Chapter 10

Optimization in CSPs

10.1 Introduction

In previous chapters, we have looked at techniques for solving CSPs in which all solutions are equally good. In applications such as industrial scheduling, some solutions are better than others. In other cases, the assignment of different values to the same variable incurs different costs. The task in such problems is to find optimal solutions, where optimality is defined in terms of some application-specific functions. We call these problems Constraint Satisfaction Optimization Problems (CSOP) to distinguish them from the standard CSP in Definition 1-12.

Moreover, in many applications, the constraints are so tight that one normally cannot satisfy all of them. When this is the case, one may want to find compound labels which are as close to solutions as possible, where closeness may be defined in a number of ways. We call these problems Partial Constraint Satisfaction Problems (PCSP).

Relatively little research has been done in both CSOP and PCSP by the CSP research community. In this chapter, these problems will be formally defined, and relevant techniques for solving them will be identified.

10.2 The Constraint Satisfaction Optimization Problem

10.2.1 Definitions and motivation

All optimization problems studied in operations research are constraint satisfaction problems in the general sense, where the constraints are normally numerical. Here, we use the term Constraint Satisfaction Optimization Problems (CSOP) to refer to the standard constraint satisfaction problem (CSP) as defined in Definition 1-12,

plus the requirement of finding optimal solutions.

Definition 10.1:

> A **CSOP** is defined as a CSP (Definition 1-12) together with an optimization function f which maps every solution tuple to a numerical value:
>
> $$(Z, D, C, f)$$
>
> where (Z, D, C) is a CSP, and if S is the set of solution tuples of (Z, D, C), then
>
> $f: S \rightarrow$ numerical value.
>
> Given a solution tuple T, we call $f(T)$ the ***f*-value** of T. ■

The task in a CSOP is to find the solution tuple with the optimal (minimal or maximal) f-value with regard to the application-dependent optimization function f.

Resource allocation problems in scheduling are CSOPs. In many scheduling applications, finding just any solution is not good enough. One may like to find the most economical way to allocate the resources to the jobs, or allocate machines to jobs, maximizing some measurable quality of the output. These problems are CSOPs.

In order to find the optimal solution, one potentially needs to find all the solutions first, and then compare their f-values. A part of the search space can only be pruned if one can prove that the optimal solution does not lie in it — which means either no solution exists in it (which involves knowledge about solutions) or that the f-value in any solution in the pruned search space is sub-optimal (which involves knowledge about the f-values).

10.2.2 Techniques for tackling the CSOP

Finding optimal solutions basically involves comparing all the solutions in a CSOP. Therefore, techniques for finding all solutions are more relevant to CSOP solving than techniques for finding single solutions.

Among the techniques described in the previous chapters, solution synthesis techniques are designed for finding all solutions. Problem reduction methods discussed in Chapter 4 are in general useful for finding all solutions because they all aim at reducing the search space. The basic search strategies introduced in Chapter 5 are in general applicable to both finding all solutions and finding single solutions. Variable ordering techniques which aim at minimizing backtracking (the minimal width ordering) and minimizing the number of backtracks (the minimal bandwidth ordering) are more useful for finding single solutions than all solutions. On the other hand, the *fail first principle* (FFP) in variable ordering is useful for finding all solutions as well as single solutions, because it aims at detecting futility as soon as possible so as to prune off more of the search space.

Techniques for ordering the values are normally irrelevant when all solutions are required because in such a case, all values must be looked at. Values ordering will be useful in gather-information-while-searching strategies if more nogood sets can be discovered in searching one subtree rather than another, and one has the heuristics to order the branches (values) in order to maximize learning.

In the following sections, we shall introduce two important methods for tackling CSOPs which have not been introduced in this book so far. They are the branch and bound (B&B) algorithm and genetic algorithms (GAs). The former uses heuristics to prune off search space, and the latter is a stochastic approach that has been shown to be effective in combinatorial problems.

10.2.3 Solving CSOPs with branch and bound

In solving CSOPs, one may use heuristics about the f function to guide the search. Branch and bound (B&B), which is a general search algorithm for finding optimal solutions, makes use of knowledge on the f function. He we continue to use the term *solution tuple* to describe compound labels which assign values to all those variables satisfying all the constraints (Definition 1-13). Readers should note that a solution tuple here need not refer to the optimal solution in a CSOP. B&B is a well known technique in both operations research and AI. It relies on the availability of good heuristics for estimating the best values ('best' according to the optimization function) of all the leaves under the current branch of the search tree. If reliable heuristics are used, one could be able to prune off search space in which the optimal solution does not lie. Thus, although B&B does not reduce the complexity of a search algorithm, it could be more efficient than the chronological backtracking search. It must be pointed out, however, that reliable heuristics are not necessarily available. For simplicity, we shall limit our discussion to the depth first branch and bound strategy and its application to the CSOP in this section.

10.2.3.1 A generic B&B algorithm for CSOP

To apply the B&B to CSOP, one needs a heuristic function h which maps every compound label CL to a numerical value (h: $CL \rightarrow$ number). We call this value the ***h*-value** of the compound label. For the function h to be admissible, the h-value of any compound label CL must be an *over-estimation* (*under-estimation*) of the f-value of any solution tuple which projects to CL in a *maximization* (*minimization*) problem.

A global variable, which we shall refer to as the *bound*, will be initialized to minus infinity in a maximization problem. The algorithm searches for solutions in a depth first manner. It behaves like Chronological_Backtracking in Chapter 2, except that before a compound label is extended to include a new label, the h-value of the current compound label is calculated. If this h-value is less than the bound in a maximi-

zation problem, then the subtree under the current compound label is pruned. Whenever a solution tuple is found, its *f*-value is computed. This *f*-value will become the new bound if and only if it is greater than the existing bound in a maximization problem. When this *f*-value is equal to or greater than the bound, the newly found solution tuple will be recorded as one of the, or the best solution tuples so far. After all parts of the search space have been searched or pruned, the best solution tuples recorded so far are solutions to the CSOP.

