University of Leeds

SCHOOL OF COMPUTER STUDIES

RESEARCH REPORT SERIES

Report 95.14

A Tutorial on Constraint Programming

by

Barbara M. Smith

Division of Artificial Intelligence

April 1995

Abstract

A constraint satisfaction problem (CSP) consists of a set of variables; for each variable, a finite set of possible values (its domain); and a set of constraints restricting the values that the variables can simultaneously take. A 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. Many problems arising in O.R., in particular scheduling, timetabling and other combinatorial problems, can be represented as CSPs. Constraint programming tools now exist which allow CSPs to be expressed easily, and provide standard strategies for finding solutions. This tutorial is intended to give a basic grounding in constraint satisfaction problems and some of the algorithms used to solve them, including the techniques commonly used in constraint programming tools. In particular, it covers arc and path consistency; simple backtracking and forward checking, as examples of search algorithms; and the use of heuristics to guide the search. A simple example is considered in detail to show the effect of different choices of formulation and search strategy. Finding optimal solutions to CSPs is also discussed.

1 Introduction

Constraint satisfaction problems (CSPs) have been a subject of research in A.I. for many years. It has been recognised that CSPs have practical significance because many problems arising in O.R., in particular scheduling, timetabling and other combinatorial problems, can be represented as CSPs. This tutorial is intended to introduce the research into CSPs which is necessary to make use of constraint programming tools; it is not intended to be a comprehensive survey of the field.

Briefly, a constraint satisfaction problem (CSP) consists of:

- a set of *variables* X $= \{x_1, ..., x_n\}$;
- for each variable x_i, a finite set D_i of possible values (its *domain*);
- and 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.

A 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. 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.

As described below, solutions to CSPs are found by searching systematically through the possible assignments of values to variables, usually guided by heuristics. The reasons for choosing to represent and solve a problem as a CSP rather than, say, as a mathematical programming problem are two-fold. Firstly, the representation is often much closer to the original problem: the variables of the CSP directly correspond to problem entities, and the constraints can be expressed without having to be translated into linear inequalities. This makes the formulation simpler, the solution easier to understand, and the choice of good heuristics to guide the solution strategy more straightforward. Secondly, although CSP algorithms are essentially very simple, they can sometimes find solutions more quickly than if integer programming methods are used.

In recent years, with the development of constraint programming tools, it has become easier to express and solve constraint satisfaction problems. The first commercial constraint programming tools were those based on extensions of Prolog, in particular CHIP (Constraint Handling in Prolog). Currently, ICL's DecisionPower incorporates CHIP, and it is also available from Cosytec. PrologIII is an alternative constraint programming language based on Prolog, available from Prologia. Constraint logic programming (i.e. the extension of logic programming languages like Prolog to support the handling of constraints) is now a significant area of research in the logic programming community.

Logic programming is not, however, an essential basis for constraint programming, and other tools have been built, notably Solver from ILOG, which is a C++ library. (It is worth mentioning that the French computer company Bull, which had its own constraint programming tool called Charme, developed from CHIP, has now abandoned this in favour of a version of ILOG Solver.) Where necessary, examples in this tutorial will refer to ILOG Solver, which is the tool that I am most familiar with. However, examples could equally well have been presented in terms of CHIP or another tool.

2 Constraints

The constraints of a CSP are usually represented by an expression involving the affected variables, e.g.

$$x_1 \neq x_2$$
$$2x_1 = 10x_2 + x_3$$
$$x_1 x_2 < x_3$$

or, in ILOG Solver,

```
CtNeq(x1,x2);
CtEq(2*x1, 10*x2 + x3);
CtLt(x1 * x2, x3);
```

It is useful also to be aware of the formal definition of a constraint: a possible constraint $C_{ijk...}$ between the variables $x_i, x_j, x_k, ...$ is any subset of the possible combinations of values of $x_i, x_j, x_k, ...$[1]. 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 $x = y$ would be represented by the subset {(1,1), (2,2)}.

Although the constraints of real problems are not represented this way in practice, the definition does emphasise 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 (n is the number of variables in the problem). The number of affected variables is the *arity* of the constraint. It is useful to distinguish two particular cases:

Unary constraints affect just one variable. The constraint can be used to remove any value which does not satisfy the constraint from the domain of the variable at the outset. For instance, if there is a constraint $x_1 \neq 1$, the value 1 can be removed from the domain of x_1, and the constraint will then be satisfied. Since unary constraints are dealt with by preprocessing the domains of the affected variable, they can be ignored thereafter.

Binary Constraints affect two variables. If all the constraints of a CSP are binary, 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.

