Chapter 1
Introduction

Almost everybody who works in artificial intelligence should know something about the **Constraint Satisfaction Problem** (CSP). CSPs appear in many areas, for instance, vision, resource allocation in scheduling and temporal reasoning. The CSP is worth studying in isolation because it is a general problem that has unique features which can be exploited to arrive at solutions. The main objective of this book is to identify these properties and explain techniques for tackling CSPs.

In this chapter, we shall first define the standard CSP and the important concepts around it. To avoid ambiguity, concepts are defined both verbally and in first order predicate calculus (FOPC). The verbal definitions alone should be sufficient for readers who do not have enough knowledge of FOPC to understand the formal definitions.

1.1 What is a constraint satisfaction problem?

In this section, we shall give an informal definition of the Constraint Satisfaction Problem (CSP), along with two examples.

Basically, a CSP is a problem composed of a finite set of **variables**, each of which is associated with a finite **domain**, and a set of **constraints** that restricts the values the variables can simultaneously take. The task is to assign a value to each variable satisfying all the constraints.

1.1.1 Example 1 —The *N*-queens problem

The *N*-queens problem is a well known puzzle among computer scientists. Although the *N*-queens problem has very specific features (explained below) which can be exploited in solving it, it has been used extensively for illustrating CSP solving algorithms.

Given any integer *N*, the problem is to place *N* queens on *N* distinct squares in an $N \times N$ chess board, satisfying the constraint that no two queens should threaten each other. The rule is such that a queen can threaten any other pieces on the same row, column or diagonal. Figure 1.1 shows one possible solution to the 8-queens problem.

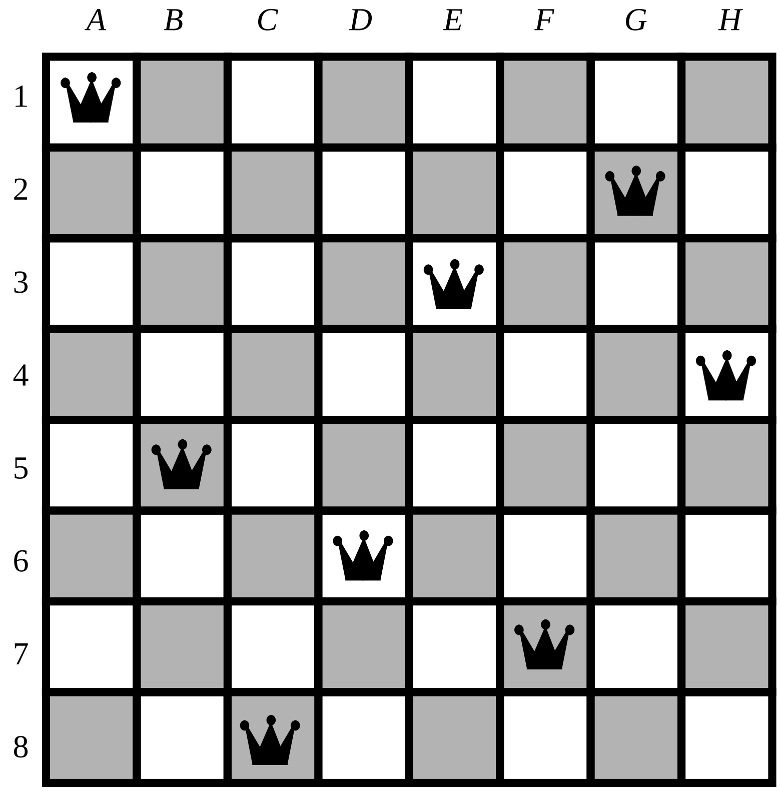


Figure 1.1 A possible solution to the 8-queens problem. The problem is to place eight queens on an 8×8 chessboard satisfying the constraint that no two queens should be on the same row, column or diagonal

One way to formalize the 8-queens problem as a CSP is to see it as a problem with eight variables (i.e. a finite set of variables), each of which may take a value from *A* to *H*. The task is to assign values to the variables satisfying the above-specified constraints.

1.1.2 Example 2 — The car sequencing problem

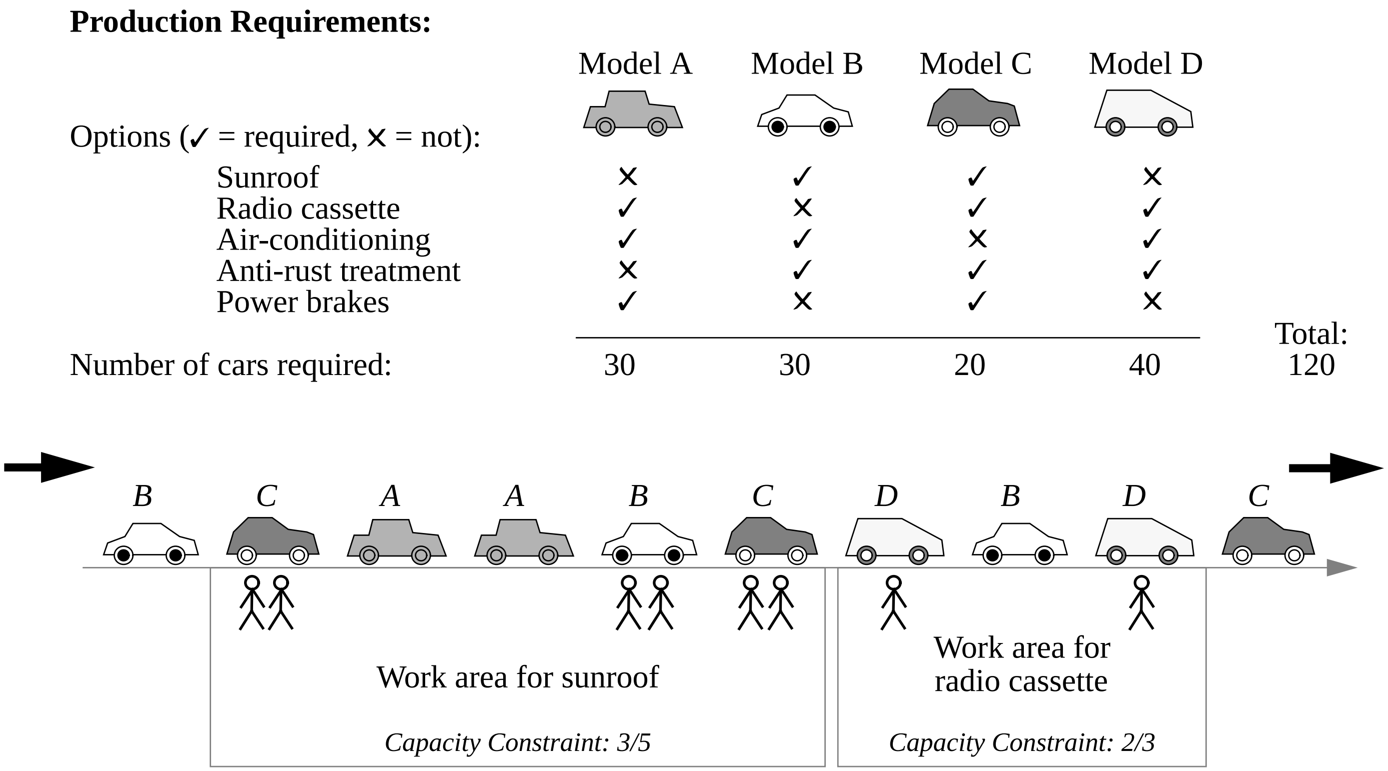
In modern car production, cars are placed on conveyor belts which move through different work areas. Each of these work areas specializes to do a particular job, such as fitting sunroofs, car radios or air-conditioners. When a car enters a work area, a team of engineers in that area travels with the car while working on it. The production line is designed so as to allow enough time for the engineers to finish this job while the car is in their work area. For example, if the time taken to install a sunroof is 20 minutes, and one car enters the conveyor belt every four minutes, then the work area for sunroof installation will be given a capacity of carrying (20 ÷ 4 =) five cars. Figure 1.2 shows a section of the production line.

A production line is normally required to produce cars of different models. The number of cars required for each model is called the *production requirement*. Since cars of different models require different options to be fitted, not every car requires work to be done in every work area. For example, one model may need air-conditioning and power brakes to be installed, but not a sunroof. The upper half of Figure 1.2 shows an example of the production requirement and the options required by four models. For example, 30 cars of model A are required, each of which needs a radio cassette, air-conditioning and power brakes to be fitted, but not a sunroof or anti-rust treatment.

