Lecture Notes in Constraint Satisfaction and Constraint Programming

by

Barbara M. Smith

These lecture notes were produced by Barbara Smith for her lecture course in constraint programming, run at the University of Leeds, up to 2001. The course title was **AR33 Constraint Satisfaction and Constraint Programming**. Barbara has very kindly given me permission to distribute these notes as supporting material for my course, CP4, a 4th year course on constraint programming. The notes are unchanged, and reflect the course taught at Leeds in 2001.

Patrick Prosser (January, 2003)

AR33 Constraint Satisfaction and Constraint Programming

1 Preface

1.1 What AR33 is all about

Constraint satisfaction problems have been an important area of AI research for many years. The idea is a simple one: we have a number of variables, each with a set of possible values, and we have to assign a value to each variable so that the constraints of the problem are satisfied.

Computer vision researchers first looked at constraint satisfaction ideas around 1970, and it was even then realised that, in theory, many complex practical problems could be expressed in terms of constraint satisfaction. In the last ten years or so, constraint programming tools have been developed which allow the variables, values and constraints of a problems to be easily expressed, and provide algorithms to find solutions. Many industries are adopting these tools to solve a wide variety of problems, and the market is expanding very fast.

AR33 combines *constraint satisfaction* - looking at how real problems can be expressed in this way, how problems can be simplified, algorithms for solving them - with *constraint programming* - using a commercial tool for solving constraint satisfaction problems. There are likely to be job opportunities available for people who have some experience of constraint programming: at the moment the demand for expertise far outstrips the supply. Two past students are now working for ILOG UK. So, AR33 could be useful for your future career.

1.2 What makes constraint satisfaction AI?

Algorithms for solving constraint satisfaction problems are based on methods that people might use, if they had to solve this kind of problem by hand, rather than on mathematical techniques. So we make use of straightforward observations about how to simplify the problem, and find solutions by systematically searching for them. However, although we adopt methods that a person might use, we are not aiming to *match* human performance (as in expert systems), but to do *much better*. By using a computer to apply simple techniques over and over again, much more quickly and thoroughly than a human could, we can solve problems beyond the reach of human problem solvers.

1.3 Organisation of the module

1.3.1 Exam

The exam is worth 80% of the overall assessment. It will be an 'open-book' exam; you will be allowed to take into the exam any notes you have from the course, but not textbooks. For obvious reasons, I will try to design the questions so that mere possession of a set of handouts is not sufficient to enable you to do well. Open-book exams, ideally, test understanding rather than memory. During the exam, you should be using the notes for occasional reference; it's too late by then to try to start working out what they mean.

This is the fifth year that this module has run. Past exam papers are attached, to give you some idea of the style of questions.

1.3.2 Coursework schedule

There will be two pieces of coursework, each worth 10% of the overall assessment. One will be available in week 4 and due at the start of week 7; the other available in week 7 and due at the start of week 11. I estimate that the total time for the two pieces of coursework will average 15 hours, plus about 6 hours to familiarise yourself with the software. The coursework will be based on using ILOG Solver, a widely-used commercial constraint programming tool, which is available on the Linux server (cslin-gps). A detailed handout on using Solver will be given out before the first piece of coursework. Solver is a C++ library, and some programming in C++ is inevitable. I shall assume a basic knowledge of the language. However, since this is not a programming module, I shall try to minimise the amount of programming effort required to do the coursework, for instance by making example programs available which can be used as a basis. (The object-oriented aspects of C++ will appear only marginally, if at all.)

1.3.3 Textbooks

There aren't any. There are very few books on constraint satisfaction or constraint programming, and none of those available are suitable. I shall rely on the module handout and perhaps occasional journal or conference papers. The Solver manuals are available online, but I am not sure how useful they will be. Unfortunately this means that you will only have access to one view of the material, most of the time, i.e. mine. On the other hand, the handout is much cheaper than a textbook and in its complete form should be sufficiently detailed. 'Its complete form' means when you have added bits to it in lectures: in its original form, as handed out, the coverage of some topics may not be adequate or even make sense. Conversely, there will be background material and examples in the notes which will not be covered in the lectures, so that the notes are a bit more like a textbook.

1.3.4 Order of topics

I shall begin with a speedy introduction to the basic ideas of constraint satisfaction, to give you a basis for starting constraint programming as soon as possible. We shall then spend some time in lectures on how to use Solver, so that you will be able to start work on the first coursework. After that, we shall revisit the topics already touched on in the introduction and cover them in more detail.

At the end of the module you should have a good understanding of an important area of AI. In addition, you will have practical experience of using commercial AI software which can be used to solve real problems.

1.4 Example: The Template Planning Problem

Many of the examples I shall use in this module, to illustrate ideas, show how algorithms work, etc., will be trivial problems, often puzzles. I am including this example here because it is a real (and very difficult) problem which can be tackled by the methods we shall cover. We shall return to it later and consider how to attack it using Solver.

This problem originates from a printing firm in Leeds, which prints such things as magazine inserts; cartons for herbs and spices, cat food, etc; and post cards. A typical order is for several related designs: different in detail, but similar overall, and in particular the same shape and size. Because of this fact, we know in advance how many items can be printed on each sheet of card, and we can mix the designs on each sheet. An order quantity is specified for each design: in general, the quantities vary from design to design.

The following example is based on data from an order for cartons for different varieties of dry cat-food (Brekkies).

Design	**Order Quantity**
Liver	250,000
Rabbit	255,000
Tuna	260,000
Chicken Twin	500,000
Pilchard Twin	500,000
Chicken	800,000
Pilchard	1,100,000
Total	3,665,000

Each design of carton is made from an identically sized and shaped piece of card, shown in Figure 1. Nine cartons can be printed on each sheet of card.

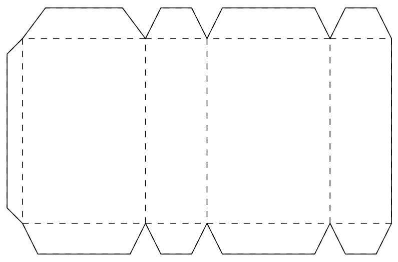


Figure 1: 2-dim. net for a Brekkies carton

The problem is to design a set of templates from which the sheets of card can be printed, in such a way that the overall cost of meeting the order is minimised.

Because in this example there are more slots in each template (9) than there are designs (7), it would be possible to fulfil the order using just one template. The best way of doing this, i.e. the layout which creates least waste, is to allocate two slots to the two largest orders (Chicken and Pilchard) and one each to the rest, as in Figure 2. We should need to print 550,000 sheets of card (filling the Pilchard order

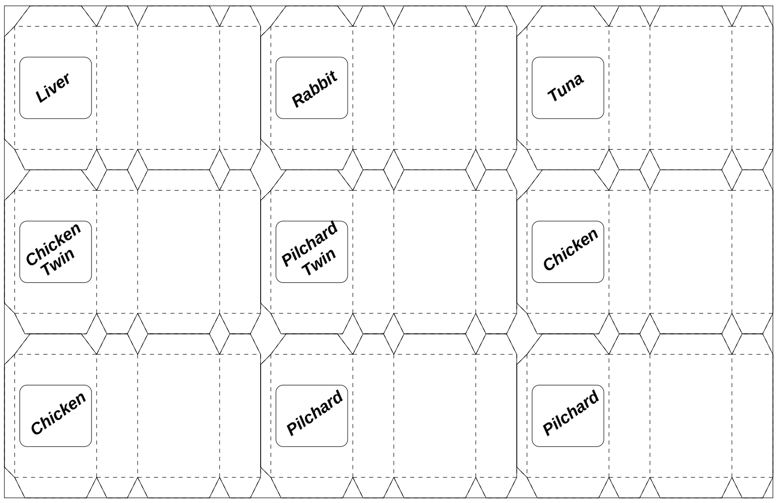


Figure 2: Optimal one-template solution for the Brekkies data

exactly and over-producing the other designs). Although this is the best solution using just one template, it would create an enormous amount of waste: 4,950,000 cartons would be produced, which is 35% more than the total order quantity. On the other hand, it minimises the cost of producing the templates.

At the other extreme, we could produce one template for each design, which would result in no over-production. However, even if we simply wanted to minimise over-production, this solution uses unnecessary templates. It is usually possible to find a solution with virtually no waste production using only a few templates.

We have little information on the relative costs to the company of card and templates. The strategy we have adopted is to find the optimal solution with one template (i.e. the solution with least over-production), then the optimal solution with two templates, and so on until a solution with virtually no over-production has been found. The company can then decide for themselves between these solutions.

The set of optimal solutions for the Brekkies problem is given below:

	Design							Run Length	% Over-production
	1	2	3	4	5	6	7		
Template 1:	1	1	1	1	1	2	2	550k	35
Template 1:	-	-	-	-	-	2	7	158k	
Template 2:	1	1	1	2	2	2	-	260k	2.6
Template 1:	-	5	3	-	-	1	-	51k	
Template 2:	-	-	1	-	-	7	1	107k	
Template 3:	1	-	-	2	2	-	4	250k	0.2

It seems very difficult to find good solutions to the template planning problem by hand, and the problem is not similar to any problem for which algorithms have been developed. On the other hand, it is a practical problem which the printing firm has to solve all the time. How should we tackle it?

2 Introduction

A constraint satisfaction problem (CSP) consists of:

- a set of *variables* $\mathrm{X} = \{x_1, ..., x_n\}$;
- 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.[1]

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 (e.g. cost) to be maximised or minimised.

2.1 Constraints

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

$$\begin{gathered} x_1 \neq x_2 \\ 2x_1 = 10x_2 + x_3 \\ x_1 x_2 < x_3 \end{gathered}$$

or, in ILOG Solver,

```
x1 != x2;
2*x1 == 10*x2 + x3;
x1 * x2 <  x3;
```

Formally, a constraint $C_{ijk...}$ between the variables $x_i, x_j, x_k, ...$ is any subset of the possible combinations of values of $x_i, x_j, x_k, ...$, i.e. $C_{ijk...} \subseteq D_i \times D_j \times D_k \times ......$ The subset is a set of tuples of values and specifies the combinations of values which the constraint allows. (Or you could equivalently define constraints by specifying the tuples which are *not* allowed.)

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)}.

The specification of a constraint by explicitly listing all the tuples which satisfy it is called its *extensional* represention; an expression like $x < y$ which (together with

[1]Strictly speaking, I have just described a *finite domain* CSP, and we could consider similar problems in which the variables have continuous domains, e.g. the real numbers. Professor Dew's research group does a lot of work with this kind of CSP for engineering design problems. However, the techniques for solving them are very different from those which we shall consider, and they don't really fall into the field of AI.

the variable domains) implicitly defines the satisfying tuples is called the *intensional* representation.

Although the constraints of real problems are not usually represented extensionally in practice, the fact that they can be defined in this way emphasises that constraints need not correspond to simple expressions. (And it will sometimes be useful to think of constraints extensionally.)

We can now also define formally what it means for a solution to satisfy a constraint. A solution to a CSP with variables $x_1, ..., x_n$ is a tuple $(a_1, a_2, ..., a_n)$, where $a_i \in D_i$. This solution *satisfies* a constraint $C_{ijk...}$ if the tuple $(a_i, a_j, a_k, ...) \in C_{ijk...}$, i.e. we pick out the values of the variables affected by this constraint (x_i, x_j, x_k,...) from the solution tuple and see whether the resulting tuple is in the list of tuples allowed by the constraint $C_{ijk...}$. So, as is obvious, $x = 1, y = 2$ does not satisfy the constraint $x = y$ and $(1, 2) \notin \{(1, 1), (2, 2)\}$.

If there are n variables in a problem, a constraint can affect any number of variables from 1 to n. 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 at the outset to remove any value which does not satisfy the constraint from the domain of the variable. 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. Binary constraints play an important role in many of the algorithms we shall be using, so we shall meet them again, even though the constraints in real problems are often not binary.

2.2 Examples of CSPs

Here, I give some examples of problems which can be represented as constraint satisfaction problems. Students often find it difficult at the start to see how to map a given problem to a CSP. In fact, even with experience, it is not always straightforward, because often there are different ways of doing the mapping, and it may not be clear which is best.

There are three basic questions to address in formulating a problem as a CSP:

- what entities in the problem should the variables represent?
- what are their possible values?
- what are the constraints?

Having answered those questions, we also need to decide how to represent the resulting CSP in whatever constraint programming tool we are using. In practice, we may need to bear in mind what kinds of constraints we can easily express in the constraint programming tool, while deciding how to represent our problem as a CSP.

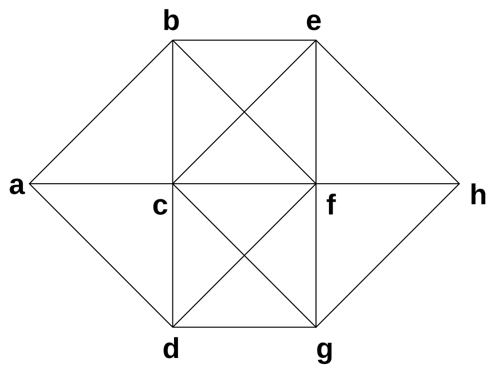
Many of these examples are puzzles rather than genuine problems. This is because it is hard to find practical problems which can be described in a few lines. (As shown by the template planning problem, for instance.)

2.2.1 Graph Colouring

Although apparently a purely theoretical problem, graph colouring can be used to represent some very practical problems, such as timetabling. We are given a collection of nodes, and some pairs of the nodes are joined by edges. Each node in the graph must be associated with a colour, from one of a set of k available colours, in such a way that any two nodes that are joined by an edge have different colours. Here, we can use the nodes as the variables of a CSP, the colours as their values and each edge as a $\neq$ constraint.

2.2.2 A Number Puzzle

A puzzle requires the numbers 1 to 8 to be placed at the positions marked `a` to `h` in the diagram, in such a way that wherever there is a line joining two of the positions, the corresponding numbers must differ by at least 2, and each number must be used exactly once.[2]



We can represent this as a CSP by having a variable a to h for each position `a` to `h`. Each variable's domain will be the set $\{1 \ldots 8\}$. There will be a constraint corresponding to every line in the diagram, that the values of the variables corresponding to the end points must differ by at least 2, e.g. $|a - b| \leq 2$.

The condition that each number must be used exactly once is equivalent to a constraint that the variables a to h must all have different values. We can either express this by a set of individual constraints between every pair of variables, e.g. $a \neq b$ or by a single constraint on all the variables at once, that they must all be different. The allDifferent constraint is one that occurs very frequently, in real problems as well as puzzles. It will be discussed at greater length later.

2.2.3 Cryptarithmetic

In a cryptarithmetic puzzles, like the following, each letter stands for a different digit. The puzzle is to find values so that the sum works out correctly.

```
    S E N D
  + M O R E
-----------
= M O N E Y
```

[2]This example appeared in a previous exam paper. But it wasn't in the handout before that!

This puzzle first appeared in *The Strand* magazine in 1924, according to `http://users.aol.com/s6sj7gt/mikealp.htm`. See also `http://www.cut-the-knot.com/cryptarithms/st_crypto.html` where many other similar puzzles can be found.

I shall use this example both to demonstrate how a problem like this can be defined as a CSP, in terms of variables, domains and constraints *and* to give a first example of how Solver is used. I shall use bits of Solver syntax to illustrate the example: I hope these are reasonably understandable. I *don't* expect you to be able to write programs in Solver on the basis of this example - the Solver tutorial comes later.

The puzzle can be represented as a CSP by creating variables to stand for the letters S, E, N, D, M, O, R, Y. The domain of each variable is the set of digits {0 .. 9} (except that S and M cannot be 0). ILOG Solver allows us to create a set of constrained integer variables, S, E,... each with domain {0 .. 9}, and we can ensure that S and M cannot have the value 1 by the constraints:

```
S != 1
M != 1
```

It is also convenient to form an array of these variables, called `letters`, say.

The constraints are:

- the eight variables must all be assigned a different value. In Solver, `IlcAllDiff` is a constraint that ensures that the variables in the array given as its argument all have different values:

  ```
  IlcAllDiff(letters);
  ```

- the sum given must work out:

  ```
   1000*S + 100*E + 10*N + D
  + 1000*M + 100*O + 10*R + E ==
     10000*M + 1000*O + 100*N + 10*E + Y;
  ```

  This equation can be treated as a single constraint on all eight variables.

