ELSEVIER

European Journal of Operational Research 119 (1999) 557–581

EUROPEAN JOURNAL OF OPERATIONAL RESEARCH

www.elsevier.com/locate/orms

Invited Review

Constraint satisfaction problems: Algorithms and applications

Sally C. Brailsford [a], Chris N. Potts [b,*], Barbara M. Smith [c]

[a] *School of Management, University of Southampton, Southampton SO17 1BJ, UK*
[b] *Faculty of Mathematical Studies, University of Southampton, Southampton SO17 1BJ, UK*
[c] *School of Computer Studies, University of Leeds, Leeds LS2 9JT, UK*

Received 1 October 1998

Abstract

A constraint satisfaction problem (CSP) requires a value, selected from a given finite domain, to be assigned to each variable in the problem, so that all constraints relating the variables are satisfied. Many combinatorial problems in operational research, such as scheduling and timetabling, can be formulated as CSPs. Researchers in artificial intelligence (AI) usually adopt a constraint satisfaction approach as their preferred method when tackling such problems. However, constraint satisfaction approaches are not widely known amongst operational researchers. The aim of this paper is to introduce constraint satisfaction to the operational researcher. We start by defining CSPs, and describing the basic techniques for solving them. We then show how various combinatorial optimization problems are solved using a constraint satisfaction approach. Based on computational experience in the literature, constraint satisfaction approaches are compared with well-known operational research (OR) techniques such as integer programming, branch and bound, and simulated annealing.

Keywords: Constraint satisfaction; Combinatorial optimization; Integer programming; Local search

1. Introduction

A (finite domain) constraint satisfaction problem (CSP) can be expressed in the following form. Given a set of variables, together with a finite set of possible values that can be assigned to each variable, and a list of constraints, find values of the variables that satisfy every constraint. Many problems in operational research (OR) fall within this general framework. As an example, consider a timetabling problem in which examinations are to be assigned to various periods. In addition to the obvious constraints that any examination involving the same student must be assigned to different periods, there may be additional constraints due to room sizes, requirements for students not to have examinations in contiguous periods, etc. Another example occurs in production scheduling. Jobs are to be processed on machines that can handle only one job at a time, so that each job is completed by a given deadline. Further examples arise from the observation that an optimization problem can be

* Corresponding author. Tel.: +44 1703 593659; fax: +44 1703 595147

0377-2217/99/$ – see front matter
PII: S0377-2217(98)00364-6

expressed as a sequence of CSPs. By setting a threshold value on the objective function value, an 'objective' constraint can be added. Successive adjustments to the threshold value according to whether there are values of the variables that satisfy all constraints, allow the optimal value of the objective function value to be obtained.

Constraint satisfaction problems are combinatorial in nature. For many categories of CSPs, an efficient algorithm is unlikely to exist (these problems are NP-complete). Thus, an algorithm that guarantees to find a solution that satisfies all constraints, assuming that such a solution exists, is enumerative and therefore has an exponential time requirement in the worst case. In practice, it may be sufficient to find a solution at reasonable computational expense, that satisfies most of the constraints, especially if the problem contains 'soft' constraints or an 'objective' constraint. If all or as many constraints as possible are satisfied, we refer to the solution as *exact*; otherwise, it is *approximate*. (When referring to an optimization problem, an approximate solution is one in which all constraints are satisfied, but the optimal value of the objective function is not necessarily attained.)

A variety of approaches can be used to tackle CSPs. Integer programming techniques (cutting plane methods and branch and bound) can be applied to find an exact solution. On the other hand, there are various approaches that provide an approximate solution, including local search methods (simulated annealing, threshold accepting, tabu search and genetic algorithms) and neural networks. However, there is a special-purpose technique that is widely used for solving CSPs; it uses tree search combined with backtracking and consistency checking.

By *constraint programming* (CP), we mean the computer implementation of an algorithm for solving CSPs. It is possible to implement these algorithms in a conventional logic programming language such as PROLOG. Unfortunately, the fundamental logical solution procedure that these languages use is inefficient since the constraints are not actively used in the search for a solution. Therefore, modifications of logic programming languages have been developed which allow constraints to be expressed and solved; for example, CHIP (Constraint Handling in PROLOG). This approach is called *constraint logic programming* (CLP). However, it is also possible to implement constraint programming algorithms in general-purpose programming languages, or specialist declarative languages. For example, the constraint programming tool ILOG SOLVER is a library of routines written in C++. Thus, CLP is just one possible CP approach, although the terms are sometimes used synonymously (and confusingly) in the literature.

The study of CSPs has largely been undertaken within the artificial intelligence (AI) community. The pioneering work was undertaken in the early 1970s by Montanari (1974), Waltz (1972) and Mackworth (1977). In those early days, the power of available computers was often insufficient to allow practical problems to be solved. During the last decade, advances in computer technology have contributed to a significant growth in research in the area of constraint satisfaction. Specifically, various enhancements to the algorithms for solving CSPs have been incorporated in software systems. Such systems are used to obtain solutions of many practical OR (and other) problems. The launch in 1996 of a new journal, *Constraints*, whose aim is to provide a common forum for the many disciplines interested in constraint satisfaction and optimization, indicates the growth of interest in this area.

In spite of the widespread use of constraint satisfaction techniques by researchers in AI, these approaches appear to be relatively unknown by operational researchers. This paper aims to provide a partial remedy by introducing operational researchers to the area of finite domain constraint satisfaction. We introduce the main approach (constraint programming) for solving CSPs, and discuss modelling considerations. A variety of application areas are given, and a comparison of constraint satisfaction techniques with some of the other approaches mentioned above is given.

The remaining sections of this paper are organized as follows. In Section 2, a formal definition of CSPs is given. Sections 3 and 4 describe tests for consistency checking, and specify the search procedures that are used in constraint programming algorithms. Sections 5 and 6 are

concerned with issues likely to affect the time required to solve a CSP; problem formulation, and the search strategy which is defined by variable and value ordering. Section 7 describes various applications of constraint satisfaction for solving combinatorial optimization problems. An evaluation of constraint programming as a technique is given in Section 8. Lastly, some concluding remarks are contained in Section 9.

2. Constraint satisfaction problems

2.1. Problem definition

A CSP consists of:

- a set of *variables* $X = \{x_1, \ldots, x_n\}$;
- for each variable x_i, a finite set D_i of possible values (its *domain*);
- a set of *constraints* restricting the values that the variables can simultaneously take.

Note that the values need not be a set of consecutive integers (although often they are); they need not even be numeric. For problems with domains that are not finite, such as problems containing continuous variables, the solution techniques that we describe in this paper require modification.

A *feasible* solution to a CSP is an assignment of a value from its domain to every variable, in such a way that every constraint is satisfied. In this case, the problem is *satisfiable*. On the other hand, if there is no assignment of values to variables from their respective domains for which all constraints are satisfied, then the problem is *unsatisfiable*. We may want to find:

- just one solution, with no preference as to which one;
- all solutions;
- an optimal, or at least a good, solution, given some objective function defined in terms of some or all of the variables.

Although algorithms for solving CSPs are aimed at simply finding a feasible solution, they can be adapted to finding an optimal solution. For instance, an objective variable can be created to represent the objective function, an initial solution is found, and then a new (objective) constraint is introduced specifying that the value of the objective variable must be better than in the initial solution. This is done repeatedly, tightening the constraint on the objective variable as each solution is found, until the problem becomes unsatisfiable: the last solution found is then an optimal solution. The number of iterations, and therefore the computation time, depends on the quality of the initial solution. A common practice is to apply a *heuristic method* for generating an initial solution.

2.2. Constraints

The constraints of a CSP are usually represented by an expression involving the affected variables, e.g. $x_1 \neq x_2$, $2x_1 = 10x_2 + x_3$ and $x_1x_2 < x_3$.

Formally, a constraint $C_{ijk\ldots}$ between the variables $x_i, x_j, x_k, \ldots$ is any subset of the possible combinations of values of $x_i, x_j, x_k, \ldots$, i.e. $C_{ijk\ldots} \subseteq D_i \times D_j \times D_k \times \cdots$. The subset specifies the combinations of values which the constraint allows.

For example, if variable x has the domain $\{1,2,3\}$ and variable y has the domain $\{1,2\}$ then any subset of $\{(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)\}$ is a valid constraint between x and y. The constraint $x = y$ is equivalent to the subset {(1,1), (2,2)}.

Although the constraints of real problems are not represented this way in practice, the definition does emphasize that constraints need not correspond to simple expressions, and, in particular, they need not be linear inequalities or equations (although they can be).

A constraint can affect any number of variables from 1 to n, where n is the number of variables in the problem. The number of affected variables is the *arity* of the constraint.

2.3. Example

Cryptarithmetic puzzles, like the following, can be expressed as CSPs. Each letter in the following sum stands for a different digit: find their values.

$$\begin{array}{r} \mathrm{D\ O\ N\ A\ L\ D} \\ +\mathrm{G\ E\ R\ A\ L\ D} \\ \hline =\mathrm{R\ O\ B\ E\ R\ T} \end{array}$$

The variables are the letters D, O, N, A, L, G, E, R, B, T and their domains are the set of digits $\{0,\ldots,9\}$ (except that D, G and R cannot be 0). The constraints are:

- the ten variables must all be assigned a different value;
- the sum given must work out:

$$\begin{aligned} &100\,000 * \mathrm{D} + 10\,000 * \mathrm{O} + 1000 * \mathrm{N} \\ &\qquad + 100 * \mathrm{A} + 10 * \mathrm{L} + \mathrm{D} \\ &+\ 100\,000 * \mathrm{G} + 10\,000 * \mathrm{E} + 1000 * \mathrm{R} \\ &\qquad + 100 * \mathrm{A} + 10 * \mathrm{L} + \mathrm{D} \\ &=\ 100\,000 * \mathrm{R} + 10\,000 * \mathrm{O} + 1000 * \mathrm{B} \\ &\qquad + 100 * \mathrm{E} + 10 * \mathrm{R} + \mathrm{T}. \end{aligned}$$

There are alternative ways of formulating this problem, which are better from the point of view of finding a solution. This will be discussed later.

3. Arc consistency

If all the constraints of a CSP are binary, i.e. they affect two variables, then the variables and constraints can be represented in a *constraint graph*: the nodes of the graph represent the variables and there is an edge joining two nodes if and only if there is a constraint between the corresponding variables.

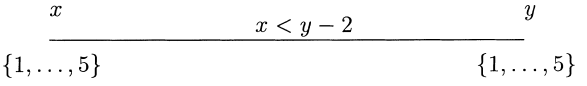
If there is a binary constraint C_{ij} between the variables x_i and x_j, then the directed arc (x_i, x_j) is *arc consistent* if for every value $a \in D_i$, there is a value $b \in D_j$ such that the assignments $x_i = a$ and $x_j = b$ satisfy the constraint C_{ij}. Any value $a \in D_i$ for which this is not true, i.e. no such value b exists, can safely be removed from D_i, since it cannot be part of any consistent solution: removing all such values makes the arc (x_i, x_j) arc consistent. The value $b \in D_j$ is a *supporting value* for $a \in D_i$; therefore, we delete any value $a \in D_i$ unless it has at least one supporting value in the domain of every variable x_k for which there is a binary constraint C_{ik}.

