8 Modelling with Finite Domain Constraints

Many difficult and important problems which arise in management decision support, for instance scheduling and resource allocation, may be expressed as satisfaction or optimization problems involving constraints over variables with finite domains. CLP languages providing constraints over finite domains have proved ideal for tackling such problems and so these CLP languages have had the most industrial impact and are now used to solve many real-world industrial problems.

In this chapter, we investigate how to model and solve such problems efficiently using a CLP language whose constraints range over finite domains. We start with simple examples to illustrate the operation of the solver and its combination with backtracking. We then discuss labelling strategies. These ensure that, by the end of every derivation, each variable is constrained to take a single value. For such derivations the finite domain solver is complete. As a larger more realistic example, we solve a simple scheduling problem with resources.

A recurring theme of the chapter is the need to consider efficiency and, in particular, to control the size of the search space. We shall discuss different labelling strategies, complex constraints, show how to add redundant information to improve the search behaviour and demonstrate how different models of the same problem may have different efficiency.

Throughout this chapter we shall use the language *CLP*(*FD*) introduced in Chapter 4. It combines finite domain constraints with tree constraints. Elements of the constraint domain are trees which may have integer constants as leaves. Finite domain variables range over either a finite domain of integers or a finite domain of arbitrary constants. In effect, however, all variables ranging over finite domains are treated as if they range over integers since the constants in each domain are mapped to integers. The usual mathematical constraints are provided for integer finite domain variables, that is to say, equality ($=$), inequality ($\leq$, $\geq$), strict inequality ($<$, $>$) and disequality ($\neq$). We will also discuss the use of complex primitive constraints such as `element` and `cumulative`. The constraints over trees are the same as those provided in $CLP(\mathcal{R})$, namely equality ($=$) and disequality ($\neq$). This means complex data structures can be created and manipulated in the same way as in $CLP(\mathcal{R})$.

The *CLP*(*FD*) system uses unification to solve tree constraints and an incomplete bounds consistency based solver for finite domain constraints. We shall see that an awareness of the incompleteness of the solver and the way in which it handles different constraints is important when considering the efficiency of a program.

8.1 Domains and Labelling

In Chapter 5 we saw how we can use multiple facts to model a relation. As an example the `likes` relation shown in Figure 3.3 consists of the ordered pairs:

$$\{\ (kim, maria),\ (kim, nicole),\ (kim, erika),\ (peter, maria),\ (bernd, nicole),\ (bernd, maria),\ (bernd, erika)\ \}.$$

It can be modelled by the simple program:

```
likes(kim, maria).
likes(kim, nicole).
likes(kim, erika).
likes(peter, maria).
likes(bernd, nicole).
likes(bernd, maria).
likes(bernd, erika).
```

Given this definition of the *likes* relation we can now solve the old-fashioned marriage problem of Example 3.3 using finite domain constraints. We can model the marriage problem by the goal:

```
    [N,M,E] :: [kim, peter,bernd], N ≠ M, N ≠ E, M ≠ E,
    likes(N, nicole), likes(M, maria), likes(E, erika).
```

This will give the desired answers:

$$N = kim \land M = peter \land E = bernd \text{ and } N = bernd \land M = peter \land E = kim.$$

The initial literal in the goal, `[N,M,E] :: [kim, peter,bernd]`, is used to tell the *CLP(FD)* system that variables N, M and E each have the domain $\{kim, peter, bernd\}$. The syntax for specifying the domains of variables is quite simple. For instance, to tell the system that variable X has the initial domain $\{1, 2, 3, 4\}$ a literal of the form `X :: [1,2,3,4]` or `X :: [1..4]` is used. Variables are assigned non-integer domains using the form `X :: [red, white, blue]`. Multiple variables can be initialised to the same domain using the syntax illustrated in the above example. If no initial domain is given for a finite domain variable it is usually assumed to be an integer with some large default range, for example $[-10000000..10000000]$.

As another example, we can model the smuggler's knapsack problem of Example 3.5 using the goal:

```
[W,P,C] :: [0..9], 4*W + 3*P + 2*C ≤ 9, 15*W + 10*P + 7*C ≥ 30.
```

Now consider what happens when this goal is executed. The *CLP(FD)* system will simply return *unknown*. This is because the consistency based finite domain solver is incomplete and, in this case, not powerful enough to determine that the constraint is satisfiable.

The reason CLP systems usually provide an incomplete solver for finite domain constraints is that complete solvers are expensive and so should not be employed every time a constraint is added to the constraint store. However, as in the above example, eventually the programmer will wish to know if the finite domain constraint is satisfiable and, if so, what the answer is. Fortunately most CLP systems also provide a complete backtracking constraint solver for finite domains. Because of the expense of the solver, it is not automatically used by the CLP system. Instead it must be explicitly called by the programmer, typically, after all of the constraints have been collected.

The complete constraint solver for finite domain constraints system is usually provided by the built-in predicate `labeling`.[1] The predicate `labeling(Vs)` takes a list of finite domain variables, Vs, and uses a backtracking search to find a solution for those variables. In Chapter 3 we gave a complete propagation based backtracking solver for finite domains. One of the strengths of the CLP approach is that such solvers can be easily programmed in CLP itself. We shall investigate how to program `labeling` in the next section. For the moment, however, we regard `labeling` as a black-box.

Execution of the goal

```
[W,P,C] :: [0..9],
4*W + 3*P + 2*C ≤ 9, 15*W + 10*P + 7*C ≥ 30,
labeling([W,P,C]).
```

finds the solutions

$$W = 0 \land P = 1 \land C = 3, \quad W = 0 \land P = 3 \land C = 0,$$
$$W = 1 \land P = 1 \land C = 1, \quad W = 2 \land P = 0 \land C = 0.$$

This goal illustrates the typical form of a constraint program over finite domains. First, the variables and their initial domains are defined. Then the constraints modelling the problem are given. Finally, a labelling predicate is used to invoke a complete solver. This is the so-called *constrain and generate* methodology: first the constraints are applied, then a solution is generated by labelling.

We can also use optimization constructs when modelling with $CLP(FD)$. For instance, the smuggler can determine his maximum profit by executing the goal

```
L = -15*W - 10*P - 7*C,
minimize(([W,P,C] :: [0..9],
        4*W + 3*P + 2*C ≤ 9, 15*W + 10*P + 7*C ≥ 30,
        labeling([W,P,C])),
        L).
```

This will give the answer $W = 1 \land P = 1 \land C = 1 \land L = -32$ corresponding to a profit of 32.

1. The American spelling of "labelling"

Some care must be taken when using `minimize` with finite domain constraints. For `minimize` to work correctly, execution of the minimization goal must ensure that the variables in the objective function take fixed values by the time the goal has executed and that a complete solver is employed when executing the goal. For this reason, it is usual for `labeling` to be a literal in the minimization goal.

The crypto-arithmetic problem from the introduction provides a slightly more complex example of modelling with $CLP(FD)$. The problem is to solve the arithmetic equation

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

where each letter represents a different digit.

This is modelled by the following program.

```
smm(S,E,N,D,M,O,R,Y) :-
     [S,E,N,D,M,O,R,Y] :: [0..9],
     constrain([S,E,N,D,M,O,R,Y]),
     labeling([S,E,N,D,M,O,R,Y]).

constrain([S,E,N,D,M,O,R,Y]) :-
     S ≠ 0, M ≠ 0,
     alldifferent_neq([S,E,N,D,M,O,R,Y]),
                 1000*S + 100*E + 10*N + D
     +           1000*M + 100*O + 10*R + E
     = 10000*M + 1000*O + 100*N + 10*E + Y.
```

The program works as follows. Each of the variables is declared to range over the values [0..9] while the constraints defined by the crypto-arithmetic problem are expressed by the user-defined constraint `constrain`. To ensure that a valuation is returned, the built-in `labeling` is called to make sure that each variable is set to one of the values in the range [0..9].

The `constrain` predicate ensures that neither S nor M take the value zero since they are digits that appear leftmost in a number. Each of the variables is forced to be different by the user-defined constraint `alldifferent_neq` from Example 6.3 which sets up disequation constraints between each pair of variables. Finally the arithmetic equation is represented by the last constraint.

The program executes the goal `smm(S,E,N,D,M,O,R,Y)` as follows. After the domain declarations the domain is

$$\begin{array}{llll} D(S) = [0..9], & D(E) = [0..9], & D(N) = [0..9], & D(D) = [0..9], \\ D(M) = [0..9], & D(O) = [0..9], & D(R) = [0..9], & D(Y) = [0..9]. \end{array}$$

Adding the constraints $S \neq 0$ and $M \neq 0$ narrows the domain so that $D(S) =$ [1..9] and $D(M) =$ [1..9]. The `alldifferent_neq` predicate adds the disequalities

$S \neq E, S \neq N, S \neq D, \ldots, R \neq Y$ to the constraint store. As each of these constraints involves variables with no fixed value, no propagation occurs. Addition of the equation initiates propagation. For simplicity, we shall assume the constraint solver treats the final equation in its simpler (equivalent) linear form of

$$1000 * S + 91 * E + D + 10 * R = 9000 * M + 900 * O + 90 * N + Y.$$

One of the propagation rules for this constraint is based on the inequality that M is less than or equal to

$$\begin{array}{l} \frac{1}{9}max_D(S) + \frac{91}{9000}max_D(E) + \frac{1}{9000}max_D(D) + \frac{1}{900}max_D(R) \\ -\frac{1}{10}min_D(O) - \frac{1}{100}min_D(N) - \frac{1}{9000}min_D(Y). \end{array}$$

Given the current domains this implies that

$$M \leq \frac{9}{9} + \frac{91*9}{9000} + \frac{9}{9000} + \frac{9}{900} - \frac{0}{10} - \frac{0}{100} - \frac{0}{9000} \;=\; 1.102.$$

Thus $D(M)$ is updated to [1..1]. From propagation using the constraint $S \neq M$ we obtain $D(S)$ = [2..9]. Next the propagation rule based on the inequality that S is greater than

$$\begin{array}{l} 9\ min_D(M) + \frac{9}{10}min_D(O) + \frac{9}{100}min_D(N) + \frac{1}{1000}min_D(Y) \\ -\frac{91}{1000}max_D(E) - \frac{1}{1000}max_D(D) - \frac{1}{100}max_D(R) \end{array}$$

is used with the current domain to infer that $S \geq 8.082$. Propagation therefore sets $D(S)$ to [9..9]. Now using the constraint $S \neq E$ we obtain that $D(E)$ is [0..8] and similarly we obtain that the domains of N, D, O, R, Y are also [0..8].

Propagation continues until the following domain is obtained:

$$\begin{array}{llll} D(S) = [9..9], & D(E) = [4..7], & D(N) = [5..8], & D(D) = [2..8], \\ D(M) = [1..1], & D(O) = [0..0], & D(R) = [2..8], & D(Y) = [2..8]. \end{array}$$

Notice that three of the variables already have fixed values and that every other variable has had its range of possible values decreased. Since this is not a valuation domain, the constraint solver cannot determine whether the constraint store is satisfiable or not and so returns *unknown*. In order to guarantee that a valid solution is found we must use `labeling`.

Essentially the action of the `labeling` built-in is to backtrack through each variable's initial domain setting the variable to each of the values from 0 to 9 in turn. The variables are tried in order of their appearance in the list passed to `labeling`. Thus, the first variable to be assigned a value is S. Trying the first value in the initial domain, 0, will cause failure since the constraint $S = 0$ is inconsistent with the current domain of S. So the next value, 1, is tried. This also causes failure. Subsequent values lead to failure until we try $S = 9$, which is consistent. Now `labeling` tries to find a value for E. First we add $E = 0$, which fails and we continue trying values for E until we try $E = 4$. Propagation reduces

$D(N)$ to [5..5] and $D(R)$ to [8..8] and then finds failure because variable D has no possible value. Removing the last choice $E = 4$, we now add $E = 5$ and propagation determines that $N = 6$, $R = 8$, $D = 7$ and $Y = 2$. Execution of `labeling` continues until (eventually) each of these values is selected and so we have found a solution to the problem. Since the resulting domain is a valuation domain, the underlying propagation based constraint solver will return the answer *true*, guaranteeing that this a valid answer.