Each work area is constrained by its resource constraint. For example, if three teams of engineers are designated to fitting sunroofs, and the sunroof work area has a space capacity for five cars, then the sunroof work area can cope with no more than three out of five cars requiring the fitting of sunroofs in any sub-sequence of cars on the conveyor belt. If more than three cars in any sequence of five cars require sunroofs, then the engineers would not have time to finish before the conveyor belt takes the cars away. The ratio 3/5 is called the *capacity constraint* of the work area for sunroof. In the example in Figure 1.2, the capacity constraints of the sunroof and radio cassette work areas are 3/5 and 2/3, respectively. We have not specified the capacity constraints of the other options there.

A *car-sequencing problem* is specified by the production and option requirements and the capacity constraints. Given the production requirements, the scheduler's task is to order the cars in the conveyor belt so that the capacity constraint of all the work areas are satisfied. In the above example, 120 cars of the four specified models must be scheduled. The sub-sequence shown in Figure 1.2 is:

$$..., B, C, A, A, B, C, D, B, D, C, ...$$



Production Requirements:

Options (✓ = required, ✗ = not):	Model A	Model B	Model C	Model D	
Sunroof	✗	✓	✓	✗	
Radio cassette	✓	✗	✓	✓	
Air-conditioning	✓	✓	✗	✓	
Anti-rust treatment	✗	✓	✓	✓	
Power brakes	✓	✗	✓	✗	
Number of cars required:	30	30	20	40	Total: 120

B
C
A
A
B
C
D
B
D
C
Work area for sunroof
Capacity Constraint: 3/5
Work area for radio cassette
Capacity Constraint: 2/3

Figure 1.2 Example of a car sequencing problem

To check that it satisfies the capacity constraint of the sunroof work area, one has to look at every sub-sequence of five cars, e.g.

B, *C*, *A*, *A*, *B*
C, *A*, *A*, *B*, *C*
A, *A*, *B*, *C*, *D*
.....

Careful examination should convince readers that the sub-sequence (*C*, *A*, *A*) in Figure 1.2 actually violates the capacity constraints of the radio cassette work area.

The car-sequencing problem is difficult when a large number (say, hundreds) of cars are to be scheduled. Failure has been reported in attempting to solve it using theorem provers and expert system tools [PaKaWo86] [Parr88]. It has been shown that this problem can be solved efficiently by formulating this problem as a CSP and applying CSP solving techniques to it [DiSiVa88b].

We have already explained that a CSP is composed of variables, domains and constraints. The car-sequencing problem can be formulated as a CSP in the following way. One variable is used to represent the car model of one position in the conveyor belt (i.e. if there are *n* cars to be scheduled, the problem consists of *n* variables). The domain of each variable is the set of car models, A to D in the above example. The task is to assign a value (a car model) to each variable (a position in the conveyor belt), satisfying both the production requirements and capacity constraints.

1.2 Formal Definition of the CSP

In this section, we shall give a more formal definition of the CSP. Before doing so, we first define *domains*, *assignments* (which we call *labels* below), and the concept of *satisfying* in terms of set relations.

1.2.1 Definitions of domain and labels

Definition 1-1:

> The **domain of a variable** is a set of all possible values that can be assigned to the variable. If x is a variable, then we use D_x to denote the domain of it. ■

When the domain contains numbers only, the variables are called **numerical variables**. The domain of a numerical variable may be further restricted to integers, rational numbers or real numbers. For example, the domain of an integer variable is an infinite set {1, 2, 3, ...}. The majority of this book focuses on CSPs with finite domains.

When the domain contains boolean values only, the variables are called **boolean**

variables. When the domain contains an enumerated type of objects, the variables are called **symbolic variables**. For example, a variable that represents a day of the week is a symbolic variable of which the domain is the finite set {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}.

Definition 1-2:

> A **label** is a variable-value pair that represents the assignment of the value to the variable. We use $<x, v>$ to denote the label of assigning the value v to the variable x. $<x, v>$ is only meaningful if v is in the domain of x (i.e. $v \in D_x$). ■

Definition 1-3:

> A **compound label** is the simultaneous assignment of values to a (possibly empty) set of variables. We use $(<x_1,v_1><x_2,v_2>...<x_n,v_n>)$ to denote the compound label of assigning $v_1, v_2, ..., v_n$ to $x_1, x_2, ..., x_n$ respectively. ■

Since a compound label is seen as a set, the ordering of the labels in our representation is insignificant. In other words, $(<x,a><y,b><z,c>)$ is treated as exactly the same compound label as $(<y,b><x,a><z,c>)$, $(<z,c><x,a><y,b>)$, etc. Besides, it is important to remember that a set does not have duplicate objects.

Definition 1-4:

> A ***k*-compound label** is a compound label which assigns k values to k variables simultaneously. ■

Definition 1-5:

> If m and n are integers such that $m \leq n$, then a **projection** of an n-compound label N to an m-compound label M, written as *projection(N, M)*, (read as: M is a projection of N) if the labels in M all appear in N.

$$\begin{aligned}&\forall\ (<x_1,v_1>...<x_m,v_m>), (<z_1,w_1>...<z_n,w_n>): \{x_1, ..., x_m\} \subseteq \{z_1, ..., z_n\} :\\ &\qquad \text{projection}((<z_1,w_1>...<z_n,w_n>),(<x_1,v_1>...<x_m,v_m>)) \equiv\\ &\qquad\quad <x_1,v_1>, ..., <x_m,v_m> \in \{<z_1,w_1>, ..., <z_n,w_n>\}\ \blacksquare^{1}\end{aligned}$$

For example, $(<a,1><c,3>)$ is a projection of $(<a,1><b,2><c,3>)$, which means the proposition *projection((<a,1><b,2><c,3>), (<a,1><c,3>))* is true.

[1] We use this notation to indicate that projection$((<z_1,w_1>...<z_n,w_n>), (<x_1,v_1>...<x_m,v_m>))$ is only defined if $(<z_1,w_1>...<z_n,w_n>)$ and $(<x_1,v_1>...<x_m,v_m>)$ are compound labels, and $\{x_1, ..., x_m\}$ is a subset of $\{z_1, ..., z_n.\}$. It is undefined otherwise.

Definition 1-6:

The **variables of a compound label** is the set of all variables which appear in that compound label:

variables_of($(<x_1,v_1> <x_2,v_2> \ldots <x_k,v_k>)$) $\equiv \{ x_1, x_2, \ldots, x_k \}$ ■

1.2.2 Definitions of constraints

A constraint on a set of variables is a restriction on the values that they can take simultaneously. Conceptually, a constraint can be seen as a set that contains all the legal compound labels for the subject variables; though in practice, constraints can be represented in many other ways, for example, functions, inequalities, matrices, etc., which we shall discuss later.

Definition 1-7:

A **constraint** on a set of variables is conceptually a set of compound labels for the subject variables. For convenience, we use C_S to denote the constraint on the set of variables S. ■

Definition 1-8:

The **variables of a constraint** is the variables of the members of the constraint:

variables_of($C_{x_1, x_2, \ldots, x_k}$) $\equiv \{ x_1, x_2, \ldots, x_k \}$ ■

"*Subsumed-by*" is a binary relationship on constraints.

Definition 1-9:

If m and n are integers such that $m \le n$, then an m-constraint M is **subsumed-by** an n-constraint N (written as *subsumed-by(M,N))* if for all elements c in M there exists an element d in N such that c is a projection of d:

$$\begin{aligned}
&\forall\ C_M, C_N\text{: } |M| = m \wedge |N| = n \wedge m \le n\text{:}\\
&\quad \text{subsumed-by}(C_M, C_N) \equiv\\
&\qquad (\forall\ (<x_1,v_1> \ldots <x_m,v_m>) \in C_M\text{:}\\
&\qquad\quad (\exists\ (<z_1,w_1> \ldots <z_n,w_n>) \in C_N\text{:}\\
&\qquad\qquad \text{projection}((<z_1,w_1> \ldots <z_n,w_n>),(<x_1,v_1> \ldots <x_m,v_m>)))) \blacksquare
\end{aligned}$$

Here $|M|$ and $|N|$ denote the number of variables in M and N respectively. If:

$C_M = \{(<a,1><c,3>), (<a,4><c,6>)\}$

and $C_N = \{(<a,1><b,2><c,3>), (<a,1><b,4><c,3>), (<a,4><b,5><c,6>)\}$, then the proposition subsumed-by(C_M, C_N) is true. In other words, if constraint *M* is subsumed by constraint *N*, then *N* is at least as restrictive as *M*. Apart from constraining the variables of *M*, *N* could possibly constrain other variables too (in the above example, C_M constrains variables *a* and *c*, while C_N constrains *a*, *b* and *c*).