The Branch_and_Bound procedure below outlines the steps in applying a depth-first branch and bound search strategy to solving the CSOP, where the maximum *f*-value is required. Minimization problems can be handled as maximization problems by substituting all *f*- and *h*-value by their negation. For simplicity, this procedure returns only one solution tuple which has the optimal *f*-value; other solution tuples which have the same *f*-value are discarded.

```
PROCEDURE Branch_and_Bound( Z, D, C, f, h );
/* (Z, D, C) is a CSP; f is the function on solution tuples, the f-value is
    to be maximized; h is a heuristic estimation of the upper-bound of
    the f-value of compound labels */
BEGIN
    /* BOUND is a global variable, which stores the best f-value found
        so far; BEST_S_SO_FAR is also a global variable, which
        stores the best solution found so far */
    BOUND ← minus infinity; BEST_S_SO_FAR ← NIL;
    BNB( Z, { }, D, C, f, h );
    return(BEST_S_SO_FAR);
END /* of Branch_and_Bound */


PROCEDURE BNB(UNLABELLED, COMPOUND_LABEL, D, C, f, h);
BEGIN
    IF (UNLABELLED = { }) THEN
        BEGIN
            IF (f(COMPOUND_LABEL) > BOUND) THEN
                BEGIN           /* only one optimal solution is returned */
                    BOUND ← f(COMPOUND_LABEL);
                    BEST_S_SO_FAR ← COMPOUND_LABEL;
                END;
        END;
    ELSE IF (h(COMPOUND_LABEL) > BOUND) THEN
        BEGIN
            Pick any variable x from UNLABELLED;
            REPEAT
                Pick any value v from D_x;
```

```
                Delete v from D_x;
                IF (COMPOUND_LABEL + {<x,v>} violates no con-
                    straints)
                THEN BNB(UNLABELLED – {x}, COMPOUND_LABEL
                    + {<x,v>}, D, C, f, h);
            UNTIL (D_x = { });
        END /* of ELSE IF */
END /* of BNB */
```

Note that the Branch_and_Bound procedure is only sound and complete if $h(CL)$ indeed returns an upper-bound of the *f*-value. If the heuristic h may underestimate the *f*-value, then the procedure may prune off search space where optimal solutions lie, which causes sub-optimal solution tuples to be returned.

The efficiency of B&B is determined by two factors: the quality of the heuristic function and whether a "good" bound is found at an early stage. In a maximization problem, if the *h*-values are always over-estimations of the *f*-values, then the closer the estimation is to the *f*-value (i.e. the smaller the *h*-value is without being smaller than the *f*-value), the more chance there will be that a larger part of the search space will be pruned.

A branch will be pruned by B&B if the *h*-value of the current node is lower than the bound (in a maximization problem). That means even with the heuristic function fixed, B&B will prune off different proportion of the search space if the branches are ordered differently, because different bounds could be found under different branches.

10.2.3.2 Example of solving CSOP with B&B

Figure 10.1 shows an example of a CSOP. The five variables x_1, x_2, x_3, x_4 and x_5 all have numerical domains. The *f*-value of a compound label is the summation of all the values taken by the variables. The task is to find the solution tuple with the maximum *f*-value.

Figure 10.2 shows the space explored by simple backtracking. Each node in Figure 10.2 represents a compound label, and each branch represents the assignment of a value to an unlabelled variable. The variables are assumed to be searched under the ordering: x_1, x_2, x_3, x_4 and x_5. As explained in the last section, B&B will perform better if a tighter bound is found earlier. In order to illustrate the effect of B&B, we assume that the branches which represent the assignment of higher values are searched first.

Figure 10.3 shows the space searched by B&B under the same search ordering as

Variables	Domains	Constraints
x_1	4, 5	
x_2	3, 5	$x_2 \neq x_1$
x_3	3, 5	$x_3 = x_2$
x_4	2, 3	$x_4 < x_3$
x_5	1, 3	$x_5 \neq x_4$

Task: to assign consistent values to the variables and maximize $\sum$(values assigned)

Figure 10.1 Example of a CSOP

simple backtracking. The h-value for a node is calculated as the values assigned so far plus the sum of the maximal values for the unlabelled variables. For example, the h-value of (<x_1,4> <x_2,5>) is 4 + 5 (the values assigned so far) plus 5 + 3 + 3 (the maximum values that can be assigned to x_3, x_4 and x_5), which is 20.

According to the Branch_and_Bound procedure described in the last section, the bound is initialized to minus infinity. When the node for (<x_1,5> <x_2,3> <x_3,3> <x_4,2> <x_5,3>) is reached, the bound is updated to (5 + 3 + 3 + 2 + 3 =) 16. This bound has no effect on the left half of the search tree in this example. When the node for (<x_1,4> <x_2,5> <x_3,5> <x_4,3> <x_5,1>) is reached, the bound is updated to 18. When the node for (<x_1,4> <x_2,5> <x_3,5> <x_4,2> <x_5,3>) is reached, the bound is updated to 19. When the node (<x_1,4> <x_2,3>) is examined, its h-value (which is 18) is found to be less than the current bound (which is 19). Therefore, the subtree under the node (<x_1,4> <x_2,3>) is pruned. After this pruning, (<x_1,4> <x_2,5> <x_3,5> <x_4,2> <x_5,3>) is concluded to be the optimal solution. In Figure 10.3, 21 nodes have been explored, as opposed to 27 nodes in Figure 10.2.

Figure 10.4 shows the importance of finding a tighter bound at an earlier stage. In Figure 10.4, we assume that <x_1,4> is searched before <x_1,5>, all other things remaining the same. The optimal solution is found after 10 nodes have been explored. The bound 19 is used to prune the subtree below (<x_1,4> <x_2,3>) (whose h-value is 18) and (<x_1,5> <x_2,3> <x_3,3>) (whose h-value is 18). Note that if a single solution is required, the subtree below (<x_1,5> <x_2,3>) will be pruned because the h-value of (<x_1,5> <x_2,3>) is just equal to the bound. Only 17 nodes have been explored in Figure 10.4.

