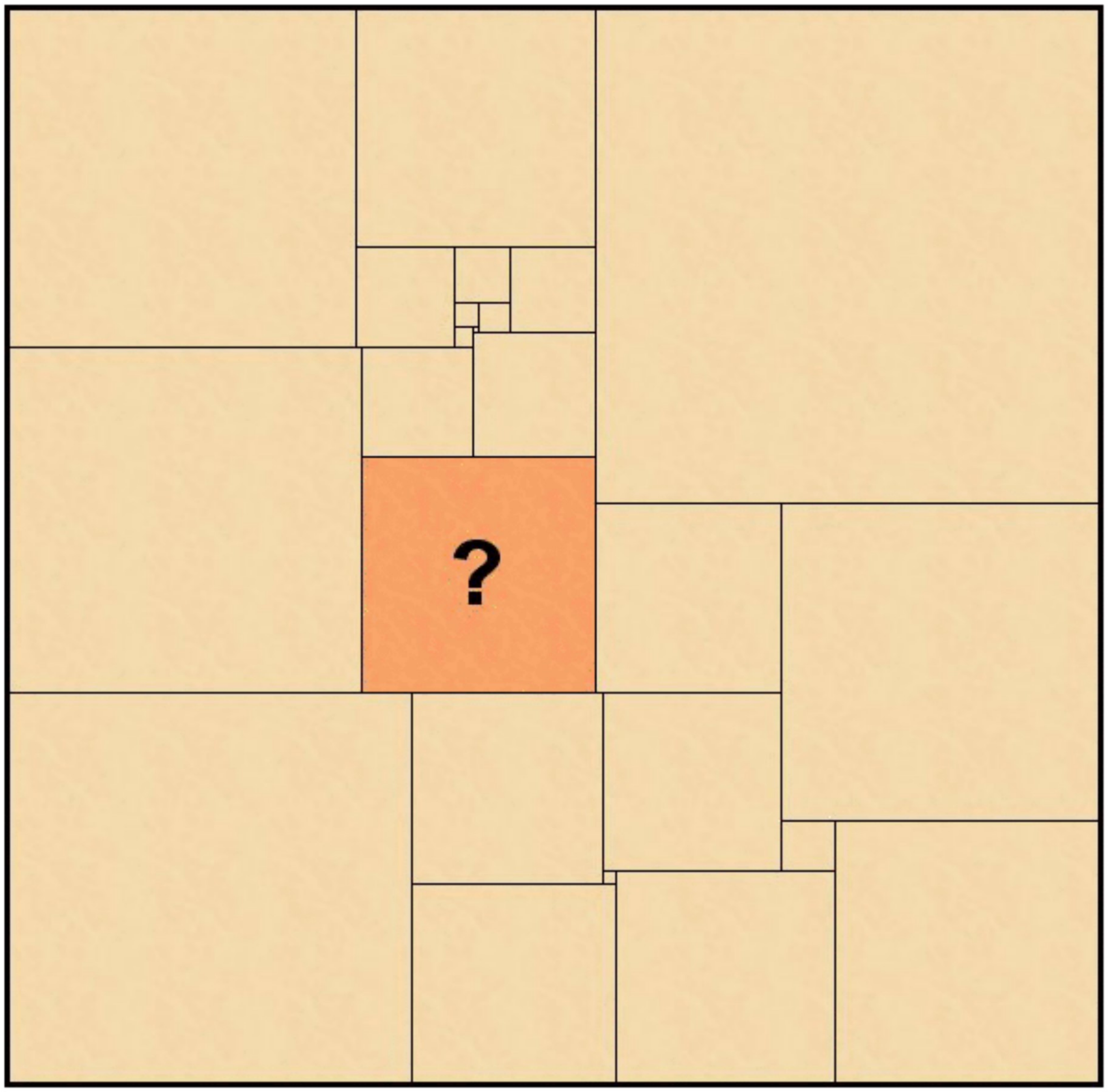
Lecture 15
Constraint Satisfaction Problems

I was perhaps most proud of AI on a Sunday. On this particular Sunday, a friend of mine found an article in the Observer about the High-IQ society, a rather brash and even more elitist version of Mensa. Their founder said that their entrance test was so difficult that some of the problems had never been solved. The problem given below was in the Observer as such an unsolved problem. After looking at it for a few minutes, I confidently told my friend that I would have the answer in half an hour.

