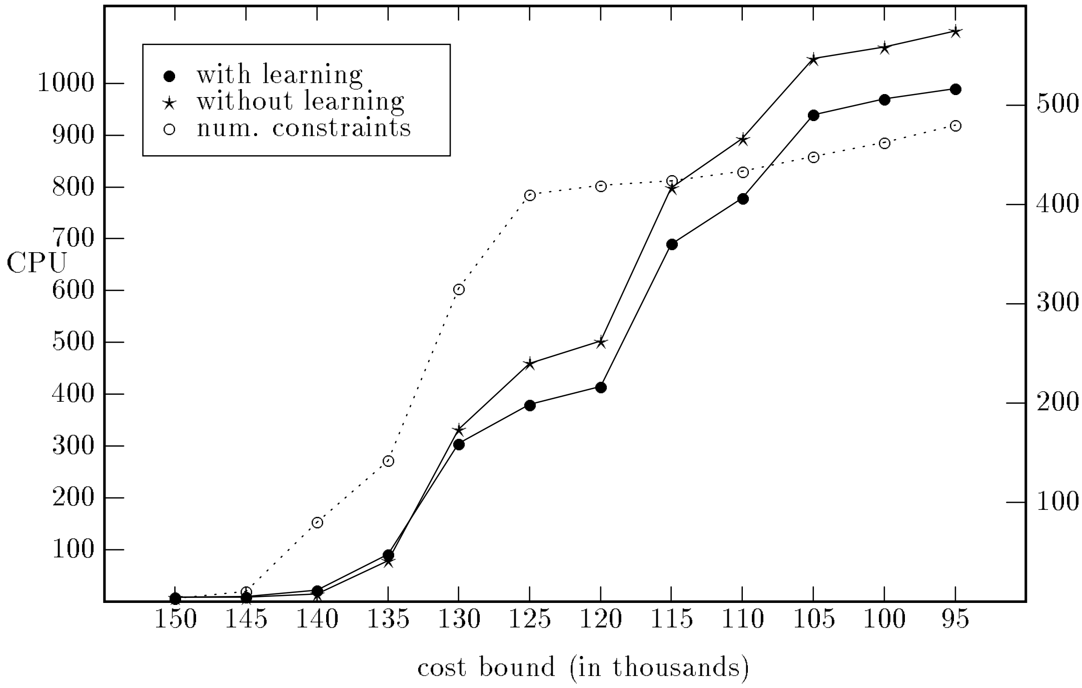

Figure 6: Average CPU seconds on 100 large problems (20 units, 20 weeks) to find a schedule meeting the cost-bound on the y-axis, using BJ+DVO with iterative learning ($\bullet$) and without learning ($\star$). Cumulative number of constraints learned corresponds to right-hand scale.

7 Related Work

Computational approaches to maintenance scheduling have been 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 [17] 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 [13]. No one approach has been found particularly superior, and the field is still one of active research. Since no standard set of benchmarks, or formalism for defining the problem, has been agreed on, it is difficult to compare the results from paper to paper. (Many studies report the result of an experiment on a single instance.) For these reasons, we have not attempted in this paper to compare our results with other work in the literature.

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

Table 1: Statistics of five algorithms on MSCSPs.

8 Conclusions

The first contribution of this paper addresses the broad problem of comparing and evaluating the performance of CSP search algorithms. We presented a methodology for evaluating algorithms on suites of application-based, structured random problems. This approach has the potential of producing results which are interestingly connected to problems arising in industry and science, while allowing statistics such as average performance to be determined and reported.

As a case-study of the methodology, we examined maintenance scheduling problems from the electric power industry, Using maintenance scheduling problems cast in the constraint framework as the structure, we constructed a random problem generator and used it to create two suites of problem instances with which we could experiment. We compared five algorithms which have the same worst-case performance. Our comparisons indicated that no one algorithm was superior to the others. When considering average CPU time, the relatively simple BJ+DVO performed best on the larger problems and worst on the smaller set. The reverse held true for BT+DVO+IAC, which does considerably more processing of uninstantiated variables after a value is assigned to the current variable. It was the best performer on the smaller problems, and the worst on the large set. BJ+DVO+LVO resulted in a good algorithm for both sizes of problems, being second best in both cases.

The fact that maintenance scheduling is inherently an optimization problem led to the third contribution reported in the paper, a new algorithm called iterative learning. Iterative learning applies learned constraints over multiple iterations of solving versions of a single problem. The multiple versions arise when a optimization problem is considered as a series of constraint problems with increasingly tight cost-bounds. While the experiments indicated that using learning was not particularly effective on single problems, the incremental approach employed by the iterative learning algorithm was shown superior to BJ+DVO for solving the optimization problem, both on large and small instances.

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.

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.