2 Simplification, Optimization and Implication

In the preceding chapter, we studied various constraint domains and investigated how to determine satisfiability of constraints in these domains. In this chapter we look at other operations on constraints that are also of interest to the constraint programmer.

The first operation we study is simplification. This operation takes a system of constraints and simplifies it, so that information implicit in the original system becomes apparent. An important case of simplification is projection, that is simplifying constraints in terms of the variables that are of interest. In general this is a difficult and expensive operation. Simplification is important in constraint programming because of the need to present the answer constraints generated by a constraint program in a form which is understandable to the user.

Next, we examine optimization. In many constraint programming problems we would like to find the "best" answer where answers are ranked by some "objective function." Such problems are called optimization problems. We describe the simplex algorithm for solving linear real arithmetic optimization problems, that is a problem with a linear arithmetic constraint and a linear objective function, and show how it may also be used to efficiently determine satisfiability of a linear arithmetic constraint.

We conclude by investigating constraint implication, that is, whether one constraint has solutions which are a subset of the solutions of another constraint, and equivalence, that is, whether two constraints have exactly the same set of solutions. These operations are of interest for constraint databases and concurrent constraint programming.

2.1 Constraint Simplification

Although a large and complicated constraint may model some system or problem accurately it may be very difficult to understand. Sometimes the apparent complexity of the constraint is misleading and the same information can be expressed more succinctly. Simplification is the process of replacing a constraint by an equivalent constraint which has a "simpler" form.

The following list of arithmetic constraints are all equivalent. This is because changing the order of the primitive constraints, inverting a primitive constraint, substituting one variable for another by using an equation, performing simple arithmetic simplification or adding a "redundant" constraint do not change the set of solutions of a constraint.

$$\begin{aligned} & X \geq 3 \land 2 = Y + X \\ \leftrightarrow\ & 3 \leq X \land X = 2 - Y \\ \leftrightarrow\ & X = 2 - Y \land 3 \leq X \\ \leftrightarrow\ & X = 2 - Y \land 3 \leq 2 - Y \\ \leftrightarrow\ & X = 2 - Y \land Y \leq -1 \\ \leftrightarrow\ & X = 2 - Y \land Y \leq -1 \land Y \leq 2. \end{aligned}$$

The most obvious method of constraint simplification is to simplify the individual primitive constraints. For example, $0 \leq 3$ can be simplified to *true*, as can $(X + 1)^2 \geq X^2 + 2X$. Primitive constraints may have much simpler equivalent forms, for example $X - 2 + 2Y + 3 \geq 2X - Z - 4$ is equivalent to $X - 2Y - Z \leq 5$. Similarly the primitive term constraint $succ(succ(succ(X))) = succ(succ(Y))$ is equivalent to $succ(X) = Y$.

An important kind of simplification is to remove from a constraint C those primitive constraints which do not affect the meaning of C. In order to determine whether removing a (not necessarily primitive) constraint C_2 from $C_1 \land C_2$ will leave the meaning unchanged, we introduce the concept of "redundancy."

Definition 2.1
One constraint C_1 *implies* another C_2, written $C_1 \rightarrow C_2$, if the solutions of C_1 are a subset of the solutions of C_2. Alternatively, we say that constraint C_2 is *redundant* with respect to constraint C_1, if $C_1 \rightarrow C_2$.
A constraint C of the form $c_1 \land \cdots \land c_n$, where $c_1, \ldots, c_n$ are primitive constraints, is *redundancy-free*, if no c_i is redundant with respect to the remaining primitive constraints. That is, it is not the case that

$$(c_1 \land \cdots \land c_{i-1} \land c_{i+1} \land \cdots \land c_n) \rightarrow c_i.$$

For example, $X \geq 3$ is redundant with respect to $X \geq 4$. Therefore, we may simplify the constraint $X \geq 4 \land X \geq 3$ to give the equivalent constraint $X \geq 4$. Similarly, $Y \leq X + 2 \land Y \geq 4 \rightarrow X \geq 1$. Thus, $Y \leq X + 2 \land Y \geq 4 \land X \geq 1$ can be simplified to $Y \leq X + 2 \land Y \geq 4$. In the tree constraint domain,

$$cons(X, X) = cons(Z, nil) \rightarrow Z = nil.$$

Thus, $cons(X, X) = cons(Z, nil) \land Z = nil$ can be simplified to $cons(X, X) = cons(Z, nil)$.

It follows that $C_1 \rightarrow C_2$ if C_1 and $C_1 \land C_2$ are equivalent. Furthermore, every constraint is implied by the unsatisfiable constraint, *false*.

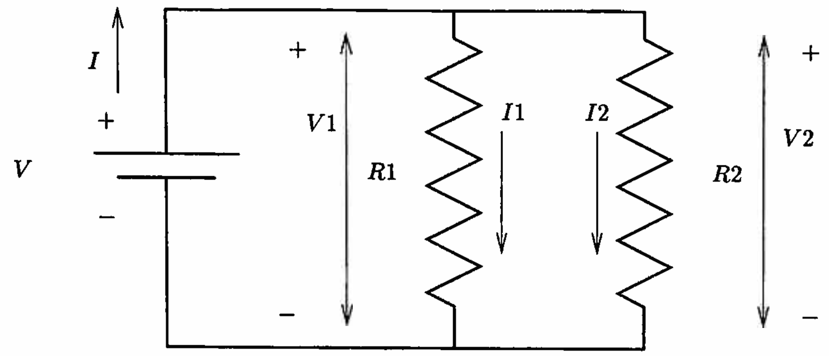

Figure 2.1 A simple electric circuit.

Normalising constraint solvers work by transforming one constraint to an equivalent constraint. They can also be seen as constraint simplifiers since the normal form is usually simpler than the original and some redundant information is removed. Consider the operation of unify on the constraint

$$cons(X, X) = cons(Z, nil) \land Y = succ(X) \land succ(Z) = Y \land Z = nil:$$

C	S
$\underline{cons(X, X) = cons(Z, nil)} \land Y = succ(X) \land succ(Z) = Y \land Z = nil$	$true$
$\underline{X = Z} \land X = nil \land Y = succ(X) \land succ(Z) = Y \land Z = nil$	$true$
$\underline{Z = nil} \land Y = succ(Z) \land succ(Z) = Y \land Z = nil$	$X = Z$
$\underline{Y = succ(nil)} \land succ(nil) = Y \land nil = nil$	$X = nil \land Z = nil$
$\underline{succ(nil) = succ(nil)} \land nil = nil$	$X = nil \land Z = nil \land Y = succ(nil)$
$\underline{nil = nil} \land nil = nil$	$X = nil \land Z = nil \land Y = succ(nil)$
$\underline{nil = nil}$	$X = nil \land Z = nil \land Y = succ(nil)$
$true$	$X = nil \land Z = nil \land Y = succ(nil)$

The resulting constraint, $X = nil \land Z = nil \land Y = succ(nil)$, is far simpler than the original constraint even though it is equivalent.

2.2 Projection

The usefulness of constraint simplification is even more apparent when we are only interested in some of the variables appearing in the constraint. For example, take the constraints modelling the circuit in Figure 2.1 together with the knowledge that $R1 = 5$ and $R2 = 10$. This gives the constraint:

$$\begin{aligned} &R1 = 5 \land R2 = 10 \land V1 = I1 \times R1 \land V2 = I2 \times R2 \land \\ &V - V1 = 0 \land V - V2 = 0 \land V1 - V2 = 0 \land \\ &I - I1 - I2 = 0 \land -I + I1 + I2 = 0. \end{aligned}$$

If we are only interested in the relationship between the overall current I and overall voltage V in the circuit, we would like to simplify the constraint so that the simplification only involves the *variables of interest*, namely V and I. As we saw in Section 1.2, the resulting simplification should be $3 \times V = 10 \times I$.

A more interesting example of constraint simplification involves reasoning about arbitrary resistance values. Imagine that we add the constraint $V = I \times R$ to the constraint describing the circuit, reflecting our belief that the overall current and voltage are related as if by a single resistor. If we only want to know the relationship between the effective resistance value R and the resistance values $R1$ and $R2$ of the components, then we would like to know the effect of these constraints restricted to only the variables $R, R1$ and $R2$. This is simply:

$$R = (R1 \times R2)/(R1 + R2)$$

which is, of course, the rule for describing the effect of resistors wired in parallel.

Restricting consideration of a constraint to some of the variables in the constraint means that we examine the solutions of the constraint, but ignore the values assigned to variables that are not of interest. This leads to the following definitions:

Definition 2.2
If θ is a valuation of the form $\{x_1 \mapsto d_1, \ldots, x_m \mapsto d_m\}$ then α is an *extension* of θ if it is of the form $\{x_1 \mapsto d_1, \ldots, x_m \mapsto d_m, x_{m+1} \mapsto d_{m+1}, \ldots, x_n \mapsto d_n\}$.
θ is a *partial solution* of C if there exists an extension of θ which is a solution of C.

For example $\{X \mapsto 3, Y \mapsto 4\}$ is an extension of $\{Y \mapsto 4\}$. Since $\{X \mapsto 3, Y \mapsto 4\}$ is a solution of $X \leq Y$, $\{Y \mapsto 4\}$ is a partial solution of $X \leq Y$.

Definition 2.3
The *projection* of a constraint C_1 on to variable set V, where $V \subseteq vars(C_1)$ is a constraint C_2 involving only variables in V such that:

(a) if θ is a solution of C_1, then it is a solution of C_2; and

(b) if θ is a valuation over V which is a solution of C_2, then it is a partial solution of C_1.

Projection formalises our informal notion of simplifying a constraint with respect to the particular variables of interest. It allows us to extract information about the interaction among these variables of interest from a large complex constraint.

For example, the projection of the constraint

$$X \geq Y \land Y \geq Z \land Z \geq T \land T \geq 0$$

on to the variable X is the constraint $X \geq 0$. This is because:
(a) Every solution θ of $X \geq Y \land Y \geq Z \land Z \geq T \land T \geq 0$ is a solution of $X \geq 0$. This follows since the original constraint implies that $X \geq Y \geq Z \geq T \geq 0$.
(b) Every solution of $X \geq 0$, say $\{X \mapsto n\}$ where $n \geq 0$, can be extended to a solution of $X \geq Y \land Y \geq Z \land Z \geq T \land T \geq 0$. One such extension is simply, $\{X \mapsto n, Y \mapsto n, Z \mapsto n, T \mapsto n\}$.