This formulation can be improved, to allow Solver to find a solution more quickly; we shall see details later on.

2.2.4 Magic Square

In a magic square, we arrange the digits 1 to 9 in a three by three square (as in the diagram below) so that the sum of the three numbers in each row, column and diagonal is the same.

We can express this as a CSP by using the variables to represent the positions `a` to `i` in the square. The possible values for each variable are the numbers 1 to 9. So, in solving the CSP, we decide which number to put in each position.

The constraints are:

- every variable has a different value;

a	b	c
d	e	f
g	h	i

- the sum of the numbers in each row, column or diagonal are the same. We can express this by deciding what the sum must be (it has to be one third of the total 1 + 2 + ... + 9, i.e. 15). Or we can use another variable, say `s`, to represent the sum, and write the constraints in terms of `s`:

```
a+b+c = s
d+e+f = s
g+h+i = s
a+d+g = s
b+e+h = s
c+f+i = s
a+e+i = s
c+e+g = s
```

An alternative formulation. Because we have to assign nine values to nine positions, we could think of this problem in terms of deciding on the position for each number, rather than the number for each position. ¿From this viewpoint, the variables would correspond to the numbers, say v_1, v_2, .., v_9, and the domain of each variable would be the set of possible positions `a` to `i`. (If we were going to use this formulation, we would probably represent the positions by numbers, but for clarity, I will continue to label them with letters instead.) The first constraint stays the same: every variable has a different value. The other constraints, however, are now very hard to express. We might attempt to do this by defining a constraint for every triple of variables whose corresponding values do not sum to 15, e.g. v_1, v_5 and v_8 and specifying that they cannot be assigned values corresponding to three positions in a line. For instance, one of the constraints might be that $v_1 \neq a$ or $v_5 \neq b$ or $v_8 \neq c$; we would need a great many such constraints. The formulation given earlier is definitely better. This illustrates the fact that where there are alternative formulations, it may be much easier to express the constraints in one formulation than in another.

2.2.5 Exam Timetabling

A practical problem which needs little explanation is that of constructing a valid examination timetable for a group of students, say the timetable for the May/June exams for the University of Leeds.

Here we have a known set of exams to be timetabled; we know which exams each student is taking; and we know how many slots there are in the timetable. (The exam period is decided a long time in advance, so this effectively limits the number of available slots: we can't make the exam period as long as it needs to be to accommodate all the exams while meeting all the constraints, as might be preferable.)

To represent this as a CSP, we can create a variable for each exam, the possible values being the available slots. The most basic constraint is that the exams taken by any individual student must all be in different slots in the timetable. (The 'allDifferent' constraint again.) In previous examples, we have had just one allDifferent constraint. Here, we have an allDifferent constraint for every student, or for every group of students taking the same exams. We would hope that these constraints can be satisfied, given the number of available slots.

As well as this, we would want to satisfy some kind of 'spreading out' constraint, of the form 'no student should take more than x exams in any set of y consecutive slots', e.g. exams in three consecutive slots are avoided at Leeds. An evening slot and the following morning slot are counted as consecutive.

We would like to be able to express both of these constraints as 'hard' constraints, i.e. they must be satisfied. However, given that the number of slots in the timetable is absolutely fixed, we might have to allow at least the theoretical possibility that they could be broken. In that case, we might want to take into account the number of students affected, and keep this to a minimum.

To express the constraint that no student has three consecutive exams, suppose that the exam slots are numbered. One way of expressing this constraint is to say: for all i, j, k, where the variables v_i, v_j, v_k represent three exams where at least one student takes all three,

$$max(v_i, v_j, v_k) - min(v_i, v_j, v_k) \geq 3$$

i.e. the latest and the earliest of the three exams must be at least three slots apart. Solver allows you to define a variable whose value is constrained to be the maximum (or minimum) of the values assigned to a set of other constrained variables.

We might also have many other preferences in exam timetabling, which could be expressed as constraints that we should like to satisfy. This converts the problem from one of finding a solution to satisfy all the constraints, to an optimisation problem where we want to satisfy as many constraints as possible, taking into account their priorities.

For instance:

- as few students as possible should have two exams in consecutive slots
- departments may express preferences about the timing of specified exams (e.g. asking for an exam taken by a large number of students to be timetabled early to allow time for marking).

To outline how preferences can be dealt with in Solver: you can define a Boolean expression to represent a constraint which does not have to be satisfied. The expression has the value true (1) if the constraint is satisfied and false (0) otherwise. We can then count how many such constraints are satisfied at any time while we are searching for a solution.

2.2.6 n-Queens

The n-queens problem is a puzzle that has been used in AI for many years to investigate and illustrate search algorithms. We imagine an $n \times n$ chessboard instead of the standard 8×8 board, and place n queens on it in such a way that no queen can attack any other. In chess, a queen can move any number of squares in a straight line forward, backward, sideways or diagonally, so this means that no two queens can occupy the same row, column or diagonal. The problem has a solution for $n > 3$; Figure 3 shows a solution for 8 queens. (For this problem, the black squares and white squares of the board have no significance, so all the squares in Figure 3 are shown as white.)

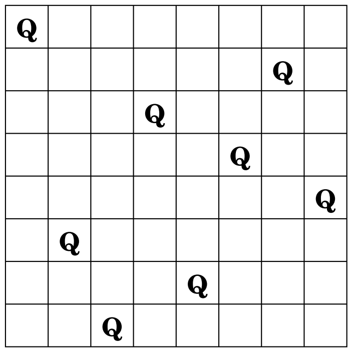


Figure 3: 8 queens on an 8×8 chessboard, no queen attacking any other

A possible formulation is to have a variable for each square of the board indicating whether or not we place a queen on this square. There are n^2 variables, each with domain $\{0,1\}$. If we number the squares 1 to n^2, and label the variables $v_1, v_2, ..., v_{n^2}$, we can express the constraints in terms of limits on the sums of subsets of the variables. For instance, the constraint that no two queens can occupy the first row of the board (squares 1 to n) can be written: $v_1 + v_2 + ... + v_n \leq 1$. Similarly, if $n = 8$, the squares 1, 10, 19, 28, 37, 46, 55, 64 form one of the main diagonals, and so there is a constraint $v_1 + v_{10} + ... + v_{64} \leq 1$. We need a constraint of this kind for every row, column and diagonal.

We also need a constraint that there must be n queens on the board (otherwise we can satisfy the other constraints by not placing any queens at all):

$$\sum_{i=1}^{n^2} v_i = n$$

An alternative formulation. If we recognise that there must be exactly one queen on each row of the board, we can represent the n-queens problem differently. We can have n variables, $q_1, q_2, ..., q_n$, where q_i represents the position of the queen on row i. So the value assigned to q_i in a solution tells us which column this queen is placed on, and the domain of q_i is the set $\{1, 2, ..., n\}$.

In any solution to a CSP, every variable must have been assigned a value (and exactly one value). Hence, we no longer need constraints to ensure that there are n queens on the board and one queen in each row. We only need to express the constraints that there is at most one queen (in fact exactly one queen) in each column, and at most one queen on each diagonal. To satisfy the first constraint, $q_1, q_2, ..., q_n$ must all have different values (the all-different constraint yet again).

The constraint that no two queens can be on the same diagonal is a little more tricky, but can be ensured by specifying:

$$q_i + i \neq q_j + j$$

$$q_i - i \neq q_j - j$$

for all pairs i, j, where $i, j = 1, 2, ..., n$ and $i \neq j$. (There are two constraints because the diagonals go in two directions.)

This formulation has fewer variables and fewer constraints than the first. However, we have had to apply our knowledge of the problem to derive it, so it might be classed as cheating. Cheating or not, it is the formulation of n-queens which is almost universally used in constraint programming: we shall meet it again later.

A curious point. Suppose we want to place only $n-1$ queens on an $n \times n$ chessboard - we might call this the $(n-1, n)$-queens problem. Placing $n-1$ queens should be easier than placing n queens, because any solution to the n-queens problem will give us n different solutions to the $(n-1, n)$-queens problem (we just remove any one of the n queens). However, if we try to represent the $(n-1, n)$-queens problem from scratch we cannot use the 'trick' that each row has exactly one queen on it, so the formulation of the easier problem is actually harder.

3 Arc Consistency

In most cases, in order to solve a CSP, we will have to *search*; i.e. try different values for the different variables, in some systematic way, until we find an assignment that satisfies all the constraints. However, before doing this, it is often useful to see if the problem can be simplified by identifying values which can never be part of a solution. If we do this, and delete the values from the domains of their variables, we save possibly a great deal of fruitless search. This is an example of a general approach to using the constraints to derive implicit information about the possible solutions to a CSP, called *constraint propagation.*

Here we consider the simplest form of constraint propagation, *arc consistency.* This deals just with binary constraints. If all the constraints of a CSP are binary, the variables and constraints can be represented in a *constraint graph*: the nodes or vertices 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. (In graph colouring, the edges in the graph to be coloured correspond to the $\neq$ constraints and the nodes correspond to the variables, so that the constraint graph and the graph defining the problem are the same. Similarly, in the example of section 2.2.2, the binary constraints exactly correspond to the lines in the puzzle, so that the puzzle itself is a constraint graph for the problem, ignoring the allDifferent constraint.)

We can see intuitively that in Figure 4, some of the values in the variables' domains cannot ever be assigned *(which ones?)*, because there is no way that they can satisfy the constraints. Hence, the domains of the variables can be reduced, if we consider each pair of variables and the constraint between them, in turn.

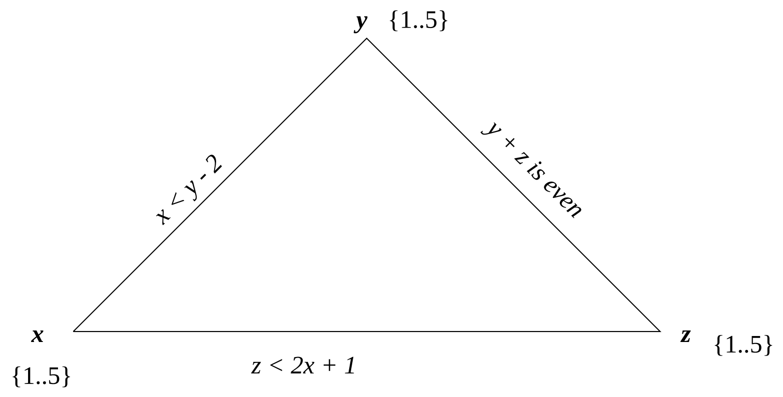


Figure 4: Binary constraints represented graphically

This intuitive idea can be formalised if we define *arc consistency.* (Note that an arc, say (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.)

If there is a binary constraint C_{ij} between the variables x_i and x_j then the 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. A value $b \in D_j$ such $x_i = a$ and $x_j = b$ satisfy $C_{ij}$is a *supporting value* for $a \in D_i$. We delete any value $a \in D_i$ unless it has at least one supporting value in the domain of every variable which constrains x_i. 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.

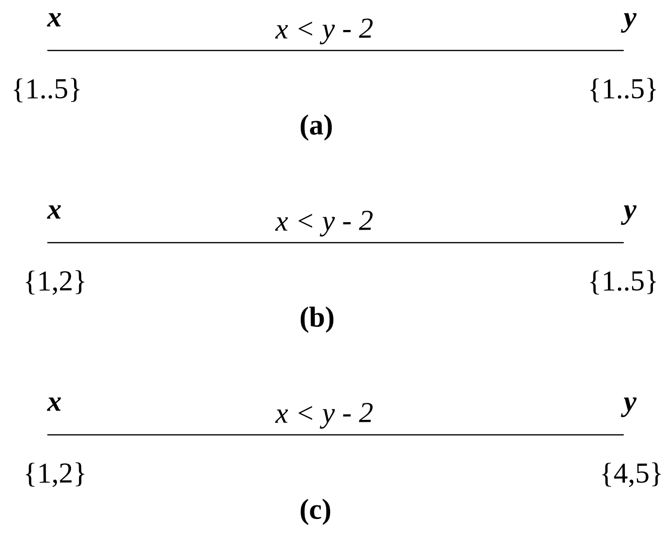


Figure 5: Making (x, y) and (y, x) arc consistent

Figure 5(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.

It is (I hope) intuitively obvious how to make a binary CSP arc consistent by hand, given a small constraint graph. Later, we shall look at algorithms for achieving arc consistency. These algorithms are fast (compared to the time taken to actually find a solution to a CSP), and as we are only removing values which cannot be part of any solution, they simplify the problem without changing the set of solutions. So it is usually a good idea to make a problem arc consistent.

3.1 Matrix representation

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[3], where 1 signifies that the constraint allows the corresponding pair of values, and 0 that it does not. (This can sometimes be a useful way of representing binary constraints. However, even if we specify a constraint intensionally in a tool such as Solver, we would not do it by a matrix, but by a list of pairs, so this is not a practical way of defining real constraints.)

For example, the constraint $x < y - 2$, where x and y both have domains $\{1 ... 5\}$ can be represented like this:

$$\begin{array}{ccccccc} & x & 1 & 2 & 3 & 4 & 5 \\ y & & & & & & \\ 1 & & & & & & \\ 2 & & & & & & \\ 3 & & & & & & \\ 4 & & & & & & \\ 5 & & & & & & \end{array}$$

Notice that, if a binary constraint is represented by a matrix, 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.

3.2 Line Labelling

Some of the earliest work on constraint satisfaction problems was done in computer vision. An early attempt at making sense of a scene made several simplifying assumptions:

- we assume that the 2-D image is composed of straight (black) lines on a plain (white) background, with no noise, missing portions, shadows, etc.
- we assume that the 3-D scene the image represents is composed of solid objects whose faces are planar and polygonal, and that no more than three faces meet at each vertex.

(Current computer vision research is of course dealing with much more complex and realistic images: the approach described here is more than 25 years old - see M

[3]...or a table, for non-mathematicians.

B Clowes, *On seeing Things*, Artificial Intelligence, vol. 2, pp. 79-116 (1971); D A Huffman, *Impossible Objects as Nonsense Sentences*, in Machine Intelligence 6, ed. B Meltzer & D Michie (1971).)

With the above assumptions, an edge of a solid body is a straight line and must be either convex or concave. Each vertex in the scene represents the meeting of up to 3 faces: in the image, anywhere between 0 and 3 faces at each vertex may be visible, the others being occluded, or hidden.

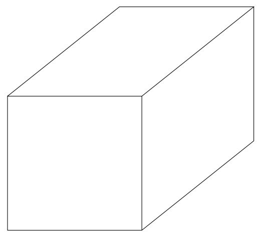


Figure 6: Example of an image to be labelled

Each edge in the scene which is visible in the image will be in one of the categories below. In order to understand the scene, we want to label every visible edge as belonging to one of three categories. (There may not be a unique way of doing this.)

- *convex edges*, marked +.
- *concave edges*, marked −.
- *occluding edges*, marked either ▶ or ◀: moving along an occluding edge in the direction of the arrow, the visible face of the object is on the right, and the occluded or hidden face is on the left.

There are four possible kinds of vertex in the image, and the possibilities for the labels on the edges meeting at a vertex are shown in Figure 7.

We can represent the line labelling problem by having a variable for each edge in the image; the domain of each variable contains the possible labels, i.e. $\{+, -, ▶, ◀\}$. There is a constraint between the edges meeting at a vertex; the labels assigned to the edges must be one of the valid combinations for that type of vertex shown in Figure 7. So, the constraints of the problem correspond to the vertices in the image.

Figure 8 shows the image of Fig. 6 marked to show the variables and their domains. We can then use Fig. 7 to reduce the possible values of each variable (Fig. 9). In this case, every remaining value is part of a possible solution, and in fact there are four possible solutions (Fig. 10).