1.2.3 Definitions of satisfiability

Satisfies is a binary relationship between a label or a compound label and a constraint.

Definition 1-10a:

> If the variables of the compound label *X* are the same as those variables of the elements of the compound labels in constraint *C*, then *X* **satisfies** *C* if and only if *X* is an element of *C*:
>
> $$\text{satisfies}(\ (<x_1,v_1><x_2,v_2>...<x_k,v_k>)\ ,\ C_{x_1, x_2, \ldots, x_k}\ ) \equiv$$
> $$(<x_1,v_1><x_2,v_2>...<x_k,v_k>) \in C_{x_1, x_2, \ldots, x_k}\ \blacksquare$$

For convenience, *satisfies* is also defined between labels and unary constraints.

Definition 1-10b:

> $$\text{satisfies}(\ <x,v>,\ C_x\ )\ \equiv\ (<x,v>) \in C_x\ \blacksquare$$

This allows us to write something like *satisfies*($<x,v>$, C_x) as well as *satisfies*(($<x,v>$),C_x). Following Freuder [1978], the concept of *satisfies(L,C)* is extended to the case when C is a constraint on a subset of the variables of the compound label L.

Definition 1-11:

> Given a compound label *L* and a constraint *C* such that the variables of *C* are a subset of the variables of *L*, the **compound label** *L* **satisfies constraint** *C* if and only if the projection of *L* onto the variables of *C* is an element of *C*:
>
> $$\forall\ x_1, x_2, ..., x_k: \forall\ v_1 \in\ D_{x_1}\ , v_2 \in\ D_{x_2}\ , ...\ , v_k \in\ D_{x_k}\ :$$
> $$(\forall\ S \subseteq \{x_1, x_2, ..., x_k\}:$$
> $$\text{satisfies}((<x_1,v_1>\ <x_2,v_2\ >...\ <x_k,v_k>),\ C_S) \equiv$$
> $$(\exists\ cl \in\ C_S\ : \text{projection}((<x_1,v_1>\ ...\ <x_k,v_k>),\ cl\ )))\ \blacksquare$$

In other words, when we say that L satisfies C, we mean that if C is a constraint on the variables $\{x_1, x_2, ..., x_k\}$ or its subset, then the labels for those variables in L are legal as far as C is concerned. For example, (<a,1> <b,2> <c,3> <d,4>) satisfies the constraint $C_{c,d}$ if and only if (<c,3> <d,4>) is a member of $C_{c,d}$:

$$C_{c,d} = \{..., (<c,3> <d,4>), ...\}.$$

1.2.4 Formal definition of constraint satisfaction problems

We stated earlier that a CSP is a problem with a finite set of variables, each associated to a finite domain, and a set of constraints which restrict the values that these variables can simultaneously take. Here we shall give this problem a more formal definition.

Definition 1-12:

A **constraint satisfaction problem** is a triple:

$$(Z, D, C)$$

where Z = a finite set of **variables** $\{x_1, x_2, ..., x_n\}$;

D = a function which maps every variable in Z to a set of objects of arbitrary type:

D: $Z \rightarrow$ finite set of objects (of any type)

We shall take D_{x_i} as the set of objects mapped from x_i by D. We call these objects possible **values** of x_i and the set D_{x_i} the **domain** of x_i;

C = a finite (possibly empty) set of **constraints** on an arbitrary subset of variables in Z. In other words, C is a set of sets of compound labels.

We use *csp*($\mathcal{P}$) to denote that $\mathcal{P}$ is a constraint satisfaction problem. ■

$C_{x_1, x_2, \ldots, x_k}$ restricts the set of compound labels that x_1, x_2, ..., and x_k can take simultaneously. For example, if the variable x can only take the values a, b and c, then we write C_x = {(<x,a>),(<x,b>),(<x,c>)}. (Note the difference between C_x and D_x: C_x is a set of labels while D_x is a set of values.) The value that x can take may be subject to constraints other than C_x. That means although <x,a> satisfies C_x, a may not be a valid value for x in the overall problem. To qualify as a valid label, <x,a> must satisfy all constraints which constrain x, including $C_{x,y}$, $C_{w,x,z}$, etc.

We focus on CSPs with finite number of variables and finite domains because, as illustrated later, efficient algorithms which exploit these features can be developed.

1.2.5 Task in a CSP

The task in a CSP is to assign a value to each variable such that all the constraints are satisfied simultaneously.

Definition 1-13:

A **solution tuple** of a CSP is a compound label for all those variables which satisfy all the constraints:

$$\forall\ \mathrm{csp}((Z, D, C))\text{: } \forall\ x_1, x_2, ..., x_n \in Z\text{: } (\forall\ v_1 \in D_{x_1}, v_2 \in D_{x_2}, ...,v_n \in D_{x_n} :$$
$$\mathrm{solution\_tuple}(\ (<x_1,v_1> <x_2,v_2> ...\ <x_n,v_n>), (Z, D, C)\ ) \equiv$$
$$((Z = \{x_1, x_2, ..., x_n\}) \wedge$$
$$(\forall c \in C\text{: satisfies}((<x_1,v_1> <x_2,v_2> ...\ <x_n,v_n>), c))) \blacksquare$$

A CSP is *satisfiable* if solution tuple exists. Depending on the requirements of an application, CSPs can be classified into the following categories:

(1) CSPs in which one has to find *any* solution tuple.
(2) CSPs in which one has to find *all* solution tuples.
(3) CSPs in which one has to find *optimal* solutions, where optimality is defined according to some domain knowledge. Optimal or near optimal solutions are often required in scheduling. This kind of problem will be discussed in Chapter 10.

1.2.6 Remarks on the definition of CSPs

The CSP defined above is sometimes referred to as the *Finite Constraint Satisfaction Problem* or the *Consistent Labelling Problem*. The term “constraint satisfaction” is often used loosely to describe problems which need not conform to the above definition. In some problems, variables may have infinite domains (e.g. numerical variables). There are also problems in which the set of variables could change dynamically — depending on the value that one variable takes, different sets of new variables could emerge. Though these problems are important, they belong to another class of problems which demand a different set of specialized techniques for solving them. We shall focus on the problems under the above definition until Chapter 10, where extensions of this definition are examined.

1.3 Constraint Representation and Binary CSPs

We said earlier that if $S = \{x_1, x_2, ..., x_k\}$, we use C_S or $C_{x_1, x_2, \ldots, x_k}$ to denote the constraint on S. C_S restricts the compound labels that the variables in S can simultaneously take.

A constraint can be represented in a number of different ways. Constraints on numerical variables can be represented by equations or inequalities; for example, a binary constraint $C_{x,y}$ may be $x + y < 10$. A constraint may also be viewed as a function which maps every compound label on the subject variables to true or false. Alternatively, a constraint may be seen as the set of all legal compound labels for the subject variables. This logical representation will be taken in this book as it helps to explain the concept of problem reduction (explained in Chapters 2 and 3) — where *tightening a constraint* means removing elements from the set. This choice of representation should not affect the generality of our discussions.

One way in which to represent binary constraints is to use matrices of boolean values. For example, assume that variable *x* can take values 1, 2 and 3, and variable *y* can take values 4, 5, 6 and 7. The constraint on *x* and *y* which states that "*x* + *y must be odd*" can be represented by a matrix, as shown in Figure 1.3.

$$\begin{array}{cc|cccc}
\multicolumn{2}{c|}{C_{xy}} & & & y & \\
 & & 4 & 5 & 6 & 7 \\
\hline
 & 1 & 1 & 0 & 1 & 0 \\
x & 2 & 0 & 1 & 0 & 1 \\
 & 3 & 1 & 0 & 1 & 0
\end{array}$$

Figure 1.3 matrix representing the constraint between *x* and *y*

The matrix in Figure 1.3 represents the fact that:

(<*x*,1><*y*,4>)
(<*x*,1><*y*,6>)
(<*x*,2><*y*,5>)
(<*x*,2><*y*,7>)
(<*x*,3><*y*,4>)
(<*x*,3><*y*,6>)

are all the compound labels that variables *x* and *y* can take.

Since a lot of research focuses on problems with unary and binary constraints only, we define the term *binary constraint problem* for future reference.