[1]The possible combinations of values of $x_i, x_j, x_k, ...$ are given by the cross-product of the domains, $D_i \times D_j \times D_k$....., so $C_{ijk...}$ is a possible constraint if:

$$C_{ijk...} \subseteq D_i \times D_j \times D_k \times .....$$

In fact, any constraint of higher arity can be expressed in terms of binary constraints, although in practice this is not likely to be worth doing. Hence, in some sense, binary CSPs are representative of all CSPs.

An individual binary constraint between two variables with domain sizes m_1 and m_2 can be represented by an $m_1 \times m_2$ matrix of 0-1 values, where 1 signifies that the constraint allows the corresponding pair of values, and 0 that it does not. Although this is not a practical way of defining real constraints, it means that a random 0-1 matrix can represent a binary constraint, and this allows experimenters who need a large number of CSPs to produce randomly-generated problems. (The pairs of variables which have constraints between them are also selected randomly. [9] describes in more detail two possible models for randomly-generated CSPs.) A great deal of experimental work on CSPs, comparing the performance of different algorithms, for instance, has been done using populations of randomly-generated binary CSPs.

2.1 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.

```
     D O N A L D
   + G E R A L D
  ---------------
  =  R O B E R T
```

The variables are the letters D, O, N, A, L, G, E, R, B, T and their domains are the set of digits {0 .. 9} (except that D, G and R cannot be 0). This expressed in ILOG Solver by creating a set of constrained integer variables:

```
CtIntVar D(1,9),O(0,9),N(0,9),A(0,9),L(0,9),
         G(1,9),E(0,9),R(1,9),B(0,9),T(0,9);
```

It is convenient to form an array containing pointers to these variables:

```
CtIntVar* AllVars[]
          ={&D,&O,&N,&A,&L,&G,&E,&R,&B,&T};
```

The constraints are:

- the ten variables must all be assigned a different value. Technically, this means that there is a 'not-equal' constraint between every pair of variables, giving 45 binary constraints. In practice, tools such as ILOG Solver allow an easy way of stating that all variables in a specified set must set have different values:

  ```
  CtAllNeq(10, AllVars);
  ```

- the sum given must work out:

  ```
  CtEq(100000*D+10000*O+1000*N+100*A+10*L+D
    +  100000*G+10000*E+1000*R+100*A+10*L+D,
       100000*R+10000*O+1000*B+100*E+10*R+T);
  ```

(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 there is a binary constraint C_{ij} between the variables x_i and x_j then the arc[2] (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. Note that we have only checked the values of x_i; there may still be values in the domain of x_j which could be removed if we reverse the operation and make the arc (x_j, x_i) arc consistent.

Figure 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.

(Note that if a binary constraint is represented by a matrix, in the manner described earlier, making the constraint arc consistent in both directions effectively removes any values from the domains of the two variables for which the corresponding row or column in the matrix is all zeros.)

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

[2]The *arc* (x_i, x_j) has a direction attached to it, so that it is distinct from the arc (x_j, x_i). The *edge* joining x_i and x_j, on the other hand, is undirected.

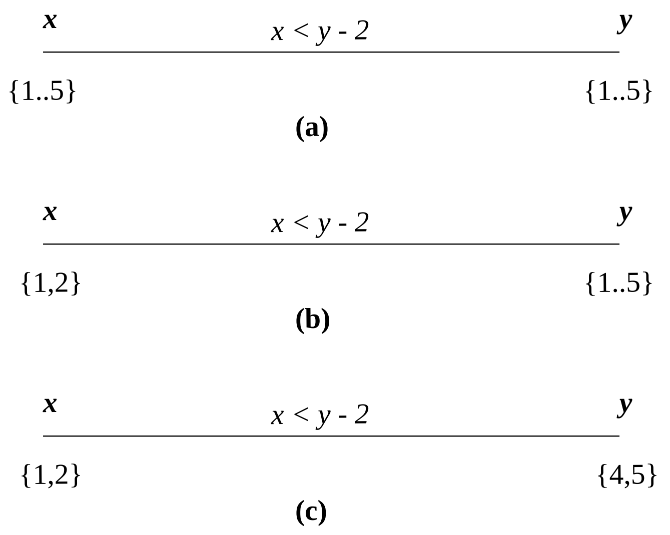

Figure 1: Making (x, y) and (y, x) arc consistent

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: the most commonly-used ones appear to be AC-3 and AC-4 (although variations up to at least AC-7 exist, each supposedly better than its predecessors in some way). AC-4 has worst case time complexity $O(d^2c)$ where d is the maximum domain size and c is the number of binary constraints; the space complexity is similarly $O(d^2c)$.

Since it is reasonably cheap to make a problem (or just the arcs corresponding to binary constraints) arc consistent, constraint programming tools normally include an arc consistency algorithm. In ILOG Solver, for instance, arc consistency is established as constraints are defined and maintained whenever any change is made, e.g. to the domains of variables. This is done automatically.