The square below contains 24 smaller squares, each with a different integral size. Determine the length of the shaded square



After just over 45 minutes, I did indeed have an answer, and my friend was suitably impressed. See the end of these notes for the details. Of course, I didn't spend my time trying to figure it out (if you want to split the atom, you don't sharpen a knife). Instead, I used the time to describe the problem to a **constraint solver**, which is infinitely better at these things than me. The constraint solver is part of good old Sicstus Prolog, so specifying the problem was a matter of writing it as a logic program - it's worth pointing out that I didn't specify how to find the solution, just what the problem was. With AI programming languages such as Prolog, every now and then the intelligence behind the scenes comes in very handy. Once I had specified the problem to the solver (a mere 80 lines of Prolog), it took only one hundredth of a second to solve the problem. So not only can the computer solve a problem which had beaten many high IQ people, it could solve 100 of these "difficult" problems every second. A great success for AI. In this lecture, we will look at how constraint solving works in general. Much of the material here is taken from Barbara Smith's excellent tutorial on Constraint Solving which is available here:

A Tutorial on Constraint Programming

15.1 Specifying Constraint Problems

As with most successful AI techniques, constraint solving is all about solving problems: somehow phrase the intelligent task you are interested in as a problem, then massage it into the format of a **constraint satisfaction problem** (CSP), put it into a constraint solver and see if you get a solution. CSPs consist of the following parts:

- A set of variables $X = \{x_1, x_2, ..., x_n\}$
- A finite set of values that each variable can take. This is called the **domain** of the variable. The domain of variable x_i is written D_i
- A set of constraints that specifies which values the variables can take simultaneously

In the high-IQ problem above, there are 25 variables: one for each of the 24 smaller square lengths, and one for the length of the big square. If we say that the smallest square is of length 1, then the big square is perhaps of length at most 1000. Hence the variables can each take values in the range 1 to 1000. There are many constraints in this problem, including the fact that each length is different, and that certain ones add up to give other lengths, for example the lengths of the three squares along the top must add up to the length of the big square.

Depending on what solver you are using, constraints are often expressed as relationships between variables, e.g., $x_1 + x_2 < x_3$. However, to be able to discuss constraints more formally, we use the following notation:

A constraint C_{ijk} specifies which tuples of values variables x_i, x_j and x_k ARE allowed to take simultaneously. In plain English, a constraint normally talks about things which can't happen, but in our formalism, we are looking at tuples (v_i, v_j, v_k) which x_i, x_j and x_k **can** take simultaneously. As a simple example, suppose we have a CSP with two variables x and y, and that x can take values {1,2,3}, whereas y can take values {2,3}. Then the constraint that x=y would be written as:

$$C_{xy}=\{(2,2), (3,3)\},$$

and the constraint that x<y would be written as

$$C_{xy} = \{(1,2),(1,3),(2,3)\}$$

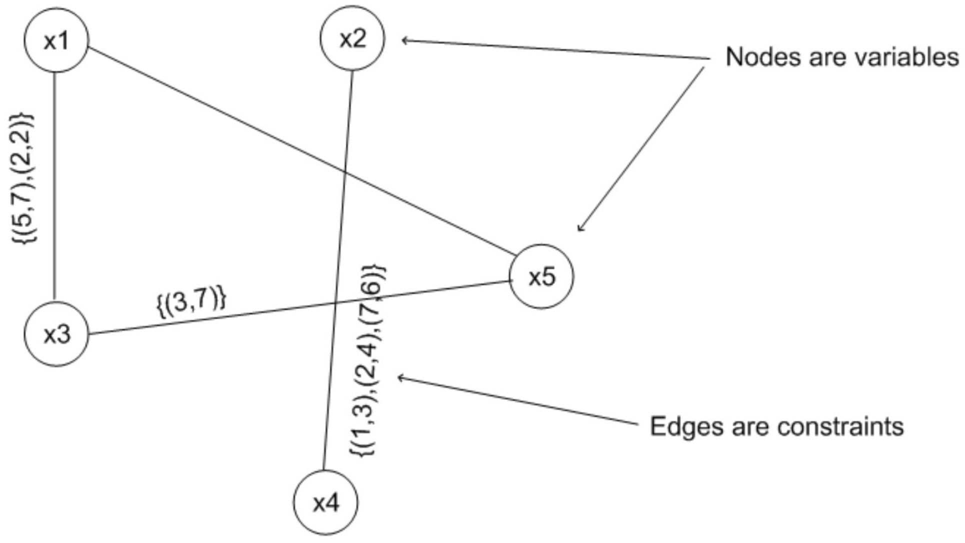
A **solution** to a CSP is an assignment of values, one to each variable in such a way that no constraint is broken.