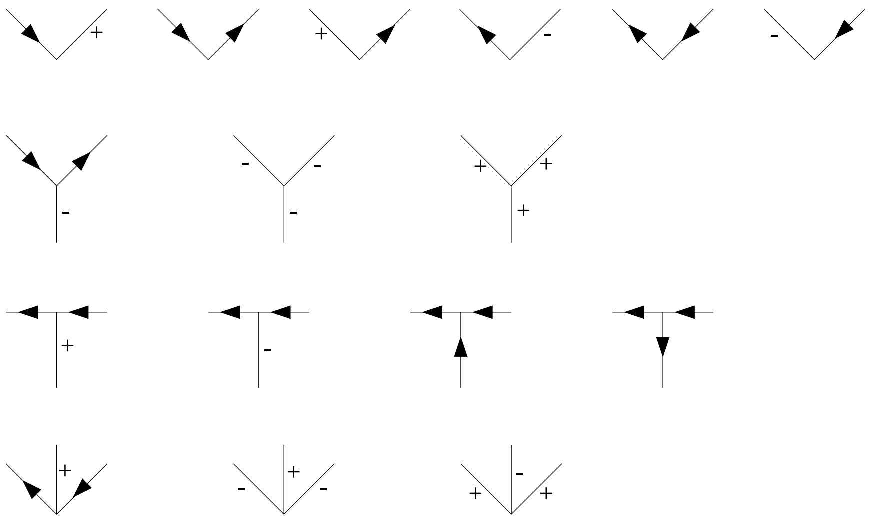


Figure 7: Valid combinations of labels at vertices

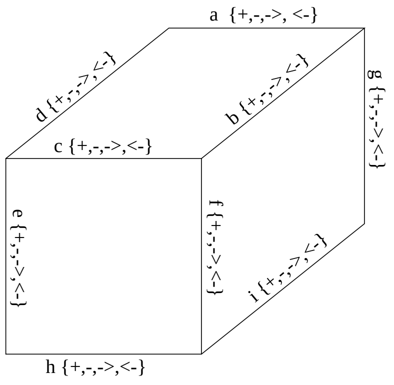


Figure 8: Image showing the variables and their domains

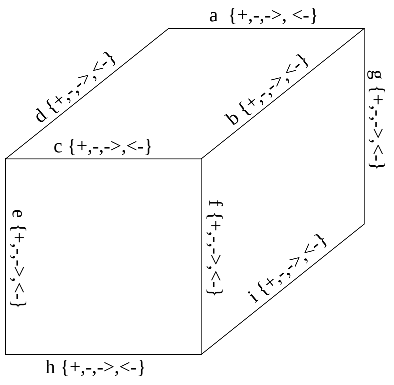


Figure 9: The reduced domains *(to be completed in lectures)*

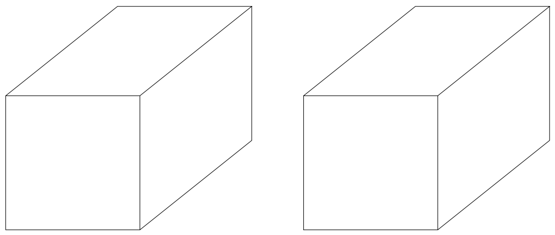


Figure 10: Two possible solutions *(to be completed in lectures)*

3.3 Two Logic Puzzles

These two puzzles are of a kind which you can find in the puzzle books sold in newsagents. They can often be expressed quite easily as binary CSPs, and if so, I have found that the ones available in this country are invariably amenable to arc consistency: if you make the problem arc consistent, there is only one value left in each variable's domain. (For some reason, you can find much more difficult puzzles of this kind in American puzzle books.) I don't know the origin of the zebra problem: it's been well-known to CSP researchers for many years. It's much more difficult than the first puzzle. You might like to try solving it by hand, before we apply Solver to it.

3.3.1 Visiting time.

Five children are in hospital and at visiting time each is visited by a friend or relative, bringing two presents: something to eat or drink, and something to play with. The task is to link each child with his/her visitor and presents, using the information given below.

The five children: Elizabeth, Jane, Peter, Simon, Wendy.
The visitors: aunt, father, friend, grandfather, mother.
The presents: bananas, cake, jelly-babies, orange squash, toffees.
colouring book, comics, doll, playing cards, story book.
Each child is to be linked with a different item from each category.
We can represent this as a binary CSP and hence draw a constraint graph.
We are also given the following information:

- The child who was given the doll did not receive any sweets.
- The bananas and playing cards were given to the same girl.
- The person who brought the orange squash was not a female relative.

which can be represented by additional constraints. Finally, using the following information, we can find the solution by reducing the domain of every variable to just one child:

- Simon's friend brought him comics.
- The doll was given to a girl.
- The cake was for Elizabeth's birthday.
- The toffees were brought for a boy.
- Wendy's visitor was not a female relative.
- Jane's grandfather brought her favourite sweets.

- Peter's visitor was his aunt; he did not receive the colouring book.

3.3.2 The zebra problem.

There are five houses of different colours, inhabited by five different nationalities, with different pets, drinks and cigarettes:

- The Englishman lives in the red house.
- The Spaniard owns a dog.
- Coffee is drunk in the green house.
- The Ukrainian drinks tea.
- The green house is immediately to the right of the ivory house.
- The Old Gold smoker keeps snails.
- Kools are smoked in the yellow house.
- Milk is drunk in the middle house.
- The Norwegian lives in the first house on the left.
- The Chesterfield smoker lives next to the fox owner.
- Kools are smoked next to the house with the horse.
- The Lucky Strike smoker drinks orange juice.
- The Japanese smokes Parliament.
- The Norwegian lives next to the blue house.

Who drinks water and who owns the zebra?

3.4 The Limits of Arc Consistency

Techniques such as arc consistency use the constraints of the problem to make explicit some information about the problem which was previously only implicit. If we did not remove the values which arc consistency tells us can be deleted, and found all solutions to the CSP, we would find that there is no solution involving these values. *Constraint propagation* is the general term for techniques such as this. If we have an arc consistent problem and add another binary constraint to it, and then make the new problem arc consistent again, the effects can indeed propagate throughout the network. As well as potentially reducing the domains of the two variables linked by the new constraint, it can indirectly reduce the domain of any other variable, if successive reductions elsewhere in the network eventually remove the only supporting values for some value in its domain.

Arc consistency is a powerful technique for reducing the domains of variables. In a few special cases, we may find that making the problem arc consistent is all we need to do:

- the result of making the problem arc consistent is that every variable has only one value remaining, as in the logic puzzle above. Assigning its remaining value to each variable yields the only possible solution. However, this is exceptional, and the puzzle was clearly designed to behave in that way.

- in the course of removing any values which have no support, we find that some variable in the problem has no values left. In that case we know that the problem has no solution.

- every value left when the problem is arc consistent is part of a solution. In some cases, we can then easily find a solution. (It is not possible to detect this by inspection, but it is true of some special classes of problem.)

In general, however, there is no guarantee that an arc consistent problem has a solution at all. To find a solution, or to prove that there is no solution, we need other techniques.

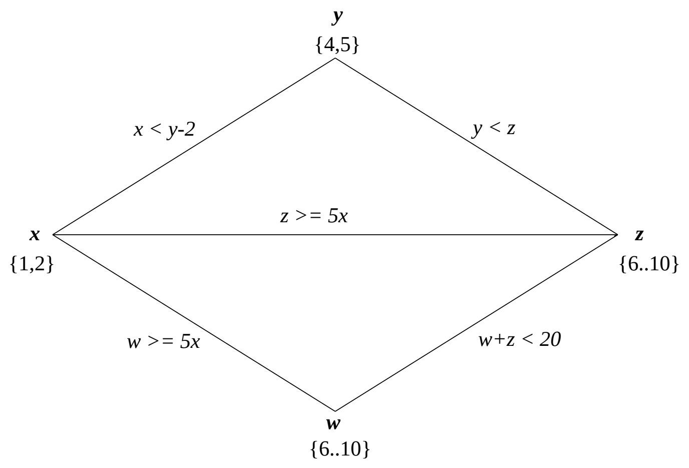


Figure 11: An arc consistent constraint network

4 Finding Solutions

The example I shall use throughout this section is the n-queens problem, already described in section 2.2.6: place n queens on an $n \times n$ chessboard in such a way that no queen attacks any other. (No two queens can be on the same row, the same column or the same diagonal.) This is the best example I know of from the point of view of visualising what a search algorithm is doing.

As described earlier, we can represent the n queens problem as a CSP by using n variables, $q_1, q_2, ..., q_n$, representing the rows of the chessboard. Each variable has domain $\{1,..,n\}$ representing the n columns.

We want to find an assignment of a value to each variable such that all the constraints are satisfied. In this case, the problem is already arc consistent, as you can check, so that we cannot simplify the problem in that way. We need some kind of algorithm to solve this and other CSPs. What properties do we want such an algorithm to have? Two that we might insist on are:

- *soundness*: any solution that the algorithm finds is a genuine solution;
- *completeness*: the algorithm guarantees to find a solution in a finite time, if there is one. (Hence, if it terminates without finding a solution, this is because the problem has no solution.)

Soundness is clearly of prime importance. Completeness may also seem a necessity, and all the algorithms we shall use will be complete. But there are incomplete algorithms for solving CSPs (for example, *local search* algorithms, which I shall discuss briefly later on). If problems are very hard to solve (and large CSPs can be), a complete algorithm may in practice take far too long to prove that a particular instance has no solution. So an algorithm which is in theory complete may be incomplete in practice. In those circumstances, it may be better to use an algorithm which we know is incomplete, if it has a better chance of finding a solution when there is one. On the other hand, if a problem has no solution, an incomplete algorithm will have no way of detecting this, no matter how long it runs for, whereas a complete algorithm may be able to prove that there is no solution.

4.1 Generate and Test

A very simple way of finding a solution to a CSP is to generate all possible assignments of values to variables, in some systematic way, and test each one in turn to see if it satisfies the constraints. If it does, we have a solution and can stop (if we only want one solution); otherwise we carry on. If all assignments have been tested and no solution has been found, then there isn't one. So this algorithm is both sound and complete.

If every variable has a m possible values and there are n variables, then there are m^n possible assignments. We shall have to examine all of these if *either* there is no solution *or* the very last assignment is the only solution *or* we want all solutions *or* we want the best solution (in which case we find all solutions, and pick the best). If there is a solution, and we only want one, we might be very lucky and find that the first assignment we examine satisfies the constraints. On the whole, however, this is a really stupid algorithm.

To see why, suppose that the constraints do not allow $v_1 = 1$ and $v_2 = 1$ (as with the n-queens problem, for instance). There are m^{n-2} assignments in which this combination of values appears. We will have to generate them all and find out each time that the assignment is invalid; the algorithm has no way of learning that this is a mistake. Most algorithms for finding solutions to CSPs are based on systematic search, and can avoid, to a greater or lesser extent, considering partial assignments which can be shown not to lead to a solution.

4.2 State-Space Search

This is a basic problem solving technique in AI, which I shall briefly review. Given an *initial state*, one or more *goal states*, and a set of *operators* which will produce the possible successors to a state, we want to find out how to get from the initial state to a goal state.

We can represent the search for a goal state as a tree, with the nodes representing the states (the initial state being the root node) and the edges the possible operators. In searching for a solution, we gradually construct this tree, by repeatedly choosing a node and one of the operators leading from it, and creating a new node representing the resulting state. Two important search techniques are *depth-first search* and *breadth-first search.* (See any general AI textbook.)

In depth-first search, we always choose to explore a node which is a child of the most recently-explored node. If this node has no children we backtrack up the path which led to this node until we find a node with an unexplored sibling. We have to keep in memory all the unexplored siblings of any node on the path from the current node back to the root of the tree (in case we need to backtrack to any of them).

In breadth-first search, every node at level d is explored before any node at level $d+1$. The memory requirements in this case are far larger (and increase as we progress down the tree.)

There are some advantages to breadth-first search:

- it will find the shortest path to the goal state (which is important in some problems).
- depth-first search can in some types of problem get trapped in exploring an infinitely long path and so never reach a goal state.

4.2.1 Finding a solution to a CSP as State-Space Search

A possible way of viewing finding a solution to a CSP in terms of state-space search is as follows:

- in the **initial state**, no variables have been assigned values
- a **goal state** is a state in which every variable has been assigned a value and all the constraints are satisfied (i.e. it corresponds to a solution).
- **intermediate states** correspond to partial solutions in which some variables have been assigned values and the relevant constraints are satisfied
- the **operators** are:
  - select a variable which has not yet been assigned a value;
  - at a node where a variable has just been selected, select one of the values in its domain

(See diagram in lectures.)

The search tree formed in this way when solving a CSP has several special characteristics:

- the search tree is finite; there can be no infinitely long paths in it.
- all solutions occur at the same depth in the tree (i.e. depth $2n$, if n is the number of variables, and we count the root node of the tree as depth 0. We have n levels corresponding to choosing each of the variables in turn, alternating with n levels at which we choose a value for the current variable.)

Because of these characteristics, we choose depth-first search rather than breadth-first search.

In the rest of this section, we shall assume that the order in which variables are selected is predetermined, so that selecting the next variable is not shown as part of the search process. However, the variable selection strategy does in fact play an important role in successfully solving CSPs and we shall return to this later.

The simplest algorithm based on depth-first search is called *simple backtracking.*

1. choose a variable which has not yet been assigned a value. If there are none, then stop: a solution has been found.

2. choose a value for this variable.
3. test whether the new partial solution satisfies all the relevant constraints (i.e. all the constraints which affect the current variable and one or more of the past variables (those that have already been assigned values):
    - if so, return to step 1.
    - if not, choose another value for this variable; if there are no more values, backtrack to a variable which still has values which have not been tried; if there is no such variable, stop: there is no solution.

Figure 12 shows the search tree for the 4-queens problem using simple backtracking. (As already mentioned, we ignore the selection of next variable in this search

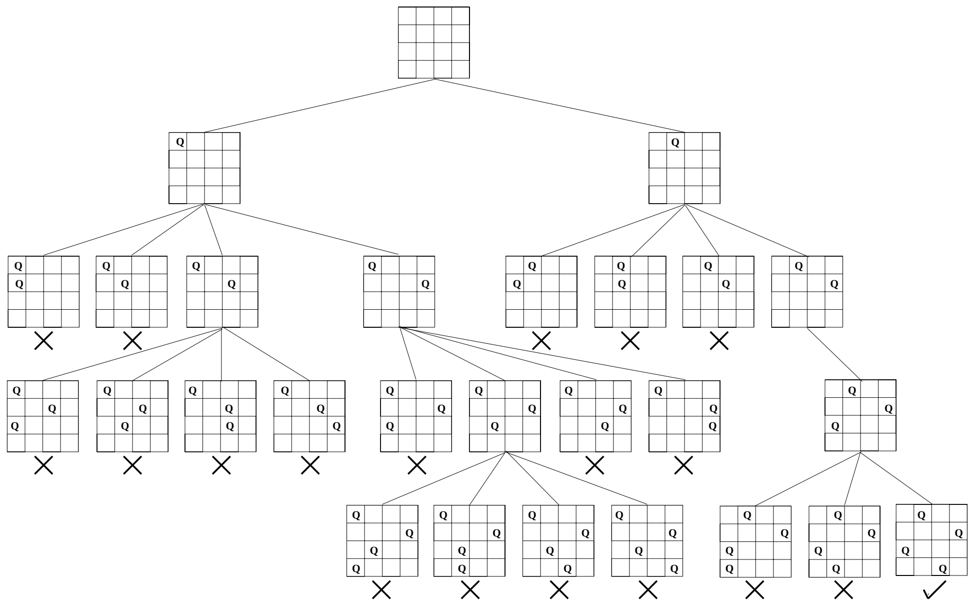


Figure 12: Search tree for 4-queens using simple backtracking

tree; we assume that the variables are considered in the order q_1, q_2, q_3, q_4.)

4.3 Forward Checking

Because the backtracking algorithm only checks the constraints between the current variable and the past variables, it tries many assignments which we can see are clearly going to fail. For instance, if the queen in the first row is in the first column, there is no point in trying to place any other queen in that column. The forward checking algorithm tries to avoid useless assignments of that kind by looking ahead at the effect of any assignment on the future variables (the future variables are those that have not yet been assigned).

When a value is assigned to the current variable, any value in the domain of a future variable which conflicts with the new partial solution is (temporarily) removed from the domain. The advantage of this is that if the domain of a future variable, say v_f, 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 will not be detected until much later. When v_f becomes the current variable and an attempt is made to assign a value to it, it will 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.

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. 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 13. Squares with crosses denote values removed from the domains of future variables by the past and current assignments.