Example. Suppose we have a set of activities, each with a specified duration. There are precedence constraints between the activities, so that if task A precedes task B, then:

$$\text{startA} + \text{durationA} \leq \text{startB}$$

Task	Duration	Precedes
A	3	B,C
B	2	D
C	4	D
D	2	

We can express this as a binary CSP by introducing variables representing the start time of each activity, and others representing the start and finish of the project. Since the project need not take longer than the total duration of all the activities (11), this gives a possible limit for each variable.

Variable	Initial domain
start	$\{0\}$
startA	$\{0..11\}$
startB	$\{0..11\}$
startC	$\{0..11\}$
startD	$\{0..11\}$
finish	$\{0..11\}$

Arc consistency reduces the domains of the variables as shown below. Any value remaining in any variable's domain is part of a consistent solution to the whole problem, which will allow the project to finish by time 11, as shown in Figure 2.

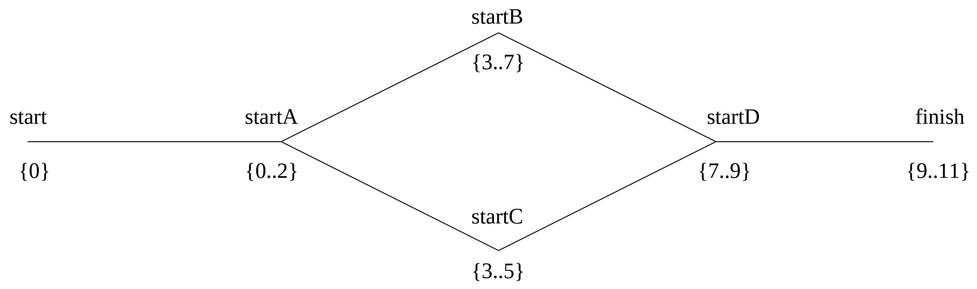

Figure 2: Simple project network

Note that if we want to minimise the project completion time and now set `finish` to its minimum possible value, further values will be removed from the domains. This is equivalent to the Critical Path Method.

Hence in a project planning problem with precedence constraints only (and no resource constraints, for example), arc consistency is sufficient by itself to remove all values from the domains which cannot be part of any solution. This is not generally true, and in order to find a solution, if there is one, we need to search for one.

4 Path Consistency and Beyond

In Figure 3, the problem is arc consistent, but it is clear that the variable x cannot have the value 2. In general, even when a problem has been made

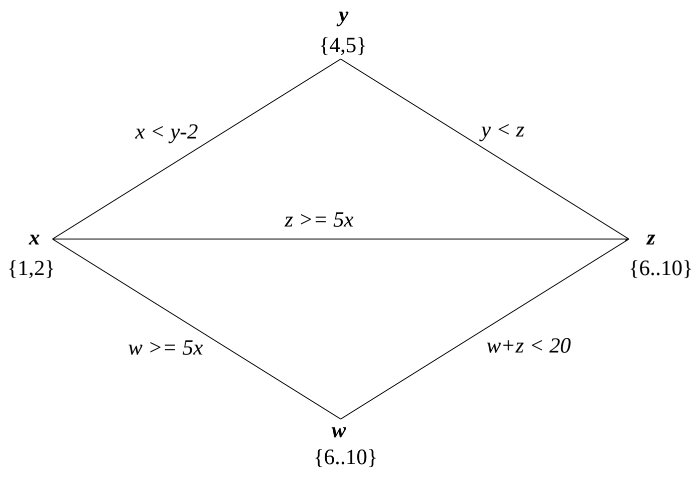

Figure 3: An arc consistent constraint network

arc consistent, it is possible to make further deductions from the constraints, short of searching for a complete solution. The next step (still with binary constraints) is to consider triples of variables, in which two pairs of variables have a non-trivial constraint between them. (A trivial constraint here is one that allows every pair of values.)

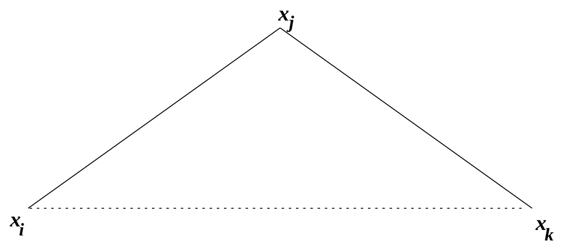

Figure 4: A triangle of constraints

In Figure 4, suppose there are non-trivial constraints between x_i and x_j, and between x_j and x_k.

The path (x_i, x_j, x_k) is path consistent iff for every pair of values $v_i \in D_i$ and $v_k \in D_k$ allowed by the constraint C_{ij} there is a value $v_j \in D_j$ such that $(v_i, v_j) \in C_{ij}$ and $(v_j, v_k) \in C_{jk}$. If there is no such value v_j then (v_i, v_k) should be removed from the constraint C_{ik}, i.e. the constraint should be tightened. In other words, if no value can be found for x_j which is simultaneously consistent with $x_i = v_i$ and $x_k = v_k$, then we cannot allow these values to be simultaneously assigned to x_i and x_k.