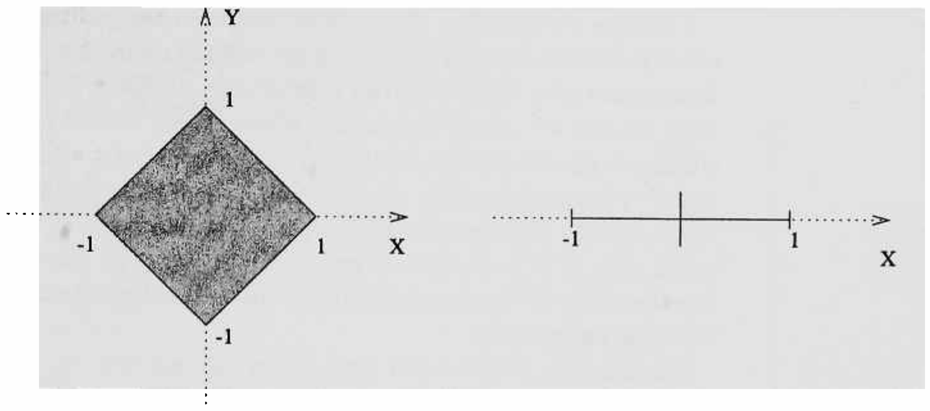

Figure 2.2 A diamond in two dimensions and its projection on to one dimension.

As another example, the projection of

$$cons(Y,Y) = cons(X,Z) \land succ(Z) = succ(T)$$

on to the variables $\{X,T\}$ is the constraint $X = T$. This is because:
(a) Every solution θ to $cons(Y,Y) = cons(X,Z) \land succ(Z) = succ(T)$ is a solution of $X = T$. This follows since the original constraint implies that $X = Y = Z = T$.
(b) Every solution to $X = T$, say $\{X \mapsto t, T \mapsto t\}$ where t is any tree, can be extended to a solution to $cons(Y,Y) = cons(X,Z) \land succ(Z) = succ(T)$. The extension is simply, $\{X \mapsto t, Y \mapsto t, Z \mapsto t, T \mapsto t\}$.

In the case of arithmetic constraints we can give an interesting geometric interpretation of projection. A constraint on $n + m$ variables defines a region in $n + m$ dimensional space. If we project this constraint on to the subspace spanned by m of the variables, the resulting constraint defines the *shadow* of the region in $n + m$ dimensional space in m dimensional space. This is easiest to visualise (and draw!) in the case where we project a 2-dimensional object on to one dimension.

The following constraint on variables X and Y defines the diamond shape shown in Figure 2.2,

$$X + Y \leq 1 \land X - Y \leq 1 \land -X + Y \leq 1 \land -X - Y \leq 1.$$

The shadow of the diamond on the X axis is the line segment shown in Figure 2.2, geometrically illustrating that the projection of this constraint on to the variable X is the constraint $-1 \leq X \land X \leq 1$.

Fourier's algorithm for variable elimination can be used to project a conjunction of linear real inequalities on to a set of variables of interest. At each step, Fourier elimination is used to *eliminate* a variable which is not of interest from the current constraint. This step is repeated until all variables which are not of interest are eliminated. We restrict our presentation to non-strict linear inequalities ($\leq$ and $\geq$), however it is straightforward to extend Fourier's algorithm to strict inequalities ($>$ and $<$) (see Exercise 2.2).

Consider a constraint, C, composed of linear inequalities which we regard as a set of linear inequalities. That is to say, we are really dealing with *primitives*(C). Elimination of a variable y from C proceeds as follows. First, C is partitioned into three subsets: C^0, those inequalities which do not involve y; C^+, those inequalities which are equivalent to an inequality of the form $y \leq t$, where t does not involve y; and C^-, those inequalities which are equivalent to an inequality of the form $t \leq y$, where again t does not involve y. Then, for each pair of the form $t_1 \leq y$ in C^- and $y \leq t_2$ in C^+, we form a new inequality $t_1 \leq t_2$. This set of new inequalities together with C^0 is the projection of the original constraint C on to the original variables except for y.

For example, consider the inequalities representing the diamond in Figure 2.2. We wish to project on to the variable set $\{X\}$, thus we must eliminate Y from the inequalities. The inequalities are partitioned as follows:

$$\begin{aligned} C^0 &= \{\}, \\ C^- &= \{X - 1 \leq Y,\ -1 - X \leq Y\}, \\ C^+ &= \{Y \leq 1 - X,\ Y \leq 1 + X\}. \end{aligned}$$

We now pair each inequality in C^- with each in C^+:

$X - 1 \leq Y$	together with	$Y \leq 1 - X$	gives	$X \leq 1$,
$X - 1 \leq Y$	together with	$Y \leq 1 + X$	gives	$0 \leq 2$,
$-1 - X \leq Y$	together with	$Y \leq 1 - X$	gives	$0 \leq 2$, and
$-1 - X \leq Y$	together with	$Y \leq 1 + X$	gives	$-1 \leq X$.

Thus, after simplification, we discover that the projection of the original constraint on to the variable set $\{X\}$ is $X \leq 1 \land -1 \leq X$.

Algorithm 2.1: Fourier elimination
INPUT: A set of linear inequalities, C, and a set of variables, V.
OUTPUT: A set of linear inequalities, which is the projection of C on to V.
METHOD: The algorithm is given in Figure 2.3. □

It is easy to extend the Fourier elimination algorithm to handle linear equations as well as linear inequalities. All we need to do is rewrite an equality $t = t'$ into the equivalent inequalities $t \leq t'$ and $t' \leq t$.

As another example of a projection algorithm, we now consider Boolean constraints. There is a simple method for eliminating a variable x from a Boolean constraint C. It relies on the observation that C is equivalent to the formula $(x \land C_x) \lor (\neg x \land C_{\neg x})$ where C_x is obtained from C by replacing all occurrences of x by 1 and $C_{\neg x}$ is obtained from C by replacing all occurrences of x by 0. It follows that the projection of C on to $vars(C) - \{x\}$ is just $C_x \lor C_{\neg x}$.

For example, we can eliminate the variable Y from the constraint C,

$$(\neg X \lor Y) \land (\neg X \lor Z) \land (X \lor \neg Y \lor \neg Z),$$

```
$y$ is a variable;
$t, t_1$ and $t_2$ are linear arithmetic terms not including $y$;
$C, C^0, C^+$ and $C^-$ are sets of linear inequalities;
$V$ is a set of variables.

fourier_simplify($C$, $V$)
    while $vars(C) - V$ is not empty
        choose $y \in vars(C) - V$
        $C$ := fourier_eliminate($C$,$y$)
    endwhile
    return $C$

fourier_eliminate($C$, $y$)
    let $C^0$ be those inequalities in $C$ not involving $y$
    let $C^+$ be those inequalities in $C$ which can be written $t \leq y$
    let $C^-$ be those inequalities in $C$ which can be written $y \leq t$
    for each $t_1 \leq y \in C^+$
        for each $y \leq t_2 \in C^-$
            $C^0 := C^0 \cup \{t_1 \leq t_2\}$
        endfor
    endfor
    return $C^0$
```

Figure 2.3 Linear inequality simplifier.

as follows. C_Y is simply

$$(\neg X \vee 1) \wedge (\neg X \vee Z) \wedge (X \vee \neg 1 \vee \neg Z),$$

which is simplified to

$$(\neg X \vee Z) \wedge (X \vee \neg Z).$$

$C_{\neg Y}$ is

$$(\neg X \vee 0) \wedge (\neg X \vee Z) \wedge (X \vee \neg 0 \vee \neg Z),$$

which is simplified to $\neg X$. Now the projection of C on to $\{X, Z\}$ is $C_Y \vee C_{\neg Y}$ which is therefore

$$((\neg X \vee Z) \wedge (X \vee \neg Z)) \vee \neg X.$$

2.3 Constraint Simplifiers

In the preceding section we have seen algorithms which can be used to simplify a constraint by projecting the constraint on to the variables of interest. Unfortunately, it is not possible to do this in all constraint domains. The problem is, there may be a constraint C_1 in the domain which involves variables V (among others) but for which there is no constraint C_2 in the domain which is the projection of C_1 on to V.

For example, consider the domain of tree constraints and the constraint $X = succ(Y)$. Some sample solutions of this constraint are

$$\begin{aligned}&\{X \mapsto succ(0), Y \mapsto 0\},\\&\{X \mapsto succ(succ(0)), Y \mapsto succ(0)\},\\&\{X \mapsto succ(succ(succ(0))), Y \mapsto succ(succ(0))\}.\end{aligned}$$

The projection of $X = succ(Y)$ on to X must therefore have solutions

$$\begin{aligned}&\{X \mapsto succ(0)\},\\&\{X \mapsto succ(succ(0))\},\\&\{X \mapsto succ(succ(succ(0)))\},\end{aligned}$$

but it cannot have $\{X \mapsto 0\}$ as a solution. From the definition of projection, the projection of $X = succ(Y)$ on to X must be a constraint only involving the variable X. But there is no tree constraint involving only the single variable X which has $\{X \mapsto succ(0)\}$ and $\{X \mapsto succ(succ(0))\}$ as solutions and does not have $\{X \mapsto 0\}$ as a solution.

To avoid this problem, we must generalize our notion of simplification with respect to a set of variables in order to allow the simplification to contain extra "local" variables. In general, a constraint simplifier takes a constraint and a set of variables and returns a constraint that has the same solutions as the original with respect to the variables of interest.

Definition 2.4
Constraints C_1 and C_2 are *equivalent* with respect to the set of variables $\{x_1, \ldots, x_n\}$ if:
(a) for each solution $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n, \ldots\}$ of C_1, $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ is a partial solution of C_2; and
(b) for each solution $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n, \ldots\}$ of C_2, $\{x_1 \mapsto d_1, \ldots, x_n \mapsto d_n\}$ is a partial solution of C_1.

Definition 2.5
A ***simplifier***, *simpl*, for constraint domain $\mathcal{D}$ takes a constraint C_1 in $\mathcal{D}$ and a set V of ***variables of interest*** and returns a constraint C_2 in $\mathcal{D}$, such that C_1 and C_2 are equivalent with respect to V.