It depends on the problem at hand, but the user might want to know that there is a solution, i.e., they will take the first answer given. Alternatively, they may require all the solutions to the problem, or they might want to know that no solutions exists. Sometimes, the point of the exercise is to find the optimum solution based on some measure of worth. Sometimes, it's possible to do this without enumerating all the solutions, but other times, it will be necessary to find all solutions, then work out which is the optimum. In the high-IQ problem, a solution is simply a set of lengths, one per square. The shaded one is the 17th biggest, which answers the IQ question.

15.2 Binary Constraints

Unary constraints specify that a particular variable can take certain values, which basically restricts the domain for that variable, and hence should be taken care of when specifying the CSP. **Binary** constraints relate two variables, and **binary constraint problems** are special CSPs which involve only binary constraints. Binary CSPs have a special place in the theory because all CSPs can be written as binary CSPs (we don't go into the details of this here, and while it is possible in theory to do so, in practice, the translation is rarely used). Also, binary CSPs can be represented both graphically and using matrices, which can make them easier to understand.

Binary constraint graphs such as the one below afford a nice representation of constraint problems, where the nodes are the variables and the edges represent the constraints on the variables between the two variables joined by the edge (remember that the constraints state which values can be taken at the same time).



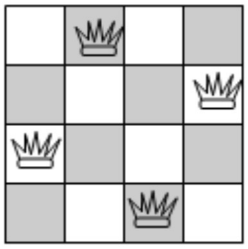
Binary constraints can also be represented as matrices, with a single matrix for each constraint. For example, in the above constraint graph, the constraint between variables x_4 and x_5 is {(1,3),(2,4),(7,6)}. This can be represented as the following matrix.

C	1	2	3	4	5	6	7
1			*				
2				*			

3							
4							
5							
6							
7						*	

We see that the asterixes mark the entry (i,j) in the table such that variable x_4 can take value i at the same time that variable x_5 takes value j. As all CSPs can be written as binary CSPs, the artificial generation of random binary CSPs as a set of matrices is often used to assess the relative abilities of constraint solvers. However, it should be noted that in real world constraint problems, there is often much more structure to the problems than you get from such random constructions.

A very commonly used example CSP, which we will use in the next section, is the "n-queens" problem, which is the problem of placing n queens on a chess board in such a way that no one threatens another along the vertical, horizontal or diagonal. We've seen this in previous lectures. There are many possibilities for representing this as a CSP (in fact, finding the best specification of a problem so that a solver gets the answer as quickly as possible is a highly skilled art). One possibility is to have the variables representing the rows and the values they can take representing the columns on the row that a queen was situated on. If we look at the following solution to the 4-queens problem below:



Then, counting rows from the top downwards and columns from the left, the solution would be represented as: X_1=2, X_2=4, X_3=1, X_4=3. This is because the queen on row 1 is in column 2, the queen in row 2 is in column 4, the queen in row 3 is in column 1 and the queen in row 4 is in column 3. The constraint between variable X_1 and X_2 would be:

$$C_{1,2} = \{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)\}$$

As an exercise, work out exactly what the above constraint is saying.

15.3 Arc Consistency

There have been many advances in how constraint solvers search for solutions (remember this means an assignment of a value to each variable in such a way that no constraint is broken). We look first at a pre-processing step which can greatly improve efficiency by pruning the search space, namely arc-consistency.

Following this, we'll look at two search methods, backtracking and forward checking which keep assigning values to variables until a solution is found. Finally, we'll look at some heuristics for improving the efficiency of the solver, namely how to order the choosing of the variables, and how to order the assigning of the values to variables.

The pre-processing routine for bianry constraints known as **arc-consistency** involves calling a pair (x_i, x_j) an **arc** and noting that this is an ordered pair, i.e., it is not the same as (x_j, x_i). Each arc is associated with a single constraint C_{ij}, which constrains variables x_i and x_j. We say that the arc (x_i, x_j) is **consistent** if, for all values a in D_i, there is a value b in D_j such that the assignment x_i=a and x_j=b satisfies constraint C_{ij}. Note that (x_i, x_j) being consistent doesn't necessarily mean that (x_j,x_i) is also consistent. To use this in a pre-processing way, we take every pair of variables and make it arc-consistent. That is, we take each pair (x_i,x_j) and remove variables from D_i which make it inconsistent, until it becomes consistent. This effectively removes values from the domain of variables, hence prunes the search space and makes it likely that the solver will succeed (or fail to find a solution) more quickly.