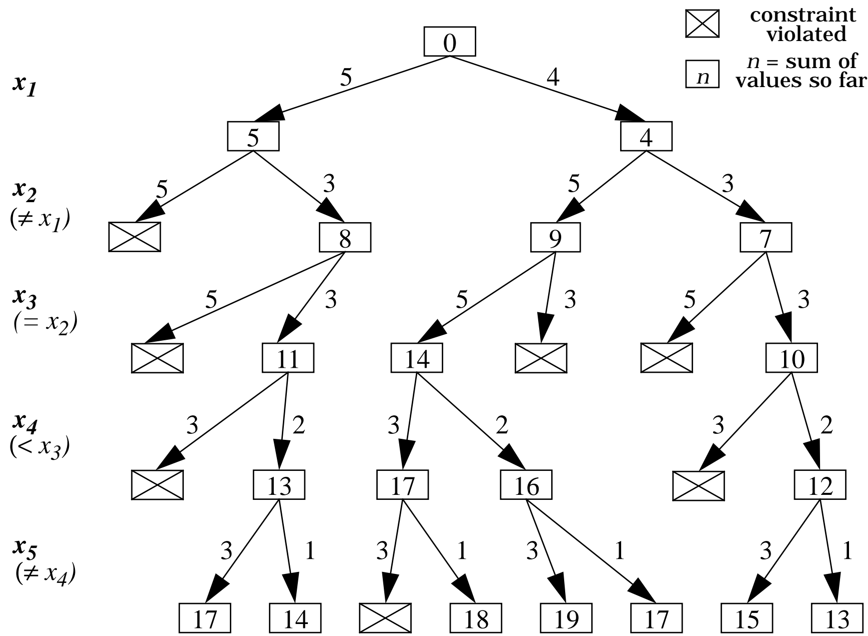


Figure 10.2 The space searched by simple backtracking in solving the CSOP in Figure 10.1 (branches which represent the assignment of greater values are searched first)

10.2.4 Tackling CSOPs using Genetic Algorithms

Like CSPs, CSOPs are NP-hard by nature. Unless a B&B algorithm is provided with a heuristic which gives fairly accurate estimations of the *f*-values, it is unlikely to be able to solve very large problems. Besides, good heuristic functions are not always available, especially when the function to be optimized is not a simple linear function.

Genetic algorithms (GAs) are a class of stochastic search algorithms which borrow their ideas from evolution in nature. GAs have been demonstrated to be effective in a number of well known and extensively researched combinatorial optimization problems, including the travelling salesman problem (TSP), the quadratic assignment problem (QAP), and applications such as scheduling. This section describes the GAs, and evaluates their potential in solving CSOPs. Preliminary research has suggested that GAs could be useful for large but loosely constrained CSOPs where near-optimal solutions are acceptable.

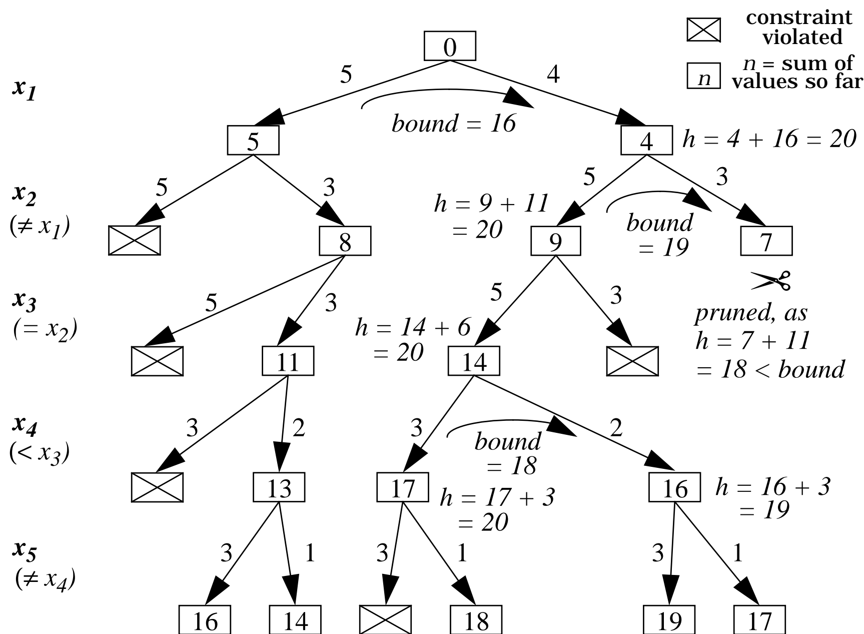


Figure 10.3 The space searched by Branch & Bound in solving the CSOP in Figure 10.1: branches which represent the assignment of greater values are searched first; $h(\mathbf{x})$ = value assigned + $\sum$(maximal values for the unlabelled variables)

10.2.4.1 Genetic Algorithms

The idea of GAs is based on evolution, where the fitter an individual is, the better chance it has to survive and produce offspring and pass its genes on to future generations. In the long run, the genes which contribute positively to the fitness of an individual will have a better chance of remaining in the population. This will hopefully improve the average fitness of the population and improve the chance of fitter strings emerging.