Clearly the algorithms given in the last section for Boolean constraint projection and Fourier elimination can be used to simplify Boolean constraints and linear arithmetic inequalities. In the case of tree constraints, the preceding example shows that there is no algorithm for projection. However, the following algorithm is a constraint simplifier for tree constraints.

```
x is a variable;
t is a term;
$C, C_1, C_2$ and S are tree constraints;
V is a set of variables.

tree_simplify(C, V)
  $C_1$ := unify(C)
  if $C_1 \equiv false$ then return false endif
  S := true
  while $C_1$ is of the form $x = t \wedge C_2$
    $C_1 := C_2$
    if $x \in V$ then
      if t is a variable and $t \notin V$ then
        substitute x for t in $C_1$ and S
      else
        $S := S \wedge x = t$
      endif
    endif
  endwhile
  return S
```

Figure 2.4 Tree constraint simplifier.

Algorithm 2.2: Tree constraint simplifier
INPUT: Conjunction of term equations C and set V of variables.
OUTPUT: Conjunction of term equations S.
METHOD: The algorithm is shown in Figure 2.4. □

Consider the use of tree_simplify to simplify the constraint

$$h(f(X,Y), Z, g(T)) = h(f(g(T), X), f(X,X), g(U))$$

in terms of the variables of interest Y and T. As a first step, the solved form, C_1, of the original constraint is computed to be

$$Z = f(g(U), g(U)) \wedge X = g(U) \wedge Y = g(U) \wedge T = U.$$

Now each primitive constraint in C_1 is examined. The first two equations, $Z = f(g(U), g(U))$ and $X = g(U)$, are discarded since Z and X are not in the variables of interest. The third equation, $Y = g(U)$ is placed into S as Y is of interest and $g(U)$ is not a variable. The fourth equation, $T = U$, is used to eliminate the variable U, which is not of interest, from S and C_1, replacing it by the variable T, which is of interest. Thus, S becomes $Y = g(T)$, which is the constraint returned by the algorithm.

In this example, tree_simplify has managed to return a constraint which is the projection of the original constraint on to the variables of interest. However, as we have seen it is not always possible to return the exact projection. In this case the answer will contain extra "local" variables. For example, if tree_simplify is applied to the constraint $X = succ(Y)$ with variable of interest X, the original constraint is returned as the answer.

The preceding definition of constraint simplifier is not very demanding. For instance, the identity function is always a constraint simplifier, that is to say, a simplifier which always returns the input constraint meets the specification. However, in practice we usually wish the simplification algorithm to remove as many variables which are not of interest as possible, and we would like the form of the constraint to be as simple as possible.

It is difficult to quantify what simple means in some respects since there is a tradeoff between different aspects of simplicity. Whether one form of a constraint is to be considered simpler than another form is not a straightforward question to answer. In some cases the result of a simplifier may not appear simpler to a human than the original constraint. For example, the result of applying tree_simplify to the constraint $X = f(X_1, X_1) \wedge X_1 = f(X_2, X_2) \wedge \cdots \wedge X_{n-1} = f(X_n, X_n)$ for the set of variables $\{X\}$ is exponentially larger than the original constraint. However, although it is larger it involves only one equation instead of n.

Individuals may argue which of many textual forms of a constraint is the simplest, but there are some guiding principles for constraint simplifiers. For some constraint domains, for example, the linear real inequalities and Boolean constraints, we have seen that we can use projection to simplify a constraint so that it only relates the variables of interest. Such simplifiers are said to be "projecting":

projecting: $vars(simpl(C, V)) \subseteq V$.

Even in the case in which no projecting simplifier exists for a constraint domain, it is desirable that the constraint simplifier does not introduce unnecessary extra variables in the simplified constraint:

weakly projecting: $|vars(simpl(C_1, V)) - V| \leq |vars(C_2) - V|$ for all constraints C_2 such that C_1 and C_2 are equivalent with respect to V. $|S|$ is defined to be the number of elements in the set S.

The tree constraint simplifier tree_simplify is weakly projecting but not projecting.

Projecting and weakly projecting simplifiers are very powerful. In effect, they give a complete solver for a constraint domain. This is because we can test if a constraint is satisfiable by projecting it on to the empty variable set. The resulting constraint cannot contain variables and so must be equivalent to either $true$, indicating the original constraint is satisfiable, or $false$, indicating the original constraint is unsatisfiable.

Redundant constraint information does not usually help in understanding a constraint, hence it is desirable that simplified constraints are redundancy-free.

redundancy-free: $simpl(C, V)$ is redundancy-free.

The tree constraint simplifier tree_simplify described above is redundancy-free. The Fourier elimination simplifier in Figure 2.3 is not redundancy-free, in fact it produces highly redundant constraints if used in the form described.

2.4 Optimization

Sometimes we are not only interested in the satisfiability of a constraint but also wish to find a solution to the constraint. In this case, we rarely desire any arbitrary solution, but instead want a solution that is "best." Finding a "best" solution to a constraint is called an *optimization problem.* This requires some way of specifying which solutions are better than others. The usual way of doing this is by giving an *objective function* that maps each solution to a real value. By convention, we will assume that the aim is to minimize the objective function f. Of course, we can also maximize an objective function, f, just by minimizing $-f$.

Definition 2.6
An *optimization problem*, written (C, f), consists of a constraint C and an *objective function* f which is an expression over the variables in C and which evaluates to a real number.
A valuation, θ, is *preferred* to valuation, θ', if the value of the objective function f under θ is less than the value under θ'. In other words, $\theta(f) < \theta'(f)$.
An *optimal solution*, θ, of (C, f) is a solution of C such that there is no other solution of C which is preferred to θ.

The optimization problem (C, f) in which C is the constraint $X + Y \geq 4$ and f is the objective function $X^2 + Y^2$ is illustrated in Figure 2.5. The shaded polygon in the figure represents the set of solutions to the constraint. The semi-circles across the polygon are contours for the objective function $X^2 + Y^2$ with the value of the objective function written next to the contour line. For instance, some solutions of C are $\{X \mapsto 0, Y \mapsto 4\}$,$\{X \mapsto 3, Y \mapsto 3\}$ and $\{X \mapsto 2, Y \mapsto 2\}$. Applied to f they give $16, 18$ and 8, respectively. From this diagram it is easy to see that the optimal solution is $\{X \mapsto 2, Y \mapsto 2\}$.

Optimization problems do not necessarily have a single optimal solution. For example, consider the above constraint $X + Y \geq 4$ together with the objective function $X + Y$. Any solution of the constraint $X + Y = 4$ is an optimal solution to this optimization problem.

On the other hand, an optimization problem may not have an optimal solution at all. There are two possible reasons for this. The first possibility is that the constraint has no solution, for instance consider the optimization problem $(X \geq 0 \land X \leq -2,\ X)$. The second possible reason is that every solution has a solution which is preferred to it. For instance, consider an arbitrary solution to $(X \leq 0, X)$ of the form $\{X \mapsto r\}$ where r is a negative number. The solution $\{X \mapsto r - 1\}$ is preferable. Since this is the case for any solution, there is no optimal solution.

Optimization problems are not only important in arithmetic constraint domains. Recall the blocks world of Section 1.5. We might wish to minimize the number of objects on the floor while satisfying the constraint

$$red(X) \land red(Y) \land X \neq Y \land pyramid(Z) \land yellow(X) \land floor(X).$$

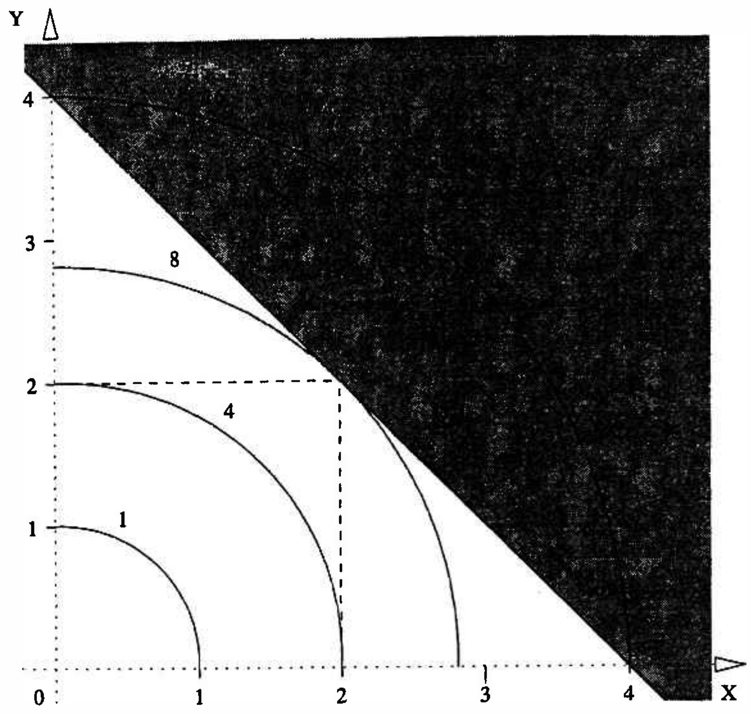

Figure 2.5 An optimization problem.

As a more practical example of the use of optimization, consider the example of building a house from Section 1.2. Suppose we wish to know the earliest time in which the house can be completed. The optimization problem is described by the constraint, C_H,

$$T_S \geq 0 \wedge T_A \geq T_S + 7 \wedge T_B \geq T_A + 4 \wedge T_C \geq T_A + 3 \wedge$$
$$T_D \geq T_A + 3 \wedge T_D \geq T_C + 2 \wedge T_E \geq T_B + 2 \wedge T_E \geq T_D + 3 \wedge T_E \geq T_C + 3$$

together with the objective function T_E. An optimal solution is

$$\{T_S \mapsto 0, T_A \mapsto 7, T_B \mapsto 11, T_C \mapsto 10, T_D \mapsto 12, T_E \mapsto 15\}.$$

Suppose that a building inspector is required to be on site during the period from stage B to stage E. Then minimizing the duration of the stay of the building inspector corresponds to minimizing the objective function $T_E - T_B$. An optimal solution of this problem is

$$\{T_S \mapsto 0, T_A \mapsto 7, T_B \mapsto 13, T_C \mapsto 10, T_D \mapsto 12, T_E \mapsto 15\}.$$

Another optimal solution is

$$\{T_S \mapsto 0, T_A \mapsto 10, T_B \mapsto 19, T_C \mapsto 14, T_D \mapsto 18, T_E \mapsto 21\}.$$

Optimization is quite closely related to simplification. If we consider the constraint C_H defining the house project problem above, and project the constraint on to the variable T_E the resulting constraint is (after redundancy elimination)

$$\mathsf{fourier\_simplify}(C_H, \{T_E\}) = T_E \geq 15.$$

Clearly the minimum completion time is 15 days, and this can be read directly from the simplified constraint. If we require an optimal solution, this may be computed by conjoining $T_E = 15$ with C_H since any solution to this constraint is an optimal solution to the original problem.

