3 Finite Constraint Domains

In the previous two chapters we have looked at several constraint domains, including real arithmetic and trees. In this chapter, we investigate another important class of constraint domains: "finite constraint domains." These are constraint domains in which the possible values that a variable can take are restricted to a finite set. Boolean constraints and the blocks world are both examples of finite constraint domains. Another example of a finite constraint domain are integer constraints in which each variable x is constrained to lie within a finite range of integers.

Finite constraint domains are widely used in constraint programming. This is because they allow the programmer to naturally model constraint problems that involve choice, by letting the possible values of a variable correspond to different choices. Suppose we wish to colour a map of Australia as shown in Figure 3.1. It is made up of seven different states and territories[1] each of which may be coloured red, yellow or blue. Our colouring of the map should also satisfy the constraint that adjacent regions have different colours. This is an example of a finite domain problem, since the colours of each region come from the finite set $\{red, yellow, blue\}$.

In this chapter we will examine methods for solving constraints over finite constraint domains. Many real life problems, notably scheduling, routing and timetabling, are simple to express using finite constraint domains since, essentially, they involve choosing amongst a finite number of possibilities. Finding solutions to these problems is of great commercial importance to many businesses. For example, deciding how air crews should be allocated to aircraft flights is a crucial part of any airline business. A good solution reduces the expenses of the airline by minimizing the number of air crew needed and the number of flights on which the air crew must travel as passengers so as to be available at the source of their next flight.

Given the importance of finite constraint domains, it is not surprising that several research communities have developed methods for solving problems in finite constraint domains. Here we cover arc and node consistency techniques, developed by the artificial intelligence community to solve constraint satisfaction problems (a general type of finite constraint domain problem), bounds propagation techniques developed by the constraint programming community and integer programming techniques developed by the operations research community.

1. For the interested, non-Australian reader, these are: New South Wales, Northern Territory, Queensland, South Australia, Tasmania, Victoria and Western Australia.

Figure 3.1 Australian states.

3.1 Constraint Satisfaction Problems

In the artificial intelligence community, satisfaction of constraint problems over finite domains has been studied under the name "constraint satisfaction problems."

Definition 3.1
A *constraint satisfaction problem*, or *CSP* for short, consists of a constraint C over variables $x_1, \ldots, x_n$ and a *domain* D that maps each variable x_i to a finite set of values, written $D(x_i)$, which it is allowed to take. The CSP is understood to represent the constraint $C \wedge x_1 \in D(x_1) \wedge \cdots \wedge x_n \in D(x_n)$.

Example 3.1
The *map colouring problem* is an archetypal CSP Problem. The problem is to colour the different regions in a particular map with a limited number of colours, subject to the conditions that no two adjacent regions have the same colour. For instance, consider the map of Australia in Figure 3.1 and the colours *red*, *yellow* and *blue*. With each region we associate a variable, WA, NT, SA, Q, NSW, V and T, which will be assigned the colour used to fill that region. Thus each variable has the domain $\{red, yellow, blue\}$. The following constraint captures that adjacent regions may not be filled with the same colour:

$$WA \neq NT \wedge WA \neq SA \wedge NT \neq SA \wedge NT \neq Q \wedge SA \neq Q \wedge$$
$$SA \neq NSW \wedge SA \neq V \wedge Q \neq NSW \wedge NSW \neq V.$$

The user is encouraged to verify that there is in fact a solution.

Example 3.2
Another well-known example of a CSP is the *N-queens problem*. 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. Consider the 4-*queens* problem. We can formalise this as a CSP by associating with the ith queen two variables, R_i and C_i, which are, respectively, its

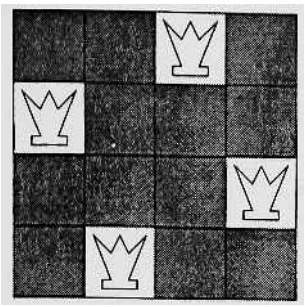

Figure 3.2 Example 4-queens solution.

row and column position on the board. The domain of each variable is $\{1, 2, 3, 4\}$. The constraint

$$R_1 \neq R_2 \wedge R_1 \neq R_3 \wedge R_1 \neq R_4 \wedge R_2 \neq R_3 \wedge R_2 \neq R_4 \wedge R_3 \neq R_4,$$

ensures that no two queens can be in the same row, the constraint

$$C_1 \neq C_2 \wedge C_1 \neq C_3 \wedge C_1 \neq C_4 \wedge C_2 \neq C_3 \wedge C_2 \neq C_4 \wedge C_3 \neq C_4,$$

ensures that no two queens can be in the same column, and the constraints

$$C_1 - R_1 \neq C_2 - R_2 \wedge C_1 - R_1 \neq C_3 - R_3 \wedge C_1 - R_1 \neq C_4 - R_4 \wedge$$
$$C_2 - R_2 \neq C_3 - R_3 \wedge C_2 - R_2 \neq C_4 - R_4 \wedge C_3 - R_3 \neq C_4 - R_4,$$

and

$$C_1 + R_1 \neq C_2 + R_2 \wedge C_1 + R_1 \neq C_3 + R_3 \wedge C_1 + R_1 \neq C_4 + R_4 \wedge$$
$$C_2 + R_2 \neq C_3 + R_3 \wedge C_2 + R_2 \neq C_4 + R_4 \wedge C_3 + R_3 \neq C_4 + R_4,$$

enforce that no two queens are on the same diagonal. One solution to the 4-queens problem is shown in Figure 3.2.

Since each variable has a finite domain, we can represent each of the primitive constraints in a CSP by its finite set of solutions over the values in the variable domains.

Example 3.3

The old-fashioned marriage problem (or, less floridly, the bipartite matching problem) aims to pair off a set of males and females, so that each pair likes each other. The problem is made up of a set of males and females, together with a constraint relation *likes* which describes friendships between males and females. A solution pairs each female with a different male whom they like. A sample problem consists of a set of males $\{kim, peter, bernd\}$, the set of females $\{nicole, maria, erika\}$ and

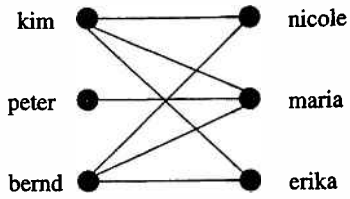

Figure 3.3 The old-fashioned marriage problem.

the relation *likes* defined as the set of pairs

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

A diagram of the *likes* relation is given in Figure 3.3.

The problem can be described using a constraint that uses three variables X_{nicole}, X_{maria}, X_{erika} which represent the males chosen to match *nicole*, *maria* and *erika*, respectively. Each variable has domain $\{kim, peter, bernd\}$ and the constraint is

$$likes(X_{nicole}, nicole) \wedge likes(X_{maria}, maria) \wedge likes(X_{erika}, erika) \wedge$$
$$X_{nicole} \neq X_{maria} \wedge X_{nicole} \neq X_{erika} \wedge X_{maria} \neq X_{erika}.$$

One solution to the problem is $\{X_{nicole} \mapsto kim, X_{maria} \mapsto peter, X_{erika} \mapsto bernd\}$.

Much research has been devoted to the *binary CSP*s. These are CSPs in which the primitive constraints have at most two variables. The map colouring problem and old-fashioned marriage problem are both examples of binary CSPs.

One pleasing feature of a binary CSP is that it can be represented by an undirected graph. Each variable is represented by a node; a unary constraint on variable v is represented by an arc, labelled with the name of the constraint, from the node representing v to itself; and a binary constraint on variables u and v is represented by an arc, labelled with the name of the constraint, between the node corresponding to u and the node corresponding to v. The graph representing the CSP in Example 3.1 is shown in Figure 3.4.

More generally, we can represent non-binary CSPs using multi-graphs. However, it is beyond the scope of this text to pursue this.

3.2 A Simple Backtracking Solver

It is always possible to determine satisfiability of a CSP by a brute force search through all combinations of different values, since the number of such combinations is finite. However, this can be prohibitively expensive. But, any complete solver for CSP problems will, almost certainly, be exponential since solving CSPs is NP-hard. Thus, it is very unlikely that there is a polynomial algorithm to determine satisfiability of an arbitrary CSP. In this section we shall describe a complete solver

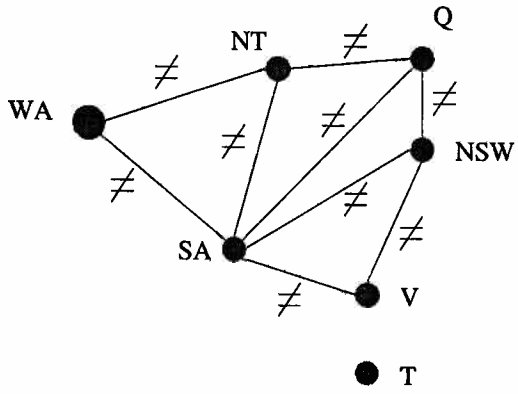

Figure 3.4 Graph of the map colouring CSP.

for CSPs which has exponential time complexity. In later sections we shall study various incomplete solvers which have polynomial time complexity.

One of the simplest techniques for determining satisfiability of an arbitrary CSP is *chronological backtracking*. The idea is to determine satisfiability of a CSP by choosing a variable, and then for each value in its domain, determining satisfiability of the constraint which results by replacing the variable with that value. This is done by calling the backtracking algorithm recursively. It makes use of a parametric function $satisfiable(c)$ which takes a primitive constraint c that involves no variables and returns *true* or *false* indicating whether c is satisfiable or not.

Algorithm 3.1: Chronological backtracking solver
INPUT: A CSP with constraint C and domain D.
OUTPUT: Returns *true* if C is satisfiable, otherwise *false* .
METHOD: The algorithm is shown in Figure 3.5. □

Example 3.4
Consider the CSP consisting of the constraint $X < Y \wedge Y < Z$ and the variables X, Y, Z with domain $\{1, 2\}$. Execution proceeds by choosing a variable in the constraint, say X. The algorithm now iterates through the domain of X which is $\{1, 2\}$. First the domain value 1 is chosen and used to replace X in the original constraint giving $1 < Y \wedge Y < Z$. A call to partial_satisfiable determines that $1 < Y \wedge Y < Z$ is partially satisfiable in the sense that all primitive constraints without variables are satisfiable (in this case there are none so the test is trivial). Now a recursive call is made to the backtracking algorithm so as to determine satisfiability of the new constraint.

In this recursive call another variable, say Y, is chosen. The algorithm iterates through the domain values $\{1, 2\}$. First the domain value of 1 is chosen. Elimination of Y gives the constraint $1 < 1 \wedge 1 < Z$. This is not partially satisfiable, since $1 < 1$ is unsatisfiable. Therefore, in the next iteration the domain value 2 is chosen. Elimination of Y gives the constraint $1 < 2 \wedge 2 < Z$. This is partially satisfiable,

```
C and C1 are constraints;
c1, ... , cn are primitive constraints;
D a domain;
and x is a variable.

back_solve(C,D)
   if vars(C) ≡ ∅ then
      return partial_satisfiable(C)
   else
      choose x ∈ vars(C)
      for each value d ∈ D(x) do
         let C1 be obtained from C by replacing x by d
         if partial_satisfiable(C1) then
            if back_solve(C1,D) then
               return true
            endif
         endif
      endfor
      return false
   endif

partial_satisfiable(C)
   let C be of the form c1 ∧ ··· ∧ cn
   for i := 1 to n do
      if vars(ci) ≡ ∅ then
         if satisfiable(ci) ≡ false then
            return false
         endif
      endif
   endfor
   return true
```

Figure 3.5 Backtracking solver for CSPs.

so the backtracking algorithm is called with $1 < 2 \wedge 2 < Z$.

In this second recursive call the variable Z must be chosen. The recursive call now iterates through the domain values $\{1, 2\}$. First the domain value of 1 is chosen. Elimination of Z gives the constraint $1 < 2 \wedge 2 < 1$. This is not partially satisfiable, so in the next iteration the domain value 2 is chosen. Elimination of Z gives the constraint $1 < 2 \wedge 2 < 2$. Again this is not partially satisfiable, so the second recursive call returns with $false$, indicating that $1 < 2 \wedge 2 < Z$ is not satisfiable.

The first recursive call now returns $false$, since both domain values for Y have been tried and neither can satisfy the constraint $1 < Y \wedge Y < Z$. The original call now tries replacing X by the second domain value 2 giving rise to $2 < Y \wedge Y < Z$. This is partially satisfiable so the backtracking algorithm is called recursively to test for satisfiability. In this case, if Y is chosen for elimination, then both domain values lead to a constraint which is not partially satisfiable, so the recursive call returns $false$.

