Constraint Programming in Practice: Scheduling a Rehearsal

Report APES-67-2003, September 2003.
Available online from http://www.dcs.st-and.ac.uk/~apes

Barbara M. Smith
School of Computing & Engineering
University of Huddersfield, U.K.
email: b.m.smith@hud.ac.uk

Abstract

The basic principles of constraint programming (constraint satisfaction problems, search, constraint propagation) are introduced by discussing how constraint programming can be used to solve a specific optimization problem. A set of orchestral pieces is to be rehearsed and the problem requires finding a sequence that will minimize the time that players are at the rehearsal but not playing, if they arrive for the first piece they are involved in and leave after the last. A constraint programming model of this problem is presented. A similar problem arises in 'talent scheduling' in shooting a film; improvements to the basic model are given that allow a larger instance of this kind to be solved.

1 Introduction

The rehearsal problem originated at Lancaster University, where it is said to have been devised by a member of staff in the Management Science department, who was a member of an orchestra and formalized the problem whilst waiting to play during a rehearsal. The problem was first described by Adelson, Norman and Laporte [1], although that paper does not give the data.

A concert is to consist of nine pieces of music of different durations each involving a different combination of the five members of the orchestra. Players can arrive at rehearsals immediately before the first piece in which they are involved and depart immediately after the last piece in which they are involved. The problem is to devise an order in which the pieces can be rehearsed so as to minimize the total time that players are waiting to play, i.e. the total time when players are present but not currently playing.

In the table below, 1 means that the player is required for the corresponding piece, 0 otherwise. The duration (i.e. rehearsal time) is in some unspecified time units.

Piece	1	2	3	4	5	6	7	8	9
Player 1	1	1	0	1	0	1	1	0	1
Player 2	1	1	0	1	1	1	0	1	0
Player 3	1	1	0	0	0	0	1	1	0
Player 4	1	0	0	0	1	1	0	0	1
Player 5	0	0	1	0	1	1	1	1	0
Duration	2	4	1	3	3	2	5	7	6

Table 1: Data for the rehearsal problem

For example, if the nine pieces were rehearsed in numerical order as given above, then the total waiting time would be:

Player 1: 1+3+7=11
Player 2: 1+5=6
Player 3: 1+3+3+2=9
Player 4: 4+1+3+5+7=20
Player 5: 3

giving a total of 49 units. The optimal sequence, as we shall see, is much better than this.

2 Basics of Constraint Satisfaction Problems

Some basic definitions and concepts are given here, before showing how the rehearsal problem might be modelled and solved using constraint programming. This is not intended to be a comprehensive presentation, but is biased towards the facilities that are available in constraint programming toolkits such as ECL^{i}PS^{e}and ILOG Solver and those that are required for modelling the rehearsal problem.

A constraint satisfaction problem consists of:

- a set of variables X;
- for each variable $x_i \in X$, a finite set D_{x_i} of possible values that the variable can take, called the *domain* of x_i;
- a set of constraints, C, each of which affects a subset of X, and restricts the values that can be simultaneously assigned to these variables.

A solution to a CSP is an assignment to every variable of a value in its domain, in such a way that all the constraints are satisfied. We can also find an optimal solution, given some objective; this will be discussed further below.

Often, the domains of the variables in a CSP are a range of integers, and that is the case for the model of the rehearsal problem presented below. However, this is not essential. For instance, it is often useful to have set variables, i.e. variables whose values are sets (of integers, typically). Each member of the domain is then a possible set that could be assigned to the variables, although the members of the domain are not usually explicitly listed in such a case. It is also possible to extend the definition above to real-valued variables (which therefore no longer have a finite domain).

The set of variables affected by a constraint is its *scope*. There is no requirement for constraints to take particular forms, such as linear inequalities. The only requirement is that we should be able to tell, for each possible assignment of values to the variables in its scope, whether or not that assignment satisfies the constraint. [1] This generality allows problems from many different fields to be expressed as constraint satisfaction problems.

To give some examples, the following types of constraint will be relevant to the rehearsal problem:

- $x_i = y_i z_i$.
  Arithmetic constraints of various kinds, involving mixtures of variables and constants, can be expressed.
- $x_i = 1 \implies x_{i+1} = 1$ (i.e. if $x_i = 1$ then $x_{i+1} = 1$).
  Logical constraints of various kinds are common, and allow the logic of a problem to be directly expressed. Note that a constraint such as this does adequately allow us to tell whether an assignment to the variables in the constraint's scope satisfies the

[1] Formally, a constraint $C_{x_i x_j \ldots x_l}$ with scope $\{x_i, x_j, ..., x_l\}$ is a subset of the set of possible assignments to these variables, i.e. $C_{x_i x_j \ldots x_l} \subseteq D_{x_i} \times D_{x_2} \times ... \times D_{x_l}$.

constraint or not. If $D_{x_1} = D_{x_2} = \{0, 1\}$, then of the four possible assignments to the variables x_i and x_{i+1}, three are allowed by the constraint ($x_i = 1, x_{i+1} = 1$; $x_i = 0, x_{i+1} = 1$; and $x_i = 0, x_{i+1} = 0$) and one is not ($x_i = 1, x_{i+1} = 0$).