The problem of minimizing the building inspector's stay can be answered in the same way. We first add a new variable S, equal to the length of the building inspector's stay, and then use fourier_simplify to project on to this variable:

$$\mathsf{fourier\_simplify}(S = T_E - T_B \land C_H, \{S\}) = S \geq 2.$$

Again the minimum stay length is clear from the simplified constraint. Thus we can use fourier_simplify to answer optimization problems over linear arithmetic constraints with a linear objective function. In general however, it is extremely expensive to do so. A better algorithm is given in the next section.

2.5 The Simplex Algorithm

One of the algorithms most widely used in practice is Dantzig's simplex algorithm. This algorithm answers optimization problems for linear real arithmetic constraints in which the objective function is a linear real arithmetic expression. It is much more efficient than using fourier_simplify and in practice usually has polynomial cost.

To understand how the simplex algorithm works, it is useful to view the optimization problem from a geometric perspective. Consider the problem of minimizing $X - Y$ subject to the constraints $1 \leq X \land X \leq 3 \land 0 \leq Y \land 2Y - X \leq 3$. The problem is shown in Figure 2.6. The shaded polygon in the figure represents the set of solutions to the constraint. The dashed lines across the polygon are contours for the objective function $X - Y$ with the value of the objective function written next to the contour line. From this diagram it is easy to see that the minimum value for the objective function is -1 which occurs at the upper left hand vertex of the polygon. Now, consider the same constraint with the objective function $Y + X$. In this case the minimum value, 1, for the objective function occurs at the bottom left hand vertex of the polygon.

More generally, it can be seen from this diagram, that, for any linear objective function, there is a vertex of the polygon which gives the minimum value to this

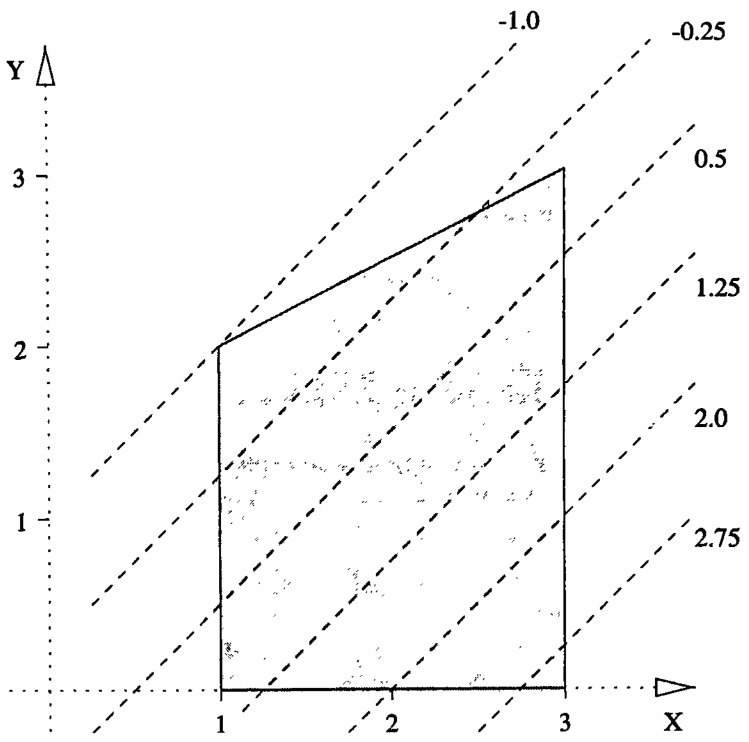

Figure 2.6 Simplex optimization problem.

objective function. Of course, if the contours of the objective function are parallel to one of the faces of the polygon, then this minimum value may occur along one face of the polygon, but, regardless of this, we still know that the minimum value of the objective function will occur at one of the polygon's vertices. This observation continues to hold true whenever we have linear real arithmetic constraints and a linear objective function. Thus to find the solution to a linear programming problem, we need only look at the vertices of the polygon containing the solutions to the constraint. The simplex method is simply a systematic procedure for searching through the vertices of the solution space of the optimization problem to find the vertex which minimizes the objective function.

We now give a presentation of the simplex algorithm which follows the format of the closely related Gauss-Jordan algorithm.

Definition 2.7

An optimization problem (C, f) is in ***simplex form*** if constraint C has the form $C_E \wedge C_I$ where C_E is a conjunction of linear arithmetic equations and C_I is $\bigwedge\{x \geq 0 \mid x \in vars(C)\}$ and f is a linear expression over the variables in C.

The following is an optimization problem in simplex form:

minimize $3X + 2Y - Z + 1$ subject to

$$\begin{array}{rcrcrcrcl} X & + & Y & & & & & = & 3 \quad \land \\ -X & - & 3Y & + & 2Z & + & T & = & 1 \quad \land \\ \multicolumn{9}{c}{X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0.} \end{array} \qquad (P1)$$

It is not difficult to rewrite an arbitrary optimization problem over linear real equations and inequalities into an equivalent simplex form. Each variable x which is not constrained to be non-negative can be replaced by $x^+ - x^-$, where x^+ and x^- are two new variables that are constrained to be non-negative. Each inequality of the form $e \leq r$ where e is a linear real expression and r is a number can be replaced with $e + s = r$ where s is a new non-negative *slack* variable. For example, minimizing $X + 2Y$ subject to $X - 2Y \leq 5 \land -3X + 5Y = 4 \land Y \geq 0$ is written in simplex form as

minimize $X^+ - X^- + 2Y$ subject to

$$\begin{array}{rcrcrcrcl} X^+ & - & X^- & - & 2Y & + & S & = & 5 \quad \land \\ -3X^+ & + & 3X^- & + & 5Y & & & = & 4 \quad \land \\ \multicolumn{9}{c}{X^+ \geq 0 \land X^- \geq 0 \land Y \geq 0 \land S \geq 0.} \end{array}$$

The simplex method works by taking an optimization problem in "basic feasible solved" form and repeatedly applying an operation called "pivoting" to obtain a new basic feasible solved form.

Definition 2.8
A simplex form optimization problem is in *basic feasible solved form* if the equations are of the form $x_0 = b + a_1x_1 + \ldots + a_nx_n$ where the variable x_0 does not occur in any other equation or in the objective function and the constant b is non-negative. The variable x_0 is said to be *basic* and the other variables are *parameters*.
A problem in basic feasible solved form has a corresponding *basic feasible solution* which is obtained by setting each parametric variable to 0 and each basic variable to the value of the constant on the right hand side of its equation.

For instance, the following constraint is in basic feasible solved form and is equivalent to the problem (P1) above.

minimize $10 - Y - Z$ subject to

$$\begin{array}{rcrcrcrl} X & = & 3 & - & Y & & & \land \\ T & = & 4 & + & 2Y & - & 2Z & \land \\ \multicolumn{8}{c}{X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0.} \end{array}$$

X and T are basic variables and Y and Z are parameters. The basic feasible solution corresponding to this basic feasible solved form is $\{X \mapsto 3, Y \mapsto 0, Z \mapsto 0, T \mapsto 4\}$. The value of the objective function with this solution is 10.

The simplex algorithm finds the optimum by repeatedly looking for an "adjacent" basic feasible solved form whose basic feasible solution decreases the value of the objective function. When no such adjacent basic feasible solved form can be found, the optimum has been found. By adjacent we mean that the new basic feasible solved form can be reached by performing a single pivot.

For instance, in our example increasing Y or Z from 0 will decrease the value of the objective function. We arbitrarily choose to increase Y. We must be careful since we cannot increase the value of Y indefinitely as this may cause the constant value associated with some other basic variable to become negative. We must examine the equations to determine the maximum value we can choose for Y, while still maintaining basic feasible solved form. The first equation $X = 3 - Y$ allows Y to take at most a value of 3, since if Y becomes larger than this, X will become negative. The second equation $T = 4+2Y-2Z$ does not restrict Y, since increasing Y will simply increase T. In general, we choose the most restrictive equation, and use it to eliminate Y. In the case of ties, we arbitrarily break the tie. In this example we choose $Y = 3 - X$. We replace Y everywhere by $3 - X$ and obtain

minimize $7 + X - Z$ subject to

$$\begin{array}{ccccccccc} Y & = & 3 & - & X & & & \land \\ T & = & 10 & - & 2X & - & 2Z & \land \\ \multicolumn{8}{c}{X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0.} \end{array}$$

This step is called ***pivoting***, as we have moved X out of the set of basic variables and replaced it by Y. The variable selected to enter the set of basic variables, Y, is called the ***entry variable***, while the variable selected to leave the set of basic variables, X, is called the ***exit variable***.

We continue this process. Increasing the value of Z will also decrease the value of the objective. Note that while decreasing X will also decrease the objective function value, since X is constrained to be non-negative, it already takes its minimum value of 0 in the associated basic feasible solution. We choose Z as the entry variable and T as the exit variable and use the second equation to substitute for Z obtaining:

minimize $2 + 2X + 0.5T$ subject to

$$\begin{array}{ccccccccc} Y & = & 3 & - & X & & & \land \\ Z & = & 5 & - & X & - & 0.5T & \land \\ \multicolumn{8}{c}{X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0.} \end{array}$$

The objective value is 2 and clearly neither increasing the value of X nor that of T will improve this. Thus this is the optimum value.