To demonstrate the worth of performing an arc-consistency check before starting a serarch for a solution, we'll use an example from Barbara Smith's tutorial. Suppose that we have four tasks to complete, A, B, C and D, and we're trying to schedule them. They are subject to the constraints that:

- Task A lasts 3 hours and precedes tasks B and C
- Task B lasts 2 hours and precedes task D
- Task C lasts 4 hours and precedes task D
- Task D lasts 2 hours

We will model this problem with a variable for each of the task start times, namely *startA, startB, startC and startD*. We'll also have a variable for the overall start time: *start*, and a variable for the overall finishing time: *finish*. We will say that the domain for variable *start* is {0}, but the domains for all the other variables is {0,1,...,11}, because the summation of the duration of the tasks is 3 + 2 + 4 + 2 = 11. We can now translate our English specification of the constraints into our formal model. We start with an intermediate translation thus:

- $start \leq startA$
- $startA + 3 \leq startB$
- $startA + 3 \leq startC$
- $startB + 2 \leq startD$
- $startC + 2 \leq startD$
- $startD + 2 \leq finish$

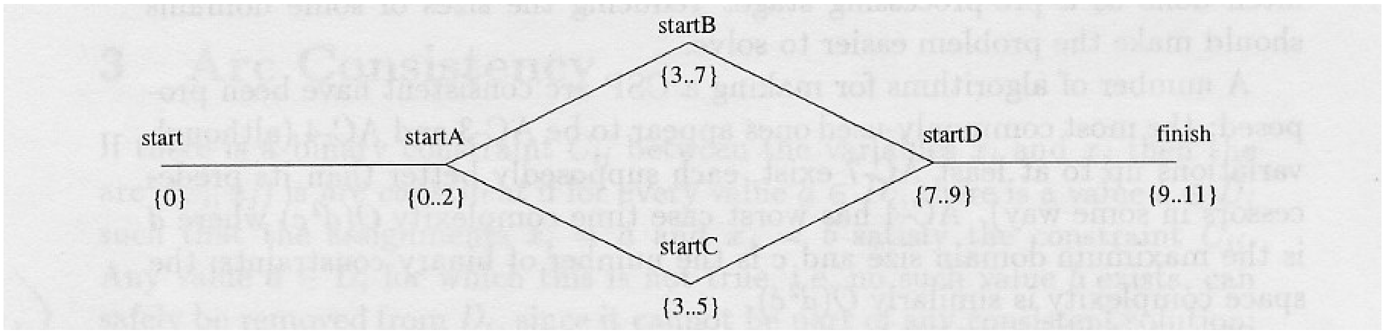
Then, by thinking about the values that each pair of variables can take simultatneously, we can write the constraints as follows:

- $C_{start,startA}$ = {(0,0), (0,1), (0,2), ..., (0,11)}
- $C_{startA,start}$ = {(0,0), (1,0), (2,0), ..., (11,0)}
- $C_{startA,startB}$ = {(0,3), (0,4), ..., (0,11), (1,4), (1,5), ..., (8,11)}
- etc.

Now, we will check whether each arc is arc-consistent, and if not, we will remove values from the domains of variables until we get consistency. We look first at the arc *(start, startA)* which is associated with the constraint {(0,0), (0,1), (0,2), ..., (0,11)} above. We need to check whether there is any value, P, in D_{start} that does not have a corresponding value, Q, such that (P,Q) satisfies the constraint, i.e., appears in the set of assignable pairs. As D_{start} is just {0}, we are fine. We next look at the arc *(startA, start)*, and check whether there is any value in D_{startA}, P, which doesn't have a corresponding Q such that (P,Q) is in $C_{startA, start}$. Again, we are OK, because all the values in D_{startA} appear in $C_{startA, start}$.

If we now look at the arc (startA, startB), then the constraint in question is: {(0,3), (0,4), ..., (0,11), (1,4), (1,5), ..., (8,11)}. We see that their is no pair of the form (9,Q) in the constraint, similarly no pair of the form (10,Q) or (11,Q). Hence, this arc is not arc-consistent, and we have to remove the values 9, 10 and 11 from the domain of *startA* in order to make the arc consistent. This makes sense, because we know that, if task B is going to start after task A, which has duration 3 hours, and they are all going to have started by the eleventh hour, then task A cannot start after the eighth hour. Hence, we can - and do - remove the values 9, 10 and 11 from the domain of *startA*.