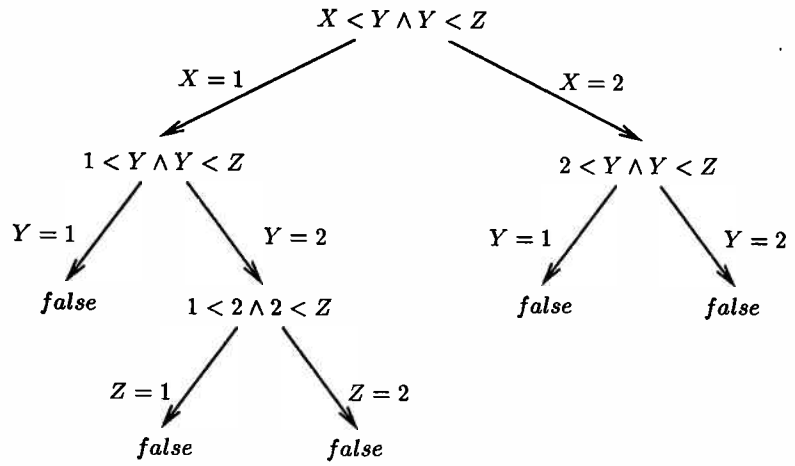

Figure 3.6 Search tree for the backtracking example.

The original call now returns *false* indicating that the original constraint is unsatisfiable. The search tree with recursive calls is shown in Figure 3.6.

As this example illustrates, the backtracking solver is expensive. Indeed, in the worst case it has exponential time complexity. This is to be expected since it is a complete solver. Note also that different choices of variables at different stages lead to different search trees. Good heuristics such as "choose the most constrained variable first" can lead to much smaller search trees. Nonetheless, the underlying worst case complexity remains the same.

3.3 Node and Arc Consistency

We now look at a variety of solvers for CSPs which have polynomial time worst case complexity, but are incomplete. These solvers are based on the observation that if the domain for any variable in the CSP is empty, then the CSP is unsatisfiable. The idea behind each of these solvers is to transform the CSP into an equivalent CSP but one in which the domains of the variables are decreased. If any of the domains become empty, then this CSP, and so the original CSP, are unsatisfiable. By equivalent, we mean that the constraints represented by the CSPs have the same set of solutions.

These solvers work by considering each primitive constraint in turn, and using information about the domain of each variable in the constraint to eliminate values from the domains of the other variables. The solvers are said to be "consistency based" since they propagate information about allowable domain values from one variable to another until the domains are "consistent" with the constraint.

We shall also see that consistency based solvers can be combined with backtracking search. This gives rise to complete solvers in which consistency tests are used

to prune the domain of the variables at each step in the search.

It is useful to identify two special domains which may result after application of a consistency based solver. The first is a "false" domain which indicates that the original problem is unsatisfiable. The second is a "valuation" domain. In such a domain, the domain of each variable is a single value, and so the domain is essentially a valuation.

Definition 3.2
A domain is a *false domain* if some variable in the domain is mapped to the empty set.
A domain is a *valuation domain* if every variable in the domain is mapped to a singleton set.

Given a valuation domain we can straightforwardly determine whether a primitive constraint or constraint evaluates to *true* or *false* when each variable is assigned the value in its domain. We overload the parametric function *satisfiable* to do this. The function $satisfiable(C, D)$ takes a constraint C and a valuation domain D and returns *true* or *false* indicating whether C is satisfiable or not under this valuation.

For instance, consider the variables X, Y, Z. The domain D_1 in which $D_1(X) = D_1(Y) = \{1, 2\}$ and $D_1(Z) = \emptyset$ is a false domain. The domain D_2 in which $D_2(X) = \{1\}$, $D_2(Y) = \{2\}$ and $D_2(Z) = \{1\}$ is a valuation domain and $satisfiable(C, D_2)$ where C is the constraint $X < Y \wedge Y < Z$ is *false*.

The first propagation based solver we will look at is for binary CSPs. The solver is based on the following two notions of "consistency" which are based on the graph representation for binary CSPs.

Definition 3.3
A primitive constraint c is *node consistent* with domain D if either $|vars(c)| \neq 1$ or, if $vars(c) = \{x\}$, then for each $d \in D(x)$, $\{x \mapsto d\}$ is a solution of c.
A CSP with constraint $c_1 \wedge \cdots \wedge c_n$ and domain D is *node consistent* if each primitive constraint c_i is node consistent with D for $1 \leq i \leq n$.

Definition 3.4
A primitive constraint c is *arc consistent* with domain D if either $|vars(c)| \neq 2$ or, if $vars(c) = \{x, y\}$, then for each $d_x \in D(x)$, there is some $d_y \in D(y)$ such that $\{x \mapsto d_x, y \mapsto d_y\}$ is a solution of c and for each $d_y \in D(y)$, there is some $d_x \in D(x)$ such that $\{x \mapsto d_x, y \mapsto d_y\}$ is a solution of c.
A CSP with constraint $c_1 \wedge \cdots \wedge c_n$ and domain D is *arc consistent* if each primitive constraint c_i is arc consistent with D for $1 \leq i \leq n$.

For instance, the CSP in Example 3.1 is node and arc consistent. It is node consistent simply because there are no constraints involving only a single variable. Every disequation in the example is arc consistent since for every colour in the domain of the first variable in the disequality there is a colour not equal to that colour in the domain of the second variable, and vice versa. Thus all primitive

constraints are arc consistent, so the CSP is arc consistent. Similarly, we can argue that the CSP in Example 3.2 is node and arc consistent.

If we modify the map colouring problem of Example 3.1 so as to allow only two colours, then it becomes unsatisfiable, since for example, there is no way to colour the three regions WA, NT and SA with two colours. However, it remains arc consistent, demonstrating that a binary CSP can be unsatisfiable even when it is node and arc consistent.

The CSP in Example 3.4 is node consistent, since it has no constraints that involve only a single variable. It is not arc consistent, however, because, for example, considering the constraint $X < Y$ and the value 1 for Y, there is no value in the domain of X, which is $\{1, 2\}$, that will satisfy the constraint.

An example of a CSP which is neither node nor arc consistent is the constraint

$$X < Y \wedge Y < Z \wedge Z \leq 2$$

with the domain D where $D(X) = D(Y) = D(Z) = \{1, 2, 3\}$. It is not node consistent because the value 3 for Z is not consistent with the primitive constraint $Z \leq 2$. It is not arc consistent using the same reasoning as in the previous paragraph.

It is straightforward to transform a CSP into an equivalent CSP which is node consistent—for each variable we intersect the solutions to the constraints involving only that variable with its original domain to give the new domain.

Algorithm 3.2: Node consistency
INPUT: A CSP with constraint C and domain D_1.
OUTPUT: Returns a domain D_2 such that the CSP with constraints C and domain D_2 is node consistent and is equivalent to the input CSP.
METHOD: The algorithm is shown in Figure 3.7. □

It is also easy to transform a CSP into an equivalent CSP which is arc consistent—for each variable x, the algorithm takes each binary constraint involving that variable and uses the domain of the other variable, y say, that appears in the constraint to restrict the domain of x. This process is repeated until the domain of no variable can be decreased. The algorithm given here is simple but it is quite inefficient. More efficient algorithms are referred to in the Notes section.

Algorithm 3.3: Arc consistency
INPUT: A CSP with constraint C and domain D_1.
OUTPUT: Returns a domain D_2 such that the CSP with constraints C and domain D_2 is arc consistent and is equivalent to the input CSP.
METHOD: The algorithm is shown in Figure 3.8. □

We can employ the arc and node consistency algorithms to give an incomplete solver for determining if a CSP is satisfiable as follows. Given a CSP, transform it to an equivalent node consistent CSP, then transform this to an equivalent arc consistent CSP with a new domain. If this domain is a false domain, the algorithm returns *false* since the original CSP must be unsatisfiable. If this

```
x is a variable;
C is a constraint;
D is a domain;
c_1, ..., c_n are primitive constraints;
and d is a domain value.

node_consistent(C,D)
    let C be of form c_1 ∧ ··· ∧ c_n
    for i := 1 to n do
        D := node_consistent_primitive(c_i,D)
    endfor
    return D

node_consistent_primitive(c,D)
    if |vars(c)| = 1 then
        let {x} = vars(c)
        D(x) := {d ∈ D(x) | {x ↦ d} is a solution of c}
    endif
    return D
```

Figure 3.7 Node consistency algorithm.

```
C is a constraint;
D and W are domains;
x, y are variables;
d_x, d_y are domain values;
and c, c_1, ..., c_n are primitive constraints.

arc_consistent(C,D)
    let C be of form c_1 ∧ ··· ∧ c_n
    repeat
        W := D
        for i := 1 to n do
            D := arc_consistent_primitive(c_i,D)
        endfor
    until W ≡ D
    return D

arc_consistent_primitive(c,D)
    if |vars(c)| = 2 then
        let {x, y} = vars(c)
        D(x) := {d_x ∈ D(x) | for some d_y ∈ D(y), {x ↦ d_x, y ↦ d_y} is a solution of c}
        D(y) := {d_y ∈ D(y) | for some d_x ∈ D(x), {x ↦ d_x, y ↦ d_y} is a solution of c}
    endif
    return D
```

Figure 3.8 Arc consistency algorithm.

```
C is a constraint;
and D is a domain.

arc_solv(C,D)
    D := node_arc_consistent(C,D)
    if D is a false domain then
        return false
    elseif D is a valuation domain then
        return satisfiable(C, D)
    else
        return unknown
    endif

node_arc_consistent(C, D)
    D := node_consistent(C,D)
    D := arc_consistent(C,D)
    return D
```

Figure 3.9 Arc consistency solver.

domain is a valuation domain it returns the result of evaluating the constraint with that valuation. Otherwise it returns *unknown* since the CSP may or may not be satisfiable.

Although a constraint solver usually takes a single argument, a constraint, we relax this restriction for finite domain solvers, giving two arguments: a constraint and a domain. The two arguments can be extracted from a single constraint of the form $C \wedge x_1 \in D(x_1) \wedge \cdots \wedge x_n \in D(x_n)$ or some default domain D can be used which assigns some fixed finite set of values to all variables.

Algorithm 3.4: Incomplete node and arc consistency solver
INPUT: A CSP with constraint C and domain D.
OUTPUT: Returns *true*, *false* or *unknown*.
METHOD: The algorithm arc_solv is shown in Figure 3.9. □

For instance, consider the CSP for the constraint

$$X < Y \ \wedge \ Y < Z \ \wedge \ Z \leq 2,$$

with domain $D(X) = D(Y) = D(Z) = \{1, 2, 3\}$. Application of the node consistency algorithm will leave the domain of X and Y as $\{1, 2, 3\}$, but the domain of Z will become $\{1, 2\}$. Application of the arc consistency algorithm proceeds as follows. In the first iteration the constraint $X < Y$ is considered. The new domain of X is computed to be $\{1, 2\}$ from interaction of the constraint $X < Y$ and the domain $\{1, 2, 3\}$ for Y. The value of 3 is removed from the domain of X because there is no value in Y's domain which satisfies $3 < Y$. Similarly the new domain of Y is calculated to be $\{2, 3\}$. Examining the constraint $Y < Z$ the domain of Y is computed to be $\emptyset$ since there are no values lower than a value for Z. The domain of Z also becomes $\emptyset$. On the next iteration the domain of X will become the empty

```
C is a constraint;
D and D_1 are domains;
x is a variable;
and d is a domain value.

back_arc_solv(C,D)
   D := node_arc_consistent(C,D)
   if D is a false domain then
      return false
   elseif D is a valuation domain then
      if satisfiable(C, D) then
         return D
      else
         return false
      endif
   endif
   choose variable x such that |D(x)| ≥ 2
   for each d ∈ D(x)
      D_1 := back_arc_solv(C ∧ x = d,D)
      if D_1 ≢ false then
         return D_1
      endif
   endfor
   return false
```

Figure 3.10 Backtracking arc consistency solver.

set. The next iteration will not change the domains and the algorithm will return that the original system is unsatisfiable.

The node and arc consistency algorithms can also be combined with the backtracking algorithm to give a complete solver. Initially the node consistency and arc consistency algorithms are used to restrict the domains. Subsequently, whenever the backtracking algorithm chooses a value for a variable, the node and arc consistency algorithm is used to restore the domains to node and arc consistency. If this makes the domain false, then the resultant CSP is unsatisfiable and backtracking to another domain value is required.

Algorithm 3.5: Complete node and arc consistency solver
INPUT: A CSP with constraint C and domain D.
OUTPUT: Returns *true* or *false*
METHOD: If the result of back_arc_solv(C, D) returns *false* then return *false*, otherwise return *true*. The algorithm for back_arc_solv is shown in Figure 3.10.
□

3.4 Bounds Consistency

Arc and node consistency work well for pruning the domains in binary CSPs. However, they do not work well if the problem contains primitive constraints that involve more than two variables since such primitive constraints are ignored when performing consistency checks. We would, therefore, like to extend this technique to handle primitive constraints involving more than two variables.