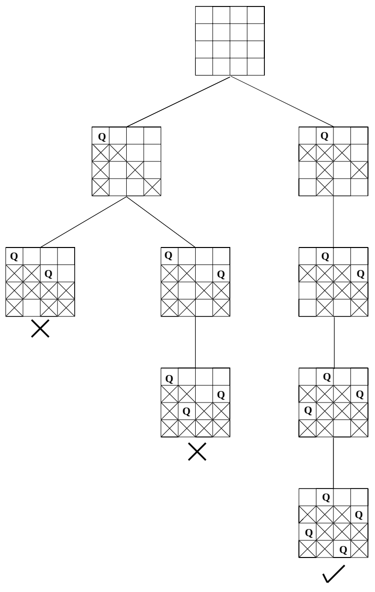


Figure 13: Search tree for 4-queens using forward checking

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.

The only constraints that we need to consider when checking forward are those that involve both the current variable and exactly one future variable. These constraints could be either binary constraints or constraints of higher arity in which all but these two variables have already been assigned values. In the latter case the constraint has effectively become binary (although this may only be temporary: we may find later that we have to backtrack and undo some of these assignments). This is an example of why binary constraints or constraints involving only a small number of variables are so important in CSPs; we can use binary constraints to propagate the effects of an assignment to unassigned variables, and even if the constraints are not binary at the outset, they will eventually become binary during search, as the variables are assigned values.

Provided that the same variable ordering is used in both cases, it is guaranteed that forward checking will explore no more nodes than simple backtracking, usually far fewer. (Changing the variable ordering will have unpredictable effects on the search tree: for instance, a different solution may be found. The guarantee then no longer holds.)

However, at each node, forward checking does more work than simple backtracking, so that overall it can take longer to find a solution. For instance, if it is very easy to satisfy the constraints, it may be that simple backtracking can find a solution without ever having to backtrack to a previous variable; in that case, checking the effects of assignments on the domains of future variables is a waste of effort. This rarely happens in practice, however, and almost always forward checking is much faster than simple backtracking.

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.

4.4 Maintaining Arc Consistency

The MAC, or full-lookahead, algorithm does still more work than forward checking 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 partial solution, as forward checking does, this checks the future variables against each other. Any value in the domain of a future variable which has no supporting value in the domain of some other future variable which constrains it is deleted, in addition to those which are not consistent with the current partial solution. This may remove more values from the domains of future variables than forward checking does, and as with forward checking, the hope is that in doing additional work at the time of the assignment, there will be an overall time-saving.

Figures 14 and 15 compare forward checking and MAC on 6-queens. As before, squares marked with a cross are those which were eliminated by a previous assign-

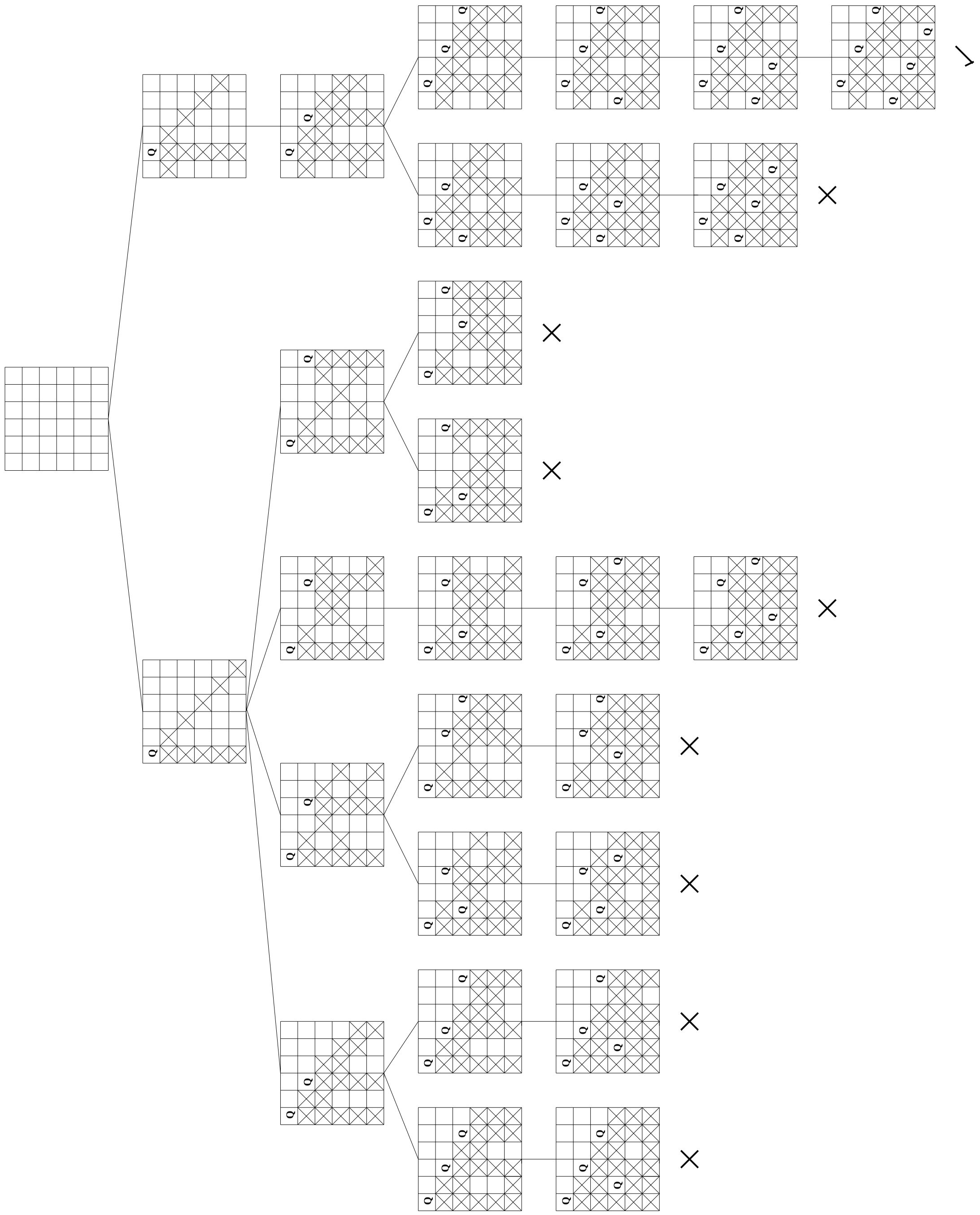


Figure 14: Search tree for 6-queens using forward checking

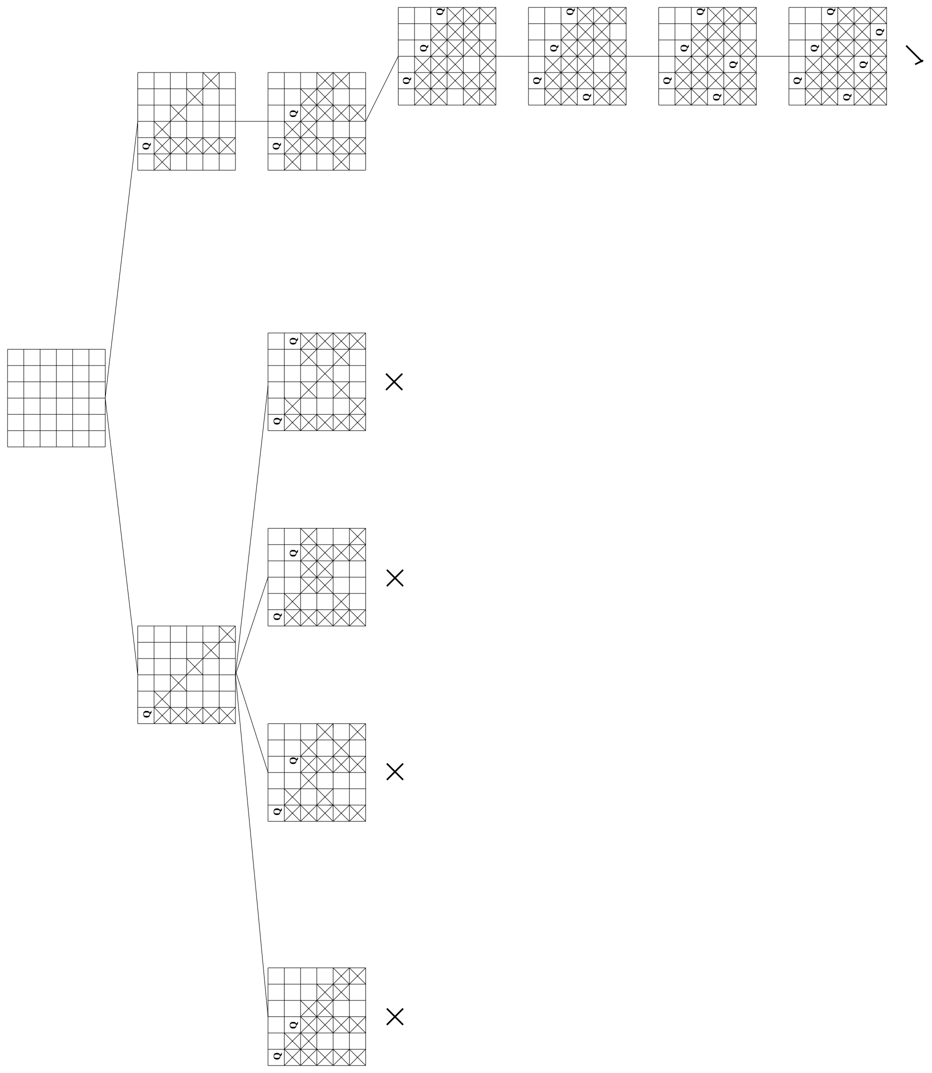


Figure 15: Search tree for 6-queens using MAC

ment, or are inconsistent with the current assignment. We can cross out additional squares by making the future subproblem arc consistent. *(The diagram needs to be completed in lectures.)*

The rightmost branch in Figure 15 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 Figure 14. In fact, as with simple backtracking v. forward checking, it is guaranteed that the nodes explored by MAC are a subset of those explored by forward checking, provided that the same variable ordering is used.

The MAC algorithm interleaves constraint propagation and search. Re-establishing arc consistency after each variable instantiation can be done efficiently using an incremental AC algorithm (which we shall consider in more detail later) rather than starting from scratch. This is essentially the algorithm used by constraint programming tools, including ILOG Solver.

5 Variable and Value Ordering

In the n-queens examples considered so far, the variables have been considered in the order $q_1, q_2, ..., q_n$ and the values in the order 1, 2, ..., n. There is no particular reason to believe that this is the best order in which to consider the variables and values, and selecting the next variable to consider and the value to assign to it are clearly part of the search process.

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. For instance, suppose the problem has a solution in which $v_1 = l_1, v_2 = l_2, ...$, and we happen to choose the values for each variable in such a way that the first value considered for each variable is its value in this solution. Then we can 'find' this solution, even using simple backtracking, very quickly.

There are general principles which are commonly used in selecting the variable and value ordering, and a few specific heuristics.

5.1 Variable Ordering

A good variable ordering will reduce the size of the tree explored by the search algorithm, in comparison with a poor ordering. Hence, even if there is no solution, so that a complete search is required, or if all solutions are required, the search can be expected to take less time. (See the example given in lectures.)

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: all the future variables look the same as they did at the start, so there is no reason to change the ordering. We shall consider static variable orderings later.

5.1.1 Dynamic Variable Ordering

With the FC and MAC algorithms, the current state includes the domains of the variables as they have been pruned by the assignments already made, and so it is possible to base the choice of next variable on this information, resulting in a dynamic variable ordering. To put it another way, the future subproblem which these algorithms are faced with changes after every assignment, so there is no reason why the initial ordering of the variables should remain the best. Dynamic variable ordering heuristics generally perform much better than static ordering, since they use information about the current state of the search.

With a dynamic variable ordering, the order may be different on different paths through the search tree: this still, however, falls within the framework of depth-first state space search.

A common heuristic for variable ordering is to choose the *most constrained variable*, i.e. the one with smallest current domain. The explanation given by Russell and Norvig in *Artificial Intelligence: A Modern Approach* (Prentice-Hall, 1995) is that this minimises the *branching factor* at the current node, i.e. the number of potential branches formed.

When several variables have the same minimum number of values, we need a tie-breaker: for instance, in many problems all the domains are initially the same size, as in the n-queens problem, for instance. The *most constraining variable* heuristic says that we should choose the variable which constrains the greatest number of unassigned variables. Russell and Norvig's justification for this is that it attempts to reduce the branching factor at future nodes.

This combination, choosing the variable with the smallest remaining domain and breaking ties by choosing the variable which constrains most unassigned variables, was first proposed for graph colouring by Brélaz ('New Methods to Color the Vertices of a Graph', *Communications of the ACM*, **22**, pp. 251-256, 1979). In terms of graph colouring, we choose next the node (or vertex) with fewest available colours, and break ties by choosing the vertex which is adjacent to (i.e. is linked by edges to) the largest number of uncoloured vertices. This works well in graph colouring, and in many CSPs in general. In graph colouring, it tends to explore the densest areas of the graph first; if there is a large clique[4], which will make the graph uncolourable if it has more nodes than the available number of colours, this increases the chances of finding it early in the search. However, in graph colouring, all the domains are the same size initially and all the constraints are the same. For general CSPs, where these conditions may not hold, it doesn't always work so well. For instance, if some of the constraints are much harder to satisfy than others, it might be a good idea to assign values to the variables involved in these constraints before the others, irrespective of the domain sizes.

5.2 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

[4]A clique is a set of nodes in which every node is connected to every other node.

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.[5]

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. N. Keng and D.Y.Y. Yun, A Planning/Scheduling Methodology for the Constrained Resource Problem, *Proceedings IJCAI'89*, pp. 998-1003, 1989; P.A. Geelen, Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems, *Proceedings ECAI'92*, pp. 31-35, 1992. In both cases, in order to select a value for the current variable, the state of the domains of the future variables which would result from each choice is found, i.e. forward checking is done for each value in turn. Keng & Yun 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 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. In general, the benefit of choosing a value which seems more likely to lead to a solution than the default choice is probably not worth the work involved in assessing each value. In particular problems, on the other hand, there may be information available which can be used to choose a value likely to succeed.

6 A Digression on Complexity

(A lot of this section is based on Kevin McEvoy's notes.)

[5]At least this is true for the algorithms considered so far. It has been reported that if an algorithm employs a fancy backtracking strategy, the value ordering can make a difference even if the problem has no solution. And I have found that in Solver, the value ordering can affect the size of the search tree when finding all solutions to a problem.

6.1 Big-O notation

Suppose we are dealing with functions which for every natural number n give us a 'result' which is a real number. If f is such a function, we write f: N $\rightarrow$ R. We will use functions of this kind to describe the running time of an algorithm in terms of the size of the problem instance it is applied to. The problem is assumed to be represented by an integer (e.g. the length of a list to be sorted) and the time by a real number (e.g. the cpu time in seconds).

Given two functions f: N $\rightarrow$ R and g: N $\rightarrow$ R, 'big-O' notation allows us to express the idea that the function g grows at least as fast as f and so will eventually be as big or bigger than f. $f = O(g)$ means:

$$\exists k, m \in \mathrm{N} \text{ such that for all } n \geq m, f(n) \leq k \times g(n)$$

So from some value of n onwards, f is no bigger than g (give or take a constant factor k).

6.2 Complexity of Algorithms

Suppose we can roughly estimate how long an algorithm takes to run, in terms of some measure of its size, n. We will not be concerned with precise measurements, just how fast the cost of running the algorithm grows with n. Suppose the algorithm has two main steps, one of which has to be executed n^2 times and the other n times. Everything else is just 'housekeeping' which has to be done whatever the size of the problem. Then the time taken to run the algorithm can be expressed in the form $f(n) = an^2 + bn + c$ where a, b, c are positive constants (the time to execute each of the two main steps just once, and the housekeeping time).

Then $f(n) = O(n^2)$ and we say that the algorithm is $O(n^2)$.

The point of this is that it gives us a simple way of comparing the running time of two algorithms: an $O(n^3)$ algorithm will take longer to solve a problem, if the problem size n is large enough, than an $O(n^2)$ algorithm.

Usually we cannot find a simple function to describe the time taken to run the algorithm in all cases; it will depend on the input to the algorithm in a particular instance. We then concentrate on the *worst case*, which gives us an upper bound on the growth rate of the time it takes to run the algorithm, or the *best case*, or sometimes the *average case*. We use the big-O notation to describe the worst case performance of the algorithm.