In general, the simplex algorithm is described as follows. We are given a problem in basic feasible solved form in which the variables $x_1, \ldots, x_n$ are basic and the variables $y_1, \ldots, y_m$ are parameters.

minimize $e + \sum_{j=1}^{m} d_j y_j$ subject to

$$\begin{array}{rcl} \bigwedge_{i=1}^{n}(x_i & = & b_i + \sum_{j=1}^{m} a_{ij} y_j) \wedge \\ \multicolumn{3}{c}{\bigwedge_{i=1}^{n}(x_i \geq 0) \wedge \bigwedge_{j=1}^{m}(y_j \geq 0).} \end{array}$$

Select an entry variable y_J such that $d_J < 0$. Pivoting on such a variable can only decrease the value of the objective function. If no such variable exists, the optimum has been reached. Now determine the exit variable x_I. We must choose this variable so that it maintains basic feasible solved form by ensuring that the new b_i's are still positive after pivoting. This is achieved by choosing an x_I so that $-b_I/a_{IJ}$ is a minimum element of the set

$$\{-b_i/a_{iJ} | a_{iJ} < 0 \text{ and } 1 \leq i \leq n\}.$$

If there is no i for which $a_{iJ} < 0$ then we can stop since the optimization problem is unbounded, and so it does not have a minimum. This is because we can choose y_J to take an arbitrarily large value, and so make the objective function arbitrarily small. Otherwise we choose x_I and now pivot x_I out and replace it with y_J to obtain the new basic feasible solution form. We continue this process until either an optimum is reached or we discover that the problem is unbounded.

Algorithm 2.3: Simplex optimization.
INPUT: An optimization problem $(C_E \wedge C_I, f)$ in basic feasible solved form.
OUTPUT: Either *false* indicating that $(C_E \wedge C_I, f)$ does not have an optimal solution or else an optimal solution to $(C_E \wedge C_I, f)$.
METHOD: Call the algorithm shown in Figure 2.7 with simplex_opt(C_E,f), and let $\langle F, C', f' \rangle$ be the result. If F is *false*, output *false*; otherwise output the basic feasible solution corresponding to C'. □

At this point it is instructive to go back and revisit the geometric example, shown in Figure 2.6. We can rewrite the constraints from this example into simplex form as follows:

$$\begin{array}{rcrcrcrcrcl} X & & & & & - & S_2 & & & = & 1 \wedge \\ X & & & & & & & + & S_3 & = & 3 \wedge \\ -X & + & 2Y & + & S_1 & & & & & = & 3 \wedge \\ \multicolumn{11}{c}{X \geq 0 \wedge Y \geq 0 \wedge S_1 \geq 0 \wedge S_2 \geq 0 \wedge S_3 \geq 0.} \end{array}$$

Each of the vertices of the feasible region corresponds to a basic feasible solved form equivalent (with respect to the original variables) to the original constraints. For example, the upper right hand side vertex corresponds to the solved form

```
$C$ is a conjunction of equations;
$i, I, j, J, n, m$ are integers;
$a_{ij}$, $b_i$, $e$, $d_i$ are real constants;
$f$, $t$ are linear expressions;
$c_1, \ldots, c_n$ are equations;
$x_i$, $y_j$ are variables.

simplex_opt($C$,$f$)
    let $C$ be of the form $c_1 \wedge \cdots \wedge c_n$
    for each $i \in \{1, \ldots, n\}$
    let $c_i$ be of the form $x_i = b_i + \sum_{j=1}^{m} a_{ij} y_j$
    endfor
    let $f$ be of the form $e + \sum_{j=1}^{m} d_j y_j$
    % Choose variable $y_J$ to become basic
    if for all $j \in \{1, \ldots, m\}$ $d_j \geq 0$ then
        return $\langle true, C, f \rangle$
    endif
    choose $J \in \{1, \ldots, m\}$ such that $d_J < 0$
    % Choose variable $x_I$ to become non-basic
    if for all $i \in \{1, \ldots, n\}$ $a_{iJ} \geq 0$ then
        return $\langle false, C, f \rangle$
    endif
    choose $I \in \{1, \ldots, n\}$ such that
        $-b_I/a_{IJ} = \min\{-b_i/a_{iJ} | a_{iJ} < 0 \text{ and } 1 \leq i \leq n\}$
    $t := (x_I - b_I - \sum_{j=1, j \neq J}^{m} a_{Ij} y_j)/a_{IJ}$
    $c_I := (y_J = t)$
    replace $y_J$ by $t$ in $f$
    for each $i \in \{1, \ldots, n\}$
        if $i \neq I$ then replace $y_J$ by $t$ in $c_i$ endif
    endfor
    return simplex_opt($\bigwedge_{i=1}^{n} c_i$,$f$).
```

Figure 2.7 Simplex optimization.

minimize $0 + 0.5S_1 - 0.5S_3$ subject to

$$\begin{array}{rcrrrl} Y & = & 3 & - \; 0.5S_1 & - \; 0.5S_3 & \wedge \\ S_2 & = & 2 & & - \; S_3 & \wedge \\ X & = & 3 & & - \; S_3 & \wedge \\ \multicolumn{6}{l}{X \geq 0 \wedge Y \geq 0 \wedge S_1 \geq 0 \wedge S_2 \geq 0 \wedge S_3 \geq 0.} \end{array}$$

The corresponding basic feasible solution is

$$X = 3 \wedge Y = 3 \wedge S_2 = 2 \wedge S_1 = 0 \wedge S_3 = 0.$$

Now consider the operation of the simplex optimization algorithm when we start from this basic feasible solved form. The variable S_3 will be selected as the entry variable, since this is the only variable with a negative coefficient in the objective function. S_3 appears in all three constraints with a negative coefficient so the algorithm will examine the "$-b_i/a_{iJ}$" value for each constraint c_i where a_{iJ} is the coefficient of S_3 in the constraint. These values are, respectively, $(-3/-0.5) = 6$, $(-2/-1) = 2$, and $(-3/-1) = 3$, so the basic variable, S_2, of the second constraint is chosen as the exit variable. After pivoting, the new basic feasible solved form is

minimize $-1 + 0.5S_1 + 0.5S_2$ subject to

$$\begin{array}{lcllll} Y & = & 2 & - \; 0.5S_1 & + \; 0.5S_2 & \wedge \\ S_3 & = & 2 & & - \; S_2 & \wedge \\ X & = & 1 & & + \; S_2 & \wedge \\ \multicolumn{6}{l}{X \geq 0 \wedge Y \geq 0 \wedge S_1 \geq 0 \wedge S_2 \geq 0 \wedge S_3 \geq 0.} \end{array}$$

Examination of the objective function reveals that no variable has a negative coefficient, therefore a minimum has been reached. The algorithm will return the corresponding basic feasible solution which is:

$$Y = 2 \wedge S_3 = 2 \wedge X = 1 \wedge S_1 = 0 \wedge S_2 = 0.$$

Examination of this solution reveals that, as the reader would expect, it corresponds to the upper left hand vertex of the original solution space.

This example illustrates that the simplex optimization algorithm moves from one vertex to another vertex of the solution space in a direction which decreases the objective function. This process continues until the minimum is found.

However, we have neglected to answer one rather important question, namely, how do we obtain the first basic feasible solved form? Actually, this is not as difficult as it may first appear. We can do this by solving an optimization problem for which we can trivially find an initial basic feasible solved form and which has the property that its optimal solution occurs at an initial basic feasible solved form for the original problem. More exactly, consider the optimization problem over the variables $x_1, \ldots, x_m$:

minimize $e + \sum_{j=1}^{m} d_j x_j$ subject to

$$\begin{array}{rcl} \bigwedge_{i=1}^{n}(\sum_{j=1}^{m} a_{ij}x_j & = & b_i) \wedge \\ \bigwedge_{j=1}^{m}(x_j \geq 0). & & \end{array}$$

We assume that the constraints have been rewritten so that for each $1 \leq i \leq n$, $b_i \geq 0$. This is always possible by negating both sides.

The ***first phase problem*** is obtained by adding ***artificial variables***, $z_1, \ldots, z_n$, to get the problem in basic feasible solved form. The problem becomes:

minimize $\sum_{i=1}^{n}(b_i - \sum_{j=1}^{m} a_{ij}x_j)$ subject to

$$\begin{array}{rcl} \bigwedge_{i=1}^{n}(z_i & = & b_i - \sum_{j=1}^{m} a_{ij}x_j) \wedge \\ \multicolumn{3}{c}{\bigwedge_{j=1}^{m}(x_j \geq 0) \wedge \bigwedge_{i=1}^{n}(z_i \geq 0).} \end{array}$$

The key idea is that we wish to minimize $\sum_{i=1}^{n} z_i$. An optimal solution to this problem where each z_i is 0 corresponds to a solution to the original constraints, since in this case the constraints with artificial variables are equivalent to the original constraints.

The optimization of the first phase problem can end in three ways.

(a) The optimal value of the objective function is > 0. In this case the original problem is unsatisfiable, since there are no solutions in which all of the artificial variables are 0.

(b) The optimal value of the objective function is 0 and all artificial variables are parametric. In this case, after removal of the artificial variables, we have a basic feasible solved form for the original problem.

(c) The optimal value of the objective function is 0 but not all artificial variables are parametric. Consider such a non-parametric artificial variable z. It must occur in an equation of form

$$z = 0 + \sum_{i=1}^{n} d'_i z_i + \sum_{j=1}^{m} a'_j x_j.$$

The first possibility is that all variables in the equation are artificial, that is all of the a'_j are 0. In this case we can delete the equation. This occurs when one of the original constraints is redundant. Otherwise, one of the original variables x_J, say, has a non-zero coefficient a'_J. We can use this equation to pivot z out of the basic variables and replace it by x_J. This maintains basic feasible solved form because the constant in the equation is 0. We can continue this process until all artificial variables become parameters.