- $t = \sum_i a_i x_i$.
  Constraint programming toolkits allow arrays of variables and constraints on arrays of variables. Here, t, a_i and x_i might all be variables, or t and/or a_i might be constants; either are equally allowed.

- allDifferent($x_1, x_2, ..., x_n$).
  It is very common in CSPs to require some or all of the variables to take different values. For the rehearsal problem, it is sufficient to consider an allDifferent constraint as shorthand for a set of pairwise $\neq$ constraints, i.e. $x_1 \neq x_2$, $x_1 \neq x_3$, ...

Constraint programming systems such as ECL^{i}PS^e, ILOG Solver and Sicstus Prolog allows constraint satisfaction problems to be expressed and solved. They provide a means of defining variables and their domains, and provide a set of pre-defined constraints; it is usually possible to define new constraints if necessary. They provide algorithms for solving CSPs, with scope for tailoring or modifying the algorithms; the basic algorithms are discussed in the next section.

3 Solving a CSP

To solve a CSP we first need to identify the decision variables. These may be all the variables in the CSP, but often some of the variables in the CSP are there only to allow relationships between other variables to be expressed more easily; we may not want to explicitly assign values to these variables. The minimum requirement is that we identify a set of variables such that an assignment to those variables that satisfies all the constraints will yield a complete solution to the original problem we are trying to solve.

The problem is then solved by a combination of systematic *search* through the set of all possible assignments to the decision variables, and *constraint propagation*: making use of the constraints to derive new information from the assignments already made that will avoid considering assignments that cannot lead to a solution.

The algorithm roughly follows these steps:

- choose a decision variable that has not yet been assigned a value;
- choose a value from those remaining in this variable's domain, and assign the value to the variable;
- use the constraints to prune values from the domains of other variables not yet assigned that can no longer be used as a result of this assignment;
- if as a result of this constraint propagation, any variable has no values left in its domain, backtrack: undo the assignment just made and choose another value of the current variable. If all values for the current variable have now been tried, backtrack to the previously assigned variable and reassign it.

The search terminates either when a solution is found, i.e. when a value has successfully been assigned to every variable (if we only require one solution) or when there are no further choices to consider (if we want all solutions, or there are no solutions).

To illustrate the idea of constraint propagation, suppose we have decision variables $x_1, x_2, ..., x_n$, each with initial domain {0,1}, and that some of the constraints are:

$$x_i = 1 \implies x_{i+1} = 1, 1 \leq i < n.$$

Suppose the problem also contains a variable w, which is defined in terms of the decision variables by the constraint:

$$w = \sum_i d_i x_i$$

where $d_1, d_2, ..., d_n$ are positive constants.

Initial propagation of the constraints in the problem will calculate the domain of w as $\{0, ..., \sum_i d_i\}$, if it is not otherwise specified. That is, its minimum value is achieved by setting $x_1 = x_2 = ... = x_n = 0$, and its maximum value by $x_1 = x_2 = ... = x_n = 1$. Note that not every value in the range $\{0, ..., \sum_i d_i\}$ may be attainable, but it would be potentially very time-consuming to test every value.

Suppose the search assigns the value 0 to x_1. In constraint propagation terms, an assignment is viewed as the removal of all other values from the variable's domain. Removing the value 1 from the domain of x_1 has no effect on x_2. Consequently, it can have no effect on $x_3, x_4, ..., x_n$ since there is no direct constraint between x_1 and these variables. However, the maximum value of w, i.e. the largest value in its domain, will be reduced to $d_2 + d_3 + ... + d_n$.

When the value 1 is assigned to x_1, constraint propagation will remove the value 0 from the domain of x_2, and then from the domains of $x_3, ..., x_n$ in turn. Thus constraint propagation can trigger a cascade of domain reductions, not restricted to the constraints which have the original variable x_1 in their scope. The domain of w will also be reduced, to the single value $\sum_i d_i$. Any variable with only one value left in its domain must be assigned that value, so all these assignments will be made as a result of the constraint propagation step.