Definition 1-14:

> A **binary CSP**, or **binary constraint problem**, is a CSP with unary and binary constraints only. A CSP with constraints not limited to unary and binary will be referred to as a **general CSP**. ■

It is worth pointing out that all problems can be transformed into binary constraint problems, though whether one would benefit from doing so is another question. In general, if x_1, x_2, ..., x_k are k variables, and there exists a k-ary constraint CC on them, then this constraint can be replaced by a new variable W and k binary constraints. The domain of W is the set of all compound labels in CC (we mentioned earlier that we see constraints as sets of compound labels). Each of the k newly created binary constraints connects W and one of the k variables x_1 to x_k. The binary constraint which connects W and a variable x_i requires x_i to take a value which is projected from some values in W. This could be illustrated by the example in Figure 1.4. Let x, y and z be three variables in which the domains are all {1, 2}, and there exists a 3-constraint insisting that not all three variables must take the same value (as shown in Figure 1.4(a)). This problem can be transformed into the binary constraint problem shown in Figure 1.4(b). In the transformed problem the variable W is created. The domain of W is the set of compound labels in $C_{x,y,z}$:

(<*x*,1>,<*y*,1>,<*z*,2>)
(<*x*,1>,<*y*,2>,<*z*,2>)
(<*x*,1>,<*y*,2>,<*z*,1>)
(<*x*,2>,<*y*,1>,<*z*,2>)
(<*x*,2>,<*y*,1>,<*z*,1>)
(<*x*,2>,<*y*,2>,<*z*,1>)

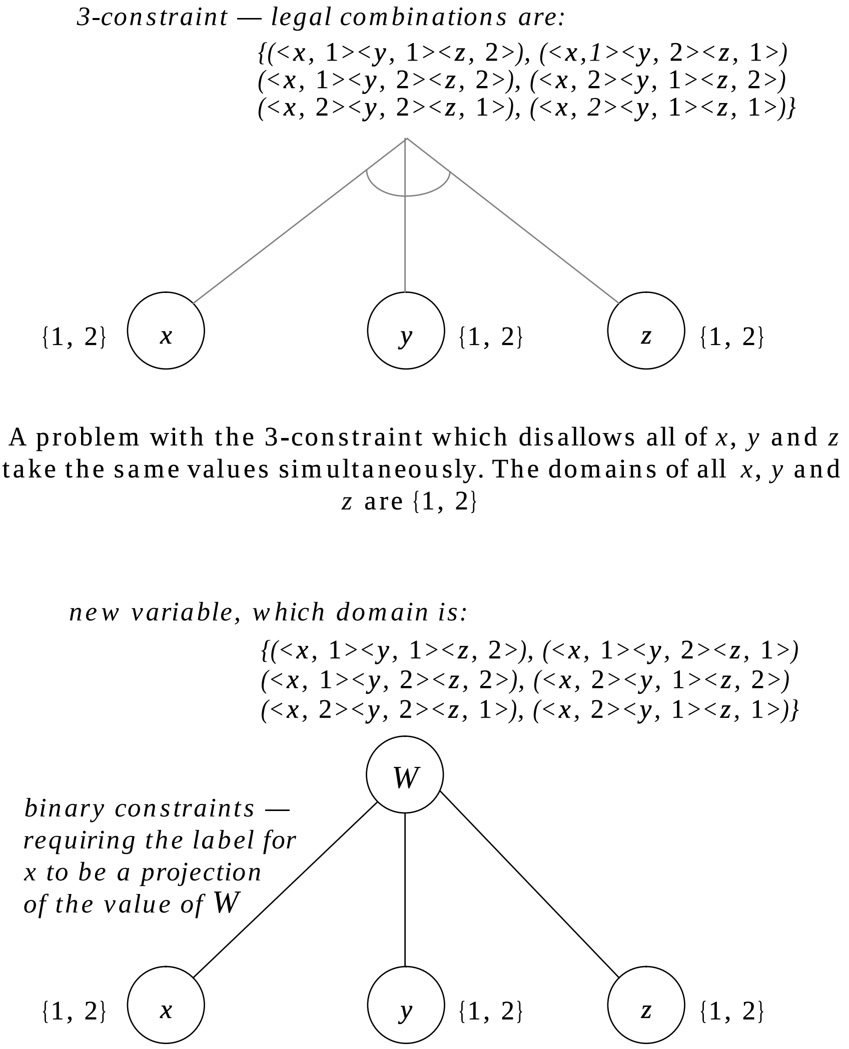
The constraint between W and x, say, is that *projection*(v_W , (<x,v_x>)) must hold, where v_W and v_x are the values that W and x take, respectively. For example, according to this constraint, if W takes the value (<*x*,1>,<*y*,1>,<*z*,2>), then x must take the value 1.

By removing k-ary constraints for all $k > 2$, we introduce new variables which have large domains. Whether one could benefit from this transformation depends on what we can do with the resulting problem. A number of CSP solving techniques which we shall illustrate in this book are applicable only to binary CSPs.

In Chapter 7, we shall explain that every CSP is associated to a *dual CSP*, which is also a binary CSP.

1.4 Graph-related Concepts

Since graph theory plays an important part in CSP research, we shall define some


3-constraint — legal combinations are:
{(<x, 1><y, 1><z, 2>), (<x,1><y, 2><z, 1>)
(<x, 1><y, 2><z, 2>), (<x, 2><y, 1><z, 2>)
(<x, 2><y, 2><z, 1>), (<x, 2><y, 1><z, 1>)}
{1, 2}
x
y
{1, 2}
z
{1, 2}
new variable, which domain is:
{(<x, 1><y, 1><z, 2>), (<x, 1><y, 2><z, 1>)
(<x, 1><y, 2><z, 2>), (<x, 2><y, 1><z, 2>)
(<x, 2><y, 2><z, 1>), (<x, 2><y, 1><z, 1>)}
W
binary constraints —
requiring the label for
x to be a projection
of the value of W
{1, 2}
x
y
{1, 2}
z
{1, 2}

(a) A problem with the 3-constraint which disallows all of *x*, *y* and *z* to take the same values simultaneously. The domains of all *x*, *y* and *z* are {1, 2}

(b) A binary constraint problem which is transformed from (a). A new variable *W* is created, in which the domain is the set of all compound labels for *x*, *y* and *z*. The constraints between *W* and the other three variables require that **labels for** *x*, *y* and *z* must be projections of *W* 's **value**

Figure 1.4 Transformation of a 3-constraint problem into a binary constraint

terminologies in graph theory which we shall refer to later in this book.

Definition 1-15:

A **graph** is a tuple (V, U) where V is a set of nodes and U ($\subseteq V \times V$) is a set of **arcs**. A node can be an object of any type and an arc is a pair of nodes. For convenience, we use *graph(G)* to denote that G is a graph.

An **undirected graph** is a tuple (V, E) where V is a set of nodes and E is a set of **edges**, each of which being a collection of exactly two elements in V. ■

The nodes in an arc are ordered whereas the nodes in an edge are not. An edge can be seen as a pair of arcs (x,y) and (y,x). A binary CSP is often visualized as a constraint graph, which is an undirected graph where the nodes represent variables and each edge represents a binary constraint.

Definition 1-16:

For all graphs (V, E), node x is **adjacent** to node y if and only if (x, y) is in E:

$\forall$ graph((V, E)): ($\forall x, y \in V$: adjacent(x, y, (V, E)) $\equiv$ (x, y) $\in E$) ■

Definition 1-17:

A **hypergraph** is a tuple (V, E) where V is a set of nodes and E is a set of **hyperedges**, each of which is a set of nodes. For convenience we use *hypergraph((V, E))* to denote that (V, E) is a hypergraph, *hyperedges(F , V)* to denote that F is a set of hyperedges for the nodes V (i.e. F is a set of set of nodes in V), and *nodes_of(e)* to denote the nodes involved in the hyperedge e. ■

Hypergraphs are a generalization of graphs. In a hypergraph, each hyperedge may connect more than two nodes. In general, every CSP is associated with a constraint hypergraph.

Definition 1-18:

The **constraint hypergraph of a CSP** (Z, D, C) is a hypergraph in which each node represents a variable in Z, and each hyperedge represents a constraint in C. We denote the constraint hypergraph of a CSP P by *H(P)*. If P is a binary CSP and we exclude hyperedges on single nodes, then *H(P)* is a graph. We denote the **constraint graph of a CSP** by *G(P)*:

$\forall$ csp((Z, D, C)):

$$\begin{aligned}&(V\ , E\ ) = \mathrm{H}((Z, D, C)) \ \equiv \\ &\quad ((V\ \ = Z) \wedge (E = \{S \mid \exists\, c \in\ C \wedge S = \text{variables\_of}(c)\})) \ \blacksquare\end{aligned}$$

What a constraint hypergraph does not show are those domains and the compound labels which are allowed or disallowed by the constraints in the CSP.