Fig. 1(a) shows the original domains of x and y. In (b), (x, y) has been made arc consistent; in (c), both (x, y) and (y, x) have been made arc consistent.

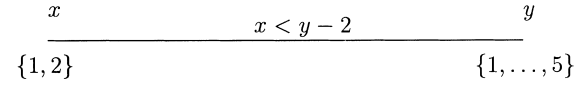
If every arc in a binary CSP is made arc consistent, then the whole problem is said to be arc consistent. Making the problem arc consistent is often done as a pre-processing stage: reducing the sizes of some domains should make the problem easier to solve.

A number of algorithms for making a CSP arc consistent have been proposed: a worst-case time complexity $O(d^2c)$ can be achieved, where d is the maximum domain size and c the number of binary constraints; this can be reduced to $O(dc)$ for many classes of constraint (Van Hentenryck et al., 1992a). Generalized arc consistency is an extension to constraints of higher arity: in general, it is too time-consuming to achieve generalized arc consistency, but for some types of constraint, specialized algorithms can achieve it reasonably efficiently. For instance, Régin (1994) presents such an algorithm for the 'all-different' constraint.

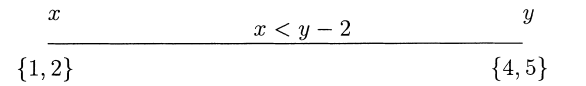
In some simple cases, the result of making a problem arc consistent is that every value left in the variable domains is part of a feasible solution to the CSP. In general this is not true, and a



(a) Original domains



(b) Domains when (x, y) arc consistent



(c) Domains when (x, y) and (y, x) arc consistent

Fig. 1. Creating arc consistency: (a) Original domains; (b) Domains when (x, y) arc consistent; (c) Domains when (x, y) and (y, x) arc consistent.

problem which is arc consistent and has at least one value remaining in every domain may still not have a feasible solution. Constraint programming algorithms usually employ a search procedure that enumerates assignments of values to variables. When the search fixes the value of a variable, *constraint propagation* is applied to restrict the domains of other variables whose values are not currently fixed.

4. Search algorithms

Most algorithms for solving CSPs search systematically through the possible assignments of values to variables. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem is unsatisfiable.

This paper is not intended to provide a comprehensive survey of constraint satisfaction algorithms. However, we present three systematic search algorithms. The first, a simple backtracking algorithm, is not used in practice, because in most cases it is very inefficient, but it is presented here for comparison with the more sophisticated algorithms. The second, forward checking, is proposed by Haralick and Elliott (1980), and the third, MAC (maintaining arc consistency), is developed by Sabin and Freuder (1994).

In all three algorithms, a search tree, similar to that used in a branch and bound algorithm, can be used to represent the current state of the search. Each node of the tree corresponds to a partial solution in which the values of some variables are determined; these are termed the *past* variables. The values of *future* variables remain to be decided. The branches of the tree correspond to the different possible values of some variable. By choosing a branch of the tree to search next, the algorithms *instantiate* a variable (i.e. they assign a value to the variable). A *deadend* is detected if the domain of a future variable becomes empty. The differences between the algorithms lie in their treatment of the future variables.

In the backtracking algorithm, the current variable is assigned a value from its domain. This assignment is then checked against the current partial solution; if any of the constraints between this variable and the past variables is violated, the assignment is abandoned and another value for the current variable is chosen. If all values for the current variable have been tried, the algorithm *backtracks* to the previous variable and assigns it a new value. If a complete solution is found, i.e. a value has been assigned to every variable, the algorithm terminates if only one solution is required, or continues to find new solutions. If there are no solutions, the algorithm terminates when all possibilities have been considered.

4.1. Forward checking

The backtracking algorithm only checks the constraints between the current variable and the past variables. On the other hand, forward checking and MAC are lookahead algorithms that check the constraints between the current and past variables and the future variables. In the forward checking algorithm, when a value is assigned to the current variable, any value in the domain of a future variable which conflicts with this assignment is (temporarily) removed from the domain. The advantage of this is that if the domain of a future variable becomes empty, it is known immediately that the current partial solution is inconsistent, and as before, either another value for the current variable is tried or the algorithm backtracks to the previous variable; the state of the domains of future variables, as they were before the assignment which led to failure, is restored. With simple backtracking, this failure would not have been detected until the future variable was considered, and it would then be discovered that none of its values are consistent with the current partial solution. Forward checking, therefore, allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking.

The forward checking algorithm is illustrated in Fig. 2 using the 6-queens problem. The n-queens problem requires placing n queens on an $n \times n$ chessboard in such a way that no queen can take any other: no two queens can be on the same row, the same column or the same diagonal of the board. The colours of the squares are irrelevant in

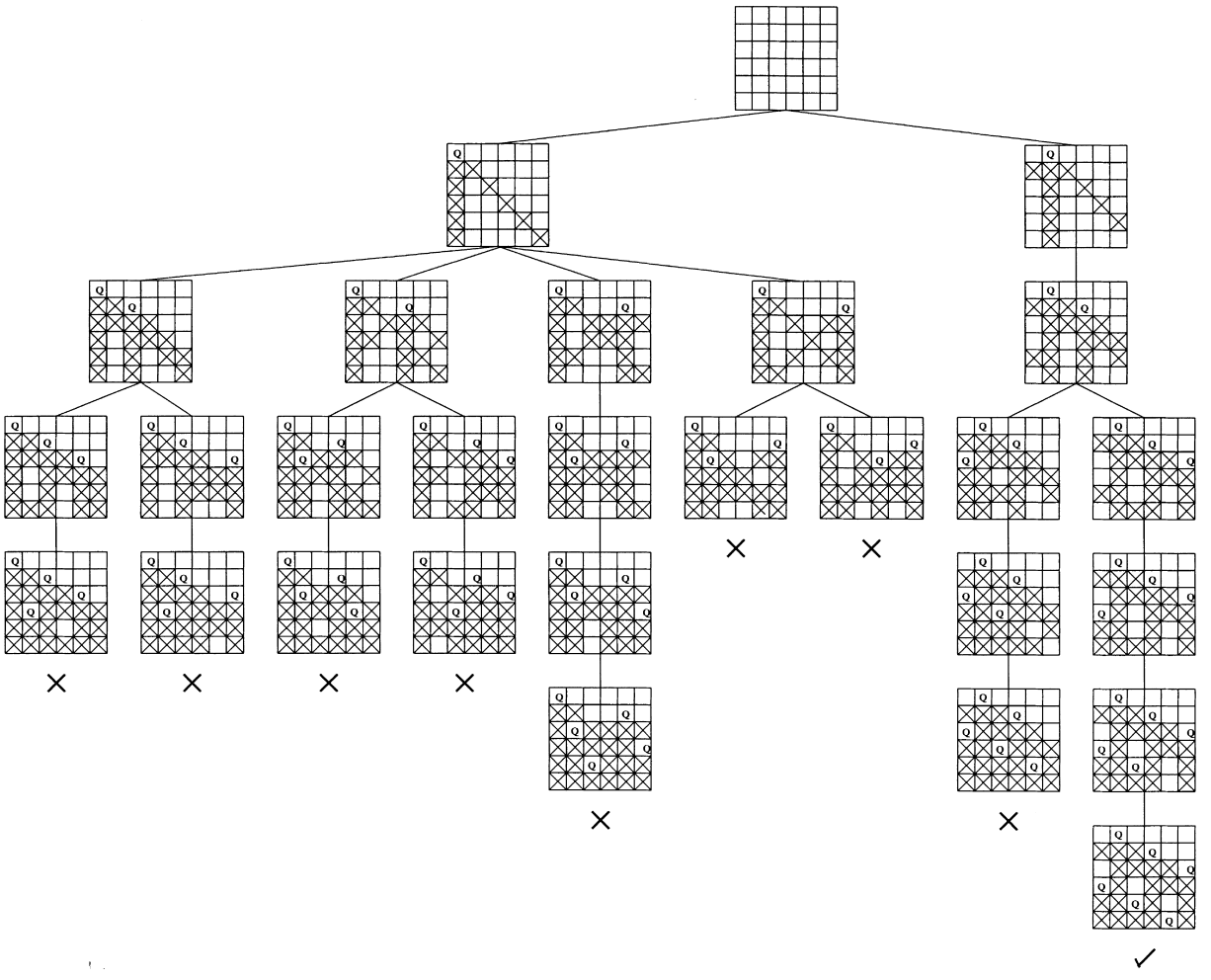


Fig. 2. Search tree for 6-queens using forward checking.

this problem, and are not shown. This problem has received a great deal of attention in CSP research, although it is in many ways not representative of the kind of problem met in practice. However, it has the merit that the state of the search at any point can be compactly represented, so that the progress of the search algorithms is easily shown.

As a CSP, this problem can be formulated using n variables, $v_1, \ldots, v_n$, corresponding to the rows of the chessboard, and each variable has domain $\{1, \ldots, n\}$ representing the column in which the queen is placed.

The full search tree built by forward checking for the 6-queens problem is shown in Fig. 2. The state of the problems after each variable instantiation is shown using a chessboard representation: a Q on a particular square should be taken as meaning that the variable for that row has been assigned the value corresponding to that column. If we start by placing a queen in the first column of the first row, then none of the other queens can be placed in the same column or on the same diagonal, and the values corresponding to the squares attacked by this queen can be removed from the domains of the variables for the queens in rows 2–6. Squares with crosses denote values removed from the domains of the variable corresponding to that row by the current or a previous assignment.

If a whole row of crosses occurs, then this means that a future variable has no remaining values, and the algorithm backtracks to a variable which still has an untried value. Such a deadend is marked in Fig. 2 by a cross below the board where an empty domain is encountered, and the solution eventually found is marked by a tick.

Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past assignments, so that checking an assignment for consistency with the current partial solution is no longer necessary. Forward checking does more work than simple backtracking when each assignment is added to the current partial solution, in order to reduce the size of the search tree and thereby reduce the overall amount of work done.

4.2. MAC

The MAC algorithm does still more work in looking ahead when an assignment is made. Whenever a new subproblem consisting of the future variables is created by a variable instantiation, the subproblem is made arc consistent. As well as checking the values of future variables against the current assignment, as forward checking does, MAC checks the future variables against each other. Thus, for each future variable, every value which is not supported in the domain of some other future variable is deleted, as well as those values which are not supported by the current assignment. This removes further values from the domains of future variables, and as with forward checking, the hope is that in doing additional work at the time of the assignment, there will be an overall saving in computation time.

Fig. 3 shows the effect of MAC on 6-queens. As before, squares marked with a cross are those which are eliminated by a previous assignment, or are inconsistent with the current assignment. Numbered squares are those which are eliminated in the course of making the subproblem arc consistent. The numbering indicates a possible order in which this can be done, since once some values are eliminated, this can trigger further deletions. The squares marked "1" are those which have no support once the values inconsistent with the current assignment are deleted. Squares marked "2" are those whose only supporting values were squares marked "1" and so on. For instance, in the leftmost board in Fig. 3, forward checking after the assignment of $v_2 = 3$ leaves v_3 with domain $\{5,6\}$; since the value 6 for v_4 is compatible with neither of these, it is eliminated. Similarly, $v_6 = 4$ is inconsistent with both of the possible values for v_4 remaining after forward checking (even before one of these values is deleted). Once 2 is the only possible value for v_4, any value in the domain of any future variable which is inconsistent with this is also eliminated and the square is marked "2". Finally, v_5 has domain $\{4\}$, so that the last remaining value for v_6 is eliminated as inconsistent with $v_5 = 4$, and it has been proved that no solution can be found from the first two assignments.