How long does it take to solve a CSP? Suppose we measure the size of an instance by the number of variables, n, and every variable has m values in its domain (to simplify the analysis). (Other factors might also be taken into account in measuring size, notably the number of constraints, and we shall sometimes include these.) Let's consider the generate-and-test algorithm. There are m^n possible assignments of values to variables, and in the worst case the algorithm will have to consider all of them. The worst case will occur either if there is no solution, or if there is just one solution and it happens to be the last one that the algorithm looks at. If we assume that the cost of checking an assignment is constant and does not increase as the instances get larger, then the cost of the algorithm is proportional to m^n, so this is an $O(m^n)$ algorithm. (In practice, the number of constraints is likely to increase as n increases, so that the cost of checking an assignment will also increase;

but exponential worst-case performance is bad enough, so we will disregard this extra element in the cost.) On the other hand, if an instance has a solution, the algorithm might find that the first assignment it considers satisfies the constraints; this is obviously more likely to happen the more solutions there are, but it could happen even if there is only one solution. So the best-case performance, if there is a solution, is the cost of checking a single assignment.

Any algorithm which is $O(n^k)$ for some fixed constant k is said to be a *polynomial* algorithm. An algorithm which is $O(k^n)$ for fixed k is *exponential.* The rate of growth of exponential functions is so much larger than any polynomial function that problems which have a polynomial-time algorithm are said to be *tractable*, whereas those that only have exponential-time algorithms, or worse, are said to be *intractable.*

The constraint satisfaction problem belongs to a class of problems called NP-complete[6], which have the following properties:

- No problem in the class is known to be tractable, i.e. to have a polynomial-time algorithm.
- If one problem in the class is tractable, then they all are.
- It has not been proved that these problems cannot have a polynomial-time algorithm (but it is strongly suspected that this is the case.)
- If by some means you can find a solution to an instance of an NP-complete problem, you can check whether it is a valid solution in polynomial time.

This means that although generate-and-test is the worst of the algorithms we have considered, the others are all also exponential-time algorithms. In fact, any complete CSP algorithm, i.e. any algorithm which is guaranteed to find a solution, is exponential in the worst case. Hence even the best algorithm that we have considered may take a very long time to solve a large instance.

However, although they are both exponential, there is a big difference between say 2^n and 10^n. Hence, finding a better CSP algorithm will allow us to solve bigger instances, in practice, even though we will still eventually run into trouble, in the worst case, as the number of variables increases.

If you want to read more about big-O notation, complexity of algorithms and NP-complete problems, I recommend 'Algorithmics - The Spirit of Computing' by David Harel, which is an excellent book. It gives an informal treatment and is very readable. Chapters 6 and 7 are particularly relevant ('The Efficiency of Algorithms, or Getting it Done Cheaply' and 'Inefficiency and Intractability, or You Can't Always get It Done Cheaply').

6.3 When does the worst case happen?

A considerable amount of research has been done in recent years into when you can expect to meet 'worst-case' behaviour in NP-complete problems. This research, as far as CSPs are concerned, has been done with *randomly-generated binary CSPs*;

[6]This stands for Non-deterministic Polynomial-time Complete.

since we can represent a binary constraint as a matrix of 0s and 1s, binary constraints can easily be generated randomly.

We generate randomly a large sample of instances all with the same characteristics and then measure the cost of solving each instance. For random binary CSPs, the cost is measured by the number of consistency checks required to either find a solution or to prove that there is no solution; a consistency check means referring to the matrix representing the constraint between a pair of variables to see whether a particular pair of values is allowed for that pair of variables. The number of consistency checks correlates reasonably well with the cpu time required, and is machine independent. (Note that in Solver we use the number of fails as a surrogate for cpu time; because of the way the constraints are represented and used, there is no direct equivalent to a consistency check. Also note that we don't use Solver to solve random binary CSPs.)

By measuring the cost of solving a large number of similar instances in this way, we can get an idea of the distribution of cost and in particular the average cost. If we repeat the exercise, varying the characteristics of the set of instances while keeping the size fixed, we can find out when CSPs of this kind are likely to be easy to solve and when they are likely to be difficult (i.e. when we are likely to encounter worst case behaviour).

If we have a sample of CSP instances all with the same number of variables, the same number of values in each variable's domain and the same number of constraints, it has been found that the average solution cost depends on the *constraint tightness*, i.e. the proportion of 0s in each constraint matrix. Constraints represented by matrices with a large proportion of 0s are hard to satisfy, or tight; those with a large proportion of 1s are easy to satisfy, or loose. Figure 16 shows the median cost of solving a large number of randomly-generated instances, each with 20 variables, 10 values in each variable's domain, and a *constraint density* of 0.3, i.e. 30% of variable pairs have a constraint between them. 500 instances were generated for each value of the constraint tightness, and were solved using both the simple backtracking algorithm (BT) and the forward checking algorithm with the smallest-remaining-domain variable ordering heuristic (FC with SD).

It has been found that this graph is qualitatively the same, whether we use the simple backtracking algorithm or MAC (or even something more sophisticated). The peak in cost occurs at the same constraint tightness, whatever the algorithm, but the better the algorithm, the lower the peak.

- if the constraints are extremely loose, i.e. each constraint forbids only a few pairs of values, the instances have very many solutions, and it is easy to find one without ever having to backtrack.

- as the constraints get tighter, there are still many solutions, but the algorithms have to do more search so that the cost increases.

- the peak in cost occurs when half the instances have a solution and half do not. The instances with solutions have relatively few; they are 'only just' soluble, and making a few constraints a little tighter would make them insoluble. Hence, it takes a lot of searching to find a solution. The instances which have no solutions are only just insoluble; loosening a few constraints just a little

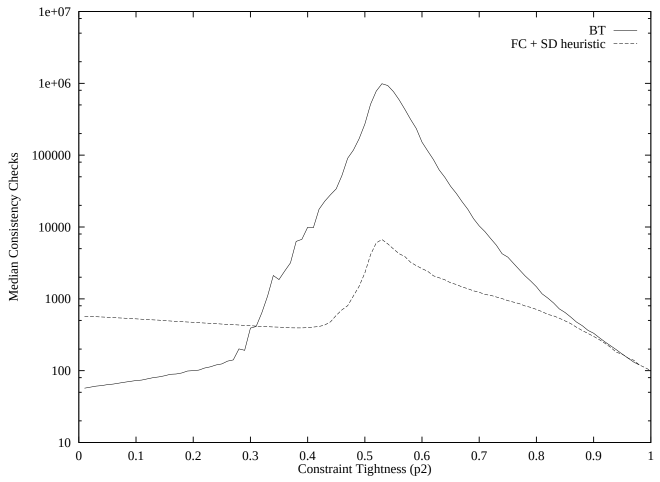


Figure 16: Median cost of solving randomly-generated CSPs

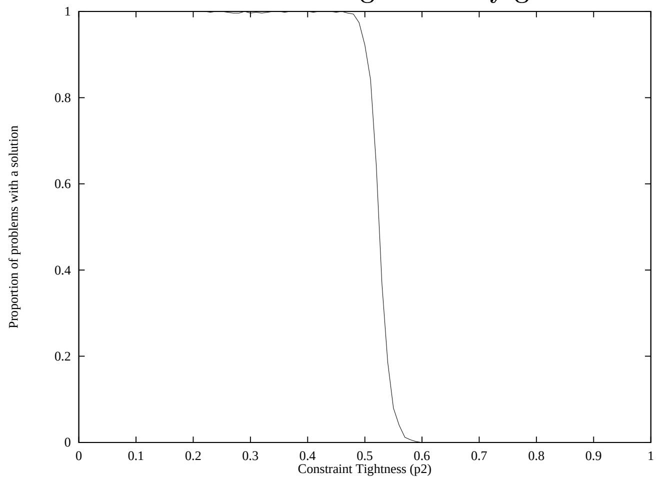


Figure 17: Proportion of problems that have a solution

would allow a solution. Hence, there are many partial solutions involving most of the variables which satisfy the constraints; it is often only possible to detect that the partial solution is inconsistent a long way down the search tree.

- When the constraints are tighter, all instances in the sample are insoluble, and as the constraints become progressively tighter, it gets easier to prove that there is no solution.

Although the experiments only tell us about the overall behaviour of *classes* of problem, and we are usually faced with solving just one instance, the research does indicate the circumstances when we are likely to run into 'worst-case' performance from our search algorithm.

7 Consistency Techniques

Consistency techniques, for example algorithms which make a binary CSP arc consistent, intend to change the original CSP into an equivalent problem which has the same set of solutions, but is hopefully easier to solve. Some consistency techniques remove values from the domains of some of the variables (as arc consistency does); others, which are less useful, add new constraints.

7.1 Node Consistency

A node consistent problem is one in which every value in every variable's domain satisfies the unary constraints on that variable. It is trivial to remove values which do not satisfy the unary constraints from the domain of the affected variable[7]: the unary constraints have then been dealt with and can be forgotten about. We have seen examples of this in Solver, e.g. in the program sendmory, where the constraint `S != 0` is used to remove this value from the domain of `S`.

7.2 Arc Consistency Algorithms

As a reminder, an arc consistency algorihm considers each binary constraint and removes any value from the domain of either variable if there is no supporting value in the domain of the other variable.

A number of arc consistency algorithms of varying degrees of sophistication have been proposed. We shall look at two: AC-3 and AC-5. AC-3 was introduced by Mackworth in 1977 (A.K. Mackworth, Consistency in networks of relations, *Artificial Intelligence*, **8**, pp. 99-118, 1977). AC-5 (P. van Hentenryck, Y. Deville and C.-M. Teng, A generic arc-consistency algorithm and its specializations,*Artificial Intelligence*, **57**, pp. 291-321, 1992) is more complex, but more efficient, and furthermore can be adapted to re-establish arc consistency during search, which is what is required for the full-lookahead algorithm.

In the algorithm descriptions, C is the set of binary constraints in the problem. (There may be some non-binary constraints, but, at least for the time being, we shall only use the binary constraints.) If there is a binary constraint between the variables x and y, we will refer to it as either C_{xy} and C_{yx}. We assume that at the start, the problem is already node consistent, so that there are no longer any active unary constraints. All constraints in C are assumed to be at least binary.

[7]at least in the cases we shall see: we can imagine constraints which are hard to check, e.g. if the domain of x consists of extremely large integers and the constraint is 'x is prime'

```
Algorithm AC-3
begin
    { form a queue consisting of all arcs $(x, y)$ where there is a constraint
      $C_{xy}$ between the variables $x$ and $y$ }
    $Q \leftarrow \{(x, y) \mid C_{xy} \in C\}$;
    { while the queue is not empty ... }
    while $(Q \neq \emptyset)$ do
        begin
            { choose an arc $(x, y)$ from the queue and remove it }
            $Q \leftarrow Q - \{(x, y)\}$;
            { delete any values in $D_x$ which are not supported by $y$ }
            Deleted $\leftarrow$ false;
            for each $i \in D_x$ do
                if $\forall j \in D_y : x = i, y = j$ does not satisfy $C_{xy}$ then
                    begin
                        $D_x \leftarrow D_x - \{i\}$;
                        Deleted $\leftarrow$ true;
                    end
            { if any values of $x$ have been deleted, Deleted is true }
            if Deleted then
                $Q \leftarrow Q \bigcup \{(z, x) \mid C_{zx} \in C \wedge z \neq y\}$;
        end;
end; { of AC-3}
```

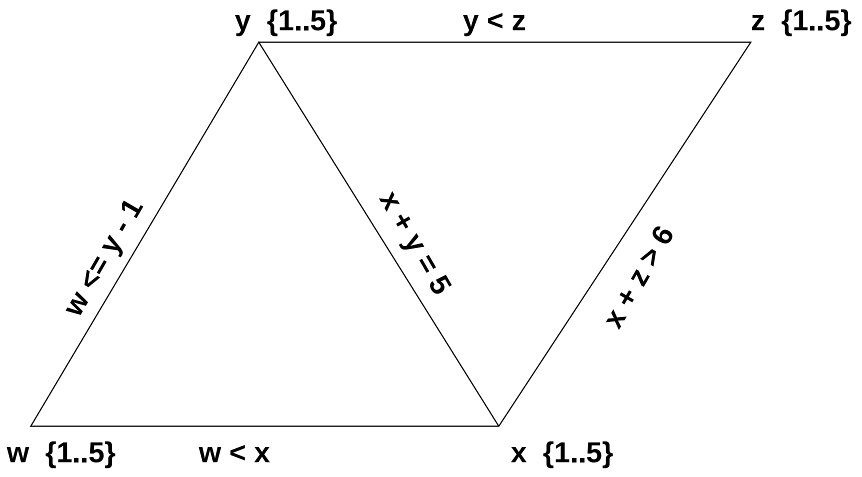


Figure 18: Example constraint graph

Arcs in the queue	Values deleted	Arcs added	*Elements added by AC-5*
x → y y → x x → z z → x y → z z → y x → w w → x y → w w → y			

7.2.1 Complexity of AC-3

To see how long AC-3 takes to run, we estimate how many times the algorithm has to test whether a pair of values, say $x = i, y = j$, satisfies the constraint between x and y, in the worst case.

Let m be the maximum size of any domain and let c be the number of edges in the constraint graph (i.e. the number of binary constraints).

In processing an arc, AC-3 considers at most m^2 pairs of values, if every value in D_x is supported by only the last value in D_y, or by none of them. (At *best*, it considers only m pairs of values, if every value in D_x is supported by the *first* value in D_y.)

Initially, the queue contains $2c$ arcs. In the worst case, processing an arc will delete exactly one value from D_x and will cause arcs to be added to the queue each time. Any arc (y, x) can be added to the queue at most $|D_x|$ times (it is added to the queue whenever a value is deleted from D_x, which cannot happen more than $|D_x|$ times). Hence, AC-3 processes at most $2c(m + 1)$ arcs. So the overall time complexity of AC-3 is $O(2cm^3)$.

Note that the *result* of making a problem arc consistent is unique. The order in which the arcs are taken from or added to the queue does not affect the final set of domains, although it may affect the time taken to reach the result.

7.2.2 The AC-5 Algorithm

The queue in AC-3 consists of arcs (x, y). In AC-5, the queue contains elements $\langle(x, y), j\rangle$, where (x, y) is an arc and j is a value which has been removed from D_y and so justifies the need to reconsider this arc.

```
Algorithm AC-5
begin
    Q ← $\emptyset$;
    { consider each arc once }
    for each $C_{xy} \in$ C do
        begin
            { find $\Delta_1$, the set of values removed from $D_x$ as a result of
              considering the arc $(x, y)$ }
            $\Delta_1 \leftarrow \{i \in D_x \mid \forall j \in D_y : x = i, y = j$ does not satisfy $C_{xy}\}$;
            { add elements to the queue for each value in $\Delta_1$ }
            Q ← Q $\bigcup\{\langle(z, x), i\rangle \mid C_{zx} \in$ C and $i \in \Delta_1\}$;
            { remove the values in $\Delta_1$ from the domain of $x$ }
            $D_x \leftarrow D_x - \Delta_1$;
        end;
    { now look at the elements added to the queue on the first pass (and new
      elements still to be added) }
    while (Q $\neq \emptyset$) do
        begin
            { choose an element from the queue and remove it }
            Q ← Q $-\{\langle(x, y), j\rangle\}$;
```

```
                { find the set of values removed from $D_x$ as a result of
                    considering this element }
                $\Delta_2 \leftarrow \{i \in D_x \mid x = i, y = j$ satisfies $C_{xy} \wedge$
                            $\forall k \in D_y : x = i, y = k$ does not satisfy $C_{xy}\}$;
                { add elements to the queue }
                Q $\leftarrow$ Q $\bigcup\{\langle (z, x), i\rangle \mid C_{zx} \in$ C and $i \in \Delta_2\}$;
                { remove the values in $\Delta_2$ from the domain of $x$ }
                $D_x \leftarrow D_x - \Delta_2$;
            end;
    end; { of AC-5 }
```

AC-5 has two phases: in the first, we go through all the arcs once. For each arc, we find Δ_1, the set of values to delete, building up a queue of elements $\langle (x, y), j\rangle$ as we go. In the second phase, we process the queue and for each element we again compute the set of values to be deleted, Δ_2.