This example illustrates that sometimes assigning a value to a variable can have little or no effect on other variables; at the other extreme, it may result in every other variable being assigned a value. An important effect of constraint propagation is that it triggers backtracking when a variable's domain becomes empty. If every value in the domain has been removed as a result of cumulative reductions when other variables have been assigned values, there cannot be any solution extending the assignments made so far. The current assignment is undone, and the values removed in propagating this assignment are restored, so that search can proceed. In this way, constraint propagation allows the search to avoid considering many partial solutions that cannot be completed, while ensuring that no solutions are missed.

Constraint programming toolkits define a propagation mechanism for every built-in constraint. The mechanism is triggered whenever a reduction of a specified kind occurs in the domain of a variable in the scope of the constraint. For some constraints, the propagation is as strong as possible: whenever any value is removed from the domain of any variable in the scope of the constraint, all values in the domains of other variables in the scope are checked and any value that can no longer satisfy the constraint is removed. However, for many constraints this is very time-consuming and often unnecessary. For other constraints only the bounds (i.e. the minimum and maximum values in the domains) are checked, as in the constraint $w = \sum_i x_i$, in the example. As already noted, there are likely to be values in the calculated domain $\{0, ..., \sum_i d_i\}$ of w which cannot satisfy the constraint, but these are not removed. Other constraints are only checked if a variable in the scope is assigned a value: for instance, in the constraint $x \neq y$, if x is assigned a value, then that value is removed from the domain of y.

Some constraints allow different levels of propagation, notably the allDifferent constraint: as mentioned earlier, this can be treated as a collection of $\neq$ constraints, and propagation only involves removing an assigned value from the domain of every other variable in the constraint. Alternatively, it can be treated as a single constraint and if, for instance, the total number of different values available in the domains of the variables is ever less than n, it will be detected that the constraint cannot be satisfied. If this happens as the result of an assignment, then the assignment will fail and the search will backtrack. Note that the additional processing does not change the constraint, nor the problem that the constraint is part of: the same solutions will be found. However, stronger constraint propagation should reduce the search required to find a solution, at the expense of longer processing time for the allDifferent constraint; sometimes the additional effort pays off, and sometimes not.

Because the search is systematic and implicitly considers every possible assignment to the decision variables, if the search continues until no further choices remain to consider, then all possible solutions have been found; if no solution has been found, this is because there are no solutions satisfying the constraints. Because the search propagates the constraints after every assignment, unnecessary choices need not be explicitly explored, and hence considering all possible assignments can be relatively efficient.

4 Decision Variables for the Rehearsal Problem

We have to decide the order of the pieces, and an obvious choice for the decision variables is to have a variable for each position in the sequence, with values corresponding to the pieces. If there are n pieces, and hence n slots in the sequence, we can define variables $s_1, s_2, ..., s_n$, each with domain $\{1, 2, ..., n\}$, where $s_i = j$ if piece j is in position i in the sequence, $1 \leq i \leq n$.

An assignment to these variables is a valid sequence, and so could represent a feasible order for the rehearsal, if every variable has a different value. Hence, the basic constraint is allDifferent($s_1, s_2, ..., s_n$). Any solution will then be a permutation of {1,2,3,..., n}. However, since we are required to find an optimal sequence, not simply a feasible one, this very simple CSP will need to be extended to allow optimization.

5 Optimization

To adapt a constraint satisfaction problem to become an optimization problem, we include a variable, say t, that represents the objective. Typically, the objective variable is not a decision variable. The CSP must include constraints linking the objective variable to other variables in the problem, so that when the decision variables have been assigned a value, the objective variable must also have been assigned a value as a result of constraint propagation. When a solution is found, the value of t in this solution becomes a bound on the value in any future solution. Suppose we are minimizing t and that the value of t in the first solution found is t_0. The constraint solver adds a constraint $t < t_0$, and attempts to find a new solution satisfying this constraint. This is repeated, with the constraint on the objective becoming progressively tighter as each new solution is found. At some point, there will be no further solution satisfying the current constraint; when the search terminates, having found no solution, the last solution found has been proved optimal.

In the rehearsal problem, the objective variable t is the total waiting time. Hence, we need to provide a link between the sequence of pieces and the total waiting time: this will require the introduction of further variables and constraints.

The total waiting time is the sum of the waiting times for the individual players. The waiting time for a player depends on when they arrive and when they leave, which in turn depend on where the pieces that they play in appear in the sequence. More specifically, if a player does not play in a piece, in order to know whether the player is waiting while that particular piece is rehearsed, we need to know when the piece occurs in the sequence and whether the player has already arrived by then and has not yet left.