Hyper-arc consistency extends the arc consistency condition to apply to every primitive constraint, regardless of the number of variables it contains.

Definition 3.5
A primitive constraint c is *hyper-arc consistent* with domain D if for each variable $x \in vars(c)$ and domain assignment $d \in D(x)$, there is an assignment to the remaining variables in c, say $x_1, x_2, \ldots, x_k$, such that $d_j \in D(x_j)$ for $1 \leq j \leq k$ and $\{x \mapsto d, x_2 \mapsto d_2, \ldots, x_k \mapsto d_k\}$ is a solution of c.
A CSP with constraint $c_1 \wedge \cdots \wedge c_n$ and domain D is *hyper-arc consistent* if each primitive constraint c_i is hyper-arc consistent with D for $1 \leq i \leq n$.

Hyper-arc consistency is a true generalization of node and arc consistency. In the case in which the primitive constraint has two variables, hyper-arc consistency is equivalent to arc consistency, and in the case when a primitive constraint has one variable, it is equivalent to node consistency.

Unfortunately, testing for hyper-arc consistency can be very expensive even for fairly simple primitive constraints which involve more than two variables. Consider the primitive constraint

$$X = 3Y + 5Z$$

with domain D where $D(X) = \{2,3,4,5,6,7\}$, $D(Y) = \{0,1,2\}$ and $D(Z) = \{-1,0,1,2\}$. Hyper-arc consistency applied to this primitive constraint must determine which possible values of X are legitimate. Take the first value, 2, for instance. It is consistent if there is a solution to $2 = 3Y + 5Z$ where $Y \in \{0,1,2\}$ and $Z \in \{-1,0,1,2\}$. This is a non-trivial problem and is not simple to check. If values which are not hyper-arc consistent are removed the new domain is D_1 where $D_1(X) = \{3,5,6\}$, $D_1(Y) = \{0,1,2\}$ and $D_1(Z) = \{0,1\}$.

For slightly more complex primitive constraints, such as

$$12X + 107Y - 17Z + 5T = 2$$

where the initial domains for all variables are $\{0,1,\ldots,1000\}$, we can see that the hyper-arc consistency check is very expensive. Indeed, in general, determining if an arbitrary primitive constraint is hyper-arc consistent is NP-hard and so is as expensive as determining if an arbitrary CSP is satisfiable. Thus, hyper-arc consistency is too expensive to use in an incomplete solver or to use at each step in a backtracking solver.

We therefore need some other consistency condition for primitive constraints involving more than two variables. Luckily, there is a large and useful class of CSPs for which there is a natural choice of a consistency condition which can be used with primitive constraints involving an arbitrary number of variables.

Definition 3.6
A CSP is *arithmetic* if each variable in the CSP ranges over a finite domain of integers and the primitive constraints are arithmetic constraints.

The 4-queens problem is an example of an arithmetic CSP. Arithmetic CSPs are, arguably, the most important class of CSPs since most problems of commercial interest can be naturally modelled using integer CSPs. Furthermore, it is possible to encode many non-arithmetic CSPs as arithmetic CSPs, simply by representing the domain values by integers. For instance, if we represent colours by integers, the map colouring problem becomes an arithmetic CSP.

The CSP solving methods discussed in the previous section can, of course, be used to solve arithmetic CSPs since they are simply a particular type of CSP. Indeed Examples 3.2 and 3.4 solve arithmetic CSPs. However, the restriction to integers and arithmetic constraints allows us to define a new type of consistency: bounds consistency. There are two ideas behind bounds consistency. The first is to approximate the domain of an integer variable using a lower and an upper bound. The second is to use real number consistency of primitive constraints rather than integer consistency.

Definition 3.7
A *range*, $[l..u]$, represents the set of integers $\{l, l+1, \ldots, u\}$ if $l \leq u$, otherwise it represents the empty set.
If D is a domain over the integers, $min_D(x)$ is the minimum element in $D(x)$ and $max_D(x)$ is the maximum element in $D(x)$.

Definition 3.8
An arithmetic primitive constraint c is *bounds consistent* with domain D if for each variable $x \in vars(c)$, there is:

- an assignment of *real* numbers, say $d_1, d_2, \ldots, d_k$, to the remaining variables in c, say $x_1, x_2, \ldots, x_k$, such that $min_D(x_j) \leq d_j \leq max_D(x_j)$ for each d_j and

$$\{x \mapsto min_D(x), x_1 \mapsto d_1, \ldots, x_k \mapsto d_k\}$$

is a solution of c and

- another assignment of *real* numbers, say $d'_1, d'_2, \ldots, d'_k$, to $x_1, x_2, \ldots, x_k$ such that $min_D(x_j) \leq d'_j \leq max_D(x_j)$ for each d'_j and

$$\{x \mapsto max_D(x), x_1 \mapsto d'_1, \ldots, x_k \mapsto d'_k\}$$

is a solution of c.

An arithmetic CSP with constraint $c_1 \wedge \cdots \wedge c_n$ and domain D is *bounds consistent* if each primitive constraint c_i is bounds consistent with D for $1 \leq i \leq n$.

Since bounds consistency only depends on the upper and lower bounds of the domains of the variables, when testing for bounds consistency we need only consider domains that assign ranges to each variable. Therefore, in the following examples we will often use ranges to describe the domain of each variable.

Consider the constraint from the previous example, $X = 3Y + 5Z$, with domain D where

$$D(X) = [2..7], \quad D(Y) = [0..2], \quad D(Z) = [-1..2].$$

This constraint is not bounds consistent with D. To see this, consider $5Z = X - 3Y$ which is equivalent to the original constraint. If Z takes its maximum value 2, then the left hand side has value 10, but the maximum value of the right hand side expression $X - 3Y$ is $7 - 3 \times 0 = 7$. Hence the range for Z must be changed to obtain bounds consistency. The domain D_1 where

$$D_1(X) = [2..7], \quad D_1(Y) = [0..2], \quad D_1(Z) = [0..1],$$

however, is bounds consistent with the constraint.

Given a current range for each of the variables in a primitive constraint we can devise an efficient method for calculating a new range for each variable in the constraint which is bounds consistent with the constraint. We will refer to such methods as *propagation rules.*

Consider the simple constraint $X = Y + Z$. By writing the constraint in the three forms:

$$X = Y + Z, \quad Y = X - Z \quad \text{and} \quad Z = X - Y$$

and reasoning about the minimum and maximum values of the right hand sides, we can see that

$$\begin{aligned} X &\geq min_D(Y) + min_D(Z), & X &\leq max_D(Y) + max_D(Z), \\ Y &\geq min_D(X) - max_D(Z), & Y &\leq max_D(X) - min_D(Z), \\ Z &\geq min_D(X) - max_D(Y), & Z &\leq max_D(X) - min_D(Y). \end{aligned}$$

From these inequalities we can derive simple rules for ensuring bounds consistency which use the current domains of each variable to calculate the values of the right hand side expressions, and use these values to appropriately update the minimum and maximum values of each variable's domain. An algorithm implementing the propagation rules for the constraint $X = Y + Z$ is shown in Figure 3.11.

For example, given the domain

$$D(X) = [4..8], \quad D(Y) = [0..3], \quad D(Z) = [2..2]$$

D is a domain;
X, Y and Z are fixed variables;
X_{min}, X_{max}, d_X, Y_{min}, Y_{max}, d_Y, Z_{min}, Z_{max} and d_Z are domain values.

bounds_consistent_addition(D)
  $X_{min} :=$ maximum $\{min_D(X), min_D(Y) + min_D(Z)\}$
  $X_{max} :=$ minimum $\{max_D(X), max_D(Y) + max_D(Z)\}$
  $D(X) := \{d_X \in D(X) | X_{min} \leq d_X \leq X_{max}\}$
  $Y_{min} :=$ maximum $\{min_D(Y), min_D(X) - max_D(Z)\}$
  $Y_{max} :=$ minimum $\{max_D(Y), max_D(X) - min_D(Z)\}$
  $D(Y) := \{d_Y \in D(Y) | Y_{min} \leq d_Y \leq Y_{max}\}$
  $Z_{min} :=$ maximum $\{min_D(Z), min_D(X) - max_D(Y)\}$
  $Z_{max} :=$ minimum $\{max_D(Z), max_D(X) - min_D(Y)\}$
  $D(Z) := \{d_Z \in D(Z) | Z_{min} \leq d_Z \leq Z_{max}\}$
  return D

Figure 3.11 Propagation rules for the primitive constraint $X = Y + Z$.

we can determine that

$$\begin{aligned} 2 &\leq X \leq 5, \\ 2 &\leq Y \leq 6, \\ 1 &\leq Z \leq 8. \end{aligned}$$

Therefore, we can update the domain to

$$D(X) = [4..5], \quad D(Y) = [2..3], \quad D(Z) = [2..2]$$

without removing any solutions to the constraint.

If we now update this new domain using the propagation rules we determine that

$$\begin{aligned} 4 &\leq X \leq 5, \\ 2 &\leq Y \leq 3, \\ 1 &\leq Z \leq 3. \end{aligned}$$

The new domain satisfies all these constraints. Thus, $X = Y + Z$ is bounds consistent with the new domain. In fact, we need only apply these propagation rules once to any domain to obtain a domain which is bounds consistent with the constraint $X = Y + Z$.

We can also determine propagation rules for more complicated linear arithmetic constraints. For example, consider the constraint

$$4W + 3P + 2C \leq 9.$$

We can rewrite this into the three forms

$$W \leq \tfrac{9}{4} - \tfrac{3}{4}P - \tfrac{2}{4}C, \quad P \leq \tfrac{9}{3} - \tfrac{4}{3}W - \tfrac{2}{3}C, \quad C \leq \tfrac{9}{2} - 2W - \tfrac{3}{2}P$$

D is a domain;
W, P and C are fixed variables;
W_{max}, d_W, P_{max}, d_P, C_{max}, and d_C are domain values;

bounds_consistent_inequality(D)
  $W_{max} := \text{minimum } \{max_D(W), \lfloor \frac{9}{4} - \frac{3}{4} min_D(P) - \frac{2}{4} min_D(C) \rfloor\}$
  $D(W) := \{d_W \in D(W) | d_W \leq W_{max}\}$
  $P_{max} := \text{minimum } \{max_D(P), \lfloor \frac{9}{3} - \frac{4}{3} min_D(W) - \frac{2}{3} min_D(C) \rfloor\}$
  $D(P) := \{d_P \in D(P) | d_P \leq P_{max}\}$
  $C_{max} := \text{minimum } \{max_D(C), \lfloor \frac{9}{2} - 2\, min_D(W) - \frac{3}{2} min_D(P) \rfloor\}$
  $D(C) := \{d_P \in D(C) | d_C \leq C_{max}\}$
  return D

Figure 3.12 Propagation rules for $4W + 3P + 2C \leq 9$.

and so we obtain the inequalities

$$\begin{aligned} W &\leq \tfrac{9}{4} - \tfrac{3}{4} min_D(P) - \tfrac{2}{4} min_D(C), \\ P &\leq \tfrac{9}{3} - \tfrac{4}{3} min_D(W) - \tfrac{2}{3} min_D(C), \\ C &\leq \tfrac{9}{2} - 2\, min_D(W) - \tfrac{3}{2} min_D(P). \end{aligned}$$

From these inequalities we obtain the propagation rules shown in Figure 3.12. Note that the function $\lfloor F \rfloor$ takes a real number and rounds it down to the nearest integer.

Given the initial domain

$$D(W) = [0..9], \quad D(P) = [0..9], \quad D(C) = [0..9]$$

we can determine that $W \leq \frac{9}{4}$, $P \leq \frac{9}{3}$, and $C \leq \frac{9}{2}$. Using the propagation rules we update the domain to give

$$D(W) = [0..2], \quad D(P) = [0..3], \quad D(C) = [0..4].$$

Notice that $W \leq \frac{9}{4}$ means we can lower the upper bound for W to $\lfloor \frac{9}{4} \rfloor = 2$, since we know W takes integer values only. Similarly, $C \leq \frac{9}{2}$ means we can lower the upper bound for C to 4.

Example 3.5

Consider the *smuggler's knapsack problem.* A smuggler has a knapsack of limited capacity, say 9 units. He can smuggle in bottles of whiskey of size 4 units, bottles of perfume of size 3 units, and cartons of cigarettes of size 2 units. The profit from smuggling a bottle of whiskey, bottle of perfume or carton of cigarettes is 15 dollars, 10 dollars and 7 dollars respectively. If the smuggler will only take a trip if he makes a profit of 30 dollars or more, what can he take?

This problem can be modelled as an arithmetic non-binary CSP using three variables W, P and C whose values are the number of whiskey, perfume and cigarette items in the knapsack, respectively. An initial domain for each variable is

[0..9]. The capacity constraint is

$$4W + 3P + 2C \leq 9,$$

while the profit constraint is

$$15W + 10P + 7C \geq 30.$$

Using the propagation rules given above for the capacity constraint we update the domain to get

$$D(W) = [0..2], \quad D(P) = [0..3], \quad D(C) = [0..4].$$

Application of the propagation rules for the profit constraint does not change the domain.