This example demonstrates the power of the constraint solver to direct the search. In contrast, consider a *generate and test* methodology for solving the same problem. Generate and test is the reverse of the constrain and generate methodology. In generate and test, first possible solutions are generated and then tested to see if they satisfy the constraints. The generate and test methodology gives rise to the following program:

```
smm_gt(S,E,N,D,M,O,R,Y) :-
        [S,E,N,D,M,O,R,Y] :: [0..9],
        labeling([S,E,N,D,M,O,R,Y]),
        constrain([S,E,N,D,M,O,R,Y]).
```

In this program a unique value for each of the variables is first chosen by `labeling`, then the constraints of the problem are checked by `constrain`. For this program the solution is found after testing 95671082 choices, contrasting with the first program in which the solution was found after making only one choice, $E = 4$, that did not result in immediate failure.

In this chapter we shall often compare different programs for the same problem, by discussing the number of choices each makes. This is because for the same problem, the size of the derivation tree is roughly proportional to the number of choices in the tree. Thus comparing the number of choices provides a rough measure of the program's relative efficiency.

As this example demonstrates, constrain and generate is almost always preferable to generate and test. This is a simple consequence of the guideline we discussed in Section 7.4, namely that it is usually more efficient to call predicates with many possible answers after those with a few answers. Since labelling usually has a huge number of possible answers, it is better to delay calling `labeling` until last.

8.2 Complex Constraints

In Section 3.5 we saw consistency based solvers for finite domains often provide complex primitive constraints such as `alldifferent`, `cumulative` or `element`. When possible, the constraint programmer should make use of these complex primitive constraints. One reason is that they often allow a succinct description of the problem. A second, more important reason, is that they will, usually, lead to a more efficient program since the constraint solver provides specialized propagation

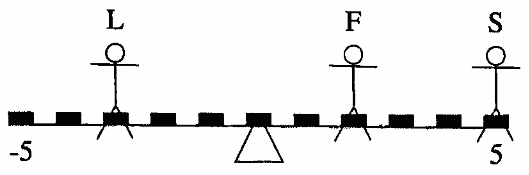

Figure 8.1 Balancing a seesaw.

rules for these constraints which will be more effective at domain pruning than those for the equivalent "simple" primitive constraints.

For example, the old fashioned marriage problem above could be modelled by the goal

```
[N,M,E] :: [kim, peter,bernd], alldifferent([N,M,E]),
likes(N, nicole), likes(M, maria), likes(E, erika).
```

in which the primitive constraints $N \neq M$, $N \neq E$, and $M \neq E$ have been replaced by a call to the complex primitive constraint `alldifferent`.

Sometimes a degree of ingenuity is required in order to find a way of modelling a problem by using complex primitive constraints. This effort is worthwhile provided the constraint solver handles the complex constraint well.

Example 8.1

Suppose that Liz, Fi and Sarah are playing on a 10 foot long seesaw which has seats placed uniformly one foot apart along the bar. The seesaw is shown in Figure 8.1. They wish to position themselves on the seesaw so that it balances. They also wish to be able to swing their arms freely, requiring that they are at least three feet apart. The weights of Liz, Fi and Sarah are respectively 9, 8 and 4 stone.

To solve the problem, we need to assign seats to Liz, Fi and Sarah. If we number the seats on the seesaw from -5 to 5, we can use the variables L, F and S to represent the number of the seat of Liz, Fi and Sarah, respectively. The constraint that the seesaw balances is simply that the moments of inertia sum to 0, that is:

$$9 \times L + 8 \times F + 4 \times S = 0.$$

The "at least 3 apart" constraint can be modelled with the user-defined constraint `apart(X,Y,N)` which holds if integers X and Y are at least N apart. It is defined by the two rules:

```
apart(X, Y, N) :- X ≥ Y + N.
apart(X, Y, N) :- Y ≥ X + N.
```

The complete problem is modelled by the goal

```
[L,F,S] :: [-5..5], 9 * L + 8 * F + 4 * S = 0,
apart(L,F,3), apart(L,S,3), apart(F,S,3),
labeling([L,F,S]).
```

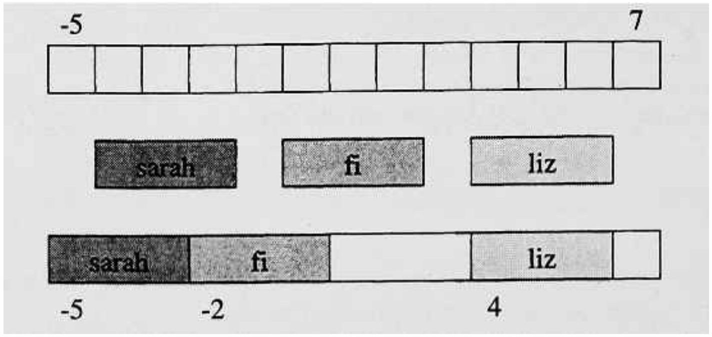

Figure 8.2 A cumulative constraint for apartness.

Notice that for efficiency, we have ordered the literals in the goal according to their degree of non-determinism. The deterministic literals `[L,F,S] :: [-5..5]` and `9 * L + 8 * F + 4 * S = 0` are first, while the literal with most possible answers, `labeling`, is last.

However, using the complex primitive constraint `cumulative` it is possible to give a model which is more efficient. The key is that the three apartness constraints can be modelled by a single cumulative constraint. If we think of a child as a box of width 3 to be placed in the range −5 to 7, then, if their boxes do not overlap, all children are at least three feet apart. Figure 8.2 shows how this is an instance of a cumulative constraint. Once *sarah* is placed at position −5, then no other box (child) can be placed in the range [−5..−3]. We can therefore model the problem using the goal

```
    [L,F,S] :: [-5..5], 9 * L + 8 * F + 4 * S = 0,
    cumulative([L,F,S],[3,3,3],[1,1,1],1),
    labeling([L,F,S]).
```

Since the `cumulative` constraint is deterministic but the `apart` constraint is non-deterministic, this model is more efficient than our original model. Each call to `apart` gives rise to a binary choice in the derivation tree. This means the derivation tree of the original goal is considerably larger than that of the new goal.

8.3 Labelling

As we have seen, *CLP*(*FD*) programs typically end with a call to a labelling predicate which uses a backtracking search to find a solution to the constraints. For many problems, most of the execution time occurs during the execution of this labelling predicate. Indeed, it is quite common for all of the search to occur during labelling. Thus for many *CLP*(*FD*) programs, efficiency is synonymous with efficient labelling and the constraint programmer is well advised to consider strategies for labelling which reduce the search space and lead to finding a (first) solution faster.

Fortunately, labelling predicates may be readily programmed in $CLP(FD)$ itself, making it easy for the programmer to develop and experiment with different strategies. To facilitate such programming $CLP(FD)$ systems provide several built-in predicates which allow the programmer to access information about the current domain of a variable. We will make use of:

- `dom(V,D)` which returns the list of values D in the current domain of variable V,
- `maxdomain(V,Max)` which returns the maximum value Max in the current domain of variable V, and
- `mindomain(V,Min)` which returns the minimum value Min in the current domain of variable V.

Using `dom(X,D)` it is simple to write a recursive definition of `labeling`:

```
labeling([]).
labeling([V|Vs]) :-
     indomain(V),
     labeling(Vs).

indomain(V) :-
     dom(V,D),
     member(V,D).
```

The predicate `labeling` iterates through each variable V in the list of variables to be labelled, calling `indomain(V)` to set V to each of the remaining values in its domain. The predicate `indomain(V)` calls `dom(V,D)` to determine the values in the domain of V and then uses `member` to backtrack through these values. Note that this definition of `labeling` is marginally more efficient than that described in Section 8.1 since it only backtracks through the values in the current domain of a variable instead of the values in the original domain. To understand its definition consider execution of the goal `smm(S,E,N,D,M,O,R,Y)`. Execution proceeds as described in Section 8.1 until the literal `labeling([S,E,N,D,M,O,R,Y])` is called. This calls `indomain(S)` to backtrack through the values in the current domain of S. This immediately adds the equation $S = 9$, rather than trying $S = 0$, $S = 1, \dots, S = 8$. This is because the current domain of S is [9..9] and so `dom(S,D)` will constrain D to be the list [9] and the call to `member` adds $S = 9$. Next `indomain(E)` is called. This is rewritten to `member(E, [4, 5, 6, 7])` since $D(E) = [4..7]$. Execution of the literal `member(E, [4, 5, 6, 7])` first adds $E = 4$ which fails, and then adds $E = 5$. Since the domains of the remaining variables are now all singletons, execution of the subsequent calls to `indomain` essentially does nothing.

There are two choices to be made when labelling: the order in which variables are labelled and the order in which the values for a particular variable are explored. We will investigate these in turn.

8.3.1 Choice of Variable

Altering the order in which variables are labelled can have dramatic effects on the size of the derivation tree. We have seen that a useful heuristic to minimize the amount of repeated work and the size of the derivation tree is to delay choice as long as possible, calling literals with many possible answers after those with fewer possible answers. This heuristic suggests that when labelling a list of variables, we should choose to label those variables with the smallest current domain first. A further motivation for this strategy is that, with luck, once these variables are assigned a value, propagation will trim the domains of the remaining variables.

In the smm program when the labelling predicate is reached the range domain is

$$\begin{array}{llll} D(S) = [9..9], & D(E) = [4..7], & D(N) = [5..8], & D(D) = [2..8], \\ D(M) = [1..1], & D(O) = [0..0], & D(R) = [2..8], & D(Y) = [2..8]. \end{array}$$

Since the variables S, M and O have singleton domains we should label these variables first (effectively doing nothing but remembering they are "dealt with"). Both E or N have domains with 4 elements, while the remaining variables have larger domains. Thus, a good strategy is to choose either E or N to label next.

We can program a labelling predicate that chooses the variable to label based on its domain size as follows:

```
labelingff([]).
labelingff(Vs) :-
     deleteff(Vs, V, Rest),
     indomain(V),
     labelingff(Rest).

deleteff([V0|Vs], V, Rest) :-
     get_domain_size(V0, S0),
     min_size(Vs, S0, V0, V, Rest).

min_size([], _, V, V, []).
min_size([V1|Vs], S0, V0, V, [V0|Rest]) :-
     get_domain_size(V1, S1),
     S1 < S0,
     min_size(Vs, S1, V1, V, Rest).
min_size([V1|Vs], S0, V0, V, [V1|Rest]) :-
     get_domain_size(V1, S1),
     S1 ≥ S0,
     min_size(Vs, S0, V0, V, Rest).

get_domain_size(V, S) :- dom(V, D), length(D, S).

length([], 0).
length([_|L], N) :- N = N1 + 1, length(L, N1).
```

At each stage the predicate `labelingff` selects the variable with the fewest elements in its current domain. The call `get_domain_size(V, S)` determines the current domain size of variable V using the built-in `dom(V,D)` and the user-defined constraint `length` which counts the number of elements in a list. We note that `get_domain_size` for range domains could also have been defined by the more efficient code:

```
get_domain_size(V, S) :-
    mindomain(V, Min), maxdomain(V, Max), S = Max - Min + 1.
```

However, this code does not give an accurate size if the domain is not a range.

The predicate `min_size(Vs, S0, V0, V, Rest)` iterates through the list of variables Vs looking for a variable which has a smaller domain than the variable $V0$ which has $S0$ elements in its domain. On return, V is set to the variable with smallest domain size amongst $V0$ and the Vs and the list of remaining variables is returned in *Rest*.

This labelling strategy can be significantly more efficient than the naive strategy of simply labelling the variables in order; so much so that `deleteff` is often provided as a built-in predicate. It is an example of a labelling based on the *first fail principle*.

"To succeed, try first where you are most likely to fail."

Choosing the variable with smallest domain limits the branching factor and potentially allows failure of the entire branch to be determined quickly, thus guiding the search to more profitable areas.