Given variables representing these aspects of the problems and constraints to define them, any assignment to the sequence variables $s_1, s_2, ..., s_n$ will result in a value being assigned to the objective variable t, and hence the scheme outlined above for finding an optimal solution can be used. However, in a problem such as this, simply building a correct model will not be enough, except for small problems. To find an optimal solution, and prove that it is optimal, in any reasonable time, the link between the decision variables and the objective variable needs to be as tight as possible, so that:

- as the sequence of pieces is built up, constraint propagation should ensure that the upper and lower bounds on t, given by the maximum and minimum values in its

domain, reflect as accurately as possible the cost of completing the sequence. Then the search can abandon a partial sequence and try another when the minimum value of t becomes larger than the value of the current best solution.

- conversely, constraint propagation in the other direction should allow a tight bound on t, once a good solution has been found, to prune values from the domains of the sequence variables, so that partial sequences that cannot lead to a better solution are never explored.

Devising a model which takes these aims into account so that the problem can be efficiently solved takes some skill. Some possibilities are discussed below.

6 A CSP Model

As discussed in the last section, in building the link between the sequence variables and the objective, t, it will be useful to know not only which piece is in each place in the sequence, but also where in the sequence each piece appears. For each piece j, let d_j be a variable whose value will be the position in the sequence of piece j. These variables are linked to the decision variables, $s_1, s_2, ...s_n$ by:

$$s_i = j \text{ iff } d_j = i.$$

The variables $d_1, d_2, .., d_n$ are the *duals* of $s_1, s_2..., s_n$ and are frequently used in modelling problems requiring finding a permutation (in this case, of the pieces to be rehearsed) [2, 5].

Let π_{kj}, $1 \leq k \leq m$, $1 \leq j \leq n$ be an array of 0 and 1 values representing the data given in Table 1, i.e. π_{kj} = 1 iff player k plays in piece j.

The dual variables allow new variables to be introduced, e.g. $p_{kj} = 1$ iff player k is playing in slot j, defined by:

$$p_{kd_j} = \pi_{k,j}$$

From these we can define variables a_{ki}, l_{ki} for each player k and each slot i in the sequence, with domains $\{0, 1\}$, such that:

$a_{ki} = 1$ if player k has arrived by the start of slot i, 0 otherwise.
$l_{ki} = 1$ if player k leaves at the end of slot i or later, 0 otherwise.

To relate these variables to the sequence variables $s_1, s_2, ..., s_n$, we can introduce constraints expressing that:

- $a_{k1} = 1$ iff player k is playing in the piece in slot 1 of the sequence, i.e. $a_{k1} = p_{k1}$;
- for $i > 1$, $a_{ki} = 1$ iff player k has already arrived at the start of the previous slot or player k is playing in the piece in slot i of the sequence, i.e. $a_{ki} = 1$ iff $a_{k,i-1} = 1$ or $p_{ki} = 1$;

The variables $l_{ki}, 1 \leq k \leq m, 1 \leq i \leq n$ can be defined by similar constraints.

For each player, the first non-zero value in $a_{k1}, a_{k2}, ..., a_{kn}$ corresponds to the first piece in the sequence that player k plays in, and all subsequent values of a_{ki} have the value 1. Similarly, the last non-zero value in $l_{i1}, l_{i2}, ..., l_{in}$ corresponds to the last piece in the sequence that player i plays in, and all previous values are 1. For instance, if the sequence is 1, 2, 3, 4, 5, 6, 7, 8, player 5 arrives at the start of the 3rd slot and leaves after the 8th:

Player i is at the rehearsal in slot j iff he/she has arrived and not yet left, i.e. $a_{ki} = 1$ and $l_{ki} = 1$. Defining new variables $r_{ki} = 1$ iff player k is at the rehearsal during slot i, we have the constraints: $r_{ki} = a_{ki} l_{ki}$.

i	1	2	3	4	5	6	7	8	9
π_{5i}	0	0	1	0	1	1	1	1	0
a_{5i}	0	0	1	1	1	1	1	1	1
l_{5i}	1	1	1	1	1	1	1	1	0
r_{5i}	0	0	1	1	1	1	1	1	0

Hence we can define variables $w_{kj} = 1$ if player k is waiting while piece j is being rehearsed, 0 otherwise, i.e. $w_{kj} = 1$ iff player k does not play in piece j and player k is at the rehearsal while piece j is being rehearsed. The constraints defining w_{kj} can be expressed as:

$$\begin{aligned} w_{kj} &= r_{k,d_j} \quad \text{if } \pi_{kj} = 1 \\ &= 0 \quad \text{otherwise} \end{aligned}$$

Given these variables, the waiting time for player k is $\sum_j w_{kj}\delta_j$, where δ_j is the time to rehearse piece j (from Table 1). The objective, i.e. the total waiting time, t, is given by

$$t = \sum_k \left(\sum_j w_{kj}\delta_j \right)$$

Hence, a set of assignments to the sequence variables $s_1, s_2, ..., s_n$ will lead to a value being assigned to t, the total waiting time, via the other variables. The resulting model has a large number of variables, but still only n decision variables, $s_1, s_2, ..., s_n$: all the other variables will be assigned values by constraint propagation as the decision variables are assigned during search.