In the example of Figure 3, making (x, w, z) path consistent would show that the constraint between x and z has to be tightened to exclude the simultaneous assignment of $x = 2$ and $z = 10$; making the problem arc consistent again would show that the value 2 must be removed from the domain of x. So in this case, path consistency would allow the problem to be considerably simplified.

It is, of course, possible to check every triple of variables in a binary CSP and tighten the constraints where possible. Clearly, the number of possible triples is greater than the number of pairs of variables that need to be checked to make the problem arc consistent, and the best algorithm has worst case time complexity $O(d^3n^3)$. This is one reason why path consistency algorithms are not in common use. Another is that since constraints are not generally expressed as allowed tuples of values, it is not easy to remove individual pairs of values in order to tighten a binary constraint.

In practice, the constraints required for path consistency are often easy to see when the problem is formulated. For instance, in the triangle of constraints in Figure 5, path consistency shows that the constraint between x and z must be $x \neq z$; a constraint such as this, which must effectively exist, but may not be specified in the original statement of the problem, is called an *induced* constraint. (See [8] for further discussion of this.)

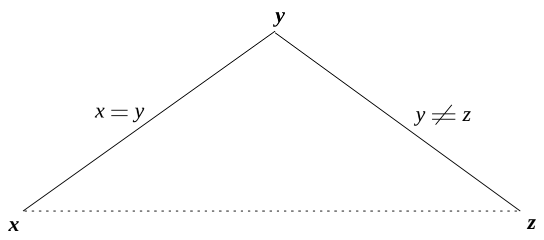

Figure 5: Example of an induced constraint between x and z

It is also possible to consider groups of 4 or more variables and attempt to induce new constraints (which will in general be non-binary), but this is still more time-consuming: in practice, preprocessing rarely goes beyond making a problem arc consistent.

5 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 insoluble, but they may take a very long time to do so.

This paper is not intended to be a comprehensive survey of constraint satisfaction algorithms. I shall consider only two systematic search algorithms:

a simple backtracking algorithm, and the forward checking algorithm, which is the default search algorithm in the constraint programming tools that I am aware of.

Both algorithms instantiate (i.e. assign a value to) each variable in turn, and build up a partial solution consisting of the variables already considered, with their assigned values; these are termed the past variables. The variables which are so far uninstantiated are 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 program may terminate, if only one solution is required, or carry on to find new solutions. If there are no solutions, the algorithm terminates when all possibilities have been considered.

If the CSP has n variables, each with m possible values, the maximum depth of any branch in the search tree is n and up to m branches are created from each node. There are up to m^n possible assignments of values to variables (many of which will not be allowed by the constraints, of course). Hence, the tree has up to m^n leaf nodes.

An example of a search tree built by the backtracking algorithm is shown in Figure 6, using the 4-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: hence no two queens can be on the same row, the same column or the same diagonal of the board.) As a CSP, this problem has 4 variables, representing the rows of the chessboard, and each variable has domain $\{1,..,4\}$ representing the 4 columns. However, it is easier to follow the progress of the search if the chessboard representation is used: a `Q` on a particular square should be taken as meaning that the variable corresponding to that row has been assigned the value corresponding to that column. Deadends, where the algorithm has to backtrack to a previous choice, are marked by crosses, and the solution eventually found is marked by a tick.

The backtracking algorithm only checks the constraints between the current variable and the past variables. The forward checking algorithm, on the other hand, checks the constraints between the current (and past) variables and the future variables. 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 immedi-