Note that the chain of inferences leading to an empty domain is not necessarily unique; if the algorithm continued, it would eliminate all the remaining values for all future variables, and the situation when the domain of some variable first becomes empty depends on the order in which values are deleted. On the other hand, if the subproblem can be made arc consistent without deleting all values, the resulting state does not depend on the order in which variables are considered.

The rightmost branch in Fig. 3 shows that there is only one possible solution following the first two assignments, and again reduces the amount of searching along this branch compared with Fig. 2.

The MAC algorithm interleaves constraint propagation and search. Re-establishing arc consistency after each variable instantiation can be done efficiently using an incremental arc consistency algorithm (Van Hentenryck et al., 1992a), rather than starting from scratch. Sometimes more elaborate consistency checks are also undertaken. An algorithm of this kind is generally used in constraint programming software.

5. Formulating problems

Although there may be an obvious way to formulate a problem as a CSP, there is often a

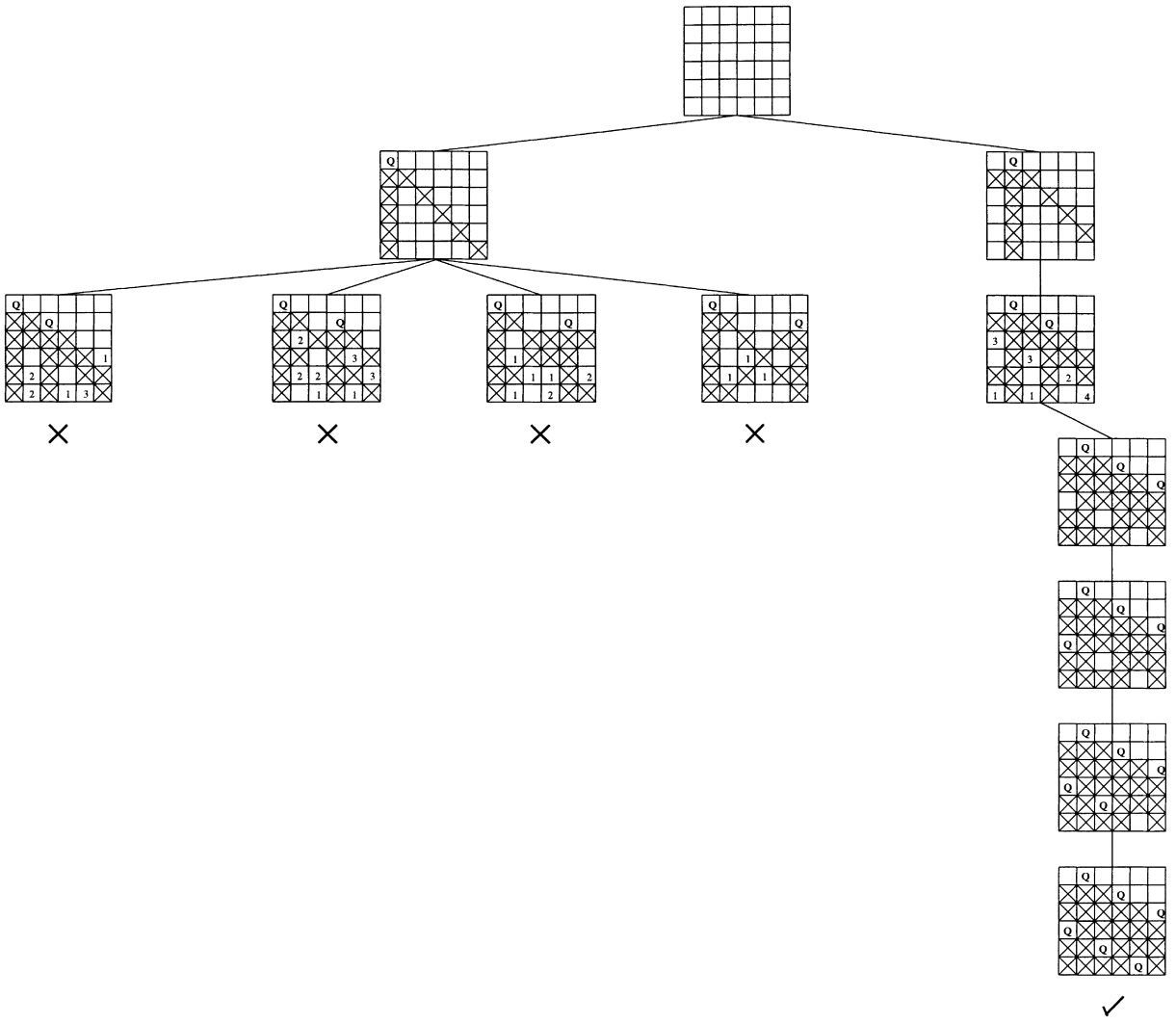


Fig. 3. Search tree for 6-queens using MAC.

choice of more than one formulation, with a little more thought.

Since arc consistency is a property associated with binary constraints, arc consistency algorithms can only be used to reduce the domains of variables involved in constraints which either are binary, or have all but two variables instantiated, so that the constraints have effectively become binary. Similarly, forward checking can only use those constraints in which exactly one variable remains to be instantiated when pruning the domains of the future variables. For these reasons, high arity constraints (i.e. constraints involving a large number of variables) are undesirable, and should if possible be avoided, except in cases where an efficient generalized arc consistency algorithm exists.

As an example, the cryptarithmetic puzzle given in Section 2.3 has a constraint involving all ten variables, which is not of much use for constraint propagation. The problem can be reformulated by adding extra variables $C1, C2, \ldots, C5$, each with domain $\{0, 1\}$, representing the quantities carried

from one column to the next as the addition is done. The constraint representing the sum can then be rewritten as

$$2 * D = 10 * C1 + T,$$
$$2 * L + C1 = 10 * C2 + R,$$
$$2 * A + C2 = 10 * C3 + E,$$
$$N + R + C3 = 10 * C4 + B,$$
$$O + E + C4 = 10 * C5 + O,$$
$$D + G + C5 = R,$$

where each of these six constraints contains at most five variables. This considerably speeds up the process of finding a solution.

Another technique is to incorporate redundant constraints that are implied by those already in the problem, but having fewer variables. An example occurs in the following problem: a number of cars requiring different options are to be assembled on a production line. Capacity constraints at the workstations on the production line prescribe constraints of the form "at most r cars out of any s consecutive cars can require option j". It is required to sequence the cars so that these constraints are satisfied. If too few cars requiring a particular option are used in one part of the sequence, than all the remaining cars requiring the option have to be fitted into the remainder with the result that the capacity constraints are violated. Hence, for each option, new constraints can be derived, that any sub-sequence of a certain length must contain *at least* a specified number of cars requiring that option. This car sequencing problem is discussed in more detail in Section 7.3.

Another consideration in formulating problems is that a few variables with large domains are in general preferable to many variables with small domains. In the former case, the initial size of the search space (i.e. before any pruning is done), which is given by the total number of possible instantiations of the variables, is much smaller, and hence usually the problem can be solved more quickly. In particular, zero–one variables are not desirable and should be avoided if possible.

As an example, the n-queens problem described earlier can alternatively be formulated using n^2 variables x_{ij}, where

$$x_{ij} = \begin{cases} 1 & \text{if there is a queen in row } i, \text{column } j, \\ 0 & \text{otherwise.} \end{cases}$$

The search space in this formulation is much larger than before: 2^{n^2}, as opposed to n^n. In addition, the new formulation requires a new set of constraints to represent the fact that there must be exactly one queen on each row:

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, \ldots, n.$$

Previously, this constraint is satisfied automatically: the domain of each variable represents the squares on the corresponding row, and in any solution exactly one value is assigned to the variable, i.e. exactly one square in the row is occupied.

Sometimes, a formulation with n variables, each with m values, can be transformed into m variables each with n values; if m is smaller than n this reduces the size of the search space. For instance, Dincbas et al. (1988a) use a reformulation to reduce the search space size from 4^{72} to 72^4.

In many problems, if solutions exist, there are classes of equivalent solutions. For instance, in timetabling problems, it may be possible to interchange the allocations to the time slots and still have a feasible solution; in rostering problems, a group of staff may have the same skills and the same availability, and therefore be interchangeable in the roster. Such *symmetries* in the problem may cause difficulties for a search algorithm: if the problem turns out to be unsatisfiable, or the algorithm is exploring a branch of the search tree which does not lead to a solution, then all symmetrical assignments will be explored in turn. This is a waste of effort, because if one such assignment is infeasible, then they all are. Such symmetries should be avoided, if possible, by including additional constraints in the formulation which allow only one solution from each class of equivalent solutions (Puget, 1993).

6. Variable and value ordering heuristics

The order in which variables are considered for instantiation has a dramatic effect on the time

taken to solve a CSP, as does the order in which each variable's values are considered. There are general principles which are commonly used in selecting the variable and value ordering, and a few specific heuristics.

The variable ordering may be either a *static* ordering, in which the order of the variables is specified before the search begins, and is not changed thereafter, or a *dynamic* ordering, in which the choice of next variable to be considered at any point depends on the current state of the search.

Dynamic ordering is not feasible for all tree search algorithms: for instance, with simple backtracking there is no extra information available during the search that could be used to make a different choice of ordering from the initial ordering. However, with forward checking and MAC, the current state includes the domains of the variables as they have been pruned by the current set of instantiations, and so it is possible to base the choice of next variable on this information. A common heuristic is to choose next the variable with smallest current domain. This is often explained as an implementation of the 'fail-first' principle (Haralick and Elliott, 1980) or as choosing the most constrained variable. As a tie-breaker, the most constraining variable can be chosen, for instance the one which constrains the largest number of future variables (in the absence of more specific information: for instance, on which constraints are likely to be difficult to satisfy).

Changing the value ordering produces a rearrangement of the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch which leads to a solution is searched earlier than branches which lead to deadends, provided that only one solution is required. If all solutions are required, or if the whole tree has to be searched because there are no solutions, then the order in which the branches are searched is immaterial. A good general principle is to choose a value which seems most likely to lead to a solution (if we can detect such a value), i.e. a 'succeed-first' principle. There are no cheap, generally applicable, dynamic heuristics implementing this principle, but in particular cases, problem-specific information may allow a heuristic to be devised.

7. Applications of constraint satisfaction

In this section, we present several types of OR problems, and describe how they can be tackled using constraint satisfaction approaches. Some of the studies that we review consider 'standard' models for which a variety of approaches have been adopted in the literature. Other work is of the 'case study' type for which some of the constraints are usually specific to the particular practical problem under consideration. All of the problems considered are known to be NP-complete (feasibility problems) or NP-hard (optimization problems).