In AC-3, whenever an arc (x, y) is added to the queue, as a result of one or more values being deleted from D_y, then for each value of x, we potentially have to check every value of y to see if there is one that supports this value of x. This is wasteful: unless a value of x was previously supported by one (or more) of the values deleted from D_y, it will still be supported by at least one of the values left in D_y. So the computation of Δ_2 for an element $\langle (x, y), j\rangle$ in the second phase of AC-5, i.e.

$$\Delta_2 \leftarrow \{i \in D_x \mid x = i, y = j \text{ satisfies } C_{xy} \wedge \forall k \in D_y : x = i, y = k \text{ does not satisfy } C_{xy}\};$$

finds out which values of x should now be deleted because their only support in D_y was j (and possibly other values which have now been deleted along with j). For each possible value of x, say i, we first check that j supported i, i.e. that $x = i, y = j$ satisfies C_{xy}. We then find out whether there is any other supporting value amongst the values remaining in D_y; if not, i should be deleted from D_x, and is added to Δ_2.

7.2.3 Complexity of AC-5

In the first phase of AC-5 we go through the $2c$ arcs in the constraint graph, so we compute Δ_1 at most $2c$ times. There are (at most) $2cm$ possible elements $\langle (x, y), j\rangle$, and each one can be added to the queue at most once: so we compute Δ_2 at most $2cm$ times.

Each computation of Δ_1 is equivalent to the loop in AC-3 which checks whether any value should be deleted from the domain of a variable x because of the constraint C_{xy}, and so requires at most m^2 tests to see whether a pair $x = i, y = j$ satisfies C_{xy}.

However, the computation of Δ_2 similarly requires at most m^2 tests. If $\langle (x, y), j\rangle$ is the element selected from the queue, we need to test $x = i, y = j$ for every value i in D_x. At worst, if $y = j$ supported every value in D_x and none of the remaining values in D_y do, we shall have to test every possible pair of values $x = i, y = k$ for all i in D_x and all k in D_y. Overall, this requires up to m^2 tests.

So it appears that the worst-case complexity of AC-5 is no better than that of AC-3. In practice, however, it will be quicker.

AC-5 has other significant advantages over AC-3. First, it allows the computation of Δ_1 and Δ_2 to be tailored to each type of constraint, and there are many commonly-occurring constraint classes for which they can be computed very quickly. Secondly, AC-5 can be easily adapted for integration with the full-lookahead algorithm.

There are AC algorithms which have lower complexity ($O(cm^2)$) than AC-3 and AC-5, in general, notably AC-4. However, AC-4 and similar algorithms require more complex data structures: AC-4 for instance is based on the idea of *counting* how many supporting values there are in D_y for each value $x = i$, for every arc (x, y). Unless at least this many values are deleted from D_y, we know that $x = i$ must still have at least one supporting value, and so does not need to be checked. This algorithm has a time-consuming first phase in which all the supporting values are counted; thereafter it runs more quickly than AC-3. However, it is not used in any of the constraint programming tools: it is hard to see how it could be integrated with full-lookahead in order to deal with non-binary constraints which effectively become binary as the variables involved are assigned values.

7.2.4 Monotonic constraints

As an example of how AC-5 can be tailored for particular types of constraint, we here consider monotonic constraints. A monotonic constraint C_{xy} on integer variables[8] x and y is such that:

if $x = i, y = j$ satisfies C_{xy} and $i' \leq i$ and $j' \geq j$ then $x = i', y = j'$ satisfies C_{xy}. $x \leq y + 3$ is a monotonic constraint: for instance, $x = 1, y = 4$ satisfies the constraint and so does $x = i', y = j'$ for any $i' \leq 1$ and $j' \geq 4$. $x + y = 5$ is not monotonic: $x = 1, y = 4$ satisfies the constraint, but $x = 0, y = 6$ does not.

(Note that if C_{xy} is monotonic according to the above definition, then C_{yx} is not; the algorithms below for computing Δ_1 and Δ_2 need to be adapted for the reverse constraint C_{yx}, for instance replacing *max* by *min*.)

```
**Computation of $\Delta_1$ if $C_{xy}$ is monotonic**
begin
    $\Delta_1 \leftarrow \emptyset$;
    { find the largest integer $i^*$ such that $x = i^*, y = max(D_y)$ satisfies $C_{xy}$ }
    { $i^*$ is not necessarily in $D_x$. We assume that $i^*$ can be directly computed
      from $C_{xy}$, e.g. if the constraint is $x \leq y - 3$, $i^* = max(D_y) - 3$ }
    $i^* \leftarrow max\{i' \mid x = i', y = max(D_y)$ satisfies $C_{xy}\}$;
    $i \leftarrow max(D_x)$;
    { add to $\Delta_1$ all the values in $D_x$ which are larger than $i^*$ }
    while $(i > i^*)$ and $(i \geq min(D_x))$ do
        begin
            $\Delta_1 \leftarrow \Delta_1 \bigcup \{i\}$;
            $i \leftarrow i - 1$;
        end
end
```

[8](or any other type of variable whose domain has a total ordering).

For example, if the domains of x and y are $\{1..5\}$ and the constraint is $x \leq y-3$, then $max(D_y) = 5$, $i^* = 2$ and $\Delta_1 = \{3,4,5\}$.

In computing Δ_2, we observe that if any value in D_y supports a value in D_x, so does any larger value of y. It is only when the largest value in D_y has been removed, i.e. the maximum has been reduced, that we may need to remove further values from D_x.

```
Computation of $\Delta_2$ for the element $\langle (x,y), j \rangle$ if $C_{xy}$ is monotonic
begin
        $\Delta_2 \leftarrow \emptyset$;
        { deleting $j$ makes no difference to $D_x$ if $j < max(D_y)$, for monotonic
          constraints (and $j$ cannot be equal to $max(D_y)$ since $j \notin D_y$)}
        if  $j > max(D_y)$ then
                begin
                        $i^* \leftarrow max\{i' \mid x = i', y = max(D_y)$ satisfies $C_{xy}\}$;
                        $i \leftarrow max(D_x)$;
                        while $(i > i^*)$ and $(i \geq min(D_x))$ do
                                begin
                                        $\Delta_2 \leftarrow \Delta_2 \bigcup \{i\}$;
                                        $i \leftarrow i - 1$;
                                end
                end
end
```

e.g. Suppose the constraint between x and y is $x < y$ and both variables initially have domain $\{1..5\}$. In the first phase these will be reduced to $\{1..4\}$ for D_x and $\{2..5\}$ for D_y. Then suppose that as a result of some other constraint involving y, D_y is reduced to $\{3\}$.

We will then have three elements $\langle (x,y), 2\rangle$, $\langle (x,y), 4\rangle$ and $\langle (x,y), 5\rangle$ in the queue. Since $max(D_y) = 3$, the deletion of $y = 2$ makes no difference to x.

When we process the element $\langle (x,y), 4\rangle$, $i^* = 2$ and we delete any values in D_x which are greater than i^*, i.e. $\Delta_2 = \{3,4\}$. Next we process $\langle (x,y), 5\rangle$ and again $i^* = 2$, but now $max(D_x) = 2$, so we make no further deletions.

So, if we consider the values deleted from the top end of the range of y in ascending order, it is only the smallest that creates a non-empty Δ_2. (Solver creates the same Δ_2 for monotonic constraints by only considering the reduction in the *range* of y, i.e. it computes Δ_2 based on the new domain maximum, rather than on the individual values deleted. The result will be the same, but the Solver version is more efficient.)

Several other classes of constraint can similarly be treated very efficiently, giving an overall complexity for AC-5 for these constraints which is O(cm). For instance, although $x + y = 5$ is not monotonic, it is *functional*, i.e. for each value of x there is at most one value in D_y which satisfies the constraint. (Van Hentenryck, Deville and Teng (cited earlier) give more details of these constraint classes, and the proof that their complexity is $O(cm)$.)

For any class of constraint, the computation of Δ_1 and Δ_2 can be written specifically to make use of any special properties of the constraint, and to be as efficient as possible. Solver, for instance, does this for its built-in constraints, and users who define their own constraints are expected to define how the constraint propagates (roughly, how to do forward checking and how to compute Δ_2) themselves.

7.2.5 Combination of AC-5 and full lookahead

We can use an arc consistency algorithm both as a pre-processing step and also during search, as in the full lookahead algorithm.

Suppose the algorithm has chosen a variable to assign and a value to assign to it. At this point, the problem is arc consistent (because the algorithm made it arc consistent at the start and has kept it so ever since). Hence, it is only necessary to propagate the effects of the changes resulting from this assignment.

The algorithm does forward checking from the assignment, i.e. it removes values from the domains of future variables which are not consistent with it. At the same time, a queue of elements $\langle (x, y), j \rangle$ is formed, where j is a value which has just been removed from the domain of y, and x is some other future variable. (We can at this point take account of non-binary constraints: the constraint between x and y can be of any arity as long as these are the only unbound variables involved in it.) The algorithm then processes this queue, adding new elements to it if necessary, as in the second phase of AC-5.

By this process, re-establishing arc-consistency can be done very efficiently: it is only necessary to follow up the effects of deleting values in the forward checking step.

7.2.6 The Importance of Complexity in AC Algorithms

We have seen that AC-5 has complexity at worst $O(cm^3)$ where c is the number of (binary) constraints and m is the maximum domain size; and for many types of constraint, the algorithm can be made to run much faster ($O(cm)$).

This compares very favourably with the cost of searching for a solution. In the worst case, as we have seen, the time taken to solve a CSP increases exponentially with the size of the problem, e.g. it increases like k^n, where k is a constant (probably depending in some way on m).

The time taken to make a problem arc consistent increases roughly like n^2 (because c, the number of edges in the constraint graph, is at most $n(n-1)/2$). Hence, even if making a small problem arc consistent may not give a time-saving overall (because it may take very little time to find a solution anyway), as problems get larger, the time taken to achieve arc consistency rapidly becomes relatively insignificant *and* the time saved by deleting values from the domains of some of the variables becomes potentially very large. If the original size of every domain is m then removing just one value from the domain of one variable reduces the number of possible assignments which potentially need to be considered from m^n to $m^{n-1}(m-1)$, i.e. m^{n-1} possible assignments are immediately ruled out.

So making a problem arc consistent at the start potentially saves a lot of search effort and is not expensive to do: the algorithm is quick and we only run it once. Integrating constraint propagation with search, as in full lookahead, is a different matter; since we then run an arc consistency algorithm after *every* variable instantiation, we cannot afford to spend much time on it. If implemented as described above (i.e. so that it is incremental, and has linear complexity for many commonly-occurring constraint types), it has been shown to be cost effective.

7.3 Path Consistency

In Figure 11, 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 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.)

In Figure 19, suppose there are non-trivial constraints between x_i and x_j, and between x_j and x_k.

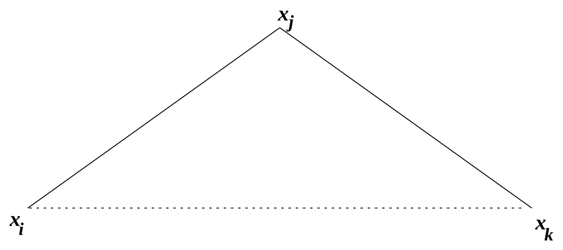


Figure 19: A triangle of constraints

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_{ik} 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 v_i and v_k to be simultaneously assigned x_i and x_k respectively.

In the example of Figure 11, 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.

Mathematical note: if we represent the constraints by matrices, as described earlier, we can achieve path consistency by multiplying matrices (although this is not the most efficient way to do it). In Figure 11, the relevant constraint matrices are:

$$R_{xw} : \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad R_{wz} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix} \quad R_{xz} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

If we now multiply R_{xw} and R_{wz} (bearing in mind that the values are booleans, so that any value > 0 is written as 1) we get the constraint *induced* between x and z by the path (x, w, z):

$$R_{xw} \times R_{wz} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

We can then combine R_{xz} and $R_{xw} \times R_{wz}$ to get the new constraint between x and z: a pair of values must be allowed (i.e. have an entry of 1) in both the induced constraint and the original constraint:

$$R_{xz} \leftarrow R_{xz} \wedge R_{xw} \times R_{wz} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

so that, as we have already seen, x cannot have the value 2.

We can make a binary CSP path consistent by checking every triple of variables and tightening the constraints where necessary. The number of possible triples is much 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 $O(m^3n^3)$. This is one reason why path consistency algorithms are not in common use: it would almost certainly be out of the question to integrate a path consistency algorithm with search, as we can do with arc consistency. 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 Solver, each pair of values removed would have to be represented by a new constraint, e.g. to prevent X=1 and Y=2 being simultaneously assigned, the constraint

```
!((X==1) && (Y == 2))
```

(or an equivalent constraint) must be posted.

The *induced* or *implied* constraints required for path consistency are occasionally easy to see when the problem is formulated. For instance, in the triangle of constraints in Figure 20 (from the zebra problem), path consistency shows that the constraint between *red* and *spain* must be *red* $\neq$ *spain*. Because the Englishman lives in the red house, and the Englishman and the Spaniard live in different houses, the Spaniard does not live in the red house.

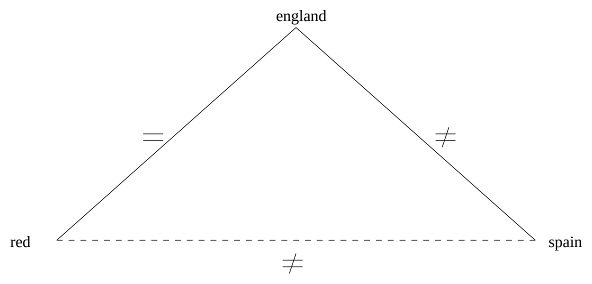


Figure 20: Example of an induced constraint between 'red' and 'spain'

7.4 k-consistency

It is also possible, when we have non-binary constraints, to consider groups of three or more variables and attempt to induce new constraints (which will also in general be non-binary).

To make a problem k-consistent ($k > 1$) we have to consider all possible subsets of $k-1$ variables, say $x_1, x_2, ..., x_{k-1}$, and all possible tuples of values $(a_1, a_2, ..., a_{k-1})$ that satisfy the constraints on these variables, and check that for every possible choice of a kth variable x_k there is at least one value a_k in its domain such that the tuple $(a_1, a_2, ..., a_{k-1}, a_k)$ satisfies the constraints on the k variables. If not, the tuple $(a_1, a_2, ..., a_{k-1})$ must be removed from the constraint $C_{x_1 x_2 .... x_{k-1}}$ (which may not previously have been specified, i.e. may have been the trivial constraint).

Note that arc consistency is 2-consistency and (if there are only binary constraints) path consistency is 3-consistency. The induced constraints found by this process are $(k-1)$-ary, and in the form of forbidden tuples of values. These again cannot easily be handled by constraint programming software like Solver, and in any case are less useful than deleted values (as in arc consistency) and binary constraints (as in path consistency), since the search algorithms can only make use of non-binary constraints when all but two variables have been instantiated. Making a problem k-consistent also rapidly becomes very time-consuming as k increases. However, as with path consistency, if it is possible to identify implied constraints when formulating the problem, they should usually be included in the model.

7.5 Generalised Arc Consistency

The essential difference between arc consistency and the higher levels of consistency just discussed is that arc consistency considers only one constraint at a time; path consistency for instance, considers the combination of two constraints to induce a third. We can generalise the idea of arc consistency to non-binary constraints. We again consider a single variable in relation to a single constraint and try to remove values from the domain of this variable if they are not supported by the other variables involved in the constraint.

Suppose there is a constraint $C_{x_1 x_2 x_3 ... x_k}$. If for any value $a_1 \in D_1$ there is no set of values $a_2, a_3, ..., a_k$ for the variables $x_2, x_3, ..., x_k$ such that $(a_1, a_2, a_3 ..., a_k)$ satisfies $C_{x_1 x_2 x_3 ... x_k}$, then a_1 can be removed from the domain of x_1.

A problem is arc consistent in this generalised sense if, for every value a_i of every variable x_i, for every constraint $C_{x_i x_j ..... x_k}$ that this variable is involved in, there is a set of values $a_j, ..., a_k$ such that $(a_i, a_j, ..., a_k)$ satisfies the constraint. (NB Just as both arcs (x, y) and (y, x) have to be checked when there is a binary constraint between x and y, in the general case, each variable $x_i, x_j, ..., x_k$ has to be checked separately to see whether any values in its domain should be deleted.)