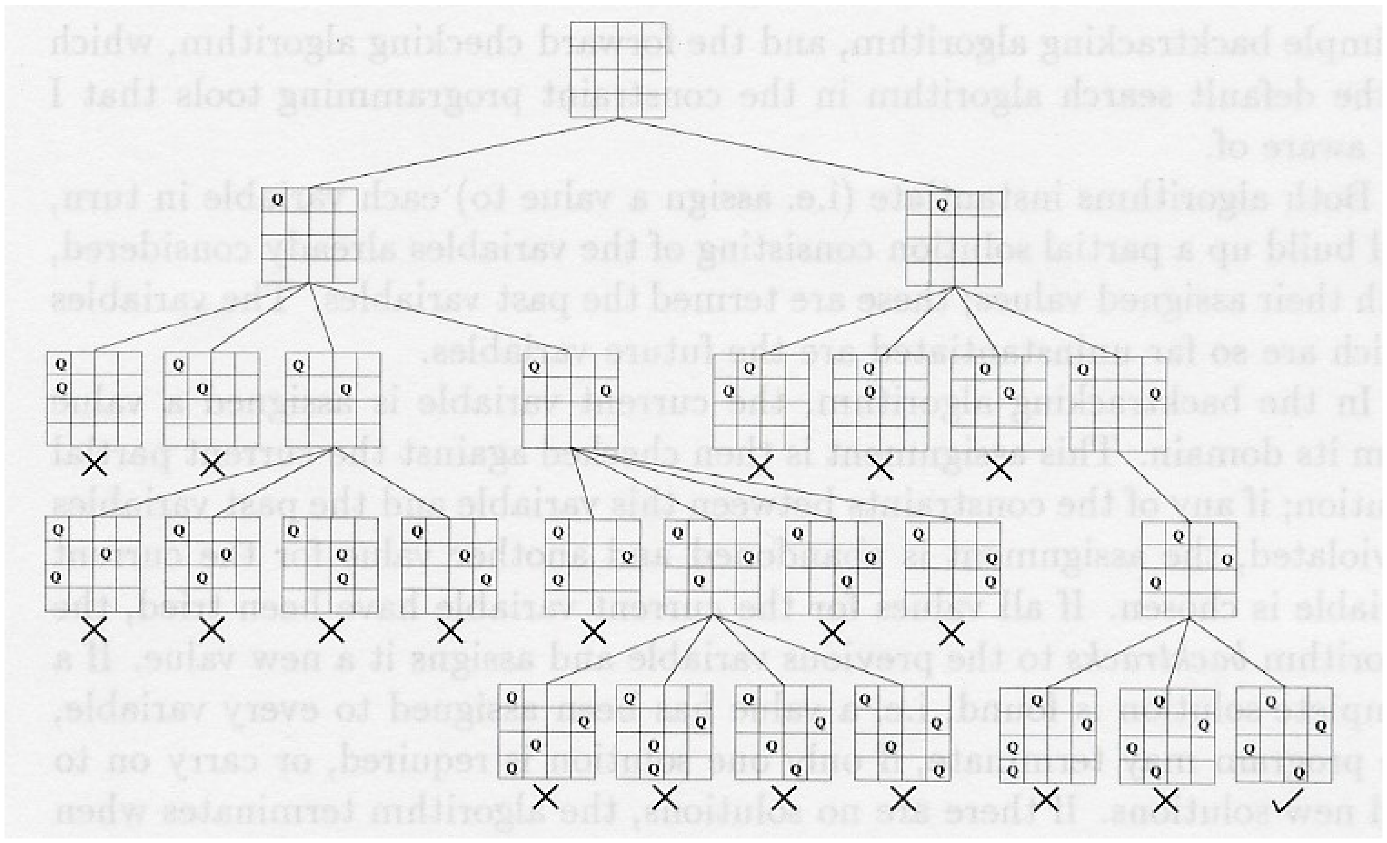
This method of removing values from domains is highly effective. As reported in Barbara Smith's tutorial, the domains become quite small, as reflected in the following scheduling network:



We see that the largest domain size has only 5 values in it, which means that quite a lot of the search space has been pruned. In practice, to remove as many variables as possible in a CSP which is dependent on precedence constraints, we have to work backwards, i.e., look at the start time of the task, T, which must occur last, then make each arc of the form (*startT, Y*) consistent for every variable *Y*. Following this, move on to the task which must occur second to last, etc. In CSPs which *only* involve precedence constraints, arc-consistency is guaranteed to remove all values which cannot appear in a solution to the CSP. In general, however, we cannot make such a guarantee, but arc-consistency usually has some effect on the initial specification of a problem.