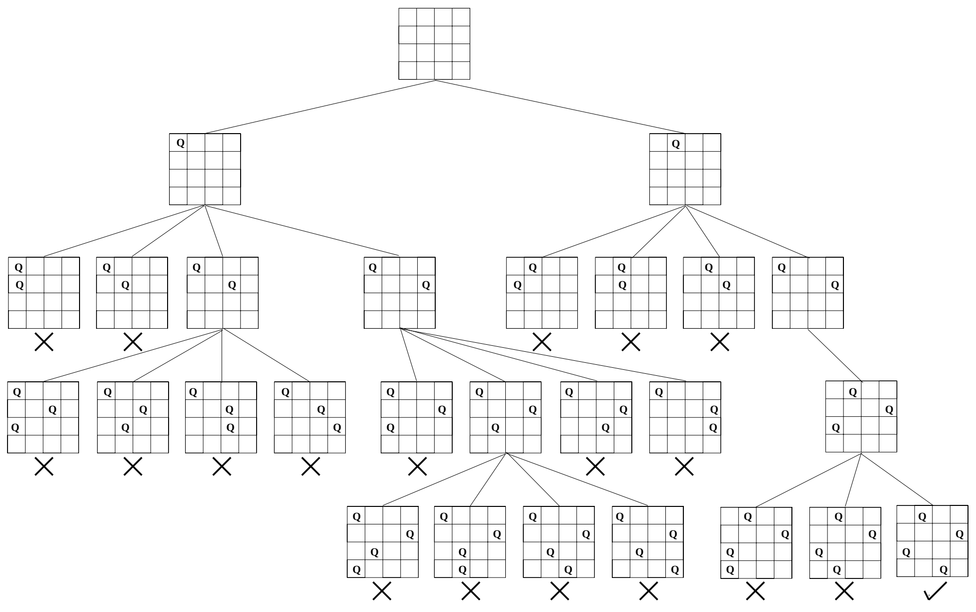

Figure 6: Search tree for 4-queens using simple backtracking

ately 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 have been discovered that none of its values were 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.

This can again be illustrated using the 4-queens problem. If we start by placing a queen on 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 representing the queens in rows 2 to 4, unless and until this branch leads to a dead end, and the first row queen has to be moved. The full search tree built by forward checking for this problem is shown in Figure 7. Squares with crosses denote values removed from the domains of future variables by the past and current assignments.

Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so that checking an assignment against the past assignments is no longer necessary. If all the constraints are binary, then only the constraints between the current

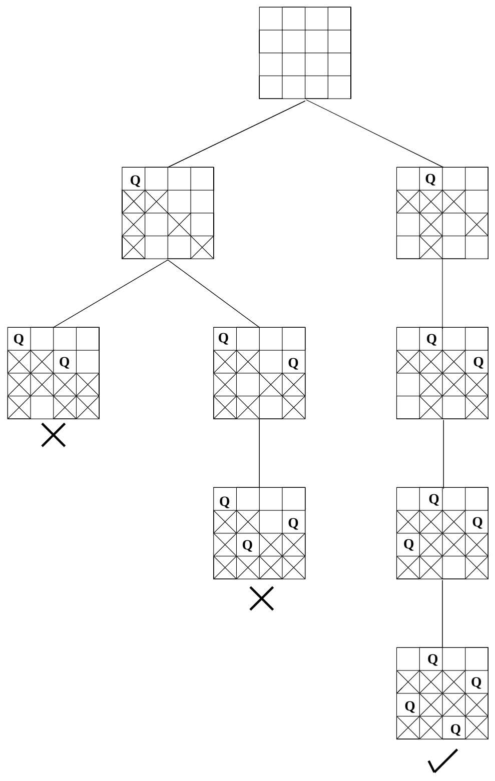

Figure 7: Search tree for 4-queens using forward checking

variable and future variables need be checked; it is only if there are constraints of higher arity that the past variables may need to be involved. (A k-ary constraint ($k > 2$) needs to be checked if and only if the current variable is one of the variables involved in the constraint, and all but one of the other variables in the constraint have already been instantiated; the past instantiations then effectively reduce the constraint to a binary constraint between the current variable and a future variable.)

Forward checking does more work when each assignment is added to the current partial solution, in order to reduce the size of the search tree and so (hopefully) reduce the overall amount of work done. In fact, in 4-queens forward checking does about the same amount of work as the backtracking algorithm in checking for consistency; there is not enough scope in such a small search tree for early pruning of large branches. However, the following artificial example will show that forward checking can save an arbitrary amount of work compared with simple backtracking: suppose we have variables $x_1, x_2, x_3, ..., x_n$, where x_1, x_2, x_n all have domain {1,2} and the constraints on these three variables are that they should all have different values.

Clearly this subproblem, and thus the whole problem, is infeasible. (It does not matter what the domains of the remaining variables $x_3, ..., x_{n-1}$ are, or what the constraints on these variables are: we assume that this part of the problem can be solved without any difficulty.) The backtracking algorithm will instantiate variables x_1 and x_2 to 1 and 2 respectively, and then assign values to $x_3, ..., x_{n-1}$ in turn, before discovering that there is no value for x_n which is consistent with the first two assignments. It will then backtrack to x_{n-1} and try all the alternative assignments for this variable, then backtrack to x_{n-2}, and so on, even though these variables are not part of the subproblem which is causing the difficulty. It could take a very long time to discover that the problem has no solution. Forward checking, on the other hand, will discover that there is no remaining value in the domain of x_n as soon as values have been assigned to x_1 and x_2: it will never consider assigning values to the remaining variables. Forward checking is not the only way of avoiding this kind of stupidity, and it can get into difficulties itself; on the whole, however, it performs reasonably well compared with other algorithms, when combined with good heuristics as described in the next section, and it is almost always a much better choice than simple backtracking.