7.1. Location

A well-known problem in OR involves the location of facilities, such as warehouses, to supply demand to customers. A set $\{1,\ldots,m\}$ of potential facility locations is given, and a fixed cost f_i is incurred if a facility is established at location i $(i=1,\ldots,m)$. There is a set of customers $\{1,\ldots,n\}$, and the cost of supplying customer j $(j=1,\ldots,n)$ from a facility at location i is c_{ij}. The location problem is to choose a subset of the potential locations at which to establish facilities, and then to assign each customer to one of these facilities, so that the total cost is minimized.

A standard zero–one programming formulation of this problem is as follows. By defining variables

$$y_i = \begin{cases} 1 & \text{if a facility is established at location } i, \\ 0 & \text{otherwise,} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{if customer } j \text{ is supplied from a facility} \\ & \text{established at location } i, \\ 0 & \text{otherwise,} \end{cases}$$

we obtain the formulation

$$\text{minimize} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij}x_{ij} + \sum_{i=1}^{m} f_i y_i$$

subject to

$$\sum_{i=1}^{m} x_{ij} = 1, \quad j = 1, \ldots, n,$$

$$y_i - x_{ij} \geqslant 0, \quad i = 1, \ldots, m, \qquad j = 1, \ldots, n,$$

$$x_{ij} \in \{0, 1\}, \quad i = 1, \ldots, m, \qquad j = 1, \ldots, n,$$

$$y_i \in \{0, 1\}, \quad i = 1, \ldots, m.$$

The equality constraints ensure that all customers are supplied from one of the facilities, while the inequality constraints prevent customers from being assigned to any location where a facility is not established. In this uncapacitated model, there is no limitation on the number of customers assigned to any facility that is established. Thus, when the facility locations are chosen, it is straightforward to assign each customer to a facility with the smallest supply cost.

Optimal solutions of large instances of the uncapacitated facility location problem can be obtained using branch and bound algorithms. The algorithm of Erlenkotter (1978) is widely regarded as being one of the most efficient.

Van Hentenryck and Carillon (1988) evaluate a constraint satisfaction approach. They use the variables y_i, as defined above, z_j which is the location of the facility that supplies customer j, where $z_j \in \{1, \ldots, m\}$, and v_j which is the supply cost for customer j, where $v_j \in \{c_{1j}, \ldots, c_{mj}\}$. The constraints are

$$v_j = c_{z_j, j}, \quad j = 1, \ldots, n,$$

$$y_i = 0 \Rightarrow z_j \neq i, \quad i = 1, \ldots, m, \quad j = 1, \ldots, n,$$

$$\sum_{i=1}^{m} f_i y_i + \sum_{j=1}^{n} v_j \leqslant C,$$

where C defines an upper bound on the objective function value.

For the first set of constraints, arc consistency is useful to restrict the domain of v_j when the domain of z_j changes, and vice versa. Similarly, after fixing $y_i = 0$, arc consistency in the second constraint allows the value i to be removed from the domains of $z_1, \ldots, z_n$. In an attempt to detect a deadend, the left-hand side of the third constraint is replaced by a lower bound in which each of the future variables v_j is replaced by the smallest value in its domain.

The computational evaluation of the constraint satisfaction approach appears to be strictly limited. Van Hentenryck and Carillon report that a problem with 21 potential facility locations and 80 customers is solved in 90 seconds on a SUN 3/160, and that the computation time is comparable to that required by a simple branch and bound algorithm with rather weak lower bounds. However, we conclude that a more sophisticated branch and bound algorithm such as that of Erlenkotter, where the lower bounds are computed quickly and are very effective in restricting the search, is far more efficient than this constraint satisfaction approach.

7.2. Scheduling

There are many problems arising in production industries that require jobs to be scheduled on machines. One of the most general and challenging is the job shop scheduling problem in which n jobs are to be scheduled on m machines. Each job j ($j = 1, \ldots, n$) comprises a set O_j of operations that must be executed in a specified order. The machine on which any operation o must be performed is m_o and the corresponding processing time is p_o. Since machine capacity constraints prevent any machine from processing more than one operation at the same time, a sequence of operations must be specified for each machine. A common objective is to find a schedule that minimizes the makespan, which is the completion time of the last job.

The job shop problem is notoriously difficult to solve. For example, there is an instance of Lawrence (1984) with 15 jobs and 10 machines that has not been solved with the currently available branch and bound algorithms. These algorithms suffer from the disadvantage that they employ lower bounds that are too weak to be effective in restricting the search. Local search heuristics (simulated annealing, tabu search and genetic algorithms) are successful in generating near-optimal solutions at moderate computational expense. Vaessens et al. (1996) review these heuristics and give a comparison of computational results.

Constraint satisfaction approaches often use start time variables s_o, where s_o is the time that the

processing of operation o starts. Clearly, $s_o \in \{0, 1, \ldots, C\}$, where C is an upper bound on the maximum completion time. If $o \in O_j$, let pred(o) and succ(o) denote the set of operations on job j that are predecessors and successors of o, respectively. Then the domain of s_o can be restricted so that $s_o \in \{\sum_{o' \in \mathrm{pred}(o)} p_{o'}, \ldots, C - p_o - \sum_{o' \in \mathrm{succ}(o)} p_{o'}\}$. If O denotes the set of all operations, then the constraints are

$$
\begin{aligned}
&s_o + p_o \leqslant s_{o'}, \quad o, o' \in O_j, \quad o' \in \mathrm{succ}(o), \\
&\quad j = 1, \ldots, n, \\
&s_o + p_o \leqslant C, \quad o \in O, \\
&s_o + p_o \leqslant s_{o'} \text{ or } s_{o'} + p_{o'} \leqslant s_o, \quad o, o' \in O, \\
&\quad o \neq o', \quad m_o = m_{o'}.
\end{aligned}
$$

The first set of constraints ensures that no operation on a job can start before all previous operations on that job are completed. The limit on the maximum completion time is enforced by the second set. The third set arises through the capacity of the machines: for two operations o and o' that require the same machine, either o is processed before o', or o' is processed before o.

Using this formulation, Nuijten and Aarts (1996) derive procedures for establishing arc consistency. For any operation o, let est_o denote its earliest start time, which is the smallest value in the domain of s_o. Similarly, we define lst_o to be the latest start time, which is the largest value in the domain of s_o. From the first constraint, operation o must complete no later than time $\mathrm{lst}_{o'}$ and operation o' can start no earlier than $\mathrm{est}_o + p_o$. Therefore, all values larger than $\mathrm{lst}_{o'} - p_o$ are removed from the domain of s_o, and all values smaller than $\mathrm{est}_o + p_o$ are removed from the domain of $s_{o'}$. Now suppose that $m_o = m_{o'}$, so that the third constraint applies. If o is scheduled before o', then $s_o \leqslant \mathrm{lst}_{o'} - p_o$; if o is scheduled after o', then $s_o \geqslant \mathrm{est}_{o'} + p_{o'}$. Therefore, if $\mathrm{lst}_{o'} - p_o + 1 \leqslant \mathrm{est}_{o'} + p_{o'} - 1$, then all values in $\{\mathrm{lst}_{o'} - p_o + 1, \ldots, \mathrm{est}_{o'} + p_{o'} - 1\}$ are deleted from the domain of s_o. An analogous update is made to the domain of $s_{o'}$.

In addition to achieving arc consistency, further constraint propagation is often useful. For example, Thuriot et al. (1994) propose a method for testing whether one operation must be scheduled before another operation that requires the same machine. Let $\overline{O}_k \subset O$ denote the set of operations that requires some machine k, and let $o, o' \in \overline{O}_k$. A lower bound on the amount of processing of operation $\overline{o}$, where $\overline{o} \in \overline{O}_k$, in any interval $[t_1, t_2]$ is given by

$$
\begin{aligned}
W(\overline{o}, t_1, t_2) = \min \{ & p_{\overline{o}}, t_2 - t_1, \\
& \max\{\mathrm{est}_{\overline{o}} + p_{\overline{o}} - t_1, 0\}, \\
& \max\{t_2 - \mathrm{lst}_{\overline{o}}, 0\}\}.
\end{aligned}
$$

By considering the amount of processing in the interval $[\mathrm{est}_o, \mathrm{lst}_{o'} + p_{o'}]$, we deduce that if

$$
\begin{aligned}
&\mathrm{est}_o + p_o + p_{o'} + \sum_{\overline{o} \in \overline{O}_k \setminus \{o, o'\}} W(\overline{o}, \mathrm{est}_o, \mathrm{lst}_{o'} + p_{o'}) \\
&\quad > \mathrm{lst}_{o'} + p_{o'},
\end{aligned}
$$

then operation o cannot precede o'. In this case, all values larger than $\mathrm{lst}_o - p_{o'}$ are removed from the domain of $s_{o'}$, and all values smaller than $\mathrm{est}_{o'} + p_{o'}$ are removed from the domain of s_o. Another useful constraint propagation device arises from the work of Carlier and Pinson (1989) on the development of a branch and bound algorithm. Let S be a set of operations requiring the same machine, and let $o \notin S$ be another operation requiring this machine. If

$$
\mathrm{est}_o + p_o + \sum_{o' \in S} p_{o'} > \max_{o' \in S} \{\mathrm{lst}_{o'} + p_{o'}\},
$$

then o cannot be scheduled before all operations in S. Therefore, all values smaller than $\min_{o' \in S}\{\mathrm{est}_{o'} + p_{o'}\}$ are removed from the domain of s_o. Symmetric arguments are used to obtain a further restriction on the domain of s_o when o cannot be scheduled after all operations in S. Tests of this type are often referred to as *edge finding*.

The usefulness of the constraint propagation methods described above is dependent on the availability of an efficient implementation. Baptiste and Le Pape (1995) review the methods and their implementations, and conclude that superior results are obtained by algorithms that employ edge finding methods.

Although there are various studies on constraint satisfaction for the job shop, the most thorough is undertaken by Nuijten and Aarts (1996). Their algorithm uses edge finding constraint propagation. They obtain lower and upper

bounds on the minimum makespan for 40 test problems of Lawrence (1984) and 3 test problems of Fisher and Thompson (1963) as follows. To find a lower bound, they use a bisection search to find the smallest value of C for which the problem is unsatisfiable, where zero and the sum of all processing times are initial lower and upper bounds on the makespan. The check for unsatisfiability is only performed through constraint propagation at the root node, with no variable instantiation. To find an upper bound, they perform a series of iterations with decreasing values of C, where C is set to be the minimum makespan found thus far less some decrement, and the decrement decreases in successive iterations. They adopt randomization in the selection of the operation start time to be instantiated next, and they restart the search after a specified number of backtracks. For 31 of the 43 test problems, an optimal solution is obtained, and the lower bound is exact for 23 of these 31 problems. The makespan found by the algorithm deviates by less than 1% from the best known lower bound for 35 test problems and by less than 3% for 40 test problems. Typical computation times are 1500 seconds on a SPARC-station ELC (for each of 5 independent runs) for problems with 15 jobs and 10 machines, and 2500 seconds for problems with 15 jobs and 15 machines.