Later we shall extend our definition of a *constraint graph of a CSP* to general CSPs (see Definition 4-1 in Chapter 4).

Definition 1-19:

A **path** in a graph is a sequence of nodes drawn from it, where every pair of adjacent nodes in this sequence is connected by an edge (or an arc, depending on whether the graph is directed or undirected) in the graph:

$$\begin{aligned}&\forall\ \text{graph}((V, E)): \\ &\quad \forall\ x_1, x_2, ..., x_k \in\ V: \\ &\qquad (\text{path}((x_1, x_2, ..., x_k), (V, E)) \ \equiv ((x_1, x_2) \in\ E) \wedge ... \wedge ((x_{k-1}, x_k) \in\ E))) \\ &\qquad \blacksquare\end{aligned}$$

Definition 1-20:

A **path of length *n*** is a path which goes through n + 1 (not necessarily distinct) nodes:

$$\text{length\_of\_path}(\ (x_1, x_2, ..., x_k)\ ) \ \equiv k - 1 \ \blacksquare$$

Definition 1-21:

A node y is **accessible** from another node x if there exists a path from x to y:

$$\begin{aligned}&\forall\ \text{graph}((V, E)): (\forall x, y \in\ V: \\ &\quad \text{accessible}(x, y, (V, E)) \ \equiv \\ &\qquad ((x,y) \in\ E \vee (\exists\ z_1, z_2, ..., z_k: \text{path}((x, z_1, z_2, ..., z_k, y), (V, E)))) \ \blacksquare\end{aligned}$$

Definition 1-22:

A graph is **connected** if there exists a path between every pair of nodes:

$$\begin{aligned}&\forall\ \text{graph}((V, E)): \\ &\quad (\text{connected}(\ (V, E)\ ) \ \equiv\ (\forall\ \ x, y \in\ V: \text{accessible}(x, y, (V, E)))) \ \blacksquare\end{aligned}$$

A constraint graph need not be connected (some variables may not be constrained, and sometimes variables may be partitioned into mutually unconstrained groups).

Definition 1-23:

A **loop** in a graph is an edge or an arc which goes from a node to itself, i.e. a loop is (x,x), where x is a node. ■

Definition 1-24:

A **network** is a graph which is connected and without loops:

$\forall$ graph((V, E)):
(network((V, E)) $\equiv$ (connected((V, E)) $\wedge$ ($\forall x \in V$: $(x, x) \notin E$)))) ■

Definition 1-25:

A **cycle** is a path on which end-points coincide:

$\forall$ graph((V, E)): ($\forall x_0, x_1, x_2, ..., x_k \in V$:
(cycle(($x_0, x_1, x_2, ..., x_k$), (V, E)) $\equiv$
(path(($x_0, x_1, x_2, ..., x_k$), (V, E)) $\wedge$ $x_0 = x_k$))) ■

Definition 1-26:

An **acyclic graph** is a graph which has no cycles:

$\forall$ graph(G): (acyclic(G) $\equiv$ ($\neg$ $\exists$ path(p, G): cycle(p, G))) ■

Definition 1-27:

A **tree** is a connected acyclic graph:

$\forall$ graph(G): (tree(G) $\equiv$ (network(G) $\wedge$ ($\neg$ $\exists$ path(p, G): cycle(p, G)))) ■

Definition 1-28:

A binary relation (<) on a set S is called an **ordering** of S when it is *irreflexive, asymmetric* and *transitive*:

irreflexive(S, <): $\forall x \in S$: $\neg\, x < x$
asymmetric(S, <): $\forall x, y \in S$: ($x <$ y $\Rightarrow \neg\, y < x$)
transitive(S, <): $\forall x, y, z \in S$: ($x < y \wedge y < z \Rightarrow x < z$). ■

Definition 1-29:

A set S is **totally ordered** if every two elements in S are ordered. Such an ordering is called a **total ordering** of the elements in S:

total_ordering(S, <) $\equiv$ ($\forall\, x, y \in S$: $x < y \vee y < x$). ■

1.5 Examples and Applications of CSPs

To help understand what a CSP is and where they appear, we shall look at some examples and applications of CSPs in this section.

1.5.1 The *N*-queens problem

In this section, we shall formulate the *N*-queens problem that we introduced in Section 1.1.1 according to the formal definition of CSP, and illustrate that a problem can be formulated as a CSP in different ways.

1.5.1.1 Problem formalization

To formalize a problem as a CSP, we must identify a set of variables, a set of domains and a set of constraints. One way to formalize the 8-queens problem as a CSP is to make each of the eight rows in the 8-queens problem a variable: the set of variables $Z = \{Q_1, Q_2, ..., Q_8\}$. Each of these eight variables can take one of the eight columns as its value. If we label the columns with values 1 to 8 (for computation purposes, which will be made clear below), then the domains of all the variables in this CSP are as follows:

$$D_{Q_1} = D_{Q_2} = ... = D_{Q_8} = \{1, 2, 3, 4, 5, 6, 7, 8\}.$$

Now let us look at the set of constraints. The fact that we represent each row as a variable has ensured that no two queens can be on the same row. To make sure that no two queens are on the same column, we have the following constraint:

Constraint (1): $\forall\, i, j$: $Q_i \neq Q_j$

To make sure that no two queens are on the same diagonal, we can include the following constraint in our set of constraints:

Constraint (2): $\forall\, i, j$, if Q_i = a and Q_j = b, then $i - j \neq$ a – b, and $i - j \neq$ b –a.

(Making the values integers allows us to do arithmetic with them.) To represent these constraints, we could explicitly record the set of all compatible values between each pair of variables. Alternatively, we can make them functions or procedures — given a pair of labels, these functions or procedures return true or false, depending on whether the given labels are compatible or not. Program 1.1 is a simple example of such a piece of code.

```
/*compatible1( X/Vx, Y/Vy )
X/Vx represents the X-th row, Vx-th column; and Y/Vy repre-
    sents the Y-th row, Vy-th column (where Vx and Vy both
    range from 1 to 8), compatible1/2 succeeds if and only if X/
    Vx and Y/Vy are compatible according to the constraints in
    the N-queens problem.
*/
    compatible1( X/Vx, Y/Vy ) :-
    Vx =\= Vy,                              /* Constraint (1) */
    X - Y =\= Vx - Vy,                      /* Constraint (2) */
    X - Y =\= Vy - Vx.                      /* Constraint (2) */
```

Program 1.1: Functional representation of a constraint in the *N*-queens problem

Under this problem formalization, there are 8^8 combinations of values for the eight variables to be considered. In general, an *N*-queens problem has N^N candidate solutions to be considered.

1.5.1.2 Alternative formalization of the *N*-queens problem

It is worth pointing out here that there is often more than one way to formalize a problem as a CSP. The *N*-queens problem need not be formalized in the above way. An alternative representation is to use $Q_1, Q_2, ..., Q_8$ to represent the positions of the queen (rather than the column of each queen in the above formalization). If the 64 squares in the 8×8 board are numbered 1 to 64, then the domain of each variable becomes {1, 2, ..., 64}. In other words:

$$Z = \{Q_1, Q_2, ..., Q_8\}$$
$$D_{Q_1} = D_{Q_2} = ... = D_{Q_8} = \{1, 2, 3, 4, ..., 64\}.$$

Let us assume that we number the squares from left to right, top to bottom. Then given a number which represents a square, the row and column of that square can be computed as follows:

row = (number div 8) + 1
column = (number mod 8) + 1

Given the rows and columns of two squares, we can check whether they are compatible with each other using the codes shown in Program 1.2.

```
/*compatible2( N1, N2 )
given N1 and N2, which both range from 1 to 64, this predicate
    succeeds if and only if N1 and N2 are compatible according
    to the constraints in the 8-queens problem.
*/
compatible2( N1, N2 ) :-
    R1 is (N1 div 8) + 1, C1 is (N1 mod 8) + 1,
    R2 is (N2 div 8) + 1, C2 is (N2 mod 8) + 1,
    R1  =\= R2,
    C1  =\= C2,
    R1 - R2  =\= C1 - C2,
    R1 - R2  =\= C2 - C1.
```

Program 1.2: Alternative functional representation of a constraint in the 8-queens problem

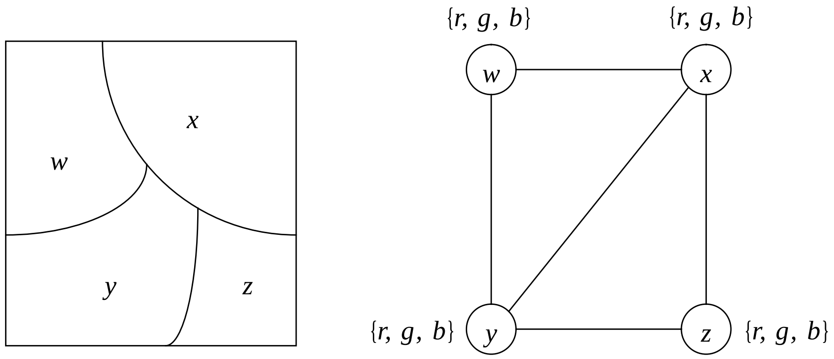
Some formalizations of a problem are easier to solve than others. The 8-queens problem formalized in this section allows 64^8 combinations of 8-compound labels, which makes the problem potentially more difficult to solve than the CSP formalized in the preceding section (which allows only 8^8 combinations of 8-compound labels). The formalization in the preceding section in fact has built into it the constraint that no two queens can be placed in the same row.