There are other factors that can be taken into account in applying the first fail principle. For example, we might assume that a variable involved in many constraints is more likely to cause failure when it is set to a value. Thus some $CLP(FD)$ systems provide a built-in predicate `deleteffc` which takes a list of variables and first selects the variables with smallest current domains, and then selects from these, the variable involved in the most constraints.

8.3.2 Choice of Domain Value

The second choice in labelling is the order in which to try the different domain values. Different choices will change the order in which solutions are found and the order in which branches in the derivation tree are explored. Using problem-specific knowledge, we may be able to select a value that is more likely to lead to a solution, helping us to find a (first) solution faster. Ordering of domain values is also important when evaluating optimization goals. In this case, it may mean that the optimum solution is found more quickly and other branches which cannot lead to a better solution are pruned.

The following example illustrates the importance of appropriately ordering the choice of domain values.

Example 8.2
Consider the *N-queens problem* introduced in Chapter 3. This is the problem of placing N queens on a chess board of a size $N \times N$ so that no queen can capture another queen. Another way of modelling this problem (different from that in Chapter 3) is to recognize that in any solution there is exactly one queen in every column. We associate with the i^{th} column the variable R_i which represents the row number of the queen in that column. The following program models the N-queens problem using this representation.

```
goal(N, Queens) :-
     length(Queens, N),
     Queens :: [1..N],
     queens(Queens),
     labeling(Queens).

queens([]).
queens([X|Y]) :-
     safe(X,Y,1),
     queens(Y).

safe(_,[],_).
safe(X,[F|T],Nb) :-
     noattack(X,F,Nb),
     Newnb = Nb + 1,
     safe(X,T,Newnb).

noattack(X,Y,Nb) :- Y + Nb ≠ X, X + Nb ≠ Y, X ≠ Y.
```

Evaluation of the literal `goal(N, Queens)` first uses the user-defined predicate `length` to constrain *Queens* to be a list of N distinct variables. These correspond to the row variables $R_1, \ldots, R_N$. Next each row variable is given an initial domain of 1 to N. The predicate `queens` ensures that no queen falls on the same row or diagonal as any other queen. It iterates through the list of queens calling `safe` to ensure that each queen X does not fall on the same row or diagonal as the remaining queens in the list Y. The predicate `safe` iterates through the queens in Y enforcing that each queen F in the list is not on the same row or diagonal as X. Finally the labelling occurs.

For large N it is well-known that solutions to the N-queens problems are more likely to be found by starting in the middle of the domain. Unfortunately, execution of the N-queens program using `indomain` in labelling will first try the smallest value in the domain, then the next smallest, and so on. We can program a better labelling strategy as follows.

```
indomain_middle(Var) :-
     make_list_of_values(Var, List),
     member(Var, List).
```

No. of queens	10–19	20–29	30–39	40–49	50–59	60–69	70–79	100–109
Straight	777.6	—	—	—	—	—	—	—
First fail	26.0	51.0	80.5	176	1331.3	189.6	1337.4	—
Middle out	28.5	47.7	43.1	50.7	79.2	73.6	81.5	106.7

Table 8.1 Empirical comparison of different labelling strategies.

```
make_list_of_values(Var, List) :-
     dom(Var, Domain),
     length(Domain, Size),
     N = Size div 2,
     halve_list(N, Domain, [], RFirst, Second),
     merge(Second, RFirst, List).

halve_list(0, L2, L1, L1, L2).
halve_list(N, [H|R], Li, L1, L2) :-
     N ≥ 1,
     N1 = N-1,
     halve_list(N1, R, [H|Li], L1, L2).

merge([], L, L).
merge([X|L1], L2, [X|L3]) :- merge(L2, L1, L3).
```

The program works by taking the current domain of a variable Var as a list and breaking it into two halves using `halve_list`. The predicate `halve_list` succeeds with $RFirst$ as the reverse of the first half of the list and *Second* set to the remaining part. In the call,

```
     halve_list(3, [1,2,5,6,8,9,10], [], RFirst, Second).
```

succeeds with $Rfirst = [5,2,1]$ and $Second = [6,8,9,10]$. The predicate `merge` then merges the two lists, interleaving the elements. In this example, the resulting list is $[6,5,8,2,9,1,10]$. Finally `indomain_middle` uses `member` to set Var to each of these values in turn.

To see the benefit of the middle ordering heuristic we can compare the different labellings empirically. Table 8.1 gives figures for the average number of choices required to find the first solution over different ranges for N, using straight labelling, the first fail principle, and the first fail principle with the middle ordering heuristic above. We use "—" to indicate greater than 10000.

Our results clearly demonstrate that the first fail principle is more efficient than straight labelling and that combining first fail with the middle out labelling heuristic reduces the search space even more. This example also illustrates the importance of empirical testing to compare different labelling strategies.

8.3.3 Domain Splitting

Labelling strategies, however, need only ensure that every variable is eventually constrained to take a single domain value (so that the incomplete solver will answer *true* or *false*). The strategies we have seen so far choose a variable and a value and assign the value to the variable, but other types of labelling strategies are also possible.

One useful strategy is to repeatedly reduce the domain of each variable, by splitting the current domain into two possibilities, for example the lower and upper halves. This approach to labelling is justified by the *principle of least commitment*. This states that when making a choice for some variable we should make the choice which "commits us as little as possible." The advantage is that, if we detect unsatisfiability after making a weak commitment, more of the search space is removed than if we detect unsatisfiability after a strong commitment. Setting a variable to a value commits the variable to a single value, which is very restrictive. Domain splitting is less restrictive. It only removes half of the remaining values in the variable's domain rather than all but one. The principle of least commitment, therefore, tells us to prefer domain splitting since the usual labelling approach leads to a stronger commitment. Of course the drawback is that less commitment may create less information for the incomplete solver to determine satisfiability or unsatisfiability.

The following program captures the domain splitting strategy. It repeatedly splits the domains of variables in half until they are singletons.

```
labelingsplit([]).
labelingsplit([V|Vs]) :-
    mindomain(V, Min),
    maxdomain(V, Max),
    (Min = Max ->
        labelingsplit(Vs)
    ;
        Mid = (Min + Max) div 2,
        labelsplit(V, Mid, Vs)
    ).
labelsplit(V, M, Vs):-
    V ≤ M,
    append(Vs, [V], NVs),
    labelingsplit(NVs).
labelsplit(V, M, Vs):-
    V ≥ M+1,
    append(Vs, [V], NVs),
    labelingsplit(NVs).
```

The predicate `labelingsplit` selects the first variable V and, using the built-ins `mindomain` and `maxdomain`, determines its minimum and maximum current domain values. If V's current domain is a singleton, then the domain is completely split and so `labelingsplit` calls itself recursively to split the domains of the remaining variables. Otherwise, the midpoint Mid of the range is determined and `labelsplit` forces the choice of either the higher or lower half of the values for V. V is placed on the end of the list of variables to be handled using append and domain splitting continues.

Example 8.3

Consider the seesaw problem of Example 8.1. The problem can be modelled by the goal

```
[L,F,S] :: [-5..5], 9 * L + 8 * F + 4 * S = 0,
cumulative([L,F,S],[3,3,3],[1,1,1],1),
labelingsplit([L,F,S]).
```

Initial propagation does nothing so when computation reaches the labelling the domain is

$$D(L) = [-5..5], \quad D(F) = [-5..5], \quad D(S) = [-5..5].$$

The first split is on the domain of L. This adds $L \leq 0$. This modifies the domain to

$$D(L) = [-5..0], \quad D(F) = [-2..5], \quad D(S) = [-5..5].$$

The next split is on the domain of F. This adds $F \leq 1$ and propagation modifies the domain to

$$D(L) = [-3..-2], \quad D(F) = [0..1], \quad D(S) = [3..5].$$

Now we split on S. This adds $S \leq 4$ which fails, and then tries $S \geq 5$ which also fails.

Execution backtracks to try the other split on F, $F \geq 2$. Consistency modifies the domain to

$$D(L) = [-5..0], \quad D(F) = [2..5], \quad D(S) = [-5..5].$$

Now the domain of S is split. This adds $S \leq 0$ which leaves the domain as

$$D(L) = [-4..0], \quad D(F) = [2..5], \quad D(S) = [-5..0].$$

Now the domain of L is split again (the second split for L in this branch). This adds $L \leq -2$ giving rise to

$$D(L) = [-4..-2], \quad D(F) = [3..5], \quad D(S) = [-5..0].$$

Finally the domain of F is split again. This adds $F \leq 4$ which fails, and then $F \geq 5$ which reduces the domain to

$$D(L) = [-4..-4], \quad D(F) = [5..5], \quad D(S) = [-1..-1].$$

well, more natural models of the problem are possible.

We shall now model the problem by using four variables W_1, W_2, W_3 and W_4 which represent the four workers. If the i^{th} worker works on product j then W_i takes value j. The assignment constraints are simply that each variable takes a different value. That is, $W_i \neq W_k$ for all $1 \leq i \neq k \leq 4$. We can use the complex primitive constraint `alldifferent` to model this.

Evaluating the total profit is more difficult. We need to know the profit generated by each worker (one working on each of the four products). This can be encoded using four arrays, one for each worker. The array $profit_i$ details the profit of worker i for each product with $profit_i[j]$ being the profit if product j is made by that worker. For instance,

$$profit_1[1] = 7, \quad profit_1[2] = 1, \quad profit_1[3] = 3, \quad profit_1[4] = 4.$$

The profit of worker i, WP_i, is found by looking up $profit_i$ for the value of W_i.

In Chapter 3 we introduced the `element` complex primitive constraint. It is used to mimic array lookup. We can use a `element` constraint to model each of the $profit_i$ arrays. For instance,

$$element(W_1, [7,1,3,4], WP_1)$$

mimics the constraint that WP_1 is $profit_1[W_1]$.

Example 8.6
A second model for the simple assignment problem using worker variables is

```
[W1,W2,W3,W4] :: [1..4],
alldifferent([W1,W2,W3,W4]),
element(W1,[7,1,3,4],WP1),
element(W2,[8,2,5,1],WP2),
element(W3,[4,3,7,2],WP3),
element(W4,[3,1,6,3],WP4),
P = WP1 + WP2 + WP3 + WP4,
P ≥ 19,
labeling([W1,W2,W3,W4]).
```

Since only 14 choices are required to find all solutions to the problem, this model is more efficient than the first model. Arguably, this model is also simpler since there are 7 primitive constraints, as opposed to 10 in the first model, although five of these (the `alldifferent` and `element` constraints) are more complex.

The principal difference between these models occurs because of the different solver and primitive constraints with which the constraint programmer had to work. However, it is not only different solvers which lead to different models. We can just as easily model the assignment problem by using variables to represent the products and by selecting a worker for each product. This is analogous to the second model but swaps the role of worker and product.

Example 8.7

Therefore, a third model for the simple assignment problem using product variables is

```
[T1,T2,T3,T4] :: [1..4],
alldifferent([T1,T2,T3,T4]),
element(T1,[7,8,4,3],TP1),
element(T2,[1,2,3,1],TP2),
element(T3,[3,5,7,6],TP3),
element(T4,[4,1,2,3],TP4),
P = TP1 + TP2 + TP3 + TP4,
P ≥ 19,
labeling([T1,T2,T3,T4]).
```

Like the second model there are 7 primitive constraints. Somewhat surprisingly, however, this model is more efficient than the second since only 7 choices are required to find all solutions to the problem. Why is the derivation tree smaller in this model? The reason is that profit depends more on the product than on the worker. If we examine the profits for each product TP_1,TP_2,TP_3 and TP_4 we see that the initial domains are [3..8], [1..3], [3..7] and [1..4] respectively. The propagation rule for $P = TP_1 + TP_2 + TP_3 + TP_4$ based on the inequality

$$TP_1 \geq min_D(P) - max_D(TP_2) - max_D(TP_3) - max_D(TP_4)$$

determines that $TP_1 \geq 5$. The constraint $element(T_1, [7,8,4,3], TP_1)$ therefore reduces the domain of T_1 to [1..2]. Similar reasoning also reduces the domain of T_3. This reduction to the domain of T_1 and T_3 before labelling starts leads to the improvement in search.