Baptiste and Le Pape (1995) confirm the effectiveness of the edge finding approach of Nuijten and Aarts (1996) through an implementation in ILOG SCHEDULE, an add-on to ILOG SOLVER. Baptiste et al. (1995) further improve this approach. Specifically, they include the randomization and restart devices of Nuijten and Aarts in their heuristic algorithm for finding an upper bound. Moreover, at a restart, a pair of operations that are sequenced in adjacent positions on the same machine in the previous schedule are constrained to be sequenced in the same order with a certain probability, where this probability decreases at each restart. The algorithm is tested on 13 of the harder instances that are considered by Nuijten and Aarts. The makespan found by the algorithm deviates by less than 1% from the best known lower bound for 11 test problems; deviations are 1.54% and 5.84% for the two other problems. Typical computation times are 500 seconds on a RS6000 workstation (for each of 5 independent runs) for problems with 15 jobs and 10 machines, and 750 seconds for problems with 15 jobs and 15 machines. Baptiste et al. also use constraint satisfaction to obtain optimal solutions, where their heuristic algorithm is employed to obtain an initial upper bound. Results with this optimization algorithm are encouraging.

Cheng and Smith (1997) use an alternative formulation in which variables indicate the order between each pair of operations that require the same machine. This type of formulation is also used in several branch and bound algorithms. By limiting the amount of backtracking, they are able to obtain reasonable quality solutions to a selection of the test problems of Lawrence (1984) and Fisher and Thompson (1963) in under 10 seconds on a SPARC 10. They do not perform tests with longer run times, so it is impossible to assess whether their approach can compete in terms of solution quality with that of Baptiste et al. (1995).

7.3. Car sequencing

The car sequencing problem is one of the classical problems in the CSP literature, although it is not widely known in the OR community. The problem arises in the car production industry. After the basic model has been manufactured, various options (for example air-conditioning, metallic paint, ABS brakes, etc.) are added. Different versions of the car require different combinations of options. The cars are placed on a moving assembly line and pass through a number of workstation areas, where these options are installed. Depending on the total time the cars spend within each workstation area, and the set-up and installation times for each option, there are limitations on the rates at which the workstations can handle cars. These are expressed in the form of a ratio: "at most r out of every s consecutive cars can require this option". Given a set of N cars, each requiring a known set of options, the problem is to sequence the cars on the assembly line so that no workstation capacity is exceeded.

In reality, many other constraints in addition to the ratio capacity constraints are imposed. Some of these are described by David and Chew (1995).

These include *grouping constraints,* where it is desirable to group together cars requiring a particular process (for example painting, since the spray guns need to be purged each time the colour is changed). There may be *calendar constraints* on the start and end dates for particular vehicles, and *just-in-time constraints* that require production of each version of the vehicle to be distributed evenly over time, to reduce stocks. These clearly make the real-life problem more complex. Moreover, certain constraints may conflict with each other; the grouping and capacity constraints may require that some cars are simultaneously grouped together and spaced apart! In this case, a distinction is made between hard constraints (which must not be violated) and soft constraints, which are desirable but not essential. A weight is added according to the degree of preference for each soft constraint, and the problem then becomes one of optimizing over the preferences.

Consider the basic problem with the simple ratio constraints. If there are m options, let the workstation capacity for option j be such that at most r_j of every s_j cars can require option j ($j = 1, \ldots, m$). We divide the set of N cars up into n classes such that all the cars in a given class require the same set of options. Let a_i denote the number of cars in class i ($i = 1, \ldots, n$), where $\sum_{i=1}^{n} a_i = N$. For each class i and each option j, the data of the problem can be represented by constants δ_{ij}, where $\delta_{ij} = 1$ if class i needs option j, $\delta_{ij} = 0$ otherwise.

A zero–one programming formulation of this problem is obtained by defining variables

$$x_{ik} = \begin{cases} 1 & \text{if the car in position } k \text{ in the} \\ & \text{sequence is in class } i, \\ 0 & \text{otherwise.} \end{cases}$$

The constraints of the problem are

$$\sum_{k=1}^{N} x_{ik} = a_i, \quad i = 1, \ldots, n,$$

$$\sum_{i=1}^{n} \sum_{l=0}^{s_j - 1} \delta_{ij} x_{i,k+l} \leqslant r_j, \quad j = 1, \ldots, m,$$

$$k = 1, \ldots, N - s_j + 1,$$

$$x_{ik} \in \{0, 1\}, \quad i = 1, \ldots, n, \quad k = 1, \ldots, N.$$

The first set of constraints ensures that every car must appear somewhere in the sequence, and the second set prevents the workstation capacities from being exceeded. We may choose an arbitrary objective function since in this case we merely wish to find a feasible solution.

Dincbas et al. (1988b) use a constraint satisfaction approach that is based on the following formulation. As before, the set of N cars is partitioned into n classes according to their options, variables y_k are used to represent the class of the car assigned to position k in the sequence, where $y_k \in \{1, \ldots, n\}$, and variables z_{jk} are used to represent whether the car in position k requires option j, where $z_{jk} \in \{0, 1\}$. The constraints are

$$y_k = i \quad \text{and} \quad \delta_{ij} = 1 \Rightarrow z_{jk} = 1,$$

$$i = 1, \ldots, n, \quad j = 1, \ldots, m, \quad k = 1, \ldots, N,$$

$$|\{k | y_k = i\}| = a_i, \quad i = 1, \ldots, n,$$

$$\sum_{l=0}^{s_j - 1} z_{j,k+l} \leqslant r_j, \qquad j = 1, \ldots, m,$$

$$k = 1, \ldots, N - s_j + 1.$$

The first set of constraints links variables y_k and z_{jk}, the second set ensures that exactly a_i cars are of class i, and the third set comprises the ratio capacity constraints.

An alternative formulation has a variable for each car, each with domain $\{1, \ldots, N\}$ representing that car's position in the sequence; this is a kind of 'dual' formulation, where the variables become the values and vice versa. It has the disadvantage that many symmetrical solutions would be possible, and moreover the size of the search space would be N^N as opposed to n^N in the first formulation. However, as pointed out by Smith (1996), this dual formulation highlights the fact that in some CSP formulations the fail-first heuristic for variable ordering and succeed-first heuristic for value ordering, which are described in Section 6, may not necessarily be best.

Dincbas et al. (1988b) suggest introducing implied constraints, as described in Section 5. Clearly, if too few cars requiring a particular option are placed in one part of the sequence, so that all the remaining cars requiring that option have to be placed in the rest of the sequence, it will not be

possible to satisfy the capacity constraints. Thus, for each option, we can derive additional constraints which state that any subsequence of a certain length must contain at least a specified number of cars requiring that option.

Smith (1996) discusses the choice of heuristics for variable and value ordering. She argues that in this problem, and indeed in any problem where any possible solution is a permutation of a fixed set of values, it is better to assign the difficult cars first. The 'difficulty factor' should take into account both the option utilization and the total number of cars requiring that option: a '1 out of 5' option may appear harder to satisfy than a '1 out of 2', but if more than half the cars require the latter option, the problem would of course be infeasible.

Computational results for sets of about 100 randomly generated problems with between 5 and 200 cars and workstation utilization of 70–80% are reported by Van Hentenryck et al. (1992b). For less than 50 cars, the problems are solved in a few seconds on a Sun 3/160; for 100 cars, the solution time is less than a minute, but this rises to about 5 minutes for 200 cars. The differences in utilization do not greatly affect the solution time. They find empirically that the average solution time increases quadratically with the problem size. Smith (1996) compares different ordering heuristics for 20 problems involving 200 cars, and obtains solutions in just a few seconds for every feasible problem with at least one of the heuristics, although there is no overall winner.

David and Chew (1995) deal with a practical problem at Renault involving 7500 cars, each of which has 50–100 characteristics, incorporating both soft and hard constraints. They adopt a hybrid approach using simulated annealing in addition to CLP, and obtain good solutions in about 2 hours on a Sun Sparcstation II.

Régin and Puget (1997) introduce a generalized arc consistency algorithm for the sequencing constraints found in this problem, and report good results on a set of difficult 100-car problems.

7.4. Cutting stock

A cutting stock problem requires material to be cut into smaller pieces according to customer requirements, so that the waste is minimized. An example of a one-dimensional problem involves the cutting of metal rods. Cutting large wooden boards to produce the parts that are required for items of furniture is an example of a two-dimensional problem.

Consider a (two-dimensional) cutting stock problem in which m different shaped pieces are to be cut from stock, and the demand for piece i ($i = 1, \ldots, m$) is d_i units. There are n different cutting patterns, and cutting pattern j ($j = 1, \ldots, n$) yields q_{ij} units of piece i for $i = 1, \ldots, m$. A total of p cutting patterns are to be selected. The cost associated with cutting pattern j is c_j.

To obtain an integer programming formulation of this problem, we define variables x_j to be the number of times cutting pattern j is used. This yields the formulation

$$\text{minimize} \quad \sum_{j=1}^{n} c_j x_j$$

subject to

$$\sum_{j=1}^{n} q_{ij} x_j \geqslant d_i, \quad i = 1, \ldots, m,$$

$$\sum_{j=1}^{n} x_j = p,$$

$$x_j \in \{0, 1, \ldots, p\}, \quad j = 1, \ldots, n.$$

Note that this formulation exhibits a close resemblance with set covering.

For many practical problems, the number of cutting patterns is too large to allow all possibilities to be generated. This difficulty can be overcome by using a column generation approach. Alternatively, there is a wide variety of heuristics that can be employed to generate a selection of cutting patterns. Thus, research has not focused on developing efficient algorithms for solving the above set covering formulation. Nevertheless, algorithms for solving pure set covering problems are successful in solving large instances, and several of these approaches could be adapted for the formulation given above.

Dincbas et al. (1988a) propose a constraint satisfaction approach. They use variables z_k to represent the cutting pattern for the kth unit of

stock ($k = 1, \ldots, p$), where $z_k \in \{1, \ldots, n\}$, v_k to be the cost of cutting the kth unit of stock, where $v_k \in \{c_1, \ldots, c_n\}$, and u_{ik} to be the number of units of piece i ($i = 1, \ldots, m$) obtained from cutting the kth unit of stock, where $u_{ik} \in \{q_{i1}, \ldots, q_{in}\}$. The constraints are:

$$v_k = c_{z_k}, \quad k = 1, \ldots, p,$$

$$\sum_{k=1}^{p} u_{ik} \geqslant d_i, \quad i = 1, \ldots, m,$$

$$u_{ik} = q_{i,z_k}, \quad i = 1, \ldots, m, \quad k = 1, \ldots, p,$$

$$\sum_{k=1}^{p} v_k \leqslant C,$$

where C defines an upper bound on the objective function value.

Arc consistency in the first set of constraints restricts the domain of v_k when the domain of z_k changes, and vice versa. A similar observation applies to u_{ik} and z_k in the third set. To eliminate many of the symmetric solutions that result from interchanging z_k and z_l for $k \neq l$, the additional constraints $v_k \leqslant v_{k+1}$ for $k = 1, \ldots, p-1$ are imposed. Another feature of the approach used by Dincbas et al. (1988a) is a dichotomous search procedure. Rather than instantiating variables in the normal way, the median value of the current domain of some variable is computed, and then the search partitions the solutions according to whether the value of the variable is less than or equal to the median, or is greater than the median.

Dincbas et al. (1988a) give computational results for a problem with 72 cutting patterns from which 4 are to be selected, and 6 different shaped pieces are to be cut, thus giving $n = 72$, $p = 4$ and $m = 6$. Under the standard method of instantiating variables the problem is solved in 105 seconds on a VAX785. However, use of the dichotomous search allows the solution time to be reduced to 3 seconds.