1.5.1.3 Caution about benchmarking using the *N*-queens problem

The *N*-queens problem will be used to illustrate a number of CSP solving algorithms in this book, but it is worth pointing out that benchmarks on different algorithms produced using this problem must be interpreted with caution. This is because the *N*-queens problem has very specific features: firstly, it is a binary constraints problem; secondly, every variable is constrained by every other variable, which need not be the case in other problems. More importantly, in the *N*-queens problem, each label for every variable conflicts with at most three values of each other variable, regardless of the number of variables (i.e. N) in the problem. For example, <1,2> has conflict with <2,1>, <2,2> and <2,3>. In an 8-queens problem, for example, when 2 is assigned to Queen 1, there are 5 out of 8 values that Queen 2 can take. But in the 1,000,000-queens problem, there are 999,997 out of 1,000,000 values that Queen 2 can take after <1,2> has been committed to. Therefore, constraints get looser as N grows larger (see formal definition of tightness in Definition 2-13). Such features may not be shared by many other CSPs.

1.5.2 The graph colouring problem

Another problem which is often used to explain concepts and algorithms for the CSP is the *colouring problem*. Given a graph and a number of colours, the problem is to assign colours to those nodes satisfying the constraint that no adjacent nodes should have the same colour assigned to them. One instance of the colouring problem is the *map colouring problem*: the problem is to colour the different areas of a given map with a limited number of colours, subject to the constraint that no adjacent areas in the map have the same colour. Figure 1.5(a) shows an example of a map which is to be coloured. The map colouring problem is an instance of the general *graph colouring problem*, as a map can be represented by a graph where each node represents an area in the map, and every pair of nodes which represent two adjacent areas in the map is connected by an edge (see Figure 1.5(b)).



(a) map to be coloured

(b) constraint graph of the CSP in (a): w, x, y and z are variables which all have the same domain $\{r, g, b\}$

Figure 1.5 Example of a map colouring problem

The areas to be coloured in Figure 1.5(a) are w, x, y and z. Assume that we are allowed to label the map with three colours only: r (for red), g (for green) and b (for blue). The values in {} next to the nodes (i.e. variables) in Figure 1.5(b) specify the domain. Each of the edges in the graph in Figure 1.5(b) represents a constraint which states that the connected nodes must not take the same value. The constraint on variables A and B (denoted $C_{A,B}$) is conceptually seen as the set {(<*A*,*r*><*B*,*g*>), (<*A*,*r*><*B*,*b*>), (<*A*,*g*><*B*,*r*>), (<*A*,*g*><*B*,*b*>), (<*A*,*b*><*B*,*r*>), (<*A*,*b*><*B*,*g*>)}. (In practice, it can be represented by other means, e.g. a function). One solution tuple for this problem is: (<*A*,*r*> <*B*,*g*> <*C*,*b*> <*D*,*r*>). To summarize, the CSP (Z, D, C) for this problem is:

$$Z = \{w, x, y, z\}$$
$$D_w = D_x = D_y = D_z = \{r, g, b\}$$
$$C = \{C_{w,x}, C_{w,y}, C_{x,y}, C_{x,z}, C_{y,z}\}$$

1.5.3 The scene labelling problem

The *scene labelling problem* in computer vision is probably the first CSP to be formalized. In vision, the scenes are normally captured as images by cameras. After some preprocessing, lines can be recognized from the images, then scenes like the one shown in Figure 1.6 are generated.

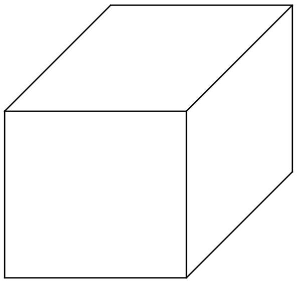


Figure 1.6 Example of a scene to be labelled

To recognize the objects in the scene, one must first interpret the lines in the drawings. One can categorize the lines in a scene into the following types:

(1) **convex edges**
A *convex edge* is an edge formed by two planes, both of which extend (from the edge) away from the viewer. Convex edges are marked by "+".

(2) **concave edges**
A *concave edge* is an edge formed by two planes both of which extend (from the edge) towards the viewer. Concave edges are marked by "–".

(3) **occluding edges**
An *occluding edge* is a convex edge where one of the planes is hidden behind the other and therefore not seen by the viewer. Occluding edges are marked by either "→" or "←", depending on the situation. If one moves along an occluding edge following the direction of the arrow, the area on the right represents the face of an object which can be seen by the viewer, and the area on the left represents the background or some faces of the objects at the back.

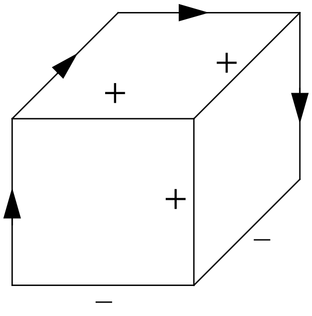


Figure 1.7 The scene in Figure 1.6 with labelled edges

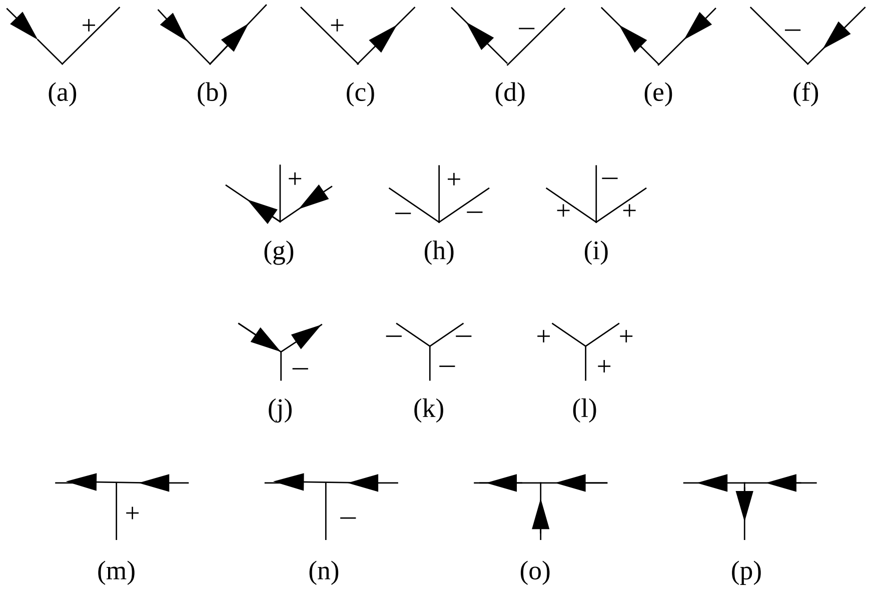


Figure 1.8 Legal labels for junctions (from Huffman, 1971)

The scene in Figure 1.6 can be labelled as shown in Figure 1.7. Given any junction independent of the scene, there are limited choices of labels. These choices are shown in Figure 1.8.

One way in which to formalize the scene labelling problem as a CSP is to use one variable to represent the value of a line in the scene. For example, in the scene in Figure 1.6 we have the following variables: {*A, B, C, D, E, F, G, H, I*}, as shown in Figure 1.9. The domain of each variable is therefore the set {+, –, →, ←}.