Perhaps it is time for an example to clarify this. Consider the original problem:

minimize $3X + 2Y - Z + 1$ subject to

$$\begin{array}{rcrcrcrcrl} X & + & Y & & & & & = & 3 & \land \\ -X & - & 3Y & + & 2Z & + & T & = & 1 & \land \\ \multicolumn{10}{l}{X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0.} \end{array}$$

Adding two artificial variables we obtain the first phase problem:

minimize $A_1 + A_2$ subject to

$$\begin{array}{rcrcrcrcrcrl} A_1 & = & 3 & - & X & - & Y & & & & & \land \\ A_2 & = & 1 & + & X & + & 3Y & - & 2Z & - & T & \land \\ \multicolumn{12}{l}{X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0 \land A_1 \geq 0 \land A_2 \geq 0.} \end{array}$$

To put it into basic feasible solved form, we rewrite the objective function in terms of the parametric variables (by substituting for the artificial variables), giving the new objective

$$4 + 2Y - 2Z - T.$$

The simplex algorithm is now used to solve this problem. We must first choose Z or T as the new entry variable as they have negative coefficients in the objective. Arbitrarily we choose T. The exit variable is therefore A_2. After pivoting we obtain:

minimize $3 - X - Y + A_2$ subject to

$$\begin{array}{lcccccccccl} A_1 & = & 3 & - & X & - & Y & & & & \land \\ T & = & 1 & + & X & + & 3Y & - & 2Z & - & A_2 \quad \land \end{array}$$
$$X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0 \land A_1 \geq 0 \land A_2 \geq 0.$$

In the next step we can choose either X or Y as the entry variable. Say we choose X. The exit variable must therefore be A_1. This gives:

minimize $A_1 + A_2$ subject to

$$\begin{array}{lcccccccccl} X & = & 3 & - & Y & & & - & A_1 & & \land \\ T & = & 4 & + & 2Y & - & 2Z & - & A_1 & - & A_2 \quad \land \end{array}$$
$$X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0 \land A_1 \geq 0 \land A_2 \geq 0.$$

We have reached the optimal value and constructed a basic feasible solution form for the original problem since

$$\begin{array}{rcccccccl} X & + & Y & & & & & = & 3 \quad \land \\ -X & - & 3Y & + & 2Z & + & T & = & 1 \end{array}$$

is equivalent to

$$\begin{array}{lcccccl} X & = & 3 & - & Y & & \land \\ T & = & 4 & + & 2Y & - & 2Z. \end{array}$$

The optimization problem can now be solved by starting from the above basic feasible solution although we must remember to first eliminate the basic variables from the original objective function. This gives:

minimize $10 - Y - Z$ subject to

$$\begin{array}{lcccccl} X & = & 3 & - & Y & & \land \\ T & = & 4 & + & 2Y & - & 2Z \quad \land \end{array}$$
$$X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0.$$

Of course, finding a basic feasible solution of the original constraint is exactly a constraint satisfaction problem—thus the first phase of the simplex method provides an efficient constraint solver for linear inequalities. The algorithm is:

Algorithm 2.4: Simplex solver.
INPUT: A constraint $C_E \land C_I$ in simplex form.
OUTPUT: *true* if $C_E \land C_I$ is satisfiable and *false* if it is unsatisfiable.
METHOD: Return the result of simplex_solve(C_E) using the algorithm in Figure 2.8.
□

```
$C$, $C'$ are conjunction of equations;
$i, j, m$ are integers;
$a_{ij}$, $b_i$, $e$, $d'_j$, $a'_i$ are real constants;
$f$, $f'$, $f_i$ are linear expressions;
$c_i$, $c'_i$ are equations;
$x_i$ are variables, $z_i$ are new variables ;
$flag$ is either $true$ or $false$.

simplex_solve($C$)
    let $C$ be of the form $c_1 \wedge \cdots \wedge c_n$
    for each $i \in \{1, \ldots, n\}$
        let $c_i$ be of the form $b_i = \sum_{j=1}^{m} a_{ij}x_j$ where $b_i \geq 0$
        $f_i := b_i - \sum_{j=1}^{m} a_{ij}x_j$
        $c'_i := (z_i = f_i)$
    endfor
    $f := \sum_{i=1}^{n} f_i$
    $\langle flag, C', f' \rangle :=$ simplex_opt($\bigwedge_{i=1}^{n} c'_i, f$)
    let $f'$ be of the form $e + \sum_{j=1}^{n} d'_j z_j + \sum_{i=1}^{m} a'_i x_i$
    if $e \equiv 0$ then
        return $true$
    else return $false$
    endif
```

Figure 2.8 Simplex solver.

No discussion of the simplex algorithm is complete without considering the problem of cycling. A basic feasible solution is *degenerate* if some of the basic variables take the value 0. If this is the case, we can perform a pivot which does not decrease the value of the objective function and which does not change the corresponding basic feasible solution. However, in such a case there is a danger of pivoting back to the original problem. This is called cycling. In practice it does not occur very often, and there are simple methods to avoid it. One of these is Bland's anti-cycling rule. This approach numbers the variables and, when selecting a variable to enter the basis, chooses the lowest possible numbered variable. Similarly when selecting a variable to exit the basis, and a tie occurs, it chooses the lowest numbered basic variable to break the tie.

2.6 (*) Canonical Form Simplifiers

When we discussed simplifiers previously we gave a number of properties which are desirable in a simplifier. Another desirable property for a simplifier to have is that it returns constraints which are in a *canonical form*. Such simplifiers are useful because they make it easy to determine if two constraints are equivalent since they will simplify equivalent constraints to exactly the same syntactic form.

Definition 2.9
A simplifier, *simpl*, is a *canonical form* simplifier if $simpl(C_1, V) \equiv simpl(C_2, V)$ whenever constraints C_1 and C_2 are equivalent with respect to the variables V. The *canonical form* of a constraint C is $simpl(C, vars(C))$.

One might hope that simplifiers based on solved form solvers, such as the tree constraint simplifier tree_simplify, are canonical as these simplify the constraints to a solved form. However, tree_simplify is not canonical. To see this, consider the constraint C_1 which is $X = Y \land Z = f(X)$ and the constraint C_2 which is $Y = X \land Z = f(X)$. Clearly, C_1 and C_2 are equivalent with respect to the variable set $\{X, Y, Z\}$. However, tree_simplify$(C_1, \{X, Y, Z\})$ is $X = Y \land Z = f(Y)$ while tree_simplify$(C_2, \{X, Y, Z\})$ is C_2.

This example illustrates one of the issues to be considered when designing a canonical form simplifier. It must take into account the name of variables since, for instance, it cannot treat $X = Y$ and $Y = X$ identically as one must be rewritten to the other. Another issue is the need for a standard way to write expressions. For instance, $3 \times X + Y/2$ and $(6/2) \times X + 0.5 \times Y$ must be rewritten to the same form.

As an example of a canonical form simplifier, we give one for linear equations. It is a modification of the Gauss-Jordan algorithm which ensures that variables are eliminated in a particular way. We take variable names into account by assuming that there is an ordering on variable names which allows us to determine the least variable in a set of variables in a constraint. In the examples we will use lexicographic ordering. For instance, X_1, X_2 and A have the relative ordering $A < X_1 < X_2$. The canonical form of a linear expression t is $b_0 + b_1 x_1 + \cdots b_n x_n$ where each b_i is a constant and $x_1 < x_2 < \cdots < x_n$. We assume within the algorithm that all linear expressions are maintained in canonical form.

Algorithm 2.5: Gauss-Jordan canonical form simplifier.
INPUT: A conjunction of linear arithmetic equations C and a set of variables V.
OUTPUT: A conjunction of linear arithmetic equations S in canonical form.
METHOD: The algorithm is shown in Figure 2.9. □

For example, the execution of gauss_jordan_simplify$(C, \{X, Z, T\})$ where C is the constraint $1 + X = 2Y + Z \land Z = 3 + X \land T - Y = 2$ is detailed in the following table. It gives C and S and the substitution used in each elimination step.

Substitution	C	S
	$\underline{1 + X = 2Y + Z} \land Z = 3 + X \land T - Y = 2$	*true*
$Y = 0.5 + 0.5X - 0.5Z$	$\underline{Z = 3 + X} \land -0.5 + T - 0.5X + 0.5Z = 2$	*true*
$X = -3 + Z$	$\underline{1 + T = 2}$	$X = -3 + Z$
$T = 1$	*true*	$X = -3 + Z \land T = 1$
sorting S		$X = -3 + Z \land T = 1$

```
$r_1, r_2$ are real constants;
$x$ is a variable;
$e$ is a linear arithmetic expression;
$c$ is an equation;
$C, C_1$ and $S$ are conjunctions of equations.


gauss_jordan_simplify($C$,$V$)
  $S := true$
  while $C$ is not the empty conjunction
    let $C$ be of the form $c \land C_1$
    $C := C_1$
    if $c$ is of the form $r_1 = r_2$ then
      if $r_1 \not\equiv r_2$ then
        return $false$
      endif
    else
      if $c$ can be written in the form $y = e$ where
        $e$ does not involve $x$ and $y \notin V$ then
        let $x$ be this variable ($y$)
      else
        let $x$ be the least variable in $V$ such that
          $c$ can be written in the form $x = e$ where $e$ does not involve $x$
      endif
      substitute $e$ for $x$ throughout $C$ and $S$
      if $x \in V$ then
        $S := S \land (x = e)$
      endif
    endif
  endwhile
  sort $S$ on the variables appearing on the left hand sides
  return $S$
```

Figure 2.9 Gauss-Jordan canonical form simplifier.

Producing a canonical form simplifier for a constraint domain which does not always allow a variable to be eliminated is slightly more complicated, since we may need to rename variables. For example, consider the tree constraint domain. If the variable of interest is X and we simplify $X = succ(Z)$ and $X = succ(Y)$ using the non-canonical tree constraint simplifier we will obtain $X = succ(Z)$ and $X = succ(Y)$. This is because we cannot eliminate Z or Y. However, $X = succ(Z)$ and $X = succ(Y)$ are equivalent with respect to the variable X and so a canonical form simplifier must return the same result if either is simplified with respect to X. We can solve this problem by eliminating as many of the variables as possible, and then systematically renaming those remaining variables which are not of interest. Thus, a canonical form simplifier for trees might simplify both $X = succ(Z)$ and $X = succ(Y)$ to $X = succ(W)$ if X is the variable of interest. See Exercise 2.7.

2.7 (*) Implication and Equivalence

Another important question involving constraints is that of implication. Given two constraints C_1 and C_2, this is the problem of determining whether $C_1 \rightarrow C_2$. That is to say, determining if every solution of C_1 is also a solution of C_2. We have already met with a use of implication for simplification, namely redundancy removal, but implication also has other uses.

Suppose we wish to model the actions of an air-conditioning system whose temperature control is set to some desired temperature T. We can model the temperature at time period X_{i+2} in terms of the temperature at the previous two time periods, X_i and X_{i+1}, and the temperature D_{i+2} which reflects the outside influence on the temperature by using the formula:

$$X_{i+2} = X_{i+1} + 3/4 \times (T - X_{i+1}) + 1/2 \times (T - X_i) + D_{i+2}.$$

Apart from D_{i+2} and X_{i+1} the remaining terms model the effect of the air-conditioning system which is trying to change the temperature to T.