Proll and Smith (1998) consider a template design problem, which can be viewed as a variant of the cutting stock problem. A template is essentially a cutting pattern, and each template has a given of slots to which any selection of pieces can be assigned. The objective is to minimize the number of different templates used, subject to constraints which limit the amount of under and over production relative to the demand. Proll and Smith propose a CSP formulation which includes variables y_{ij} that represent the number of units of piece i in template j. Computational results are provided for three problems, the largest of which has 40 template slots and 50 pieces, where at most 4 templates are used.

7.5. Vehicle routing

In the classical vehicle routing problem, there are n customers to be supplied from a single depot. Customer i ($i = 1, \ldots, n$) has a requirement of q_i, and each vehicle has a capacity Q (although in some variants of the problem capacities vary according to the vehicle). There is a cost c_{ij} and a time t_{ij} for travelling between each pair of customers i and j ($i = 0, 1, \ldots, n$, $j = 0, 1, \ldots, n$, $i \neq j$), where the depot is regarded as customer 0. It is required to find tours for the vehicles, where each tour starts and ends at the depot, so that each customer is visited once, the vehicle capacity constraints are satisfied, and the total cost is minimized. In a common variant of this basic model, there is a *time window* within which the delivery must take place.

Branch and bound algorithms for the vehicle routing problem have met with only limited success: obtaining optimal solutions for instances with more than 50 customers often requires excessive computation time. However, simulated annealing and tabu search algorithms are successful in generating solutions of very good quality. A review of these algorithms is given by Gendreau et al. (1997).

Christodoulou et al. (1994) propose a constraint satisfaction approach that is based on a standard zero-one programming formulation with variables

$$x_{ij} = \begin{cases} 1 & \text{if a vehicle travels directly from} \\ & \text{customer } i \text{ to customer } j, \\ 0 & \text{otherwise.} \end{cases}$$

The constraints are:

$$\sum_{j=0}^{n} x_{ij} = 1, \quad i = 1, \ldots, n,$$

$$\sum_{i=0}^{n} x_{ij} = 1, \quad j = 1, \ldots, n,$$

$$\sum_{j=1}^{n} x_{0j} - \sum_{i=1}^{n} x_{i0} = 0,$$

subtour elimination constraints,

vehicle capacity constraints.

The first two sets of constraints ensure that a vehicle travels from and travels to each customer. The third constraint forces all vehicles that leave the depot to return to the depot. The subtour elimination constraints prevent a solution in which the vehicle makes a subtour that does not include the depot (for example, the subtour defined by $x_{ij} = 1$, $x_{jk} = 1$ and $x_{ki} = 1$ for three distinct customers i, j and k). For the problem with time windows, additional variables representing the arrival time at each customer and the corresponding departure time are introduced. Each of these variables must lie within the time window for that customer.

After fixing $x_{ij} = 1$, arc consistency for the implied constraints $x_{ij} + x_{ik} \leqslant 1$ and $x_{hj} + x_{ij} \leqslant 1$, where $h \neq i$ and $j \neq k$, assigns the values $x_{ik} = 0$ and $x_{hj} = 0$. Similarly, if setting $x_{hk} = 1$ would create a subtour or would cause the vehicle capacity constraint to be violated, then the value $x_{hk} = 0$ is assigned.

The computational results obtained by Christodoulou et al. indicate that, without time windows, constraint satisfaction is inferior to other approaches. With the introduction of time windows, the domains of many of the x_{ij} variables are reduced to a single value of zero, and the time required to solve the problem is significantly reduced. The authors claim that problems with up to 50 customers can either be solved optimally, or a near-optimal solution is obtained. Since the precise characteristics of their test problems are not given, it is difficult to assess the significance of their results.

Shaw (1998) proposes a hybrid approach for vehicle routing in which local search is combined with constraint satisfaction techniques. More precisely, he considers a large neigbourhood in which several customers are removed from the current solution, where a heuristic selects the customers for removal, and are then reinserted. To find optimal or good reinsertions, a constraint satisfaction approach is used. Computational results for a variety of test problems, some having time windows, indicate that this hybrid approach is nearly competitive with the best available local search heuristics that are based on tabu search.

7.6. Timetabling

Timetabling problems arise in many forms, most usually in an educational context, and basically involve resource allocation. As a simple example of examination timetabling, we have a set of students, a set of examinations, a set of rooms and a set of available time periods, together with a list specifying the students who take each of the examinations. The objective is to allocate a room and a time to each examination, subject to various constraints. For example, clashes must be avoided by scheduling every pair of examinations to be taken by the same student at different times. Capacities of the rooms cannot be exceeded. Moreover, if two examinations are taken by the same set of students, they must be separated in time, in order to allow the students sufficient preparation time in between.

The most basic timetabling problem, which requires classes or examinations to be assigned to a minimum number of periods subject to the no-clash constraints, is NP-hard (Karp, 1972); it is reducible to graph colouring, and indeed this is the approach most frequently used to solve it. The computational study of Johnson et al. (1991) shows that optimal solutions cannot be obtained using reasonable computational resources for instances of the graph colouring problem with as few as 90 vertices. Although good quality solutions can be obtained with simulated annealing at large computational expense, heuristics with more modest computational requirements tend to generate solutions that are far from the optimum. For practical timetabling problems, heuristics are only moderately successful, because of the difficulty of

incorporating all the relevant factors appropriately into a single cost function. Another difficulty common to all solution approaches is that problems are often very sensitive to small changes in the data.

Nuijten et al. (1994) consider the examination timetabling problem as a special case of the resource constrained project scheduling problem. They apply the consistency checking techniques and randomization procedures that are developed for the constraint satisfaction approach to job shop scheduling (see Section 7.2), for a practical problem of examination timetabling at the Eindhoven University of Technology. The variables are the start times of all the operations, i.e. the examinations. The problem involves separation constraints, which ensure that particular examinations are sufficiently spaced in time, and no-clash constraints. Since students taking an examination need not all be assigned to the same room, the models only consider the overall available capacity of all the examination rooms.

Nuijten et al. (1994) consider two practical problems, involving 275 and 651 examinations respectively, with varying capacities, separation and no-clash constraints. They use two models: the first checks full arc-consistency only, whereas the second model transforms the separation constraints into capacity constraints in order to apply extensive consistency-checking techniques of the type described in Section 7.2. For each model, they also try a modified operation selection procedure, which gives preference to those examinations that are most difficult to assign. The second model outperforms the first for both problems, but interestingly the modified operation selection procedure improves the performance of the first model but not that of the second. Nuijten et al. conclude that anything less than full arc consistency checking degrades performance.

The problem of timetabling school lessons or university lectures is similar to examination timetabling but more complicated. Practical problems have many constraints other than the simple no-clash and capacity constraints described above. Lajos (1995) describes the problem of timetabling for the University of Leeds, where there are 1000 so-called 'offerings' (essentially, courses) comprising about 2500 classes per week. Additional constraints include:

- modular option groups, where some clashes are allowed since students chose a subset of options from a given set;
- time constraints (e.g., certain classes are constrained to a particular day because they are taught by external lecturers);
- smoothness constraints, i.e. reducing the maximum number of classes that can take place simultaneously in order to smooth out room utilization;
- spreading constraints, so that classes belonging to the same course are timetabled on separate days.

This problem is a good candidate for solution by constraint programming methods. Optimization is not always required; a feasible timetable is often the main objective, since the qualitative factors can usually be expressed as constraints. Constraint programming has the great advantage that its declarative nature allows the constraints to be expressed in a natural way, and lookahead algorithms are particularly effective for this problem in reducing the size of the search space. Nuijten et al. (1994) apply the extensive consistency-checking techniques described above for a school timetabling problem, with good results, and in this case too the modified operation selection procedure improves performance even further. Menezes and Barahona (1994) discuss the interaction between value and variable ordering heuristics and lookahead schemes for a timetabling problem at the New University of Lisbon.

A natural formulation is to represent each class as a variable whose domain is the set of available time periods. This is the formulation used by Lajos (1995) but others are possible; for example, blocking classes together and defining the variables to be the starting time of each block. This approach uses fewer variables, which in general is advantageous, but it greatly increases the complexity of the constraints. Lajos uses a fail-first variable-ordering heuristic, timetabling the most difficult offerings first, with some manual adjustment to deal with infeasibilities, and a random number generator for value ordering, which eliminates the need for spreading con-

straints and greatly improves performance. The program is written in PROLOG and solves typical instances of the problem in about 5 minutes on a Sun 10/20.

7.7. Rostering

Crew rostering problems are well known in OR, and have received considerable attention in the air and rail transport industries. Usually, the general problem of allocating staff to duties is divided into two distinct problems: the scheduling stage and the rostering stage. The scheduling stage involves designing trips (sometimes called tours of duty or rotations) where, in the case of airlines, all the timetabled flights are grouped into short sequences, which may occupy one or more days. For example, one such trip might be London → Paris, Paris → Rome, Rome → London. This can be formulated as a set-partitioning problem and solved using a branch and bound algorithm; however, there are many extra constraints governing the feasibility of a trip with respect to legal or trade union requirements on health and safety. Because of the huge problem size, heuristic methods have been developed, although in recent years advances in computer technology have enabled optimization techniques such as column generation to be applied.

The rostering stage consists of combining the tours of duty into rosters (known in the airline industry as lines of work) for the given planning period, which is usually one month. These rosters are then allocated to individual staff members or crews. In addition to further mandatory constraints on rosters, for example on the total number of rest days that a crew must have each month, there are also soft constraints expressing the preferences of the individual crews for the different rosters. Ryan (1992) formulates this problem as a generalized set-partitioning problem, where the variables are

$$x_{ij} = \begin{cases} 1 & \text{if crew } i \text{ is assigned to roster } j, \\ 0 & \text{otherwise.} \end{cases}$$

Because of the vast numbers of alternative rosters, real-life problems may potentially have hundreds of millions of variables. In fact, the main objective with practical problems may simply be to find a (legally) feasible roster, but other objective functions can be constructed which weight the soft constraints and minimize the deviation from the ideal.

Constraint programming has been used for the crew scheduling problem (see Guerinik and Van Caneghem, 1995) as well as the rostering problem. Caprara et al. (1998) solve a rostering problem for the Italian Railway Company, essentially adopting a CLP approach, but combined with an OR method, in that their system uses a lower bound obtained by a Lagrangian procedure. The railway company requires rosters in the form of cyclic sequences of duties representing a monthly working plan for a set of crews. The number of crews required for each roster is equal to the number of days in that roster. The principal objective is to find a feasible set of rosters minimizing the total number of crews required. A secondary objective is to minimize the total number of rest days between two duties in the same week, a technically allowable but undesirable feature in a roster.

The variables in the model of Caprara et al. are the duties, and the values they take represent the roster in which the duty is placed and its position in that roster. Additional variables associated with duties represented the preceding and succeeding duties and are used to express the constraints. The rosters are constructed sequentially on a succeed-first basis. This leads to a partial solution. The lower bound is then used to evaluate the number of remaining weeks to an optimal solution, and in turn this allows the difficulty of each constraint to be evaluated. This provides a heuristic for making the remaining assignments on a fail-first basis.