Determining the relative efficiency of different models is a difficult problem and one which relies upon an understanding of the underlying constraint solver. The best model will be the one in which information is propagated earliest. This often means that the more direct the mapping from the problem into the primitive constraints in the program, the better the model. Another guideline is that the fewer the variables the better, so long as propagation of information is similar. This is because it means there will be less variables to label. Given the difficulty of understanding interaction between the constraints and the solver it is often useful to empirically evaluate the different models.

In the above example the last two models are better since they model the problem more naturally as an assignment of one of four objects to each of four objects. The Boolean formulation is less natural, requiring more variables and more constraints. Since these constraints are quite complex, their propagation rules are slower to propagate information. Of course the choice of model also depends on the constraint solver. The first model is clearly the best choice when disequations and complex primitive constraints, such as `element`, are not available.

This is a valuation domain so the domain is returned as the answer.

For the example above, domain splitting produces a larger derivation tree than ordinary labelling. This is because the problem is so small. However, when there are many variables and their domains are large, domain splitting can be far more efficient than labelling. In general, the domain splitting approach is better than regular labelling when the constraints involving the variables to be labelled can gain substantial consistency information from the split domain since, in this case, by using domain splitting we may be able to eliminate half of a variable's possible domain values by adding a single constraint.

8.4 Different Problem Modellings

An important issue that arises when solving problems over finite domains is that there is often more than one way to model the problem. Different models arise because other ways of viewing a problem lead to different formulations involving different variables and constraints. These models may have very different efficiency depending on the number of variables and types of constraints in the model.

Ideally, the constraint programmer wishes to find a model which can be efficiently evaluated by the solver in their CLP system. However, reasoning about efficiency is quite difficult. This is because of the incomplete nature of the constraint solver—in some models search will be pruned by the solver while in others it will not, even though the constraints are equivalent. If efficiency is an issue (as it usually is), it is worthwhile investigating different models of the problem empirically.

Example 8.4
Consider a simple assignment problem. A factory has four workers w_1, w_2, w_3 and w_4, and four products p_1, p_2, p_3 and p_4. The problem is to assign workers to products so that each worker is assigned to one product, and each product is assigned to one worker. The profit made by worker w_i working on product p_j is given in the table below.

	p_1	p_2	p_3	p_4
w_1	7	1	3	4
w_2	8	2	5	1
w_3	4	3	7	2
w_4	3	1	6	3

A solution to this assignment problem is acceptable if the total profit is at least 19.

A traditional approach to modelling this kind of problem comes from the operations research community. The problem is modelled by 16 Boolean variables B_{ij}, with each corresponding to the proposition "worker i is assigned to product j". Each Boolean is represented by an integer ranging between 0 and 1. Using Boolean variables we can model the problem with linear arithmetic constraints. The constraint c_{wi} which is $B_{i1} + B_{i2} + B_{i3} + B_{i4} = 1$ ensures that worker w_i is assigned to exactly one task. Similarly, the constraint c_{pj} which is $B_{1j} + B_{2j} + B_{3j} + B_{4j} = 1$ ensures that product p_j is assigned to exactly one worker. The total profit P is

$$P = 7 \times B_{11} + B_{12} + 3 \times B_{13} + 4 \times B_{14} + 8 \times B_{21} + 2 \times B_{22} + 5 \times B_{23} + B_{24} + \\ 4 \times B_{31} + 3 \times B_{32} + 7 \times B_{33} + 2 \times B_{34} + 3 \times B_{41} + B_{42} + 6 \times B_{43} + 3 \times B_{44}.$$

Thus, the requirement for enough profit is simply the constraint $P \geq 19$.

Note that the conjunction of the four worker constraints $\wedge_{i=1}^{4} c_{wi}$ and the four product constraints $\wedge_{j=1}^{4} c_{pj}$ imply that the sum of all 16 variables is 4. It follows that any one of these primitive constraints is redundant and so can be eliminated from the model. Whether or not this is worthwhile depends on the underlying solver. With a propagation based solver, the extra constraint may lead to more propagated information and so it is better to include it.

Example 8.5

The following constraint models the simple assignment problem using Boolean variables:

```
[B11,B12,B13,B14,B21,B22,B23,B24,B31,B32,
 B33,B34,B41,B42,B43,B44] :: [0,1],
B11 + B12 + B13 + B14 = 1,
B21 + B22 + B23 + B24 = 1,
B31 + B32 + B33 + B34 = 1,
B41 + B42 + B43 + B44 = 1,
B11 + B21 + B31 + B41 = 1,
B12 + B22 + B32 + B42 = 1,
B13 + B23 + B33 + B43 = 1,
B14 + B24 + B34 + B44 = 1,
P =  7 * B11 + B12 + 3 * B13 + 4 * B14
     + 8 * B21 + 2 * B22 + 5 * B23 + B24
     + 4 * B31 + 3 * B32 + 7 * B33 + 2 * B34
     + 3 * B41 + B42 + 6 * B43 + 3 * B44,
P ≥ 19,
labeling([B11,B12,B13,B14,B21,B22,B23,B24,B31,B32,
          B33,B34,B41,B42,B43,B44]).
```

The goal above will find each of the four solutions to the problem using a total of 28 choices. That is, when executing the labelling predicate, there are 28 derivation steps in the tree in which there is a choice of a value for a variable.

This modelling approach arises from the operations research approach to solving integer constraints. They use a real linear constraint solver together with some kind of enumeration such as "branch and bound" (discussed in Section 3.6). Because of the choice of solver, there is no direct way to represent disequations ($X \neq Y$) and so Boolean variables are used. Given a constraint solver that handles disequations well, more natural models of the problem are possible.

We shall now model the problem by using four variables W_1, W_2, W_3 and W_4 which represent the four workers. If the i^{th} worker works on product j then W_i takes value j. The assignment constraints are simply that each variable takes a different value. That is, $W_i \neq W_k$ for all $1 \leq i \neq k \leq 4$. We can use the complex primitive constraint `alldifferent` to model this.

Evaluating the total profit is more difficult. We need to know the profit generated by each worker (one working on each of the four products). This can be encoded using four arrays, one for each worker. The array $profit_i$ details the profit of worker i for each product with $profit_i[j]$ being the profit if product j is made by that worker. For instance,

$$profit_1[1] = 7, \quad profit_1[2] = 1, \quad profit_1[3] = 3, \quad profit_1[4] = 4.$$

The profit of worker i, WP_i, is found by looking up $profit_i$ for the value of W_i.

In Chapter 3 we introduced the `element` complex primitive constraint. It is used to mimic array lookup. We can use a `element` constraint to model each of the $profit_i$ arrays. For instance,

$$element(W_1, [7,1,3,4], WP_1)$$

mimics the constraint that WP_1 is $profit_1[W_1]$.

Example 8.6

A second model for the simple assignment problem using worker variables is

```
[W1,W2,W3,W4] :: [1..4],
alldifferent([W1,W2,W3,W4]),
element(W1,[7,1,3,4],WP1),
element(W2,[8,2,5,1],WP2),
element(W3,[4,3,7,2],WP3),
element(W4,[3,1,6,3],WP4),
P = WP1 + WP2 + WP3 + WP4,
P ≥ 19,
labeling([W1,W2,W3,W4]).
```

Since only 14 choices are required to find all solutions to the problem, this model is more efficient than the first model. Arguably, this model is also simpler since there are 7 primitive constraints, as opposed to 10 in the first model, although five of these (the `alldifferent` and `element` constraints) are more complex.

The principal difference between these models occurs because of the different solver and primitive constraints with which the constraint programmer had to work. However, it is not only different solvers which lead to different models. We can just as easily model the assignment problem by using variables to represent the products and by selecting a worker for each product. This is analogous to the second model but swaps the role of worker and product.

Example 8.7
Therefore, a third model for the simple assignment problem using product variables is

```
[T1,T2,T3,T4] :: [1..4],
alldifferent([T1,T2,T3,T4]),
element(T1,[7,8,4,3],TP1),
element(T2,[1,2,3,1],TP2),
element(T3,[3,5,7,6],TP3),
element(T4,[4,1,2,3],TP4),
P = TP1 + TP2 + TP3 + TP4,
P ≥ 19,
labeling([T1,T2,T3,T4]).
```

Like the second model there are 7 primitive constraints. Somewhat surprisingly, however, this model is more efficient than the second since only 7 choices are required to find all solutions to the problem. Why is the derivation tree smaller in this model? The reason is that profit depends more on the product than on the worker. If we examine the profits for each product TP_1,TP_2,TP_3 and TP_4 we see that the initial domains are [3..8], [1..3], [3..7] and [1..4] respectively. The propagation rule for $P = TP_1 + TP_2 + TP_3 + TP_4$ based on the inequality

$$TP_1 \geq min_D(P) - max_D(TP_2) - max_D(TP_3) - max_D(TP_4)$$

determines that $TP_1 \geq 5$. The constraint $element(T_1, [7, 8, 4, 3], TP_1)$ therefore reduces the domain of T_1 to [1..2]. Similar reasoning also reduces the domain of T_3. This reduction to the domain of T_1 and T_3 before labelling starts leads to the improvement in search.

Determining the relative efficiency of different models is a difficult problem and one which relies upon an understanding of the underlying constraint solver. The best model will be the one in which information is propagated earliest. This often means that the more direct the mapping from the problem into the primitive constraints in the program, the better the model. Another guideline is that the fewer the variables the better, so long as propagation of information is similar. This is because it means there will be less variables to label. Given the difficulty of understanding interaction between the constraints and the solver it is often useful to empirically evaluate the different models.

In the above example the last two models are better since they model the problem more naturally as an assignment of one of four objects to each of four objects. The Boolean formulation is less natural, requiring more variables and more constraints. Since these constraints are quite complex, their propagation rules are slower to propagate information. Of course the choice of model also depends on the constraint solver. The first model is clearly the best choice when disequations and complex primitive constraints, such as `element`, are not available.

Efficiency is not the only criteria by which to compare models. Flexibility is also an important issue. It is unusual for an application to remain static. Often there will be a need to extend the program to model constraints which were not part of the problem's original formulation. Flexibility measures how easy it is to express these additional constraints in the model. Of course the flexibility of a model greatly depends on the form of these new constraints. A model may be flexible for one class of constraints but not for another.

As an example consider the three models for the assignment problem. Suppose we wish to add the requirement that we cannot have both "worker 1 assigned to product 1" and "worker 4 assigned to product 4." In the Boolean formulation this is easily expressed as the constraint $B_{11} + B_{44} \leq 1$. It is considerably more difficult to express this requirement in either of the other two models. In the second model, one approach would be to add Boolean variables B_{11} and B_{44} and define them in terms of W_1 and W_4 using `element` constraints:

```
[B11,B44] :: [0,1],
element(W1,[1,0,0,0],B11),
element(W4,[0,0,0,1],B44),
B11 + B44 ≤ 1.
```

The element constraints ensure that if $W_1 = 1$ then $B_{11} = 1$ and vice versa. A similar extension can also be made to the third model.

As another example, suppose a new requirement is that worker 3 must work on a product numbered greater than that which is worked on by worker 2. In the second model, this is simply the constraint $W_3 > W_2$. In the first model the requirement can be modelled by

$$\begin{aligned}
&B_{31} = 0 \land \\
&B_{32} \leq B_{21} \land \\
&B_{33} \leq B_{21} + B_{22} \land \\
&B_{34} \leq B_{21} + B_{22} + B_{23} \land \\
&B_{24} = 0.
\end{aligned}$$

Although this formulation is equivalent in terms of information propagation to $W_3 > W_2$, it is considerably more complex, which means there is more chance of an error in specifying it, and larger, meaning that there may be more overhead in propagating the information. It is much more difficult to see how the requirement could be added to the third model.