Removing values from the domains of variables is a more useful way of reducing the problem than tightening the constraints by removing individual tuples of values, so generalised arc consistency for constraints of arity k is a more useful property, if it can be achieved, than k-consistency.

General algorithms for achieving generalised arc consistency are too expensive to apply. In specific types of constraint, however, it may be economical to make at least some reductions to the domains of the variables involved.

Arithmetic constraints are a clear example where it is economical to look for some reductions in the domains of the variables, but not to look for all possible reductions. For instance, Solver uses the constraint `SEND + MORE == MONEY`, where the variable domains are `SEND: {1000..9999}, MORE: {1000..9999}, MONEY: {10000..99999}`, to reduce the domain of the variable `MONEY` to `{10000..19998}`. This can be done very quickly by reasoning about the maximum and minimum values of the variables *(bounds consistency).*

However, in the `DONALD + GERALD = ROBERT` puzzle, we have a constraint

```
2*D == T + 10*C1
```

where the domains of the variables are `D: {1..9}, T: {0..9}, C1: {0,1}`. If we applied a generalised arc consistency algorithm to this constraint, we would find that `T` must be even. Solver misses this inference, although it is obvious to us. Using a GAC algorithm even in this case, where the domains are small, would be time-consuming in relation to the size of the problem. In this constraint alone, we should have to check for reductions to the domains of `D` and `C`, as well as `T`. Since we cannot reduce the domains of `D` and `C`, this would be wasted effort.

Solver is a compromise between efficiency and completeness...In the example... [of constraint propagation of arithmetic constraints] the incompleteness comes from the fact that arithmetic expressions only propagate bounds..

This is an example of the choice we made. Propagating holes in expressions would require much more memory and time than the current implementation. ¿From tests made on a very large set of examples, we found that the current compromise is by far better." Jean-Francois Puget, ILOG

7.5.1 The 'allDifferent' constraint

A common constraint where there is an efficient algorithm for achieving full generalised arc consistency is the allDifferent constraint. Solver has a constraint IlcAllDiff which can be applied to an array of any number of variables and which specifies that they must all have different values. By default, the IlcAllDiff constraint behaves like a collection of binary $\neq$ constraints.

However, this may miss opportunities to reduce the domains of variables. For instance, if we have four variables x_1, x_2, x_3, x_4 each with domain {1,2,3}, then an allDifferent constraint cannot be satisfied. However, each $\neq$ constraint can be satisfied, if considered individually.

In fact, if we treat an allDifferent constraint as a set of binary $\neq$ constraints, we can *only* remove values from the domains of the variables if a value has been assigned to one of the variables. Then that value can be removed from the domains of every other variable in the allDifferent constraint. (Note that $\neq$ is an anti-functional constraint.)

Régin[9] gives a specialised algorithm for the allDifferent constraint which achieves generalised arc consistency relatively cheaply.

For example, suppose we have an allDifferent constraint between the following variables:

Variable	Original domain	Reduced domain
x_1	{1,2}	{1,2}
x_2	{2,3}	{2,3}
x_3	{1,3}	{1,3}
x_4	{2,4}	{4}
x_5	{3,4,5,6}	{5,6}
x_6	{6,7}	{6,7}

If we use this algorithm, the constraint is propagated whenever there is *any* change to the domain of any variable in the constraint.

The time complexity of the algorithm, for a single allDifferent constraint, is $O(p^2d^2)$, where p is the number of variables involved in the constraint and d is the size of the union of their domains (i.e. the number of values which they share). Hence it is much more time-consuming than treating the constraint as a collection of binary constraints, though much more efficient than the general-purpose GAC algorithms.

Solver also has a consistency algorithm for the allDifferent constraint which propagates whenever there is a change to the *range* of one of the variables in the constraint. This is intermediate, in terms of the amount of domain reduction it can do and the time it takes to run, between the other two.

The user can choose how an allDifferent constraint should be propagated when the constraint is defined, by choosing the type of propagation event when the constraint should be propagated (value, range or domain). Typically, choosing the full GAC algorithm reduces the number of fails, in comparison with the basic propagation of $\neq$ constraints, but increases the cpu time overall. However, there could be cases where it does reduce the running time.

7.6 How Solver Really Works

Having looked in more detail at constraint propagation algorithms, we can reconsider how the integration of search and maintaining arc consistency works in Solver. Whenever a constraint is defined (in no-edit mode) or as soon as Solver starts searching for a solution (in edit mode), the constraints that have been posted are propagated, and this continues until no further deductions can be made. Solver then searches for a solution by repeatedly selecting a variable *var* and then a value for that variable, *val*. It then constructs a choice between two alternatives: $var = val$

[9] A filtering algorithm for constraints of difference in CSPs, in *Proceedings AAAI-94*, pp. 362-367, 1994.

or $var \neq val$. Each gives a branch in the search tree; the first is explored first. The choice defining the branch is added to the CSP as a (possibly temporary) constraint. The effects of that constraint are propagated, and again this continues until no further deductions can be made *or* until an empty variable domain is encountered, in which case this branch fails. If the branch does not fail, another variable is selected and a value for it; otherwise, the constraint $var = val$ is retracted, the constraint $var \neq val$ is imposed, and an alternative value for the variable *var* is chosen.

During constraint propagation, a constraint is considered if one of the propagation events (value, range or domain) has occurred for one of the variables involved in the constraint, and some action for this constraint following this kind of propagation event has been specified. For instance, the $var = val$ constraint triggers the value event for the variable *var* (i.e. *var* has been assigned the value *val*). Since a value event is also both a domain event and a range event, any constraint in the problem which involves the variable *var* will be activated, and some domain reduction of the other variables in those constraints is likely to follow. These reductions in turn may have an effect on the domains of other variables.

This is essentially the MAC algorithm, although its implementation is somewhat different from what we saw before. In particular, there is no way of distinguishing a 'forward checking' phase from a 're-establishing arc consistency in the future subproblem' phase.

The search procedure that I have described is what IlcGenerate does. However, it is reasonably straightforward to rewrite IlcGenerate in order to search in a different way (the code for IlcGenerate is given in the Solver manual). For instance, it might be better in some problems to search by splitting the domain of the chosen variable, rather than committing to a specific value. So we could create a choice point between say $var < mid$ and $var \geq mid$, where *mid* is the current midpoint of *var*'s domain.

8 Static Variable Ordering Heuristics

A number of general static variable ordering heuristics have been developed, but in practice they are not much used. They have been developed for algorithms such as the simple backtracking algorithm, where you have to use a static ordering rather than a dynamic ordering, because the information needed for a dynamic ordering is not available. But the simple backtracking algorithm is so much worse than the other available algorithms that it is no longer used in practice. If you are using an algorithm which looks ahead, such as forward checking or MAC, you could in theory use either a static or a dynamic ordering. However, most if not all of the research on general ordering strategies has been done on binary CSPs, and for such problems the best available dynamic ordering heuristics (e.g. choose a variable with smallest remaining domain; break ties by choosing the variable which constrains the largest number of unassigned variables) are better than the available static ordering heuristics.

The situation is different for non-binary problems. You may then have a constraint which will not be propagated until most of the variables in it have been assigned values and in that case it is often better to use the constraints to guide the variable ordering, i.e. choose a constraint, rather than a variable, and assign the variables involved in the constraint, until constraint propagation can take place. Or there may be a specific ordering of the variables indicated by some underlying structure in the problem.

This happens, for instance, in the Golomb ruler problem. A Golomb ruler may be defined as a set of m integers $0 = a_1 < a_2 < ... < a_m$ such that the $m(m-1)/2$ differences $a_j - a_i, 1 \leq i < j \leq m$ are distinct. Such a ruler is said to contain m marks and is of length a_m. The objective is to find optimal (minimum length) or near optimal rulers. These problems are said to have many practical applications including sensor placements for x-ray crystallography and radio astronomy.

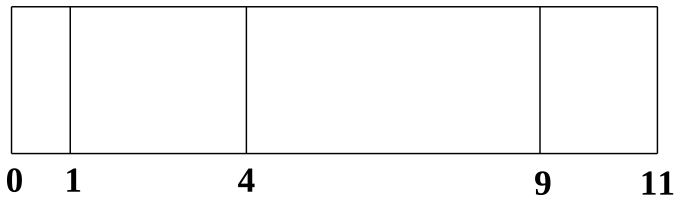


Figure 21: Minimum length Golomb ruler with 5 marks

This problem can be modelled as a CSP by using variables corresponding to the marks $a_2, a_3, ..., a_m$ (a_1 is fixed). We can also use additional variables d_{ij} to represent the differences $a_j - a_i \quad (j > i)$. Experiments with this problem have shown that choosing variables as if we were building up the ruler from left to right solves the problem much more easily than the smallest-domain ordering. The smallest-domain ordering tends to 'jump' from one part of the ruler to another. Possibly the reason why this is a bad strategy is that the smallest available values are assigned first; if we assign a small value to a mark which is some way along the ruler from those we have already assigned, this is more likely to be a wrong choice (or to put it another way, we may not have left enough room for the intervening marks).

So, static orderings have their uses, particularly when the constraints are non-binary, but the ones used in practice tend to be chosen with a specific problem in mind. However, we shall consider one of the general purpose static ordering heuristics, because it leads on to the next main topic.

8.1 Minimum Bandwidth Ordering

This heuristic (like most, if not all, general-purpose static variable ordering heuristics) has been developed for binary CSPs, which can be represented by a constraint graph.

If we order the nodes in a (constraint) graph, the *bandwidth* of a node v is the maximum distance in the ordering between v and any other node such that there is an edge joining joining it to v (i.e. there is a constraint between the two variables). The bandwidth of an ordering is the maximum bandwidth of all the nodes in the graph under that ordering.

In the pathological example considered in section 4.3, to show that FC can do arbitrarily better than BT, the variables $x_1, x_2, ..., x_n$ were considered in *maximum* bandwidth order, since there is a constraint between x_1 and x_n. The poor performance of BT was caused by having to backtrack from x_n to x_{n-1} to ... to x_2

and then x_1 in order to reach the source of the failure, taking an enormous amount of time. A minimum bandwidth ordering would presumably place x_1, x_2, x_n close together (if the other constraints allow), thus avoiding the difficulty. The intuition behind this heuristic is therefore that variables that constrain each other should be placed as close together as possible in the ordering, so that backtracking can be minimised.

Finding the minimum bandwidth ordering is in itself a difficult problem, but an approximate algorithm which will give a small bandwidth ordering quickly can be used. However, the difficulty which the minimum bandwidth heuristic is addressing is that an algorithm might have to backtrack a long way before arriving at the true cause of the failure. A more intelligent approach is to redesign the algorithm so that it backtracks directly to a possible cause of the failure, rather than backtracking to unrelated variables which cannot possibly have any role in the failure.

9 Intelligent Backtracking Algorithms

All the backtracking search algorithms for constraint satisfaction that we have met are both sound and complete. Soundness means that any solution that they find is a genuine solution, satisfying all the constraints. Completeness means that if there is a solution they will find one, and if there is no solution they will prove it. However, in carrying out the search, the algorithms sometimes do unnecessary work. Each improvement to the simple backtracking algorithm was introduced to eliminate an observed inefficiency, and we now extend this to inefficiencies in backtracking, while maintaining soundness and completeness.

So far algorithms have always backtracked to the previous variable when they encounter a failure; this is called chronological backtracking. Why is this (sometimes) inefficient, and how can the inefficiency be avoided?

Suppose the algorithm has attempted to assign a value to a variable x_i and failed. This means that the current partial solution, consisting of the current assignments to the previous variables $x_1 = l_1, x_2 = l_2, ..., x_{i-1} = l_{i-1}$ cannot be consistently extended to include an assignment to x_i; at least one of these assignments will have to be changed. A chronological backtracking algorithm undoes the last assignment (i.e. to x_{i-1}) and tries another; assuming that the new assignment is consistent (either with the past assignments or the future variables, depending on the algorithm), the algorithm again tries to extend the partial solution to x_i.

However, what if the assignment to x_{i-1} has nothing to do with the failure to assign a value to x_i? An obvious example of this is when there is no constraint between x_i and x_{i-1}. In that case, changing the value assigned to x_{i-1} will make no difference to x_i: once again attempting to assign a value to x_i will fail. The algorithm will again backtrack to x_{i-1} and try yet another value for it, and this will be repeated until all values for x_{i-1} have been tried. Only then will the algorithm backtrack to x_{i-2}.

If we can instead identify the most recently-assigned variable whose assignment could possibly be responsible for the failure to assign a value to x_i, the algorithm could jump back immediately to that variable. This would avoid stepping back to each of the intervening variables in turn, and fruitlessly trying alternative assignments for those variables, none of which will make any difference to the failure.

How do we choose the variable to backtrack to, say x_h? As I have suggested, there must at least be a constraint between x_h and x_i. We can impose a stronger condition and insist that at least one of the possible assignments to x_i failed because it was inconsistent with the current assignment $x_h = l_h$. In simple backtracking, this means that when the algorithm was trying to assign a value to x_i and checking against past assignments, at least one value of x_i was found to be inconsistent with $x_h = l_h$ (and so was not inconsistent with the assignments $x_1 = l_1, ..., x_{h-1} = l_{h-1}$). In forward checking, it means that when the assignment $x_h = l_h$ was made, at least one value was removed from the domain of x_i as a result. And we must choose to jump back to the most recent such variable, otherwise we might miss a solution and the algorithm would no longer be complete.

Having identified the variable x_h to jump back to, what should we do about the intervening variables? Some of the current assignments will have to be undone, because values for these variables which were previously ruled out by the assignments to x_h will now be possible and should be chosen instead. But some of these variables may be unaffected by the reassignment to x_h: this would be true, for instance, if one of these variables say x_j is not constrained by x_h nor by any of the variables between x_h and x_j. In that case, when x_h is assigned a new value, and the variables $x_{h+1}, ..., x_{j-1}$ are assigned values, x_j will be assigned the value which it had previously. So in theory, it would be possible to leave some of the variables between x_h and x_i with their current instantiations. In practice, however, this is difficult to manage (although there is an algorithm that does it) and instead, we simply undo all the assignments back to x_h.

The second, more serious, complication is: what should happen if x_h in turn has no alternative values left and the algorithm has to backtrack again? We cannot just repeat the process and jump back to the most recent variable which had an effect on x_h because that ignores the fact that we are primarily trying to find a way of assigning a value to x_i. It may be that the most recent variable responsible for the fact that there are no other values to try for x_h is (say) x_f, but that x_g, which was assigned a value more recently than x_f, was also responsible for the failure to find an assignment for x_i. It could be that simply by assigning a new value to x_g the algorithm will be able to find a value for x_i while still keeping the rest of the original partial solution, including the current assignment to x_f. Again, if we jump back to x_f rather than x_g the algorithm will no longer be complete. Some algorithms avoid this complication by jumping back from the first failure, but then if further backtracking is required before moving forward again, stepping back chronologically. However, the algorithm we shall look at always jumps back (if possible) to the most recent variable whose assignment could be implicated in the failures to assign values to any of the variables it is backtracking from.

9.1 Implementing Backtracking Algorithms

We can implement any of the backtracking search algorithms we have considered using two basic procedures (which Patrick Prosser[10] of Glasgow University calls

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

label and unlabel) which are repeatedly called in turn, until either a solution is found or it has been shown that the problem has no solution; implementing these procedures in different ways will give us different algorithms. (There are of course other ways of implementing these algorithms: we could for instance use recursion.)

The procedure label attempts to find a consistent assignment for the current variable x_i. If it succeeds, it returns consistent = **true** and V[i], which contains the value assigned to the variable. If it fails, it returns consistent = **false**. In either case, it returns a variable *next* indicating which variable label should assign next or which variable unlabel should backtrack from.

So label can terminate in one of three states:

1. consistent = **true**, $1 \leq next \leq n$: on the next iteration, label is called again to try to assign a value to the next variable.
2. consistent = **true**, $next = n + 1$: a value has been consistently assigned to every variable and so a solution has been found and the program terminates.
3. consistent = **false**, $1 \leq next \leq n$: on the next iteration, unlabel is called to backtrack from the unsuccessful attempt to assign a value to variable x_i.