A search strategy which does still more work in looking ahead when an assignment is made combines forward checking with maintaining arc consistency. Whenever a new subproblem is created, by removing values from the domains of future variables which are inconsistent with the current assignment, the subproblem is made arc consistent. This will remove further values from the domains of future variables, and as with forward checking itself, the hope is that in doing additional work at the time of the assignment, there will be an overall time-saving. Forward checking combined with maintaining arc consistency is the default algorithm used in ILOG Solver, for instance.

6 Variable Ordering

A tree search algorithm for constraint satisfaction requires the order in which variables are to be considered to be specified. (Even if no particular order is specified, the algorithm must have some default ordering to fall back on, probably the order in which the variables were defined.) The 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, 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 variable ordering heuristic is based on what Haralick and Elliott [4] termed the "fail-first" principle, which they explained as *"To succeed, try first where you are most likely to fail."* In forward checking, this principle is implemented by choosing next the variable with fewest remaining values in its domain, on the assumption that any value is equally likely to participate in a solution, so that the more values there are, the more likely it is that one of them will be a successful choice.

Calling this the fail-first principle is, it seems to me, slightly misleading, or at least one-sided: after all, we do not want to fail, so it seems at first sight perverse to deliberately choose the variable that is most likely to lead to failure. The reasoning is that if the current partial solution will not lead to a complete solution, so that the current branch will eventually prove to be a dead end, then the sooner we discover this the better. Hence encouraging early failure, if failure is inevitable, is beneficial in the long run. On the other hand, if the current partial solution can be expanded to a complete solution, then every remaining variable must be instantiated and the one with smallest domain is likely to be the most difficult to find a value for: instantiating other variables first may further reduce its domain and lead to a failure. Hence the principle could equally well be stated as *"Deal with hard cases first: they can only get more difficult if you put them off."* So whether the current partial solution will lead to a complete solution, in which case we want to get to the solution straightaway without further backtracking, or will not lead to a solution, and we want to find that out as soon as possible, choosing the variable with smallest remaining domain makes sense.

This heuristic should reduce the average depth of branches in the search tree by triggering early failure. Hence, even if there is no solution, so that a complete search is required, or if all solutions are required, the size of the search tree explored is less than if a static ordering is used.

When all variables have the same number of values, which is the case in some problems at the start, then the fail-first principle indicates that we should still try to choose the variable which is likely to be most difficult to instantiate, and a good choice is the variable which participates in most constraints (in the absence of more specific information on which constraints are likely to be difficult to satisfy, for instance).

A word of warning. Most experiments with CSPs have been done with randomly-generated problems in which every variable has the same domain size initially and all constraints are equally difficult to satisfy. For these problems, choosing the variable with smallest domain works extremely well. For real problems, too, it is often a good choice, but sometimes needs a little thought. For instance, [10] describes a rostering problem in which the variables represent tasks and the values people who can do those tasks. Some of the variables have very small domains initially, not because they are difficult to assign, but because they represent tasks which particular individuals can do, if there is nothing of higher priority available. So in this case, choosing the variable with smallest domain first would be wrong.

7 Value Ordering

Having selected the next variable to assign a value to, a search algorithm has to select a value to assign. As with variable ordering, unless values are to be assigned simply in the order in which they appear in the domain of each variable, we should decide how to choose the order in which values should be assigned. A different value ordering will rearrange 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 dead ends, *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.

Suppose we have selected a variable to instantiate: how should we choose which value to try first? It may be that none of the values will succeed; we are in fact exploring what will turn out to be a dead end, and we shall have to backtrack to the previous variable. In that case, every value for the current variable will eventually have to be considered, and the order does not matter. On the other hand, if we can find a complete solution based on the past instantiations, we want to choose a value which will lead to such a solution; a good general principle, then, is to choose a value which is likely to succeed, and unlikely to lead to a conflict (if we can detect such a value).

Some value ordering heuristics based on this principle have been proposed for use with forward checking, e.g. [5, 3]. In both cases, in order to select a value for the current variable, the state of the domain of future variables which would result from each choice is found, i.e. forward checking is done for each value in turn. Keng & Yun [5] suggest then calculating the percentage of values in future domains which will no longer be usable, as a measure of the cost of making this choice: the best choice would be the value with lowest cost. Geelen [3] suggests instead calculating the 'promise' of each value, that is the product of the domain sizes of the future variables after choosing this value (this is an upper bound on the number of possible solutions resulting from the assignment): the value with highest promise should be chosen.

Unfortunately, there is a great deal of work involved in forward checking from each possible value in turn. For randomly-generated problems, and probably in general, the work involved in assessing each value is not worth the benefit of choosing a value which will on average be more likely to lead to a solution than the default choice.

In particular problems, on the other hand, there may be information available which allows the values to be ordered according to the principle of choosing first those most likely to succeed.