To apply this idea to optimization problems, one must first be able to represent the candidate solutions as a string of **building blocks**. (In some other GA applications, a candidate solution is represented by a set of strings rather than a single string.) Each building block must take a value from a finite domain. Many researches focus on using binary building blocks (i.e. building blocks which can only take on 0 or 1 as their values). To apply GAs to optimization problems, one must also be able to express the optimization function in the problem as a function of the values being taken by the building blocks in a string. The optimization function, which need not

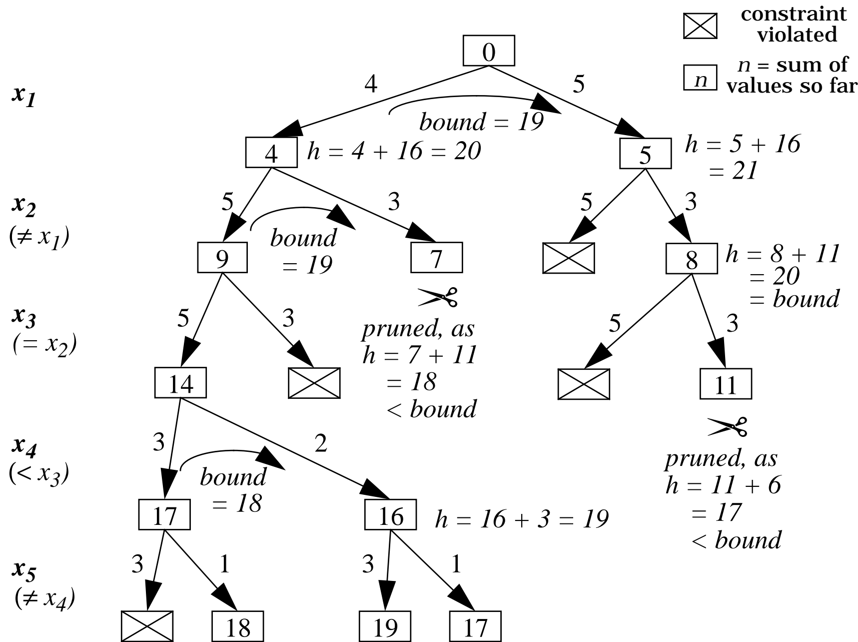
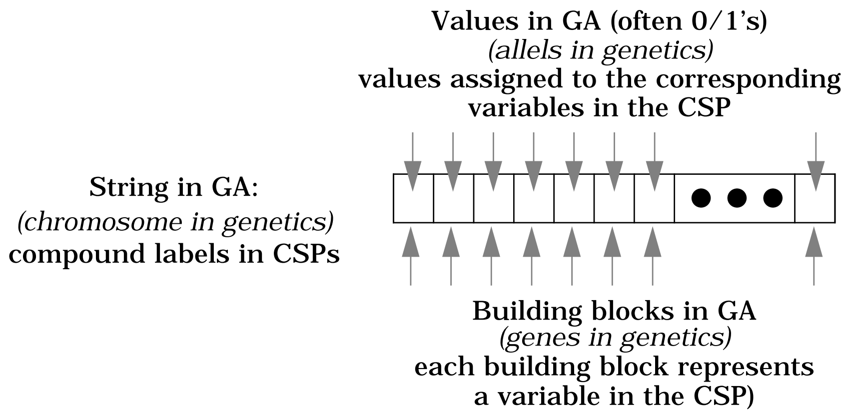


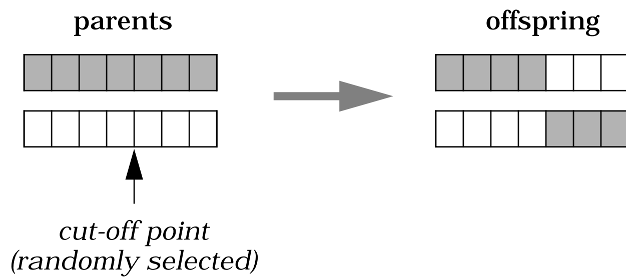
Figure 10.4 The space searched by Branch & Bound in solving the CSOP in Figure 10.1 when good bounds are discovered early in the search (note that the node for (<x_1,5> <x_2,3>) would have been pruned if single solution is required); $h(x)$ = values assigned + $\sum$ maximal values for the unlabelled variables

be linear, is referred to in GAs as the **evaluation function** or the **fitness function**. A string is analogous to a *chromosome* in biological cells, and the building blocks are analogous to *genes* (see Figure 10.5(a)). The values taken by the building blocks are called *allels*. The value of the string returned by the evaluation function is referred to as the **fitness** of the string.

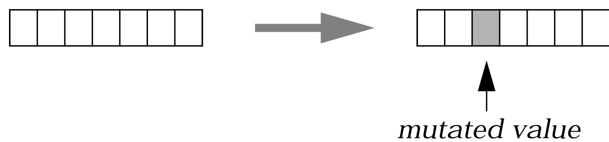
To apply GAs to optimization problems, a population of candidate solutions is generated and maintained. In a simple implementation, random strings could be generated in the initialization process. A more sophisticated initialization may ensure that all allels of all the building blocks are present in the initial population. The size of the population is one of the parameters of the GA which has to be set by the program designer. After the initial population is generated, the population is allowed to *evolve* dynamically. One commonly used control flow is the *canonical GA*, shown in Figure 10.6.



(a) Representation of candidate solutions of PCSPs in GA



(b) Crossover — the building block of the parents are exchanged to form the new offspring