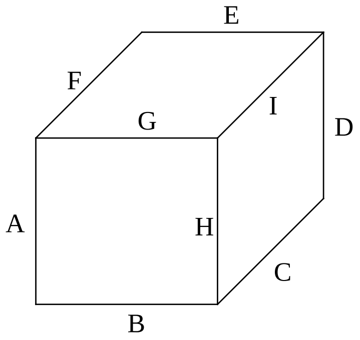
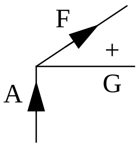
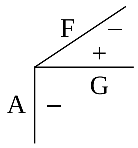


Figure 1.9 Variables in the scene labelling problem in Figure 1.6

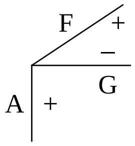
The limited choices of label combinations in the junctions (as shown in Figure 1.8) impose constraints on the variables. Since lines *A*, *F*, *G* form an arrow, according to (g)-(i) in Figure 1.8, the values that these three variables can take simultaneously are restricted to:



(<A,→> <F,→> <G,+>)



(<A,–> <F,–> <G,+>)



(<A,+> <F,+> <G,–>)

Similarly, every other junction posts constraints to the labelling of the lines which form it. The task is to label all the variables, satisfying all the constraints. One may want to find one or all solutions to this problem, depending on the need of subsequent processing. Waltz introduced an algorithm, referred to as the *Waltz filtering algorithm*, for solving this problem. The algorithm is based on constraint propagation, and is discussed in Chapter 4.

1.5.4 Temporal reasoning

Temporal reasoning, which involves constraint satisfaction, is an important area in *AI planning* and many other applications (e.g. see Tsang, 1987b; Dechter *et al*. 1991). Events are all temporally related to each other. Depending on the time structure that one uses, different sets of temporal relations apply. In early research in planning, the world is simplified in such a way that all events are assumed to be instantaneous. In that case, three relations are possible between any two events *A* and *B*: “*A before B*”, “*B before A*” or “*A equals B*”. Allen [1983] points out that when durations in events are reasoned about, 13 relations are possible between any two events. These relations are shown in Figure 1.10.

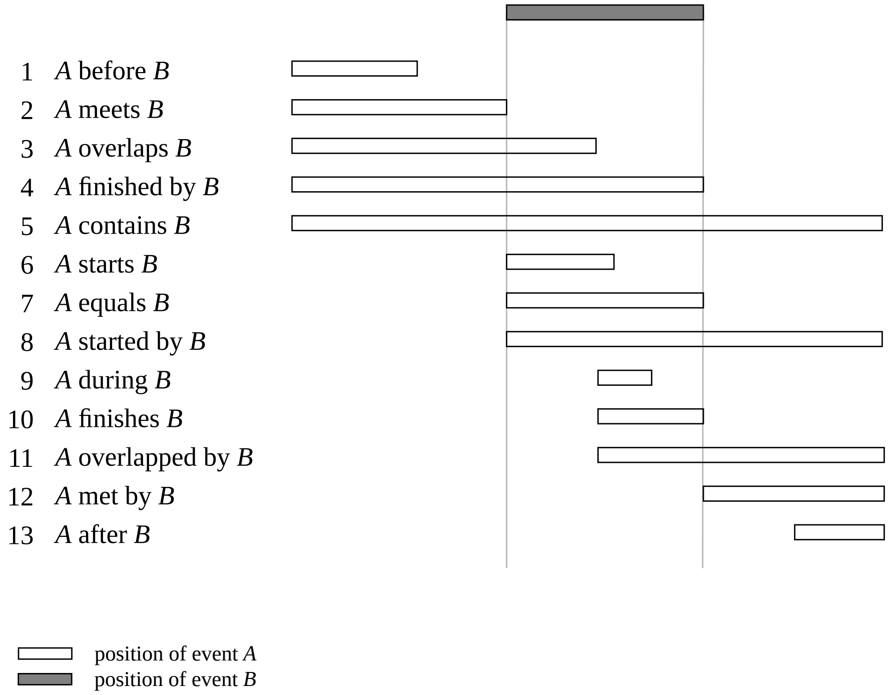


Figure 1.10 Thirteen possible temporal relations between two events

In planning and scheduling, one has to determine the temporal relationship between events. There are basically two approaches. One is to assume one temporal relation per pair of events at a time, and backtrack when the hypothetical situation has been proved to be over-constrained. Most conventional planners do this (for example, see Fikes & Nilsson, 1971; Sacerdoti, 1974; Tate, 1977; and Wilkins, 1988). These planners adopt the assumption that events are instantaneous, therefore their ability to represent real life temporal knowledge is limited.

The other approach is to reason with all disjunctive temporal relations simultaneously. This approach is taken by Allen & Koomen [1983]. In order to schedule the events, one has to assign one temporal relation between each pair of events [Tsan86,87b]. The CSP in temporal reasoning under this approach is one where each variable represents the temporal relationship between a pair or events. (Among n events, there are $n \times (n - 1) \div 2$ temporal relations, i.e. variables.) Each variable may take one of the 13 primitive relations in Figure 1.10 as its value. The property of time imposes constraints on the values that we can assign to each variable. For example, if *A* is before *B*, and *B* is before *C*, then *A* must be before *C*. If *A* overlaps *B*, and *B* overlaps *C*, then *A* must overlap, meet or be before *C*. The task is then to find a consistent set of primitive relations between the intervals — a set which satisfies all the constraints.

1.5.5 Resource allocation in AI planning and scheduling

Resource allocation and scheduling are better known applications of CSP. The car sequencing problem described in Section 1.1.2 is an example of a scheduling problem to which CSP solving techniques have been applied successfully. A typical scheduling problem is a problem in which one is given a set of jobs and asked to allocate resources to them. Each job may require a number of resources, which include time (during which these jobs are finished), machines, tools, manpower, etc.

Resource allocation, especially when time and shared resources are involved, is basically a CSP. Each variable in the CSP represents one shared resource requirement. For example, variable *X* may represent the machine requirement of a job. The domain of a variable is the set of possible values that this variable can take. The domain of *X* in the above example may be the set of machines available in the workshop which have the capacity to do the job, e.g. {machine-203, machine-208, machine-209}. Assigning a value to a variable represents the allocation of a resource to a job. The allocation of resources is normally constrained in many ways. For example, among the *M* machines available to a job *J*, only machines *P*, *Q* and *R* have the capacity to cope with job *J*. Very often, one machine can only process one job at a time. Sometimes, if job *J* is to use machine M_1, then it must also be given certain tools and certain engineers. The task is to allocate to each variable a value such that all the constraints are satisfied.

1.5.6 Graph matching

In semantic networks, one may want to check whether a particular concept is present. This problem can be seen as a graph matching problem, as defined below. Given two graphs G_1 and G_2, the problem is to check whether G_2 has a subgraph which matches G_1. Graph (V_1, E_1) contains graph (V_2, E_2) if:

(1) every node in V_2 can be mapped to a distinct node in V_1; and
(2) for all x_1, y_1 in V_1 and x_2, y_2 in V_2, if x_2 and y_2 are mapped to x_1 and y_1, respectively, then whenever (x_2,y_2) is an edge in E_2, then (x_1,y_1) is an edge in E_1.

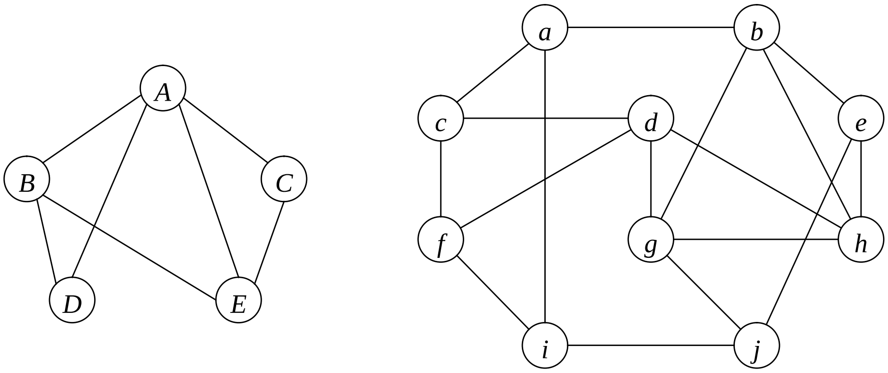
Figure 1.11 shows an example of a graph matching problem. Given the graphs G_1 and G_2 (shown in Figures 1.11(a) and (b), respectively), the task is to find out whether G_2 contains a subgraph of G_1. This can be formalized as a CSP where the variables are *A*, *B*, *C*, *D* and *E*, and the domains for all of them are {a,b,c,d,e,f,g,h,i,j}. The constraint is that for all compound labels (<*x*,p><*y*,q>), if *x* and *y* are connected in G_1, then p and q must be connected in G_2. For example, (<*A*,h><*B*,g>) satisfies the constraint on *A* and *B* because (g,h) is an edge in G_2. A little reflection should convince the readers that the compound label (<*A*,h><*B*,g><*C*,e><*D*,d><*E*,b>) is a solution tuple to this problem.