7 Variable ordering

The outline search algorithm given earlier says only "choose a decision variable that has not yet been assigned a value"; it does not say how this choice should be made, and in fact in constraint programming toolkits this choice is left up to the programmer, as is the choice of value to assign. An ordered list of the decision variables is passed to the search algorithm, and by default the first unassigned variable in the list will be chosen next. A little thought shows that just assigning the variables in the obvious order $s_1, s_2, ..., s_n$, i.e. constructing the sequence consecutively from the beginning, will be a very poor choice in this case.

One of the aims in building the CSP model is to ensure that as the sequence is constructed, i.e. as the decision variables are assigned values, the constraints should propagate to the objective variable, so that the lower bound, in particular, is increased. However, if the sequence is constructed from the beginning to the end, the waiting time for many players will not be known until most of the sequence has been decided, i.e. until the last piece that the player plays in is reached. Until then, the lower bound on the waiting time for an individual player will calculated by assuming that all remaining pieces that this player does not play in can be moved to the end of the sequence. Hence, the lower bound will be equal to the waiting time already incurred, i.e. due to the pieces already sequenced.

A much better search order is to construct the sequence "ends to middle", i.e. to choose the first piece, then the last piece, then the second piece, and so on. The advantage is that it will be known much earlier in the search when a player leaves, as well as when they arrive. With this search order and the model given earlier, the waiting time for player k is known exactly as soon as the earliest and latest pieces that they play in appear in the sequence. It is simple to assign the variables in this order: the ordered list of variables passed to the search algorithm is simply constructed as $s_1, s_n, s_2, s_{n-1}, ...$

For instance, suppose the first four variable assignments are s_1 = 3, s_9 = 9, s_2 = 8, s_8 = 4 (which in fact can be extended to an optimal solution). At that point in the search, the domains of the sequence and dual variables, following propagation of the allDifferent constraint and the constraints $s_i = j$ iff $d_j = i$, are:

i	1	2	3	4	5	6	7	8	9
s_i	3	8	{1, 2, 5..7}	{1, 2, 5..7}	{1, 2, 5..7}	{1, 2, 5..7}	{1, 2, 5..7}	4	9
d_i	{3..7}	{3..7}	1	8	{3..7}	{3..7}	{3..7}	2	9

Given the data in Table 1, it is now known that player 1 arrives in time for the second slot, and leaves just before the last slot, and the variables a_{1i}, l_{1i}, r_{1i} recording whether player 1 has arrived by slot i, leaves after slot i and is present during slot i are fixed by constraint propagation:

i	1	2	3	4	5	6	7	8	9
π_{1i}	1	1	0	1	0	1	1	0	1
a_{1i}	0	1	1	1	1	1	1	1	1
l_{1i}	1	1	1	1	1	1	1	1	0
r_{1i}	0	1	1	1	1	1	1	1	0

Clearly, player 1 will have to wait during piece 5. At this point, it has not been decided when piece 5 will appear in the sequence: as shown by the domain of d_5, it can be in any of slots 3 to 7. However, because r_{1i} =1 for $3 \leq i \leq 7$, i.e. player 1 is at the rehearsal, it can be determined that w_{15} = 1, i.e. that player 1 is waiting during piece 5. It is also known that w_{1i} = 0 for $i \neq 5$, either because player 1 is not at the rehearsal while the piece is played, or because player plays in that piece. Hence, the total waiting time for player 1 is known to be 3, the duration of piece 5.[2]

Thus, assigning the variables from the ends of the sequence to the middle allows the search to determine the waiting time for a player as soon as the first and last pieces that they play in have been determined. This should allow the optimal sequence to be found much more quickly than if the sequence were constructed consecutively, from first piece to last piece.

Cheng, Diamond and Lin [6], in a paper on a similar problem discussed below, constructed the sequence in the same fashion in a heuristic solution method. However, their rationale for this ordering is different; they argue that the 'outside' slots in the sequence (i.e. the earliest and latest) are potentially the most expensive if the wrong choices are made, and making these choices first gives the widest choice of piece to place there.

Having chosen a variable to assign, we could choose a value to assign to it (rather than choosing the smallest value in the domain, by default). At the start, good value selection could ensure that the first solution found is of low cost. For the rehearsal problem, the value selection could be based on Cheng *et al.*'s heuristic, for instance. However, in what follows the default value choice has been used, i.e. the smallest value in the domain is chosen.