(c) Mutation — a random building block is picked, and its value changed

Figure 10.5 Possible objects and operations in a Genetic Algorithm

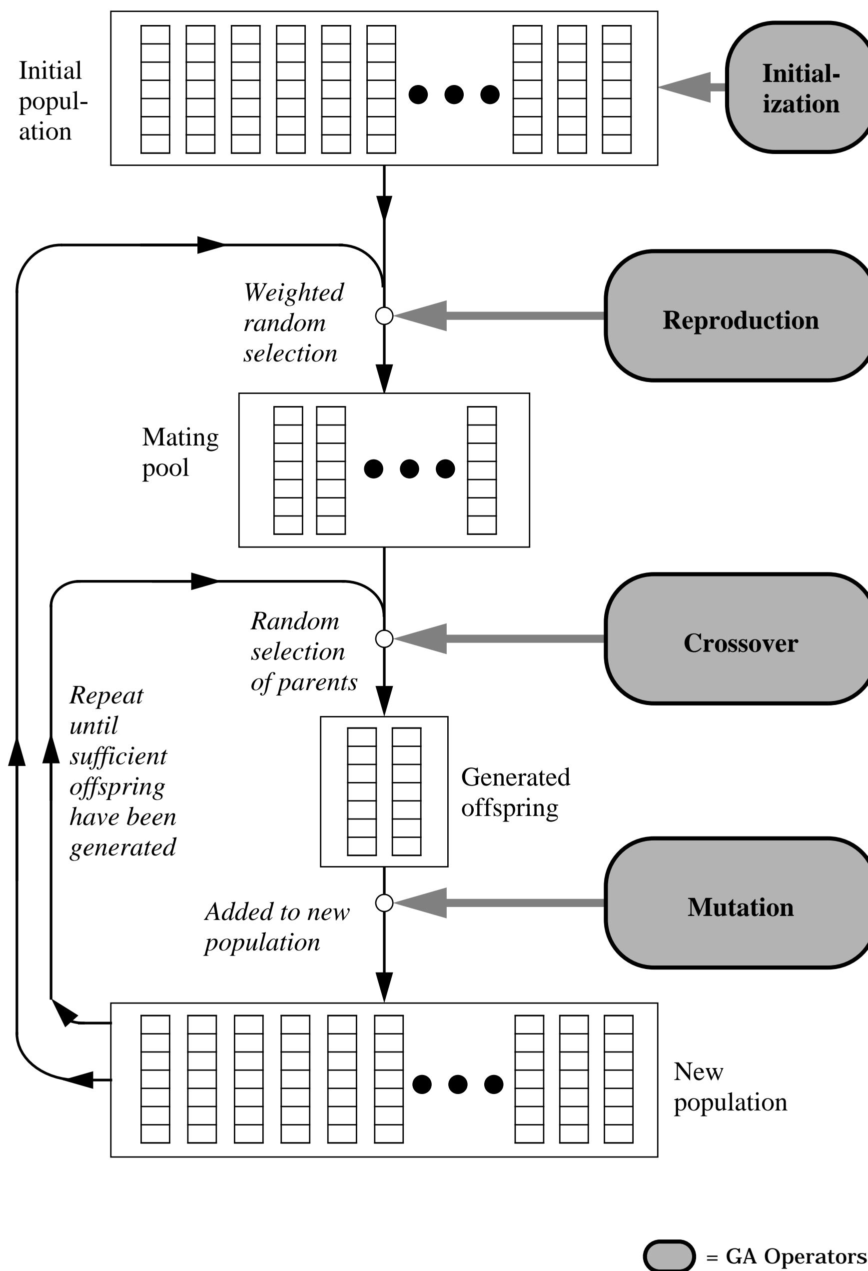


Figure 10.6 Control flow and operations in the Canonical Genetic Algorithm

Certain members of the population are selected and put into a set called the *mating pool*. Members are normally selected weighted randomly — the fitter a member is (according to the evaluation function) in the population, the greater chance it has of being selected. This operation is called *reproduction*. The size of the mating pool is another parameter of the GA which must be set by the program designer.

After the mating pool has been formed, the old population is discarded and the new population is generated. Normally, a pair of new strings, called *offspring* are generated from a pair of *parent* strings from the mating pool. To generate a pair of offspring, parents are often picked from the mating pool randomly. Offspring are normally generated by combining the building blocks in two parent strings. This operation is called *crossover*. The simplest form of crossover is to pick a random cutting point, and exchange the building blocks of the two parent strings at that point, as illustrated in Figure 10.5(b). For example, if the parents are:

parent 1: 11001100
parent 2: 01010101

and the cutting point is between the 4-th and the 5-th bits, then the offspring would be:

offspring 1: 11000101
offspring 2: 01011100

Occasionally, individual building blocks of some offspring are picked and have their allels modified. This operation is called *mutation*. Normally, mutation is allowed to occur infrequently. The purpose of introducing mutation is to allow the building blocks to take on new allels, which could form part of the optimal solution, but is missing in the current population.

Since the stronger strings get more chances to reproduce, it is possible that after a number of iterations (sometimes called *generations* or *cycles*), all the strings in the population will become identical. We call this phenomenon *convergence* of the population. When this happens, there is no point in allowing the GA to continue, because apart from the occasional mutations, the population will remain unchanged.

Apart from the population being converged, a GA may be terminated when it runs out of resources, e.g. time. Unless the population has converged, the longer a GA is allowed to run, the more search space it is allowed to explore, and in principle it has a better chance of finding better solutions. The CGA procedure below shows the pseudo codes of the canonical GA:

```
PROCEDURE CGA(f, PZ, ZMP, MutationRate)
/* f = the evaluation function; PZ = Population Size; ZMP = Size of the
   Mating Pool */
```

```
BEGIN
    Population ← Initialization(PZ);
    REPEAT
        Mating_Pool ← Reproduction(f, ZMP, Population);
        Population ← { };
        REPEAT
            Parent1 ← random element from Mating_Pool;
            Parent2 ← random element from Mating_Pool;
            Offspring ← Crossover(Parent1, Parent2);
            FOR each element os in Offspring DO
                IF (random number (between 0 and 1) ≤ MutationRate)
                THEN os ← Mutation(os);
            Population ← Population + Offspring;
        UNTIL size_of(Population) = PZ;
    UNTIL (converged(Population) ∨ resources_exhausted);
END /* of CGA */
```

The evaluation function, population size, size of mating pool and mutation rates are parameterized in CGA. Procedures for population initialization, reproduction, crossover and mutation in a GA are collectively called *GA operators*. The control flow and the GA operators are shown in Figure 10.6.

GAs may vary in many ways. A population dynamic different from the Canonical GA may be used. Instead of generating a mating pool and discarding the old population (as described above), the *Steady State GA* removes strings from and adds strings to the current population in each cycle.

Within a particular control flow, GAs may still vary in their operators. One may add new operators to the above mentioned ones. Besides, it is possible to perform hill-climbing within or after applying the crossover operator.