15.4 Search Methods and Heuristics

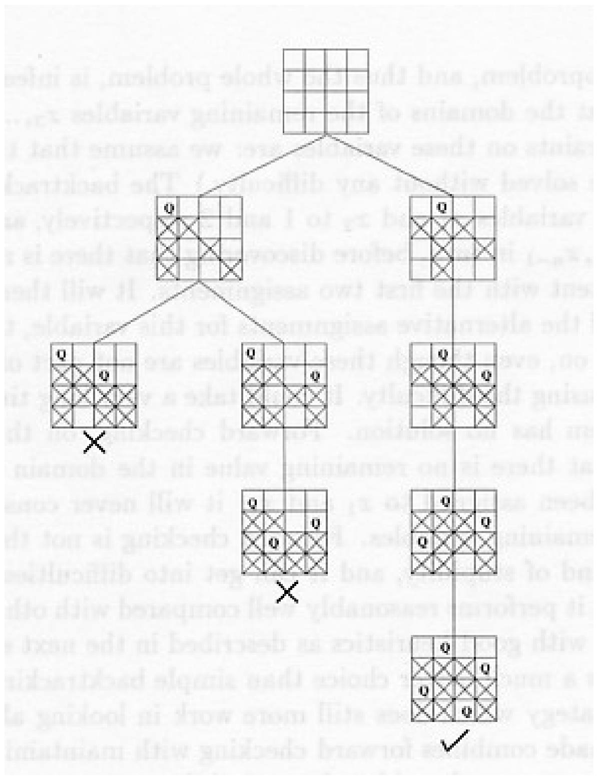
We now come to the question of how constraint solvers search for solutions - constraint preserving assignments of values to variables - to the CSPs they are given. The most obvious approach is to use a depth first search: assign a value to the first variable and check that this assignment doesn't break any constraints. Then, move on to the next variable, assign it a value and check that this doesn't break any constraints, then move on to the next variable and so on. When an assignment does break a constraint, then choose a different value for the assignment until one is found which satisfies the constraints. If one cannot be found, then this is when the search must **backtrack**. In such a situation, the previous variable is looked at again, and the next value for it is tried. In this way, all possible sets of assignments will be tried, and a solution will be found. The following search diagram - taken from Smith's tutorial paper - shows how the search for a solution to the 4-queens problem progresses until it finds a solution:



We see that the first time it backtracks is after the failure to put a queen in row three given queens in positions (1,1) and (2,3). In this case, it backtracked and move the queen in (2,3) to (2,4). Eventually, this didn't work out either, so it had to backtrack further and moved the queen in (1,1) to (1,2). This led fairly quickly to a solution.

To add some sophistication to the search method, constraint solvers use a technique known as **forward checking**. The general idea is to work the same as a backtracking search, but, when checking compliance with constraints after assigning a value to a variable, the agent also checks whether this assignment is going to break constraints with *future* variable assignments. That is, supposing that V_c has been assigned to the current variable c, then for each unassigned variable x_i, (temporarily) remove all values from D_i which, along with V_c break a constraint. It may be that in doing so, D_i becomes empty. This means that the choice of V_c for the current variable is bad - it will not find its way into a solution to the problem, because there's no way to assign a value to x_i without breaking a constraint. In such a scenario, even though the assignment of V_c may not break any constraints with already assigned variables, a new value is chosen (or backtracking occurs if there are no values left), because we know that V_c is a bad assignment.

The following diagram (again, taken from Smith's tutorial) shows how forward checking improves the search for a solution to the 4-queens problem.



In addition to forward checking to improve the intelligence of the constraint solving agent, there are some possibilities for a heuristic search. Firstly, our agent can worry about the order in which it looks at the variables, e.g., in the 4-queens problem, it might try to put a queen in row 2, then one in row 3, one in row 1 and finally one in row 4. A solver taking such care is said to be using a **variable-ordering** heuristic. The ordering of variables can be done before a search is started and rigidly adhered to during the search. This might be a good idea if there is extra knowledge about the problem, e.g., that a particular variable should be assigned a value sooner rather than later. Alternatively, the ordering of the variables can be done dynamically, in response to some information gathered about how the search is progressing during the search procedure.

One such dynamic ordering procedure is called "fail-first forward checking". The idea is to take advantage of information gathered while forward checking during search. In cases where forward checking highlights the fact that a future domain is effectively emptied, then this signals that it's time to change the current assignment. However, in the general case, the domain of the variable will be reduced but not necessarily emptied. Suppose that of all the future variables, x_f has the most values removed from D_f. The fail-first approach specifies that our agent should choose to assign values to x_f next. The thinking behind this is that, with fewer possible assignments for x_f than the other future variables, we will find out most quickly whether we are heading down a dead-end. Hence, a better name for this approach would be "find out if its a dead end quickest". However, this isn't as catchy a phrase as "fail-first".