Suppose the temperature control T is set to 10 and this is the initial temperature of the room, that is to say $X_0 = 10$. Then $X_0 = 10 \land C_1$ models the room temperature over the next 10 time periods where C_1 is

$$\begin{array}{rcl} X_1 & = & X_0 + D_1 \land \\ X_2 & = & X_1 + 3/4 \times (10 - X_1) + 1/2 \times (10 - X_0) + D_2 \land \\ \vdots & & \\ X_{10} & = & X_9 + 3/4 \times (10 - X_9) + 1/2 \times (10 - X_8) + D_{10}. \end{array}$$

A graph of the change in room temperature over time is illustrated at the top of Figure 2.10, where the outside influence D_i is given by the graph illustrated at the bottom of Figure 2.10.

How can we judge how well the air-conditioning system works? One criterion might be that it is successful if the temperature always stays within two degrees of the set temperature, assuming that the outside effects never make a difference of more than one degree, that is $-1 \leq D_i \land D_i \leq 1$ for all $1 \leq i \leq 10$. Let C_2 be the constraint $X_0 = 10 \land C_1 \land \bigwedge_{i=1}^{10}(-1 \leq D_i \land D_i \leq 1)$. Then this question is formalised as the implication question:

$$C_2 \rightarrow 8 \leq X_0 \land X_0 \leq 12 \land \cdots \land 8 \leq X_{10} \land X_{10} \leq 12.$$

In this case the answer is no. However, the system does ensure that the temperature stays within three degrees of the set temperature (that is, between 7 and 13).

A more interesting question is whether the system reaches a stable state after 10 time periods. Suppose the system is switched on when the initial temperature X_0 is between 0 and 20 degrees. We can describe this situation using the constraint C_3

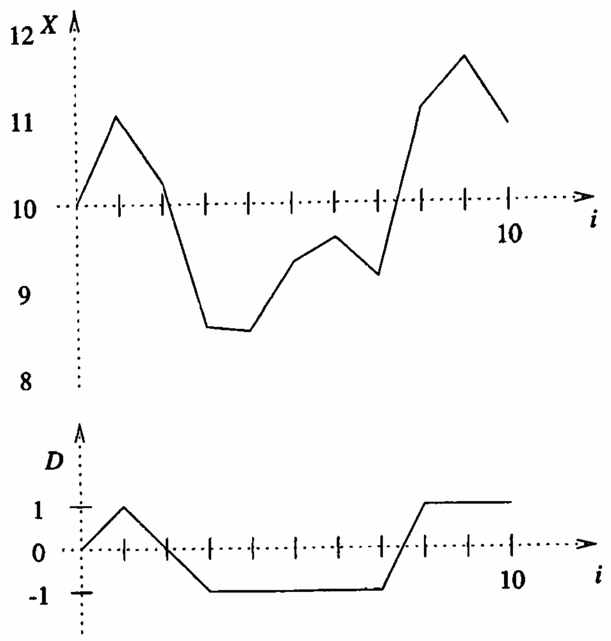

Figure 2.10 A graph of room temperature versus outside influence over time.

which is

$$0 \leq X_0 \wedge X_0 \leq 20 \wedge C_1 \wedge \bigwedge_{i=1}^{10}(-1 \leq D_i \wedge D_i \leq 1)$$

and we can ask if X_{10} is within three degrees of the set temperature, that is does

$$C_3 \rightarrow 7 \leq X_{10} \wedge X_{10} \leq 13.$$

In this case the answer is yes.

In both cases we are interested in determining whether an implication holds.

Definition 2.10

An ***implication tester***, *impl*, takes two constraints C_1 and C_2 and returns *true*, *false* or *unknown*. Whenever *impl*(C_1, C_2) returns *true* then $C_1 \rightarrow C_2$, while when it returns *false*, $C_1 \rightarrow C_2$ is not true.

An implication tester that always returns *true* or *false* is *complete*.

One important application of an implication tester is for simplification where it can be used to remove redundant primitive constraints. We will see later in Chapter 11 that determining if one constraint implies another is also important for deciding when a bottom-up evaluation of a constraint logic program can finish.

We shall now consider how we might implement an implication tester. The first point to note is that in any constraint domain, an arbitrary implication test can be simplified to a sequence of implication tests in which the second argument to the implication tester is always a primitive constraint. This is because if C_2 is $c_1 \wedge \cdots \wedge c_n$, then $C_1 \rightarrow C_2$ if and only if $C_1 \rightarrow c_i$ for each $i = 1, \ldots, n$.

The second point to note is that in many constraint domains (essentially those domains in which the primitive constraints are closed under negation), we can use a constraint solver to determine if $C_1 \rightarrow c$ where c is a primitive constraint. This is because $C_1 \rightarrow c$ holds if and only if $C_1 \wedge \neg c$ is unsatisfiable.

Consider the problem of the air-conditioning system. To determine whether $C_2 \rightarrow (7 \leq X_{10} \wedge X_{10} \leq 13)$, we decompose it into the two tests $C_2 \rightarrow 7 \leq X_{10}$ and $C_2 \rightarrow X_{10} \leq 13$. Each of these can be answered by using a complete solver. Since $solv(C_2 \wedge 7 > X_{10})$ returns *false* we know that $C_2 \rightarrow 7 \leq X_{10}$. Similarly, since $solv(C_2 \wedge X_{10} > 13)$ also returns *false* we know that $C_2 \rightarrow X_{10} \leq 13$. Thus $C_2 \rightarrow (7 \leq X_{10} \wedge X_{10} \leq 13)$.

Another operation over constraints which is closely related to implication is equivalence. Given two constraints C_1 and C_2, this is the problem of determining whether $C_1 \leftrightarrow C_2$. That is, determining if C_1 and C_2 have the same solutions.

Definition 2.11
An *equivalence tester*, *equiv*, takes two constraints C_1 and C_2 and returns *true*, *false* or *unknown*. Whenever $equiv(C_1, C_2) = true$, $C_1 \leftrightarrow C_2$ holds and whenever $equiv(C_1, C_2) = false$, $C_1 \leftrightarrow C_2$ does not hold.
An equivalence tester *equiv* is *complete* if it always returns *true* or *false*.

Given that we have an implication tester for a constraint domain, it is simple to build an equivalence tester for the domain since $C_1 \leftrightarrow C_2$ if and only if $C_1 \rightarrow C_2$ and $C_2 \rightarrow C_1$. Thus we can define an equivalence tester by

$$equiv(C_1, C_2) = impl(C_1, C_2) \wedge impl(C_2, C_1).$$

Conversely, given an equivalence tester it is also easy to build an implication tester since, using the definition of implication, $C_1 \rightarrow C_2$ if and only if $C_1 \leftrightarrow (C_1 \wedge C_2)$. Therefore,

$$impl(C_1, C_2) = equiv(C_1, C_1 \wedge C_2).$$

Thus, we can test if $Y = X - 1 \wedge X \geq 2$ is equivalent to $X = Y + 1 \wedge Y \geq 3$ by testing that

$$Y = X - 1 \wedge X \geq 2 \;\rightarrow\; X = Y + 1 \wedge Y \geq 3$$

and

$$X = Y + 1 \wedge Y \geq 3 \;\rightarrow\; Y = X - 1 \wedge X \geq 2.$$

Each of these implications can be tested using a constraint solver.

Since a canonical form simplifier can be used to determine equivalence, equivalence testing is closely related to canonical simplification More exactly, given a canonical simplifier *simpl*, we can test if two constraints C_1 and C_2 are equivalent, by simplifying both with respect to all variables of interest and checking if the result is syntactically identical. C_1 and C_2 are equivalent if and only if,

$$simpl(C_1, vars(C_1) \cup vars(C_2)) \equiv simpl(C_2, vars(C_1) \cup vars(C_2)).$$

For this reason, equivalence testers are often based on canonical form simplifiers.

Imagine we wish to determine if

$$Z - X = 3 \land T + Z - X = 4 \leftrightarrow T + Z - X = 4 \land 2T + Z - X = 5.$$

The canonical simplifier gauss_jordan_simplify applied to $Z-X = 3 \land T+Z-X = 4$ for the variables of interest $\{T, X, Z\}$ gives the answer $X = -3 + Z \land T = 1$. Similarly, gauss_jordan_simplify applied to $T + Z - X = 4 \land 2T + Z - X = 5$ with the same variables of interest gives the same result. Thus, $Z-X = 3 \land T+Z-X = 4$ is equivalent to $T + Z - X = 4 \land 2T + Z - X = 5$.

We have already seen that we can implement an implication tester in terms of an equivalence tester. Therefore a canonical simplifier can also be used to test implications. For example, imagine we wish to determine if $C_1 \rightarrow C_2$ where C_1 is

$$Z - X = 3 \land T + Z - X = 4$$

and C_2 is

$$X - Z + 2 = T.$$

The variables of interest, V, are $\{T, X, Z\}$. Since gauss_jordan_simplify(C_1, V) and gauss_jordan_simplify($C_1 \land C_2, V$) both return $X = -3 + Z \land T = 1$, the implication holds.

2.8 Summary

We have examined constraint simplification. This is needed in constraint programming languages because execution of a constraint program usually results in an "answer" constraint which must be simplified so as to make its meaning intelligible. Simplification is particularly important when we are not interested in all the variables in the constraints and only wish to understand the constraint in terms of the variables we are interested in. One desirable property of a simplifier is that it is projecting. That is, after simplification the constraint will only contain variables which are of interest. Unfortunately, for some constraint domains such as tree constraints, it is not possible to give a simplifier which is projecting. All we can hope for in such domains is a solver which is weakly projecting. We have given a simplifier for linear arithmetic inequalities which is projecting, and a simplifier for tree constraints which is weakly projecting.

Optimization is another important operation on constraints. This deals with the problem of finding a solution to a constraint which is the best in the sense that it minimizes some given objective function. We have studied the simplex algorithm which solves optimization problems for linear arithmetic constraints with a linear arithmetic objective function. A variant of the simplex algorithm can also be used to give an efficient complete constraint solver for linear arithmetic constraints.

Finally, we have investigated algorithms to determine if two constraints are equivalent or if one implies the other. Tests for equivalence and implication can be easily implemented if we have access to a constraint simplifier which is canonical.