Another interesting case is for the primitive linear disequation constraint $Y \neq Z$. In this case the requirement for bounds consistency is quite weak since as long as each variable has a domain of two or more possible values then the disequation is bounds consistent. However when one variable, say Z, has a fixed value, that is $min_D(Z) \equiv max_D(Z)$, then we have

$$Y \neq min_D(Z).$$

With bounds consistency, the only time this disequality can be inconsistent is when the current minimum or maximum of Y equals the (fixed) value of Z. In this case we can remove this value from the domain of Y and increase or decrease the respective bound.

For instance, given the domain $D(Y) = [2..3], D(Z) = [2..2]$ and primitive constraint $Y \neq Z$, the minimum value of Y is disallowed by the constraint. Therefore, the updated domain is $D(Y) = [3..3], D(Z) = [2..2]$ which is now bounds consistent. Note that a simple disequation constraint of the form $Y \neq d$, where d is some integer, is handled analogously to $Y \neq Z$.

We can also devise propagation rules for nonlinear constraints. Consider, for example, the constraint

$$X = \text{minimum}\{Y, Z\}$$

which holds whenever X takes the minimum of the values of Y and Z. The propagation rules for this constraint follow directly from the following inequalities

$$\begin{aligned} Y &\geq min_D(X), \\ Z &\geq min_D(X), \\ X &\geq \text{minimum}\{min_D(Y), min_D(Z)\}, \\ X &\leq \text{minimum}\{max_D(Y), max_D(Z)\}. \end{aligned}$$

The first inequality holds because Y cannot take a value less than the minimum

value of X since otherwise the constraint could not be satisfied. Thus, Y's minimum is greater than that of X. The second inequality has an analogous justification. The final two inequalities hold because the value of X is one of the values of Y and Z and so it cannot be less than both of their possible values, and it cannot be greater than either of their possible values.

However, for some nonlinear constraints determining propagation rules is more difficult. Consider the deceptively simple multiplication constraint

$$X = Y \times Z.$$

Let us first consider the case when all the numbers are strictly positive. In this case propagation rules can be based on the inequalities

$$\begin{aligned} X &\geq min_D(Y) \times min_D(Z), \\ X &\leq max_D(Y) \times max_D(Z), \\ Y &\geq min_D(X)/max_D(Z), \\ Y &\leq max_D(X)/min_D(Z), \\ Z &\geq min_D(X)/max_D(Y), \\ Z &\leq max_D(X)/min_D(Y). \end{aligned}$$

For example, for the domain

$$D(X) = [1..4], \quad D(Y) = [1..2], \quad D(Z) = [1..4]$$

these inequalities give

$$X \geq 1,\ X \leq 8,\ Y \geq 1,\ Y \leq 4,\ Z \geq 1,\ Z \leq 4.$$

Thus, propagation rules based on these inequalities will not change the domain.

With the possibility of variables taking zero or negative values the propagation rules become much more complex. In general, we can only calculate the bounds for X by examining all combinations of the extreme positions of Y and Z. Thus,

$$\begin{aligned} X &\geq \text{minimum} \quad \{min_D(Y) \times min_D(Z), min_D(Y) \times max_D(Z), \\ &\qquad\qquad max_D(Y) \times min_D(Z), max_D(Y) \times max_D(Z)\}, \\ X &\leq \text{maximum} \quad \{min_D(Y) \times min_D(Z), min_D(Y) \times max_D(Z), \\ &\qquad\qquad max_D(Y) \times min_D(Z), max_D(Y) \times max_D(Z)\}. \end{aligned}$$

Given domain

$$D(X) = [4..8], \quad D(Y) = [0..3], \quad D(Z) = [-2..2]$$

we can determine that

$$\begin{aligned} X &\geq \text{minimum}\{0, 0, -6, 6\} = -6, \\ X &\leq \text{maximum}\{0, 0, -6, 6\} = 6. \end{aligned}$$

Propagation rules for the other variables are even more complex. This is because variables may take values very near zero.

Given $D(X) = [4..8]$ and $D(Z) = [-2..2]$, any non-zero value of Y, say d, is bounds consistent because the valuation $\{X \mapsto 4, Y \mapsto d, Z \mapsto (4/d)\}$ is a solution of $X = Y \times Z$ since $-2 \leq 4/d \leq 2$ for all d where $|d| \geq 2$. Hence, updating Y is complicated. As long as the domain of Z is such that $min_D(Z) < 0$ and $max_D(Z) > 0$, the valuations used in testing bounds consistency for Y can contain arbitrarily small positive and negative real values for Z. This means that the multiplication constraint is bounds consistent with any value of Y. Therefore, as long as $min_D(Z) < 0$ and $max_D(Z) > 0$, bounds consistency cannot be used to reduce the range of variable Y. When this is not the case, we can propagate, again by simply examining the four possibilities at the extremes of the ranges for X and Z using the inequalities

$$\begin{array}{lll} Y \geq \text{minimum} & \{min_D(X)/min_D(Z), min_D(X)/max_D(Z), \\ & max_D(X)/min_D(Z), max_D(X)/max_D(Z)\}, \\ Y \leq \text{maximum} & \{min_D(X)/min_D(Z), min_D(X)/max_D(Z), \\ & max_D(X)/min_D(Z), max_D(X)/max_D(Z)\} \end{array}$$

where division by zero yields $+\infty$ when dividing a positive number, $-\infty$ when dividing a negative number and $0/0$ is considered as $-\infty$ on the right of a $\geq$ and $+\infty$ on the right hand side of a $\leq$. Using the same reasoning, the inequalities for Z are, when $min_D(Y) \geq 0$ or $max_D(Y) \leq 0$,

$$\begin{array}{lll} Z \geq \text{minimum} & \{min_D(X)/min_D(Y), min_D(X)/max_D(Y), \\ & max_D(X)/min_D(Y), max_D(X)/max_D(Y)\}, \\ Z \leq \text{maximum} & \{min_D(X)/min_D(Y), min_D(X)/max_D(Y), \\ & max_D(X)/min_D(Y), max_D(X)/max_D(Y)\}. \end{array}$$

Another complication of the multiplication constraint is that, unlike the previous constraints we have examined, applying the propagation rules to a domain once does not guarantee that the resulting domain will be bounds consistent. This is because the conditions required for applying the propagation rules for Y and Z may not be satisfied initially, but may be satisfied after applying the propagation rules. Thus we may need to apply the same rules repeatedly to obtain a bounds consistent domain.

For instance, given the domain

$$D(X) = [4..8], \quad D(Y) = [-1..1], \quad D(Z) = [-4..-1]$$

we initially determine that no propagation is possible for Z. However for X and Y

we have that

$$\begin{aligned} X &\geq \text{minimum}\{4,-4,1,-1\} = -4, \\ X &\leq \text{maximum}\{4,-1,1,-1\} = 4, \\ Y &\geq \text{minimum}\{\tfrac{4}{-4},\tfrac{8}{-4},\tfrac{4}{-1},\tfrac{8}{-1}\} = \text{minimum}\{-1,-2,-4,-8\} = -8, \\ Y &\leq \text{maximum}\{-1,-2,-4,-8\} = -1. \end{aligned}$$

Thus, after applying the corresponding propagation rules, the domain becomes

$$D(X) = [4..4], \quad D(Y) = [-1..-1], \quad D(Z) = [-4..-1].$$

Another propagation step is required to determine that $D(Z) = [-4..-4]$.

As we have seen from these examples, determining the propagation rules for a particular primitive constraint is quite tedious. For this reason, bounds consistency constraint solvers usually handle only a restricted class of primitive constraints. A typical restriction is that constraints must either be linear arithmetic equations or inequalities or a nonlinear constraint in a specific form such as $t_1 = t_2 \times t_3$, $t_1 \neq t_2$, $t_1 = \text{minimum}\{t_2, t_3\}$, where each t_i is either a variable or a constant and no variable occurs more than once in the primitive constraint.

More complex primitive constraints must be replaced by conjunctions of these legitimate primitive constraints. Unfortunately, because the consistency methods are incomplete, replacing a complex constraint by different but equivalent legitimate primitive constraints may not lead to equivalent behaviour.

Example 3.6

Consider the constraint $(X-1)^2 = Y$. We could rewrite this to

$$T_1 = X - 1 \wedge T_1 = T_2 \wedge Y = T_1 \times T_2$$

or to

$$U_1 = X \wedge U_2 = U_1 \times X \wedge U_2 - 2X + 1 = Y.$$

If the initial domain is $D(X) = [2..5]$, $D(Y) = [-3..10]$ then, using the first rewriting, bounds consistency determines the domain

$$D(X) = [2..5], \quad D(Y) = [1..10], \quad D(T_1) = [1..4], \quad D(T_2) = [1..4].$$

Using the second rewriting, bounds consistency determines the domain

$$D(X) = [2..5], \quad D(Y) = [-3..10], \quad D(U_1) = [2..5], \quad D(U_2) = [4..25].$$

In this case, the second rewriting leads to less propagation than the first.

One potential problem, therefore, is that if the breakdown of complex constraints is handled automatically by the constraint solver, the user of the solver may not understand why it fails to detect some inconsistencies.

It is simple to give an algorithm which transforms an arithmetic CSP to an equivalent bounds consistent CSP. The algorithm uses the parametric function

```
C is an arithmetic constraint;
C_0 is a set of primitive constraints;
D and D_1 are domains;
c_1, ..., c_n are primitive constraints;
and x is a variable.

bounds_consistent(C,D)
   let C be of form c_1 ∧ ··· ∧ c_n
   C_0 := {c_1, ..., c_n}
   while C_0 ≠ ∅ do
      choose c ∈ C_0
      C_0 := C_0 \ {c}
      D_1 := bounds_consistent_primitive(c, D)
      if D_1 is a false domain then return D_1 endif
      for i := 1 to n do
         if there exists x ∈ vars(c_i) such that D_1(x) ≠ D(x) then
            C_0 := C_0 ∪ {c_i}
         endif
      endfor
      D := D_1
   endwhile
   return D
```

Figure 3.13 Bounds consistency algorithm.

bounds_consistent_primitive(c, D) which applies the propagation rules for primitive constraint c to the domain D and returns the new domain. Examples of this function for particular linear constraints have been given above. We assume that the original CSP has been transformed into a CSP containing only legitimate primitive constraints.

The algorithm works by repeatedly processing the *active set* of primitive constraints C_0. A primitive constraint is active if it may not be bounds consistent with the current domain D. The main **while** loop repeatedly selects a primitive constraint c from the active set and modifies the domain D to be bounds consistent with c. If this gives a false domain, the algorithm terminates, otherwise those constraints which might no longer be bounds consistent because of the changes made to D are added to the active set. The algorithm terminates when there are no primitive constraints in the active set since this means all of the primitive constraints are bounds consistent with the current domain.

Algorithm 3.6: Bounds consistency
INPUT: An arithmetic CSP with constraint C and domain D.
OUTPUT: bounds_consistent(C, D) returns a domain D_1 such that the CSP with constraints C and domain D_1 is bounds consistent and is equivalent to the input CSP.
METHOD: The algorithm is shown in Figure 3.13. □

For example, consider the constraint $X = Y + Z \wedge Y \neq Z$ and domain

$$D(X) = [4..8], \quad D(Y) = [0..3], \quad D(Z) = [2..2].$$

Execution of bounds_consistent proceeds as follows. Initially, C_0 is set to the set $\{X = Y + Z, Y \neq Z\}$. Now some c, say $X = Y + Z$, is removed from C_0. The procedure bounds_consistent_primitive evaluates the propagation rules for $X = Y + Z$ giving the updated domain

$$D(X) = [4..5], \quad D(Y) = [2..3], \quad D(Z) = [2..2].$$

Since the variable X has changed its range, the constraint $X = Y + Z$ is added to C_0 (although this is unnecessary since its propagation rules ensure bounds consistency with respect to itself). Now another primitive constraint, say $Y \neq Z$, is removed from C_0. The call to bounds_consistent_primitive removes the value 2 from the range of Y giving the updated domain

$$D(X) = [4..5], \quad D(Y) = [3..3], \quad D(Z) = [2..2].$$

The constraint $Y \neq Z$ is added to C_0 because Y's range has changed. Another constraint is removed from C_0, say $X = Y + Z$, and its propagation rules are applied yielding the domain

$$D(X) = [5..5], \quad D(Y) = [3..3], \quad D(Z) = [2..2].$$

Since the range of X has changed, $X = Y + Z$ is added to C_0. Further processing of the constraints, $X = Y + Z$ and $Y \neq Z$, in C_0 does not change the domain so the procedure terminates returning this domain which is bounds consistent with the constraint.