As we have seen, different models may behave very differently in terms of the amount of information they can infer and propagate. In addition, some constraints may be very difficult to express in one model, but easy in another. For these reasons it may be worthwhile to combine different models. Combining models is a particular example of adding solver redundant constraints as discussed in Section 7.5. The constraints in the second model are redundant, but they may give better solver behaviour.

In general, we can combine models by using relations which the original problem dictates must hold between the variables in each model. Given that the variables specify an answer to the problem, the values the variables take in one model will correspond to values the variables take in another model. For instance, in the different models for the assignment problem, B_{11} is 1 precisely when W_1 is 1, and W_3 is 2 precisely when T_2 is 3.

Because of the importance of combining models, many CLP systems provide complex primitive constraints which facilitate this process. We will assume that there is a primitive constraint iff($var_1, val_1, var_2, val_2$) which holds if $var_1 \equiv val_1$ implies that $var_2 = val_2$ and $var_2 \equiv val_2$ implies that $var_1 = var_2$.

For example, `iff(W3, 2, T2, 3)` constrains W_3 to be 2 exactly when T_2 is 3. Thus if T_2 becomes set to 3 then this constraint will set W_3 to 2. Conversely, suppose the domain of W_3 is reduced to [3..4], then since $W_3 \neq 2$, the constraint $T_2 \neq 3$ can be added.

Example 8.8

A goal which combines the second and third model is

```
[W1,W2,W3,W4] :: [1..4],
alldifferent([W1,W2,W3,W4]),
element(W1,[7,1,3,4],WP1),
element(W2,[8,2,5,1],WP2),
element(W3,[4,3,7,2],WP3),
element(W4,[3,1,6,3],WP4),
P = WP1 + WP2 + WP3 + WP4,
[T1,T2,T3,T4] :: [1..4],
alldifferent([T1,T2,T3,T4]),
element(T1,[7,8,4,3],TP1),
element(T2,[1,2,3,1],TP2),
element(T3,[3,5,7,6],TP3),
element(T4,[4,1,2,3],TP4),
P = TP1 + TP2 + TP3 + TP4,
P ≥ 19,
iff(W1,1,T1,1), iff(W1,2,T2,1), iff(W1,3,T3,1), iff(W1,4,T4,1),
iff(W2,1,T1,2), iff(W2,2,T2,2), iff(W2,3,T3,2), iff(W2,4,T4,2),
iff(W3,1,T1,3), iff(W3,2,T2,3), iff(W3,3,T3,3), iff(W3,4,T4,3),
iff(W4,1,T1,4), iff(W4,2,T2,4), iff(W4,3,T3,4), iff(W4,4,T4,4),
labeling([W1,W2,W3,W4]).
```

Note that we can choose to label either the variables of the second or those of the third model. It is unnecessary to label both because, if all of the W_i variables are fixed, we are guaranteed that all of the T_j variables are fixed by the `iff` constraints, and vice versa.

For this model only 5 choices are required to find all solutions to the problem. However, since there are 39 primitive constraints, we might expect execution to take longer overall because of the extra work in the propagation solver.

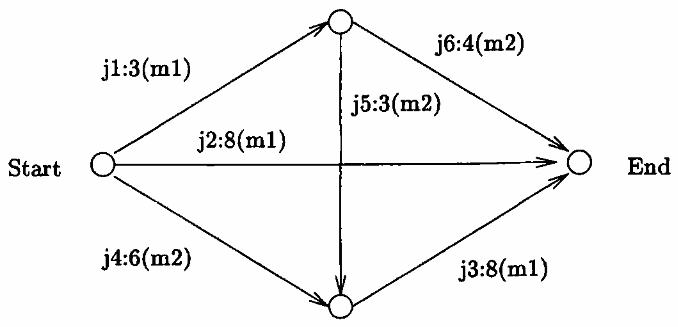

Figure 8.3 Tasks and precedences for scheduling problem.

8.5 An Extended Example: Scheduling

Combinatorial search problems such as those in the examples above are useful for illustrating the way in which a *CLP*(*FD*) system works but they are not particularly realistic. We now investigate a more standard commercial problem, that of scheduling. In a scheduling problem we are given a list of tasks, each of which must be performed on one of a selection of machines, and a list of precedences between the tasks indicating which tasks must be completed before a particular task can begin. The aim is to find a schedule, that is to say, a start time for each task, so that the precedences are maintained and no more than one task is being performed on any machine at any given moment. This problem is most commonly couched as an optimization problem in which the aim is to minimize the overall time for completion of all of the tasks. However, we will first consider the simpler problem of determining whether it is possible to schedule the tasks given an upper bound on the total time available.

We shall model the scheduling problem using the following data structure. A task is a record of the form

$$task(name, duration, [names], machine)$$

where *name* is a constant identifying the task, *duration* is an integer detailing the number of minutes the task requires, the third argument is a list of task names which must be completed before this task can begin and *machine* is the machine on which the task must be performed. A problem description is simply a list of tasks.

Consider the specific problem illustrated in Figure 8.3 where tasks are represented by arcs in the graph. The label of an arc is of the form *name* : *duration*(*machine*). Thus, $j3 : 8(m1)$ means that job $j3$ takes 8 minutes to complete and requires machine $m1$. The precedence constraints are given by "no task can start before all tasks ending at its start point are completed." This problem instance is represented by the following problem description, which we shall abbreviate to *problem*:

```
[task(j1, 3, [], m1), task(j2, 8, [], m1), task(j3, 8, [j4,j5], m1),
 task(j4, 6, [], m2), task(j5, 3, [j1], m2), task(j6, 4, [j1], m2)]
```

A relatively simple program to model the scheduling problem is as follows:

```
 schedule(Data, End, Joblist) :-
     makejoblist(Data, Joblist, End),
     precedences(Data, Joblist),
     machines(Data, Joblist),
     labeltasks(Joblist).

makejoblist([],[],_).
makejoblist([task(N,D,_,_)|Ts], [job(N,D,TS)|Js], End) :-
     TS :: [0..1000],
     TS + D ≤ End,
     makejoblist(Ts, Js, End).

getjob(JL, N, D, TS) :- once(member(job(N,D,TS), JL)).

precedences([],_).
precedences([task(N,_,Pre,_)|Ts], Joblist) :-
     getjob(Joblist, N, _, TS),
     prectask(Pre, TS, Joblist),
     precedences(Ts, Joblist).

prectask([], _, _).
prectask([Name|Names], PostStart, Joblist) :-
     getjob(Joblist, Name, D, TS),
     TS + D ≤ PostStart,
     prectask(Names, PostStart, Joblist).

machines([], _).
machines([task(N,_,_,M)|Ts], Joblist) :-
     getjob(Joblist, N, D, TS),
     machtask(Ts, M, D, TS, Joblist),
     machines(Ts, Joblist).

machtask([], _, _, _, _).
machtask([task(SN,_,_,M0)|Ts], M, D, TS, Joblist) :-
     (M = M0 ->
          getjob(Joblist, SN, SD, STS),
          exclude(D, TS, SD, STS)
     ; true ),
     machtask(Ts, M, D, TS, Joblist).

exclude(_D, TS, SD, STS) :- STS + SD ≤ TS.
exclude(D, TS, _SD, STS) :- TS + D ≤ STS.
```

```
labeltasks([]).
labeltasks([job(_,_,TS)|Js]) :-
    indomain(TS),
    labeltasks(Js).
```

Execution of the program proceeds as follows. First the predicate `makejoblist` is called. This builds a list that holds the task names with their duration and a variable which represents their start time. The start times are each constrained so that the task will begin after time zero and end before the time *End*.

The predicate `getjobs` is used to look up the joblist constructed by `makejoblist`. Given a task name it will find the corresponding duration and start time. The joblist is an association list between names and start times. Since a job only appears once in the joblist there is only ever one solution. Thus it is has been programmed using `once`.

The predicate `precedences` builds the precedence constraints for each task. The predicate `prectask` makes sure that the start time for a given task is after the end of each of the tasks in its precedence list.

The predicate `machines` sets up the exclusion constraints between jobs on the same machine. For each task on machine M, it finds every task later in the problem description that is also on machine M and makes sure (using `exclude`) that they are not performed simultaneously.

Finally, the call `labeltasks` labels the start time variables, ensuring that every start time takes a fixed value.

Executing the program on the test data *problem* using the goal

`End = 20, schedule(`*problem*`, End, Joblist)`

creates many constraints, most of which deal with the data structures. The result of `makejoblist` is to equate *Joblist* to a data structure of the form

[job(j1, 3, TS_1), job(j2, 8, TS_2), job(j3, 8, TS_3), job(j4, 6, TS_4), job(j5, 3, TS_5), job(j6, 4, TS_6)]

where the variables are given initial domain [0..1000] and constrained by:

$$\begin{array}{llll} 0 \le TS_1, & TS_1 + 3 \le 20, & 0 \le TS_2, & TS_2 + 8 \le 20, \\ 0 \le TS_3, & TS_3 + 8 \le 20, & 0 \le TS_4, & TS_4 + 6 \le 20, \\ 0 \le TS_5, & TS_5 + 3 \le 20, & 0 \le TS_6, & TS_6 + 4 \le 20. \end{array}$$

At this point the propagation solver has computed the domain

$$\begin{array}{lll} D(TS_1) = [0..17], & D(TS_2) = [0..12], & D(TS_3) = [0..12], \\ D(TS_4) = [0..14], & D(TS_5) = [0..17], & D(TS_6) = [0..16]. \end{array}$$

The effect of the user-defined constraint `precedences` is to add the constraints

$$TS_1 + 3 \le TS_5, \quad TS_1 + 3 \le TS_6, \quad TS_4 + 6 \le TS_3, \quad TS_5 + 3 \le TS_3.$$

After these are added propagation gives rise to the domain

$$D(TS_1) = [0..6], \quad D(TS_2) = [0..12], \quad D(TS_3) = [6..12],$$
$$D(TS_4) = [0..6], \quad D(TS_5) = [3..9], \quad D(TS_6) = [3..16].$$

The effect of the predicate `machines` is to build choices of constraints that correspond to the requirement that no two tasks are executed on the same machine at the same time. The choices are

$$\begin{aligned}
&TS_2 + 8 \leq TS_1 \text{ or } TS_1 + 3 \leq TS_2,\\
&TS_3 + 8 \leq TS_1 \text{ or } TS_1 + 3 \leq TS_3,\\
&TS_3 + 8 \leq TS_2 \text{ or } TS_2 + 8 \leq TS_3,\\
&TS_5 + 3 \leq TS_4 \text{ or } TS_4 + 6 \leq TS_5,\\
&TS_6 + 4 \leq TS_4 \text{ or } TS_4 + 6 \leq TS_6,\\
&TS_6 + 4 \leq TS_5 \text{ or } TS_5 + 3 \leq TS_6.
\end{aligned}$$

The first sequence of choices which leads to an answer is to choose the second choice in each pair, except for the fourth pair. This sequence of choices results in the propagation solver finding the domain:

$$D(TS_1) = [0..0], \quad D(TS_2) = [3..4], \quad D(TS_3) = [12..12],$$
$$D(TS_4) = [6..6], \quad D(TS_5) = [3..3], \quad D(TS_6) = [12..16].$$

The `labeltasks` predicate chooses the first element of each domain obtaining the final answer

$$D(TS_1) = [0..0], \quad D(TS_2) = [3..3], \quad D(TS_3) = [12..12],$$
$$D(TS_4) = [6..6], \quad D(TS_5) = [3..3], \quad D(TS_6) = [12..12].$$

The scheduling program again illustrates the typical form of a constraint satisfaction problem expressed in CLP using the constrain and generate methodology. First, the constraints of the problem are added to the store and, then, a labelling method is used to ensure a definite solution is found. The order in which constraints are added is important since it may affect efficiency. Since the basic time constraints and precedence constraints are deterministic they are added first. The machine exclusion constraints are not deterministic since these are modelled by choosing to place one task before or after another task on the same machine. This choice is made during execution, so, if a wrong choice is made, the other will be tried. Following the guidelines from Chapter 7 we, therefore, place the machine exclusion constraints after the basic time constraints and precedence constraints. This ensures that choices are made as late as possible, reducing both repeated computation and the size of the derivation tree.