8 Symmetry

Given any sequence of pieces, reversing the sequence does not change any player's waiting time. This symmetry in the problem can cause difficulties when we try to find the optimal solution to the model given. For instance, having found an optimal sequence starting with piece 3 and ending with piece 9, the search will eventually consider sequences starting with piece 9 and ending with piece 3, since at least one such sequence will appear to offer the potential to improve on the optimal value, until the sequence is nearly complete. This is clearly a waste of effort, and can easily be prevented by adding a constraint to the model, which is only satisfied by one of each pair of mirror-image sequences. An obvious constraint is that the number of the piece in slot 1 must be less than the number of the piece in slot n, i.e. $s_1 < s_n$. The optimal sequence with s_1 = 3 and s_n = 9 satisfies this constraint; the reverse sequence does not.

[2] In ILOG Solver, the constraints propagate as described; it is possible that in other constraint programming systems, the equivalent constraints would not propagate in the same way.

9 Results

The minimum waiting time for the rehearsal problem is 17 time units, and an optimal sequence is 3, 8, 2, 7, 1, 6, 5, 4, 9. Table 2 compares a few variants of the model in finding an optimal sequence. The number of times the search backtracks is a good measure of search effort; the table gives the number of backtracks incurred up to the point where an optimal sequence is found and the total number of backtracks, including the proof of optimality. It is useful to distinguish between these two phases of the search. The table also gives the total runtime.

Search order	F	P	Time
First to last	37,213	65,090	23.9
Ends to middle	1,170	1,828	1.42
First to last with $s_1 < s_n$	35,679	48,664	18.4
Ends to middle with $s_1 < s_n$	1,125	1,365	0.99

Table 2: Solving the rehearsal problem using ILOG Solver. F is the number of backtracks to find the optimal solution, P is the total number of backtracks to prove optimality. Time is the cpu time in seconds on a 600MHz Celeron PC.

As expected, the 'ends-to-middle' ordering of the variables makes a huge difference to the time and effort required to solve the problem. Adding the constraint $s_1 < s_n$ to prevent considering symmetrically equivalent sequences also makes a significant difference to the search effort required to prove optimality, though not much difference to finding an optimal solution. As the table shows, the rehearsal problem can be satisfactorily solved using the constraint programming model, given a good ordering of the search variables.

10 A Talent Scheduling Problem

A very similar problem occurs in devising a schedule for shooting a film, as described by Cheng, Diamond and Lin [6]. Different scenes require different subsets of the cast, and cast members are paid for time they spend on set waiting. The only difference between this *talent scheduling* problem and the rehearsal problem is that different cast members are paid at different rates, so that the cost of waiting time depends on who is waiting.

Sample data (relating to a film called *Mob Story*) is given in [6]. Some preprocessing of their data was done, by combining two days requiring the same set of actors into a single piece of work of two days duration; clearly we can assume that these two days can be consecutive in an optimal sequence. This gives the data in Table 3 with 20 'pieces' and 8 'players'.[3] In the following, the same terminology of players and pieces will be used as for the rehearsal problem.

The model described so far can be easily modified to allow for the cost of each player, and this does not appear to make the problem significantly more difficult to solve. However, this is clearly a much larger problem than the original rehearsal problem, and the modified model cannot be solved for this problem in any reasonable time. The following sections describe how the model can be improved by adding further constraints, until it can be successfully solved.

[3]The data given by Cheng *et al.* seems inconsistent: the value they give for their solution does not appear to match the costs. They give a heuristic method for finding a good solution, and give the value of the solution at 17,900, but from their data it appears that its cost should be 16,100.

Piece	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	Cost/100
Player 1	1	1	1	1	0	1	0	1	0	1	1	0	0	0	0	0	0	0	0	0	10
Player 2	1	1	1	0	0	0	1	1	0	1	0	0	1	1	1	0	1	0	0	1	4
Player 3	0	1	1	0	1	0	1	1	0	0	0	0	1	1	1	0	0	0	0	0	5
Player 4	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	5
Player 5	0	1	0	0	0	0	1	1	0	0	0	1	0	1	0	0	0	1	1	1	5
Player 6	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	40
Player 7	0	0	0	0	1	0	1	1	0	0	0	0	0	0	1	0	0	0	0	0	4
Player 8	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	20
Duration	2	1	1	1	1	3	1	1	1	2	1	1	2	1	2	1	1	2	1	1	

Table 3: Data for the film production problem

11 Implied Constraints

It is common in modelling problems as CSPs to add constraints to the model which are logically redundant: they are implied by the existing constraints and so do not change the solutions of the problem. However, if well-chosen they can have a dramatic effect on the time taken to solve the problem. Good implied constraints allow dead-ends in the search to be recognised sooner than they would be otherwise; they represent aspects of the problem that are not currently reflected in the way that the existing constraints propagate.