The procedure unlabel backtracks from the current variable x_i when all values have been tried for x_i without success. If $i = 1$, the algorithm has backtracked all the way to the start, and there is no solution; *next* is set to 0. Otherwise, the procedure selects a variable x_h to backtrack to, where $h < i$. If $h < i - 1$, the variables between x_h and x_i are reset.

The unlabel procedure is also responsible for keeping track of which values have already been tried without success. The value in V[h] is removed from CurrentDomain[h] (since it is supposedly responsible for the failure to assign a value to variable x_i) and if CurrentDomain[h] is still non-empty, consistent is set to **true** and *next* is set to h.

So unlabel terminates in one of three states:

1. $next = 0$: the problem has no solution and the program terminates.
2. consistent = **true** and $1 \leq next < i$: on the next iteration, label is called to try to assign a new value to variable x_h.
3. consistent = **false** and $1 \leq next < i$: on the next iteration, unlabel is called to backtrack again from variable x_h.

So for instance, to implement simple backtracking, the label procedure considers each value of the current variable in turn and checks it for consistency with the previous assignments. If any of these checks fails, the next value is tried; if they all fail, label terminates with consistent = **false**, and unlabel is called to backtrack from this variable to a previous variable. In simple backtracking, unlabel always backtracks to the immediately preceding variable, if there is one.

9.2 Conflict-directed BackJumping (CBJ)

This algorithm was introduced by Patrick Prosser in 1993 and incorporates the ideas already discussed. The ideas can be used simply as a way of introducing intelligent backtracking into simple backtracking, or in conjunction with forward checking to produce the algorithm FC-CBJ. (It is also possible to link it with MAC to produce MAC-CBJ.)

The first algorithm, CBJ, differs from the basic algorithm BT mainly in the backtracking step (of course), but is also slightly different in the labelling step. We record for each variable x_i which previous variables it failed consistency checks with, for some value in its domain. This list of previous variables is called the *conflict-set.* The conflict-sets (an array of lists) are passed as an extra parameter to cbj_label and cbj_unlabel.

cbj_unlabel backtracks to x_h, the most recent variable recorded in variable x_i's conflict-set, i.e. $h = max$ (ConflictSet[i]). It then combines the conflict-sets of variables x_h and x_i, so that if the algorithm backtracks again, it jumps to the most recent variable which has affected either x_h or x_i, i.e. ConflictSet[h] becomes ConflictSet[i] $\bigcup$ ConflictSet[h] - h.

cbj_unlabel also undoes all the assignments after x_h in the current partial solution and resets the domains of all these variables to their original values. As in simple backtracking, it removes the value currently assigned to x_h from its current domain.

9.3 FC-CBJ

The combination of forward checking with conflict-directed backjumping is slightly tricky, but the implementation will turn out to be straightforward. Suppose that the algorithm has to backtrack from variable x_i because none of the remaining values can be assigned without causing a failure. The reason for the failure is partly that previous assignments have removed values from x_i's current domain. But the failure is also due to the fact that every remaining value causes a domain wipeout in some future variable when we check forward. This is not directly the fault of the past assignments, but indirectly it is, because past assignments may have removed from the domains of these future variables some values which otherwise would have been consistent with the assignment to x_i which now causes a wipeout.

As with standard forward checking, whenever the algorithm tries to assign a value to a variable, it checks this assignment against the future variables, and removes values from the domains of future variables which are inconsistent with the current assignment. fc-cbj_label keeps a record of which past variables have reduced the domain of any variable in an array of lists Past_fc, such that Past_fc[j] is the list of past variables whose assignments have reduced the domain of variable x_j. If the attempt to assign a value to a variable x_i fails (because some future variable x_j has an empty domain after forward checking) we add the variables listed in Past_fc[j] to x_i's conflict set.

fc-cbj_unlabel starts by computing the conflict set for variable x_i (the variable it is backtracking from). The conflict-set is the union of the conflict-set already composed by fc-cbj_label, and Past_fc[i], i.e. it consists of any past variables which have reduced the domain of x_i, together with any variables which reduced the domains of future variables whose domains were then wiped out by the assignment of a value to x_i: these variables thereby indirectly prevented x_i from being assigned a value. As with cbj_unlabel, fc-cbj_unlabel backtracks to the variable x_h, where h is the largest value in x_i's conflict set.

The other main difference between fc-cbj_unlabel and other versions of unlabel is that in jumping back over the variables between x_h and x_i, it has to restore the current domains of these variables to their proper state. This is not their original domain (as in cbj_unlabel), but the state the domains were in after any values inconsistent with the assignments to variables x_1 to x_{h-1} had been removed.

9.4 Is CBJ of any practical use?

Intelligent backtracking (not just in constraint satisfaction algorithms) has been an active research topic in AI for many years. In constraint satisfaction, CBJ seems to be generally accepted as one of the best ways to do it. (There are some potential improvements in theory, to do with either remembering about failures in one part of the tree and applying this experience elsewhere in the tree, or not undoing all intervening assignments when jumping back, but these have either a very large space requirement or are very complex.) Yet intelligent backtracking is not used in constraint programming tools such as Solver. Why not?

Some possible reasons are:

- CBJ is a big improvement on simple backtracking. But then simple backtracking is such an inefficient algorithm that it is very easy to improve on it. Experiments have shown that the improvement from adding CBJ to FC or MAC is much less. To a large extent, looking forward as these algorithms do makes jumping back less necessary: they avoid making some kinds of mistake in the first place, and so don't need to jump back from them.

- Most experiments with constraint satisfaction algorithms have been carried out on randomly-generated problems. (A binary constraint can be represented as a Boolean matrix, and so can easily be generated by choosing the 0's and 1's randomly.) The constraints in most real problems, however, are not binary. Although the CBJ algorithm can be adapted to non-binary constraints, it is somewhat tricky. Furthermore, the nearest culprit for a failure is less likely to be a long way back in the tree if there are constraints of large arity in the problem.

- Good variable ordering heuristics tend to ensure that the next variable is chosen from those that were affected by the last assignment (although this is not always possible). This means that chronological backtracking is already doing what CBJ would do, for the most part. For instance, if all domain sizes are initially the same, the variable with smallest remaining domain will tend to be one whose domain was reduced by the last assignment. When there are non-binary constraints, a good heuristic is often to choose the next variable so as to reduce the arity of one or more of the constraints; this helps constraint propagation, because if all but two of the variables in a constraint have been assigned, the constraint is effectively a binary constraint and can be made arc consistent. This heuristic again tends to assign a variable which is constrained by the previous variable.

The current feeling amongst developers of constraint programming tools is that intelligent backtracking algorithms such as CBJ are probably not worthwhile. However, in particular types of problem it could improve performance significantly; it might be useful if the facility were available, to be turned on or off as appropriate.

10 Practical Problem Solving

In this section, I want to look at some of the things that need to be taken into account when solving a practical problem as a constraint satisfaction problem. In theory, if we can express a problem as a CSP, then the algorithms that we have will be able to solve it. In practice, however, *how* the problem is translated into a CSP can have a huge effect on the time taken to solve it, and thus on whether it can actually be solved in a reasonable time or not.

10.1 Formulating Problems

There may be an 'obvious' way to formulate a problem as a CSP, i.e. to represent the problem in terms of variables, values and constraints. But there is often a choice of more than one formulation, with a little more thought. To make the idea of different formulations of the same problem more formal, we need the idea of two CSPs being equivalent.

Rossi, Petrie and Dhar[11] defined the idea of equivalence for CSPs. Their definition is that two CSPs are equivalent if they are *mutually reducible*: a CSP P_1 is reducible to another, P_2, if it is possible to obtain the set of solutions of P_1 from the set of solutions of P_2 by mapping the variables and values of one problem to the variables and values of the other. A special case of this is two CSPs which have the same variables and values but different constraints: if they have the same set of solutions they are equivalent. (This covers the case of implied constraints, discussed below.) However, two CSPs can have different variables and values, but still be equivalent; for instance, we considered two different formulations of the DONALD + GERALD = ROBERT puzzle, one having the 'carry' variables, and the other not; these are still equivalent according to the above definition.

10.2 Reducing the Arity of the Constraints

Since arc consistency (as opposed to generalised arc consistency) is a property associated with binary constraints, arc consistency algorithms can only be used to reduce the domains of variables involved in either binary constraints or constraints in which all but two variables have been instantiated, so that the constraint has 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, constraints involving a large number of variables are undesirable, if they can be avoided.

We have already considered the cryptarithmetic puzzle DONALD + GERALD = ROBERT, which can be formulated using a constraint involving all ten variables. This constraint will not be of much use for constraint propagation. The problem can be re-formulated by adding the extra variables C1, C2, ..., 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 re-written as the following six constraints:

[11] On the Equivalence of Constraint Satisfaction Problems, In *Proceedings of the European AI Conference (ECAI'90)*, pp. 550-556, 1990.

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

involving at most five variables. As we have seen, this considerably speeds up finding a solution.

10.3 Reducing the Number of Possible Assignments

Another consideration in formulating problems is the initial size of the search space (i.e. before any pruning is done), which is given by the total number of possible assignments of the variables[12]. It is often a good heuristic to choose a model with a smaller initial search space, if there is a choice. This means that we should prefer models using a few variables with large domains to models using many variables with small domains, for the same problem. In particular, a formulation using Boolean variables, with domains {0,1}, is not usually desirable if an alternative can be found using variables with larger domains.

As an example, we have seen in section 2.2.6 two different formulations of the n-queens problem. One has n^2 variables v_i, where

$$\begin{aligned} v_i &= 1 \text{ if there is a queen on the } i^{th}\text{square} \\ &= 0 \text{ otherwise} \end{aligned}$$

The size of the search space in this formulation is 2^{n^2}. The formulation with one variable for each row, each with domain $\{1..n\}$, has a search space of size n^n. (In addition, as described earlier, the first formulation requires more constraints than the second.)

It should be remembered that this rule is only a heuristic, and what we are really trying to minimize is the size of the search tree that will be explored before a solution is found. The total number of possible assignments is an upper bound on this, but it will depend to a great extent on how much pruning results from the constraints.

10.4 Implied Constraints

Another technique is to look for constraints implied by those already in the problem, especially constraints with fewer variables than the existing constraints, or which constrain variables which did not previously have an explicit constraint between them. The constraints resulting from making a problem path consistent or k-consistent are an example of implied constraints, but we normally find implied constraints just by considering the problem, rather than applying any systematic algorithm (because of the reasons given earlier for not using path or k-consistency algorithms).

[12]The total number of possible assignments is the product of all the domain sizes.

10.5 Symmetries

In many problems, if there are any solutions at all, there are classes of equivalent solutions. To be specific, there is a subset of the problem variables such that if there is a solution to the problem, some permutation of the assignments to these variables yields another solution. 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. As a simple example suppose there are two problem variables x_1 and x_2 which represent entities which are, from the point of view of the CSP, identical: the variables will for instance have the same domain, and the variables will be involved in identical constraints with other variables. Then x_1 and x_2 are interchangeable and for any solution in which $x_1 = 1$ and $x_2 = 2$ (say) there is another solution with $x_1 = 2$ and $x_2 = 1$.

Symmetries of this kind 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. (In the example, if assigning $x_1 = 1$ and $x_2 = 2$ will not lead to a solution, neither will $x_1 = 2$ and $x_2 = 1$, so there is no point in exploring this possibility.)

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. So we could for instance insist that $x_1 < x_2$; this creates a distinction between two variables which were originally indistinguishable.

It is very common to artificially distinguish between identical variables in this way, by imposing an ordering on them.

By eliminating symmetries we are producing a non-equivalent CSP which has fewer solutions. But the solutions eliminated can be derived from those which are still allowed, and for all practical purposes we have only eliminated solutions which are equivalent to solutions which still exist. On the other hand, implied constraints do not change the set of possible solutions at all.

11 When Systematic Search is Not Good Enough

Constraint satisfaction problems are difficult to solve; there is no known method which has reasonable complexity, so that as we try to solve larger and larger problems, sooner or later we shall meet a problem which cannot be solved, or proved not to have a solution, in a reasonable time. For instance, I have tried to solve instances of the template design problem where there are 50 different designs and 40 slots in each template, and sometimes no solution can be found even if the program is left running overnight. It is often possible to improve the performance of the algorithms by thinking of better variable and value ordering heuristics, new implied constraints, etc. (for instance, a program which previously found no solution overnight may now find a solution in a few seconds) but suppose we have done everything we can think of. What do we do then?

It is often tempting to assume that the problem we are trying to solve has no solution, especially if we have an optimisation problem; this means that the last solution we have (if we have one) is the optimal solution. However, in reality we may be a long way from the optimal solution, unless we have some independent evidence that suggests that the solution we already have may be 'good enough'.

In other cases, we may simply be trying to find a solution that satisfies the constraints; then, if the algorithm fails to find a solution, and appears to be running for an indefinite amount of time, we have no answer to our problem at all. Sometimes in these circumstances, if we suspect that there really is no solution, we may be willing to relax some of the constraints so as to find a solution of some kind. Even if there is a solution, but the algorithm cannot find one, we may prefer to settle for a solution that satisfies most of the constraints rather than having no solution at all.

It is possible to express the problem of satisfying as many constraints as possible as a CSP; rather than posting every constraint, the constraints which we allow to be relaxed are simply defined. In Solver, each such constraint corresponds to a constrained Boolean expression. We can create another constrained (integer) expression representing the number of these constraints that are satisfied, and then maximise its value. Unfortunately, the resulting problem is likely to be even harder to solve than the original problem (although it will have a solution, whereas the original problem may have been infeasible). Unless a constraint definitely applies (i.e. has been posted) its effects cannot be propagated, so that the algorithm has to do correspondingly more search.

An alternative is to abandon the systematic search algorithms we have used so far, and use some kind of *local search* procedure, which always has a solution of sorts, and continually attempts to improve it.

An example is the *min-conflicts* heuristic (S. Minton, M.D. Johnston, A.B. Philips and P. Laird, 'Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method', Proceedings AAAI-90, pp. 17-254, 1990). This grew out of a neural network developed for scheduling the use of the Hubble Space Telescope. The neural network was extremely successful, and it was analysed to try to find out why; the min-conflicts heuristic is a simple algorithm which was developed from this analysis, and is reported to perform better than the neural network.

This heuristic can be built into a procedure which attempts to find a solution which maximises the number of satisfied constraints as follows:

- form an initial solution by assigning a value to each variable at random
- until a solution satisfying all the constraints is found, or a pre-set time limit is reached:
  - select a variable that is in conflict i.e. the value assigned to it violates one or more constraints involving one or more other variables
  - assign this variable a value that minimizes the number of conflicts (i.e. attempt to minimize the number of other variables that will need to be repaired)
  - break ties randomly

It is important to have a time-limit on an algorithm of this kind, because in abandoning systematic search, we have abandoned completeness; the algorithm has no means of telling if the problem has no solution. In common with other *local-search* algorithms (i.e. algorithms which repeatedly move from one solution to a 'neighbouring' solution), it may also get stuck in a *local optimum* where there is a neighbourhood of equally good solutions surrounded by worse solutions. The algorithm as defined above will then endlessly loop around this neighbourhood, but if this is not the best solution, the only way to improve is to temporarily choose a worse solution.

One way of avoiding this situation is to run the algorithm a number of times, starting with a different random initial solution each time.

Minton *et al.* claim that the min-conflicts heuristic works well in its original scheduling domain and on problems such as n-queens; they claim for instance that it can easily solve the one million queens problem (this is not all that impressive, apart from the memory problems associated with problems of this size, since the n-queens problem is not especially difficult, and gets relatively easier as n gets larger, because the constraints get looser).

One difficulty with methods of this kind is that they do not integrate easily with constraint propagation. Nevertheless, they may offer the only way of finding a solution of some kind to very difficult or very large problems, when the methods we have considered earlier fail. The latest version of Solver (5.0) does in fact offer local search as an option, as well as complete search.

12 Conclusions

Many different kinds of problem can be expressed as constraint satisfaction problems, and constraint programming tools offer a relatively easy way to express such problems. We have seen a number of algorithms which will guarantee to find a solution (or even an optimal solution) if given enough time. We have also seen that solving a large problem successfully may require careful development of a solution strategy based on insights into how the available methods will go about solving our particular problem; so constraint programming is no panacea for difficult problems. Nevertheless, for the right kind of problem and with a good solution strategy, it can be the best available way to solve the problem by far.

THE END