Caprara et al. implement their system on a Pentium 100 MHz and use the CLP language ECL^{i}PS^{e}; the lower bound procedure is written in C and linked to the CLP program. The computational results contrast the combined approach with a pure OR approach and are comparable in terms of quality and performance, although the CLP method is much quicker to develop and can more easily be modified to incorporate different constraints.

8. Evaluation of constraint satisfaction

In this section, we evaluate CP as a technique for solving CSPs, and compare it with OR methods. Two general comparative papers, written from an AI standpoint, are those of Van Hentenryck (1995) and Simonis (1996). The former considers various techniques for solving combinatorial search problems, such as branch and bound, branch and cut, and local search, and compares them with CP. However, this paper is descriptive rather than analytical. Simonis considers different application areas, and attempts to identify reasons for the failure of CP in some problems and to determine the critical issues involved. He stresses the important difference between end-user application development (for a client's day-to-day operational use) and theoretical studies to test a solver on one particular instance of a problem. Simonis concludes that there are four areas where CP is most successful; scheduling, allocation, transportation and rostering. One of the practical reasons for this success is the ease with which additional problem-specific constraints can be added without any need to revise the whole program. Another reason cited is the incorporation of search strategies based on OR methods!

In many cases, the evidence on which Van Hentenryck and Simonis base their conclusions is quite weak. Thus, we provide our own evaluation of CP in the following subsections.

8.1. Criteria for evaluation

There are various reasons why a particular technique may be chosen for a problem, including:

- ease of implementation;
- flexibility to handle a variety of constraints that occur in practical problems;
- computation time;
- solution quality.

Solution quality becomes important when the technique is not guaranteed to find a 'best' solution. For optimization problems, solution quality is the ability to generate a solution with a near-optimal objective function value, whereas for feasibility problems, it is the ability to find a solution in which most of the constraints are satisfied.

In practice, the interesting question is, "which technique should be chosen for a given combinatorial problem?" Obviously, CP is one possible technique. If a best solution is required, then an enumerative method such as branch and bound is also a candidate. The branch and bound algorithm may be a general solver for integer or mixed-integer programming problems, or a special-purpose algorithm with bounds and pruning devices developed for a specific problem. If the aim is to find an approximate solution, then the competitors of CP are special-purpose heuristics, and local search methods such as simulated annealing, tabu search and genetic algorithms.

For ease of implementation and flexibility, CP scores highly relative to most of the OR techniques that are mentioned above. Thus, it remains to compare CP and OR techniques in terms of computation time and solution quality. However, this task is quite difficult. The literature contains very few direct comparisons of CP with other approaches, and some of those that exist are open to criticism. For example, the range of instances in some comparisons is too small to form meaningful conclusions, and the OR techniques with which CP is compared are not always the best available. Many authors simply present solutions of problems using CP, and are naturally mainly interested in emphasizing the benefits of this technique. Moreover, when optimization problems are considered, it is not always clear whether the presented solutions are claimed to be optimal.

8.2. Comparison with integer programming and branch and bound

For many of the highly structured combinatorial problems, such as the uncapacitated facility location problem considered in Section 7.1, tight lower bounds (for minimization problems) can be generated at low computational expense. In such cases, special-purpose branch and bound algorithms are successful in solving quite large problem instances due to the ability of the lower bounds to

prune the search tree, and consequently CP approaches find it difficult to compete.

On the other hand, some problems such as job shop scheduling have resisted attempts to develop a tight lower bounding scheme. For such problems, we would expect CP approaches to be, at the very least, competitive with branch and bound. Results of Baptiste et al. (1995) for optimal solutions of job shop problems confirm the competitiveness of a CP approach.

Puget (1995) suggests that a critical factor in the success of a CP approach is the amount of propagation that the constraints permit; if the nature of a problem is such that a variable instantiation triggers the pruning of many values from the domains of other variables, then CP is likely to be successful. He argues that scheduling and timetabling problems are good candidates for CP.

As a general problem solver, CP should be compared with integer programming (IP) or mixed integer programming (MIP). IP and MIP approaches suffer from the disadvantage that the constraints must be linear, whereas the non-linearities that are permitted in a CSP sometimes allow the variables to be chosen in a way that allows a more structured search for an optimal solution. Van Hentenryck and Carillon (1988) emphasize the potential for CP in problems with a relatively small number of 'key' variables that effectively specify the solution (such as the variables y_i in the uncapacitated facility location problem described in Section 7.1). Instantiation of a key variable increases the potential for constraint propagation, whereas in an IP model the key variables are swamped numerically by the other variables. Little and Darby-Dowman (1995) emphasize the difficulty of comparisons when the performance of both techniques is highly variable and sensitive to data or problem size, but argue that an advantage of CP is that problem-specific features can be easily represented in a high-level language, whereas in IP they often hugely increase the size of the model.

The progressive party problem (Smith et al., 1996), where guests are to be assigned to groups for n successive time periods in such a way that no pair of people ever meets more than once, is a problem where CP succeeds very easily in finding a (feasible) solution, whereas IP does not. For this problem, capacity constraints on the size of the groups are very effective in propagating the results of assignments to the domains of other variables. Another reason for the lack of success of IP is the large number of 'all-different' constraints, leading to enormous formulations with many zero-one variables, whereas these constraints can be very compactly expressed using a CSP formulation. However, this problem is primarily of theoretical interest as a benchmark problem for CP solvers, and has few practical applications!

Rodosek et al. (to appear) present some empirical results for five specific problems (the progressive party problem, a generalized assignment problem, a packing problem, a set-partitioning problem, and a logical problem to show that it is not possible to put 10 pigeons into 9 pigeon-holes). They use the CP solver ECL^{i}PS^e and the MIP solver CPLEX. All of these problems involve all-different constraints and capacity constraints; the results show that MIP tends to outperform CP if the capacity limits are all equal, whereas CP is better if the capacities are different (as in the progressive party problem). Rodosek et al. present a hybrid approach, integrating CP and MIP, which gives better results than either individual method. However, it is not clear to what extent general conclusions can be drawn from this limited study.

Darby-Dowman and Little (1998) compare the performance of IP and CP on four problems; a golf scheduling problem that is similar to the progressive party problem, a crew scheduling problem (see Section 7.7), a production scheduling problem that requires jobs to be assigned to unrelated parallel machines, and an integer transportation problem in which the total supply at each source must be sent to a single destination. They conclude that, in general, CP performs better on problems that are highly constrained and therefore have a small search space, whereas IP is superior in problems with a large search space and no strong constraints. IP has the advantage of being able to detect global infeasibility. Darby-Dowman and Little argue that the linear relaxation is often of little help in zero–one problems, unless the problem has a particular structure, such as a coefficient matrix that is almost totally uni-

modular. In IP, model size is a key factor in performance, whereas in CP, the size of the search space is more important. The performance of CP can be greatly improved by the introduction of additional constraints (for example, symmetry constraints) and by the use of problem-specific information. However, this is not always straightforward in IP. Darby-Dowman and Little also discuss the potential for hybrid methods, but point out the difficulties of integrating two methods which use incompatible model and tree search structures.

The problem of compatibility is further discussed by Bockmayr and Kasper (1998), who propose a common framework for CP and IP called branch and infer. This involves extending IP by the introduction of symbolic constraints. Their hybrid method is superior to IP for instances of the uncapacitated facility location problem (see Section 7.1).

Proll and Smith (1998) present a comparison of IP and CP for a template design problem which has similarities to the cutting stock problem (see Section 7.4). Column generation, which is generally the method of choice for solving cutting stock problems, is not actually used in this study, although the IP approach bears a slight resemblance to it, since a population of potential templates (cutting patterns) is generated sequentially. Certainly, the IP solutions are not of such a high quality as the CP solutions. This problem clearly illustrates the advantages of being able to use nonlinear constraints, and incorporate 'human problem-solving ingenuity' in the solution strategy. However, for large instances, sophisticated refinements of the CP approach are required. The fact that CP solutions can easily be improved still further by employing a simple goal programming model to optimize the actual production quantities using specified templates, supports the hypothesis that using OR methods in conjunction with CP may well represent the most fruitful approach.

The car sequencing problem does not appear to have received much attention in the OR community. It is clear that the naive use of integer programming is unsuccessful in solving it. For example, the IP formulation of Section 7.3 does not succeed in finding a solution for a small test instance of the problem (with 25 cars, 5 options and 12 classes, giving 300 zero–one variables) in less than 5 minutes (on an IBM 486 DX PC), even after the addition of cuts based on the problem data. However, special-purpose algorithms have been developed for other sequencing problems for which IP is equally unsuccessful, and for this reason direct comparisons between IP techniques and CP may not be fair. Heuristic methods seem to be required for practical instances of the problem, but the pure problem may turn out to be amenable to an analytic approach.

Several of the papers mentioned here appear in a recent issue of the *INFORMS Journal on Computing* (10 (3), (1998)), which contains a cluster of articles on connections between the IP and CP approaches. These articles are introduced by Williams and Wilson (1998), who also present a brief summary of the two approaches and the links between them.

8.3. Comparison with local search heuristics

In its pure form, the CP approach is intended to search for a 'best' solution. To be used as a method for finding an approximate solution, modifications are necessary to ensure that the solution space is searched adequately. For example, the randomization and restart devices proposed by Nuijten and Aarts (1996) are introduced so that the search is diversified. It is unlikely that pure CP can compete with state-of-the-art implementations of local search methods.

Simulated annealing is compared with CP by Crabtree (1995) for a resource constrained scheduling problem, and by David and Chew (1995) for the car sequencing problem. In general, simulated annealing might intuitively be expected to perform better for problems with many solutions, whereas CP would be preferred for tightly constrained problems. However, Crabtree finds that in practice the reverse is true for the precedence constraints on the tasks; this suggests that further research is required. David and Chew describe some advantages of simulated annealing over CP; simulated annealing can deal with very large practical problems, it is possible to control the computation

time, it can handle 'soft' constraints, and it works blindly (heuristics for variable and value selection do not have to be found).

Vaessens et al. (1996) compare a variety of local search heuristics, and the CP approach of Nuijten and Aarts, for job shop scheduling. Since many computational studies are performed using the same set of test problems, solutions can be compared in terms of quality. Moreover, by applying normalizing coefficients to account for the speeds of the computers on which the tests are performed, the relative computation times for the different methods can be estimated. Clever implementations of tabu search, such as that of Nowicki and Smutnicki (1996), tend to generate the best quality solutions with relatively small computation times. However, the CP approach of Nuijten and Aarts gives better quality solutions than most of the simulated annealing methods and all of the genetic algorithms, although its computation times are larger. The improved CP approach of Baptiste et al. (1995) appears to be at least as good in terms of solution quality as all simulated annealing, tabu search and genetic algorithms, with the exception of Nowicki and Smutnicki's method, although CP is computationally expensive.

9. Concluding remarks

Our discussion in Section 8 shows that CP compares favourably with OR techniques in terms of ease of implementation and the flexibility to add new constraints. Its performance with respect to solution quality and computation time tends to be problem dependent. For a CP approach to work well, there should be a significant amount of constraint propagation: each variable that is instantiated should allow a reduction in the domains of other variables.