In the rehearsal/talent scheduling problem, a shortcoming of the model so far is that it does not allow new bounds on the total cost to have much impact on the sequence variables. As a partial sequence is constructed, the bound will only have an effect once the search tries to assign pieces that will cause the partial sequence to exceed the current bound. It would be better if the bound could be used earlier to remove values from the domains of the sequence variables, before they are tried rather than after.

For instance, player 6, in the talent scheduling problem, costs 4000 units for every unit of waiting time: once a good solution has been found, and a partial sequence including some waiting time has been constructed, player 6 can be allowed little or no waiting. As soon as one of the pieces that player 6 is involved in is included in the sequence, the others must be close to it, and should be removed from the domains of other sequence variables.

A constraint that expresses this idea can be constructed by introducing new variables for each player representing the first and last times that they are present. For player k, $f_k = i$ if the first piece that they play in is in slot i, similarly l_k represents the last piece that they play in. We can express f_k, l_k in terms of the dual variables, since the value of a dual variable d_j is the slot in the sequence occupied by piece j. We know from the problem data which subset of pieces player k plays in: if we extract the corresponding subset X_k of the dual variables, the required constraints are:

$$f_k = \min(X_k)$$
$$l_k = \max(X_k)$$

(Again, these are perfectly valid constraints.)

We can also define f_k and l_k in terms of the a_{ki} variables defined earlier, where $a_{ki} = 1$ iff player k has arrived by the start of slot i:

$$f_k = 1 \text{ iff } a_{k1} = 1$$
$$f_k = j \text{ iff } a_{kj} = 1 \text{ and } a_{k,j-1} = 0, \text{ for } j > 1.$$

and similarly for l_k. Then the waiting time for player k depends on the number of pieces that player k plays in, ν_k, and the difference between f_k and l_k. It also depends on the durations of the pieces sequenced between f_k and l_k, but the minimum duration (1 time unit) gives a lower bound:

$$\text{waiting time for } k \geq l_k - f_k - \nu_k + 1.$$

For instance, from Table 10, player 1 plays in 8 pieces (ν_1 = 8); if the pieces are

sequenced in the order 1, 2, 3, ..., 20, f_1 = 1 and l_1 = 11; the waiting time for player 1 in this sequence is at least 3, from this constraint. In fact, in this case, the waiting time is exactly 3.

This constraint will not have any propagation effect once enough sequence variables have been assigned to determine both f_k and l_k: at that point, since we are constructing the sequence ends to middle, it will be known exactly which pieces player k has to wait through, and hence the waiting time will already be known. However, it will have an effect in the opposite direction when we already have a good solution, giving a tight upper bound on the total waiting time, and hence potentially a tight upper bound on the waiting time for player k. This will propagate to l_k and f_k, and may reduce the minimum value of l_k and the maximum value of f_k, and thereby the maximum and minimum values of all the dual variables in the set X_k. In particular, once either f_k or l_k is known, i.e. a piece involving player k has been placed in the sequence, all the other pieces involving player k may have to be placed close to this one in the sequence in order to meet the current constraints on waiting time. The constraint will allow this to be reflected in the domains of the dual variables in X_k. Propagating the constraints relating the dual variables and the sequence variables then removes any piece involving player k from the domains of sequence variables relating to more distant positions in the sequence.

Search order	F	P	Time
First to last	6,792	8,363	5.64
Ends to middle	467	570	0.754

Table 4: Solving the rehearsal problem using ILOG Solver, with implied constraints on the waiting time for each player

Table 4 shows the effect of adding this implied constraint on solving the rehearsal problem (with the symmetry constraint $s_1 < s_n$). With both search orders, it reduces search dramatically. For the rehearsal problem, a further refinement is needed to solve the full problem in, say, under an hour.

12 Optimality Constraints

We can also add constraints which are not logically implied by the initial constraints but which an optimal sequence must satisfy. A set of optimality constraints for the rehearsal problem was prompted by the observation that in the talent scheduling data, there are several pairs of pieces that involve almost the same set of players. For instance, pieces 7 and 8 both involve players 2, 3, 5, 7 and 8: piece 8 also involves player 1. By considering the different cases which can occur, we can determine whether or not piece 7 should come before or after piece 8 in an optimal sequence.

First notice that if we switch the positions of pieces 7 and 8 in a sequence, it will make no difference to the waiting time except possibly for player 1. Suppose that pieces 7 and 8 are in positions a and b in the sequence, in some order.

Case (i) Player 8 plays in another piece before a in the sequence but does not play in any piece after b ($a > 1$). In this case, if piece 7 is in position b, i.e. after piece 8, the last piece that player 1 plays is before b, so player 1 is not waiting while piece 7 is rehearsed. On the other hand, if piece 7 is in position a, player 1 cannot leave until after piece 8 is rehearsed in position b. Hence player 1 is waiting during piece 7 (and possibly during other pieces that are sequenced between a and b as well). So in an optimal sequence, piece 8 must be before piece 7.