It is simple to define an incomplete constraint solver for arithmetic CSPs based on the bounds consistency algorithm.

Algorithm 3.7: Bounds consistency solver
INPUT: An arithmetic CSP with constraint C and domain D
OUTPUT: bounds_solv(C,D) returns *true*, *false* or *unknown* depending on the satisfiability of the constraint C with domain D.
METHOD: bounds_solv(C,D) is identical to arc_solv(C,D) (shown in Figure 3.9) except that the call to node_arc_consistent is replaced by a call to bounds_consistent. □

Consider the execution of this solver on the constraint $X < Y \wedge Y < Z$ where each variable lies in the range [1..4]. Initially, we have $D(X) = D(Y) = D(Z) = [1..4]$. The call to bounds_consistent(C, D) sets C_0 to $\{X < Y, Y < Z\}$. Considering the primitive constraint $X < Y$, we obtain $D(X) = [1..3], D(Y) = [2..4]$ and $D(Z)$ is unchanged. This causes the constraint $X < Y$ to be replaced in C_0. Now considering $Y < Z$ we obtain $D(Y) = [2..3], D(Z) = [3..4]$, and $Y < Z$ is replaced in C_0. Reconsidering $X < Y$ we obtain $D(X) = [1..2]$ and $X < Y$ is replaced in

C_0. Processing $X < Y$ and $Y < Z$ causes no further change to D so the call to bounds_consistent returns

$$D(X) = [1..2], \quad D(Y) = [2..3], \quad D(Z) = [3..4].$$

Thus, bounds_solv returns *unknown*.

It is also straightforward to combine the bounds consistency algorithm with a backtracking search to produce a complete bounds consistency solver, analogous to the node and arc consistency based solver given in Algorithm 3.5.

Algorithm 3.8: Complete bounds consistency solver
INPUT: An arithmetic CSP with constraint C and domain D.
OUTPUT: Returns *true* or *false*.
METHOD: If the result of back_bounds_solv(C,D) returns *false* then return *false*, otherwise return *true*. back_bounds_solv(C,D) is identical to back_arc_solv(C,D) (shown in Figure 3.10) except that the call to node_arc_consistent is replaced by a call to bounds_consistent, and the recursive call to back_arc_solv is replaced by a call to back_bounds_solv. □

Using the complete bounds consistency solver on the problem above, after bounds_consistent returns

$$D(X) = [1..2], \quad D(Y) = [2..3], \quad D(Z) = [3..4]$$

the solver selects a variable, say Z, and tries different values in its domain. First, the complete bounds consistency solver calls itself recursively with the constraint $X < Y \wedge Y < Z \wedge Z = 3$ and the domain

$$D(X) = [1..2], \quad D(Y) = [2..3], \quad D(Z) = [3..4].$$

The procedure bounds_consistent is evaluated with this new constraint and domain. Examining $Z = 3$ gives $D(Z) = [3..3]$. Now examining $Y < Z$ gives $D(Y) = [2..2]$. Finally, examining $X < Y$ gives $D(X) = [1..1]$. No more propagation results, so the resulting domain is

$$D(X) = [1..1], \quad D(Y) = [2..2], \quad D(Z) = [3..3].$$

This is returned by bounds_consistent and, since this is a valuation domain which satisfies the constraint, the solver returns *true*.

Example 3.7
Consider the smuggler's knapsack problem from Example 3.5 which has constraint

$$4W + 3P + 2C \leq 9 \ \wedge 15W + 10P + 7C \geq 30$$

and initial domain

$$D(W) = [0..9], \quad D(P) = [0..9], \quad D(C) = [0..9].$$

Using the complete bounds propagation solver we begin by applying the bounds_consistent procedure to this problem. This gives the domain

$$D(W) = [0..2], \quad D(P) = [0..3], \quad D(C) = [0..4].$$

Then, choosing to branch on W, we first try adding $W = 0$. Applying bounds_consistent returns

$$D(W) = [0..0], \quad D(P) = [1..3], \quad D(C) = [0..3].$$

Now, choosing to branch on P, we add the constraint $P = 1$. Applying bounds consistency gives rise to the domain

$$D(W) = [0..0], \quad D(P) = [1..1], \quad D(C) = [3..3],$$

and so we have found a solution to the problem: $\{W \mapsto 0, P \mapsto 1, C \mapsto 3\}$. A diagram of the search tree is shown in Figure 3.14. The solver explores the left part of the tree until the first solution is found. Notice that if all solutions were required we could modify the solver to explore the entire tree, and return each solution (leaf nodes that are not $false$). Also note that the size of the search tree is significantly smaller than that obtained if the chronological backtracking solver given in Algorithm 3.1 is used.

3.5 Generalized Consistency

We have now seen three consistency-based approaches to solving CSPs, namely arc, node and bounds consistency. In this section we consider how these can be combined with each other and also with specialized consistency methods for "complex" primitive constraints.

It is straightforward to use different consistency methods in combination with one another, since they communicate through a common medium, the domain. In particular, we can combine arc and node consistency with bounds consistency. For constraints involving only two variables we can use the stronger arc consistency test to remove values, while, for constraints with more than two variables, we can use the weaker, but more efficiently computable, bounds consistency approach.

The reason for combining these approaches is that each is better in certain circumstances. Clearly, in the case that the primitive constraints have more than two variables, bounds consistency will force more domain pruning since arc and node consistency is satisfied by any domain. Conversely, with constraints involving only two variables, arc consistency may perform more domain pruning than bounds consistency. For instance, consider the constraint $X^2 = 1 - Y^2 \wedge X \neq 0 \wedge Y \neq 0$ with the domain $D(X) = D(Y) = \{-1, 0, 1\}$. The original domain is bounds consistent but not arc consistent. Application of the arc consistency algorithm will determine that the constraint is unsatisfiable.

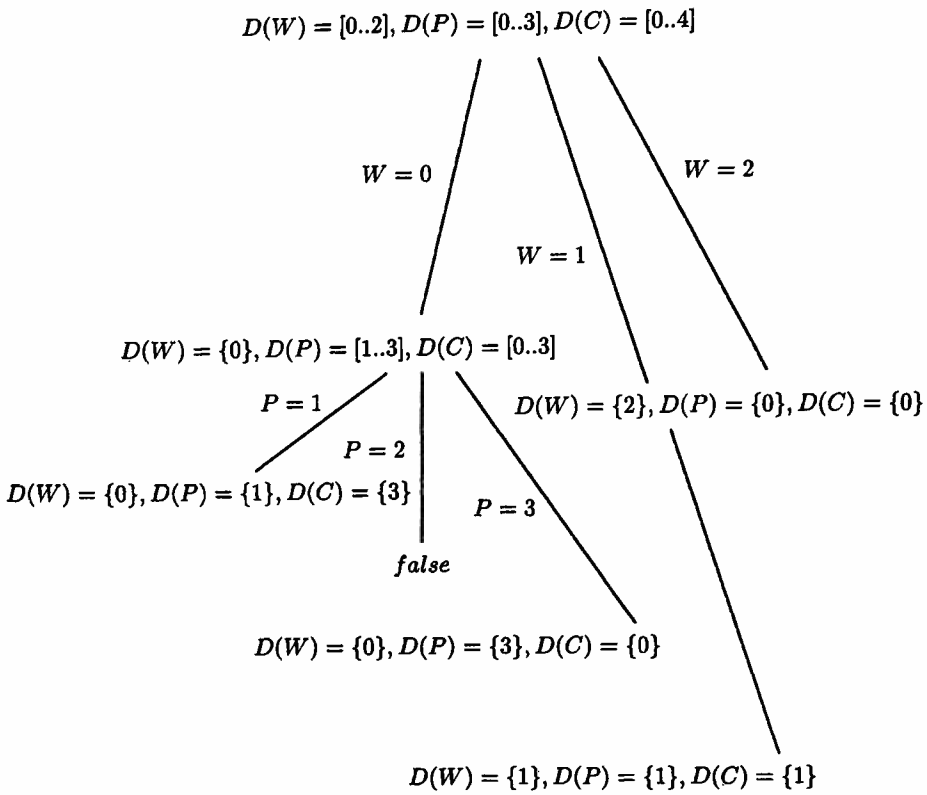

Figure 3.14 Search tree for the knapsack example.

It is easy to modify the bounds consistency solver so as to also employ arc and node consistency. We simply replace the call to bounds_consistent_primitive by a procedure that uses node_consistent_primitive for primitive constraints with one variable, arc_consistent_primitive for primitive constraints with two variables and bounds_consistent_primitive for primitive constraints with more than two variables.

One weakness of consistency based approaches is that primitive constraints are examined in isolation from each other. Sometimes knowledge about other primitive constraints can dramatically improve domain pruning. For this reason it is common to provide "complex" primitive constraints which are understood as a conjunction of simpler primitive constraints but which have specialized propagation rules.

Consider a specialized "primitive" constraint `alldifferent`$(\{V_1, \ldots, V_n\})$ which holds whenever each of the variables V_1 to V_n in its argument takes a different value. Instead of this constraint we could use a conjunction of disequality constraints. For example, the primitive constraint `alldifferent`$(\{X, Y, Z\})$ can be replaced by $X \neq Y \wedge X \neq Z \wedge Y \neq Z$. The disadvantage of using a conjunction of disequalities is that arc consistency methods for such a constraint are quite weak since the disequalities are considered in isolation. For instance, the constraint

$$X \neq Y \wedge X \neq Z \wedge Y \neq Z$$

```
c is a single alldifferent primitive constraint;
D is a domain;
V is a set of variables;
v is a variable;
d is a domain value;
r is a set of values;
and nv is an integer.

alldifferent_consistent_primitive(c,D)
   let c be of form alldifferent(V)
   while exists v ∈ V with D(v) = {d} for some d
      V := V − {v}
      for each v' ∈ V
         D(v') := D(v') − {d}
      endfor
   endwhile
   nv := |V|
   r := ∅
   for each v ∈ V
      r := r ∪ D(v)
   endfor
   if nv > |r| then return false endif
   return D
```

Figure 3.15 Consistency method for the `alldifferent` primitive constraint.

with domain

$$D(X) = \{1,2\}, \quad D(Y) = \{1,2\}, \quad D(Z) = \{1,2\}$$

has no solutions, since there are only two possible values for the three variables to take but arc consistency techniques cannot determine this.

Specialized propagation rules for the primitive constraint `alldifferent` can detect such situations, and so will find failure more often. An algorithm for updating the domains of variables involved in an `alldifferent` constraint is shown in Figure 3.15. Basically the algorithm works by removing the values of any fixed variables from the domains of other variables and checking that the number of unfixed variables, nv, does not exceed the total number of values left to variables which are not fixed, $|r|$.

For example, on the constraint `alldifferent`$(\{X, Y, Z\})$ and domain

$$D(X) = \{1,2\}, \quad D(Y) = \{1,2\}, \quad D(Z) = \{1,2\}$$

the consistency algorithm calculates $nv = 3$ and $r = \{1,2\}$. It therefore returns *false*.

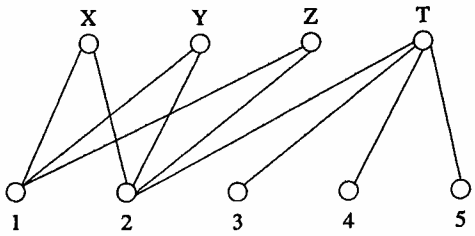

Figure 3.16 Matching variables against values.

Note that the algorithm is still not complete. For instance, consider the constraint `alldifferent`$(\{X, Y, Z, T\})$ with domain

$$D(X) = \{1,2\}, \quad D(Y) = \{1,2\}, \quad D(Z) = \{1,2\}, \quad D(T) = \{2,3,4,5\}.$$

There is no assignment to X, Y and Z which satisfies the constraint, but this is hidden by the large domain of the variable T.

One way to write a more complete consistency checker for `alldifferent` is to recognize that the constraint essentially represents a maximal bipartite matching between variables and values in which each variable must be "matched" with a different value. For example, the constraint `alldifferent`$(\{X, Y, Z, T\})$ with the domain above can be represented by the bipartite matching problem shown in Figure 3.16. A maximal bipartite matching algorithm for this graph will determine a maximal matching size of 3 indicating that at most three variables can be matched and so the constraint is unsatisfiable. However, we defer a discussion of how to find a maximal bipartite matching until the notes at the end of this chapter.

Introducing new "complex" primitive constraints with specialized consistency methods can greatly improve efficiency for solving real-life problems. Another example of a complex primitive constraint which is useful in many practical applications is `cumulative`. This was introduced for solving scheduling and placement problems. The constraint

$$\texttt{cumulative}([S_1, \ldots, S_m], [D_1, \ldots, D_m], [R_1, \ldots, R_m], L)$$

constrains the variables to satisfy a simple scheduling problem. There are m tasks to schedule whose start times are $S_1, \ldots, S_m$ and durations are $D_1, \ldots, D_m$, and which require $R_1, \ldots, R_m$ units of a single resource. At most L units of the resource are available at any one time. Consistency methods for cumulative constraints are very complex since there are numerous possibilities for inferring new information depending on which of the variables are known. In the following example we illustrate only one simple case.