CP and branch and bound are both tree search techniques. While CP relies mainly on constraint propagation to restrict the size of the search tree, the efficiency of a branch and bound is highly dependent on the bounding scheme that is used. If the lower bounds (for minimization problems) require a significant amount of computation time and are not strong enough to allow very much pruning of the branch and bound search tree, then it is likely that CP would be more efficient. On the other hand, the availability of tight lower bounds, especially if their computational requirements are low, would suggest that a branch and bound algorithm is likely to outperform CP.

CP is unlikely to be competitive with the best local search methods, such as simulated annealing, tabu search and genetic algorithms, if it is used in a pure form, since large regions of the solution space are often unexplored. However, if ideas from local search are incorporated, such as the randomization and restart procedures of Nuijten and Aarts (1996), then CP becomes a serious competitor to local search for obtaining approximate solutions.

Although CP is still relatively in its infancy, whereas OR methods are often well developed and sophisticated, there are problems for which CP is competitive. Since further improvements to CP methodology are anticipated, we feel that the operational researcher should be aware of CP as a technique for tackling combinatorial optimization problems. Moreover, there is an increasing belief that hybrid methods often perform better than pure methods. A closer collaboration between the domains of AI and OR would benefit the development of algorithms in both disciplines. For example, CP algorithms could be improved by incorporating bounding schemes that are obtained using OR techniques. Similarly, if CP is to be used as a technique for obtaining approximate solutions, then ideas from local search should be incorporated. On the other hand, branch and bound algorithms may benefit from constraint propagation techniques to help prune the search trees, and, as observed by Shaw (1998), constraint satisfaction approaches can be employed to help search large neighbourhoods in local search heuristics.

Acknowledgements

The authors are grateful to Rachel Bothamley for collecting several of the references used in this paper, and to Philippe Baptiste for some comments on the performance of job shop scheduling algorithms.

References

Baptiste, P., Le Pape, C., 1995. A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In: Mellish, C.S. (Ed.), Proceedings of The 14th International Joint Conference on Artificial Intelligence. Morgan Kaufmann, Palo Alto, CA, pp. 600–606.

Baptiste, P., Le Pape, C., Nuijten, W.P.M., 1995. Constraint-based optimization and approximation for job shop scheduling. In: Sadeh, N. (Ed.), Proceedings of The AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems (IJCAI-95), http://www.ilog.com.sg/html/products/optimization/research_papers.htm.

Bockmayr, A., Kasper, T., 1998. Branch and infer: A unifying framework for integer and finite domain constraint programming. INFORMS Journal on Computing 10, 287–300.

Carlier, J., Pinson, E., 1989. An algorithm for solving the job-shop problem. Management Science 35, 164–176.

Caprara, A., Focacci, F., Lamma, E., Mello, P., Milano, M., Toth, P., Vigo, D., 1998. Integrating constraint logic programming and operations research techniques for the crew rostering problem. Software – Practice & Experience 28, 49–76.

Cheng, C.-C., Smith, S.F., 1997. Applying constraint satisfaction techniques to job shop scheduling. Annals of Operations Research 70, 327–357.

Christodoulou, N., Wallace, M., Kuchenhoff, V., 1994. Constraint logic programming and its application to fleet scheduling. Information and Decision Technologies 19, 135–144.

Crabtree, I.B., 1995. Resource scheduling – comparing simulated annealing with constraint programming. BT Technology Journal 13, 121–127.

Darby-Dowman, K., Little, J., 1998. Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. INFORMS Journal on Computing 10, 276–286.

David, J.-M., Chew, T.L., 1995. Constraint-based applications in production planning: examples from the automotive industry. In: Proceedings of Practical Applications of Constraint Technology (PACT'95). Practical Applications Company, Blackpool, UK, pp. 37–51.

Dincbas, M., Simonis, H., Van Hentenryck, P., 1988a. Solving a cutting-stock problem in constraint logic programming. In: Kowalski, R.A., Bowen, K.A. (Eds.), Logic Programming. MIT Press, Cambridge, MA, pp. 42–58.

Dincbas, M., Simonis, H., Van Hentenryck, P., 1988b. Solving the car-sequencing problem in constraint logic programming. In: Kodratoff, Y. (Ed.), Proceedings of European Conference on Artificial Intelligence (ECAI-88), Pitman, London, pp. 290–295.

Erlenkotter, D., 1978. A dual-based procedure for uncapacitated facility location. Operations Research 26, 992–1009.

Fisher, H., Thompson, G.L., 1963. Probabilistic learning combinations of local job-shop scheduling rules. In: Muth, J.F., Thompson, G.L. (Eds.), Industrial Scheduling. Prentice-Hall, Englewood Cliffs, NJ, pp. 225–251.

Gendreau, M., Laporte, G., Potvin, J.-Y., 1997. Vehicle routing: modern heuristics. In: Aarts, E.H.L., Lenstra, J.K. (Eds.), Local Search in Combinatorial Optimization. Wiley, Chichester, UK, pp. 311–336.

Guerinik, N., Van Caneghem, M., 1995. Solving crew scheduling problems by constraint programming. In: Montanari, U., Rossi, F. (Eds.), Proceedings of The First Conference of Principles & Practice of CP, Lecture Notes in Computer Science, vol. 976. Springer, Berlin, pp. 481–498.

Haralick, R., Elliott, G., 1980. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence 14, 263–313.

Johnson, D.S., Aragon, C.R., McGeoch, L.A., Schevon, C., 1991. Optimization by simulated annealing: An experimental evaluation; Part II, graph coloring and number partitioning. Operations Research 39, 378–406.

Karp, R.M., 1972. Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (Eds.), Complexity of Computer Computations. Plenum Press, New York, pp. 85–103.

Lajos, G., 1995. Complete university modular timetabling using constraint logic programming. In: Burke, E., Ross, P. (Eds.), First International Conference on Practice and Theory of Automated Timetabling, Lecture Notes in Computer Science, vol. 1153. Springer, Berlin, pp. 146–161.

Lawrence, S., 1984. Resource constrained scheduling: An experimental investigation of heuristic scheduling techniques. Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA.

Little, J., Darby-Dowman, K., 1995. The significance of constraint logic programming to operational research. In: Lawrence, M., Wilsdon, C. (Eds.), Operational Research Society Tutorial Papers 1995. Operational Research Society, Birmingham, UK, pp. 20–45.

Mackworth, A.K., 1977. Consistency in networks of relations. Artificial Intelligence 8, 99–118.

Menezes, F., Barahona, P., 1994. Heuristics and look-ahead integration to solve constraint satisfaction problems efficiently. Annals of Operations Research 50, 411–426.

Montanari, U., 1974. Networks of constraints: fundamental properties and applications to picture processing. Information Science 7, 95–132.

Nowicki, E., Smutnicki, C., 1996. A fast taboo search algorithm for the job shop problem. Management Science 42, 797–813.

Nuijten, W.P.M., Aarts, E.H.L., 1996. A computational study of constraint satisfaction for multiple capacitated job shop scheduling. European Journal of Operational Research 90, 269–284.

Nuitjen, W.P.M., Gunnen, G.M., Aarts, E.H.L., Dignum, F.P.M., 1994. Examination time tabling: A case study for constraint satisfaction. In: Cohn, A.G. (Ed.), Proceedings of Workshop on Constraint Satisfaction Issues Raised by Practical Applications (ICAI'94), pp. 11–19.

Proll, L., Smith, B.M., 1998. Integer linear programming and constraint programming approaches to a template

design problem. INFORMS Journal on Computing 10, 265–275.

Puget, J.-F., 1993. On the satisfiability of symmetrical constrained satisfaction problems. In: Komorowski, J., Ras, Z.W. (Eds.), Proceedings of Seventh International Symposium on Methodologies for Intelligent Systems (ISMIS'93), Lecture Notes in Computer Science/Lecture Notes in Artificial Intelligence, vol. 689. Springer, Berlin, pp. 350–361.

Puget, J.-F., 1995. A comparison between constraint programming and integer programming. In: Conference on Applied Mathematical Programming and Modelling (APMOD95), Brunel University.

Régin, J.-C., 1994. A filtering algorithm for constraints of difference in CSPs. In: Proceedings of Twelfth National Conference on Artificial Intelligence 1 (AAAI-94). MIT Press, Cambridge, MA, pp. 362–367.

Régin, J.-C., Puget, J.-F., 1997. A filtering algorithm for global sequencing constraints. In: Smolka, G. (Ed.), Principles and Practice of Constraint Programming – CP97, Lecture Notes in Computer Science, vol. 1330. Springer, Berlin, pp. 32–46.

Rodosek, R., Wallace, M.G., Hajian, M.T., to appear. A new approach to integrate mixed integer programming with CLP. Annals of Operations Research.

Ryan, D., 1992. Solution of massive generalized set partitioning problems in aircrew rostering. Journal of the Operational Research Society 43, 459–467.

Sabin, D., Freuder, E.C., 1994. Contradicting conventional wisdom in constraint satisfaction. In: Cohn, A.G. (Ed.), Proceedings of European Conference on Artificial Intelligence (ECAI-94). Wiley, Chichester, UK, pp. 125–129.

Shaw, P., 1998. Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.-F. (Eds.), Principles and Practice of Constraint Programming – CP98, Lecture Notes in Computer Science, vol. 1520. Springer, Berlin, pp. 417–431.

Simonis, H., 1996. A problem classification scheme for finite-domain constraint solving (tutorial paper given at PACT'96). In: Second International Conference on the Practical Applications of Constraint Technology.

Smith, B.M., 1996. Succeed-first or fail-first: A case study in variable and value ordering heuristics. In: Proceedings of The Third International Conference on the Practical Applications of Constraint Technology (PACT'97). Practical Applications Company, Blackpool, UK, pp. 321–330.

Smith, B.M., Brailsford, S.C., Hubbard, P.M., Williams, H.P., 1996. The progressive party problem: Integer linear programming and constraint programming compared. Constraints 1, 119–136.

Thuriot, C., Ershler, J., Lopez, P., 1994. Decision approach to workload distribution. Production Planning & Control 5, 533–542.

Van Hentenryck, P., 1995. Constraint solving for combinatorial search problems: A tutorial. In: Montanari, U., Rossi, F. (Eds.), Proceedings of The First Conference of Principles & Practice of CP, Lecture Notes in Computer Science, vol. 976. Springer, Berlin, pp. 564–587.

Van Hentenryck, P., Carillon, J.-P., 1988. Generality versus specificity: An experience with AI and OR techniques. In: Proceedings of The Seventh National Conference on Artificial Intelligence 2 (AAAI 88). AAAI Press/MIT Press, Cambridge, MA, pp. 660–664.

Van Hentenryck, P., Deville, Y., Teng, T.-M., 1992a. A generic arc-consistency algorithm and its specializations. Artificial Intelligence 57, 291–321.

Van Hentenryck, P., Simonis, H., Dincbas, M., 1992b. Constraint satisfaction using constraint logic programming. Artificial Intelligence 58, 113–159.

Vaessens, R.J.M., Aarts, E.H.L., Lenstra, J.K., 1996. Job shop scheduling by local search. INFORMS Journal on Computing 8, 302–317.

Waltz, D., 1972. Generating semantic descriptions from drawings of scenes with shadows. Technical Report AI271. MIT, MA.

Williams, H.P., Wilson, J.M., 1998. Connections between integer linear programming and constraint logic programming – an overview and introduction to the cluster of articles. INFORMS Journal on Computing 10, 261–264.