2.9 Exercises

2.1. Recall the constraint describing the house project from Chapter 1:

$$T_S \geq 0 \wedge T_A \geq T_S + 7 \wedge T_B \geq T_A + 4 \wedge T_C \geq T_A + 3 \wedge$$
$$T_D \geq T_A + 3 \wedge T_D \geq T_C + 2 \wedge T_E \geq T_B + 2 \wedge T_E \geq T_D + 3 \wedge T_E \geq T_C + 3.$$

Use fourier_simplify to eliminate the variables T_S, T_B, T_C, T_A and T_D in turn.

2.2. Extend fourier_simplify to handle strict inequalities.

2.3. Using tree_simplify, simplify constraint $X = f(Y, U) \wedge U = Z \wedge Z = g(a, Y)$ in terms of variables X and Y.

2.4. Write an algorithm for Boolean simplification, based on the example in the text. Now consider the Boolean constraints describing the full adder from Chapter 1:

$I_1 \leftrightarrow X \,\&\, Y$	And gate $G1$
$I_2 \leftrightarrow X \oplus Y$	Xor gate $G2$
$I_3 \leftrightarrow I_2 \,\&\, CI$	And gate $G3$
$O \leftrightarrow I_2 \oplus CI$	Xor gate $G4$
$CO \leftrightarrow I_1 \vee I_3$	Or gate $G5$

Using your Boolean simplification algorithm, eliminate the internal variables I_1, I_2 and I_3.

2.5. In the text we saw how a projecting simplifier can be used to solve optimization functions. In the case of linear arithmetic constraints show how the simplex optimization algorithm can be used to project a set of constraints in simplex form on to a single variable.

2.6. (*) It is also possible to give an algorithm for Boolean simplification which is based on fourier_simplify. It relies on the Boolean constraints being written in conjunctive normal form. The idea is to eliminate a variable by combining those clauses in which the variable occurs positively with those in which the variable occurs negatively. The clauses in which the variable does not occur remain unchanged. This is the analogue of the Fourier elimination step.

For instance, we can eliminate the variable Y from

$$(\neg X \vee Y) \wedge (\neg X \vee Z) \wedge (X \vee \neg Y \vee \neg Z)$$

as follows. The set of clauses C^0 in which Y does not appear is $\{\neg X \vee Z\}$, the set of clauses C^+ in which Y appears positively is $\{\neg X \vee Y\}$, and the set of clauses C^- in which Y appears negatively is $\{\neg Y \vee \neg Z \vee X\}$. Combining C^+ with C^- we obtain the set $\{\neg X \vee \neg Z \vee X\}$. Thus, the projection of the original constraint on to the variables X and Z is

$$(\neg X \vee \neg Z \vee X) \wedge (\neg X \vee Z)$$

which is equivalent to $\neg X \vee Z$.

(a) Using the distribution and De Morgan's Laws for Boolean constraints:

- $\neg\neg F = F$,
- $\neg(F \vee G) = \neg F \wedge \neg G$,
- $\neg(F \wedge G) = \neg F \vee \neg G$,
- $F \vee (G \& H) = (F \vee G) \wedge (F \vee H)$;

write an algorithm which converts a Boolean constraint into a conjunctive normal form. For instance, the Boolean formula $X \leftrightarrow Y \& Z$ is equivalent to the conjunctive normal form constraint

$$(\neg X \vee Y) \wedge (\neg X \vee Z) \wedge (X \vee \neg Y \vee \neg Z).$$

(b) Write an algorithm for Boolean simplification based on the above example.

(c) Apply your algorithms to the Boolean constraints of Exercise 2.4. That is, rewrite these into conjunctive normal form, and then, using your Boolean simplification algorithm, eliminate the internal variables I_1, I_2 and I_3.

2.7. (*) Write a canonical simplifier for tree constraints based on tree_simplify.

2.8. (*) Write an implication tester for the linear arithmetic inequalities.

2.9. (*) Is the conjunctive normal form for Boolean constraints a canonical form? If not, suggest a canonical form for Boolean constraints which is based on conjunctive normal form.

2.10 Practical Exercises

We have already seen that CLP systems return a simplified form of the constraint which the user has typed. In addition, the language $CLP(\mathcal{R})$ has a library function, called **dump** for projecting and simplifying constraints and writing the result to standard output. The function **dump** takes a single argument which is a list of the variables of interest and works both for linear arithmetic constraints and for tree constraints.

For instance, to simplify constraint $X = f(Y, Z) \land Z = g(a, Y)$ in terms of the variables X and Y, we can type

```
1 ?- X = f(Y,Z), Z=g(a,Y), dump([X,Y]).
```

This leads to the output

```
X = f(Y,g(a,Y)).
```

which is the correct simplification.

As another example, we can type

```
2 ?- T = 3+Y, X = 2*Y+U, Z=3*U+Y, dump([X,T,Z]).
```

which gives

```
Z = 3*X-5*T+15.
```

demonstrating that $Z = 3X - 5T + 15$ is a simplification of

$$T = 3 + Y \land X = 2Y + U \land Z = 3U + Y$$

with respect to X, T and Z.

The SICStus Prolog real arithmetic constraint solving libraries also simplify constraints although they do not provide an explicit function for simplification (but see Section 28.7.1.1 of the SICStus Prolog v3 User's Manual [58]). For example, typing

```
| ?- {X + Y = Z + 2, 2*X + Y = Z}.
```

gives the answer

```
X = -2,
{Z=-4+Y} ?
```

The SICStus Prolog real arithmetic libraries also include an implementation of the simplex algorithm which can be used to answer optimization problems. The algorithm can be called using the library functions inf and sup (for infimum and supremum). The function inf takes two arguments, an expression to minimize and an expression to hold the minimum value. The behaviour of sup is analogous except that it attempts to maximize the expression.

For instance, to minimize $3X + 2Y - Z + 1$ subject to

$$X + Y = 3 \land -X - 3Y + 2Z + T = 1 \land X \geq 0 \land Y \geq 0 \land Z \geq 0 \land T \geq 0$$

we can type

```
| ?- {X + Y = 3, -X - 3*Y + 2*Z + T = 1, X >= 0, Y >= 0,
      Z >= 0, T >= 0}, inf(3*X + 2*Y - Z + 1, Inf).
```

The output is

```
Inf = 2,
Y=3-X,
X>=0,
X=<3,
T>=0,
Z=5-X-1/2*T,
X+1/2*T=<5 ?
```

The result `Inf = 2` indicates that the minimum value is 2 while the remainder of the answer is a simplified form of the constraint.

Neither simplification in $CLP(\mathcal{R})$ nor in SICStus Prolog is canonical. For example, using $CLP(\mathcal{R})$ we obtain different simplifications depending on the order in which variables appear in the constraint. For instance,

```
1 ?- 2*X + Y = Z, X - Y = W.
```

gives

```
Y = -0.666667*W + 0.333333*Z
X = 0.333333*W + 0.333333*Z
```

while

```
2 ?- Z = 2*X - Y, W = X - Y.
```

gives

```
X = W + Y
Z = 2*W + Y.
```

P2.1. Use dump to verify your answer to Exercises 2.1 and 2.3 above.

P2.2. Use dump to simplify $X = f(Y, Z) \land Z = g(a, Y)$ in terms of X. What do you think the symbol starting with an underscore in the output represents?

P2.3. Use sup in SICStus Prolog to find the maximum value of the function

$$X1 + X2 + X3 + X4 + X5$$

subject to the constraints

$$\begin{gathered} 3X1 + 2X2 + X3 = 1 \land \\ 5X1 + X2 + X3 + X4 = 3 \land \\ 2X1 + 5X2 + X3 + X5 = 4 \land \\ X1 \geq 0 \land X2 \geq 0 \land X3 \geq 0 \land X4 \geq 0 \land X5 \geq 0. \end{gathered}$$

Can you also find values for $X1$, $X2$, $X3$, $X4$, $X5$, at which this maximum value occurs?

P2.4. You can use projection to find maximum and minimum values by examining the resulting expression by hand. Use dump in $CLP(\mathcal{R})$ to solve the previous problem.

2.11 Notes

Fourier elimination is sometimes referred to as Fourier-Motzkin elimination. The method goes back to a report by Fourier [48] from 1827. Motzkin is associated with the algorithm because he independently discovered the algorithm at a later date. Simplification algorithms used in constraint logic programming systems over linear real arithmetic constraints are based on Fourier elimination, see [73]. Strangely, most CLP systems employing tree constraints do not use a weakly projecting simplifier, such as tree_simplify, for simplifying tree constraints. Instead they simply return the equations resulting from unify of the form $x = t$ for each variable x of interest.

Techniques for solving optimization problems over the real and integer constraint domains have been extensively studied by mathematicians from the operations research community. For a comprehensive introduction to the main approaches see, for example, [116]. The simplex algorithm dates from 1951 and is due to Dantzig [37]. Schrijver [116] gives a detailed account of its history, pointing to forerunners of simplex used by, most notably, Fourier (1826) and de la Vallée Poussin (1910). The proper mathematical foundations for the algorithm were laid by Farkas (1895) and von Neumann (1947). Bland's anti-cycling rule was suggested in 1977 [12]. We describe the primal simplex algorithm, in a form close to Gauss-Jordan elimination. Modern constraint programming systems usually use a revised simplex algorithm [38] with explicit bounds handling (see for example [110]). This approach is well suited to incremental solving (see Chapter 10) and allows checks to be made on numerical stability. Recently interior point methods [144] have been developed for solving linear real arithmetic optimization problems. They are increasingly the method of choice for large problems, although they are not presently used in CLP systems because of difficulties with incremental use.

Efficient algorithms for determining equivalence or implication and for computing canonical forms have not been systematically studied for many constraint domains. However, these operations have been extensively studied in the case of Boolean constraints, in part because of the application of the operations to digital circuit design. See, for example, Brown [20] who describes several canonical forms. In particular, reduced ordered binary decision diagrams (ROBDDs) introduced by Bryant for circuit design applications have proved to be an efficient canonical representation of Boolean constraints [21]. Recently there has been increased interest in implication testing because of its application in concurrent constraint programming languages, which we discuss in Chapter 12. See, for instance, Podelski and Van Roy [102] who give an efficient algorithm for testing implication of tree constraints.