8 Example

The cryptarithmetic problem discussed earlier was formulated with a single constraint representing the required sum:

```
CtEq(100000*D+10000*O+1000*N+100*A+10*L+D
  +  100000*G+10000*E+1000*R+100*A+10*L+D,
     100000*R+10000*O+1000*B+100*E+10*R+T);
```

This is sufficient to allow ILOG Solver to find a solution, using its default strategy of forward checking combined with arc consistency. However, it takes a relatively long time. The performance of the algorithm can be measured by the number of times it detects a failure and has to backtrack. With the constraint stated as above, it takes 8018 fails to find the solution.

The difficulty lies in the formulation of the sum as a single constraint involving all the variables. If we tried solving the puzzle by hand, considering the sum as a whole in this way would not give an obvious point of attack; if we consider how forward checking works, we can see that it also causes difficulties for the algorithm.

The power of the forward checking algorithm lies in the fact that each instantiation of a variable can be used to reduce the domains of future variables. Tools such as Solver also continually make sure that the remaining subproblem is arc consistent; having reduced the domains of two future variables which have a constraint between them, arc consistency may show that further reductions to the domain of one or other have become necessary. Both of these methods of reducing future domains can only make use of binary constraints, or constraints which have effectively become binary because all but two of the affected variables have already been instantiated. The pruning effect of higher arity constraints therefore has to be postponed until sufficient variables have been instantiated to make them into binary constraints. Hence, the forward checking algorithm combined with maintaining arc consistency is best at solving binary CSPs. If there is a choice between two formulations of a problem as a CSP, both of which represent problem entities in a natural way, then in general one that has constraints of low arity should be chosen.[3]

With this in mind, an alternative formulation representing the sum column by column, which has additional variables representing the quantities carried into the next column, is preferable:

```
CtIntVar
   C1(0,1),C2(0,1),C3(0,1),C4(0,1),C5(0,1);

CtEq(2*D, 10*C1 + T);
CtEq(2*L + C1, 10*C2 + R);
CtEq(2*A + C2, 10*C3 + E);
CtEq(N + R + C3, 10*C4 + B);
CtEq(E + C4, 10*C5);
CtEq(D + G + C5, R);
```

This finds the solution with 212 fails, a dramatic improvement.

So far, nothing has been said about variable and value ordering heuristics. It is hard to see any reason for preferring one value to another, so we shall not specify a value ordering; for the variable ordering, the heuristic of choosing the variables with smallest remaining domain, *from amongst the original variables*, seems a good choice. (Note that if we choose the variable with smallest remaining domain from amongst all the variables, the 'carries' would be chosen first, as they start off with smaller domains. In this case, choosing the carries first gives good results, but in general it may not be a good idea to allow subsidiary variables, defined entirely in terms of the original variables, to drive the search strategy.) With this heuristic, the solution is found with only 14 fails.

This is still not the best that can be done; however, the final improvement seems to depend on giving Solver a piece of additional information that it cannot spot for itself. It is clear from looking at the original sum that T must be even; however, arc consistency does not allow Solver to eliminate the odd values from the domain of T. If we do this by hand, as well as making all the previous modifications, the solution can be found without any backtracking.

[3]It has been mentioned that any CSP can in theory be expressed as a binary CSP. However, this is done by introducing new variables which represent tuples of the original variables, so that the new formulation is likely to be much more cumbersome to deal with. (See [12] for details.)

9 Symmetries

In many problems, if there are any solutions at all, 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 so be interchangeable in the roster. Such symmetries in the problem may cause difficulties for a search algorithm: if the problem turns out to be insoluble, 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 will allow only one solution from each class of equivalent solutions.

It is difficult to give general advice on how to do this, because it depends on the particular problem. As an example, in the rostering problem, we could number the staff in the group, and the tasks which the members of the group can be assigned to in the first time-period covered by the roster, and insist that if $i < j$ then the task assigned to person i must have a smaller number than the task assigned to person j. This means that the staff in the group are no longer interchangeable, thus ruling out the equivalent solutions, and avoiding doing unnecessary work. For instance, if there is no solution with person 1 assigned to task 1, then there will be no solution with person 2 assigned to task 1 either. (For more information on avoiding symmetries by adding constraints, see [6]. An example of a problem with symmetries is described in [11].)

10 Optimization Problems

Constraint programming tools adopt the same general approach to attempting to find an optimal solution: create a constrained variable which represents the objective function, find an initial solution, then introduce a new constraint that the value of the objective variable must be better than in the initial solution. Repeatedly solve the new problem and tighten the constraint on the objective variable in this way until the problem becomes insoluble: the last solution found is then the optimal solution.