For the scheduling problem, ideally, we wish to find a schedule with minimum total time. This is simply the answer to the goal

```
End :: [0..1000], schedule(problem, End, Joblist), indomain(End).
```

which minimizes the value of *End*. Thus the following goal will compute the schedule with minimum total time.

```
End :: [0..1000],
minimize((schedule(problem, End, Joblist), indomain(End)), End).
```

Executing this goal, a solution is found with $End = 19$ and

$$TS_1 = 0 \land TS_2 = 3 \land TS_3 = 11 \land TS_4 = 0 \land TS_5 = 6 \land TS_6 = 9.$$

This is a better solution than that found when solving the satisfiability problem since that solution required 20 minutes to complete the tasks.

8.5.1 Redundant Information

As is it stands, the simple scheduling program given above is not useful for solving larger scheduling problems because the search space rapidly becomes too large. In Section 7.5 we saw that adding redundant constraints to a program can sometimes reduce its search space. Here we investigate how the addition of answer redundant constraint information to the scheduling program can help reduce its search space.

The large search space arises because of the mutual exclusion constraints for machine usage. The problem is that the mutual exclusion information is represented in different rules, and so is only available once a rule has been chosen. If we can add constraints that are redundant with respect to all of the choices, we can reduce the search space by making information available before a choice is made.

Let us take a concrete example. Suppose that, for some task t_0, all of the tasks $[t_1, \ldots, t_n]$ have to be performed before t_0, and each of them must be performed on the same machine. The tasks $t_1, \ldots, t_n$ must take at least $D_1 + \cdots + D_n$ time to complete where D_i is the duration of task t_i since they must all be completed on the same machine. Furthermore, they cannot start before the minimum of their start times $TS_1, \ldots, TS_n$. Thus, t_0 cannot begin before the sum of the minimum start time of $t_1, \ldots, t_n$ together with the total duration of $D_1 + \cdots + D_n$. That is,

$$TS_0 \geq \text{minimum}(\{TS_1, \ldots, TS_n\}) + D_1 + \cdots + D_n.$$

This is an answer redundant constraint since the information is implicit in the current representation of the problem and so any answer to the problem must satisfy this constraint. However, if it is added, using a deterministic user-defined constraint, it may reduce the search space.

For example, note task `j3` can begin only after `j4` and `j5` (and hence `j1`) have finished. Since both these tasks require `m2` they cannot both be completed before 9 minutes, the sum of their durations after the earliest one begins. Thus we can add the information $TS_3 \geq \text{minimum}(\{TS_4, TS_5\}) + 9$, which can reduce the domain of TS_3 before adding any machine exclusion constraints.

```
schedule(Data, End, Machinelist, Joblist) :-
     makejoblist(Data, Joblist, End),
     precedences(Data, Joblist),
     makeemptylists(Machinelist, Emptys),
     extrainfo(Data,Joblist,Machinelist,Emptys,Joblist),
     machines(Data, Joblist),
     labeltasks(Joblist).

makeemptylists([],[]).
makeemptylists([_|L],[[]|R]) :- makeemptylists(L,R).

extrainfo(_, [], _, _, _).
extrainfo(Data, [job(N,_,TS)|Js], Machinelist, Emptys, Joblist) :-
     predecessors(N, Data, [], Pre),
     splittasks(Pre, Data, Machinelist, Emptys, Split),
     list_redundant(Split, Joblist, TS),
     extrainfo(Data, Js, Machinelist, Emptys, Joblist).

splittasks([], _, _, Split, Split).
splittasks([N|Ns], Data, Machines, Split0, Split) :-
     gettask(Data, N, task(_,_,_,M)),
     insert(Machines, M, N, Split0, Split1),
     splittasks(Ns, Data, Machines, Split1, Split).

insert([M|_], M, T, [L|Ls], [[T|L]|Ls]).
insert([M0|Ms], M, T, [L|Ls0], [L|Ls]) :-
     M0 ≠ M,
     insert(Ms, M, T, Ls0, Ls).

list_redundant([], _, _).
list_redundant([L|Ls], Joblist, TS) :-
     redundant(L, Joblist, TS, 0, []),
     list_redundant(Ls, Joblist, TS).

redundant([], _, TS, TotalD, ListTS) :-
     TS ≥ minimum(ListTS) + TotalD.
redundant([N|Ns], Joblist, TS, TotalD0, ListTS) :-
     getjob(Joblist, N, DN, TSN),
     TotalD = TotalD0 + DN,
     redundant(Ns, Joblist, TS, TotalD, [TSN|ListTS]).

gettask(Data, N, T) :-
     T = task(N, _, _, _),
     once(member(T, Data)).

lookup(Data, N, P) :- member(task(N,_,P,_), Data).
```

The above program calculates the list of predecessors for each task using the definition of predecessors from Chapter 5 (where lookup is re-defined as shown in the program above). Then splittasks splits the list of predecessors into a list of lists, where each list corresponds to the tasks performed on one machine. Finally, `list_redundant` builds the redundant constraint for each such list using `redundant`.

Conversely, we could extend the original program to calculate the descendants of each task, and ensure that each task is started early enough so that all of its descendants on the same machine can be completed before *End*. If we use both methods we can expect to reduce the search space more substantially since they will interact with each other.

It may be useful to compute and add this redundant information more than once during the computation. In particular, after we have made a number of choices about which tasks on the same machine are before or after others, we increase the total number of tasks that must occur before a particular task. Thus, if we repeat the calculation we may get stronger constraints.

Suppose we have chosen that `j5` should be executed after `j6`. This will cause us to add the constraint $TS_6 + 4 \leq TS_5$. We could now update the immediate predecessor list for `j5` to include `j6` and recalculate the redundant constraints. Now tasks `j1, j4, j5, j6` must all be completed before `j3` can begin. Since `j4`, `j5` and `j6` are all jobs for machine `m2` with a total duration of 13, we know that `j3` cannot begin before time 13 after the earliest of start time for one of `j4, j5, j6`. Therefore we can add the constraint

$$TS_3 \geq \text{minimum}(\{TS_4, TS_5, TS_6\}) + 13.$$

By maintaining lists of predecessors and updating them whenever we make a choice about the order of tasks on the same machine, we can add more information after each choice is made. For large scheduling problems this may drastically reduce the search space.

There is, however, a tradeoff between the extra computation required to generate the redundant constraints and the pruning of the search space these extra constraints produce. For some data, the redundant constraints may not reduce the search space at all. Indeed, this is the case for the data for *problem*. On the other hand, for some scheduling problems computing and adding the redundant constraints may mean an answer is found in minutes instead of requiring many hours.

One of the more difficult tasks of the constraint programmer is to understand the tradeoff between the cost of adding redundant constraints and the reduction in search space they cause. Understanding, usually gained through experimentation, allows the programmer to weigh the relative benefits and, so, to obtain an efficient solution to the particular problem they are interested in.

8.5.2 Using `cumulative` Constraints

Another way to model the scheduling problem is to use the complex primitive constraint `cumulative`. If available, this constraint provides a way to model the machine exclusion constraints without requiring multiple rules and, hence, choice. The stronger consistency information it can provide will reduce the search required for a solution. We can modify the above program to use the stronger, but more complex, cumulative constraint, as follows:

```
schedule(Data, End, Machinelist, Joblist) :-
     makejoblist(Data, Joblist, End),
     precedences(Data, Joblist),
     makeemptylists(Machinelist, Emptys),
     extrainfo(Data,Joblist,Machinelist,Emptys,Joblist),
     cumulative_machines(Data,Joblist,Machinelist,Emptys),
     labeltasks(Joblist).

cumulative_machines(Data, Joblist, Machinelist, Emptys) :-
     datatolist(Data, Names),
     splittasks(Names, Data, Machinelist, Emptys, Split),
     makecumulatives(Split, Joblist).

datatolist([],[]).
datatolist([task(N,_,_,_)|Ts],[N|Ns]) :-
     datatolist(Ts,Ns).

makecumulatives([], _).
makecumulatives([L|Ls], Joblist) :-
     makecumulative(L, Joblist, [], [], []),
     makecumulatives(Ls, Joblist).

makecumulative([], _, Ts, Ds, Rs) :-
     cumulative(Ts, Ds, Rs, 1).
makecumulative([N|Ns], Joblist, Ts, Ds, Rs) :-
     getjob(Joblist, N, DN, TSN),
     makecumulative(Ns, Joblist, [TSN|Ts], [DN|Ds], [1|Rs]).
```

The predicate `cumulative_machines` works by first building a list of all task names, *Names*, and then splitting the tasks into a list of lists of names, *Split*, one for each machine, using `splittasks`. Then the lists in *Split* are traversed one by one using `makecumulative` which collects a list of start times, *Ts*, durations, *Ds*, and resources, *Rs*, for the `cumulative` constraint which is added when the end of the list is reached. Note that the value of the available and required resource is always one since each task requires sole use of the single machine.

This formulation gains from the greater consistency reasoning available for `cumulative` and delaying the choice of ordering until labelling. For example, for the data of our example *problem* two cumulative constraints are set up

$$(m1) \quad \texttt{cumulative}([TS_1, TS_2, TS_3], [3,8,8], [1,1,1], 1)$$
$$(m2) \quad \texttt{cumulative}([TS_4, TS_5, TS_6], [6,3,4], [1,1,1], 1)$$

Suppose we have labelled TS_4 as 3. The second cumulative constraint will propagate the information that the range of TS_6 should be [9..16] since it cannot fit elsewhere (see Figure 8.4).

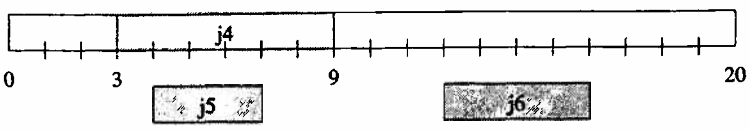

Figure 8.4 Propagation from a cumulative constraint.

After applying the cumulative constraints the resulting domain for the example problem is

$$D(TS_1) = [0..1],\quad D(TS_2) = [3..4],\quad D(TS_3) = [11..12],$$
$$D(TS_4) = [0..6],\quad D(TS_5) = [3..9],\quad D(TS_6) = [6..16].$$

Propagation using the cumulative constraints has solved most of the problem. The relative order of the tasks j1, j2 and j3 is already fixed, while the remaining domains are significantly reduced.

So far we have not discussed different labelling strategies for the scheduling problem. We have simply used the standard labelling predicate. The initial program only creates constraints of the form

$$TS_i + d \leq TS_j,\quad d \leq TS_i \text{ or } \quad TS_i \leq d$$

where d is a constant. The bounds propagation solver is actually complete for this class of constraints, so, by the time the initial program reaches the labeltasks predicate, setting each of the variables to its current lower bound must be a solution to the problem. The default labelling quickly finds this solution.

When we replace the machine exclusion constraints by cumulative constraints, the labelling method can have an impact on the amount of search required to find a solution. We should therefore consider writing a problem specific labelling strategy.