An alternative/addition to variable ordering is **value ordering**. Again, we could specify in advance the order in which values should be assigned to variables, and this kind of tweaking of the problem specification can dramatically improve search time. We can also perform value ordering dynamically: suppose that it's possible to assign values V_c, V_d and V_e to the current variable. Further suppose that, when looking at *all* the future variables, the total number of values in their domains reduces to 300, 20 and 50 for V_c, V_d and V_e respectively. We could then specify that our agent assigns V_c at this stage in the search, because it has retained the most number of values in the future domains. This is different from variable ordering in two important ways:

- If this is a dead end then we will end up visiting all the values for this variable anyway, so fail-first does not make sense for values. Rather, we try and keep our options open as much as possible, as this will help if there is a solution ahead of us.

- Unlike the variable ordering heuristics, this heuristic carries an extra cost on top of forward checking, because the reduction in domain sizes of future variables for every assignment of the current variable needs to be checked. Hence, it is possible that this kind of value ordering will slow things down. In practice, this is what happens for randomly constructed binary CSPs. On occasions, however, it can sometimes be a very good idea to employ dynamic value ordering.

15.5 Applications and Hot Topics of Constraint Solving

Constraint solving is one of the biggest success stories in Artificial Intelligence. There have been many mathematical applications of CSP techniques, for example to solving algebraic existence problems and to solving numerical problems such as finding minimal Golomb rulers: take a ruler and put marks on it at integer places, so that no two pairs of marks have the same distance between them. The question is: given a particular number of marks, what is the smallest Golomb ruler which accomodates them all.

In the past, it has been true that constraint solving was classed as a kind of "mediocre" approach: it is possible to get fairly good results for a whole host of problems using constraint solving, but using methods from operational research, or bespoke methods hand carved into a program for the problem always out-performed them. With modern advances in constraint searching, this imbalance has been addressed somewhat. However, the major advantage to constraint solving is the ease of using solvers to get answers to questions. In the high-IQ problem, it didn't take me long at all to specify the problem and get an answer. Hence, for non-expert users, the constraint solving approach is often the first choice. For this reason, there are many, many industrial applications of constraint solving. These include sports scheduling, i.e., deciding when various teams will play each other in a league, which sounds easy but is very difficult (and there's a lot of money in it!). They also include bin-packing problems, for example how to fit a certain number of crates into a ship. Constraint solving is increasingly playing a part in other sciences, most notably bioinformatics.

There is still a great deal of research into improving constraint solving approaches. One bottleneck is the formulation of CSPs in the first place. As mentioned before, this is a highly skilled job - the correct choice of variables/values/constraints to represent the problem and the ordering of the variables and values can make an insolvable problem solvable. Indeed, experts are hired to correctly specify constraint problems in industrial settings. Given this need to correctly specify a CSP, there have been some recent attempts to get a software agent to *automatically* reformulate CSPs. One good way to do this is to add some **implied constraints** to the problem specification. These are additional constraints which can be proved to follow from the original specification, hence can be added without loss of generality (no solutions will be lost by adding these constraints). Alternatively, in cases where you are only interested in finding a single solution, CSPs can be automatically specialised in the hope that a solution to the specialised solution is easier to find.

Another hot topic is the detection of symmetry, and how to break this. For instance, if one can show that two variables always take the same values (a **symmetry**), then one of these can be removed from the problem specification (**breaking** that symmetry). There are more subtle symmetries than this, and humans are good at finding and breaking them. Research is currently underway into how to get a search technique to automatically find and break symmetry before and during a CSP search. Another hot-topic is how to handle **dynamic** CSPs, which are such that the problem specification changes as you are trying to solve it. For instance, while trying to find the best way to load a ship with cargo, it may be that another crate turns up and has to be fitted into the cargo hold. Dynamic constraint solvers have to accomodate the alteration of the specification without necessarily losing all the work it has done to solve the original problem.

15.6 The High IQ Problem

The answer to the problem is 38, because Prolog says that L17=38 and the shaded square is the 17th biggest.