Under a fixed set of GA operators, there are still a number of parameters that one may set; for example:

- the size of the population;
- the size of the mating pool;
- the frequency of mutation;
- the number of offspring to be generated from each pair of parents;
- the time the GA is allowed to run;
- the maximum number of iterations the GA is allowed to run;
- etc.

The effectiveness of a GA in an application, or in a particular problem, is dependent on a large number of factors, including the representation, the definition of the eval-

uation function and both the control and the operators and parameters used. Finding the right combination is the key to the success of a GA, hence the focus of much current research.

10.2.4.2 Effectiveness of GAs

The effectiveness of GAs is based on the **schemata theorem**, or the **fundamental theorem**. To help in studying the effectiveness of GAs, the concept of a **schema** is introduced. A schema in a GA representation is a partially instantiated string. For example, assume that a GA representation uses strings of seven building blocks, which all take binary allels (0 or 1), and let ∗ represent a wildcard. The conceptual string ∗1∗∗01∗ is a schema with blocks 2, 5 and 6 instantiated. The *order* of a schema is the number of instantiated building blocks. The *defining length* of a schema is the distance between the first and the last instantiated building block. For example, the order of the schema ∗1∗∗01∗ is 3, and the defining length of it is (6 − 2 =) 4. A schema covers a whole set of instantiations, e.g. ∗1∗∗01∗ covers 0100010, 1111011, etc.

The *fitness of a schema* in a population is the average fitness of strings covered by that schema. The schema has above (below) average fitness if its fitness is above (below) the average fitness of the population. The effect of reproduction is to encourage above average schemata to appear in successive generations. The effect of crossover is to allow old schemata to be broken down and new schemata to be created.

The schema theorem is a probabilistic approach to estimating the chance of a schema surviving in the next generation. It shows that simple GAs (such as the canonical GA) will give above average schemata which have a lower order and a shorter defining length exponential chance of appearing in successive generations. Although this does not guarantee that the optimal solution will be generated in some generations, the hypothesis is that above average schemata would have a better chance of appearing in the optimal solution. This hypothesis is called the *building block hypothesis*.

For a GA to be effective, it has to be able to combine exploration and exploitation appropriately. Exploration means allowing randomness in the search. Exploiting means using the fitness values and the result of the previous iterations to guide the search. Without allowing enough exploration, the population is likely to converge in local sub-optimal. That is why randomness is introduced in all the above mentioned GA operators. Without allowing enough exploitation, the search is unguided, and therefore is no better than random search. That is why offspring are generated from members of the current population, members are picked weighted randomly to form the mating pool, mutation is not allowed too frequently, etc.

10.2.4.3 Applying GAs to CSOPs

Like most other stochastic search algorithms, GAs do not guarantee to find optimal solutions. However, many real life problems are intractable with complete methods. For such problems, near-optimal solutions are often acceptable if they can be generated within the time available. GAs offer hope in solving such problems. A GA is worth looking at as a tool for solving CSOPs because (a) GAs have been successful in many optimization problems, and (b) solution tuples in CSPs can naturally be represented by strings in GAs, as explained below.

For a CSOP with *n* variables, each string can be used to represent an *n*-compound tuple, where the building blocks represent the variables (in fixed order) each of which can take a value from a finite domain. Each schema in GA represents a compound label in a CSP.

For example, if there are five variables in the CSOP, x_1, x_2, x_3, x_4 and x_5, then a string of five building blocks will be used in GA to represent the 5-compound labels in the CSOP. If the variables are given the above ordering in a GA representation, then the compound label ($<x_1,a> <x_3,b>$) would be represented by the schema *a**b**, where * represents a wildcard.

What cannot be represented explicitly in a string are the constraints in a CSOP. To ensure that a GA generates legal compound labels (compound labels which satisfy the constraints), one may use one of the following strategies:

(a) make sure that the population contains only those strings which satisfy the constraints (by designing the initialization, crossover and mutation operators appropriately); or
(b) build into the evaluation function a *penalty function* which assigns low fitness values to strings that violate constraints. This effectively reduces the chance of those strings which violate certain constraints to reproduce.

According to the analysis in Chapter 9 (see Table 9.1), loosely constrained problems where all solutions are required are hard by nature. (This is because in loosely constrained problems a larger part of the search space contains legal compound labels, or less search space can be pruned.) In principle, the *f*-values of all solution tuples must be compared in a CSOP, and therefore a CSOP belongs to the category of CSPs where all solutions are required. When CSOPs are tightly constrained, one could use the constraints to prune off part of the search space. When the CSOP is loosely constrained, many solution tuples exist, and therefore one can easily use strategy (a) in a GA. So GAs fill the slot in Table 9.1 where no other methods so far described in this book can go. For tightly constrained CSOPs, strategy (b) can be used to handle the constraints.

10.3 The Partial Constraint Satisfaction Problem

10.3.1 Motivation and definition of the PCSP

Study of the *partial constraint satisfaction problem* (PCSP) is motivated by applications such as industrial scheduling, where one would normally like to utilize resources to their full. The constraints in a problem are often so tight that solutions are not available. Often what a problem solver is supposed to do is find near solutions when the problem is over-constrained, so that it or its user will know how much the constraints should be relaxed. In other applications, the problem solver is allowed to violate some constraints at certain costs. For example, shortage in manpower could sometimes be met by paying overtime or employing temporary staff; shortage in equipment could be met by hiring, leasing, etc. In these applications, one would often like to find near-solutions when the problem is over-constrained. We call such problems PCSPs.

Here we shall first formally define the PCSP. In the next section, we shall identify techniques which are relevant to it.

Definition 10.2:

> A **partial constraint satisfaction problem** (**PCSP**) is a quadruple:
>
> $$(Z, D, C, g)$$
>
> where (Z, D, C) is a CSP, and g is a function which maps every compound label to a numerical value, i.e. if *cl* is a compound label in the CSP then:
>
> g: $cl \rightarrow$ numerical value
>
> Given a compound label *cl*, we call $g(cl)$ the ***g*-value** of *cl*. ■

The task in a PCSP is to find the compound label(s) with the optimal g-value with regard to some (possibly application-dependent) optimization function g.