For instance, ILOG Solver has a built-in function `CtMinimize` (and another `CtMaximise`, but we shall assume that we have a minimization problem) which can replace the usual function which finds just one solution, and takes an extra parameter, which is the variable representing the objective function, say `cost`. It is the programmer's responsibility to see that `cost` is defined in terms of the other variables in the problem, and that whenever a solution to the problem is found, `cost` is thereby assigned a value. Effectively what then happens (though this is taken care of internally by `CtMinimize`) is that whenever a new solution is found, the value of `cost` is saved, say in `best`, and a new constraint is added:

`cost` <= `best` - 1

The new problem is then solved, and this is repeated until the constraint on `cost` has been tightened to the extent that the problem cannot be solved; the last constraint is then removed and the previous solution (which is now known to be optimal) is found again.

This is clearly somewhat inefficient, in that the optimal solution is found twice. However, it works well, provided that the problem is small. With large problems, as the constraint on `cost` gets tighter, it can get extremely difficult to find a solution. Whereas good heuristics may be able to find a solution quickly if there are many possible solutions, this gets more difficult if there are few solutions, and of course in order to prove that a problem has no solutions (to show that the last solution was optimal) the entire search tree must be explored. So for large problems it may not be practicable to get anywhere near the optimal solution, still less to prove optimality. This partly depends on particular circumstances: in some cases, proving optimality is straightforward, because reducing `cost` below its optimal value results in a problem which is obviously infeasible. Even then, *finding* an optimal solution may not be easy.

In many cases, therefore, it is necessary to rely on good heuristics, and allow the program to search for improvements on the initial solution for as long as is practicable before accepting the current solution as (hopefully) good enough. The advantage in that case over heuristics which will simply construct a solution is that the built-in backtracking may find significant improvements over the initial solution before the search has to be abandoned.

11 Conclusions

It is probably clear from the foregoing that the search algorithms available for solving CSPs are relatively unsophisticated, compared for instance to mathematical programming techniques. On the other hand, it is because of this that the constraints can be much more expressive, and therefore more powerful, than is allowed in mathematical programming.

Because algorithms like forward checking search systematically for solutions, they are unlikely to be able to handle large problems without good heuristics to guide them. Good variable and value ordering heuristics are often crucial, and can make the difference between finding a solution very quickly and failing to find a solution at all.

An area which needs further investigation is the comparison between mathematical programming and constraint programming. Integer programming problems can be expressed as CSPs, but when is it worthwhile to do so? In [11], I describe a problem where constraint programming succeeded in finding a solution when integer programming had failed, and discuss reasons for the difference in performance in this case.

12 Further Reading

For a thorough overview of constraint satisfaction problems, concentrating especially on search algorithms and achieving different levels of consistency in CSPs, see Edward Tsang's book [12].

Van Hentenryck's book [13] is specifically on CHIP, and so contains a good deal on how Prolog has been extended to produce CHIP. It does also, however, contain quite a lot of useful material on tackling specific problems, which would apply to most constraint programming tools. Other useful papers on applying CHIP to specific problems, and comparing this approach with traditional OR are [1] and [2].

Papers on Solver are available from the ILOG World-Wide Web site (`http://www.ilog.fr/ilog/products/solver/solver.html`), including an overview of the system [7] and the paper on symmetries already mentioned [6].

References

[1] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving a Cutting-Stock problem in constraint logic programming. In R. Kowalski and K. Brown, editors, *Logic Programming*, pages 42–58. 1988.

[2] M. Dincbas, H. Simonis, and P. van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings ECAI-88*, pages 290–295, 1988.

[3] P. A. Geelen. Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems. In B. Neumann, editor, *Proceedings ECAI'92*, pages 31–35, 1992.

[4] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[5] N. Keng and D. Y. Yun. A Planning/Scheduling Methodology for the Constrained Resource Problem. In *Proceedings IJCAI'89*, pages 998–1003, 1989.

[6] J.-F. Puget. On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In *Proceedings of ISMIS'93*, 1993.

[7] J.-F. Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS94 (Singapore International Conference on Intelligent Systems)*, 1994.

[8] B. M. Smith. How to Solve the Zebra Problem, or Path Consistency the Easy Way. In B. Neumann, editor, *Proceedings ECAI92*, pages 36–37, 1992. (A longer version is available as Research Report 92.22, School of Computer Studies, University of Leeds).

[9] B. M. Smith. Locating the Phase Transition in Constraint Satisfaction Problems. Research Report 94.16, School of Computer Studies, University of Leeds, May 1994, to appear in *Artificial Intelligence.*

[10] B. M. Smith and S. Bennett. Combining Constraint Satisfaction and Local Improvement Algorithms to Construct Anaesthetists' Rotas. In *Proceedings of CAIA-92, the 8th IEEE Conference on Artificial Intelligence Applications*, pages 106–112, Mar. 1992.

[11] B. M. Smith, S. C. Brailsford, P. M. Hubbard, and H. P. Williams. The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared. Research Report 95.8, School of Computer Studies, University of Leeds, Mar. 1995.

[12] E. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[13] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.