One possibility is to label the tasks in order of execution. That is, find the task with the earliest possible starting time, and attempt to set its start time to that time. If this fails, we remove the earliest starting time from the possible starting time of the chosen task and continue, looking for the task which now has the earliest possible start time. The reasoning behind this strategy is that placing the earliest task will increase the lower bounds on the other tasks. If it cannot lead to a solution we should discover this quickly. The following labelling predicate implements this strategy.

```
labeltasks([]).
labeltasks(Js) :-
     gettimes(Js, Ts),
     label_earliest(Ts).

gettimes([],[]).
gettimes([job(_,_,TS)|Js], [TS|Ts]) :- gettimes(Js, Ts).
```

```
label_earliest([]).
label_earliest([TS0|Ts]) :-
    mindomain(TS0, M0),
    find_min_start(Ts, TS0, M0, TS, M, RTs),
    rest_earliest(TS, M, [TS0|Ts], RTs).

rest_earliest(TS, M, _Ts, RTs) :-
    TS = M,
    label_earliest(RTs).
rest_earliest(TS, M, Ts, _RTs) :-
    TS ≠ M,
    label_earliest(Ts).

find_min_start([], TS, M, TS, M, []).
find_min_start([TS1|Ts], TS0, M0, TS, M, RTs) :-
    mindomain(TS1, M1),
    (M1 < M0 ->
        RTs = [TS0|RTs1],
        find_min_start(Ts, TS1, M1, TS, M, RTs1)
    ;
        RTs = [TS1|RTs1],
        find_min_start(Ts, TS0, M0, TS, M, RTs1)
    ).
```

The predicate `labeltasks` builds the list of start times, Ts, using `gettimes`. The predicate `label_earliest` does the real work. It searches for the start time with the smallest minimum by calling `find_min_start`. The call

```
    find_min_start(Ts, TS0, M0, TS, M, RTs)
```

finds the variable TS with earliest start time M from the variables in $\{TS0\} \cup Ts$, while RTs is set to the remaining variables in this set. The variable $TS0$ is the current selection and $M0$ is the earliest time it can start. The call `rest_earliest` tries both choices of setting TS to M and labelling the remaining variables or removing M from the domain of TS and retrying the heuristic.

We could break ties in tasks with the same minimum by choosing the most tightly constrained task (an application of the first fail principle).

8.6 (*) Arc Consistency

In our description of our idealized system $CLP(FD)$ we have assumed that it uses a bounds propagation solver. However, as we saw in Section 3.5 arc consistency allows more values to be eliminated from the domains of variables occurring in binary constraints than does bounds consistency. For this reason many CLP systems also use arc consistency.

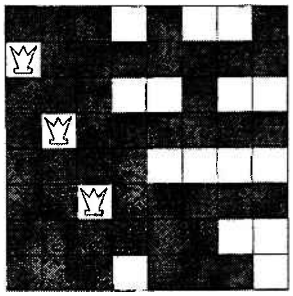

Figure 8.5 Example 8-queens partial solution.

Disequality constraints are a particular type of binary constraint on which arc consistency performs significantly better than bounds consistency. For example, consider the *8-queens* problem. The initial domain of each variable $R_1, \ldots, R_8$ is $[1..8]$. Suppose that we are part way through labelling the variables and we have found the partial solution $\{R_1 \mapsto 2, R_2 \mapsto 4, R_3 \mapsto 6\}$.

A constraint solver based on bounds consistency determines the domains of the remaining variables (given the partial solution) to be

$$D(R_4) = [1..8], \quad D(R_5) = [3..5], \quad D(R_6) = [1..5],$$
$$D(R_7) = [1..7], \quad D(R_8) = [3..8].$$

An arc consistency based solver for the same constraints and partial solution will obtain the domain,

$$D(R_4) = \{1,3,8\}, \qquad D(R_5) = \{3,5\}, \qquad D(R_6) = \{1,5\},$$
$$D(R_7) = \{1,3,5,7\}, \quad D(R_8) = \{3,5,7,8\}.$$

The arc consistent domain values are shown as unshaded squares in Figure 8.5.

The reason that arc consistency is so much more effective at pruning the domains is that all of the constraints are of the form $C_i \neq C_j + d$ where d is a constant. Bounds propagation does not perform well on these constraints since a bounds propagation solver can only makes use of disequalities when they change a lower or upper bound. However, the constraints are all binary and therefore simple to handle using arc consistency. Because of the importance of disequations, most CLP systems use arc consistency when solving disequations. Note when using labelling methods that make use of the size of the domain, such as first fail labelling, arc consistency also gives much more accurate information about the possible values a variable can take.

Another advantage of arc consistency over bounds consistency is that arc consistency works well with arbitrary binary constraints while bounds consistency works only with arithmetic constraints. Some CLP systems allow the programmer to define their own binary relations and to direct the system to employ arc consistency to solve these relations. The programmer defines the binary relation as a user-defined constraint with multiple facts. The call to the relation in the goal is annotated with a `consistent` suffix to indicate that it should be evaluated by using arc consistency rather than the standard CLP evaluation mechanism. The `consistent` suffix can also be used with unary constraints, in which case node consistency is used. The programmer may even use it with relations that have greater arity than two. In this case the system will use hyper-arc consistency.

For example, recall the program to solve an instance of the old-fashioned marriage problem given in Section 8.1. The `likes` relation is still described by the simple program consisting of seven facts. However, instead of backtracking through the different rules we can use arc consistency to evaluate the `likes` relation. The following goal does this:

```
[N,M,E] :: [kim, peter,bernd], N ≠ M, N ≠ E, M ≠ E,
likes(N, nicole) consistent,
likes(M, maria) consistent,
likes(E, erika) consistent,
labeling([N,M,E]).
```

Note that we now need to call a labelling predicate since evaluation of `likes` will no longer ensure that N, M and E are assigned fixed values.

Using arc consistency to solve the disequalities and `likes` constraints gives rise to the domain

$$D(N) = \{kim, bernd\}, \quad D(M) = \{kim, bernd, peter\}, \quad D(E) = \{kim, bernd\}.$$

Evaluation of the labelling predicate will first set N to *kim*. Arc consistency will reduce the domain of the remaining variables to

$$D(M) = \{peter\}, \quad D(E) = \{bernd\},$$

so no other choices are required to find the first answer. If the system is asked for another answer, backtracking will immediately set N to *bernd* and no other choices will be required to find the second answer. This contrasts with the original goal in which many more choices are required to find the two answers.

Providing arc consistency removes the need for the complex primitive constraint `element`. The constraint `element(I,[`a_1`, ... ,`a_n`],V)` is essentially equivalent to the constraint `array(I,V) consistent` where the relation `array` is defined by

```
array(1,a₁).
array(2,a₂).
   ⋮
array(n,aₙ).
```

For instance, the constraint `element(W1,[7,1,3,4],WP1)` from the program for the second model for the simple assignment problem (Example 8.6) can be replaced by the literal, `profit_1(W1,WP1) consistent`, where `profit_1` is defined by

```
profit_1(1,7).
profit_1(2,1).
profit_1(3,3).
profit_1(4,4).
```

8.7 (*) Reified Constraints

Using the facilities we have discussed so far, efficient modelling of certain complex combinations of constraints can be difficult. Suppose we have 5 primitive constraints $\{c_1, c_2, c_3, c_4, c_5\}$ over the variables X and Y of which we want between two and three to hold in our solution. We can model this (abstractly) using a program of the form

```
comb(X, Y)  :-  c₁, c₂, ¬c₃, ¬c₄.
comb(X, Y)  :-  c₁, c₂, ¬c₃, ¬c₅.
comb(X, Y)  :-  c₁, c₂, ¬c₄, ¬c₅.
comb(X, Y)  :-  c₁, c₃, ¬c₂, ¬c₄.
comb(X, Y)  :-  c₁, c₃, ¬c₂, ¬c₅.
                ⋮
```

This program is large and difficult to follow. A better way of modelling this requirement is to use "reified" constraints.

A *reified constraint* is of form $c \Leftrightarrow B$. It consists of a constraint c together with an attached Boolean variable B. The reified constraint $c \Leftrightarrow B$ does not require the constraint c to hold but only that the relationship between the implication of the constraint and the value of the Boolean variable B holds. The intention is that $B = 1$ when c holds in the current store and $B = 0$ if $\neg c$ holds.

Using reified constraints we can easily model the above combination by

```
c₁ ⇔ B1, c₂ ⇔ B2, c₃ ⇔ B3, c₄ ⇔ B4, c₅ ⇔ B5,
B = B1 + B2 + B3 + B4 + B5,
2 ≤ B, B ≤ 3.
```

Reified constraints are somewhat complex to implement since they require the solver to determine whether or not a constraint is implied by the current constraint store and whether or not its negation is implied. However, because of the usefulness of reified constraints, some finite domain constraint solvers provide them. Usually they are restricted so that only certain kinds of primitive constraints can be reified.

The main use of reified constraints is to program complex relationships between constraints without using choice.

For example, if we want to model the requirement that $X = 1$ or $X = 2$ but do not want to make a choice then the literal `either(X,1,2)` will do this where the predicate `either` is defined by

```
either(X,X1,X2) :-
      [B1,B2] :: [0..1],
      (X=X1 ⇔ B1), (X=X2 ⇔ B2),
      1 ≤ B1+B2.
```

Consider a call to `either(A,C,E)`. If at some later stage

$$D(A) = \{1,2\}, \quad D(C) = \{3,4\}, \quad D(E) = \{2,3\},$$

then since $A = C$ must be false, $A = E$ is true and the resulting domain is

$$D(A) = \{2\}, \quad D(C) = \{3,4\}, \quad D(E) = \{2\}.$$

One interesting application of reified constraints is to define the built-in `iff` which was introduced in Section 8.4 to facilitate combining different models of the same problem. We can define `iff` by

```
iff(X1,V1,X2,V2) :-
      B :: [0..1],
      (X1=V1 ⇔ B),
      (X2=V2 ⇔ B).
```

The literal `iff(X1,V1,X2,V2)` will hold if $X1$ has the value $V1$ exactly when $X2$ has the value $V2$.

8.8 Summary

Finite domain problems make up the largest class of real life problems tackled by CLP systems. Because of the weak and incomplete solving methods used for such problems, the constraint programmer often has to explicitly use a labelling phase in which a backtracking search tries different values for the variables. Because of this, the usual form of a CLP program solving a finite domain problem reflects the constrain and generate methodology. It consists of constraints which model the problem followed by a labelling phase which generates an answer to the problem.

Since labelling is expensive, the programmer needs to be aware of techniques for reducing the search space and of how constraints are handled by the particular solver being used. In this chapter we have seen a variety of techniques for reducing the search space.

One of the features of the CLP approach to solving constraints over finite domains is that the programmer can specialize the labelling method to take advantage of the structure of the particular problem. We have investigated different strategies for choosing which variable to label and in which order to explore the values in a variable's domain.

Another technique for reducing the search space is to use specialized complex primitive constraints. Often, these allow more concise modelling of a problem and may also improve the efficiency of a model since a complex constraint usually provides better propagation behaviour than does an equivalent conjunction of simpler primitive constraints.

A third technique for reducing the search space is to add redundant constraints. Sometimes it is useful to add solver redundant constraints. These are constraints which are redundant with respect to the constraints in the solver but which improve the propagation behaviour of the solver. It is also useful to add answer redundant constraints. These explicitly add information which is implicitly represented in the solutions to the program.

Another more fundamental way of changing the search space is by choosing a different model for the problem, that is to say, finding a different way of expressing the problem in terms of constraints. Factors influencing the efficiency and flexibility of a model are:

- ease of expression;
- strength of the constraint solver on the constraints;
- number of constraints;
- number of variables.

All of these factors need to be considered. For some problems adding redundant constraints and extra variables may be the best approach to solving the problem, because they significantly reduce the search space. For other problems which require few choices or for which the constraints are well-handled by the solver, the fewer the constraints the better.

Given the somewhat complicated runtime behaviour of $CLP(FD)$ systems, it is often difficult for even the most experienced constraint programmer to predict how a particular technique will affect the execution speed. For this reason, it is usually important to try a variety of approaches and to compare them empirically.

8.9 Practical Exercises

Most of the programs given in this chapter will execute almost unchanged under ECLiPSe, with the compiler directive `:- use_module(library(fd))`. How to obtain ECLiPSe is discussed in the Practical Exercises section of Chapter 3. Some translation of symbols is required to express constraint relations: replace $=$ with `#=`, $\neq$ with `##`, $>$ with `#>`, $<$ with `#<`, $\geq$ with `#>=` and $\leq$ with `#<=`. The `iff` constraint described in Section 8.4 is not available in ECLiPSe. Similarly the `minimum` function and `cumulative` constraint used in Sections 8.5.1 and 8.5.2 are not supported by ECLiPSe. All can be programmed using the extensible constraint library of ECLiPSe. The built-ins `dom`, `mindomain`, `maxdomain` and `indomain` are provided in ECLiPSe. ECLiPSe also provides for the declaration of (hyper-) arc consistency using the suffix `infers most` rather than `consistent` that we have used in the text. The library `propia` must be loaded. For more information see [104].