Example 3.8

Bernd is moving house again and must schedule the removal of his furniture. Only he and three of his friends are available for the move and the move must be completed in one hour. The following list details the items of furniture which must be moved, how long each takes to move and how many people are required. For example,

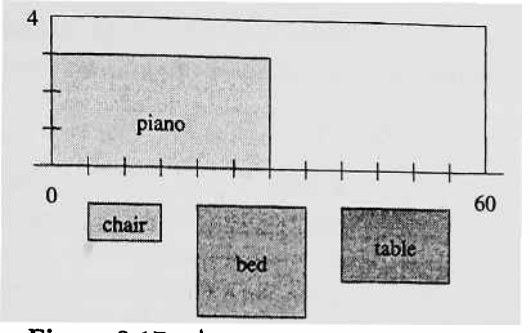

Figure 3.17 A `cumulative` constraint.

moving the piano requires 3 people and takes 30 minutes, while moving a chair requires 1 person and takes 10 minutes.

Item	Time required to move	No. of people required
piano	30	3
chair	10	1
bed	15	3
table	15	2

This problem can be modelled by the cumulative constraint

$$\texttt{cumulative}([S_p, S_c, S_b, S_t], [30, 10, 15, 15], [3, 1, 3, 2], 4)$$

where the variables S_p, S_c, S_b, S_t represent the start times for moving the piano, chair, bed and table, respectively. The initial domain for these variables is

$$D(S_p) = [0..30], \quad D(S_c) = [0..50], \quad D(S_b) = [0..45], \quad D(S_t) = [0..45]$$

which was obtained by computing the latest time that the furniture could be moved and still meet the one hour deadline.

With this initial domain the `cumulative` constraint consistency methods cannot determine any extra information about the variables. Now suppose Bernd decides to move the piano first. Then $S_p = 0$. The situation is displayed graphically in Figure 3.17. The remaining tasks (boxes) must be placed in the empty space so that they do not overlap. Clearly, the bed and table cannot be moved before the piano is moved so consistency can determine that $D(O_b) = [30..45]$ and $D(O_t) = [30..45]$.

Now suppose the furniture has to be moved within 50 minutes. Then reasoning about consistency reveals that no two of the piano, bed or table can be moved at the same time. Thus the minimum time required to move the furniture is 60 minutes. This demonstrates how specialized reasoning about consistency of the `cumulative` constraint can determine that the constraint is unsatisfiable without requiring exhaustive search through the possible time assignments for each task.

Our final example of a complex primitive constraint is the `element` constraint. The `element` constraint mimics the behaviour of an array. The constraint

$$element(I, [V_1, \ldots, V_m], X)$$

maintains the relationship that, if $I = i$, then $X = V_i$. We can understand this by thinking of the list $[V_1, \ldots, V_m]$ as an array v, and the constraint as stating that $X = v[I]$.

We can use this primitive to enforce ad-hoc functional relationships. For instance, suppose X must be equal to $I^2 - 3$ where the domain of I is [0..5]. This is described by the constraint `element(I, [-3, -2, 1, 6, 13, 22], X)`. Consistency techniques for `element` constraints can reduce the domain of I from information about X and vice versa. For example, given the above `element` constraint and a domain of X equal to [-20..20] consistency reasoning immediately reduces the domain of X to $\{-3, -2, 1, 6, 13\}$ and the domain of I to [0..4].

Example 3.9

The `element` constraint gives us another way of modelling the smuggler's knapsack problem of Example 3.5. Given the capacity constraints there can be at most five items in the knapsack. We can encode the type of each of these items as 1: no object, 2: whiskey, 3: perfume and 4: cigarettes. Each variable $O_i, 1 \leq i \leq 5$ represents the type of the i^{th} object and has domain [1..4]. We then use `element` constraints to relate the type to a weight and profit for that element. The array of weights is [0, 4, 3, 2], and the array of profits is [0, 15, 10, 7]. The following model determines the total weight and total profit of the objects and ensures the constraints hold.

$$\begin{aligned}
&element(O_1, [0,4,3,2], W_1) \wedge element(O_1, [0,15,10,7], P_1) \wedge \\
&element(O_2, [0,4,3,2], W_2) \wedge element(O_2, [0,15,10,7], P_2) \wedge \\
&element(O_3, [0,4,3,2], W_3) \wedge element(O_3, [0,15,10,7], P_3) \wedge \\
&element(O_4, [0,4,3,2], W_4) \wedge element(O_4, [0,15,10,7], P_4) \wedge \\
&element(O_5, [0,4,3,2], W_5) \wedge element(O_5, [0,15,10,7], P_5) \wedge \\
&W_1 + W_2 + W_3 + W_4 + W_5 \leq 9 \wedge P_1 + P_2 + P_3 + P_4 + P_5 \geq 30
\end{aligned}$$

For this problem the model using `element` is more complex than the original model and so is not as good. But for other problems modelling with `element` may be useful.

3.6 Optimization for Arithmetic CSPs

So far in this chapter we have considered constraint solvers for constraints over finite constraint domains, and in particular for arithmetic CSPs. While there are many problems in which finding any solution at all answers the problem, for many arithmetic CSPs the aim is not simply to find a solution but rather to find an

```
C is an arithmetic constraint;
D is a domain;
D_val is a valuation domain or false;
f is an arithmetic expression;
θ is a solution;
θ_best is either a solution or false.

retry_int_opt(C, D, f, θ_best)
    D_val := int_solv(C, D)
    if D_val ≡ false then
        return θ_best
    else
        let θ be the solution corresponding to D_val
        return retry_int_opt(C ∧ f < θ(f), D, f, θ)
    endif
```

Figure 3.18 Integer optimizer based on retry.

optimal (or at least a good) solution. In this section we examine how to solve such integer optimization problems. We first show how to use the arithmetic CSP solvers described earlier in the chapter to find an optimal solution. Finally, we examine an approach from the operations research community which makes use of algorithms for linear arithmetic constraint solving over the real numbers.

The simplest approach to finding an optimal solution to an arithmetic CSP is to make use of a complete solver for these problems and use it iteratively to find better and better solutions to the problems. More exactly, we can use the solver to find any solution to the CSP, and then add a constraint to the problem which excludes solutions that are not better than this solution. The new CSP is solved recursively, giving rise to a solution which is closer to the optimum. This process can be repeated until the augmented CSP is unsatisfiable, in which case the optimal solution is the last solution found. At each stage we keep track of the best solution found so far, θ_{best}. Initially this is set to $false$ indicating that no solutions have been found so far.

Algorithm 3.9: Integer optimizer based on retrying
INPUT: An arithmetic CSP with constraint C and domain D and an arithmetic expression f which is the objective function.
OUTPUT: An optimal solution θ or $false$ if the CSP is unsatisfiable.
METHOD: The answer is the result of evaluating retry_int_opt(C, D, f, $false$). The algorithm is shown in Figure 3.18. It assumes that int_solv is an algorithm for arithmetic CSPs (such as back_bounds_solv) which returns either a valuation domain corresponding to a solution or $false$ if the CSP is unsatisfiable. □

Example 3.10
Consider the smuggler's knapsack problem from Example 3.5. Ideally the smuggler wishes to maximize the profit made on a journey. Thus the problem is more properly

viewed as an optimization problem in which the smuggler wishes to maximize his profit, $15W + 10P + 7C$, subject to the constraint

$$4W+3P+2C \leq 9 \ \wedge \ 15W+10P+7C \geq 30 \ \wedge \ W \in [0..9] \ \wedge \ P \in [0..9] \ \wedge \ C \in [0..9].$$

Couched as a minimization problem, the smuggler's aim is to minimize the loss, $-15W - 10P - 7C$, subject to

$$4W+3P+2C \leq 9 \ \wedge \ 15W+10P+7C \geq 30 \ \wedge \ W \in [0..9] \ \wedge \ P \in [0..9] \ \wedge \ C \in [0..9].$$

Execution of the integer optimization algorithm using retrying on this optimization problem proceeds as follows. First back_bounds_solv is called to solve the original CSP. As detailed in Example 3.7, this traverses the tree shown in Figure 3.14 in a depth-first left-to-right manner. The first solution found is the valuation domain $D(W) = \{0\}, D(P) = \{1\}, D(C) = \{3\}$ which corresponds to the solution $\{W \mapsto 0, P \mapsto 1, C \mapsto 3\}$. This has a loss of -31 (or profit of 31).

Now the problem obtained by adding the constraint $-15W - 10P - 7C < -31$ is solved. This causes exploration of a search tree similar to that shown in Figure 3.14. Initial propagation gives the same domain

$$D(W) = [0..2], \quad D(P) = [0..3], \quad D(C) = [0..4].$$

However, this time after choosing $W = 0$ propagation leads to a false domain, so $W = 1$ is tried. In this case propagation finds the valuation domain $D(W) = \{1\}, D(P) = \{1\}, D(C) = \{1\}$ which has loss -32.

Now the problem obtained by adding the constraint $-15W - 10P - 7C < -32$ is solved. Initial propagation gives the same domain as above but propagation after choosing $W = 0$, $W = 1$ and $W = 2$ gives false domains. Therefore, back_bounds_solv returns *false* for this problem.

The optimization algorithm therefore terminates, returning the previous best solution, $\{W \mapsto 1, P \mapsto 1, C \mapsto 1\}$, since this is the optimal solution. Overall there are 10 visits to nodes in the various search trees.

The problem with this approach to optimization is that, given a solver that uses backtracking to find a solution to the problem—as most complete solvers for arithmetic CSPs do, much of the search required to find a solution to the problem is essentially repeated when solving the new problem resulting from the tighter bound on the optimization variable. Instead, if we are using some form of consistency based solver with backtracking, we can interleave the process of finding an optimal solution with the search for the next solution. In this way, we only traverse the search tree once instead of many times.

Algorithm 3.10: Backtracking integer optimizer
INPUT: An arithmetic CSP with constraint C and domain D and an objective function f.
OUTPUT: An optimal solution θ or *false* if the CSP is unsatisfiable.
METHOD: The answer is the result of evaluating back_int_opt(C, D, f, *false*)

```
C is an arithmetic constraint;
D is a domain;
f is an arithmetic expression;
and θ_best is a solution or false.

back_int_opt(C, D, f, θ_best)
   D := int_consistent(C, D)
   if D is a false domain then return θ_best
   elseif D is a valuation domain then
      return the solution corresponding to D
   endif
   choose variable x ∈ vars(C) for which |D(x)| ≥ 2
   W := D(x)
   for each d ∈ W
      if θ_best ≢ false then
         c := f < θ_best(f)
      else
         c := true
      endif
      θ_best := back_int_opt(C ∧ c ∧ x = d, D, f, θ_best)
   endfor
   return θ_best
```

Figure 3.19 Backtracking integer optimizer.

The algorithm is shown in Figure 3.19. It assumes that int_consistent(C,D) is a consistency based solver which returns a domain D_1 such that C with D has the same set of solutions as C with D_1. □

Example 3.11

Consider the optimization version of the smuggler's knapsack problem again. Execution of this optimization problem using the backtracking integer optimizer searches the tree shown in Figure 3.14 in a depth-first left-to-right manner. As before, the first solution found is $\{W \mapsto 0, P \mapsto 1, C \mapsto 3\}$, which has a loss of -31 (or profit of 31). θ_{best} is set to $\{W \mapsto 0, P \mapsto 1, C \mapsto 3\}$.

Backtracking returns to the node labelled $D(W) = \{0\}, D(P) = [1..3], D(C) = [0..4]$ and the new constraint c is set to $-15W - 10P - 7C < -31$ and added. Both $P = 2$ and $P = 3$ are tried. In each case propagation gives rise to a false domain. Note that $P = 3$ in the original search tree gave rise to a successful leaf $D(W) = \{0\}, D(P) = \{3\}, D(C) = \{0\}$. This now fails because its profit is 30.

Backtracking proceeds up to the node with domain $D(W) = [0..2]$, $D(P) = [0..3]$, $D(C) = [0..4]$ with θ_{best} still set to $\{W \mapsto 0, P \mapsto 1, C \mapsto 3\}$. Now $W = 1$ is tried. Propagation gives the valuation domain $D(W) = \{1\}, D(P) = \{1\}, D(C) = \{1\}$ which has loss -32. θ_{best} is set to $\{W \mapsto 1, P \mapsto 1, C \mapsto 1\}$ and $-15W - 10P - 7C < -32$ is added to the problem. Backtracking continues, trying $W = 2$ but propagation gives a false domain.