The PCSP can be seen as a generalization of the CSOP defined above, since the set of solution tuples is a subset of the compound labels. In a maximization problem, a PCSP (Z, D, C, f) is equivalent to a CSOP (Z, D, C, g) where:

$g{:}(cl) = \mathrm{f}(cl)$ if *cl* is a solution tuple
$g{:}(cl) = -\infty$ otherwise ($g{:}(cl) = \infty$ in a minimization problem)

10.3.2 Important classes of PCSP and relevant techniques

PCSPs may take various forms, depending on their optimization functions (gs), and therefore it is difficult to name the relevant techniques. In the worst case, the whole search space must be searched because unlike in CSOPs, one cannot prune any part of the search space even if one can be sure that every compound label in it violates

some constraints. Therefore, heuristics on the satisfiability of the problem become less useful for pruning off search spaces. On the other hand, heuristics on the optimization function (i.e. estimation of the g-values) are useful for PCSPs. When such heuristics are available, the best known techniques for solving a PCSP is B&B, which have been explained in Section 10.2.3. In this section, we shall introduce two classes of PCSPs which are motivated by scheduling.

10.3.2.1 The minimal violation problem

In a CSP, if one is allowed to violate the constraints at some costs, then the CSP can be formalized as a PCSP where the optimization function g is one which maps every compound label to a non-positive numerical value. The task in such problems is to find n-compound labels (where n is the number of variables in the problem) which violate the minimum amount of constraints. We call this kind of problems *minimal violation problems* (MVPs).

Definition 10.3:

A **minimal violation problem** (**MVP**) is a quadruple:

$$(Z, D, C, g)$$

where (Z, D, C) is a CSP, and g is a function which maps every compound label to a number:

$g(cl)$ = numerical value if cl is an n-compound label and $n = |Z|$
= infinity otherwise ■

The task in a MVP (Z, D, C, g) where $|Z| = n$ is to minimize $g(cl)$ for all n-compound labels cl.

MVPs can be found in scheduling, where constraints on resources can be relaxed (e.g. by acquiring additional resources) at certain costs. In over-constrained situations, the scheduling system may want to find out the minimum cost of scheduling all the jobs rather than simply reporting failure. For such applications, a function, call it c, maps every constraint to a relaxation cost:

$$c : C \rightarrow \text{numerical value}$$

The optimization function in the PCSP is then the sum of all the costs incurred in relaxing the violated constraints:

$$g(cl) = \sum_{C_S \in C \wedge \neg satisfies(cl, C_S)} c(C_S)$$

Another example of the MVP is graph matching (see Section 1.5.6 in Chapter 1) where inexact matches are acceptable when exact matches do not exist. This will be the case when noise exists in the data, as would be the case in many real life appli-

cations. When the knowledge of the labels on the nodes and edges are unreliable, inexact matching will be acceptable.

Among the techniques covered in this book, branch and bound is applicable to the MVP when good heuristics are available. The heuristic repair method, which uses the min-conflict heuristic, is one relevant technique for solving the MVP. Although the min-conflict heuristic introduced in Chapter 6 assumes the cost of violating each constraint to be 1, modifying it to reflect the cost of the violation of the constraints should be straightforward. Instead of picking values which violate the least number of constraints, one could attempt to pick values which incur the minimum cost in the constraints that they violate.

The GENET approach (Section 8.3.2), which basically adds learning to a hill-climbing algorithm using the min-conflict heuristic, is also a good candidate to the MVP. To apply GENET to MVPs, one may initialize the weights of the connections to the negation of the costs of violating the corresponding constraint.

10.3.2.2 The maximal utility problem

In some applications, no constraint can be violated. When no solution tuple can be found, the problem solver would settle for k-compound labels (with k less than the total number of variables in the problem) which have the greatest "utility", where utility is user defined. We call this kind of problems *maximal utility problems* (MUPs).

Definition 10.4:

A **maximal utility problem** (MUP) is a quadruple:

$$(Z, D, C, g)$$

where (Z, D, C) is a CSP, and g is a function which maps every compound label to a number:

$g(cl)$ = numerical value if the compound label cl violates no constraint
= minus infinity otherwise ■

The task in a MUP (Z, D, C, g) is to maximize $g(cl)$ for all k-compound labels cl.

MUPs can also be found in resource allocation in job-shop scheduling. In some applications, one would like to assign resources to tasks, satisfying constraints which must not be violated. For example, the capacity of certain machines cannot be exceeded; no extra manpower can be recruited to do certain jobs within the available time. A "utility", for example sales income, may be associated with the accomplishment of each job. If one cannot schedule to meet all the orders, one would like to meet the set of orders whose sum of utility is maximal. If all jobs have a uniform utility, then the task becomes "to finish as many jobs as possible".

Among the techniques which we have covered in this book, the branch and bound algorithm is applicable to the MUP when heuristics are available.

Problem reduction is also applicable to MUPs. Since no constraint must be violated by the optimal solution, values for the unlabelled variables which are not compatible with the committed labels can be removed. Hence, techniques such as forward checking and arc-consistency lookahead could be expected to search a smaller space. However, maintaining higher level of consistency (see Figure 3.7) may not be effective in MUP solving. This can be explained by the following example. Let us assume that there are three variables x, y and z in a MUP (whose utility function is unimportant to our discussion here), and the constraints on the variables are:

$C_{x,y}$: $x = y$
$C_{y,z}$: $y = z$

Now assume that we have already committed to: $x = a$. We can remove all the values b from the domain of y (D_y) such that $b \neq a$, because (<x,a> <y,b>) violates the constraint $C_{x,y}$ (and therefore will not be part of the optimal compound label). However, one cannot remove all the values b such that $b \neq a$ from the domain of z, although $x = y$ and $y = z$ together implies $x = z$. This is because the optimal solution need not contain a label for y. This example illustrates that achieving path-consistency maintenance may not be useful in MUPs.