Here is the specification of the problem in all its glory:

```
?- use_module(library(clpfd)).

/* type go(200, Answer) to run the program. 200 is the highest side */
/* length it will look for. */

go(Top, [L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,
         L15,L16,L17,L18,L19,L20,L21,L22,L23,L24,L25]) :-

    /* Start the timer */
    statistics(runtime,_),

    /* Domains */
    L1 in 1..Top, L2 in 1..Top, L3 in 1..Top, L4 in 1..Top,
    L5 in 1..Top, L6 in 1..Top, L7 in 1..Top, L8 in 1..Top,
    L9 in 1..Top, L10 in 1..Top, L11 in 1..Top, L12 in 1..Top,
    L13 in 1..Top, L14 in 1..Top, L15 in 1..Top, L16 in 1..Top,
    L17 in 1..Top, L18 in 1..Top, L19 in 1..Top, L20 in 1..Top,
    L21 in 1..Top, L22 in 1..Top, L23 in 1..Top, L24 in 1..Top,
    L25 in 1..Top,

    /* Ordering */
    L1 #< L2, L2 #< L3, L3 #< L4, L4 #< L5, L5 #< L6, L6 #< L7,
    L7 #< L8, L8 #< L9, L9 #< L10, L10 #< L11, L11 #< L12, L12 #< L13,
    L13 #< L14, L14 #< L15, L15 #< L16, L16 #< L17, L17 #< L18,
    L18 #< L19, L19 #< L20, L20 #< L21, L21 #< L22, L22 #< L23,
    L23 #< L24, L24 #< L25,

    all_different([L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,
                   L15,L16,L17,L18,L19,L20,L21,L22,L23,L24,L25]),

    /* Sum of Squares Constraint */
    L1*L1 + L2*L2 + L3*L3 + L4*L4 + L5*L5 + L6*L6 + L7*L7 + L8*L8 +
    L9*L9 + L10*L10 + L11*L11 + L12*L12 + L13*L13 + L14*L14 + L15*L15 +
    L16*L16 + L17*L17 + L18*L18 + L19*L19 + L20*L20 + L21*L21 + L22*L22 +
    L23*L23 + L24*L24 #= L25*L25,

    /* Length Constraints */
    L1 + L3 #= L4, L4 + L1 #= L5,
    L4 + L5 #= L7, L5 + L7 #= L8,
    L3 + L4 + L7 #= L9, L1 + L5 + L8 #= L11,
    L2 + L12 #= L14, L2 + L14 #= L15,
    L2 + L15 #= L16, L10 + L11 #= L17,
    L7 + L8 + L9 #= L18, L6 + L16 #= L19,
    L6 + L19 #= L20, L9 + L18 #= L21,
    L10 + L17 #= L22, L14 + L15 #= L23,
    L13 + L20 #= L24, L21 + L22 + L23 #= L25,
    L18 + L21 + L24 #= L25, L19 + L20 + L24 #= L25,
    L15 + L16 + L19 + L23 #= L25,

    /* Find the Answer */
    labeling([], [L1,L2,L3,L4,L5,L6,L7,L8,L9,L10,L11,L12,L13,L14,
                  L15,L16,L17,L18,L19,L20,L21,L22,L23,L24,L25]),

    /* Write the Answer */
    write('L1 = '),write(L1),nl,write('L2 = '),write(L2),nl,
    write('L3 = '),write(L3),nl,write('L4 = '),write(L4),nl,
    write('L5 = '),write(L5),nl,write('L6 = '),write(L6),nl,
    write('L7 = '),write(L7),nl,write('L8 = '),write(L8),nl,
    write('L9 = '),write(L9),nl,write('L10 = '),write(L10),nl,
    write('L11 = '),write(L11),nl,write('L12 = '),write(L12),nl,
    write('L13 = '),write(L13),nl,write('L14 = '),write(L14),nl,
    write('L15 = '),write(L15),nl,write('L16 = '),write(L16),nl,
    write('L17 = '),write(L17),nl,write('L18 = '),write(L18),nl,
    write('L19 = '),write(L19),nl,write('L20 = '),write(L20),nl,
    write('L21 = '),write(L21),nl,write('L22 = '),write(L22),nl,
    write('L23 = '),write(L23),nl,write('L24 = '),write(L24),nl,
    write('L25 = '),write(L25),nl,nl,

    /* Double check the Answer */
    LHS is L1*L1 + L2*L2 + L3*L3 + L4*L4 + L5*L5 + L6*L6 + L7*L7 +
    L8*L8 + L9*L9 + L10*L10 + L11*L11 + L12*L12 + L13*L13 + L14*L14 +
    L15*L15 + L16*L16 + L17*L17 + L18*L18 + L19*L19 + L20*L20 + L21*L21 +
    L22*L22 + L23*L23 + L24*L24,

    RHS is L25*L25,
    write(LHS),nl,
    write(RHS),nl,nl,

    /* Stop the timer */
    statistics(runtime,[_,TimeTaken]),
    write(TimeTaken),nl,nl.
```

Here is the output from the program:

```
go(10000, A).
L1 = 1
L2 = 2
L3 = 3
L4 = 4
L5 = 5
L6 = 8
L7 = 9
L8 = 14
L9 = 16
L10 = 18
L11 = 20
L12 = 29
L13 = 30
L14 = 31
L15 = 33
L16 = 35
L17 = 38
L18 = 39
L19 = 43
L20 = 51
L21 = 55
L22 = 56
L23 = 64
L24 = 81
L25 = 175

30625
30625

30
```

© Simon Colton & Jeremy Gow 2009