Therefore, since the entire search tree has been explored, the algorithm terminates

returning the optimal solution $\{W \mapsto 1, P \mapsto 1, C \mapsto 1\}$. Overall the algorithm visits each node in the tree given in Figure 3.14 once, meaning that only 7 nodes are visited.

Both of these algorithms for integer optimization are naive in the sense that they only use the objective function to eliminate solutions which are not better than the best solution already discovered. In particular, they do not use the objective function to direct the search to that part of the solution space in which it is likely to find better solutions. This contrasts to optimization algorithms for the real numbers, such as the simplex algorithm (see Section 2.4), in which the objective function is used to direct the search. In a sense, this is made possible because it is relatively easy in linear arithmetic problems over the real numbers to move from one solution to another. When the linear arithmetic problem is over integers, directing the search is more difficult since it may be difficult to move from one solution to another. However, we can use a real optimizer which does employ directed search to guide the search for an optimal solution over the integers.

One of the oldest algorithms for solving integer optimization problems does this, making use of an optimizer working over the real numbers to direct the search. Consider an optimization problem (C, f) over the integers and an optimizer real_opt which can solve optimization problems over real numbers and which returns an optimal (that is, minimal) solution to a particular problem. Now, if real_opt(C, f) returns the solution θ then the optimal solution over the integers, θ', must satisfy $\theta'(f) \geq \theta(f)$ since, if θ' was preferable to θ, then θ could not be the optimal solution over the real numbers.

This observation leads to the branch and bound solver. To solve the optimization problem (C, f) over the integers, the optimizer real_opt is used to find an optimal solution over the real numbers. If the optimal solution found is *integral*, that is, assigns integers to all variables, we are finished since, from the above observation, this must be the optimal solution over the integers. If the optimal solution returned by real_opt is not integral, the branch and bound solver must search. This is done by choosing an variable x which takes a non-integral value, say d, in the optimal solution. The two optimization problems $(C \wedge x \leq \lfloor d \rfloor, f)$ and $(C \wedge x \geq \lceil d \rceil, f)$ are recursively evaluated. Note that $\lfloor d \rfloor$ is the d rounded down to the nearest integer, while $\lceil d \rceil$ is d rounded up to the nearest integer.

The optimal solution to the original problem must be the optimal solution to one of these problems since they eliminate no integer solutions. An additional improvement is made possible by noticing that the optimal value over the real numbers can be no worse than the optimal solution over the integers. Hence, if we have already found an integer solution θ_{best}, then any subproblem with optimal solution θ over the real numbers such that $\theta(f) \geq \theta_{best}(f)$ can be discarded since it can never lead to a better integer solution than the current best, θ_{best}.

The algorithm is called a "branch and bound" algorithm because it selects a variable x to branch on, and then investigates two problems in which this variable is further bounded.

```
d is a real value;
x is a variable;
f is an integer expression;
θ and θ_best are valuations or false;
C is an integer constraint.

bb_int_opt(C,f,θ_best)
    θ := real_opt(C, f)
    if θ ≡ false then return θ_best endif
    if θ_best ≢ false and θ(f) > θ_best(f) then return θ_best endif
    if θ is integral then return θ endif
    choose variable x such that d = θ(x) is not an integer
    θ_best := bb_int_opt(C ∧ x ≤ ⌊d⌋, f, θ_best)
    θ_best := bb_int_opt(C ∧ x ≥ ⌈d⌉, f, θ_best)
    return θ_best
```

Figure 3.20 Integer optimization using a real optimizer.

Algorithm 3.11: Branch and bound integer optimizer
INPUT: A linear arithmetic constraint C and objective function f.
OUTPUT: An optimal solution θ or $false$ if no solution exists.
METHOD: The answer is the result of the call bb_int_opt(C, f, $false$) The algorithm is shown in Figure 3.20. It assumes that real_opt(C,f) is an optimizer for arithmetic constraints that either returns $false$ if the problem is unsatisfiable or an optimal solution over the real numbers. □

Example 3.12
For example, consider maximising Y, that is minimising the expression f where f is $-Y$, subject to the constraint C which is

$$Y \leq X \wedge Y \leq 8 - 2X \wedge X - Y + 1 = 3Z.$$

A diagram of the (X, Y) plane for this constraint is shown in Figure 3.21. The two constraints on X and Y are shown by solid lines, while the constraint $X-Y+1 = 3Z$ has the effect of eliminating all but the shaded possibilities for X and Y.

Initially θ_{best} is $false$ and real_opt(C, f) returns the intersection point of the two solid lines $\{X \mapsto \frac{8}{3}, Y \mapsto \frac{8}{3}, Z \mapsto \frac{1}{3}\}$. Splitting on X, the next call is

$$\text{bb\_int\_opt}(C \wedge X \leq 2, f, \theta_{best}).$$

The real optimizer returns $\{X \mapsto 2, Y \mapsto 2, Z \mapsto \frac{1}{3}\}$. Splitting on Z results in the call

$$\text{bb\_int\_opt}(C \wedge X \leq 2 \wedge Z \leq 0, f, \theta_{best}).$$

The real optimizer fails, so this call immediately returns θ_{best} which is still set to

false. The other half of the split for Z results in the call

$$\mathsf{bb\_int\_opt}(C \wedge X \leq 2 \wedge Z \geq 1, f, \theta_{best}).$$

For this call the real optimizer returns θ_1 which is $\{X \mapsto 2, Y \mapsto 0, Z \mapsto 1\}$. This is returned as the answer and so θ_{best} is set to θ_1.

The remaining half of the split for X is examined by the call

$$\mathsf{bb\_int\_opt}(C \wedge X \geq 3, f, \theta_{best}).$$

The real optimizer finds the optimal solution is $\{X \mapsto 3, Y \mapsto 2, Z \mapsto \frac{2}{3}\}$. Again we split on Z, first calling

$$\mathsf{bb\_int\_opt}(C \wedge X \geq 3 \wedge Z \leq 0, f, \theta_{best}).$$

This immediately fails, so now we call

$$\mathsf{bb\_int\_opt}(C \wedge X \geq 3 \wedge Z \geq 1, f, \theta_{best}).$$

The optimal solution over the reals is $\{X \mapsto \frac{10}{3}, Y \mapsto \frac{4}{3}, Z \mapsto 1\}$. Again we split on X. The first call is

$$\mathsf{bb\_int\_opt}(C \wedge X \geq 3 \wedge Z \geq 1 \wedge X \leq 3, f, \theta_{best}).$$

The real optimizer returns θ_2 which is $\{X \mapsto 3, Y \mapsto 1, Z \mapsto 1\}$. As this is integral it is returned and θ_{best} set to θ_2.

The remaining call for the split on X is then

$$\mathsf{bb\_int\_opt}(C \wedge X \geq 3 \wedge Z \geq 1 \wedge X \geq 4, f, \theta_{best}).$$

The optimal solution over the reals is θ_3 which is $\{X \mapsto 4, Y \mapsto 0, Z \mapsto \frac{5}{3}\}$. Since $\theta_3(f) > \theta_{best}(f)$ we cannot hope to find a solution in this part of the solution space which is better than that already found so the call returns the current θ_{best}.

3.7 Summary

Constraints in which variables range over finite domains form a well studied and useful class of constraint problems which are often called constraint satisfaction problems (CSPs) in the artificial intelligence literature. They are important because they can be used to model combinatorial problems, such as scheduling or timetabling, which have widespread commercial applications.

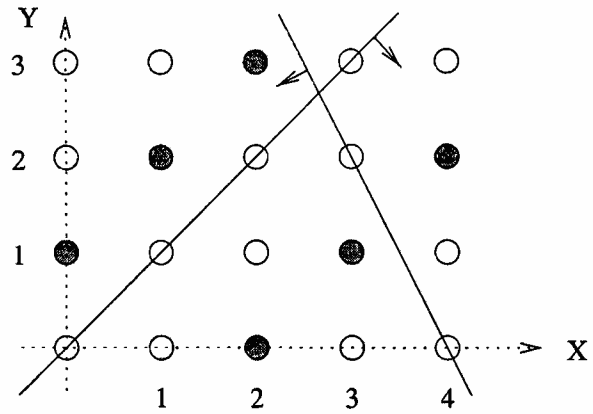

Figure 3.21 Diagram of (X,Y) space for integer optimization problem

We have considered two general techniques for solving finite domain problems:

- backtracking search, and
- consistency techniques.

Backtracking search gives rise to complete solvers with exponential worst-case cost, while consistency methods give rise to fast solvers which are, however, incomplete. We have examined three kinds of consistency: node and arc consistency, which arose from the artificial intelligence community, and bounds consistency, which arose from the constraint programming community. Completeness of consistency based approaches can be improved by combining different kinds of consistency and by providing specialized consistency techniques for complex primitive constraints such as `alldifferent`, `cumulative` and `element`. We have also detailed how backtracking can be combined with consistency techniques to give faster complete solvers for finite domain problems.

In many applications of finite domain constraints, such as scheduling, it is important to be able to find the best solution. We have shown how a complete constraint solver can be used as the basis for solving such optimization problems and have given two algorithms based on this idea. The first repeatedly solves a new problem, which only has solutions that are better than the current solution to the optimization problem. The second intermingles a backtracking search for all solutions to the problem with the search for an optimal solution. Finally, we have given an algorithm from the operations research community for solving such optimization problems. This approach, called branch and bound, makes use of a solver over the real numbers to direct the search towards an optimal solution over the integers.

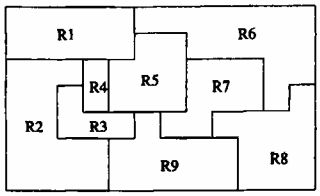

Figure 3.22 A map for colouring.

3.8 Exercises

3.1. For the map given in Figure 3.22 and the three colours $\{red, yellow, blue\}$ encode the map colouring problem for this map as a binary CSP. Give the constraint graph for this constraint problem and find a solution.

3.2. Another way to model the 4-queens problem is to assume that each queen must appear in a separate column. In this model the position of the queen in column i is given by a variable R_i which holds the row number in which the queen occurs. Using this idea give an encoding of the 4-queens problem that only uses binary constraints.

3.3. Consider the old-fashioned marriage problem in Example 3.3. Solve the problem using back_arc_solv.

3.4. Give a probabilistic solver for CSPs based on the Boolean constraint solver of Figure 1.10. Use it solve the old-fashioned marriage problem of Example 3.3.

3.5. Give propagation rules for the linear arithmetic constraint

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n - b_1y_1 - b_2y_2 - \cdots - b_my_m = d$$

and

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n - b_1y_1 - b_2y_2 - \cdots - b_my_m \leq d$$

where the constants $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ are all positive.

3.6. (*) Prove that the propagation rules for the above example need only to be applied once to give a bounds consistent domain.

3.7. Give propagation rules for the primitive constraint $x = y \mod k$ where k is a fixed but unknown integer.

3.8. Give propagation rules for the primitive constraint $X = Y^2$ which will give more propagation information than using the rules for $X = Y \times Y$.

3.9. Give propagation rules for the Boolean primitive constraints: $X = Y \,\&\, Z$, $X = Y \vee Z$ and $X = \neg Y$. You should assume that every variable x appearing in a Boolean constraint has a domain $D(x) \subseteq \{0, 1\}$.

3.10. Compare the execution of arc_solv and bounds_solv on the CSP with constraint $X \neq 4 \wedge X \neq Y \wedge X \leq Y + 2$ and domain $D(X) = [2..5]$, $D(Y) = [0..5]$.

3.11. Execute the complete bounds consistency solver on the constraint

$$X \neq Y \wedge Y \neq 1 \wedge X = Y \times Y \wedge Y \leq -1$$

with domain $D(X) = [-2..6]$, $D(Y) = [-4..4]$.

3.12. (*) Assume the propagation rules for the multiplication constraint $X = Y \times Z$ are changed to wait until either Y or Z has a fixed value (that is, either X or Y has a singleton domain) before determining any bounds for Y or Z. Argue whether bounds_solv ensures bounds consistency or not.

3.13. Consider solving the smuggler's knapsack problem using back_int_opt. Show the 4 different search trees of calls to back_int_opt (analogous to Figure 3.14) that result when the variables are selected in the order W, P, C or in the order C, P, W and when the domain values are tried in increasing order or in decreasing order.

3.14. Consider a modification of the smuggler's knapsack problem in which the capacity of the knapsack is increased to 17 units. Find the maximum profit and the items which give this profit using either retry_int_opt or back_int_opt.

3.9 Practical Exercises

There are a number of finite domain constraint solvers available as part of constraint logic programming languages. The ECLiPSe system, developed at ECRC, is (at the time of publication) available free for academic purposes. It is a PROLOG system which includes a library for finite domain constraint solving. It can be obtained by sending email to `eclipse-request@doc.ic.ac.uk`. See also the Eclipse WWW home pages at `http://www.ecrc.de/eclipse/` and `http://www-icparc.doc.ic.ac.uk/eclipse/`. Using the ECLiPSe system as a finite domain constraint solver is similar to using $CLP(\mathcal{R})$ for real number constraint solving.