Case (ii) Player 1 plays in another piece after b in the sequence but does not play in any piece before a ($b < n$). This is the reverse of the previous case, and in an optimal sequence, piece 7 must be before piece 8.

Case (iii) Player 1 plays in other pieces both before a and after b. The order of pieces 7 and 8 makes no difference to the waiting time, so we can choose either order and enforce it: say, piece 8 is before piece 7.

Case (iv) Player 1 does not play before a or after b. In this last case, the optimal order of pieces 7 and 8 then depends on the other pieces between a and b that player 1 plays in, so we cannot enforce either order.

From all these cases, we can say that in an optimal sequence, piece 7 can be before piece 8 only if player 1 has not already arrived by the time that piece 7 is played. i.e.

$$d_7 < d_8 \implies a_{1d_7} = 0.$$

Hence, we can impose this as a constraint. In the talent scheduling problem, 13 similar constraints can be added on other pairs of pieces that are played by identical sets of players except for an additional player in one of the pieces. In the rehearsal problem, there are three such pairs (pieces 2 and 1; 4 and 2; 5 and 6).

Problem	F	P	Time
Rehearsal	357	448	0.91
Talent scheduling	447,139	576,579	1120

Table 5: Adding optimality constraints on pairs of pieces differing by one player

These constraints give a further reduction in the search required to solve the rehearsal problem, and the talent scheduling problem can now be solved, in under 20 minutes. The optimal sequence found is 4, 1, 10, 11, 3, 13, 12, 2, 6, 8, 9, 7, 20, 5, 15, 14, 17, 18, 16, 19, with cost 14,600. Note that piece 8 is before piece 7 in this sequence, and piece 8 is the last piece that player 1 plays in.

13 Discussion

As the two examples considered here demonstrate, developing a correct CSP model of a problem does not guarantee that it can be solved in a reasonable time. In fact, Fink and Voß [3] have shown that this class of problems is NP-hard, so it is not surprising that proving optimality for the larger problem requires extending the model to allow more constraint propagation, as well as much longer running time.

As far as I am aware, other work on the talent scheduling problem has not attempted to prove optimality for instances of the size considered here; Cheng *et al.* [6] developed a heuristic solution method and Nordström and Tufekci [4] used a genetic algorithm on large randomly-generated problems constructed to have a known optimal solution (with no waiting time). It will clearly not be possible to use the final model presented here to solve problems much larger than the instance considered. However, some further improvements will be possible: so far, nothing has been done to direct the search to good solutions at the start, and the first solution found is very poor. A good upper bound on the value of the optimal solution reduces the search considerably, and such a bound could be derived using the heuristic method in [6].

Solving these problems in a reasonable time using constraint programming requires an appropriate variable ordering even for the small rehearsal problem. Solving the talent scheduling problem requires also the addition of implied constraints and optimality constraints. It requires experience and understanding of how constraints propagate to develop such constraints. However, the aim is that the search should emulate the process that a human problem solver would follow in solving the problem (although much faster and more exhaustively). Further constraints are added to a model when the programmer suspects, or sees evidence, that the search is missing obvious deductions. Hence, although constraint programming does require an understanding of search and constraint propagation, it is by

understanding the problem and building in that understanding that we can develop a successful model.

Acknowledgments. I should like to thank Farouk Aminu and Richard Eglese of Lancaster University for introducing me to the rehearsal problem. I am grateful to the other members of the APES group (see http://www. dcs.st-and.ac.uk/˜apes) for their interest and encouragement, and in particular to Tien Ba Dinh for discussions on the rehearsal problem. I am especially grateful to Ian Miguel for programming help. This work was supported by EPSRC grant GR/M90641.

References

[1] R. M. Adelson, J. M. Norman, and G. Laporte. A Dynamic Programming Formulation with Diverse Applications. *Operational Research Quarterly*, 27:119–121, 1976.

[2] B. M. W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4:167–192, 1999.

[3] A. Fink and S. Voß. Applications of modern heuristic search methods to pattern sequencing problems. may 1998.

[4] A.-L. Nordström and S. Tufekci. A Genetic Algorithm for the Talent Scheduling Problem. *Computers Ops Res.*, 21:927–940, 1994.

[5] B. M. Smith. Dual Models in Constraint Programming. Research Report 2001.02, School of Computing, University of Leeds, Jan. 2001.

[6] T.C.E.Cheng, J. Diamond, and B.M.T.Lin. Optimal scheduling in film production to minimize talent hold cost. *Journal of Optimization Theory and Applications*, 79:197–206, 1993.