1.5.7 Other applications

In natural language parsing, each word has a finite number of roles that it can play. The language restricts the domain of roles (e.g. “noun”, “verb”, “adverb”, etc.) that each word can play. The grammar of the language restricts the roles that a string of words can take simultaneously. Part of the parsing task is to identify the roles of each word. Rich & Knight [1991] advocate that this task is a CSP.

Database queries often have variables in them. Instantiating the variables in a database query is a CSP. Query optimization is an important database research area in which CSP solving techniques can be applied. On the other hand, techniques developed in query optimization research can be used in CSP solving. The tree-clustering method described in Chapter 8 is one example of such cross-fertilization of the two research disciplines.

CSP techniques have also been applied to parameter setting for greenhouses in agricultural applications, and demonstrated to be successful [CroMar91]. Problem reduction techniques in CSP (see Chapters 2 to 4) have been demonstrated as being effective for cutting down search spaces for spatial reasoning [duVTsa91].



(a) graph G_1

(b) graph G_2

Figure 1.11 Example of a graph matching problem. Does graph G_2 contain a subgraph which matches graph G_1? (The answer is yes, if A→h, B→g, C→e, D→d, E→b)

1.6 Constraint Programming

The generality of the CSP has lead to the development of *constraint programming languages*. These languages provide built-in functions (or predicates) for describing commonly encountered constraints, and help users to solve problems by applying techniques which have been developed in CSP research.

Many approaches for constraint programming are based on and extended from the logic programming paradigm. Some of the better known constraint logic programming languages and systems are CLP, PROLOG III and CHIP. In these languages, unification in conventional logic programming is replaced by constraint satisfaction. Numerical constraints are being focused on. The idea is to hide the constraint solving techniques from the user.

One of the main objectives of developing CLP is to define a class of logic programming languages with well defined semantics under a particular equational theory (see Jaffar *et al.* [JaLaMa86]). An instance of CLP called CLP(*R*) (*R* here stands for real numbers) has been implemented and proposed for applications such as electrical engineering by Heintze *et al.* [HeMiSt87] and option trading by Lassez *et al.* [LaMcYa87]. In CLP(*R*), constraints are handled incrementally using linear programming methods such as the simplex method.

Prolog III was developed with similar goals as CLP, but with refined manipulation on trees (including infinite trees), lists and boolean variables. It has been developed into a commercial product that has been demonstrated to be efficient and elegant in problem solving. Like CLP, Prolog III basically uses the simplex method for handling equations and inequalities with numerical variables.

CHIP is a logic programming language for handling symbolic, boolean as well as numerical variables. Search techniques, discussed later, are used to instantiate symbolic variables. The basic search strategy used in CHIP is called *forward checking* (FC). It is used together with a heuristic called the *fail first principle* (FFP). FC and FFP are discussed in Chapters 5 and 6 respectively. The combination of this search strategy and heuristic has been found to be very effective. CHIP has been applied to a number of problems, and success has been claimed. Some of the reported applications of CHIP include the car-sequencing problem [DiSiVa88b], the spares allocation problem [DVSAG88a], job-shop scheduling, warehouse location, circuit verification (to verify that an implementation of a circuit meets its specifications) [DVSAG88a] and the cut stock problem [DiSiVa88a].

The success of CHIP has led to the development of two other commercially available languages, Charme and PECOS. The basic CSP solving techniques used in them are no different from CHIP, and therefore the comparison among CHIP, Charme and PECOS is down to the differences in their language types and their implementation efficiency.

Charme uses the syntax of C, and one of its merits is that it can easily be integrated into the users' other C programs. Arrays (which have to be implemented by lists in CHIP) are introduced in Charme. It has been applied to similar problems as CHIP [Charme90]. PECOS uses LISP syntax. Both Charme and PECOS are mainly built to handle symbolic but not numerical and boolean variables (boolean variables may be represented by symbolic variables with specific constraints such as logical AND and logical OR).

1.7 Structure Of Subsequent Chapters

We emphasized earlier that the CSP is an important problem not only because of its generality, but also because it has specific features which allows one to develop specialized techniques to tackle it. The main features of CSPs will be studied in Chapter 2. There we also propose a classification of CSP solving techniques, and give an overview of them. The three classes of CSP solving techniques are: (1) problem reduction; (2) searching; and (3) solution synthesis.

In Chapter 3, some of the most important concepts related to CSP solving will be introduced. These concepts are useful for describing the techniques in the chapters that follow.

Chapter 4 covers problem reduction algorithms. These algorithms transform problems into equivalent problems which are hopefully easier to solve.

Chapters 5 to 8 are about searching techniques for CSPs. Chapter 5 describes basic control strategies of searching which are relevant to CSP solving. Chapter 6 discusses the significance of ordering the variables, values and compatibility checking in searching. Chapter 7 discusses specialized search techniques which gain their efficiency by exploiting problem specific features. Chapter 8 discusses stochastic search approaches (including hill climbing and connectionist approaches) for CSP solving.

Chapter 9 discusses how solutions can be synthesized rather than searched for.

The definition of CSP in Definition 1.12 is extended in Chapter 10 to include the notion of optimality. In many real life problems, certain solutions are preferred to others. Besides, in problems which do not contain any solutions, one may want a problem solver to find near solutions rather than simply reporting failure. These problems will be formally defined in Chapter 10, and relevant research will be summarized.

Pseudo code is used to explain most of the algorithms introduced in this book. Implementations, which are presented to help in clarifying the algorithms to an executable level, are included for those algorithms which are suitable to be implemented in Prolog. These implementations are grouped together and placed at the end of this book for easy reference.

1.8 Bibliographical Remarks

The CSP was first formalized in line labelling in vision research. The problem is tackled in Huffman [1971], Clowes [1971] and Waltz [1975]. Mackworth [1992] defines CSPs with finite domains as *finite constraint satisfaction problems*, and gives an overview to such problems. Haralick & Shapiro [1979, 1980] discuss different aspects of the CSP — from problem formalization, applications to algorithms. Meseguer [1989] and Kumar [1992] both give concise and comprehensive overviews to CSP solving. Apart from studying the basic CSP and its general characteristics, Guesgen & Hertzberg [1992] introduce the concept of *dynamic constraints*, which are constraints that are themselves subject to constraints. The usefulness of this idea is demonstrated in spatial reasoning.

Mittal & Falkenhainer [1990] extend the standard CSP to *dynamic CSPs* (CSPs in which constraints can be added and relaxed), and proposed the use of *assumption-*

based TMS (ATMS) to solve them (see de Kleer, 1986a,b,c, 1989). Definitions on graphs and networks are mainly due to Carré [1979].

The *N*-queens problem has been used to illustrate much CSP research, e.g. see Mackworth [1977] and Haralick & Elliott [1980]. Abramson & Yung [1989] and Bernhardsson [1991] independently present solutions to the *N*-queens problem which exploit the properties of the problem (and require no searching at all). The map colouring problem is a simplified version of the graph colouring problem, which is discussed extensively by Nelson & Wilson [1990]. Tsang [1987b, 1988] points out the CSP in temporal reasoning under Allen's [1983] interval-based formalism. Dechter *et al.* [1991] look at the CSPs under point-based temporal reasoning. Kautz & Selman [1992] and Yang [1992] see constraint satisfaction as a crucial part of AI planning. Tsang [1988c], Tsang & Wilby [1988b], Zweben & Eskey [1989], Minton & Philips [1990] and Prosser [1990] all propose to formalize scheduling problems as CSPs, and demonstrate favourable consequences of doing so. Other examples of CSPs are abundant. Rich & Knight [1991] and Haddock [1991] both cast part of the natural language parsing problem as CSPs. Haddock [1992] sees semantic evaluation as constraint satisfaction. Dechter & Pearl [1988b] point out the relationship between query optimization in database research and CSP solving. Cros & Martin-Clouair [1991] apply CSP techniques to greenhouse management and du Verdier & Tsang [1991] apply CSP techniques to spatial reasoning.

For constraint logic programming (CLP), see van Hentenryck *et al.* [1989a, 1992] and Cohen [1990] for general overviews. Jaffar *et al.* [1987], Heintze *et al.* [1987] and Lassez *et al.* [1987] summarize CLP, and Colmerauer [1990] summarizes Prolog III. Applications of CHIP are reported in Simonis & Dincbas [1987], Dincbas *et al.* [DiSiVa88a,b] [DVSAG88a], van Hentenryck [1989b] and Perrett [1991]. [Charme90] gives an overview of Charme. A general purpose constraint language called *Bernard* is described by Leler [1988].