For example the ECLiPSe system presents a prompt to the user

```
[eclipse 1]:
```

To invoke the finite domain constraint solver the user types what is shown in *italics*.

```
[eclipse 1]: use_module(library(fd)).
```

Constraints are entered using the following conventions: * is used for multiplication, linear expressions need an explicit multiplication symbol, so $2Y$ must be written as `2 * Y`, $\wedge$ is represented using "," and the operators $=$, $\geq$, $>$, $\leq$, $<$ and $\neq$ are represented by "#=", "#>=", "#>", "#<=", "#<" and "##" respectively, and finally a full stop "." is used to terminate the constraint. To interrupt execution the user can type *control-C*, that is type 'c' while holding down the control key.

The initial domain of the variables is specified using the following syntax. To set the variable X to have initial domain $\{1,2,3,4\}$ either the form `X :: [1,2,3,4]` or

`X :: [1..4]` can be used. Multiple variables can be initialised to the same domain using the syntax of the following example: `[X,Y,Z] :: [1..4]`. If no initial domain is given for an integer variable it is assumed to be [−10000000..10000000].

The constraint solver in ECLiPSe has approximately the same behaviour as the incomplete bounds consistency solver bounds_solv described in Algorithm 3.7. For example, if the following constraint is typed,

```
[eclipse 2]: [X,Y,Z] :: [1..4], X #< Y, Y #< Z.
```

the resulting answer is

```
X = X{[1, 2]}
Y = Y{[2, 3]}
Z = Z{[3, 4]}
Delayed goals:
      Z{[3, 4]} - Y{[2, 3]}#>=1
      Y{[2, 3]} - X{[1, 2]}#>=1
yes.
```

The initial part of the answer details the variables in the constraint and their current domains. The second part shows primitive constraints which are not guaranteed to be consistent with all valuations possible in the current domain. If any such constraints remain the solver has answered *unknown*. If there are none, the constraint solver has answered *true* as in the following examples:

```
[eclipse 3]: [X,Y,Z] :: [1..3], X #< Y, Y #< Z.
X = 1
Y = 2
Z = 3
```

and

```
[eclipse 4]: X :: [1,2], Y :: [3,4], X #< Y.
X = X{[1, 2]}
Y = Y{[3, 4]}
```

If the solver detects unsatisfiability then the result has the following form:

```
[eclipse 5]: [X,Y,Z] :: [1..3], X #< Y, Y #< Z, Z #<= 2.
no (more) solution.
```

A complete solver can be imitated using the inbuilt backtracking search of the ECLiPSe system. By appending the constraint to be solved with an expression of the form `labeling(`*Vars*`)` where *Vars* is the list of variables appearing in the constraint, the ECLiPSe system will work analogously to back_bounds_solv described in Algorithm 3.8. For example, evaluation of the constraint:

```
[eclipse 6]: [X,Y,Z] :: [1..4], X #< Y, Y #< Z, labeling([X,Y,Z]).
```

returns the answer

```
X = 1
Y = 2
Z = 3 More? (;)
```

The `More? (;)` prompts the user to ask if another solution is required. The user can type ";" or "y" to request another solution. In this case it is,

```
X = 1
Y = 2
Z = 4 More? (;)
```

Otherwise, evaluation finishes.

The SICStus Prolog v3 system also includes a library for finite domain constraints. How to obtain SICStus Prolog is discussed in the Practical Exercises section of Chapter 1. In order to make use of the finite domain constraint solving facilities of SICStus one needs to load the finite domain constraint solving library by typing:

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

Constraints are written using almost the same notation as that detailed above for ECLiPSe, except that the $\neq$ primitive constraint is written as "#\=" and the $\leq$ primitive constraint is written as "#=<." Be warned that #<= is legal SICStus code which means reverse implication. The initial domains of a variable can be specified by a declaration of the form `X in 1..4`. Multiple variables can be initialized to have the same domain using the library function `domain`, for example, the declaration

```
domain([X,Y,Z], 1, 4)
```

assigns the range [1..4] to each of the variables X, Y and Z.

The finite domain constraint solver in SICStus Prolog has again approximately the behaviour of the incomplete bounds consistency solver bounds_solv described in Algorithm 3.7. For example, if the following constraint is typed,

```
| ?- domain([X,Y,Z],1,4), X #< Y, Y #< Z.
```

the resulting output is

```
X in 1..2,
Y in 2..3,
Z in 3..4 ?
```

The answer simply shows the current domains of the variables. Section 29.6 of the SICStus Prolog Users Manual [58] shows how to change the behaviour of the system to output any constraints which are not guaranteed to be consistent. However, if the domain is a valuation domain then the corresponding valuation satisfies the constraint. This is exemplified in the following evaluation.

```
| ?- domain([X,Y,Z],1,3), X #< Y, Y #< Z.
X = 1
Y = 2
Z = 3 ?
```

If the solver detects unsatisfiability then the result has the following form:

```
| ?- domain([X,Y,Z],1,3), X #< Y, Y #< Z, Z #<= 2.
no
```

A complete solver can be imitated using the inbuilt backtracking search of the SICStus Prolog system. By appending the constraint to be solved with an expression of the form `labeling([], Vars)` where *Vars* is the list of variables appearing in the constraint the system will work analogously to back_bounds_solv. For example, the constraint:

```
| ?- domain([X,Y,Z],1,4), X #< Y, Y #< Z, labeling([], [X,Y,Z]).
```

returns the answer

```
X = 1
Y = 2
Z = 3 ?
```

The "?" prompts the user, asking if another solution is required. Typing ";" requests the next solution, in this case

```
X = 1
Y = 2
Z = 4 ?
```

Otherwise, typing Enter/Return terminates evaluation of the constraint. In the example above, after typing ";" four times the system returns no indicating that there are no more solutions.

P3.1. Type in the constraints for the 4-*queens* problem, and see what answer the solver gives. Add the `labeling` goal and examine the answers now.

P3.2. Solve the old-fashioned marriage problem using ECLiPSe or SICStus Prolog.

P3.3. Find a solution to the smuggler's knapsack problem using a complete solver.

P3.4. Examine how your system executes the CSP with constraint

$$X \neq 4 \wedge X \neq Y \wedge X = Y + 2$$

and domain $D(X) = [2..5]$, $D(Y) = [0..5]$. Determine if the solver is closer to `arc_solv` or `bounds_solv`. Experiment with similar constraints to improve your understanding.

P3.5. Examine the execution of the following goals:

(a) $X = Y \times Y \wedge 0 \leq Y \wedge Y \leq 2$,

(b) $X = Y \times Y \wedge 0 \leq X \wedge X \leq 2$,

(c) $X = Y \times Y \wedge Y = 2$ and

(d) $X = Y \times Z \wedge Z = 1 \wedge Y \leq 2$.

Can you determine the propagation rules that your system uses for the constraint $X = Y \times Z$?

P3.6. (*) Using your answer to Exercise 3.9 above, translate the Boolean constraint defining the full adder in Figure 1.9 into an integer constraint. Giving each variable an initial domain of `[0,1]` see what the complete solver gives as solutions of the constraint.

P3.7. Describe how, by hand, to use the finite domain system to imitate the behaviour of the retry_int_opt algorithm. Try it out on the smuggler's knapsack example.

3.10 Notes

CSPs as introduced in Section 1.7 date back to work in artificial intelligence, picture processing and computer vision. See for example Montanari [94] and Waltz [141]. Stefik [124] built a system for planning gene-splicing experiments, using consistency based methods.

It follows from the NP-hardness of SAT, one of the most famous NP-hard problems [53] that solving arbitrary CSPs is NP-hard. SAT is essentially the problem of finding if a Boolean formula is satisfiable or not. Thus it can be seen as a particular type of CSP.

Node and arc consistency originate from Montanari [94]. An in depth coverage of constraint satisfaction problems and their solution is provided by Tsang's book "Foundations of Constraint Satisfaction"[134]. This also describes more efficient algorithms for transforming a CSP into an equivalent arc consistent CSP. The book "Constraint Satisfaction and Logic Programming" by Van Hentenryck [135] also provides a theoretical introduction to different consistency methods. Bounds consistency has been introduced under many different names by various authors, for example [138] call it interval consistency.

The consistency based approaches described in this chapter are related to the local propagation method described in Section 1.7. In this view of local propagation, it is seen as a technique for reducing the domain to a single value, whenever this is possible. However, local propagation is relatively weak compared to the methods described here.

The tradeoff between stronger consistency methods, which remove more values, and weaker consistency methods, which are faster to compute but remove less values, is complex, especially when the consistency methods are being employed in

a complete backtracking solver. Since the optimal tradeoff may be problem specific, some finite domain solvers allow the user to define their own problem specific propagation rules for constraints. *Indexicals* introduced in [138] (see also [41]) are a language for defining propagation rules. Indexicals are of the form X in S where X is a finite domain variable and S is a expression denoting a set of values. For example,

$$X \text{ in } min(Y) + min(Z)..max(Y) + max(Z)$$

and

$$X \text{ in } dom(Y) + dom(Z)$$

are two indexicals defining the propagation rule for variable X and constraint $X = Y + Z$. The first implements bounds consistency, while the second implements arc consistency. A number of finite domain solvers are implemented by means of indexicals, for example `clp(FD)` [41] and SICStus Prolog [58].

The `element` constraint originated in the CHIP system [3], as does the `cumulative` constraint which is introduced in [2].

The `alldifferent` constraint is available in CHIP, ILOG Solver and SICStus Prolog. Implementations range from the simple consistency method illustrated in Figure 3.15 to methods based on maximal matching. Maximal bipartite matching algorithms are well studied, see for example [101]. Regin [111] gives an algorithm for maintaining hyper-arc consistency for `alldifferent` constraints. See also [107] for a discussion of `alldifferent`.

The branch and bound method for integer linear programming is due to Dakin. For detailed descriptions of branch and bound and other techniques to solve integer linear programming problems developed by the operations research community see, for example, Papadimitriou and Steiglitz [101] or Schrijver [116].

In this chapter we have concentrated on integer finite domain constraints. Another important class of finite domain constraints are those over floating point numbers. At first this seems paradoxical since floating point numbers are used to represent real numbers which clearly do not have a finite domain. However, in any particular implementation of floating point numbers, there are only a fixed number of floating point numbers. This means that the techniques described in this chapter are also applicable to solving arithmetic constraints over the floating point numbers. There are, however, a few provisos.

First, a floating point range, for instance $[1.14151, 1.14152]$, is meant to represent all the real numbers within this range, rather than just the floating point numbers in this range. This subtly alters the propagation rules. They need to round up or down to the "closest" floating point number, so as to ensure that no real solution is excluded. For example, the propagation rules for variable Y in primitive constraint $X = Y \times Z$ where $D(X) = [1..1]$, $D[Z] = [9..9]$ give $Y \leq 1/9$ and $Y \geq 1/9$. Since $1/9$ is not representable exactly as a floating point number, we need to take the nearest floating point number above, and below, respectively as defining the bounds

for Y. So the propagation rules (for the case where everything is positive) are

$$Y \leq fp\_ceil(max_D(X)/min_D(Z))$$
$$Y \geq fp\_floor(max_D(X)/min_D(Z))$$

where $fp\_ceil(r)$ gives the least floating point number bigger than real number r, and $fp\_ceil(r)$ gives the largest floating point number less than real number r. In general, propagation rules similar to those in this chapter can be used for solving constraints over floating point numbers, as long as care is taken with the direction of rounding in all floating point computations.

Second, floating point ranges include many elements, for example the range [1..2] may include 2^{32} floating point numbers. Therefore arc consistency methods are inappropriate, and domains are almost always ranges. Furthermore, any technique based on enumerating all possible elements in the domain is not practical. Instead floating point solvers rely on splitting the domain into pieces. See the discussion (for integers) in Section 8.3.

The use of floating point intervals for constraint solving was suggested by Cleary [31] and independently by Hyvönen [67]. The first implementation in a CLP system was in BNR Prolog, and is discussed in [98, 99].

In this chapter we have not mentioned an important class of methods used for solving constraint satisfaction problems—stochastic search methods. These combine heuristics with a non-deterministic search to find a solution or optimal solution to a CSP. Typically the algorithms proceed from one valuation to another attempting to find a solution. There is a wide variety of such methods including simulated annealing, genetic algorithms and local search.

We have not explored such methods because their use within a CLP system—our eventual goal for solvers—is not straightforward and the subject of current research. Methods for making use of stochastic constraint solvers in CLP systems are discussed in [129, 84]. Two general purpose stochastic methods which have been used with CLP systems are GENET [134] and E-GENET [85]. These are both artificial neural networks which make use of min-conflict search and heuristic learning.