Most of the variable ordering heuristics described in Chapter 6 are applicable to problems where all solutions are required, and therefore are applicable to MUPs. The minimal width ordering (MWO) heuristic, the minimal bandwidth ordering (MBO) heuristic, and the fail first principle (FFP) attempt to improve search efficiency by different means. To recapitulate, the MWO heuristic attempts to reduce the need to backtrack; the MBO heuristic attempts to reduce the distance of backtracking; the FFP attempts to prune off as much search space as possible when backtracking is required.

Heuristics on the g-values are useful for ordering the variables and their values in branch and bound. This follows the point made earlier that the efficiency of branch and bound is affected by the discovery of tight bounds at early stages.

Solution synthesis techniques for CSPs would only be useful for MUPs if they synthesized the set of all legal compound labels (compound labels which do not violate any constraints). Among the solution synthesis algorithms introduced in Chapter 9, only Freuder's algorithm qualifies under this criteria. The Essex Algorithms will not generate all legal compound labels, and therefore could miss solutions for MUPs. Similarly, not every legal compound label has a path in the solution graph generated by Seidel's invasion algorithm — therefore, the solution for a MUP may not be represented in Seidel's solution graph.

Hill-climbing could be applicable to MUPs if appropriate heuristic functions (for guiding the direction of the climb) are available. In MUPs, one would like to hill-climb in the space of legal compound labels. Since the min-conflict heuristic aims at reducing the total number of constraints violated, it may not be very relevant to MUPs. Instead, one might need to use a heuristic which maximally increases the utility. The GENET approach takes the min-conflict heuristic, and therefore needs modifications if it were to be used to tackle MUPs.

10.4 Summary

In this chapter, we have looked at two important extensions of the standard CSP motivated by real life applications such as scheduling applications. We have extended the standard CSP to the *constraint satisfaction optimization problem* (CSOP), CSPs in which optimal solutions are required. Most CSP techniques which are applicable to finding all solutions are relevant to solving CSOPs. Examples of such techniques are solution synthesis, problem reduction and the fail first principle. Such techniques are more effective when the problem is tightly constrained.

The most general tool for solving CSOPs is *branch and bound* (B&B). However, since CSPs are NP-hard in general, complete search algorithms may not be able to solve very large CSOPs. Preliminary research suggests that *genetic algorithms* (GAs) might be able to tackle large and loosely constrained CSOPs where near-optimal solutions are acceptable.

The CSOP can be seen as an instance of the *partial constraint satisfaction problem* (PCSP), a more general problem in which every compound label is mapped to a numerical value. Two other instances of PCSPs are the *minimal violation problem* (MVP) and the *maximal utility problem* (MUP), which are motivated by scheduling applications that are normally over-constrained.

A *minimal violation problem* (MVP) is one in which the task is to find a compound label for all the variables such that the minimum weighted constraints are violated. Since solutions may (and are most likely to) violate certain constraints, standard CSP techniques such as problem reduction, variables ordering and solution synthesis are not applicable to the MVP. B&B is the most commonly used technique for tackling MVPs. The effectiveness of B&B on MVPs relies on the availability of good heuristics on the function to be optimized. For large problems, and when heuristics are not available for B&B, completeness and optimality are often sacrificed for tractability. In this case, hill-climbing strategies (such as the *heuristic repair method*) and connectionist approaches (such as GENET) have been proposed.

A *maximal utility problem* (MUP) is a PCSP in which the objective is to find the compound label that has the greatest utility — for example, to label as many variables as possible while ensuring that no constraint is violated. Standard CSP tech-

niques, including problem reduction and variables and values ordering, are applicable to MUPs. For problems where near optimal solutions are acceptable, it is possible to tackle MUPs with hill-climbing approaches.

10.5 Bibliographical Remarks

The CSOP and the PCSP are motivated by real life problems such as scheduling. In the CSP research community, research in CSOP and PCSP is not as abundant as research in the standard CSP. The branch and bound algorithm is a well known technique for tackling optimization problems; for example, see Lawler & Wood [1966], Hall [1971], Reingold *et al*. [1977] and Aho *et al*. [1983].

The field of Genetic Algorithms (GAs) was founded by Holland [1975]. Theories (such as the schemata theorem) and a survey of GAs can be found in Goldberg [1989] and Davis [1991]. Goldberg lists a large number of applications of GA. Muehlenbein and others have applied GAs to a number of problems, including the Travelling Salesman Problem (TSP), the Quadratic Assignment Problem (QAP) and scheduling problems, and obtained remarkable results [Mueh89] [BrHuSp89] [Fili92]. Tsang & Warwick [1990] report preliminary but encouraging results on applying GAs to CSOPs.

Freuder [1989] gives the first formal definition to the PCSP. In order to conform to the convention used throughout this book, a definition different from Freuder's has been used. Voss *et al*. [1990] attempt to satisfy as much constraints as possible, and dispose "soft-constraints" — constraints which represent preferences. Freuder and Wallace [1992] define the problem of "satisfying as many constraints as possible" as the *maximal constraint satisfaction problem* and tackle it by extending standard constraint satisfaction techniques. Hubbe & Freuder [1992] propose a *cross product representation* of partial solutions.The car sequencing problem as defined by Parrello *et al*. [1986] is a minimal violation problem (MVP), which has been modified to become a standard CSP by Dincbas *et al*. [1988b] (Dincbus' formulation of the car sequencing problem is described in Chapter 1). For references on both hill-climbing and connectionist approaches to CSP solving, readers are referred to Chapter 8. Applying GENET to PCSP is an ongoing piece of research. *Tabu Search* is a generic search strategy developed in operations research for optimization problems; e.g. see Glover [1989, 1990]. Instantiations of it have been applied to a number of applications and success has been claimed. The use of it in PCSP is worth studying.