Similarly, most of the programs given in this chapter will run under SICStus Prolog with the compiler directive `:- use_module(library(clpfd))`. How to obtain SICStus Prolog is discussed in the Practical Exercises section of Chapter 1. Some translation of symbols is required to express constraint relations: replace $=$ with `#=`, $\neq$ with `#\=`, $>$ with `#>`, $<$ with `#<`, $\geq$ with `#>=` and $\leq$ with `#=<`. The labelling predicate in SICStus takes two arguments, the first of which is a list of options. Ordinary labelling is provided by `labeling([], [`*Vars*`])`. Different options can be used to label the variable with the smallest domain first, to use domain splitting or even to use a labelling that minimizes a cost function. See the SICStus Prolog manual for details. The `iff` constraint described in Section 8.4 is not available in SICStus Prolog. The `minimum` function which takes a list of arguments is not available, though a binary minimum function `min` is. Both can be programmed using the extensible constraint library of SICStus. The complex constraint `alldifferent` is available in two versions named `all_different` and `all_distinct`, which implement the consistency methods of Figure 3.15 and [111] respectively. The built-ins `mindomain`, `maxdomain` have different names in SICStus, being called `fd_min` and `fd_max` respectively. The built-in `dom` is not available but can be programmed using the built-in `fd_dom`. Reified constraints are supported in the SICStus Prolog finite domain constraint solving library using the notation *constraint* `#<=>` *Boolean*.

Neither ECLiPSe nor SICStus provide `div`. However, the constraint `s = t div 2` can be replaced by `B :: [0,1], 2*s + B = t`.

P8.1. Write a program to three colour a map represented using a binary adjacency relation. Use it to colour the map in Figure 3.22.

P8.2. Write a program to solve the crypto-arithmetic problem $DONALD + GERALD = ROBERT$. Each letter represents a different digit.

P8.3. Write two programs based on different models, to find the maximum profit for the assignment problem assigning 4 workers to one of 6 tasks, with the profit matrix given below.

	p_1	p_2	p_3	p_4	p_5	p_6
w_1	7	4	3	6	5	4
w_2	3	2	3	2	3	1
w_3	7	5	6	4	4	2
w_4	4	2	1	3	3	1

Compare the number of choices made to find all solutions in which the total profit is greater than 16.

P8.4. A perfect number X is equal to the sum of the numbers less than X that divide it exactly. For example, $6 = 1 + 2 + 3$, so 6 is a perfect number. Write a $CLP(FD)$ program for `perfect(X)` that succeeds whenever X is a perfect number.

P8.5. Write a pizza ordering program in *CLP*(*FD*). Pizza slice types are *vegetarian, margherita, bolognese, special, tropical, garlic, capricciosa.* Your program should output a pizza order, in terms of large (8 slice) and medium (6 slice) pizzas given the list of attendees. For example,

```
    pizza([peter, thomas, roland, max], Q)
```

should return a list Q detailing the pizza order. Note that a pizza can be ordered with half of one type and half of another, and that all slices in the order must be eaten.
The preferences for various individuals are: Peter requires four or five pieces, one or two vegetarian, one bolognese, no special, and at least one garlic. Roland wants four or five pieces, no more than one of any type, except garlic. Thomas asks for three or four pieces, at least one vegetarian, and one bolognese, no special, margherita, capricciosa, and at most one garlic. David requires two or three pieces, no vegetarian or tropical, no more than two of any kind of slice. Tim requires three or four pieces, at least one bolognese, no garlic, vegetarian, tropical or capricciosa. Ashley wants exactly one garlic, one bolognese and one margherita. Sandy asks for no more than four pieces and only vegetarian, margherita or garlic will do.

P8.6. Consider the following logic puzzle. There are 5 houses, each of a different colour and inhabited by a person from a different country who has a different pet, drink and make of car.

(a) The English woman lives in the red house.

(b) The Spaniard owns the dog.

(c) Coffee is drunk in the green house.

(d) The Ukrainian drinks tea.

(e) The green house is immediately to the right of the ivory house.

(f) The BMW driver owns snails.

(g) The owner of the yellow house owns a Toyota.

(h) Milk is drunk in the middle house.

(i) The Norwegian lives in the first house on the left.

(j) The person who drives a Ford lives in the house next to the owner of the fox.

(k) The Toyota driver lives in the house next to the house where the horse is kept.

(l) The Honda owner drinks orange juice.

(m) The Japanese drives a Datsun.

(n) The Norwegian lives next to the blue house.

Write a *CLP*(*FD*) program to find who owns the zebra and who drinks water.

P8.7. Boolean variables can be treated as integers over the range [0..1]. Write predicates `not(X,Y)`, `and(X,Y,Z)`, `or(X,Y,Z)`, `if(X,Y,Z)` and `iff(X,Y,Z)` which

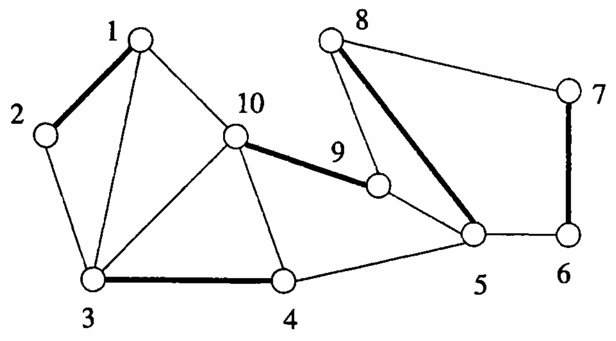

Figure 8.6 A graph and matching.

model the Boolean constraints $Y = \neg X$, $Z = X \& Y$, $Z = X \vee Y$, $Z = X \rightarrow Y$ and $Z = X \leftrightarrow Y$ respectively using integer constraints. For example, and can be defined by

```
and(X, Y, Z) :- Z ≤ X, Z ≤ Y, X + Y ≤ Z + 1.
```

Use these predicates to find a solution to the constraints of the fault analysis example in Section 1.5.1

P8.8. Give two different models for solving the matching problem for the graph shown in Figure 8.6. A matching is a set of edges from the graph so that every vertex is the endpoint of exactly one edge. In the figure, the thick edges give one matching for this graph. The first model should have variables representing edges and the second should have variables representing vertices.

P8.9. For N-queens not only is middle-out best for value ordering, but also for variable ordering. Thus, a better variable ordering strategy is to choose the variable with the smallest current domain, and in the case of a tie, to choose the variable closest to the center of the board. Write a labelling predicate which implements this strategy.

P8.10. Rewrite the N-queens program so that it makes use of the complex primitive constraint `alldifferent`. There should be three calls to `alldifferent`, one to ensure the queens are in different rows and two to ensure that they are in different diagonals.

P8.11. Often a problem is *symmetric*. Allowing symmetric answers to the same problem (answers which in some sense "encode" the same solution to the problem) increases the search space for the problem, and can mean considerable extra work. By recognising symmetric solutions and adding extra constraints to remove symmetric solutions, we can improve the search behaviour of a constraint program. As an example, consider the seesaw problem of Example 8.1. The two sides of the seesaw are identical. To avoid getting symmetric solutions such as

$$L = -4 \wedge F = 2 \wedge S = 5 \text{ and } L = 4 \wedge F = -2 \wedge S = -5$$

we can simply force one person to be on one side of the seesaw. For example, we can

force Liz to be on the negative side of the seesaw (or in the centre) by adding the constraint $L \leq 0$. The resulting goal not only avoids finding symmetric solutions, but also finds the first solution faster because of more information.
Symmetries of reflection are not the only kinds of symmetry. Another possibility is circular symmetries. Consider the following *dinner party problem*. Four couples have been invited to dinner: Peter and Maria, Kim and Nicole, Bernd and Erika, and Hugh and Chiara. Each guest must be assigned a seat at a circular table. This is a very traditional dinner party, so members of the same couple are not allowed to sit next to each other and no guest is allowed to sit next to a guest of the same sex. Write a program which will find a seating arrangement. Your program should try to avoid unnecessary symmetry.

P8.12. Extend the first program for simple scheduling so that it allows more than a single machine of each type. A description of a scheduling problem instance now also details the number of machines of each type and the task details which type of machine it needs.

P8.13. Write a program to do the same thing as described in the previous question but using the `cumulative` constraint.

P8.14. (*) Rewrite the program for the second model for the simple assignment problem (Example 8.6) so that all calls to `element` are replaced by calls to arc consistent binary relations.

P8.15. (*) Using the reification facilities of the finite domain library for SICStus Prolog write a program for a user-defined constraint `posneg(L, P, Q)` that constrains exactly P of the elements of the list L to be strictly positive integers and N of the elements to be strictly negative integers.
The goal `C ≤ -3, posneg([1, B, 2, -1, C], 2, 2)` should infer that $B = 0$, and the goal `C ≤ 1, posneg([1, B, 2, C], P, 0)` should infer that P is in the range [2..4], C is in [0..1] and that $B \geq 0$.

P8.16. (*) Reification of linear inequalities can be mimicked by translation. For example, the reified constraint $X \geq Y \Leftrightarrow B$ where the initial domains of X and Y are [0..100] is equivalent to the constraint:

```
B :: [0..1], X - Y ≥ 100 * B - 100, X - Y ≤ 101 * B - 1.
```

Verify the behaviour of the translation when B takes the values 0 or 1, or X and Y both take values from the set $\{0, 100\}$.
Give a single rule defining a user-defined constraint that is equivalent to `apart(X,Y,3)` using mimicked reification, given that the initial domains of X and Y are both [0..100]. Can you extend this to mimic `apart(X,Y,N)` given that the initial domain of N is also [0..100]?

8.10 Notes

Constraint logic programming over finite domains was first implemented in the language CHIP [4] developed at ECRC. An excellent description of constraint logic programming for finite domains is given in the book "Constraint Satisfaction and Logic Programming" [135] by Van Hentenryck using the language CHIP.

Since CHIP, implementations of propagation solvers have improved, and one approach to implementing propagation solvers is described in [41] and provided in the CLP system `clp(FD)`. The particular CLP(FD) language described in the chapter is roughly based on the ECLiPSe language's finite domain library. ECLiPSe is available from IC-PARC. See Chapter 3 for details. The latest version of CHIP is available from Cosytech.

The send-more-money and N-queens problems are classic constraint satisfaction problems. Task assignment and scheduling are also standard constraint satisfaction problems with considerable commercial application. The redundant constraints for job shop scheduling used in Section 8.5.1 are more general versions of constraints discussed by Van Hentenryck in [135] to solve a bridge scheduling problem. One of the original motivations for introducing `cumulative` constraints was, indeed, for representing resource constraints in scheduling problems.

The first fail principle was introduced in [60]. The use of a middle out search for the N-queens problem appears in an early CHIP demonstration program. Domain splitting also appears in early CHIP programs. It is, of course, closely related to the branch and bound technique for integer programming described in Chapter 3.

An excellent discussion of how to combine different problem modellings, and an example of the advantages that can be obtained, is given by Cheung *et al* [27].

Disjunctive scheduling problems are one area in which constraint logic programming can outperform even specialized programs written in a traditional language. Van Hentenryck [135] discusses a scheduling problem for building a five-segment bridge. Recently Caseau and Laburthe [23] showed how CLP techniques can be used to solve 10×10 job-shop scheduling problems efficiently. In [24] they also give an interesting discussion of cumulative scheduling with CLP.

Modelling disjunction or "or" without using choice has always been a concern of constraint programming, since it limits the size of the search tree. The *cardinality* constraint [136] was one of the first constraints used to model disjunctive combinations of constraints without using choice. Reified constraints arise from modelling techniques used in operations research (for example see [89]). Many constraint programming languages now support *reification* of (certain classes of) primitive constraint. For example the finite domain constraint solver in SICStus Prolog supports reified constraints and also provides special notation for combining constraints using reified disjunction and implication.